

eXamen.press

eXamen.press ist eine Reihe, die Theorie und Praxis aus allen Bereichen der Informatik für die Hochschulausbildung vermittelt.

Thomas Rauber · Gudula Rünger

Parallele Programmierung

2., neu bearbeitete und erweiterte Auflage
Mit 139 Abbildungen

 Springer

Thomas Rauber
Universität Bayreuth
Fakultät für Mathematik und Physik
95440 Bayreuth
rauber@uni-bayreuth.de

Gudula Rünger
Technische Universität Chemnitz
Fakultät für Informatik
09107 Chemnitz
ruenger@informatik.tu-chemnitz.de

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen
Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über
<http://dnb.d-nb.de> abrufbar.

Die erste Auflage erschien als Springer-Lehrbuch im Springer-Verlag Berlin Heidelberg unter
dem Titel *Parallele und Verteilte Programmierung*

ISSN 1614-5216
ISBN 978-3-540-46549-2 Springer Berlin Heidelberg New York
ISBN 978-3-540-66009-5 1. Auflage Springer Berlin Heidelberg New York

Dieses Werk ist urheberrechtlich geschützt. Die dadurch begründeten Rechte, insbesondere
die der Übersetzung, des Nachdrucks, des Vortrags, der Entnahme von Abbildungen und
Tabellen, der Funksendung, der Mikroverfilmung oder der Vervielfältigung auf anderen We-
gen und der Speicherung in Datenverarbeitungsanlagen, bleiben, auch bei nur auszugsweiser
Verwertung, vorbehalten. Eine Vervielfältigung dieses Werkes oder von Teilen dieses Werkes
ist auch im Einzelfall nur in den Grenzen der gesetzlichen Bestimmungen des Urheberrechts-
gesetzes der Bundesrepublik Deutschland vom 9. September 1965 in der jeweils geltenden
Fassung zulässig. Sie ist grundsätzlich vergütungspflichtig. Zuwiderhandlungen unterliegen
den Strafbestimmungen des Urheberrechtsgesetzes.

Springer ist ein Unternehmen von Springer Science+Business Media
springer.de

© Springer-Verlag Berlin Heidelberg 2000, 2007

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem
Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche
Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten
wären und daher von jedermann benutzt werden dürften. Text und Abbildungen wurden
mit größter Sorgfalt erarbeitet. Verlag und Autor können jedoch für eventuell verbliebene
fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine
Haftung übernehmen.

Satz: Druckfertige Daten der Autoren
Herstellung: LE-TeX, Jelonek, Schmidt & Vöckler GbR, Leipzig
Umschlaggestaltung: KünkelLopka Werbeagentur, Heidelberg
Gedruckt auf säurefreiem Papier 33/3100 YL – 5 4 3 2 1 0

Vorwort

Das Anliegen dieses Buches ist es, dem Leser detaillierte Kenntnisse der Parallelverarbeitung zu vermitteln und ihn insbesondere mit dem heutigen Stand der Techniken der parallelen Programmierung vertraut zu machen. Das vorliegende Buch ist die zweite Auflage des im Jahre 2000 erschienenen Buches *Parallele und Verteilte Programmierung*. Seit dem Erscheinen der ersten Auflage hat die technologische Entwicklung u.a. durch die weite Verbreitung von Clustersystemen und die Einführung von Multicore-Prozessoren dazu geführt, dass die Techniken der parallelen Programmierung enorm an Wichtigkeit zugenommen haben. Dies gilt nicht nur für die bisherigen Hauptanwendungsgebiete im Bereich wissenschaftlich-technischer Berechnungen. Die parallele Programmierung spielt auch für die effiziente Nutzung typischer Desktop-Rechner eine große Rolle, so dass sich parallele Programmiertechniken in alle Bereiche der Softwareentwicklung ausbreiten. In Zukunft werden Standard-Softwareprodukte auf Konzepten der parallelen Programmierung basieren müssen, um die parallelen Hardwareressourcen ausnutzen zu können. Dadurch ergibt sich ein enormer Bedarf an Softwareentwicklern mit parallelen Programmierkenntnissen. Dazu passt die Forderung der Prozessorhersteller, die parallele Programmierung als obligatorische Komponente in die Curricula der Informatik aufzunehmen. Die vorliegende zweite Auflage trägt dieser Entwicklung dadurch Rechnung, dass die für die Programmierung von Multicore-Prozessoren erforderlichen Programmiertechniken einen breiten Raum einnehmen. Die entsprechenden Kapitel wurden um wichtige Aspekte erweitert, neue Beispiele wurden aufgenommen und zusätzliche Programmieransätze werden vorgestellt. Auch die restlichen Kapitel wurden überarbeitet und z.T. erweitert. Dies trifft insbesondere auch auf das Kapitel über die Architektur paralleler Plattformen zu, das einen Überblick über aktuelle Plattformen und Hardwaretechnologien gibt.

Das Buch kann thematisch in drei Hauptteile gegliedert werden und behandelt alle Bereiche der Parallelverarbeitung beginnend mit der Architektur paralleler Plattformen bis hin zur Realisierung paralleler Anwendungsalgorithmen. Breiten Raum nimmt die eigentliche parallele Programmierung ein. Im ersten Teil geben wir einen kurzen Überblick über die Architektur paralleler Systeme, wobei wir uns vor allem auf wichtige prinzipielle Eigenschaften wie Cache- und Speicherorganisation oder Verbindungsnetzwerke

einschließlich der Routing- und Switching-Techniken konzentrieren, aber auch Hardwaretechnologien wie Hyperthreading oder Speicherkonsistenzmechanismen behandeln. Im zweiten Teil stellen wir Programmier- und Kostenmodelle sowie Methoden zur Formulierung paralleler Programme vor und beschreiben derzeit aktuelle portable Programmierungsumgebungen wie MPI, PVM, Pthreads, Java-Threads und OpenMP, gehen aber auch auf Sprachentwicklungen wie Unified Parallel C ein. Ausführliche Programmbeispiele begleiten die Darstellung der Programmierkonzepte und dienen zur Demonstration der Unterschiede zwischen den dargestellten Programmierungsumgebungen. Im dritten Teil wenden wir die dargestellten Programmiertechniken auf Algorithmen aus dem wissenschaftlich-technischen Bereich an. Wir konzentrieren uns dabei auf grundlegende Verfahren zur Behandlung linearer Gleichungssysteme, die für eine praktische Realisierung vieler Simulationsalgorithmen eine große Rolle spielen. Der Schwerpunkt der Darstellung liegt dabei nicht auf den mathematischen Eigenschaften der Lösungsverfahren, sondern auf der Untersuchung ihrer algorithmischen Struktur und den daraus resultierenden Parallelisierungsmöglichkeiten. Zu jedem Algorithmus geben wir z.T. mehrere, repräsentativ ausgewählte Parallelisierungsvarianten an, die sich im zugrundeliegenden Programmiermodell und der verwendeten Parallelisierungsstrategie unterscheiden. Eine Webseite mit begleitendem Material ist unter ai2.inf.uni-bayreuth.de/pp_buch eingerichtet. Dort sollen u.a. weitere Materialien zum Inhalt des Buches sowie Informationen zu neueren Entwicklungen zur Verfügung gestellt werden.

Bei der Erstellung des Manuskripts haben wir vielfältige Hilfestellung erfahren, und wir möchten an dieser Stelle all denen danken, die am Zustandekommen dieses Buches beteiligt waren. Für zahlreiche Anregungen und Verbesserungsvorschläge danken wir Jörg Dümmler, Sergei Gorlatch, Reiner Haupt, Hilmar Hennings, Klaus Hering, Michael Hofmann, Christoph Keßler, Raphael Kunis, Paul Molitor, John O'Donnell, Robert Reilein, Carsten Scholtes, Michael Schwind und Reinhard Wilhelm. Kerstin Beier, Erika Brandt, Daniela Funke, Monika Glaser, Ekkehard Petzold, Michael Stach und Michael Walter danken wir für die Mitarbeit an der $\text{\LaTeX}$ -Erstellung des Manuskriptes. Viele weitere Personen haben zum Gelingen dieses Buches durch zahlreiche Hinweise beigetragen; auch ihnen sei hiermit gedankt. Nicht zuletzt gilt unser Dank dem Springer-Verlag für die effiziente und angenehme Zusammenarbeit.

Bayreuth und Chemnitz, Dezember 2006

Thomas Rauber
Gudula Rünger

Inhaltsverzeichnis

1. Einleitung	1
1.1 Motivation	1
1.2 Begriffe der Parallelverarbeitung	4
1.3 Überblick	6
2. Architektur paralleler Plattformen	9
2.1 Überblick über die Prozessorentwicklung	10
2.2 Parallelität innerhalb eines Prozessorkerns	14
2.3 Klassifizierung von Parallelrechnern	17
2.4 Speicherorganisation von Parallelrechnern	20
2.4.1 Rechner mit physikalisch verteiltem Speicher	21
2.4.2 Rechner mit physikalisch gemeinsamem Speicher	25
2.4.3 Reduktion der Speicherzugriffszeiten	28
2.5 Verbindungsnetzwerke	32
2.5.1 Bewertungskriterien für Netzwerke	34
2.5.2 Direkte Verbindungsnetzwerke	37
2.5.3 Einbettungen	43
2.5.4 Dynamische Verbindungsnetzwerke	46
2.6 Routing- und Switching-Strategien	53
2.6.1 Routingalgorithmen	53
2.6.2 Switching	64
2.6.3 Flusskontrollmechanismen	72
2.7 Caches und Speicherhierarchien	73
2.7.1 Charakteristika von Cache-Speichern	74
2.7.2 Cache-Kohärenz	84
2.7.3 Speicherkonsistenz	92
2.8 Parallelität auf Threadebene	98
2.8.1 Simultanes Multithreading	99
2.8.2 Multicore-Prozessoren	100
2.9 Beispiele	104
2.9.1 Architektur des Intel Pentium 4	104
2.9.2 Architektur des Cell-Prozessors	107
2.9.3 IBM Blue Gene/L Supercomputer (BG/L)	108

3. Parallele Programmiermodelle	113
3.1 Modelle paralleler Rechnersysteme	114
3.2 Parallelisierung von Programmen	117
3.3 Ebenen der Parallelität	120
3.3.1 Parallelität auf Instruktionsebene	120
3.3.2 Datenparallelität	122
3.3.3 Parallelität in Schleifen	123
3.3.4 Funktionsparallelität	127
3.4 Explizite und implizite Darstellung der Parallelität	128
3.5 Strukturierung paralleler Programme	131
3.6 Datenverteilungen für Felder	134
3.7 Informationsaustausch	139
3.7.1 Gemeinsame Variablen	139
3.7.2 Kommunikationsoperationen	142
3.7.3 Parallele Matrix-Vektor-Multiplikation	150
4. Laufzeitanalyse paralleler Programme	157
4.1 Leistungsbewertung von Rechnersystemen	157
4.1.1 Bewertung der CPU-Leistung	158
4.1.2 MIPS und MFLOPS	160
4.1.3 Leistung von Prozessoren mit Cachespeichern	161
4.1.4 Benchmarkprogramme	163
4.2 Parallele Leistungsmaße	167
4.3 Modellierung von Laufzeiten	172
4.3.1 Realisierung von Kommunikationsoperationen	174
4.3.2 Kommunikationsoperationen auf dem Hyperwürfel	180
4.3.3 Kommunikationsoperationen auf einem Baum	189
4.4 Analyse von Laufzeitformeln	193
4.4.1 Paralleles Skalarprodukt	194
4.4.2 Parallele Matrix-Vektor-Multiplikation	196
4.5 Parallele Berechnungsmodelle	198
4.5.1 PRAM-Modelle	198
4.5.2 BSP-Modell	200
4.5.3 LogP-Modell	203
5. Message-Passing-Programmierung	207
5.1 Einführung in MPI	208
5.1.1 Einzeltransferoperationen	210
5.1.2 Globale Kommunikationsoperationen	224
5.1.3 Auftreten von Deadlocks	237
5.1.4 Prozessgruppen und Kommunikatoren	239
5.1.5 Prozessstopologien	245
5.1.6 Zeitmessung und Abbruch der Ausführung	250
5.2 Einführung in PVM	251
5.2.1 Programmiermodell	251

5.2.2	Prozesskontrolle	252
5.2.3	Austausch von Nachrichten	255
5.2.4	Verwaltung von Prozessgruppen	257
5.3	Einführung in MPI-2	260
5.3.1	Prozesserzeugung und -verwaltung	260
5.3.2	Einseitige Kommunikation	263
6.	Thread-Programmierung	275
6.1	Einführung in die Programmierung mit Threads	276
6.2	Programmiermodell und Grundlagen für Pthreads	281
6.2.1	Erzeugung und Verwaltung von Pthreads	285
6.2.2	Koordination von Threads	287
6.2.3	Implementierung eines Taskpools	302
6.2.4	Parallelität durch Pipelining	306
6.2.5	Realisierung eines Client-Server-Modells	312
6.2.6	Steuerung und Abbruch von Threads	317
6.2.7	Thread-Scheduling	324
6.2.8	Prioritätsinversion	329
6.2.9	Thread-spezifische Daten	333
6.3	Java-Threads	334
6.3.1	Erzeugung von Threads in Java	334
6.3.2	Synchronisation von Java-Threads	339
6.3.3	Signalmechanismus in Java	344
6.3.4	Erweiterte Java-Synchronisationsmuster	349
6.3.5	Thread-Scheduling in Java	351
6.4	OpenMP	354
6.4.1	Steuerung der parallelen Abarbeitung	355
6.4.2	Parallele Schleife	358
6.4.3	Nichtiterative parallele Bereiche	362
6.4.4	Koordination von Threads	365
6.5	Unified Parallel C	371
6.5.1	UPC Programmiermodell und Benutzung	372
6.5.2	Gemeinsame Felder	374
6.5.3	Speicherkonsistenzmodelle von UPC	375
6.5.4	Zeiger und Felder in UPC	377
6.5.5	Parallele Schleifen in UPC	379
6.5.6	UPC Synchronisation	380
7.	Lösung linearer Gleichungssysteme	383
7.1	Gauß-Elimination	384
7.1.1	Beschreibung der Methode	384
7.1.2	Parallele zeilenzyklische Implementierung	388
7.1.3	Parallele gesamtzyklische Implementierung	392
7.1.4	Laufzeitanalyse der gesamtzyklischen Implementierung	398
7.2	Direkte Verfahren für Gleichungssysteme mit Bandstruktur ..	403

7.2.1	Diskretisierung der Poisson-Gleichung	403
7.2.2	Lösung von Tridiagonalsystemen	409
7.2.3	Verallgemeinerung auf beliebige Bandmatrizen	420
7.2.4	Anwendung auf die Poisson-Gleichung	423
7.3	Klassische Iterationsverfahren	425
7.3.1	Beschreibung iterativer Verfahren	426
7.3.2	Parallele Realisierung des Jacobi-Verfahrens	430
7.3.3	Parallele Realisierung des Gauß-Seidel-Verfahrens	432
7.3.4	Rot-Schwarz-Anordnung	436
7.4	Cholesky-Faktorisierung für dünnbesetzte Matrizen	443
7.4.1	Sequentieller Algorithmus	444
7.4.2	Abspeicherungsschemata für dünnbesetzte Matrizen	450
7.4.3	Implementierung für gemeinsamen Adressraum	453
7.5	Methode der konjugierten Gradienten	460
7.5.1	Beschreibung der Methode	461
7.5.2	Parallelisierung des CG-Verfahrens	463
Literatur		469
Index		477

1. Einleitung

1.1 Motivation

Ein seit längerem zu beobachtender Trend ist die ständig steigende Nachfrage nach immer höherer Rechenleistung. Dies gilt insbesondere für Anwendungen aus dem Bereich der Simulation naturwissenschaftlicher Phänomene. Solche Anwendungen sind z.B. die Wettervorhersage, Windkanal- und Fahrsimulationen von Automobilen, das Design von Medikamenten oder computergraphische Anwendungen aus der Film-, Spiel- und Werbeindustrie. Je nach Einsatzgebiet ist die Computersimulation entweder die wesentliche Grundlage für die Errechnung des Ergebnisses oder sie ersetzt bzw. ergänzt physikalische Versuchsanordnungen. Ein typisches Beispiel für den ersten Fall ist die Wettersimulation, bei der es um die Vorhersage des Wetterverhaltens in den jeweils nächsten Tagen geht. Eine solche Vorhersage kann nur mit Hilfe von Simulationen erreicht werden. Ein weiteres Beispiel sind Havariefälle in Kraftwerken, da diese in der Realität nur schwer oder mit gravierenden Folgen nachgespielt werden könnten. Der Grund für den Einsatz von Computersimulationen im zweiten Fall ist zum einen, dass die Realität durch den Einsatz eines Computers genauer nachgebildet werden kann, als dies mit einer typischen Versuchsanordnung möglich ist, zum anderen können durch den Einsatz eines Computers vergleichbare Resultate kostengünstiger erzielt werden. So hat eine Computersimulation im Gegensatz zur klassischen Windkanalsimulation, in der das zu testende Fahrzeug in einen Windkanal gestellt und einem Windstrom ausgesetzt wird, den Vorteil, dass die relative Bewegung des Fahrzeuges zur Fahrbahn in die Simulation mit einbezogen werden kann, d.h. die Computersimulation kann prinzipiell zu realitätsnäheren Ergebnissen führen als die Windkanalsimulation. Crashtests von Autos sind ein offensichtliches Beispiel für ein Einsatzgebiet, in dem Computersimulationen in der Regel kostengünstiger sind als reale Tests.

Alle erwähnten Computersimulationen haben einen sehr hohen Berechnungsaufwand, und die Durchführung der Simulationen kann daher durch eine zu geringe Rechenleistung der verwendeten Computer eingeschränkt werden. Wenn eine höhere Rechenleistung zur Verfügung stünde, könnte diese zum einen zur schnelleren Berechnung einer Aufgabenstellung verwendet werden, zum anderen könnten aber auch größere Aufgabenstellungen, die zu genaueren Resultaten führen, in ähnlicher Rechenzeit bearbeitet werden.

Der Einsatz von Parallelverarbeitung bietet die Möglichkeit, eine wesentlich höhere Rechenleistung zu nutzen, als sie sequentielle Rechner bereitstellen, indem mehrere Prozessoren oder Verarbeitungseinheiten gemeinsam eine Aufgabe bearbeiten. Dabei können speziell für die Parallelverarbeitung entworfene Parallelrechner, aber auch über ein Netzwerk miteinander verbundene Rechner verwendet werden. Parallelverarbeitung ist jedoch nur möglich, wenn der für eine Simulation abzuarbeitende Algorithmus dafür geeignet ist, d.h. wenn er sich in Teilberechnungen zerlegen lässt, die unabhängig voneinander parallel ausgeführt werden können. Viele Simulationsalgorithmen aus dem wissenschaftlich-technischen Bereich erfüllen diese Voraussetzung.

Für die Nutzung der Parallelverarbeitung ist es notwendig, dass der Algorithmus für eine parallele Abarbeitung vorbereitet wird, indem er in einer parallelen Programmiersprache formuliert oder durch Einsatz von Programmierungsumgebungen mit zusätzlichen Direktiven oder Anweisungen versehen wird, die die parallele Abarbeitung steuern. Die dabei anzuwendenden Techniken und die dafür zur Verfügung stehenden Programmierungsumgebungen werden in diesem Buch vorgestellt. Die Erstellung eines effizienten parallelen Programms verursacht für den Anwendungsprogrammierer je nach Algorithmus z.T. einen recht großen Aufwand, der aber im Erfolgsfall ein Programm ergibt, das auf einer geeigneten Plattform um ein Vielfaches schneller abgearbeitet werden kann als das zugehörige sequentielle Programm. Durch den Einsatz portabler Programmierungsumgebungen ist das parallele Programm auf einer Vielzahl unterschiedlicher Plattformen ausführbar. Aufgrund dieser Vorteile wird die Parallelverarbeitung in vielen Bereichen erfolgreich eingesetzt.

Ein weiterer Grund, sich mit der parallelen Programmierung zu beschäftigen, besteht darin, dass die Parallelverarbeitung auch für sequentielle Rechner eine zunehmend wichtigere Rolle spielt, da nur durch parallele Technologien eine weitere Leistungssteigerung erreicht werden kann. Dies liegt auch daran, dass die Taktrate von Prozessoren, durch die ihre Verarbeitungsgeschwindigkeit bestimmt wird, nicht beliebig gesteigert werden kann. Die Gründe dafür liegen zum einen in der mit einer Erhöhung der Taktrate verbundenen erhöhten Leistungsaufnahme und Wärmeentwicklung. Zum anderen wirkt die Endlichkeit der Übertragungsgeschwindigkeit der Signale als limitierender Faktor und gewinnt mit zunehmender Taktrate an Einfluss. Ein mit 3 GHz arbeitender Prozessor hat eine Zykluszeit von etwa 0.33 ns . In dieser Zeit könnte ein Signal eine Entfernung von $0.33 \cdot 10^{-9} \text{ s} \cdot 0.3 \cdot 10^9 \text{ m/s} \approx 10 \text{ cm}$ zurücklegen, wobei als Obergrenze der Übertragungsgeschwindigkeit die Lichtgeschwindigkeit im Vakuum ($0.3 \cdot 10^9 \text{ m/s}$) angenommen wird. Bei einer Verzehnfachung der Taktrate könnten die Signale in einem Zyklus gerade noch 1 cm zurücklegen, womit die Größenordnung der Ausdehnung eines Prozessors erreicht wäre. Die maximal nutzbare Taktrate wird dann von den Signallaufzeiten bestimmt, so dass die Länge der Wege, die die Kontrollsignale und die Da-

ten zur Verarbeitung einer Instruktion durchlaufen müssen, eine wesentliche Rolle spielt.

Die Leistungssteigerung der Prozessoren ist in der Vergangenheit jedoch nicht allein auf eine Steigerung der Taktrate zurückzuführen gewesen, sondern auch durch architektonische Verbesserungen der Prozessoren erzielt worden, die zum großen Teil auf dem Einsatz interner Parallelverarbeitung beruhen. Aber auch diesen architektonischen Verbesserungen sind Grenzen gesetzt, die im Wesentlichen darin begründet sind, dass der Prozessor einen sequentiellen Befehlsstrom bearbeitet, der von einem Übersetzer aus einem sequentiellen Programm erzeugt wird und in der Regel viele Abhängigkeiten zwischen den abzuarbeitenden Instruktionen enthält. Dadurch bleibt der effektive Einsatz parallel arbeitender Funktionseinheiten innerhalb eines Prozessors begrenzt, obwohl die Fortschritte in der VLSI-Technologie eine Integration vieler Funktionseinheiten erlauben würden. Daher wurde mit der Entwicklung von Multicore-Prozessoren begonnen, die mehrere Prozessorkerne (engl. *execution cores*) auf einem Prozessorchip integrieren. Jeder Prozessorkern ist eine unabhängige Verarbeitungseinheit, die von einem separaten Befehlsstrom gesteuert wird. Zur effizienten Ausnutzung der Prozessorkerne eines Multicore-Prozessors ist es daher erforderlich, dass mehrere Berechnungsströme verfügbar sind, die den Prozessorkernen zugeordnet werden können und die beim Zugriff auf gemeinsame Daten auch koordiniert werden müssen.

Zur Bereitstellung eines Berechnungsstroms für jeden Prozessorkern können im Prinzip zwei unterschiedliche Ansätze verfolgt werden. Zum einen kann versucht werden, die Übersetzerbautechniken so zu verbessern, dass der Übersetzer aus einem sequentiellen Befehlsstrom mehrere unabhängige Berechnungsströme erzeugt, die dann gleichzeitig verschiedenen Verarbeitungseinheiten zugeordnet werden. Dieser Ansatz wird seit vielen Jahren verfolgt, die Komplexität der Problemstellung hat aber bisher eine für eine breite Klasse von Anwendungen zufriedenstellende Lösung verhindert. Ein anderer Ansatz besteht darin, dem Übersetzer bereits mehrere Befehlsströme für die Übersetzung zur Verfügung zu stellen, so dass dieser sich auf die eigentliche Übersetzung konzentrieren kann. Dies kann durch Anwendung von Techniken der parallelen Programmierung erreicht werden, indem der Programmierer ein paralleles Programm bereitstellt. Dieser Ansatz ist am vielversprechendsten, bewirkt aber, dass bei Verwendung von Multicore-Prozessoren für die effiziente Nutzung typischer Desktop-Rechner Programmierertechniken der Parallelverarbeitung eingesetzt werden müssen.

Das vorliegende Buches soll dem Leser die wichtigsten Programmieretechniken für alle Einsatzgebiete der parallelen Programmierung vermitteln. Bevor wir einen detaillierten Überblick über den Inhalt dieses Buches geben, möchten wir im folgenden Abschnitt grundlegende Begriffe der Parallelverarbeitung einführen. Diese werden dann in den späteren Kapiteln näher präzisiert.

1.2 Begriffe der Parallelverarbeitung

Eines der wichtigsten Ziele der Parallelverarbeitung ist es, Aufgaben in einer kürzeren Ausführungszeit zu erledigen, als dies durch eine Ausführung auf sequentiellen Rechnerplattformen möglich wäre. Die durch den Einsatz paralleler Rechnertechnologie erhaltene erhöhte Rechenleistung wird häufig auch dazu genutzt, komplexere Aufgabenstellungen zu bearbeiten, die zu besseren oder genaueren Lösungen führen, als sie durch den Einsatz einer sequentiellen Rechnerplattform in vertretbarer Zeit möglich wären. Andere Ziele der Parallelverarbeitung sind das Erreichen von Ausfallsicherheit durch Replikation von Berechnungen oder die Erfüllung größerer Speicheranforderungen.

Die Grundidee zur Erreichung einer kürzeren Ausführungszeit besteht darin, die Ausführungszeit eines Programms dadurch zu reduzieren, dass mehrere Berechnungsströme erzeugt werden, die gleichzeitig, also parallel, ausgeführt werden können und durch koordinierte Zusammenarbeit die gewünschte Aufgabe erledigen. Zur Erzeugung der Berechnungsströme wird die auszuführende Aufgabe in *Teilaufgaben* zerlegt. Zur Benennung solcher Teilaufgaben haben sich die Begriffe *Prozesse*, *Threads* oder *Tasks* herausgebildet, die für unterschiedliche Programmiermodelle und -umgebungen jedoch geringfügig unterschiedliche Bedeutungen haben können. Zur tatsächlichen parallelen Abarbeitung werden die Teilaufgaben auf physikalische Berechnungseinheiten abgebildet, was auch als *Mapping* bezeichnet wird. Dies kann statisch zur Übersetzungszeit oder dynamisch zur Laufzeit des Programms stattfinden. Zur Vereinfachung der Darstellung werden wir im Folgenden unabhängige physikalische Berechnungseinheiten als Prozessoren bezeichnen und meinen damit sowohl Prozessoren als auch Prozessorkerne eines Multicore-Prozessors.

Typischerweise sind die erzeugten Teilaufgaben nicht vollkommen unabhängig voneinander, sondern können durch *Daten- und Kontrollabhängigkeiten* gekoppelt sein, so dass bestimmte Teilaufgaben nicht ausgeführt werden können, bevor andere Teilaufgaben benötigte Daten oder Informationen für den nachfolgenden Kontrollfluss bereitgestellt haben. Eine der wichtigsten Aufgaben der parallelen Programmierung ist es, eine korrekte Abarbeitung der parallelen Teilaufgaben durch geeignete *Synchronisation* und notwendigen *Informationsaustausch* zwischen den Berechnungsströmen sicherzustellen. Parallele Programmiermodelle und -umgebungen stellen hierzu eine Vielzahl unterschiedlicher Methoden und Mechanismen zur Verfügung. Eine grobe Unterteilung solcher Mechanismen kann anhand der Unterscheidung in Programmiermodelle mit *gemeinsamem* und *verteiltem Adressraum* erfolgen, die sich eng an die Hardwaretechnologie der Speicherorganisation anlehnt.

Bei einem *gemeinsamen Adressraum* wird dieser über gemeinsam zugreifbare Variablen zum Informationsaustausch genutzt. Diese einfache Art des Informationsaustausches wird durch vielfältige Mechanismen zur Synchronisation der meist als Threads bezeichneten Berechnungsströme ergänzt, die den konkurrierenden Datenzugriff durch mehrere Threads koordinieren.

Bei einem *verteilten Adressraum* sind die Daten eines parallelen Programms in privaten Adressbereichen abgelegt, auf die nur der entsprechende, meist als Prozess bezeichnete Berechnungsstrom Zugriff hat. Ein Informationsaustausch kann durch explizite Kommunikationsanweisungen erfolgen, mit denen ein Prozess Daten seines privaten Adressbereichs an einen anderen Prozess senden kann. Zur Koordination der parallelen Berechnungsströme eignet sich eine Synchronisation in Form einer *Barrier-Synchronisation*. Diese bewirkt, dass alle beteiligten Prozesse aufeinander warten und kein Prozess eine nach der Synchronisation stehende Anweisung ausführt, bevor nicht die restlichen Prozesse den Synchronisationspunkt erreicht haben.

Zur Bewertung der Ausführungszeit paralleler Programme werden verschiedene Kostenmaße verwendet. Die *parallele Laufzeit* eines Programms setzt sich aus der Rechenzeit der einzelnen Berechnungsströme und der Zeit für den erforderlichen Informationsaustausch oder benötigte Synchronisationen zusammen. Zur Erreichung einer geringen parallelen Laufzeit sollte eine möglichst gleichmäßige Verteilung der Rechenlast auf die Prozessoren angestrebt werden (*Load balancing*), so dass ein *Lastgleichgewicht* entsteht. Ein Vermeiden langer Wartezeiten und möglichst wenig Informationsaustausch sind insbesondere für parallele Programme mit verteiltem Adressraum wichtig zur Erzielung einer geringen parallelen Laufzeit. Für einen gemeinsamen Adressraum sollten entsprechend Wartezeiten an Synchronisationspunkten minimiert werden.

Die Zuordnung von Teilaufgaben an Prozessoren sollte so gestaltet werden, dass Teilaufgaben, die häufig Informationen austauschen müssen, dem gleichen Prozessor zugeordnet werden. Ein gleichzeitiges Erreichen eines optimalen Lastgleichgewichts und eine Minimierung des Informationsaustausches ist oft schwierig, da eine Reduzierung des Informationsaustausches zu einem Lastungleichgewicht führen kann, während eine gleichmäßige Verteilung der Arbeit größere Abhängigkeiten zwischen den Berechnungsströmen und damit mehr Informationsaustausch verursachen kann. Zur Bewertung der resultierenden Berechnungszeit eines parallelen Programms im Verhältnis zur Berechnungszeit eines entsprechenden sequentiellen Programms werden Kostenmaße wie *Speedup* und *Effizienz* verwendet.

Das Erreichen einer Gleichverteilung der Last hängt eng mit der Zerlegung in Teilaufgaben zusammen. Die Zerlegung legt den Grad der Parallelität sowie die *Granularität*, d.h. die durchschnittliche Größe der Teilaufgaben (z.B. gemessen als Anzahl der Instruktionen) fest. Um eine hohe Flexibilität bei der Zuteilung von Teilaufgaben an Prozessoren sicherzustellen und eine gleichmäßige Lastverteilung zu ermöglichen, ist ein möglichst hoher Grad an Parallelität günstig. Zur Reduktion des Verwaltungsaufwandes für die Abarbeitung der Teilaufgaben durch die einzelnen Prozessoren ist es dagegen erstrebenswert, mit möglichst wenigen Teilaufgaben entsprechend grober Granularität zu arbeiten, d.h. auch hier muss ein Kompromiss zwischen entgegengesetzten Zielstellungen gefunden werden. Die Abarbeitung

der erzeugten Teilaufgaben unterliegt den geschilderten Einschränkungen, die die erreichbare Anzahl von parallel, also gleichzeitig auf verschiedenen Prozessoren ausführbaren Teilaufgaben bestimmen. In diesem Zusammenhang spricht man auch vom erreichbaren *Grad der Parallelität* oder dem *potentiellen Parallelitätsgrad* einer Anwendung. Der Entscheidungsvorgang, in welcher Reihenfolge die Teilaufgaben (unter Berücksichtigung der Abhängigkeiten) parallel abgearbeitet werden, wird *Scheduling* genannt. Es werden statische, d.h. zur Übersetzungszeit arbeitende, oder dynamische, d.h. während des Programmaufbaues arbeitende, Schedulingalgorithmen auf verschiedenen Parallelisierungsebenen genutzt. Schedulingverfahren und -algorithmen werden in der Parallelverarbeitung in sehr unterschiedlicher Form benötigt. Dies reicht von Thread-Scheduling in vielen Modellen des gemeinsamen Adressraums bis zum Scheduling von Teilaufgaben mit Abhängigkeiten auf Programmebene in der Programmierung für verteilten Adressraum.

Die Granularität und Anzahl der Teilaufgaben wird also Wesentlich von den für die betrachtete Anwendung durchzuführenden Berechnungen und den Abhängigkeiten zwischen diesen Berechnungen bestimmt. Die genaue Abbildung der Teilaufgaben auf die Prozessoren hängt zusätzlich von der Architektur des verwendeten Parallelrechners und von der verwendeten Programmiersprache oder Programmierungsumgebung ab. Dieses Zusammenspiel der parallelen Eigenschaften des zu bearbeitenden Anwendungsproblems, der Architektur des Parallelrechners und der Programmierungsumgebung ist grundlegend für die parallele Programmierung. Wir werden dem Rechnung tragen, indem wir in den einzelnen Kapiteln zunächst auf die unterschiedlichen Typen von Parallelrechnern und parallelen Plattformen eingehen, einen Überblick über parallele Programmierungsumgebungen geben und abschließend Charakteristika wichtiger Algorithmen aus dem Bereich des wissenschaftlichen Rechnens behandeln. Wir stellen die Inhalte der einzelnen Kapitel nun noch etwas genauer vor.

1.3 Überblick

Kapitel 2 gibt einen Überblick über die Architektur paralleler Plattformen und behandelt deren Ausprägungen hinsichtlich der Kontrollmechanismen, der Speicherorganisation und des Verbindungsnetzwerkes. Bei der Speicherorganisation wird im wesentlichen unterschieden zwischen Rechnern mit verteiltem Speicher, bei denen der Speicher in Form lokaler Speicher für die einzelnen Prozessoren vorliegt, und Rechnern mit gemeinsamem Speicher, bei denen alle Prozessoren den gleichen globalen Speicher gemeinsam nutzen. Diese Unterscheidung wird Grundlage für die später vorgestellten Programmierungsumgebungen sein. Zu Plattformen mit gemeinsamem Speicher gehören auch Desktop-Rechner, die mit Multicore-Prozessoren ausgestattet sind. Es gibt auch aktuelle Hybridmodelle, die sich durch Speicherhierarchien und Caches verschiedener Stufen auszeichnen. Dazu gehören Clustersysteme, die aus

mehreren Multicore-Prozessoren bestehen, deren Prozessorkerne jeweils auf einen gemeinsamen Speicher zugreifen, während Prozessorkerne unterschiedlicher Prozessoren Informationen und Daten über ein Verbindungsnetzwerk austauschen müssen. Diese Abschnitte können bei einer stärkeren Konzentration auf die Programmierung übersprungen werden, ohne dass das Verständnis für spätere Kapitel beeinträchtigt wird. Die Verbindungsnetzwerke und deren Routing- und Switchingstrategien sind ein erster Ansatzpunkt für Kostenmodelle für den Rechenzeitbedarf paralleler Programme, an den spätere Kapitel zu Kostenmodellen anknüpfen.

Kapitel 3 stellt parallele Programmiermodelle und -paradigmen vor und beschreibt die auf verschiedenen Programmebenen verfügbare Parallelität sowie die Möglichkeiten ihrer Ausnutzung in parallelen Programmumgebungen. Insbesondere werden die für einen gemeinsamen oder verteilten Adressraum benötigten Koordinations-, Synchronisations- und Kommunikationsoperationen vorgestellt. Kapitel 4 führt grundlegende Definitionen zur Bewertung paralleler Programme ein und beschreibt, wie Kostenmodelle dazu verwendet werden können, eine quantitative Abschätzung der Laufzeit paralleler Programme für einen speziellen Parallelrechner zu erhalten. Dadurch ist die Grundlage für eine statische Planung der Abarbeitung paralleler Programme gegeben. Kapitel 5 beschreibt portable Programmierumgebungen für einen verteilten Adressraum, die oft in Form von Message-Passing-Bibliotheken für Plattformen mit verteiltem Speicher eingesetzt werden. Dies sind MPI (Message Passing Interface), PVM (Parallel Virtual Machine) und MPI-2, das eine Erweiterung von MPI darstellt. Wir geben eine Beschreibung der durch diese Umgebungen zur Verfügung gestellten Funktionen und demonstrieren die zum Entwurf paralleler Programme notwendigen Techniken an Beispielprogrammen. Kapitel 6 beschreibt Programmierumgebungen für einen gemeinsamen Adressraum und gibt einen detaillierten Überblick über die Pthreads-Bibliothek, die von vielen UNIX-ähnlichen Betriebssystemen unterstützt wird, und über OpenMP, das als Standard vor allem für Programme des wissenschaftlichen Rechnens vorgeschlagen wurde. Dieses Kapitel geht auch auf sprachbasierte Threadansätze wie Java-Threads oder Unified Parallel C (UPC) ein.

Kapitel 7 behandelt die algorithmischen Eigenschaften direkter und iterativer Verfahren zur Lösung linearer Gleichungssysteme und beschreibt für jedes Verfahren mehrere Möglichkeiten einer parallelen Implementierung für einen verteilten Adressraum. Um dem Leser die Erstellung der zugehörigen parallelen Programme zu erleichtern, geben wir Programmfragmente an, die die relevanten Details der Steuerung der parallelen Abarbeitung enthalten und die relativ einfach zu kompletten Programmen ausgebaut werden können. Für Programmiermodelle mit verteiltem Adressraum werden die erforderlichen Kommunikationsoperationen in MPI ausgedrückt.

2. Architektur paralleler Plattformen

Wie in der Einleitung bereits angerissen wurde, hängen die Möglichkeiten einer parallelen Abarbeitung stark von den Gegebenheiten der benutzten Hardware ab. Wir wollen in diesem Kapitel daher den prinzipiellen Aufbau paralleler Plattformen behandeln, auf die die auf Programmebene gegebene Parallelität abgebildet werden kann, um eine tatsächlich gleichzeitige Abarbeitung verschiedener Programmtteile zu erreichen. In den Abschnitten 2.1 und 2.2 beginnen wir mit einer kurzen Darstellung der innerhalb eines Prozessors oder Prozessorkerns zur Verfügung stehenden Möglichkeiten einer parallelen Verarbeitung. Hierbei wird deutlich, dass schon bei einzelnen Prozessorkernen eine Ausnutzung der verfügbaren Parallelität (auf Instruktionsebene) zu einer erheblichen Leistungssteigerung führen kann. Die weiteren Abschnitte des Kapitels sind Hardwarekomponenten von Parallelrechnern gewidmet. In den Abschnitten 2.3 und 2.4 gehen wir auf die Kontroll- und Speicherorganisation paralleler Plattformen ein, indem wir zum einen die Flynn'sche Klassifikation einführen und zum anderen Rechner mit verteiltem und Rechner mit gemeinsamem Speicher einander gegenüberstellen.

Eine weitere wichtige Komponente paralleler Hardware sind Verbindungsnetzwerke, die Prozessoren und Speicher bzw. verschiedene Prozessoren physikalisch miteinander verbinden. Verbindungsnetzwerke spielen auch bei Multicore-Prozessoren eine große Rolle, und zwar zur Verbindung der Prozessorkerne untereinander sowie mit den Caches des Prozessorchips. Statische und dynamische Verbindungsnetzwerke und deren Bewertung anhand verschiedener Kriterien wie Durchmesser, Bisektionsbandbreite, Konnektivität und Einbettbarkeit anderer Netzwerke werden in Abschnitt 2.5 eingeführt. Zum Verschieben von Daten zwischen zwei Prozessoren wird das Verbindungsnetzwerk genutzt, wozu meist mehrere Pfade im Verbindungsnetzwerk zur Verfügung stehen. In Abschnitt 2.6 beschreiben wir Routingtechniken zur Auswahl eines solchen Pfades durch das Netzwerk und betrachten Switchingverfahren, die die Übertragung der Nachricht über einen vorgegebenen Pfad regeln. In Abschnitt 2.7 werden Speicherhierarchien sequentieller und paralleler Plattformen betrachtet. Wir gehen insbesondere auf die bei parallelen Plattformen auftretenden Cachekohärenz- und Speicherkonsistenzprobleme ein. In Abschnitt 2.8 werden Prozessortechnologien wie simultanes Multithreading oder Multicore-Prozessoren zur Realisierung prozessorinterner Parallelverar-

beitung auf Thread- oder Prozessebene vorgestellt. Abschließend enthält Abschnitt 2.9 eine kurze Beschreibung der Architektur ausgewählter Prozessoren und Parallelrechner.

2.1 Überblick über die Prozessorentwicklung

Für Prozessoren, die als Kernbausteine von Computern verwendet werden, sind bestimmte Trends festzustellen, die die Basis für Prognosen über die weitere voraussichtliche Entwicklung bilden. Ein wichtiger Trend ist eine ständige Steigerung der Leistungsmerkmale der Prozessoren. Insbesondere ist zu beobachten, dass die Taktrate der Prozessoren seit mehr als 20 Jahren durchschnittlich um etwa 30 % pro Jahr steigt. Entsprechend sinkt die Zykluszeit und damit die Zeit für das Ausführen von Instruktionen. Die Taktrate der Prozessoren wird auch in den nächsten Jahren ansteigen, allerdings wird dies wegen des mit einer Steigerung der Taktrate verbundenen erhöhten Stromverbrauchs und der damit verbundenen Wärmeentwicklung nach Prognosen [74, 95] nicht mehr mit der bisherigen Geschwindigkeit erfolgen können. Dagegen wächst die für die Prozessorchips verwendete Anzahl der Transistoren, die ein ungefähres Maß für die Komplexität des Schaltkreises ist, pro Jahr um etwa 60 bis 80 %. Dadurch steht ständig mehr Platz für Register, Caches und Funktionseinheiten zur Verfügung. Diese von der Prozessorfertigungstechnik getragene, seit über 40 Jahren gültige empirische Beobachtung wird auch als Gesetz von Moore (engl. *Moore's law*) bezeichnet. Ein typischer Prozessor aus dem Jahr 2006 besteht aus 200 bis 400 Millionen Transistoren. Ein Intel Pentium Dual-Core hat beispielsweise ca. 233 Millionen Transistoren, ein IBM Cell-Prozessor ca. 250 Millionen Transistoren, ein Itanium 2 hat etwa 592 Millionen Transistoren.

Zur Leistungsbewertung von Prozessoren können Benchmarks verwendet werden, die meist eine Sammlung von Programmen aus verschiedenen Anwendungsbereichen sind und deren Ausführung repräsentativ für die Nutzung eines Rechnersystems sein soll. Häufig verwendet werden die SPEC-Benchmarks (*System Performance and Evaluation Cooperative*), die zur Messung der Integer- bzw. Floating-Point-Performance eines Rechners dienen [81, 121, 152], vgl. auch www.spec.org. Messungen dieses Benchmarks zeigen, dass die Integer-Performance von Prozessoren um durchschnittlich etwa 55 % pro Jahr steigt, die Floating-Point-Performance sogar um durchschnittlich etwa 75 %. Diese Erhöhung der Leistung der Prozessoren über die Erhöhung der Taktrate hinaus lässt erkennen, dass die Erhöhung der Anzahl der Transistoren zu architektonischen Verbesserungen genutzt wurde, die die durchschnittliche Zeit für die Ausführung einer Instruktion reduzieren. Wir werden im Folgenden einen kurzen Überblick über diese Verbesserungsmöglichkeiten geben, wobei der Einsatz der Parallelverarbeitung im Vordergrund steht. Es sind vier Stufen der Prozessorentwicklung zu beobachten [31], deren zeitliche Entstehung sich z.T. überlappt:

1. **Parallelität auf Bitebene:** Bis etwa 1986 wurde die Wortbreite der Prozessoren, d.h. die Anzahl der Bits, die parallel zueinander verarbeitet werden können, sukzessive auf 32 Bits und bis Mitte der 90er Jahre allmählich auf 64 Bits erhöht. Diese Entwicklung wurde zum einen durch die Anforderungen an die Genauigkeit von Floating-Point-Zahlen getragen, zum anderen durch den Wunsch, einen genügend großen Adressraum ansprechen zu können. Die Entwicklung der Erhöhung der Wortbreite stoppte (vorläufig) bei einer Wortbreite von 64 Bits, da mit 64 Bits für die meisten Anwendungen eine ausreichende Genauigkeit für Floating-Point-Zahlen und die Adressierung eines ausreichend großen Adressraumes von 2^{64} Worten gegeben ist.
2. **Parallelität durch Pipelining:** Die Idee des Pipelinings auf Instruktionsebene besteht darin, die Verarbeitung einer Instruktion in Teilaufgaben zu zerlegen, die von zugeordneten Hardwareeinheiten (sogenannten Pipelinestufen) nacheinander ausgeführt werden. Eine typische Zerlegung besteht z.B. aus folgenden Stufen:
 - a) dem Laden der nächsten auszuführenden Instruktion (*fetch*),
 - b) dem Dekodieren dieser Instruktion (*decode*),
 - c) der Bestimmung der Adressen der Operanden und der Ausführung der Instruktion (*execute*) und
 - d) dem Zurückschreiben des Resultates (*write back*).

Der Vorteil der Pipelineverarbeitung besteht darin, dass die verschiedenen Pipelinestufen parallel zueinander arbeiten können (Fließbandprinzip), falls keine Kontroll- und Datenabhängigkeiten zwischen nacheinander auszuführenden Instruktionen vorhanden sind, vgl. Abbildung 2.1. Zur Vermeidung von Wartezeiten sollte die Ausführung der verschiedenen Pipelinestufen etwa gleich lange dauern. Diese Zeit bestimmt dann den Maschinenzzyklus der Prozessoren. Im Idealfall wird bei einer Pipelineverarbeitung in jedem Maschinenzzyklus die Ausführung einer Instruktion beendet und die Ausführung der folgenden Instruktion begonnen. Damit bestimmt die Anzahl der Pipelinestufen den erreichbaren *Grad an Parallelität*. Die Anzahl der Pipelinestufen hängt üblicherweise von der auszuführenden Instruktion ab und liegt meist zwischen 2 und 26 Stufen. Prozessoren, die zur Ausführung von Instruktionen Pipelineverarbeitung einsetzen, werden auch als (skalare) **ILP-Prozessoren** (*instruction level parallelism*) bezeichnet. Prozessoren mit relativ vielen Pipelinestufen heißen auch **superpipelined**. Obwohl der ausnutzbare Grad an Parallelität mit der Anzahl der Pipelinestufen steigt, kann die Zahl der verwendeten Pipelinestufen nicht beliebig erhöht werden, da zum einen die Instruktionen nicht beliebig in gleich große Teilaufgaben zerlegt werden können, und zum anderen eine vollständige Ausnutzung der Pipelinestufen oft durch Datenabhängigkeiten verhindert wird.

3. **Parallelität durch mehrere Funktionseinheiten: Superskalare Prozessoren und VLIW-Prozessoren** (*very long instruction word*) ent-

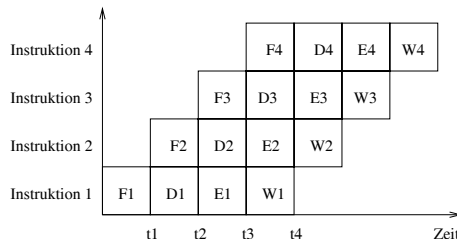


Abb. 2.1. Überlappende Ausführung voneinander unabhängiger Instruktionen nach dem Pipelining-Prinzip. Die Abarbeitung jeder Instruktion ist in vier Teilaufgaben zerlegt: *fetch* (F), *decode* (D), *execute* (E), *write back* (W).

halten mehrere unabhängige Funktionseinheiten wie ALUs (*arithmetic logical unit*), FPUs (*floating point unit*), Speicherzugriffseinheiten (*load/store unit*) oder Sprungeinheiten (*branch unit*), die parallel zueinander verschiedene unabhängige Instruktionen ausführen und die zum Laden von Operanden auf Register zugreifen können. Damit ist eine weitere Steigerung der mittleren Verarbeitungsgeschwindigkeit von Instruktionen möglich. In Abschnitt 2.2 geben wir einen kurzen Überblick über den Aufbau superskalarer Prozessoren. Beispiele superskalarer Prozessoren sind in Tabelle 2.1 dargestellt.

Die Grenzen des Einsatzes parallel arbeitender Funktionseinheiten sind durch die Datenabhängigkeiten zwischen benachbarten Instruktionen vorgegeben, die für superskalare Prozessoren zur Laufzeit des Programms ermittelt werden müssen. Dafür werden zunehmend komplexere Schedulingverfahren eingesetzt, die die auszuführenden Instruktionen den Funktionseinheiten zuordnen. Die Komplexität der Schaltkreise wird dadurch z.T. erheblich vergrößert, ohne dass dies mit einer entsprechenden Leistungssteigerung einhergeht. Simulationen haben außerdem gezeigt, dass die Möglichkeit des Absetzens von mehr als vier Instruktionen pro Maschinenzyklus gegenüber einem Prozessor, der bis zu vier Instruktionen pro Maschinenzyklus absetzen kann, für viele Programme nur zu einer geringen Leistungssteigerung führen würde, da Datenabhängigkeiten und Sprünge oft eine parallele Ausführung von mehr als vier Instruktionen verhindern [31, 88].

- 4. Parallelität auf Prozess- bzw. Threadebene:** Die bisher beschriebene Ausnutzung von Parallelität geht von *einem* sequentiellen Kontrollfluss aus, der vom Übersetzer zur Verfügung gestellt wird und der die gültigen Abarbeitungsreihenfolgen festlegt, d.h. bei Datenabhängigkeiten muss die vom Kontrollfluss vorgegebene Reihenfolge eingehalten werden. Dies hat den Vorteil, dass für die Programmierung eine sequentielle Programmiersprache verwendet werden kann und dass trotzdem eine parallele Abarbeitung von Instruktionen erreicht wird. Dem durch den Einsatz mehrerer Funktionseinheiten und Pipelining erreichbaren Potential an

Tabelle 2.1. Überblick über verschiedene superskalare Prozessoren nach [75]. Die Spalten enthalten von links nach rechts die maximale Anzahl der in einem Zyklus absetzbaren Instruktionen, die maximale Anzahl der darin enthaltenen Integer-Instruktionen (ALU), Floating-Point-Instruktionen (FPU), Speicherzugriffs-Instruktionen (LS) und Sprung-Instruktionen (B für Branch). Die angegebene Taktrate ist die Taktrate der jeweils ersten ausgelieferten Prozessoren im angegebenen Jahr der Einführung. Die Itanium-Prozessoren arbeiten nach dem VLIW-Prinzip, wobei je drei Instruktionen in eine Makro-Instruktion der Länge 128 Bits gepackt werden. Die zugrundeliegende IA64-Architektur erlaubt die zusätzliche Integration mehrerer Blöcke von je drei Funktionseinheiten (ALU, FPU, LS). Je nach Ausführung des Prozessors kann die Anzahl der angegebenen Funktionseinheiten daher auch ein Vielfaches der angegebenen Werte sein.

Prozessor	absetzbare Instruktionen					Taktrate (MHz)	Jahr
	Max.	ALU	FPU	LS	B		
Intel Pentium	2	2	1	2	1	66	1993
DEC Alpha 21164	4	2	2	2	1	300	1995
MIPS R10000	4	2	2	1	1	200	1995
Sun Ultra-SPARC	4	1	1	1	1	167	1995
IBM PowerPC 604	4	3	1	1	1	166	1995
HP 8000	4	2	2	2	1	200	1996
Intel Pentium II	3	2	1	3	1	450	1998
DEC Alpha 21264	6	2	2	2	1	575	1998
Intel Pentium III	3	2	1	2	1	1100	1999
Intel Pentium 4	3	3	2	2	1	1400	2001
AMD Athlon	3	3	3	2	1	1330	2001
Intel Itanium	6	4	2	2	3	800	2001
Intel Itanium 2	6	6	2	4	3	1500	2004
AMD Opteron	3	3	2+1	2	1	1800	2003

Parallelität sind jedoch Grenzen gesetzt, die – wie dargestellt – für aktuelle Prozessoren bereits erreicht sind. Nach dem Gesetz von Moore stehen aber ständig mehr Transistoren auf einer Chipfläche zur Verfügung. Diese können zwar z.T. für die Integration größerer Caches auf der Chipfläche genutzt werden, die Caches können aber auch nicht beliebig vergrößert werden, da größere Caches eine erhöhte Zugriffszeit erfordern, vgl. Abschnitt 2.7.

Als eine alternative Möglichkeit zur Nutzung der steigenden Anzahl von verfügbaren Transistoren werden seit 2005 sogenannte Multicore-Prozessoren gefertigt, die mehrere unabhängige Prozessorkerne auf der Chipfläche eines Prozessors integrieren. Im Unterschied zu bisherigen Prozessoren muss jeder der Prozessorkerne eines Multicore-Prozessors mit einem separaten Kontrollfluss versorgt werden. Da die Prozessorkerne eines Multicore-Prozessors auf den Hauptspeicher und auf evtl. gemeinsame Caches gleichzeitig zugreifen können, ist ein koordiniertes Zusammenarbeiten dieser Kontrollflüsse erforderlich. Dazu können Techniken der parallelen Programmierung verwendet werden, wie sie in diesem Buch besprochen werden.

Wir werden im folgenden Abschnitt einen kurzen Überblick darüber geben, wie die Parallelität durch mehrere Funktionseinheiten realisiert wird. Für eine detailliertere Darstellung verweisen wir auf [31, 75, 121, 153]. In Abschnitt 2.8 gehen wir auf Techniken der Prozessororganisation wie simultanes Multithreading oder Multicore-Prozessoren ein, die eine explizite Spezifikation der Parallelität erfordern.

2.2 Parallelität innerhalb eines Prozessorkerns

Die meisten der heute verwendeten und entwickelten Prozessoren sind **superskalare** Prozessoren oder **VLIW-Prozessoren**, die mehrere Instruktionen *gleichzeitig* absetzen und unabhängig voneinander verarbeiten können. Dazu stehen mehrere Funktionseinheiten zur Verfügung, die unabhängige Instruktionen parallel zueinander bearbeiten können. Der Unterschied zwischen superskalaren Prozessoren und VLIW-Prozessoren liegt im Scheduling der Instruktionen: Ein Maschinenprogramm für superskalare Prozessoren besteht aus einer sequentiellen Folge von Instruktionen, die per Hardware auf die zur Verfügung stehenden Funktionseinheiten verteilt werden, wenn die Datenabhängigkeiten zwischen den Instruktionen dies erlauben. Dabei wird ein *dynamisches*, d.h. zur Laufzeit des Programmes arbeitendes Scheduling der Instruktionen verwendet, was eine zusätzliche Erhöhung der Komplexität der Hardware erfordert. Im Unterschied dazu wird für VLIW-Prozessoren ein *statisches* Scheduling verwendet. Ein spezieller Übersetzer erzeugt dazu Maschinenprogramme mit Instruktionsworten, die für jede Funktionseinheit angeben, welche Instruktion zum entsprechenden Zeitpunkt ausgeführt wird. Ein Beispiel für ein solches statisches Schedulingverfahren ist Trace-Scheduling [43]. Die Instruktionsworte für VLIW-Prozessoren sind also in Abhängigkeit von der Anzahl der Funktionseinheiten recht lang, was den Prozessoren den Namen gegeben hat. Beispiele für VLIW-Prozessoren sind die Intel IA64-Prozessoren (Itanium und Itanium 2), die für eingebettete Systeme verwendete Trimedia TM32-Architektur und der für mobile Geräte verwendete Transmeta Crusoe. Wir betrachten im Folgenden superskalare Prozessoren, da diese z.Z. verbreiteter als VLIW-Prozessoren sind.

Abbildung 2.2(a) zeigt den typischen Aufbau eines superskalaren Prozessors. Zur Verarbeitung einer Instruktion wird diese von einer Zugriffseinheit (engl. *fetch unit*) über den Instruktionscache geladen und an eine Dekodiereinheit (engl. *decode unit*) weitergegeben, die die auszuführende Operation ermittelt. Damit mehrere Funktionseinheiten versorgt werden können, können die Zugriffseinheit und die Dekodiereinheit in jedem Maschinenzyklus mehrere Instruktionen laden bzw. dekodieren. Nach der Dekodierung der Instruktionen werden diese, wenn keine Datenabhängigkeiten zwischen ihnen bestehen, an die zugehörigen Funktionseinheiten zur Ausführung weitergegeben. Die Ergebnisse der Berechnungen werden in die angegebenen Ergebnisregister

zurückgeschrieben. Um bei superskalaren Prozessoren die Funktionseinheiten möglichst gut auszulasten, sucht der Prozessor in jedem Verarbeitungsschritt ausgehend von der aktuellen Instruktion nachfolgende Instruktionen, die wegen fehlender Datenabhängigkeiten bereits ausgeführt werden können (dynamisches Scheduling). Dabei spielen sowohl die Reihenfolge, in der die Instruktionen in die Funktionseinheiten geladen werden, als auch die Reihenfolge, in der Resultate der Instruktionen in die Register zurückgeschrieben werden, eine Rolle. Die größte Flexibilität wird erreicht, wenn in beiden Fällen die Reihenfolge der Instruktionen im Maschinenprogramm nicht bindend ist (engl. *out-of-order issue*, *out-of-order completion*).

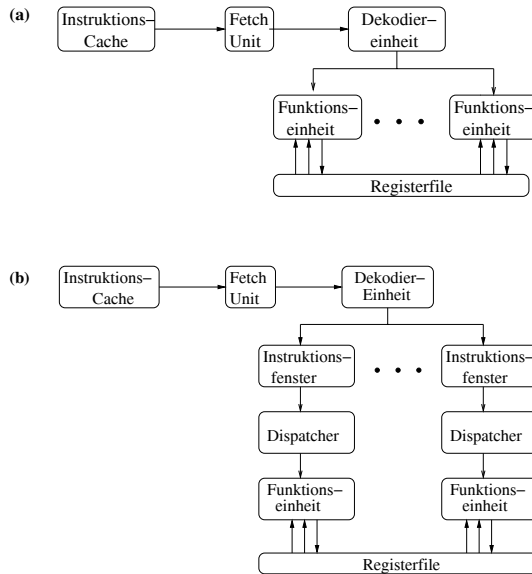


Abb. 2.2. Superskalare Prozessoren: (a) allgemeiner Aufbau, (b) Verwendung eines Instruktionsfensters.

Um eine flexible Abarbeitung der Instruktionen zu realisieren, wird ein zusätzliches **Instruktionsfenster** (engl. *instruction window*, *reservation station*) verwendet, in dem die Dekodiereinheit bereits dekodierte Instruktionen ablegt, ohne zu überprüfen, ob diese aufgrund von Datenabhängigkeiten evtl. noch nicht ausgeführt werden können. Vor der Weitergabe einer Instruktion aus dem Instruktionsfenster an eine Funktionseinheit (*Dispatch*) wird ein Abhängigkeitstest durchgeführt, der sicherstellt, dass nur solche Instruktionen ausgeführt werden, deren Operanden verfügbar sind. Das Instruktionsfenster kann für jede Funktionseinheit getrennt oder für alle zentral realisiert werden. Abbildung 2.2(b) zeigt die Prozessororganisation für getrennte In-

struktionsfenster. In der Praxis werden beide Möglichkeiten und Mischformen verwendet.

Im Instruktionsfenster abgelegte Instruktionen können nur dann ausgeführt werden, wenn ihre Operanden verfügbar sind. Werden die Operanden erst geladen, wenn die Instruktion in die Funktionseinheit transportiert wird (*dispatch bound*), kann die Verfügbarkeit der Operanden mit Hilfe einer Anzeigetafel (engl. *scoreboard*) kontrolliert werden. Die Anzeigetafel stellt für jedes Register ein zusätzliches Bit zur Verfügung. Das Bit eines Registers wird auf 0 gesetzt, wenn eine Instruktion an das Instruktionsfenster weitergeleitet wird, die ihr Ergebnis in dieses Register schreibt. Das Bit wird auf 1 zurückgesetzt, wenn die Instruktion ausgeführt und das Ergebnis in das Register geschrieben wurde. Eine Instruktion kann nur dann an eine Funktionseinheit weitergegeben werden, wenn die Anzeigenbits ihrer Operanden auf 1 gesetzt sind. Wenn die Operandenwerte zusammen mit der Instruktion in das Instruktionsfenster eingetragen werden (engl. *issue bound*), wird für den Fall, dass die Operandenwerte noch nicht verfügbar sind, ein Platzhalter in das Instruktionsfenster eingetragen, der durch den richtigen Wert ersetzt wird, sobald die Operanden verfügbar sind. Die Verfügbarkeit wird mit einer Anzeigetafel überprüft. Um die Operanden im Instruktionsfenster auf dem aktuellen Stand zu halten, muss nach Ausführung jeder Instruktion ein evtl. errechnetes Resultat zum Auffüllen der Platzhalter im Instruktionsfenster verwendet werden. Dazu müssen alle Einträge des Instruktionsfensters überprüft werden. Instruktionen mit eingetragenen Operandenwerten sind ausführbar und können an eine Funktionseinheit weitergegeben werden.

In jedem Verarbeitungsschritt werden im Fall, dass ein Instruktionsfenster mehrere Funktionseinheiten versorgt, so viele Instruktionen wie möglich an diese weitergegeben. Wenn dabei die Anzahl der ausführbaren Instruktionen die der verfügbaren Funktionseinheiten übersteigt, werden diejenigen Instruktionen ausgewählt, die *am längsten* im Instruktionsfenster liegen. Wird diese Reihenfolge jedoch strikt beachtet (engl. *in-order dispatch*), so kann eine im Instruktionsfenster abgelegte Instruktion, deren Operanden nicht verfügbar sind, die Ausführung von später im Instruktionsfenster abgelegten, aber bereits ausführbaren Instruktionen verhindern. Um diesen Effekt zu vermeiden, wird meist auch eine andere Reihenfolge erlaubt (engl. *out-of-order dispatch*), wenn dies zu einer besseren Auslastung der Funktionseinheiten führt.

Die meisten aktuellen Prozessoren stellen sicher, dass die Instruktionen in der Reihenfolge beendet werden, in der sie im Programm stehen, so dass das Vorhandensein mehrerer Funktionseinheiten keinen Einfluss auf die Fertigstellungsreihenfolge der Instruktionen hat. Dies wird meist durch den Einsatz eines Umordnungspuffers (engl. *reorder buffer*) erreicht, in den die an die Instruktionsfenster abgegebenen Instruktionen in der vom Programm vorgegebenen Reihenfolge eingetragen werden, wobei für jede Instruktion vermerkt wird, ob sie sich noch im Instruktionsfenster befindet, gerade ausgeführt wird oder bereits beendet wurde. Im letzten Fall liegt das Ergebnis der Instruk-

tion vor und kann in das Ergebnisregister geschrieben werden. Um die vom Programm vorgegebene Reihenfolge der Instruktionen einzuhalten, geschieht dies aber erst dann, wenn alle im Umordnungspuffer vorher stehenden Instruktionen ebenfalls beendet und ihre Ergebnisse in die zugehörigen Register geschrieben wurden. Nach der Aktualisierung des Ergebnisregisters werden die Instruktionen aus dem Umordnungspuffer entfernt. Für aktuelle Prozessoren können in einem Zyklus mehrere Ergebnisregister gleichzeitig beschrieben werden. Die zugehörigen Instruktionen werden aus dem Umordnungspuffer entfernt. Die Rate, mit der dies geschehen kann (engl. *retire rate*) stimmt bei den meisten Prozessoren mit der Rate überein, mit der Instruktionen an die Funktionseinheiten abgegeben werden können (engl. *issue rate*). In Abschnitt 2.9.1 beschreiben wir als Beispiel eines superskalaren Prozessors die Architektur des Intel Pentium 4.

Diese kurze Darstellung der prinzipiellen Funktionsweise superskalarer Prozessoren zeigt, dass ein nicht unerheblicher Aufwand für die Ausnutzung von Parallelität auf Hardwareebene nötig ist und dass diese Form der Parallelität begrenzt ist. Aktuelle Informationen zur Entwicklung von Prozessoren können über die *WWW Computer Architecture Home Page* der Universität Wisconsin (www.cs.wisc.edu/~arch/www) erhalten werden.

2.3 Klassifizierung von Parallelrechnern

Parallelrechner sind mittlerweile seit vielen Jahren im Einsatz, wobei bei der Realisierung dieser Rechner viele unterschiedliche Architekturansätze verfolgt wurden. Aus der Sicht des Programmierers ist es sinnvoll, die unterschiedlichen Architekturansätze grob zu klassifizieren. Zuerst wollen wir uns jedoch der Frage zuwenden, was man überhaupt unter einem Parallelrechner versteht. Häufig verwendet wird folgende Definition [11]:

Parallelrechner

Ein Parallelrechner ist eine Ansammlung von Berechnungseinheiten (Prozessoren), die durch koordinierte Zusammenarbeit große Probleme schnell lösen können.

Diese Definition ist bewusst vage gehalten, um die Vielzahl der entwickelten Parallelrechner zu erfassen und läßt daher auch viele z.T. wesentliche Details offen. Dazu gehören z.B. die Anzahl und Komplexität der Berechnungseinheiten, die Struktur der Verbindungen zwischen den Berechnungseinheiten, die Koordination der Arbeit der Berechnungseinheiten und die wesentlichen Eigenschaften der zu lösenden Probleme. Für eine genauere Untersuchung von Parallelrechnern ist eine Klassifizierung nach wichtigen Charakteristika nützlich. Wir beginnen mit der *Flynn'schen Klassifizierung*, die häufig als erste grobe Unterscheidung von Parallelrechnern verwendet wird. Es handelt sich hierbei um eine eher theoretische Klassifizierung, die auch historisch am Anfang der Parallelrechnerentwicklung stand. Als erste

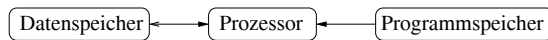
Einführung in wesentliche Unterschiede möglichen parallelen Berechnungsverhaltens und als Abgrenzung gegenüber dem sequentiellen Rechnen ist diese Klassifizierung aber durchaus sinnvoll.

Flynnschen Klassifizierung. Die Flynnsche Klassifizierung [47] charakterisiert Parallelrechner nach der Organisation der globalen Kontrolle und den resultierenden Daten- und Kontrollflüssen. Es werden vier Klassen von Rechnern unterschieden:

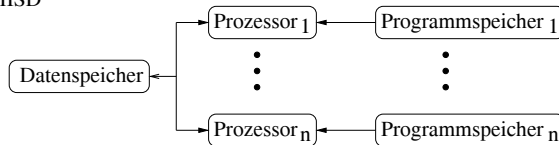
- a) SISD – **S**ingle **I**nstruction, **S**ingle **D**ata,
- b) MISD – **M**ultiple **I**nstruction, **S**ingle **D**ata,
- c) SIMD – **S**ingle **I**nstruction, **M**ultiple **D**ata und
- d) MIMD – **M**ultiple **I**nstruction, **M**ultiple **D**ata.

Jeder dieser Klassen ist ein idealisierter Modellrechner zugeordnet, vgl. Abbildung 2.3. Wir stellen im Folgenden die jeweiligen Modellrechner kurz vor.

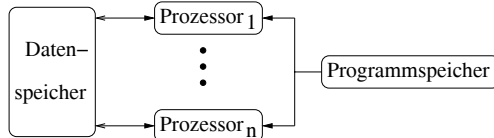
a) SISD



b) MISD



c) SIMD



d) MIMD

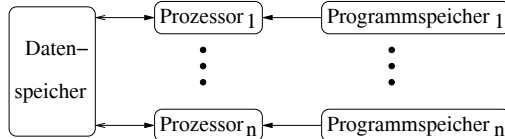


Abb. 2.3. Darstellung der Modellrechner des Flynnschen Klassifikationsschemas: a) SISD – Single Instruction, Single Data, b) MISD – Multiple Instruction, Single Data, c) SIMD – Single Instruction, Multiple Data und d) MIMD – Multiple Instruction, Multiple Data.

Der **SISD**-Modellrechner hat *eine* Verarbeitungseinheit (Prozessor), die Zugriff auf *einen* Datenspeicher und *einen* Programmspeicher hat. In jedem Verarbeitungsschritt lädt der Prozessor eine Instruktion aus dem Programmspeicher, dekodiert diese, lädt die angesprochenen Daten aus dem Datenspeicher in interne Register und wendet die von der Instruktion spezifizierte Operation auf die geladenen Daten an. Das Resultat der Operation wird in den Datenspeicher zurückgespeichert, wenn die Instruktion dies angibt. Damit entspricht der SISD-Modellrechner dem klassischen *von-Neumann*-Rechnermodell, das die Arbeitsweise aller sequentiellen Rechner beschreibt.

Der **MISD**-Modellrechner besteht aus *mehreren* Verarbeitungseinheiten, von denen jede Zugriff auf einen eigenen Programmspeicher hat. Es existiert jedoch nur ein gemeinsamer Zugriff auf den Datenspeicher. Ein Verarbeitungsschritt besteht darin, daß jeder Prozessor das *gleiche* Datum aus dem Datenspeicher erhält und eine Instruktion aus seinem Programmspeicher lädt. Diese evtl. unterschiedlichen Instruktionen werden dann von den verschiedenen Prozessoren parallel auf die erhaltene Kopie desselben Datums angewendet. Wenn ein Ergebnis berechnet wird und zurückgespeichert werden soll, muss jeder Prozessor den gleichen Wert zurückspeichern. Das zugrundeliegende Berechnungsmodell ist zu eingeschränkt, um eine praktische Relevanz zu besitzen. Es gibt daher auch keinen nach dem MISD-Prinzip arbeitenden Parallelrechner.

Der **SIMD**-Modellrechner besteht aus *mehreren* Verarbeitungseinheiten, von denen jede einen separaten Zugriff auf einen (gemeinsamen oder verteilten) Datenspeicher hat. Auf die Unterscheidung in gemeinsamen oder verteilten Datenspeicher werden wir in Abschnitt 2.4 näher eingehen. Es existiert jedoch nur *ein* Programmspeicher, auf den eine für die Steuerung des Kontrollflusses zuständige Kontrolleinheit zugreift. Ein Verarbeitungsschritt besteht darin, daß jeder Prozessor von der Kontrolleinheit die *gleiche* Instruktion aus dem Programmspeicher erhält und ein separates Datum aus dem Datenspeicher lädt. Die Instruktion wird dann synchron von den verschiedenen Prozessoren parallel auf die jeweiligen Daten angewendet und ein eventuell errechnetes Ergebnis wird in den Datenspeicher zurückgeschrieben.

Der **MIMD**-Modellrechner besteht aus mehreren Verarbeitungseinheiten, von denen jede einen separaten Zugriff auf einen (gemeinsamen oder verteilten) Datenspeicher und auf einen lokalen Programmspeicher hat. Ein Verarbeitungsschritt besteht darin, daß jeder Prozessor eine separate Instruktion aus seinem lokalen Programmspeicher und ein separates Datum aus dem Datenspeicher lädt, die Instruktion auf das Datum anwendet und ein eventuell errechnetes Ergebnis in den Datenspeicher zurückschreibt. Dabei können die Prozessoren asynchron zueinander arbeiten.

Der Vorteil der SIMD-Rechner gegenüber MIMD-Rechnern liegt darin, dass SIMD-Rechner einfacher zu programmieren sind, da es wegen der streng synchronen Abarbeitung nur einen Kontrollfluss gibt, so dass keine Synchronisation auf Programmebene erforderlich ist. Ein Nachteil der SIMD-Rechner

liegt darin, dass die verwendeten Berechnungseinheiten speziell für den Einsatz in SIMD-Rechnern entworfene Prozessoren sind und dass damit ein Anschluss an die Prozessorentwicklung schwierig oder teuer wird. Ein weiterer Nachteil liegt in dem eingeschränkten Berechnungsmodell, das eine streng synchrone Arbeitsweise der Prozessoren erfordert. Daher muss eine bedingte Anweisung der Form

```
if (b==0) c=a; else c = a/b;
```

in zwei Schritten ausgeführt werden. Im ersten Schritt setzen alle Prozessoren, deren lokaler Wert von **b** Null ist, den Wert von **c** auf den Wert von **a**. Im zweiten Schritt setzen alle Prozessoren, deren lokaler Wert von **b** nicht Null ist, den Wert von **c** auf $c = a/b$. Insbesondere der fehlende Anschluss an die Prozessorentwicklung hat dazu geführt, dass heutzutage keine SIMD-Parallelrechner mehr eingesetzt werden. Das SIMD-Konzept wird jedoch von manchen Prozessoren für die prozessorinterne Datenverarbeitung eingesetzt. Diese Prozessoren stellen SIMD-Instruktionen für eine schnelle Verarbeitung großer, gleichförmiger Datenmengen zur Verfügung. Dies wird z.B. im Cell-Prozessor genutzt, vgl. Abschnitt 2.9. Fast alle der heute verwendeten Parallelrechner arbeiten nach dem MIMD-Prinzip, da diese als Knoten normale Prozessoren verwenden können.

2.4 Speicherorganisation von Parallelrechnern

Fast alle der heute verwendeten Parallelrechner arbeiten nach dem MIMD-Prinzip, haben aber viele verschiedene Ausprägungen, so dass es sinnvoll ist, diese Klasse weiter zu unterteilen. Dabei ist eine Klassifizierung nach der Organisation des Speichers gebräuchlich, wobei zwischen der physikalischen Organisation des Speichers und der Sicht des Programmierers auf den Speicher unterschieden werden kann. Bei der physikalischen Organisation des Speichers unterscheidet man zwischen *Rechnern mit physikalisch gemeinsamem Speicher*, die auch *Multiprozessoren* genannt werden, und *Rechnern mit physikalisch verteiltem Speicher*, die auch *Multicomputer* genannt werden. Weiter sind *Rechner mit virtuell gemeinsamem Speicher* zu nennen, die als Hybridform angesehen werden können, vgl. auch Abbildung 2.4.

Bzgl. der Sicht des Programmierers wird zwischen *Rechnern mit verteiltem Adressraum* und *Rechnern mit gemeinsamem Adressraum* unterschieden. Die Sicht des Programmierers muss dabei nicht unbedingt mit der physikalischen Organisation des Rechners übereinstimmen, d.h. ein Rechner mit physikalisch verteiltem Speicher kann dem Programmierer durch eine geeignete Programmierungsumgebung als Rechner mit gemeinsamem Adressraum erscheinen und umgekehrt. Wir betrachten in diesem Abschnitt die physikalische Speicherorganisation von Parallelrechnern.

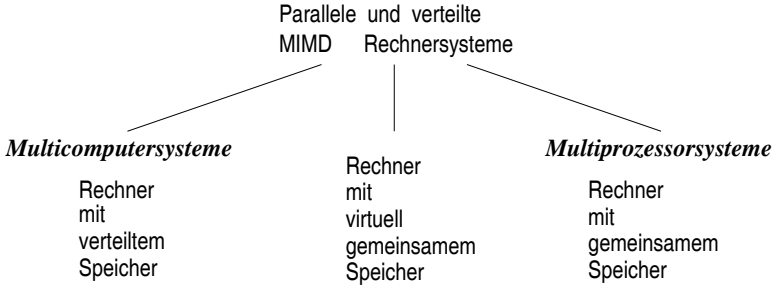


Abb. 2.4. Unterteilung der MIMD-Rechner bzgl. ihrer Speicherorganisation.

2.4.1 Rechner mit physikalisch verteiltem Speicher

Rechner mit physikalisch verteiltem Speicher (auch als **DMM** für engl. *distributed memory machine* bezeichnet) bestehen aus mehreren Verarbeitungseinheiten (Knoten) und einem Verbindungsnetzwerk, das die Knoten durch physikalische Leitungen verbindet, über die Daten übertragen werden können. Ein Knoten ist eine selbständige Einheit aus Prozessor, lokalem Speicher und evtl. I/O-Anschlüssen. Eine schematisierte Darstellung ist in Abbildung 2.5 a) wiedergegeben.

Die Daten eines Programmes werden in einem oder mehreren der lokalen Speicher abgelegt. Alle lokalen Speicher sind *privat*, d.h. nur der zugehörige Prozessor kann direkt auf die Daten zugreifen. Wenn ein Prozessor zur Verarbeitung seiner lokalen Daten auch Daten aus lokalen Speichern anderer Prozessoren benötigt, so müssen diese durch Nachrichtenaustausch über das Verbindungsnetzwerk bereitgestellt werden. Rechner mit verteiltem Speicher sind daher eng verbunden mit dem Programmiermodell der **Nachrichtenübertragung** (engl. *message-passing programming model*), das auf der Kommunikation zwischen kooperierenden sequentiellen Prozessen beruht und auf das wir in Kapitel 3 näher eingehen werden. Zwei miteinander kommunizierende Prozesse P_A und P_B auf verschiedenen Knoten A und B des Rechners setzen dabei zueinander komplementäre Sende- und Empfangsbefehle ab. Sendet P_A eine Nachricht an P_B , so führt P_A einen Sendebefehl aus, in dem die zu verschickende Nachricht und das Ziel P_B festgelegt wird. P_B führt einen Empfangsbefehl mit Angabe eines Empfangspuffers, in dem die Nachricht gespeichert werden soll, und des sendenden Prozesses P_A aus.

Die Architektur von Rechnern mit verteiltem Speicher hat im Laufe der Zeit eine Reihe von Entwicklungen erfahren, und zwar insbesondere im Hinblick auf das benutzte Verbindungsnetzwerk bzw. den Zusammenschluss von Netzwerk und Knoten. Frühe Multicomputer verwendeten als Verbindungsnetzwerk meist **Punkt-zu-Punkt-Verbindungen** zwischen Knoten. Ein Knoten ist dabei mit einer festen Menge von anderen Knoten durch physikalische Leitungen verbunden. Die Struktur des Verbindungsnetzwerkes kann als

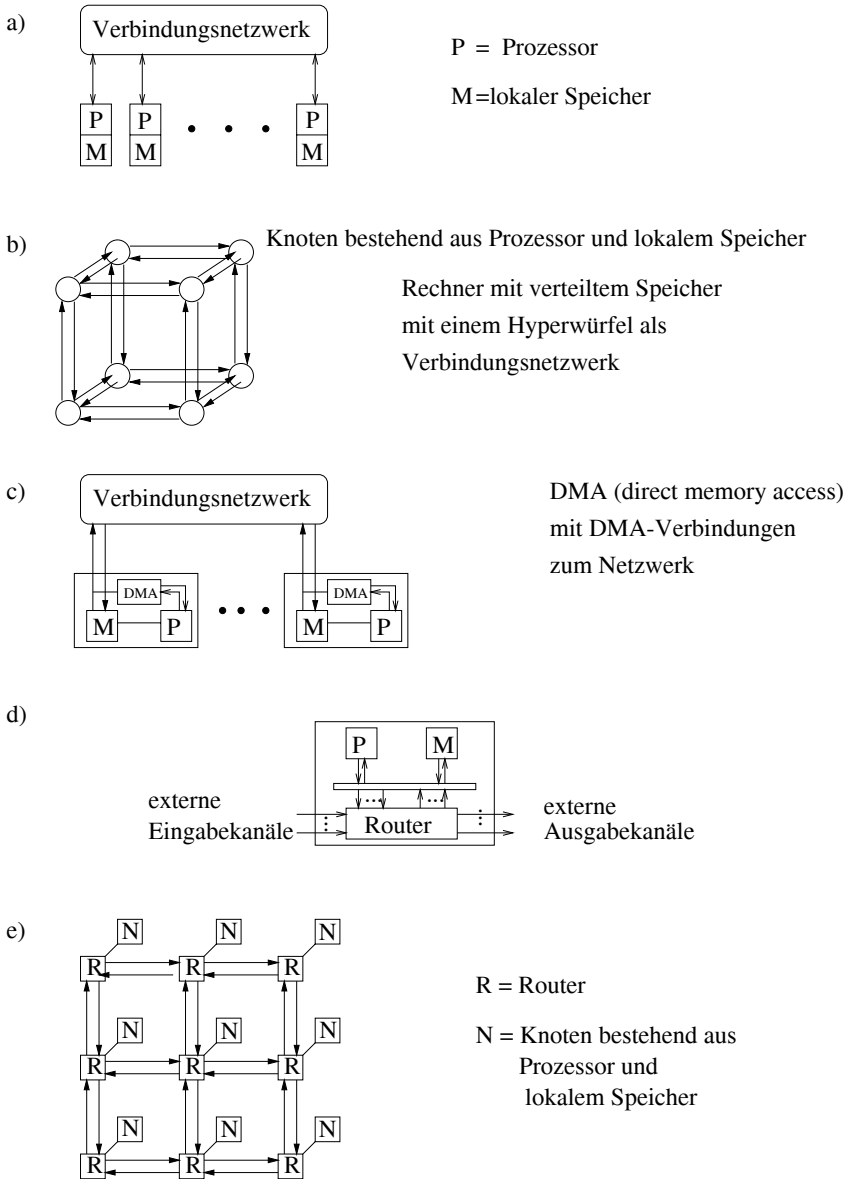


Abb. 2.5. Illustration zu Rechnern mit verteiltem Speicher a) Abstrakte Struktur, b) Rechner mit verteiltem Speicher und Hyperwürfel als Verbindungsstruktur, c) DMA (direct memory access), d) Prozessor-Speicher-Knoten mit Router und e) Verbindungsnetzwerk in Form eines Gitters zur Verbindung der Router der einzelnen Prozessor-Speicher-Knoten.

Graph dargestellt werden, dessen Knoten die Prozessoren und dessen Kanten die physikalischen Verbindungsleitungen, auch Links genannt, darstellen. Die Struktur des Graphen ist meist regelmäßig. Ein häufig verwendetes Netzwerk ist zum Beispiel der *Hyperwürfel*, der in Abbildung 2.5 b) zur Veranschaulichung verwendet wird. Beim Hyperwürfel und anderen Verbindungsnetzwerken mit Punkt-zu-Punkt-Verbindungen ist die Kommunikation durch die Gestalt des Netzwerkes vorgegeben, da Knoten nur mit ihren direkten Nachbarn kommunizieren können. Nur direkte Nachbarn können in Sende- und Empfangsoperationen als Absender bzw. Empfänger genannt werden. Kommunikation kann nur stattfinden, wenn benachbarte Knoten *gleichzeitig* auf den verbindenden Link schreiben bzw. von ihm lesen. Es werden zwar typischerweise Puffer bereitgestellt, in denen die Nachricht zwischengespeichert werden kann, diese sind aber relativ klein, so dass eine größere Nachricht nicht vollständig im Puffer abgelegt werden kann und so die Gleichzeitigkeit des Sendens und Empfangens notwendig wird. Dadurch ist die parallele Programmierung sehr stark an die verwendete Netzwerkstruktur gebunden und zum Erstellen von effizienten parallelen Programmen sollten parallele Algorithmen verwendet werden, die die vorhandenen Punkt-zu-Punkt-Verbindungen des vorliegenden Netzwerkes effizient ausnutzen [6, 101].

Das Hinzufügen eines speziellen **DMA-Controllers** (*DMA - direct memory access*) für den direkten Datentransfer zwischen lokalem Speicher und I/O-Anschluss ohne Einbeziehung des Prozessors entkoppelt die eigentliche Kommunikation vom Prozessor, so dass Sende- und Empfangsoperationen nicht genau zeitgleich stattfinden müssen, siehe Abbildung 2.5 c). Der Sender kann nun eine Kommunikation initiieren und dann weitere Arbeit ausführen, während der Sendebefehl unabhängig beendet wird. Beim Empfänger wird die Nachricht vom DMA-Controller empfangen und in einem speziell dafür vorgesehenen Speicherbereich abgelegt. Wird beim Empfänger eine zugehörige Empfangsoperation ausgeführt, so wird die Nachricht aus dem Zwischenspeicher entnommen und in dem im Empfangsbefehl angegebenen Empfangspuffer gespeichert. Die ausführbaren Kommunikationen sind aber immer noch an die Nachbarschaftsstruktur im Netzwerk gebunden. Kommunikation zwischen Knoten, die keine physikalischen Nachbarn sind, wird durch Software gesteuert, die die Nachrichten entlang aufeinanderfolgender Punkt-zu-Punkt-Verbindungen verschickt. Dadurch sind die Laufzeiten für die Kommunikation mit weiter entfernt liegenden Knoten erheblich größer als die Laufzeiten für Kommunikation mit physikalischen Nachbarn und die Verwendung von speziell für das Verbindungsnetzwerk entworfenen Algorithmen ist aus Effizienzgründen weiterhin empfehlenswert.

Moderne Multicomputer besitzen zu jedem Knoten einen **Hardware-Router**, siehe Abbildung 2.5 d). Der Knoten selbst ist mit dem Router verbunden. Die Router allein bilden das eigentliche Netzwerk, das hardwaremäßig die Kommunikation mit allen auch weiter entfernten Knoten übernimmt, siehe Abbildung 2.5 e). Die abstrakte Darstellung des Rechners mit

verteilem Speicher in Abbildung 2.5 a) wird also in dieser Variante mit Hardware-Routern am ehestens erreicht. Das hardwareunterstützte Routing verringert die Kommunikationszeit, da Nachrichten, die zu weiter entfernt liegenden Knoten geschickt werden, von Routern entlang eines ausgewählten Pfades weitergeleitet werden, so dass keine Mitarbeit der Prozessoren in den Knoten des Pfades erforderlich ist. Insbesondere unterscheiden sich die Zeiten für den Nachrichtenaustausch mit Nachbarknoten und mit entfernt gelegenen Knoten in dieser Variante nicht wesentlich. Da jeder physikalische I/O-Kanal des Hardware-Routers nur von einer Nachricht zu einem Zeitpunkt benutzt werden kann, werden Puffer am Ende von Eingabe- und Ausgabekanälen verwendet, um Nachrichten zwischenspeichern zu können. Zu den Aufgaben des Routers gehört die Ausführung von Pipelining bei der Nachrichtenübertragung und die Vermeidung von Deadlocks. Dies wird in Abschnitt 2.6.1 näher erläutert.

Rechner mit physikalisch verteiltem Speicher sind technisch relativ einfach zu realisieren, da die einzelnen Knoten im Extremfall einfache Desktop-Rechner sein können, die mit einem schnellen Netzwerk miteinander verbunden werden. Die Programmierung von Rechnern mit physikalisch verteiltem Speicher gilt als schwierig, da im natürlich zugehörigen Programmiermodell der Nachrichtenübertragung der Programmierer für die lokale Verfügbarkeit der Daten verantwortlich ist und alle Datentransfers zwischen den Knoten durch Sende- und Empfangsanweisungen explizit steuern muss. Üblicherweise dauert der Austausch von Daten zwischen Prozessoren durch Sende- und Empfangsoperationen wesentlich länger als ein Zugriff eines Prozessors auf seinen lokalen Speicher. Je nach verwendetem Verbindungsnetzwerk und verwendeter Kommunikationsbibliothek kann durchaus ein Faktor von 100 und mehr auftreten. Die Platzierung der Daten kann daher die Laufzeit eines Programmes entscheidend beeinflussen. Sie sollte so erfolgen, dass die Anzahl der Kommunikationsoperationen und die Größe der zwischen den Prozessoren verschickten Datenblöcke möglichst klein ist.

Der Aufbau eines Rechners mit verteiltem Speicher gleicht in vielem einem Netzwerk oder Cluster von Workstations (auch als NOW für *network of workstations* oder COW für *cluster of workstations* bezeichnet), in dem vollkommen eigenständige Computer durch ein lokales Netzwerk (LAN für engl. *local area network*) miteinander verbunden sind. Ein wesentlicher Unterschied besteht darin, dass das Verbindungsnetzwerk eines Parallelrechners üblicherweise eine wesentlich höhere Übertragungskapazität bereitstellt, so dass dadurch ein effizienterer Nachrichtenaustausch über das Netzwerk ermöglicht wird.

Ein **Cluster** ist eine parallele Plattform, die in ihrer Gesamtheit aus einer Menge von vollständigen, miteinander durch ein Kommunikationsnetzwerk verbundenen Rechnern besteht und das in seiner Gesamtheit als *ein einziger Rechner* angesprochen und benutzt wird. Von außen gesehen sind die einzelnen Komponenten eines Clusters anonym und gegenseitig aus-

tauschbar. Der Popularitätsgewinn des Clusters als parallele Plattform auch für die Anwendungsprogrammierung begründet sich in der Entwicklung von standardmäßiger Hochgeschwindigkeitskommunikation, wie z.B. FCS (*Fibre Channel Standard*), ATM (*Asynchronous Transfer Mode*), SCI (*Scalable Coherent Interconnect*), switched Gigabit Ethernet, Myrinet oder Infiniband, vgl. [124, 75, 121]. Ein natürliches Programmiermodell ist das Message-Passing-Modell, das durch Kommunikationsbibliotheken wie MPI und PVM unterstützt wird, siehe Kapitel 5. Diese Bibliotheken basieren z.T. auf Standardprotokollen wie TCP/IP [98, 123].

Von den *verteilten Systemen* (die wir hier nicht näher behandeln werden) unterscheiden sich Clustersysteme dadurch, dass sie eine geringere Anzahl von Knoten (also einzelnen Rechnern) enthalten, Knoten *nicht* individuell angesprochen werden und oft das gleiche Betriebssystem auf allen Knoten benutzt wird. Clustersysteme können mit Hilfe spezieller Middleware-Software zu Gridsystemen zusammengeschlossen werden. Diese erlauben ein koordiniertes Zusammenarbeiten über Cluster Grenzen hinweg. Die genaue Steuerung der Abarbeitung der Anwendungsprogramme wird von der Middleware-Software übernommen.

2.4.2 Rechner mit physikalisch gemeinsamem Speicher

Ein Rechner mit physikalisch gemeinsamem Speicher (auch als **SMM** für engl. *shared memory machine* bezeichnet) besteht aus mehreren Prozessoren oder Prozessorkernen, einem **gemeinsamen** oder **globalen** Speicher und einem Verbindungsnetzwerk, das Prozessoren und globalen Speicher durch physikalische Leitungen verbindet, über die Daten in den gemeinsamen Speicher geschrieben bzw. aus ihm gelesen werden können. Die Prozessorkerne eines Multicore-Prozessors greifen auf einen gemeinsamen Speicher zu und bilden daher eine Rechnerplattform mit physikalisch gemeinsamem Speicher, vgl. Abschnitt 2.8. Der gemeinsame Speicher setzt sich meist aus einzelnen Speichermodulen zusammen, die gemeinsam einen einheitlichen Adressraum darstellen, auf den alle Prozessoren gleichermaßen lesend (*load*) und schreibend (*store*) zugreifen können. Eine abstrakte Darstellung zeigt Abbildung 2.6.

Prinzipiell kann durch entsprechende Softwareunterstützung jedes parallele Programmiermodell auf Rechnern mit gemeinsamem Speicher unterstützt werden. Ein natürlicherweise geeignetes paralleles Programmiermodell ist die Verwendung **gemeinsamer Variablen** (engl. *shared variables*). Hierbei wird die Kommunikation und Kooperation der parallel arbeitenden Prozessoren über den gemeinsamen Speicher realisiert, indem Variablen von einem Prozessor beschrieben und von einem anderen Prozessor gelesen werden. *Gleichzeitiges* unkoordiniertes Schreiben verschiedener Prozessoren auf dieselbe Variable stellt in diesem Modell eine Operation dar, die zu nicht vorhersagbaren Ergebnissen führen kann. Für die Vermeidung dieses sogenannten *Schreibkon-*

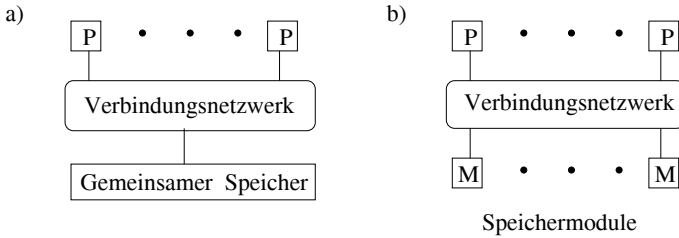


Abb. 2.6. Illustration eines Rechners mit gemeinsamem Speicher a) Abstrakte Sicht und b) Realisierung des gemeinsamen Speichers mit Speichermodulen.

fliktes gibt es unterschiedliche Ansätze, die in den Kapiteln 3 und 6 besprochen werden.

Für den Programmierer bietet ein Rechnermodell mit gemeinsamem Speicher große Vorteile gegenüber einem Modell mit verteiltem Speicher, weil die Kommunikation über den gemeinsamen Speicher einfacher zu programmieren ist und weil der gemeinsame Speicher eine gute Speicherausnutzung ermöglicht, da ein Replizieren von Daten nicht notwendig ist. Für die Hardware-Hersteller stellt die Realisierung von Rechnern mit gemeinsamem Speicher aber eine größere Herausforderung dar, da ein Verbindungsnetzwerk mit einer hohen Übertragungskapazität eingesetzt werden muss, um jedem Prozessor schnellen Zugriff auf den globalen Speicher zu geben, wenn nicht das Verbindungsnetzwerk zum Engpass der effizienten Ausführung werden soll. Die Erweiterbarkeit auf eine große Anzahl von Prozessoren ist daher oft schwieriger zu realisieren als für Rechner mit physikalisch verteiltem Speicher. Rechner mit gemeinsamem Speicher arbeiten aus diesem Grund meist mit einer geringen Anzahl von Prozessoren.

Eine spezielle Variante von Rechnern mit gemeinsamem Speicher sind die **symmetrischen Multiprozessoren** oder **SMP** (*symmetric multiprocessor*) [31]. SMP-Maschinen bestehen üblicherweise aus einer kleinen Anzahl von Prozessoren, die oft über einen zentralen Bus miteinander verbunden sind. Jeder Prozessor hat über den Bus Zugriff auf den gemeinsamen Speicher und die angeschlossenen I/O-Geräte. Es gibt keine zusätzlichen privaten Speicher für Prozessoren oder spezielle I/O-Prozessoren. Lokale Caches für Prozessoren sind aber üblich. Das Wort *symmetrisch* in der Bezeichnung SMP bezieht sich auf die Prozessoren und bedeutet, dass alle Prozessoren die gleiche Funktionalität und die gleiche Sicht auf das Gesamtsystem haben, d.h. insbesondere, dass die Dauer eines Zugriffs auf den gemeinsamen Speicher für jeden Prozessor unabhängig von der zugegriffenen Speicheradresse gleich lange dauert. In diesem Sinne ist jeder aus mehreren Prozessorkernen bestehende Multicore-Prozessor ein SMP-System. Wenn sich die zugegriffene Speicheradresse im lokalen Cache eines Prozessors befindet, findet der Zugriff entsprechend schneller statt. Die Zugriffszeit auf seinen lokalen Ca-

che ist für jeden Prozessor gleich. SMP-Rechner werden üblicherweise mit einer kleinen Anzahl von Prozessoren betrieben, weil der zentrale Bus nur eine konstante Bandbreite zur Verfügung stellt, aber die Speicherzugriffe aller Prozessoren nacheinander über den Bus laufen müssen. Wenn zu viele Prozessoren an den Bus angeschlossen sind, steigt die Gefahr von Kollisionen bei Speicherzugriffen und damit die Gefahr einer Verlangsamung der Verarbeitungsgeschwindigkeit der Prozessoren. Zum Teil kann dieser Effekt durch den Einsatz von Caches und geeigneten Cache-Kohärenzprotokollen abgemildert werden, vgl. Abschnitt 2.7.2. Die maximale Anzahl von Prozessoren liegt für busbasierte SMP-Rechner meist bei 32 oder 64 Prozessoren.

Das Vorhandensein mehrerer Prozessoren ist bei der Programmierung von SMP-Systemen prinzipiell sichtbar. Insbesondere das Betriebssystem muss die verschiedenen Prozessoren explizit ansprechen. Ein geeignetes paralleles Programmiermodell ist das *Thread-Programmiermodell*, wobei zwischen *Betriebssystem-Threads* (engl. *kernel threads*), die vom Betriebssystem erzeugt und verwaltet werden, und Benutzer-Threads (engl. *user threads*), die vom Programm erzeugt und verwaltet werden, unterschieden werden kann, siehe Abschnitt 6.1. Für Anwendungsprogramme kann das Vorhandensein mehrerer Prozessoren durch das Betriebssystem verborgen werden, d.h. Anwenderprogramme können normale sequentielle Programme sein, die vom Betriebssystem auf einzelne Prozessoren abgebildet werden. Die Auslastung aller Prozessoren wird dadurch erreicht, dass verschiedene Programme evtl. verschiedener Benutzer zur gleichen Zeit auf unterschiedlichen Prozessoren laufen. Bei SMP-Rechnern handelt es sich um einen Quasi-Industrie-Standard, der bereits vor über 25 Jahren zum Einsatz kam. Der typische Einsatzbereich liegt im Serverbereich.

Allgemein können Rechner mit gemeinsamem Speicher in vielerlei Varianten realisiert werden. Dies umfasst die Verwendung eines nicht-busbasierten Verbindungsnetzwerkes, das Vorhandensein mehrerer I/O-Anschlüsse oder einen zusätzlichen, nur vom jeweiligen Prozessor zugreifbaren, privaten Speicher. Es können also z.B. vollständige Knoten mit Prozessor, I/O und lokalem Speicher auftreten, wobei aber die Größe der lokalen Speicher nicht für ein eigenständiges Arbeiten ausreichen würde und typischerweise kein vollständiges Betriebssystem pro Knoten vorhanden ist, siehe z.B. [31, 124] für einen Überblick.

SMP-Systeme können zu größeren Parallelrechnern zusammengesetzt werden, indem ein Verbindungsnetzwerk eingesetzt wird, das den Austausch von Nachrichten zwischen Prozessoren verschiedener SMP-Maschinen erlaubt. Alternativ können Rechner mit gemeinsamem Speicher hierarchisch zu größeren Clustern zusammengesetzt werden, was z.B. zu Hierarchien von Speichern führt, vgl. Abschnitt 2.7. Durch Einsatz geeigneter Kohärenzprotokolle kann wieder ein logisch gemeinsamer Adressraum gebildet werden, d.h. jeder Prozessor kann jede Speicherzelle direkt adressieren, auch wenn sie im Speicher eines anderen SMP-Systems liegt. Da die Speichermodule physi-

kalisch getrennt sind und der gemeinsame Adressraum durch den Einsatz eines Kohärenzprotokolls realisiert wird, spricht man auch von Rechnern mit **virtuell-gemeinsamem Speicher** (engl. *virtual shared memory*).

Dadurch, dass der Zugriff auf die gemeinsamen Daten tatsächlich physikalisch verteilt auf lokale, gruppenlokale oder globale Speicher erfolgt, unterscheiden sich die Speicherzugriffe zwar (aus der Sicht des Programmiers) nur in der mitgegebenen Speicheradresse, die Zugriffe können aber in Abhängigkeit von der Speicheradresse zu *unterschiedlichen Speicherzugriffszeiten* führen. Der Zugriff eines Prozessors auf gemeinsame Variablen, die in dem ihm physikalisch am nächsten liegenden Speichermodul abgespeichert sind, wird schneller ausgeführt als Zugriffe auf gemeinsame Variablen mit Adressen, die einem physikalisch weiter entfernt liegenden Speicher zugeordnet sind. Zur Unterscheidung dieses für die Ausführungszeit wichtigen Phänomens der Speicherzugriffszeit wurden die Begriffe **UMA-System** (**U**niform **M**emory **A**ccess) und **NUMA-System** (**N**on-**U**niform **M**emory **A**ccess) eingeführt. UMA-Systeme weisen für alle Speicherzugriffe eine einheitliche Zugriffszeit auf. Bei NUMA-Systemen hängt die Speicherzugriffszeit von der relativen Speicherstelle einer Variablen zum zugreifenden Prozessor ab, vgl. Abbildung 2.7.

2.4.3 Reduktion der Speicherzugriffszeiten

Allgemein stellt die Speicherzugriffszeit eine kritische Größe beim Entwurf von Rechnern dar. Dies gilt insbesondere auch für Rechner mit gemeinsamem Speicher. Die technologische Entwicklung der letzten Jahre führte zu erheblicher Leistungssteigerung bei den Prozessoren. Die Speicherkapazität stieg in der gleichen Größenordnung. Die Steigerung der Speicherzugriffszeiten fiel jedoch geringer aus [31]. So stieg zwischen 1980 und 1998 die Leistung der Integer-Operationen von Mikroprozessoren durchschnittlich um 55 % pro Jahr und bei Floating-Point-Operationen um 75 % pro Jahr, vgl. Abschnitt 2.1. Im gleichen Zeitraum stieg die Speicherkapazität von DRAM-Chips, die zum Aufbau des Hauptspeichers verwendet werden, um ca. 60 % pro Jahr. Die Zugriffszeit verbesserte sich dagegen nur um 25 % pro Jahr. Da die Entwicklung der Speicherzugriffszeit nicht mit der Entwicklung der Rechenleistung Schritt halten konnte, stellt der Speicherzugriff den wesentlichen Engpass bei der Erzielung von hohen Rechenleistungen dar. Auch die Leistungsfähigkeit von Rechnern mit gemeinsamem Speicher hängt somit wesentlich davon ab, wie Speicherzugriffe gestaltet bzw. Speicherzugriffszeiten verringert werden können.

Zur Verhinderung großer Verzögerungszeiten beim Zugriff auf den lokalen Speicher werden im wesentlichen zwei Ansätze verfolgt [11]: die Simulation von **virtuellen Prozessoren** durch jeden physikalischen Prozessor (Multithreading) und der Einsatz von **lokalen Caches** zur Zwischenspeicherung von häufig benutzten Werten.

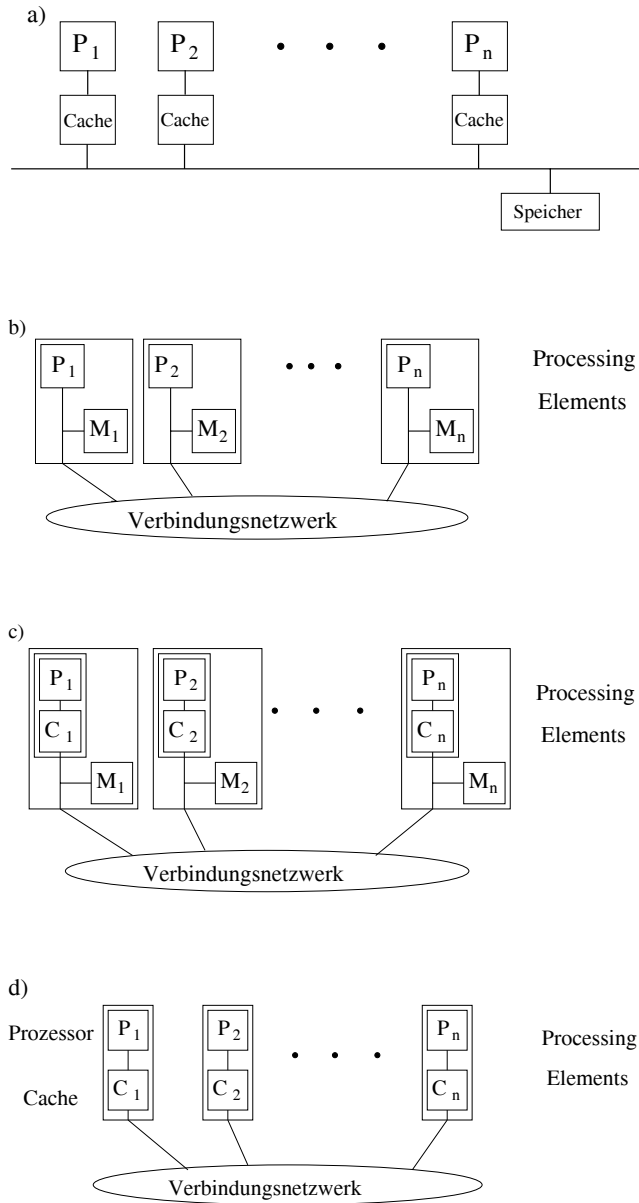


Abb. 2.7. Illustration der Architektur von Rechnern mit gemeinsamem Speicher:
a) SMP – symmetrische Multiprozessoren, b) NUMA – non-uniform memory access,
c) CC-NUMA – cache coherent NUMA und d) COMA – cache only memory access.

Multithreading. Die Idee des verschränkten Multithreading (engl. *interleaved multithreading*) besteht darin, die Latenz der Speicherzugriffe dadurch zu verbergen, dass jeder physikalische Prozessor eine feste Anzahl v von virtuellen Prozessoren simuliert. Für jeden zu simulierenden virtuellen Prozessor enthält ein physikalischer Prozessor einen eigenen Programmzähler und üblicherweise auch einen eigenen Registersatz. Nach jeder Ausführung eines Maschinenbefehls findet ein impliziter Kontextwechsel zum nächsten virtuellen Prozessor statt, d.h. die virtuellen Prozessoren eines physikalischen Prozessors werden von diesem pipelineartig reihum simuliert. Die Anzahl der von einem physikalischen Prozessor simulierten virtuellen Prozessoren wird so gewählt, dass die Zeit zwischen der Ausführung aufeinanderfolgender Maschinenbefehle eines virtuellen Prozessors ausreicht, evtl. benötigte Daten aus dem globalen Speicher zu laden, d.h. die Verzögerungszeit des Netzwerkes wird durch die Ausführung von Maschinenbefehlen anderer virtueller Prozessoren verdeckt. Der Pipelining-Ansatz reduziert also nicht die Menge der über das Netzwerk laufenden Daten, sondern bewirkt nur, dass ein virtueller Prozessor die von ihm aus dem Speicher angeforderten Daten erst dann zu benutzen versucht, wenn diese auch eingetroffen sind. Der Vorteil dieses Ansatzes liegt darin, dass aus der Sicht eines virtuellen Prozessors die Verzögerungszeit des Netzwerkes nicht sichtbar ist. Damit kann für die Programmierung ein PRAM-ähnliches Programmiermodell realisiert werden, das für den Programmierer sehr einfach zu verwenden ist, vgl. Abschnitt 4.5.1. Der Nachteil liegt darin, dass für die Programmierung die vergleichsweise hohe Gesamtzahl der virtuellen Prozessoren zugrundegelegt werden muss. Daher muss der zu implementierende Algorithmus ein ausreichend großes Potential an Parallelität besitzen, damit alle virtuellen Prozessoren sinnvoll beschäftigt werden können. Ein weiterer Nachteil besteht darin, dass die verwendeten physikalischen Prozessoren speziell für den Einsatz in den jeweiligen Parallelrechnern entworfen werden müssen, da übliche Mikroprozessoren die erforderlichen schnellen Kontextwechsel nicht zur Verfügung stellen. Beispiele für Rechner, die nach dem Pipelining-Ansatz arbeiteten, waren die Denelcor HEP (*Heterogeneous Element Processor*) mit 16 physikalischen Prozessoren [149], von denen jeder bis zu 128 Threads unterstützte, der NYU Ultracomputer [63], die SB-PRAM [1] mit 64 physikalischen und 2048 virtuellen Prozessoren, und die Tera MTA [31, 85].

Ein alternativer Ansatz zum verschränkten Multithreading ist das blockorientierte Multithreading [31]. Bei diesem Ansatz besteht das auszuführende Programm aus einer Menge von Threads, die auf den zur Verfügung stehenden Prozessoren ausgeführt werden. Der Unterschied zum verschränkten Multithreading liegt darin, dass nur dann ein Kontextwechsel zum nächsten Thread ausgeführt wird, wenn der gerade aktive Thread einen Speicherzugriff ausführt, der nicht über den lokalen Speicher des Prozessors befriedigt werden kann. Dieser Ansatz wurde z.B. von der MIT Alewife verwendet [2, 31].

Einsatz von Caches. *Caches* oder *Cache-Speicher* sind kleine schnelle Speicher, die zwischen Prozessor und Hauptspeicher geschaltet werden. Im Gegensatz zum Pipelining-Ansatz versucht man beim Einsatz von Caches, die Menge der über das Netzwerk transportierten Daten zu reduzieren, indem man jeden (physikalischen) Prozessor mit einem Cache ausstattet, in dem, von der Hardware gesteuert, häufig zugriffene Daten gehalten werden. Damit braucht ein Zugriff auf diese Daten nicht über das Netzwerk zu laufen, sondern kann durch einen lokalen Zugriff realisiert werden. Technisch wird dies so umgesetzt, dass jeder aus dem globalen Speicher geladene Wert automatisch im Cache des zugreifenden Prozessors zwischengespeichert wird. Dabei wird vor jedem Zugriff auf den globalen Speicher untersucht, ob die angeforderte Speicherzelle bereits im Cache enthalten ist. Wenn dies der Fall ist, wird der Wert aus dem Cache geladen und der Zugriff auf den globalen Speicher entfällt. Dies führt dazu, dass Speicherzugriffe, die über den Cache erfolgen können, wesentlich schneller sind als Speicherzugriffe, deren Werte noch nicht im Cache liegen.

Cache-Speicher werden für fast alle Rechnertypen einschließlich Ein-Prozessor-Systeme, SMPs und Parallelrechner mit verschiedenen Speicherorganisationen zur Verringerung der Speicherzugriffszeit eingesetzt. Bei Multiprozessorssystemen mit lokalen Caches, bei denen jeder Prozessor auf den gesamten globalen Speicher Zugriff hat, tritt das Problem der Aufrechterhaltung der *Cache-Kohärenz* (engl. *cache coherence*) auf, d.h. es kann die Situation eintreten, dass verschiedene Kopien einer gemeinsamen Variablen in den lokalen Caches einzelner Prozessoren geladen und möglicherweise mit anderen Werten belegt sein können. Die Cache-Kohärenz würde verletzt, wenn ein Prozessor p den Wert einer Speicherzelle in seinem lokalen Cache ändern würde, ohne diesen Wert in den globalen Speicher zurückzuschreiben. Wenn ein anderer Prozessor q danach diese Speicherzelle laden würde, würde er fälschlicherweise den noch nicht aktualisierten Wert benutzen. Aber selbst ein Zurückschreiben des Wertes durch p in den globalen Speicher ist nicht ausreichend, wenn q die gleiche Speicherzelle in seinem lokalen Cache hat. In diesem Fall muss der Wert der Speicherzelle auch im lokalen Cache von q aktualisiert werden.

Zur korrekten Realisierung der Programmierung mit gemeinsamen Variablen muss sichergestellt sein, dass alle Prozessoren den *aktuellen* Wert einer Variable erhalten, wenn sie auf diese zugreifen. Zur Aufrechterhaltung der Cache-Kohärenz gibt es mehrere Ansätze, von denen wir in Abschnitt 2.7.2 einige genauer vorstellen wollen. Eine ausführliche Beschreibung ist auch in [11] und [71] zu finden. Da die Behandlung der Cache-Kohärenzfrage auf das verwendete Berechnungsmodell wesentlichen Einfluss hat, werden Multiprozessoren entsprechend weiter untergliedert.

CC-NUMA-Rechner (Cache Coherent NUMA) sind Rechner mit gemeinsamem Speicher, bei denen Cache-Kohärenz sichergestellt ist. Die Caches der Prozessoren können so nicht nur Daten des lokalen Speichers des

Prozessoren, sondern auch globale Daten des gemeinsamen Speichers aufnehmen. Die Verwendung des Begriffes CC-NUMA macht deutlich, dass die Bezeichnung NUMA einem Bedeutungswandel unterworfen ist und mittlerweile zur Klassifizierung von Hard- und Software in Systemen mit Caches benutzt wird. Multiprozessoren, die keine Cache-Kohärenz aufweisen (manchmal auch *NC-NUMA-Rechner* für engl. *Non-Coherent NUMA* genannt), können nur Daten der lokalen Speicher oder Variablen, die nur gelesen werden können, in den Cache laden. Eine Loslösung von der statischen Speicherallokation des gemeinsamen Speichers stellen die **COMA-Rechner** (für engl. **C**ache **O**nly **M**emory **A**ccess) dar, deren Speicher nur noch aus Cache-Speicher besteht, siehe [31] für eine ausführlichere Behandlung. Es gibt also weder globalen Speicher noch verteilten Speicher mit virtuell gemeinsamem Adressraum. Daten sind auf die lokalen Cache-Speicher verteilt und werden gemäß vorhandener Cache-Kohärenz-Protokolle angesprochen. Eine Realisierung waren die KSR1 und KSR2 [139].

2.5 Verbindungsnetzwerke

Eine physikalische Verbindung der einzelnen Komponenten eines parallelen Systems wird durch das **Verbindungsnetzwerk** (engl. *interconnection network*) hergestellt. Neben den Kontroll- und Datenflüssen und der Organisation des Speichers kann das eingesetzte Verbindungsnetzwerk zur Klassifikation paralleler Systeme verwendet werden. Intern besteht ein Verbindungsnetzwerk aus Leitungen und Schaltern, die meist in regelmäßiger Weise angeordnet sind. In Multicomputersystemen werden über das Verbindungsnetzwerk verschiedene Prozessoren bzw. Verarbeitungseinheiten miteinander verbunden. Interaktionen zwischen verschiedenen Prozessoren, die zur Koordination der gemeinsamen Bearbeitung von Aufgaben notwendig sind und die entweder dem Austausch von Teilergebnissen oder der Synchronisation von Bearbeitungsströmen dienen, werden durch das Verschicken von Nachrichten, der sogenannten **Kommunikation**, über die Leitungen des Verbindungsnetzwerkes realisiert. In Multiprozessorsystemen werden die Prozessoren durch das Verbindungsnetzwerk mit den Speichermodulen verbunden, die **Speicherzugriffe** der Prozessoren erfolgen also über das Verbindungsnetzwerk.

Die Grundaufgabe eines Verbindungsnetzwerkes besteht in beiden Fällen darin, eine Nachricht, die Daten oder Speicheranforderungen enthält, von einem gegebenen Prozessor zu einem angegebenen Ziel zu transportieren. Dabei kann es sich um einen anderen Prozessor oder ein Speichermodul handeln. Die Anforderung an ein Verbindungsnetzwerk besteht darin, diese Kommunikationsaufgabe in möglichst geringer Zeit korrekt auszuführen, und zwar auch dann, wenn mehrere Nachrichten gleichzeitig übertragen werden sollen. Da die Nachrichtenübertragung bzw. der Speicherzugriff einen wesentlichen Teil der Bearbeitung einer Aufgabe auf einem parallelen System mit verteiltem oder gemeinsamem Speicher darstellt, ist das benutzte Verbindungsnetzwerk

ein wesentlicher Bestandteil des Designs eines parallelen Systems und kann großen Einfluß auf dessen Leistung haben. Gestaltungskriterien eines Verbindungsnetzwerkes sind

- die **Topologie**, die die Form der Verschaltung der einzelnen Prozessoren bzw. Speichereinheiten beschreibt, und
- die **Routingtechnik**, die die Nachrichtenübertragung zwischen den einzelnen Prozessoren bzw. zwischen den Prozessoren und den Speichermodulen realisiert.

Topologie. Die **Topologie** eines Verbindungsnetzwerkes beschreibt die geometrische Struktur, mit der dessen Leitungen und Schalter angeordnet sind, um Prozessoren und Speichermodule miteinander zu verbinden. Diese Verbindungsstruktur wird oft als Graph beschrieben, in dem Schalter, Prozessoren oder Speichermodule die Knoten darstellen und die Verbindungsleitungen durch Kanten repräsentiert werden. Unterschieden wird zwischen *statischen* und *dynamischen* Verbindungsnetzwerken. **Statische Verbindungsnetzwerke** verbinden Prozessoren *direkt* durch eine zwischen den Prozessoren liegende physikalische Leitung miteinander und werden daher auch **direkte** Verbindungsnetzwerke oder **Punkt-zu-Punkt-Verbindungsnetze** genannt. Die Anzahl der Verbindungen für einen Knoten variiert zwischen einer minimalen Anzahl von einem Nachbarn in einem Stern-Netzwerk und einer maximalen Anzahl von Nachbarn in einem vollständig verbundenen Graphen, vgl. Abschnitte 2.5.1 und 2.5.4. Statische Netzwerke werden im Wesentlichen für Systeme mit verteiltem Speicher eingesetzt, wobei ein Knoten jeweils aus einem Prozessor und einer zugehörigen Speichereinheit besteht. **Dynamische Verbindungsnetzwerke** verbinden Prozessoren und/oder Speichereinheiten *indirekt* über mehrere Leitungen und dazwischenliegende Schalter miteinander und werden daher auch als **indirekte** Verbindungsnetzwerke bezeichnet. Varianten sind *busbasierte* Netzwerke oder *schalterbasierte* Netzwerke (engl. *switching network*), bestehend aus Leitungen und dazwischenliegenden Schaltern (engl. *switches*). Eingesetzt werden dynamische Netzwerke sowohl für Systeme mit verteiltem Speicher als auch für Systeme mit gemeinsamem Speicher. Für letztere werden sie als Verbindung zwischen den Prozessoren und den Speichermodulen verwendet. Häufig werden auch hybride Netzwerktopologien benutzt.

Routingtechnik. Eine **Routingtechnik** beschreibt, *wie* und *entlang welchen Pfades* eine Nachricht über das Verbindungsnetzwerk von einem Sender zu einem festgelegten Ziel übertragen wird, wobei sich der Begriff des Pfades hier auf die Beschreibung des Verbindungsnetzwerkes als Graph bezieht. Die Routingtechnik setzt sich zusammen aus dem **Routing**, das mittels eines *Routingalgorithmus* einen Pfad vom sendenden Knoten zum empfangenden Knoten für die Nachrichtenübertragung auswählt und einer **Switching-Strategie**, die festlegt, wie eine Nachricht in Teilstücke unterteilt wird, wie einer Nachricht ein Routingpfad zugeordnet wird und wie eine Nachricht über

die auf dem Routingpfad liegenden Schalter oder Prozessoren weitergeleitet wird.

Die Kombination aus Routing-Algorithmus, Switching-Strategie und Topologie bestimmt wesentlich die Geschwindigkeit der zu realisierenden Kommunikation. Die nächsten Abschnitte 2.5.1 bis 2.5.4 enthalten einige gebräuchliche direkte und indirekte Topologien für Verbindungsnetzwerke. Spezielle Routing-Algorithmen und Varianten von Switching-Strategien stellen wir in den Abschnitten 2.6.1 bzw. 2.6.2 vor. Effiziente Verfahren zur Realisierung von Kommunikationsoperationen für verschiedene Verbindungsnetzwerke enthält Kapitel 4. Verbindungsnetzwerke und ihre Eigenschaften werden u.a. in [14, 31, 85, 65, 101, 146, 41] detailliert behandelt.

2.5.1 Bewertungskriterien für Netzwerke

In statischen Verbindungsnetzwerken sind die Verbindungen zwischen Schaltern oder Prozessoren fest angelegt. Ein solches Netzwerk kann durch einen Kommunikationsgraphen $G = (V, E)$ beschrieben werden, wobei V die Knotenmenge der zu verbindenden Prozessoren und E die Menge der direkten Verbindungen zwischen den Prozessoren bezeichnet, d.h. es ist $(u, v) \in E$, wenn es eine direkte Verbindung zwischen den Prozessoren $u \in V$ und $v \in V$ gibt. Da für die meisten parallelen Systeme das Verbindungsnetzwerk bidirektional ausgelegt ist, also in beide Richtungen der Verbindungsleitung eine Nachricht geschickt werden kann, wird G meist als ungerichteter Graph definiert. Soll eine Nachricht von einem Knoten u zu einem anderen Knoten v gesendet werden, zu dem es *keine* direkte Verbindungsleitung gibt, so muss ein Pfad von u nach v gewählt werden, der aus mehreren Verbindungsleitungen besteht, über die die Nachricht dann geschickt wird. Eine Folge von Knoten $(v_0, \dots, v_k)$ heißt *Pfad* der Länge k zwischen den Knoten v_0 und v_k , wenn $(v_i, v_{i+1}) \in E$ für $0 \leq i < k$. Als Verbindungsnetzwerke sind nur solche Netzwerke sinnvoll, für die es zwischen beliebigen Prozessoren $u, v \in V$ mindestens einen Pfad gibt.

Statische Verbindungsnetzwerke können anhand verschiedener Eigenschaften des zugrunde liegenden Graphen G bewertet werden. Neben der Anzahl der Knoten n werden folgende Eigenschaften betrachtet:

- Durchmesser,
- Grad,
- Bisektionsbandbreite,
- Knoten- und Kantenkonnektivität und
- Einbettung in andere Netzwerke.

Als **Durchmesser** $\delta(G)$ eines Netzwerkes G wird die maximale Distanz zwischen zwei beliebigen Prozessoren bezeichnet:

$$\delta(G) = \max_{u, v \in V} \min_{\substack{\varphi \text{ Pfad} \\ \text{von } u \text{ nach } v}} \{k \mid k \text{ ist Länge des Pfades } \varphi \text{ von } u \text{ nach } v\}.$$

Der Durchmesser ist ein Maß dafür, wie lange es dauern kann, bis eine von einem beliebigen Prozessor abgeschickte Nachricht bei einem beliebigen anderen Prozessor ankommt.

Der **Grad** $g(G)$ eines Netzwerkes G ist der maximale Grad eines Knotens des Netzwerkes, wobei der Grad eines Knotens der Anzahl der adjazenten, d.h. ein- bzw. auslaufenden, Kanten des Knotens entspricht:

$$g(G) = \max\{g(v) \mid g(v) \text{ Grad von } v \in V\}.$$

Die **Bisektionsbreite** bzw. **Bisektionsbandbreite** eines Netzwerkes G ist die minimale Anzahl von Kanten, die aus dem Netzwerk entfernt werden müssen, um das Netzwerk in zwei gleichgroße Teilnetzwerke zu zerlegen, d.h. in zwei Teilnetzwerke mit einer bis auf 1 gleichen Anzahl von Knoten. Die Bisektionsbandbreite $B(G)$ ist also definiert als

$$B(G) = \min_{\substack{U_1, U_2 \text{ Partition von } V \\ ||U_1| - |U_2|| \leq 1}} |\{(u, v) \in E \mid u \in U_1, v \in U_2\}|.$$

Bereits $B(G) + 1$ Nachrichten können das Netzwerk sättigen, falls diese zur gleichen Zeit über die entsprechenden Kanten übertragen werden sollen. Damit ist die Bisektionsbandbreite ein Maß für die Belastbarkeit des Netzwerkes bei der gleichzeitigen Übertragung von Nachrichten.

Die **Knotenkonnektivität** und **Kantenkonnektivität** sind verschiedene Beschreibungen des Zusammenhangs der Knoten des Netzwerkes. Der Zusammenhang hat Auswirkungen auf die Ausfallsicherheit des Netzwerkes. Die Knotenkonnektivität eines Netzwerkes G ist die minimale Anzahl von Knoten, die gelöscht werden müssen, um das Netzwerk zu unterbrechen, d.h. in zwei unverbundene Netzwerke (nicht unbedingt gleicher Größe) zu zerlegen. Für eine genauere Definition bezeichnen wir mit $G_{V \setminus M}$ den Restgraphen, der durch Löschen der Knoten von $M \subset V$ und aller zugehörigen Kanten entsteht. Es ist also $G_{V \setminus M} = (V \setminus M, E \cap ((V \setminus M) \times (V \setminus M)))$. Die Knotenkonnektivität $nc(G)$ von G ist damit definiert als

$$nc(G) = \min_{M \subset V} \{|M| \mid \text{es existieren } u, v \in V \setminus M, \text{ so dass es in } G_{V \setminus M} \text{ keinen Pfad von } u \text{ nach } v \text{ gibt}\}.$$

Analog bezeichnet die Kantenkonnektivität eines Netzwerkes G die minimale Anzahl von Kanten, die man löschen muss, damit das Netzwerk unterbrochen wird. Für eine beliebige Teilmenge $F \subset E$ bezeichne $G_{E \setminus F}$ den Restgraphen, der durch Löschen der Kanten von F entsteht, d.h. $G_{E \setminus F} = (V, E \setminus F)$. Die Kantenkonnektivität $ec(G)$ von G ist definiert durch

$$ec(G) = \min_{F \subset E} \{|F| \mid \text{es existieren } u, v \in V, \text{ so dass es in } G_{E \setminus F} \text{ keinen Pfad von } u \text{ nach } v \text{ gibt}\}.$$

Die Knoten- oder Kantenkonnektivität eines Verbindungsnetzwerkes ist ein Maß für die Anzahl der unabhängigen Wege, die zwei beliebige Prozessoren

u und v miteinander verbinden. Eine hohe Konnektivität sichert eine hohe Zuverlässigkeit bzw. Ausfallsicherheit des Netzwerkes, da viele Prozessoren bzw. Verbindungen ausfallen müssen, bevor das Netzwerk zerfällt. Eine Obergrenze für die Knoten- oder Kantenkonnektivität eines Netzwerkes bildet der kleinste Grad eines Knotens im Netzwerk, da ein Knoten dadurch vollständig von seinen Nachbarn separiert werden kann, dass alle seine Nachbarn bzw. alle Kanten zu diesen Nachbarn gelöscht werden. Man beachte, dass die Knotenkonnektivität eines Netzwerkes kleiner als seine Kantenkonnektivität sein kann, vgl. Abbildung 2.8.

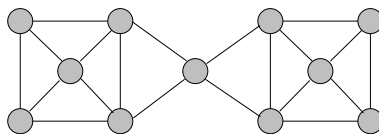


Abb. 2.8. Netzwerk mit Knotenkonnektivität 1, Kantenkonnektivität 2 und Grad 4. Der kleinste Grad eines Knotens ist 3.

Ein Maß für die Flexibilität eines Netzwerkes wird durch den Begriff der Einbettung bereitgestellt. Seien $G = (V, E)$ und $G' = (V', E')$ zwei Netzwerke. Eine **Einbettung** von G' in G ordnet jeden Knoten von G' einem Knoten von G so zu, dass unterschiedliche Knoten von G' auf unterschiedliche Knoten von G abgebildet werden und dass Kanten zwischen zwei Knoten in G' auch zwischen den zugeordneten Knoten in G existieren [14]. Eine Einbettung von G' in G wird beschrieben durch eine Funktion $\sigma : V' \rightarrow V$, für die gilt:

- Wenn $u \neq v$ für $u, v \in V'$ gilt, dann folgt $\sigma(u) \neq \sigma(v)$.
- Wenn $(u, v) \in E'$ gilt, dann folgt $(\sigma(u), \sigma(v)) \in E$.

Kann ein Netzwerk G' in ein Netzwerk G eingebettet werden, so besagt dies, dass G mindestens so flexibel ist wie Netzwerk G' , da ein Algorithmus, der die Nachbarschaftsbeziehungen in G' ausnutzt, durch eine Umnummerierung gemäß σ in einen Algorithmus auf G abgebildet werden kann, der auch in G Nachbarschaftsbeziehungen ausnutzt.

Anforderungen an ein Netzwerk. Das Netzwerk eines parallelen Systems sollte entsprechend den Anforderungen an dessen Architektur ausgewählt werden. Allgemeine Anforderungen an das Netzwerk im Sinne der eingeführten Eigenschaften der Topologie sind:

- ein kleiner Durchmesser für kleine Distanzen bei der Nachrichtenübertragung,
- ein kleiner Grad jedes Knotens zur Reduzierung des Hardwareaufwandes,
- eine hohe Bisektionsbandbreite zur Erreichung eines hohen Durchsatzes,
- eine hohe Konnektivität zur Erreichung hoher Zuverlässigkeit,

- die Möglichkeit der Einbettung von möglichst vielen anderen Netzwerken sowie
- eine einfache Erweiterbarkeit auf eine größere Anzahl von Prozessoren (Skalierbarkeit).

Da sich diese Anforderungen z.T. widersprechen, gibt es kein Netzwerk, das alle Anforderungen gleichzeitig erfüllt. Im Folgenden werden wir einige häufig verwendete direkte Verbindungsnetzwerke vorstellen. Die Topologien sind in Abbildung 2.9 dargestellt, die Eigenschaften sind in Tabelle 2.2 zusammengefasst.

2.5.2 Direkte Verbindungsnetzwerke

Die üblicherweise verwendeten direkten Verbindungsnetzwerke haben eine regelmäßige Struktur und sind daher durch einen regelmäßigen Aufbau des zugrunde liegenden Graphen $G = (V, E)$ gekennzeichnet. Bei der Beschreibung der Topologien wird die Anzahl der Knoten bzw. Prozessoren $n = |V|$ als Parameter benutzt, so dass die jeweilige Topologie kein einzelnes Netzwerk, sondern eine ganze Klasse von Netzwerken beschreibt.

- (a) Ein **vollständiger Graph** ist ein Netzwerk G , in dem jeder Knoten direkt mit jedem anderen Knoten verbunden ist, vgl. Abbildung 2.9(a). Dies ergibt einen Durchmesser $\delta(G) = 1$. Entsprechend gilt für den Grad $g(G) = n - 1$ und für die Knoten- bzw. Kantenkonnektivität $nc(G) = ec(G) = n - 1$, da die Verbindung zu einem Knoten durch Entfernen der $n - 1$ adjazenten Kanten unterbrochen werden kann. Die Bisektionsbandbreite ist $n^2/4$ für gerades n . Eine Einbettung in den vollständigen Graphen ist für alle anderen Netzwerke möglich. Die physikalische Realisierbarkeit ist wegen des hohen Knotengrades jedoch nur für eine kleine Anzahl n von Prozessoren gegeben.
- (b) In einem **linearen Feld** können die Knoten linear angeordnet werden, so dass zwischen benachbarten Prozessoren eine bidirektionale Verbindung besteht, vgl. Abbildung 2.9(b), d.h. es ist $V = \{v_1, \dots, v_n\}$ und $E = \{(v_i, v_{i+1}) | 1 \leq i < n\}$. Bedingt durch den Abstand von Knoten v_1 zu Knoten v_n ist der Durchmesser $\delta(G) = n - 1$. Die Konnektivität ist $nc(G) = ec(G) = 1$, da bereits durch den Ausfall eines Knotens oder einer Kante das Netzwerk unterbrochen wird. Der Grad ist $g(G) = 2$ und die Bisektionsbandbreite ist $B(G) = 1$. Eine Einbettung ist in fast alle hier aufgeführten Netzwerke mit Ausnahme des Baumes (siehe (h) dieser Auflistung und Abbildung 2.9 (h)) möglich. Da es nur genau eine Verbindung zwischen zwei Knoten gibt, ist keine Fehlertoleranz bzgl. der Übermittlung von Nachrichten gegeben.
- (c) In einem **Ring-Netzwerk** können die Knoten in Form eines Ringes angeordnet werden, d.h. zusätzlich zu den Kanten des linearen Feldes existiert eine zusätzliche bidirektionale Kante vom ersten Prozessor der linearen

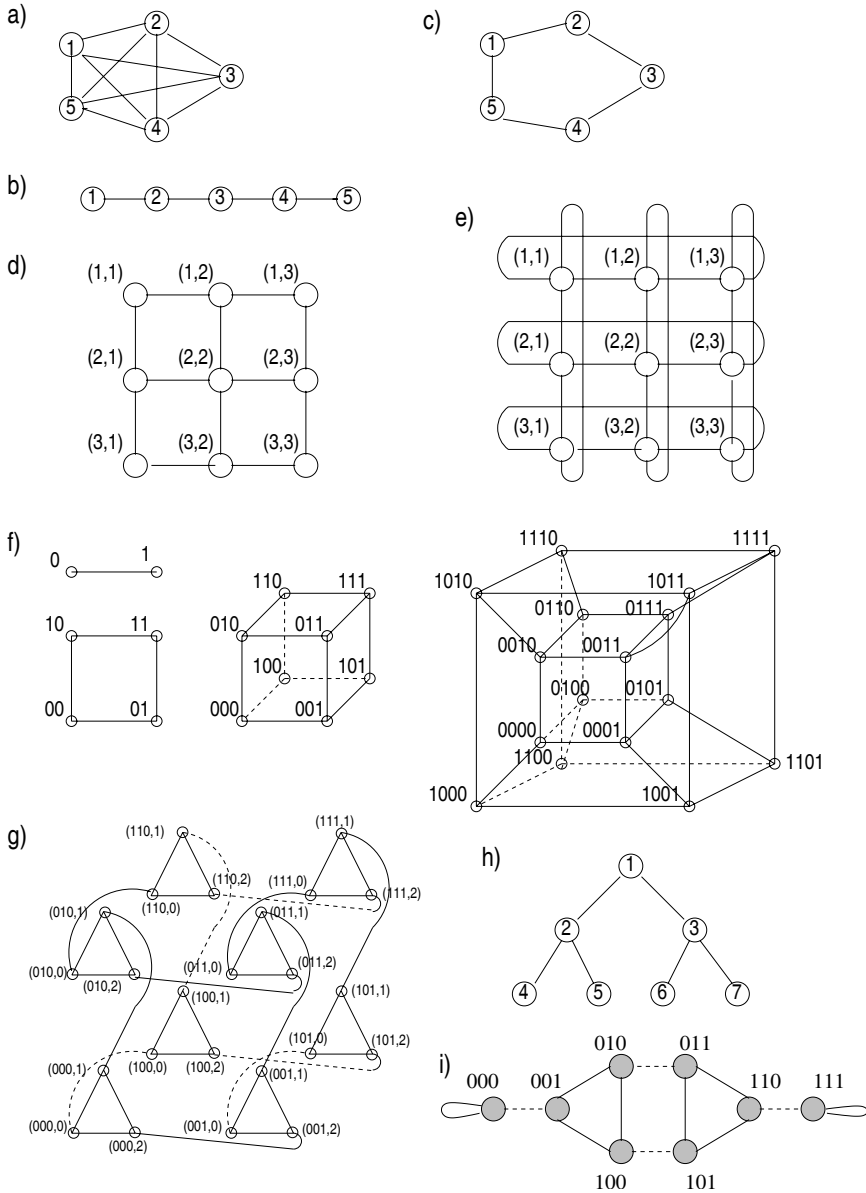


Abb. 2.9. Spezielle statische Verbindungsnetzwerke: a) Vollständiger Graph, b) Lineares Feld, c) Ring, d) 2-dimensionales Gitter e) 2-dimensionaler Torus, f) k -dimensionaler Würfel für $k=1,2,3,4$, g) Cube-connected-cycles Netzwerk für $k=3$, h) vollständiger binärer Baum, i) Shuffle-Exchange-Netzwerk mit 8 Knoten, wobei die gestrichelten Kanten Austauschanten und die durchgezogenen Kanten Mischkanten darstellen.

Anordnung zum letzten Prozessor, vgl. Abbildung 2.9(c). Bei bidirektionalen Verbindungen ist der Durchmesser $\delta(G) = \lfloor n/2 \rfloor$, die Konnektivität $nc(G) = ec(G) = 2$, der Grad $g(G) = 2$ und die Bisektionsbandbreite $B(G) = 2$. In der Praxis ist der Ring für kleine Prozessoranzahlen und als Bestandteil komplexerer Netzwerke einsetzbar.

- (d) Ein **d -dimensionales Gitter** oder **d -dimensionales Feld** ($d \geq 1$) besteht aus $n = n_1 \cdot n_2 \cdot \dots \cdot n_d$ Knoten, die in Form eines d -dimensionalen Gitters angeordnet werden können, vgl. Abbildung 2.9(d). Dabei bezeichnet n_j für $j = 1, \dots, d$ die Ausdehnung in Dimension j . Jeder Knoten in diesem d -dimensionalen Gitter wird durch seine Position $(x_1, \dots, x_d)$ mit $1 \leq x_j \leq n_j$ für $j = 1, \dots, d$ beschrieben. Zwischen Knoten $(x_1, \dots, x_d)$ und Knoten $(x'_1, \dots, x'_d)$ gibt es genau dann eine Kante, wenn es ein $\mu \in \{1, \dots, d\}$ gibt mit

$$|x_\mu - x'_\mu| = 1 \text{ und } x_j = x'_j \text{ für alle } j \neq \mu.$$

Ist $n_j = r = \sqrt[d]{n}$ für alle $j = 1, \dots, d$ (also $n = r^d$), so ist der Durchmesser $\delta(G) = d \cdot (\sqrt[d]{n} - 1)$. Die Knoten- und Kantenkonnektivität ist $nc(G) = ec(G) = d$, da z.B. die Eckknoten durch Löschen der d Nachbarknoten oder der d einlaufenden Kanten vom Rest des Netzwerkes abgetrennt werden können. Der Grad ist $g(G) = 2d$. Ein 2-dimensionales Gitter wurde z.B. für den Terascale-Chip von Intel vorgeschlagen, vgl. Abschnitt 2.8.

- (e) Ein **d -dimensionaler Torus** ist eine Variante des d -dimensionalen Gitters, die zusätzlich zu den Knoten und Kanten des Gitters für jede Dimension $j = 1, \dots, d$ Kanten zwischen den Knoten $(x_1, \dots, x_{j-1}, 1, x_{j+1}, \dots, x_d)$ und $(x_1, \dots, x_{j-1}, n_j, x_{j+1}, \dots, x_d)$ enthält, vgl. Abbildung 2.9(e). Für den Spezialfall $n_j = \sqrt[d]{n}$ für alle $j = 1, \dots, d$, reduziert sich der Durchmesser gegenüber dem Gitter dadurch auf $\delta(G) = d \cdot \lfloor \sqrt[d]{n}/2 \rfloor$. Der Grad ist für alle Knoten $g(G) = 2d$ und die Konnektivität ist ebenfalls $nc(G) = ec(G) = 2d$. Ein 3-dimensionaler Torus wird als Topologie für die Cray XT3 und XT4 sowie für die Übertragung von Punkt-zu-Punkt-Nachrichten in den IBM BlueGene/L-Systemen verwendet.
- (f) Ein **k -dimensionaler Würfel** oder **Hyperwürfel** hat $n = 2^k$ Knoten, zwischen denen Kanten entsprechend eines rekursiven Aufbaus aus niedrigerdimensionalen Würfeln existieren, vgl. Abbildung 2.9(f). Jedem Knoten wird ein binäres Wort der Länge k als Namen zugeordnet, wobei diese k -Bitworte den Dezimalzahlen $0, \dots, 2^k - 1$ entsprechen. Ein 1-dimensionaler Würfel besteht aus zwei Knoten mit den 1-Bitnamen 0 bzw. 1 und einer Kante, die diese beiden Knoten verbindet. Ein k -dimensionaler Würfel wird aus zwei $(k-1)$ -dimensionalen Würfeln (mit jeweiliger Knotennummerierung $0, \dots, 2^{k-1} - 1$) konstruiert. Dazu werden alle Knoten und Kanten der beiden $(k-1)$ -dimensionalen Würfel übernommen und zusätzliche Kanten zwischen zwei Knoten gleicher Nummer gezogen. Die Knotennummerierung wird neu festgelegt, indem die Knoten des ersten $(k-1)$ -dimensionalen Würfels eine zusätzliche 0 und die

Knoten des zweiten $(k - 1)$ -dimensionalen Würfels eine zusätzliche 1 vor ihre Nummer erhalten. Werden die Knoten des k -dimensionalen Würfels mit ihrer Nummerierung identifiziert, also $V = \{0, 1\}^k$, so existiert entsprechend der Konstruktion eine Kante zwischen Knoten $\alpha_0 \dots \alpha_j \dots \alpha_{k-1}$ und Knoten $\alpha_0 \dots \bar{\alpha}_j \dots \alpha_{k-1}$ für $0 \leq j \leq k - 1$, wobei $\bar{\alpha}_j = 1$ für $\alpha_j = 0$ und $\bar{\alpha}_j = 0$ für $\alpha_j = 1$ gilt. Es gibt also Kanten zwischen Knoten, die sich genau in einem Bit unterscheiden. Dieser Zusammenhang wird oft mit Hilfe der Hamming-Distanz beschrieben.

Hamming-Distanz

Die Hamming-Distanz zweier gleich langer binärer Worte ist als die Anzahl der Bits definiert, in denen sich die Worte unterscheiden.

Zwei Knoten des k -dimensionalen Würfels sind also direkt miteinander verbunden, falls ihre Hamming-Distanz 1 ist. Zwischen zwei Knoten $v, w \in V$ mit Hamming-Distanz d , $1 \leq d \leq k$, existiert ein Pfad der Länge d , der v und w verbindet. Dieser Pfad kann bestimmt werden, indem die Bitdarstellung von v von links nach rechts durchlaufen wird und nacheinander die Bits invertiert werden, in denen sich v und w unterscheiden. Jede Bitumkehrung entspricht dem Übergang zu einem Nachbarknoten. Der Durchmesser eines k -dimensionalen Würfels ist $\delta(G) = k$, da sich die Bitdarstellungen der Knoten in höchstens k Positionen unterscheiden können und es daher zwischen beliebigen Knoten einen Pfad der Länge $\leq k$ gibt. Der Grad ist $g(G) = k$, da es in Bitworten der Länge k genau k einzelne Bitumkehrungen, also direkte Nachbarn, gibt. Für die Knoten- und Kantenkonnektivität gilt ebenfalls $nc(G) = k$, wie aus folgender Betrachtung ersichtlich ist.

Die Konnektivität ist höchstens k , d.h. $nc(G) \leq k$, da durch das Löschen der k Nachbarknoten bzw. -kanten ein Knoten vollständig vom Gesamtgraphen abgetrennt werden kann. Um zu zeigen, dass die Konnektivität nicht kleiner als k sein kann, wird nachgewiesen, dass es zwischen zwei beliebigen Knoten v und w genau k unabhängige Pfade gibt, d.h. Pfade, die keine gemeinsamen Kanten und nur gleiche Anfangs- und Endknoten haben. Seien nun A und B die Bitnummerierungen der Knoten v und w , die sich in l Bits, $1 \leq l \leq k$, unterscheiden und seien dies (nach evtl. Umnummerierung) die ersten l Bits. Man kann l Pfade der Länge l zwischen Knoten v und w durch Invertieren der ersten l Bits von v konstruieren. Für Pfad i , $i \in \{0, \dots, l - 1\}$, werden nacheinander zunächst die Bits $i, \dots, l - 1$ und anschließend die Bits $0, \dots, i - 1$ invertiert. Weitere $k - l$ Pfade zwischen Knoten v und w , jeweils der Länge $l + 2$, werden konstruiert, indem für $0 \leq i < k - l$ zunächst das $(l + i)$ -te Bit von v invertiert wird, dann nacheinander die Bits der Positionen $0, \dots, l - 1$ invertiert werden und abschließend das $(l + i)$ -te Bit wieder zurückinvertiert wird. Abbildung 2.10 zeigt ein Beispiel. Alle k konstruierten Pfade sind unabhängig voneinander und es folgt, dass $nc(G) \geq k$ gilt.

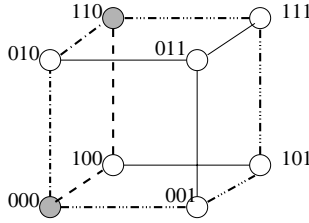


Abb. 2.10. In einem 3-dimensionalen Hyperwürfel gibt es drei unabhängige Pfade von Knoten 000 zu Knoten 110. Die Hamming-Distanz zwischen Knoten 000 und Knoten 110 ist $l = 2$. Es existieren zwei Pfade zwischen Knoten 000 und Knoten 110 der Länge $l = 2$, nämlich die Pfade $(000, 100, 110)$ und $(000, 010, 110)$, und $k - l = 1$ Pfade der Länge $l + 2 = 4$, nämlich $(000, 001, 101, 111, 110)$.

In einen k -dimensionalen Würfel können sehr viele Netzwerke eingebettet werden, worauf wir später noch eingehen werden.

- (g) Ein CCC-Netzwerk (**C**ube **C**onnecte**C**ycles) entsteht aus einem k -dimensionalen Würfel, indem jeder Knoten durch einen Zyklus (Ring) aus k Knoten ersetzt wird. Jeder dieser Knoten im Zyklus übernimmt eine Verbindung zu einem Nachbarn des ehemaligen Knotens, vgl. Abbildung 2.9(g). Die Knotenmenge des CCC-Netzwerkes wird gegenüber dem k -dimensionalen Würfel erweitert auf $V = \{0, 1\}^k \times \{0, \dots, k - 1\}$, wobei $\{0, 1\}^k$ die Knotenbezeichnung des k -dimensionalen Würfels ist und $i \in \{0, \dots, k - 1\}$ die Position im Zyklus angibt. Die Kantenmenge besteht aus einer Menge F von Zykluskanten und einer Menge E von Hyperwürfelkanten, d.h.

$$F = \{((\alpha, i), (\alpha, (i + 1) \bmod k)) \mid \alpha \in \{0, 1\}^k, 0 \leq i < k\},$$

$$E = \{((\alpha, i), (\beta, i)) \mid \alpha_i \neq \beta_i \text{ und } \alpha_j = \beta_j \text{ für } j \neq i\}.$$

Jeder der insgesamt $k \cdot 2^k$ Knoten hat den Grad $g(G) = 3$, wodurch der Nachteil des großen Grades beim k -dimensionalen Würfel beseitigt wird. Die Konnektivität ist $nc(G) = ec(G) = 3$, denn ein Knoten kann durch Löschen von 3 Kanten bzw. 3 Knoten vom Restgraphen abgehängt werden. Eine obere Schranke für den Durchmesser ist $\delta(G) = 2k - 1 + \lfloor \frac{k}{2} \rfloor$. Zur Konstruktion eines Pfades mit Durchmesserlänge betrachten wir zwei Knoten in Zyklen mit maximalem Hyperwürfelabstand k , d.h. Knoten (α, i) und (β, j) , bei denen sich die k -Bitworte α und β in jedem Bit unterscheiden. Wir wählen einen Pfad von (α, i) nach (β, j) , indem wir nacheinander jeweils eine Hyperwürfelverbindung und eine Zyklusverbindung durchwandern. Der Pfad startet mit $(\alpha_0 \dots \alpha_i \dots \alpha_{k-1}, i)$ und erreicht den nächsten Knoten durch Invertierung von α_i zu $\bar{\alpha}_i = \beta_i$. Von $(\alpha_0 \dots \beta_i \dots \alpha_{k-1}, i)$ gelangen wir über eine Zykluskante zum nächsten Knoten; dieser ist $(\alpha_0 \dots \beta_i \dots \alpha_{k-1}, (i + 1) \bmod k)$. In den nächsten

Schritten werden ausgehend vom bereits erreichten Knoten nacheinander die Bits $\alpha_{i+1}, \dots, \alpha_{k-1}$, und dann $\alpha_0, \dots, \alpha_{i-1}$ invertiert. Dazwischen laufen wir jeweils im Zyklus um einen Position weiter. Dies ergibt $2k - 1$ Schritte. In maximal $\lfloor \frac{k}{2} \rfloor$ weiteren Schritten gelangt man von $(\beta, i + k - 1 \bmod k)$ durch Verfolgen von Zykluskanten zum Ziel (β, j) .

- (h) Das Netzwerk eines **vollständigen, binären Baumes** mit $n = 2^k - 1$ Knoten ist ein binärer Baum, in dem alle Blattknoten die gleiche Tiefe haben und der Grad aller inneren Knoten $g(G) = 3$ ist. Der Durchmesser ist $\delta(G) = 2 \log \frac{n+1}{2}$ und wird durch den Pfad zwischen zwei Blättern in verschiedenen Unterbäumen des Wurzelknotens bestimmt, der sich aus dem Pfad des einen Blattes zur Wurzel des Baumes und dem Pfad von der Wurzel zum zweiten Blatt zusammensetzt. Die Konnektivität ist $nc(G) = ec(G) = 1$, da durch Wegnahme der Wurzel bzw. einer der von der Wurzel ausgehenden Kanten der Baum zerfällt.
- (i) Ein k -dimensionales **Shuffle-Exchange-Netzwerk** besitzt $n = 2^k$ Knoten und $3 \cdot 2^{k-1}$ Kanten [156]. Werden die Knoten mit den k -Bitworten für $0, \dots, n-1$ identifiziert, so ist ein Knoten α mit einem Knoten β genau dann verbunden, falls gilt:
- a) α und β unterscheiden sich nur im letzten (rechtsten) Bit (Austauschkante, *exchange edge*) oder
 - b) α entsteht aus β durch einen zyklischen Linksshift oder einen zyklischen Rechtsshift von β (Mischkante, *shuffle edge*).

Abbildung 2.9i) zeigt ein Shuffle-Exchange-Netzwerk mit 8 Knoten. Die Permutation (α, β) , wobei β aus α durch zyklischen Linksshift entsteht, heißt auch **perfect shuffle**. Die Permutation (α, β) , wobei β aus α durch zyklischen Rechtsshift entsteht, heißt entsprechend auch **inverse perfect shuffle**. Viele Eigenschaften von Shuffle-Exchange-Netzwerken sind in [101] beschrieben. Tabelle 2.2 fasst die Eigenschaften der beschriebenen Netzwerke zusammen.

Ein **k -facher d -Würfel** (engl. *k-ary d-cube*) mit $k \geq 2$ ist eine Verallgemeinerung des d -dimensionalen Gitters mit $n = k^d$ Knoten, wobei jeweils k Knoten in einer Dimension i liegen, $i = 0, \dots, d-1$. Jeder Knoten im k -fachen d -Würfel hat einen Namen aus n Ziffern $(a_0, \dots, a_{d-1})$ mit $0 \leq a_i \leq k-1$. Die i -te Ziffer a_i repräsentiert die Position des Knotens in Dimension i , $i = 0, \dots, d-1$. Zwei Knoten A und B mit Namen $(a_0, \dots, a_{d-1})$ bzw. $(b_0, \dots, b_{d-1})$ sind genau dann durch eine Kante verbunden, wenn für ein $j \in \{0, \dots, d-1\}$ gilt: $a_j = (b_j \pm 1) \bmod k$ und $a_i = b_i$ für alle $i = 0, \dots, d-1, i \neq j$. Bedingt durch einen bzw. zwei Nachbarn in jeder Dimension hat ein Knoten für $k = 2$ den Grad $g(G) = d$ und für $k > 2$ den Grad $g(G) = 2d$. Der k -fache d -Würfel umfasst einige der oben genannten speziellen Topologien. So entspricht ein k -facher 1-Würfel einem Ring mit k Knoten, ein k -facher 2-Würfel einem Torus mit k^2 Knoten, ein 3-facher 3-Würfel einem 3-dimensionalen Torus mit $3 \times 3 \times 3$ Knoten und ein 2-facher d -Würfel einem d -dimensionalen Hyperwürfel.

Tabelle 2.2. Zusammenfassung der Parameter statischer Verbindungsnetzwerke für ausgewählte Topologien.

Netzwerk G mit n Knoten	Grad $g(G)$	Durchmesser $\delta(G)$	Kanten- konnektivität $ec(G)$	Bisektions- bandbreite $B(G)$
Vollständiger Graph	$n - 1$	1	$n - 1$	$\left(\frac{n}{2}\right)^2$
Lineares Feld	2	$n - 1$	1	1
Ring	2	$\left\lfloor \frac{n}{2} \right\rfloor$	2	2
d -dimensionales Gitter ($n = r^d$)	$2d$	$d(\sqrt[d]{n} - 1)$	d	$n^{\frac{d-1}{d}}$
d -dimensionaler Torus ($n = r^d$)	$2d$	$d \left\lfloor \frac{\sqrt[d]{n}}{2} \right\rfloor$	$2d$	$2n^{\frac{d-1}{d}}$
k -dimensionaler Hyper- würfel ($n = 2^k$)	$\log n$	$\log n$	$\log n$	$\frac{n}{2}$
k -dimensionales CCC-Netzwerk ($n = k2^k$ für $k \geq 3$)	3	$2k - 1 + \lfloor k/2 \rfloor$	3	$\frac{n}{2k}$
Vollständiger binärer Baum ($n = 2^k - 1$)	3	$2 \log \frac{n+1}{2}$	1	1
k -facher d -Würfel ($n = k^d$)	$2d$	$d \left\lfloor \frac{k}{2} \right\rfloor$	$2d$	$2k^{d-1}$

2.5.3 Einbettungen

Einbettung eines Rings in einen k -dimensionalen Würfel. Zur Konstruktion einer Einbettung eines Rings mit $n = 2^k$ Knoten in einen k -dimensionalen Würfel wird die Knotenmenge $V' = \{1, \dots, n\}$ des Rings durch eine bijektive Funktion so auf die Knotenmenge $V = \{0, 1\}^k$ abgebildet, dass die Kanten $(i, j) \in E'$ des Rings auf Kanten in E des Würfels abgebildet werden. Da die Knoten des Ringes mit $1, \dots, n$ durchnummeriert werden können, kann eine Einbettung dadurch konstruiert werden, dass eine entsprechende Aufzählung der Knoten im Würfel konstruiert wird, so dass zwischen aufeinanderfolgend aufgezählten Knoten eine Kante im Würfel existiert. Die Einbettungskonstruktion verwendet (gespiegelte) **Gray-Code-Folgen** (engl. *reflected Gray code* - RGC).

Gespiegelter Gray-Code – RGC

Ein k -Bit Gray-Code ist ein 2^k -Tupel aus k -Bitzahlen, wobei sich aufeinanderfolgende Zahlen im Tupel in genau einer Bitposition unterscheiden. Der gespiegelte k -Bit Gray-Code wird folgendermaßen rekursiv definiert:

- Der 1-Bit Gray-Code ist $RGC_1 = (0, 1)$.
- Der 2-Bit Gray-Code wird aus RGC_1 aufgebaut, indem einmal 0 und einmal 1 vor die Elemente von RGC_1 gesetzt wird und die beiden resultierenden Folgen

(00, 01) und (10, 11) nach Umkehrung der zweiten Folge konkateniert werden. Damit ergibt sich $RGC_2 = (00, 01, 11, 10)$.

- Der k -Bit Gray-Code RGC_k für $k \geq 2$ wird aus dem $(k-1)$ -Bit Gray-Code $RGC_{k-1} = (b_1, \dots, b_m)$ mit $m = 2^{k-1}$ konstruiert, dessen Einträge b_i für $1 \leq i \leq m$ binäre Worte der Länge $k-1$ sind. Zur Konstruktion von RGC_k wird RGC_{k-1} dupliziert, vor jedes binäre Wort des Originals wird eine 0 und vor jedes binäre Wort des Duplikats wird eine 1 gesetzt. Die resultierenden Folgen sind $(0b_1, \dots, 0b_m)$ und $(1b_1, \dots, 1b_m)$. RGC_k resultiert durch Umkehrung der zweiten Folge und Konkatenation, also $RGC_k = (0b_1, \dots, 0b_m, 1b_m, \dots, 1b_1)$.

Die so konstruierten Gray-Codes RGC_k haben für beliebige k die Eigenschaft, dass sie alle Knotennummierungen eines k -dimensionalen Würfels enthalten, da die Konstruktion der oben beschriebenen Konstruktion eines k -dimensionalen Hyperwürfels aus zwei $(k-1)$ -dimensionalen Hyperwürfeln entspricht. Weiter unterscheiden sich benachbarte Elemente von RGC_k in genau einem Bit. Dies lässt sich durch Induktion beweisen: Die Aussage gilt für Nachbarn aus den ersten bzw. letzten 2^{k-1} Elementen von RGC_k nach Induktionsannahme, da im Vergleich zu RGC_{k-1} nur eine 0 oder 1 vorangestellt wurde. Die Aussage gilt auch für die mittleren Nachbarelemente $0b_m$ und $1b_m$. Analog unterscheiden sich das erste und das letzte Element von RGC_k nach Konstruktion im vordersten Bit. Damit sind in RGC_k benachbarte Knoten durch eine Kante im Würfel miteinander verbunden. Die Einbettung eines Rings in einen k -dimensionalen Würfel wird also durch die Abbildung

$$\sigma : \{1, \dots, n\} \rightarrow \{0, 1\}^k \text{ mit } \sigma(i) := RGC_k(i)$$

definiert, wobei $RGC_k(i)$ das i -te Element der Folge RGC_k bezeichnet. Abbildung 2.11 a) zeigt ein Beispiel.

Einbettung eines 2-dimensionalen Gitters in einen k -dimensionalen Würfel. Die Einbettung eines zweidimensionalen Feldes mit $n = n_1 \cdot n_2$ Knoten in einen k -dimensionalen Würfel mit $n = 2^k$ Knoten stellt eine Verallgemeinerung der Einbettung des Rings dar. Für k_1 und k_2 mit $n_1 = 2^{k_1}$ und $n_2 = 2^{k_2}$, also $k_1 + k_2 = k$, werden Gray-Code $RGC_{k_1} = (a_1, \dots, a_{n_1})$ und Gray-Code $RGC_{k_2} = (b_1, \dots, b_{n_2})$ benutzt, um eine $n_1 \times n_2$ Matrix M mit Einträgen aus k -Bitworten zu konstruieren, und zwar $M(i, j) = \{a_i b_j\}_{i=1, \dots, n_1, j=1, \dots, n_2}$:

$$M = \begin{bmatrix} a_1 b_1 & a_1 b_2 & \dots & a_1 b_{n_2} \\ a_2 b_1 & a_2 b_2 & \dots & a_2 b_{n_2} \\ \vdots & \vdots & & \vdots \\ a_{n_1} b_1 & a_{n_1} b_2 & \dots & a_{n_1} b_{n_2} \end{bmatrix}.$$

Benachbarte Elemente der Matrix M unterscheiden sich in genau einer Bitposition. Dies gilt für in einer Zeile benachbarte Elemente, da identische Elemente von RGC_{k_1} und benachbarte Elemente von RGC_{k_2} verwendet werden. Analog gilt dies für in einer Spalte benachbarte Elemente, da identische

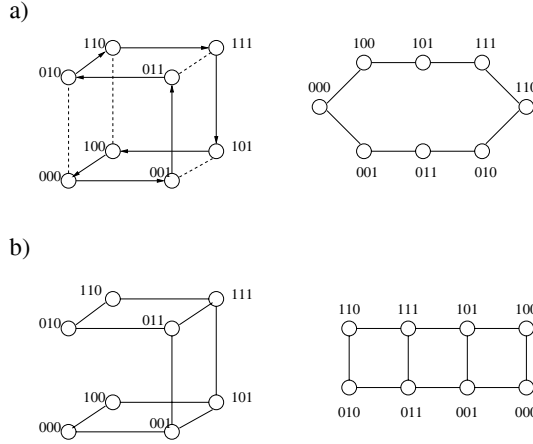


Abb. 2.11. Einbettungen in einen Hyperwürfel: a) Einbettung eines Ringes mit 8 Knoten in einen 3-dimensionalen Hyperwürfel und b) Einbettung eines 2-dimensionalen 2×4 Gitters in einen 3-dimensionalen Hyperwürfel.

Elemente von RGC_{k_2} und benachbarte Elemente von RGC_{k_1} verwendet werden. Alle Elemente von M sind unterschiedliche Bitworte der Länge k . Die Matrix M enthält also alle Namen von Knoten im k -dimensionalen Würfel genau einmal und Nachbarschaftsbeziehungen der Einträge der Matrix entsprechen Nachbarschaftsbeziehungen der Knoten im k -dimensionalen Würfel. Die Abbildung

$$\sigma : \{1, \dots, n_1\} \times \{1, \dots, n_2\} \rightarrow \{0, 1\}^k \text{ mit } \sigma((i, j)) = M(i, j)$$

ist also eine Einbettung in den k -dimensionalen Würfel. Abbildung 2.11 b) zeigt ein Beispiel.

Einbettung eines d -dimensionalen Gitters in einen k -dimensionalen Würfel. In einem d -dimensionalen Gitter mit $n_i = 2^{k_i}$ Gitterpunkten in der i -ten Dimension, $1 \leq i \leq d$, werden die insgesamt $n = n_1 \cdot \dots \cdot n_d$ Gitterpunkte jeweils als Tupel $(x_1, \dots, x_d)$ dargestellt, $1 \leq x_i \leq n_i$. Die Abbildung

$$\sigma : \{(x_1, \dots, x_d) \mid 1 \leq x_i \leq n_i, 1 \leq i \leq d\} \longrightarrow \{0, 1\}^k$$

$$\text{mit } \sigma((x_1, \dots, x_d)) = s_1 s_2 \dots s_d \text{ und } s_i = RGC_{k_i}(x_i)$$

(d.h. s_i ist der x_i -te Bitstring im Gray-Code RGC_{k_i}) stellt eine Einbettung in den k -dimensionalen Würfel dar. Für zwei Gitterpunkte $(x_1, \dots, x_d)$ und $(y_1, \dots, y_d)$, die durch eine Kante im Gitter verbunden sind, gilt nach der Definition des d -dimensionalen Gitters, dass es genau eine Komponente $i \in \{1, \dots, d\}$ mit $|x_i - y_i| = 1$ gibt und dass für alle anderen Komponenten $j \neq i$ $x_j = y_j$ gilt. Für die Bilder $\sigma((x_1, \dots, x_d)) = s_1 s_2 \dots s_d$ und $\sigma((y_1, \dots, y_d)) = t_1 t_2 \dots t_d$ sind also alle Komponenten $s_j = RGC_{k_j}(x_j) = RGC_{k_j}(y_j) = t_j$

für $j \neq i$ identisch und $RGC_{k_i}(x_i)$ unterscheidet sich von $RGC_{k_i}(y_i)$ in genau einer Bitposition. Die Knoten $s_1 s_2 \dots s_d$ und $t_1 t_2 \dots t_d$ sind also durch eine Kante im k -dimensionalen Würfel verbunden.

2.5.4 Dynamische Verbindungsnetzwerke

Dynamische Verbindungsnetzwerke stellen keine physikalischen Punkt-zu-Punkt-Verbindungen zwischen Prozessoren bereit, sondern bieten stattdessen die Möglichkeit der indirekten Verbindung zwischen Prozessoren (bei Systemen mit verteiltem Speicher) bzw. zwischen Prozessoren und Speichermodulen (bei Systemen mit gemeinsamem Speicher), worauf auch die Bezeichnung *indirektes* Verbindungsnetzwerk beruht. Aus der Sicht der Prozessoren stellt ein dynamisches Verbindungsnetzwerk eine Einheit dar, in die Nachrichten oder Speicheranforderungen eingegeben werden und aus der Nachrichten oder zurückgelieferte Daten empfangen werden. Intern ist ein dynamisches Verbindungsnetzwerk aus mehreren physikalischen Leitungen und dazwischenliegenden Schaltern aufgebaut, aus denen gemäß der Anforderungen einer Nachrichtenübertragung dynamisch eine Verbindung zwischen zwei Komponenten aufgebaut wird, was zur Bezeichnung *dynamisches Verbindungsnetzwerk* geführt hat.

Dynamische Verbindungsnetzwerke werden hauptsächlich für Systeme mit gemeinsamem Speicher genutzt, siehe Abbildung 2.6. In diesem Fall kann ein Prozessor nur *indirekt* über das Verbindungsnetzwerk auf den gemeinsamen Speicher zugreifen. Besteht der gemeinsame Speicher, wie dies meist der Fall ist, aus mehreren Speichermodulen, so leitet das Verbindungsnetzwerk die Datenzugriffe der Prozessoren anhand der spezifizierten Speicheradresse zum richtigen Speichermodul weiter.

Auch dynamische Verbindungsnetzwerke werden entsprechend ihrer topologischen Ausprägungen charakterisiert. Neben busbasierten Verbindungsnetzwerken werden mehrstufige Schaltnetzwerke, auch Switchingnetzwerke genannt, und Crossbars unterschieden.

Busnetzwerke. Ein *Bus* besteht im Wesentlichen aus einer Menge von Leitungen, über die Daten von einer Quelle zu einem Ziel transportiert werden können, vgl. Abbildung 2.12. Um größere Datenmengen schnell zu transportieren, werden oft mehrere Hundert Leitungen verwendet. Zu einem bestimmten Zeitpunkt kann jeweils nur ein Datentransport über den Bus stattfinden (*time-sharing*). Falls zum gleichen Zeitpunkt mehrere Prozessoren gleichzeitig einen Datentransport über den Bus ausführen wollen, muss ein spezieller **Busarbiter** die Ausführung der Datentransporte koordinieren (*contention-Bus*). Busnetzwerke werden meist nur für eine kleine Anzahl von Prozessoren eingesetzt, also etwa für 32 bis 64 Prozessoren.

Crossbar-Netzwerke. Die höchste Verbindungskapazität zwischen Prozessoren bzw. zwischen Prozessoren und Speichermodulen stellt ein *Crossbar-Netzwerk* bereit. Ein $n \times m$ -Crossbar-Netzwerk hat n Eingänge, m Ausgänge

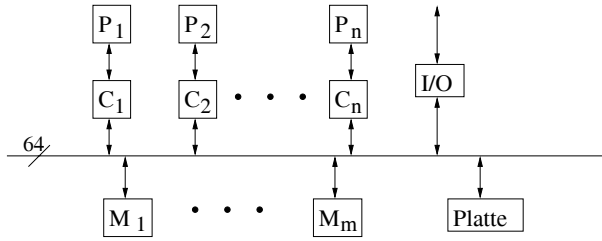


Abb. 2.12. Bus mit 64 Bit-Leitung zur Verbindung der Prozessoren $P_1, \dots, P_n$ und ihrer Caches $C_1, \dots, C_n$ mit den Speichermodulen $M_1, \dots, M_m$.

und besteht aus $n \cdot m$ Schaltern wie in Abbildung 2.13 skizziert ist. Für jede Datenübertragung bzw. Speicheranfrage von einem bestimmten Eingang zum gewünschten Ausgang wird ein Verbindungspfad im Netzwerk aufgebaut. Entsprechend der Anforderungen der Nachrichtenübertragung können die Schalter an den Kreuzungspunkten des Pfades die Nachricht geradeaus oder mit einer Richtungsänderung um 90 Grad weiterleiten. Wenn wir davon ausgehen, dass jedes Speichermodul zu jedem Zeitpunkt nur eine Speicheranfrage befriedigen kann, darf in jeder Spalte von Schaltern nur *ein* Schalter auf Umlenken (also nach unten) gestellt sein. Ein Prozessor kann jedoch gleichzeitig mehrere Speicheranfragen an verschiedene Speichermodule stellen. Üblicherweise werden Crossbar-Netzwerke wegen des hohen Hardwareaufwandes nur für eine kleine Anzahl von Prozessoren realisiert.

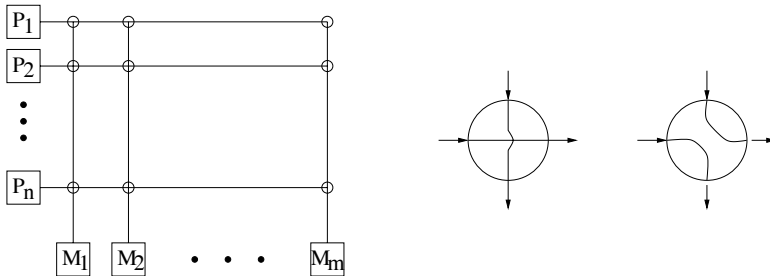


Abb. 2.13. Illustration eines $n \times m$ -Crossbar-Netzwerkes mit n Prozessoren und m Speichermodulen (links) und der beiden möglichen Schalterstellungen der Schalter an Kreuzungspunkten des Crossbar-Netzwerkes (rechts).

Mehrstufige Schaltnetzwerke. Mehrstufige Schalt- oder Switchingnetzwerke (engl. *multistage switching network*) sind aus mehreren Schichten von Schaltern und dazwischenliegenden Leitungen aufgebaut. Ein Ziel besteht dabei darin, bei größerer Anzahl von zu verbindenden Prozessoren einen geringeren tatsächlichen Abstand zu erhalten als dies bei direkten Verbindungs-

netzwerken der Fall wäre. Die interne Verbindungsstruktur dieser Netzwerke kann durch Graphen dargestellt werden, in denen die Schalter den Knoten und Leitungen zwischen Schaltern den Kanten entsprechen. In diese Graphdarstellung werden häufig auch die Verbindungen vom Netzwerk zu Prozessoren bzw. Speichermodulen einbezogen: Prozessoren und Speichermodule sind ausgezeichnete Knoten, deren Verbindungen zu dem eigentlichen Schaltnetzwerk durch zusätzliche Kanten dargestellt werden. Charakterisierungskriterien für mehrstufige Schaltnetzwerke sind die *Konstruktionsvorschrift* des Aufbaus des entsprechenden Graphen und der *Grad* der den Schaltern entsprechenden Knoten im Graphen.

Die sogenannten **regelmäßigen mehrstufigen Verbindungsnetzwerke** zeichnen sich durch eine *regelmäßige* Konstruktionsvorschrift und einen gleichgroßen Grad von eingehenden bzw. ausgehenden Leitungen für alle Schalter aus. Die Schalter in mehrstufigen Verbindungsnetzwerken werden häufig als $a \times b$ -Crossbars realisiert, wobei a den Eingangsgrad und b den Ausgangsgrad des entsprechenden Knotens bezeichnet. Die Schalter sind in einzelnen *Stufen* angeordnet, wobei benachbarte Stufen durch feste Verbindungsleitungen miteinander verbunden sind, siehe Abbildung 2.14. Die Schalter der ersten Stufe haben Eingangskanten, die mit den Prozessoren verbunden sind. Die ausgehenden Kanten der letzten Schicht stellen die Verbindung zu Speichermodulen (bzw. ebenfalls zu Prozessoren) dar. Speicherzugriffe von Prozessoren auf Speichermodule (bzw. Nachrichtenübertragungen zwischen Prozessoren) finden über das Netzwerk statt, indem von der Eingangskante des Prozessors bis zur Ausgangskante zum Speichermodul ein Pfad über die einzelnen Stufen gewählt wird und die auf diesem Pfad liegenden Schalter dynamisch so gesetzt werden, dass die gewünschte Verbindung entsteht.

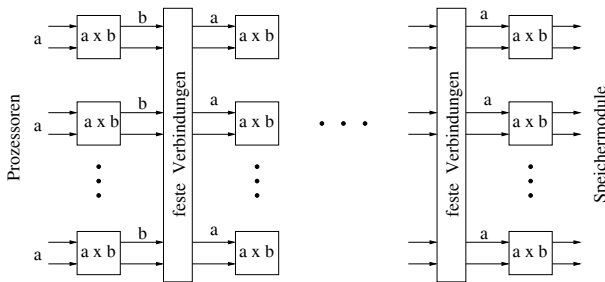


Abb. 2.14. Mehrstufige Schaltnetzwerke mit $a \times b$ -Crossbars als Schalter nach [85].

Der Aufbau des Graphen für regelmäßige mehrstufige Verbindungsnetzwerke entsteht durch „Verkleben“ der einzelnen *Stufen* von Schaltern. Jede Stufe ist durch einen gerichteten azyklischen Graphen der Tiefe 1 mit w Knoten dargestellt. Der Grad jedes Knotens ist $g = n/w$, wobei n die Anzahl der zu verbindenden Prozessoren bzw. der nach außen sichtbaren Lei-

tungen des Verbindungsnetzwerkes ist. Das Verkleben der einzelnen Stufen wird durch eine Permutation $\pi : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$ beschrieben, die die durchnummerierten ausgehenden Leitungen einer Stufe i so umsortiert, dass die entstehende Permutation $(\pi(1), \dots, \pi(n))$ der durchnummerierten Folge von eingehenden Leitungen in die Stufe $i + 1$ des mehrstufigen Netzwerkes entspricht. Die Partition der Permutation $(\pi(1), \dots, \pi(n))$ in w Teilstücke ergibt die geordneten Mengen der Empfangsleitungen der Knoten der nächsten Schicht. Bei regelmäßigen Verbindungsnetzwerken ist die Permutation für alle Schichten gleich, kann aber evtl. mit der Nummer i der Stufe parametrisiert sein. Bei gleichem Grad der Knoten ergibt die Partition von $(\pi(1), \dots, \pi(n))$ w gleichgroße Teilmengen.

Häufig benutzte regelmäßige mehrstufige Verbindungsnetzwerke sind das Omega-Netzwerk, das Baseline-Netzwerk und das Butterfly-Netzwerk (oder Banyan-Netzwerk), die jeweils Schalter mit Eingangs- und Ausgangsgrad 2 haben und aus $\log n$ Schichten bestehen. Die 2×2 Schalter können vier mögliche Schalterstellungen annehmen, die in Abbildung 2.15 dargestellt sind.

Im Folgenden stellen wir reguläre Verbindungsnetzwerke wie das Omega-, das Baseline- und das Butterfly-Netzwerk sowie das Benes-Netzwerk und den Fat-Tree vor. Ausführliche Beschreibungen dieser Netzwerke sind z.B. in [101] zu finden.

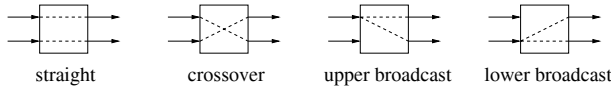


Abb. 2.15. Schalterstellungen, die ein Schalter in einem Omega-, Baseline- oder Butterfly-Netzwerk realisieren kann.

Omega-Netzwerk. Ein $n \times n$ -Omega-Netzwerk besteht aus 2×2 -Crossbar-Schaltern, die in $\log n$ Stufen angeordnet sind, wobei jede Stufe $n/2$ Schalter enthält und jeder Schalter zwei Eingänge und zwei Ausgänge hat. Insgesamt gibt es also $(n/2) \log n$ Schalter. Dabei sei $\log n \equiv \log_2 n$. Jeder der Schalter kann vier Verbindungen realisieren, siehe Abbildung 2.15. Die festen Verbindungen zwischen den Stufen, d.h. also die Permutationsfunktion zum Verkleben der Stufen, ist beim Omega-Netzwerk für alle Stufen gleich und hängt nicht von der Nummer der Stufe ab. Die Verklebungsfunktion des Omega-Netzwerkes ist über eine Nummerierung der Schalter definiert. Die Namen der Schalter sind Paare (α, i) bestehend aus einem $(\log n - 1)$ -Bitwort α , $\alpha \in \{0, 1\}^{\log n - 1}$, und einer Zahl $i \in \{0, \dots, \log n - 1\}$, die die Nummer der Stufe angibt. Es gibt jeweils eine Kante von Schalter (α, i) in Stufe i zu den beiden Schaltern $(\beta, i + 1)$ in Stufe $i + 1$, die dadurch definiert sind, dass

1. entweder β durch einen zyklischen Linksshift aus α hervorgeht oder

2. β dadurch entsteht, dass nach einem zyklischen Linksshift von α das letzte (rechtste) Bit invertiert wird.

Ein $n \times n$ -Omega-Netzwerk wird auch als $(\log n - 1)$ -dimensionales Omega-Netzwerk bezeichnet. Abbildung 2.16 a) zeigt ein 16×16 , also ein dreidimensionales, Omega-Netzwerk mit vier Stufen und acht Schaltern pro Stufe.

Butterfly-Netzwerk. Das k -dimensionale Butterfly-Netzwerk, das auch als **Banyan-Netzwerk** bezeichnet wird, verbindet ebenfalls $n = 2^{k+1}$ Eingänge mit $n = 2^{k+1}$ Ausgängen über ein Netzwerk, das aus $k + 1$ Stufen mit jeweils 2^k Knoten aus 2×2 -Crossbar-Switches aufgebaut ist. Die insgesamt $(k + 1)2^k$ Knoten des Butterfly-Netzwerkes können eindeutig durch Paare (α, i) bezeichnet werden, wobei i ($0 \leq i \leq k$) die Stufe angibt und das k -Bit-Wort $\alpha \in \{0, 1\}^k$ die Position des Knotens in dieser Stufe. Die Verklebungsfunktion der Stufen i und $i + 1$ mit $0 \leq i < k$ des Butterfly-Netzwerkes ist folgendermaßen definiert. Zwei Knoten (α, i) und $(\alpha', i + 1)$ sind genau dann miteinander verbunden, wenn:

1. α und α' identisch sind (direkte Kante, engl. *straight edge*) oder
2. α und α' sich genau im $(i + 1)$ -ten Bit von links unterscheiden (Kreuzkante, engl. *cross edge*).

Abbildung 2.16 b) zeigt ein 16×16 -Butterfly-Netzwerk mit vier Stufen.

Baseline-Netzwerk. Das k -dimensionale Baseline-Netzwerk hat dieselbe Anzahl von Knoten, Kanten und Stufen wie das Butterfly-Netzwerk. Die Stufen werden durch folgende Verklebungsfunktion verbunden. Knoten (α, i) ist für $0 \leq i < k$ genau dann mit Knoten $(\alpha', i + 1)$ verbunden, wenn:

1. das k -Bit-Wort α' aus α durch einen zyklischen Rechtsshift der letzten $k - i$ Bits von α entsteht oder
2. das k -Bit-Wort α' aus α entsteht, indem zuerst das letzte (rechtste) Bit von α invertiert wird und dann ein zyklischer Rechtsshift auf die letzten $k - i$ Bits des entstehenden k -Bit-Wortes angewendet wird.

Abbildung 2.16 c) zeigt ein 16×16 -Baseline-Netzwerk mit vier Stufen.

Benes-Netzwerk. Das k -dimensionale Benes-Netzwerk setzt sich aus zwei k -dimensionalen Butterfly-Netzwerken zusammen und zwar so, dass die ersten $k + 1$ Stufen ein Butterfly-Netzwerk bilden und die letzten $k + 1$ Stufen ein bzgl. der Stufen umgekehrtes Butterfly-Netzwerk bilden, wobei die $(k + 1)$ -te Stufe des ersten Butterfly-Netzwerkes und die erste Stufe des umgekehrten Butterfly-Netzwerkes zusammenfallen. Insgesamt hat das k -dimensionale Benes-Netzwerk also $2k + 1$ Stufen mit je $N = 2^k$ Schalter pro Stufe und eine Verklebungsfunktion der Stufen, die (entsprechend modifiziert) vom Butterfly-Netzwerk übernommen wird. Ein Beispiel eines Benes-Netzwerkes für 16 Eingangskanten ist in Abbildung 2.17 a) gegeben, vgl. [101].

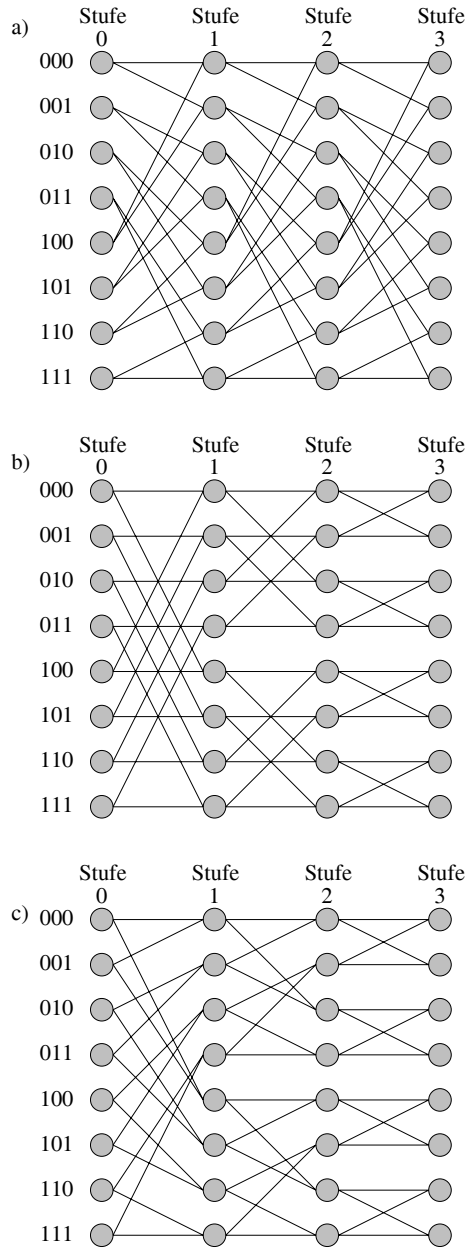


Abb. 2.16. Spezielle dynamische Verbindungsnetzwerke: a) 16×16 Omega-Netzwerk, b) 16×16 Butterfly-Netzwerk, c) 16×16 Baseline-Netzwerk. Alle Netzwerke sind 3-dimensional.

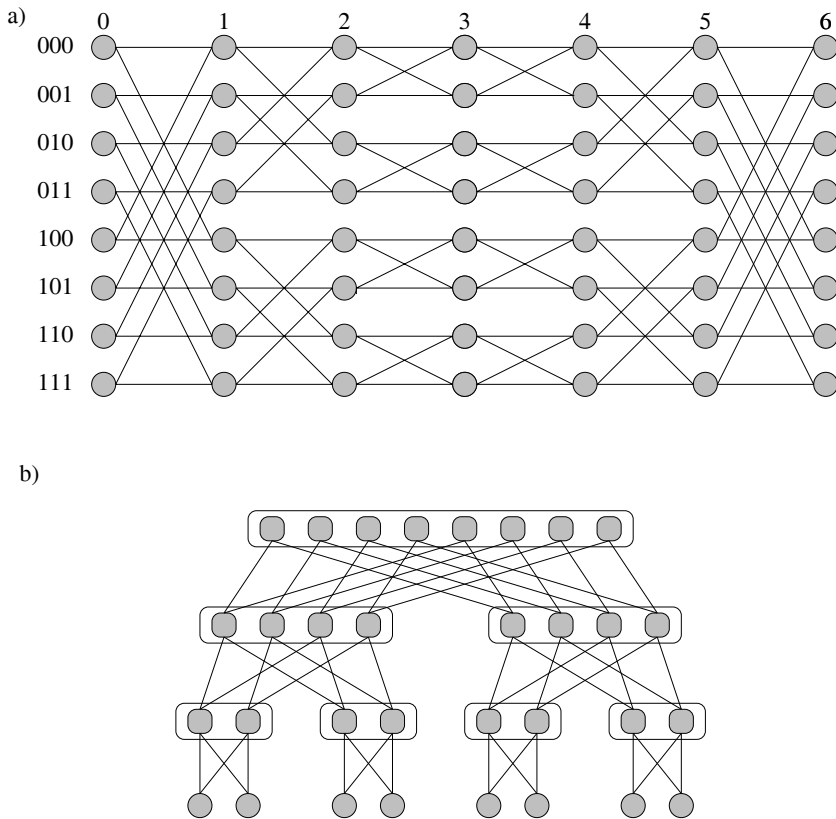


Abb. 2.17. Spezielle dynamische Verbindungsnetzwerke: a) 3-dimensionales Benes-Netzwerk und b) Fattree für 16 Prozessoren.

Fat-Tree. Ein *dynamischer Baum* oder *Fat-Tree* hat als Grundstruktur einen vollständigen, binären Baum, der jedoch (im Gegensatz zum Baum-Netzwerk aus Abschnitt 2.5.2) zur Wurzel hin mehr Kanten aufweist und so den Flaschenhals des Baumes an der Wurzel überwindet. Innere Knoten des Fat-Tree bestehen aus Schaltern, deren Aussehen von der Ebene im Baum abhängen. Stellt ein Fat-Tree ein Netzwerk für n Prozessoren dar, die durch die Blätter des Baumes repräsentiert sind, so hat ein Knoten auf Ebene i für $i = 1, \dots, \log n$ genau 2^i Eingangskanten und 2^i Ausgangskanten. Dabei ist Ebene 0 die Blattebene. Realisiert wird dies z.B. dadurch, dass die Knoten auf Ebene i intern aus 2^{i-1} Schaltern mit je zwei Ein- und Ausgangskanten bestehen. Damit besteht jede Ebene i aus insgesamt $n/2$ Schaltern, die in $2^{\log n - i}$ Knoten gruppiert sind. Dies ist in Abbildung 2.17 b) für einen Fat-Tree mit vier Ebenen skizziert, wobei nur die inneren Schalterknoten, nicht aber die die Prozessoren repräsentierenden Blattknoten dargestellt sind.

2.6 Routing- und Switching-Strategien

Direkte und indirekte Verbindungsnetzwerke bilden die physikalische Grundlage zum Verschicken von Nachrichten zwischen Prozessoren bei Systemen mit physikalisch verteiltem Speicher oder für den Speicherzugriff bei Systemen mit gemeinsamem Speicher. Besteht zwischen zwei Prozessoren keine direkte Punkt-zu-Punkt-Verbindung und soll eine Nachricht von einem Prozessor zum anderen Prozessor geschickt werden, muss ein Pfad im Netzwerk für die Nachrichtenübertragung gewählt werden. Dies ist sowohl bei direkten als auch bei indirekten Netzwerken der Fall.

2.6.1 Routingalgorithmen

Ein **Routingalgorithmus** bestimmt einen Pfad im Netzwerk, über den eine Nachricht von einem Sender A an einen Empfänger B geschickt werden soll. Üblicherweise ist eine topologiespezifische Vorschrift gegeben, die an jedem Zwischenknoten auf dem Pfad vom Sender zum Ziel angibt, zu welchen Folgeknoten die zu transportierende Nachricht weitergeschickt werden soll. Hierbei bezeichnen A und B zwei Knoten im Netzwerk (bzw. die zu den Knoten im Netzwerk gehörenden Verarbeitungseinheiten).

Üblicherweise befinden sich mehrere Nachrichtenübertragungen im Netz, so dass ein Routingalgorithmus eine gleichmäßige Auslastung der Leitungen im Netzwerk erreichen und Deadlockfreiheit garantieren sollte. Eine Menge von Nachrichten befindet sich in einer **Deadlocksituation**, wenn jede dieser Nachrichten jeweils über eine Verbindung weitergeschickt werden soll, die von einer anderen Nachricht derselben Menge gerade benutzt wird. Ein Routingalgorithmus wählt nach Möglichkeit von den Pfaden im Netzwerk, die Knoten A und B verbinden, denjenigen aus, der die geringsten Kosten verursacht. Die Kommunikationskosten, also die Zeit zwischen dem Absetzen einer Nachricht bei A und dem Ankommen bei B , hängen nicht nur von der Länge eines Pfades ab, sondern auch von der Belastung der Leitungen durch andere Nachrichten. Bei der Auswahl eines Routingpfades werden also die folgenden Punkte berücksichtigt:

- *Topologie:* Die Topologie des zugrunde liegenden Netzwerkes bestimmt die Pfade, die den Sender A mit Empfänger B verbinden und damit zum Versenden prinzipiell in Frage kommen.
- *Netzwerk-Contention bei hohem Nachrichtenaufkommen:* **Contention** liegt vor, wenn zwei oder mehrere Nachrichten zur gleichen Zeit über dieselbe Verbindung geschickt werden sollen und es durch die konkurrierenden Anforderungen zu Verzögerungen bei der Nachrichtenübertragung kommt.
- *Vermeidung von Staus bzw. Congestion.* **Congestion** entsteht, falls zu viele Nachrichten auf eine beschränkte Ressource (also Verbindungsleitung oder Puffer) treffen, so dass Puffer überfüllt werden und es dazu kommt,

dass Nachrichten weggeworfen werden. Im Unterschied zu Contention treten also bei Congestion so viele Nachrichten auf, dass das Nachrichtenaufkommen nicht mehr bewältigt werden kann [123].

Routingalgorithmen werden in verschiedensten Ausprägungen vorgeschlagen. Eine Klassifizierung bzgl. der Pfadlänge unterscheidet **minimale** Routingalgorithmen und **nichtminimale** Routingalgorithmen. Minimale Routingalgorithmen wählen für eine Nachrichtenübertragung immer den kürzesten Pfad aus, so dass die Nachricht durch Verschieben über jede Einzelverbindung des Pfades näher zum Zielknoten gelangt, aber die Gefahr von Staus gegeben ist. Nichtminimale Routingalgorithmen verschicken Nachrichten über nichtminimale Pfade, wenn die Netzwerkauslastung dies erforderlich macht. Die Länge eines Pfades muss je nach Switching-Technik (vgl. Abschnitt 2.6.2) zwar nicht direkt Auswirkungen auf die Kommunikationszeit haben, kann aber indirekt zu mehr Möglichkeiten für Contention oder Congestion führen. Ist das Netzwerk sehr belastet, muss aber der kürzeste Pfad zwischen Knoten *A* und *B* nicht der beste sein.

Eine weitere Klassifizierung ergibt sich durch Unterscheidung von **deterministischen** Routingalgorithmen und **adaptiven** Routingalgorithmen. Deterministisches Routing legt einen eindeutigen Pfad zur Nachrichtenübermittlung nur in Abhängigkeit von Sender und Empfänger fest. Die Auswahl des Pfades kann *quellenbasiert*, also nur durch den Sendeknoten, oder *verteilt* an den Zwischenknoten vorgenommen werden. Deterministisches Routing kann zu ungleichmäßiger Netzauslastung führen. Ein Beispiel für deterministisches Routing ist das **dimensionsgeordnete Routing** (engl. *dimension ordered routing*), das den Routing-Pfad entsprechend der Position von Quell- und Zielknoten und der Reihenfolge der Dimensionen der zugrunde liegenden Topologie auswählt. Adaptives Routing hingegen nutzt Auslastungsinformationen zur Wahl des Pfades aus, um Contention zu vermeiden. Bei adaptivem Routing werden mehrere mögliche Pfade zwischen zwei Knoten zum Nachrichtenaustausch bereitgestellt, wodurch nicht nur eine *größere Fehlertoleranz* für den möglichen Ausfall einzelner Verbindungen erreicht wird, sondern auch eine *gleichmäßigere* Auslastung des Netzwerkes. Auch bei adaptiven Routingalgorithmen wird zwischen minimalen und nichtminimalen Algorithmen unterschieden. Insbesondere für minimale adaptive Routingalgorithmen wird das Konzept von **virtuellen Kanälen** verwendet, auf die wir weiter unten eingehen. Routingalgorithmen werden etwa im Übersichtartikel [111] vorgestellt, siehe auch [31, 85, 101]. Wir stellen im Folgenden eine Auswahl von Routingalgorithmen vor.

Dimensionsgeordnetes Routing.

XY-Routing in einem 2-dimensionalen Gitter. XY-Routing ist ein dimensionsgeordneter Routingalgorithmus für zweidimensionale Gittertopologien. Die Positionen der Knoten in der Gittertopologie werden mit *X*- und *Y*-Koordinaten bezeichnet, wobei die *X*-Koordinate der horizontalen und die

Y -Koordinate der vertikalen Ausrichtung entspricht. Zum Verschicken einer Nachricht von Quellknoten A mit Position (X_A, Y_A) zu Zielknoten B mit Position (X_B, Y_B) wird die Nachricht so lange in (positive oder negative) X -Richtung geschickt, bis die X -Koordinate X_B von Knoten B erreicht ist. Anschließend wird die Nachricht in Y -Richtung geschickt, bis die Y -Koordinate Y_B erreicht ist. Die Länge der Pfade ist $|X_A - X_B| + |Y_A - Y_B|$. Der Routingalgorithmus ist also deterministisch und minimal.

E-Cube-Routing für den k -dimensionalen Hyperwürfel. In einem k -dimensionalen Würfel ist jeder der $n = 2^k$ Knoten direkt mit k physikalischen Nachbarn verbunden. Wird jedem Knoten, wie in Abschnitt 2.5.2 eingeführt, ein binäres Wort der Länge k als Namen zugeordnet, so ergeben sich die Namen der k physikalischen Nachbarn eines Knotens genau durch Invertierung eines der k Bits seines Namens. Dimensionsgerichtetes Routing für den k -dimensionalen Würfel [158] benutzt die k -Bitnamen von Sender und Empfänger und von dazwischenliegenden Knoten zur Bestimmung des Routing-Pfades. Soll eine Nachricht von Sender A mit Bitnamen $\alpha = \alpha_0 \dots \alpha_{k-1}$ an Empfänger B mit Bitnamen $\beta = \beta_0 \dots \beta_{k-1}$ geschickt werden, so wird beginnend bei A nacheinander ein Nachfolgerknoten entsprechend der Dimension gewählt, zu dem die Nachricht im nächsten Schritt geschickt werden soll. Ist A_i mit Bitdarstellung $\gamma = \gamma_0 \dots \gamma_{k-1}$ der Knoten auf dem Routing-Pfad $A = A_0, A_1, \dots, A_l = B$, von dem aus die Nachricht im nächsten Schritt weitergeleitet werden soll, so:

- berechnet A_i das k -Bitwort $\gamma \oplus \beta$, wobei der Operator $\oplus$ das bitweise ausschließende Oder (d.h. $0 \oplus 0 = 0, 0 \oplus 1 = 1, 1 \oplus 0 = 1, 1 \oplus 1 = 0$) bezeichnet, und
- schickt die Nachricht in Richtung der Dimension d , wobei d die am weitesten rechts liegende Position von $\gamma \oplus \beta$ ist, die den Wert 1 hat. Den zugehörigen Knoten A_{i+1} auf dem Routingpfad erhält man durch Invertierung des d -ten Bits in γ , d.h. der Knoten A_{i+1} hat den k -Bit-Namen $\delta = \delta_0 \dots \delta_{k-1}$ mit $\delta_j = \gamma_j$ für $j \neq d$ und $\delta_d = \bar{\gamma}_d$ (Bitumkehrung). Wenn $\gamma \oplus \beta = 0$ ist, ist der Zielknoten erreicht.

Beispiel: Um eine Nachricht von A mit Bitnamen $\alpha = 010$ nach B mit Bitnamen $\beta = 111$ zu schicken, wird diese also zunächst in Richtung Dimension $d = 2$ nach A_1 mit Bitnamen 011 geschickt (da $\alpha \oplus \beta = 101$ gilt) und dann in Richtung Dimension $d = 0$ zu β (da $011 \oplus 111 = 100$ gilt). $\square$

Deadlockgefahr bei Routingalgorithmen. Befinden sich mehrere Nachrichten im Netzwerk, was der Normalfall ist, so kann es zu Deadlocksituationen kommen, in denen der Weitertransport einer Teilmenge von Nachrichten für immer blockiert wird. Dies kann insbesondere dann auftreten, wenn Ressourcen im Netzwerk nur von jeweils einer Nachricht genutzt werden können. Werden z.B. die Verbindungskanäle zwischen zwei Knoten jeweils nur einer Nachricht zugeteilt und wird ein Verbindungskanal nur frei-

gegeben, wenn der folgende Verbindungskanal für den Weitertransport zugeteilt werden kann, so kann es durch wechselseitiges Anfordern von Verbindungskanälen zu einem solchen Deadlock kommen. Genau dieses Zustandekommen von Deadlocksituationen kann durch geeignete Routingalgorithmen vermieden werden. Andere Deadlocksituationen, die durch beschränkte Ein- und Ausgabepuffer der Verbindungskanäle oder ungünstige Reihenfolgen von Sende- und Empfangsbefehlen entstehen können, werden in den Abschnitten über Switching bzw. Message-Passing-Programmierung betrachtet, siehe Abschnitt 2.6.2 und Kapitel 5.

Zum Beweis der Deadlockfreiheit von Routingalgorithmen werden mögliche Abhängigkeiten zwischen Verbindungskanälen betrachtet, die durch beliebige Nachrichtenübertragungen entstehen können. Eine *Abhängigkeit zwischen den Verbindungskanälen* l_1 und l_2 besteht, falls es durch den Routingalgorithmus möglich ist, einen Pfad zu wählen, der eine Nachricht über Verbindung l_1 und direkt danach über Verbindung l_2 schickt. Diese Abhängigkeit zwischen Verbindungskanälen kann im **Kanalabhängigkeitsgraph** (engl. *channel dependency graph*) ausgedrückt werden, der Verbindungskanäle als Knoten darstellt und für jede Abhängigkeit zwischen Kanälen eine Kante enthält. Enthält dieser Kanalabhängigkeitsgraph keine Zyklen, so ist der entsprechende Routingalgorithmus auf der gewählten Topologie deadlockfrei, da kein Kommunikationsmuster eines Deadlocks entstehen kann.

Für Topologien, die keine Zyklen enthalten, ist jeder Kanalabhängigkeitsgraph zyklensfrei, d.h. jeder Routingalgorithmus auf einer solchen Topologie ist deadlockfrei. Für Netzwerke mit Zyklen muss der Kanalabhängigkeitsgraph analysiert werden. Wir zeigen im Folgenden, dass das oben eingeführte XY-Routing für zweidimensionale Gitter mit bidirektionalen Verbindungen deadlockfrei ist.

Deadlockfreiheit für XY-Routing. Der für das XY-Routing resultierende Kanalabhängigkeitsgraph enthält für jede unidirektionale Verbindung des zweidimensionalen $n_x \times n_y$ -Gitters einen Knoten, also zwei Knoten für jede bidirektionale Kante des Gitters. Es gibt eine Abhängigkeit von Verbindung u zu Verbindung v , falls sich v in der gleichen horizontalen oder vertikalen Ausrichtung oder in einer 90-Grad-Drehung nach oben oder unten an u anschließt. Zum Beweis der Deadlockfreiheit werden alle unidirektionalen Verbindungen des Gitters auf folgende Weise nummeriert.

- Horizontale Kanten zwischen Knoten mit Position (i, y) und Knoten mit Position $(i + 1, y)$ erhalten die Nummer $i + 1$, $i = 0, \dots, n_x - 2$, und zwar für jede y -Position. Die entgegengesetzten Kanten von $(i + 1, y)$ nach (i, y) erhalten die Nummer $n_x - 1 - (i + 1) = n_x - i - 2$, $i = 0, \dots, n_x - 2$. Die Kanten in aufsteigender x -Richtung sind also aufsteigend mit $1, \dots, n_x - 1$, die Kanten in absteigender x -Richtung sind aufsteigend mit $0, \dots, n_x - 2$ nummeriert.

- Die vertikalen Kanten von (x, j) nach $(x, j + 1)$ erhalten die Nummern $j + n_x$, $j = 0, \dots, n_y - 2$ und die entgegengesetzten Kanten erhalten die Nummern $n_x + n_y - (j + 1)$.

Abbildung 2.18 zeigt ein 3×3 -Gitter und den zugehörigen Kanalabhängigkeitsgraphen bei Verwendung von XY-Routing, wobei die Knoten des Graphen mit den Nummern der zugehörigen Netzwerkkanten bezeichnet sind. Da alle Kanten im Kanalabhängigkeitsgraphen von einer Netzwerkkante mit einer niedrigeren Nummer zu einer Netzwerkkante mit einer höheren Nummer führen, kann eine Verzögerung einer Übertragung entlang eines Routingpfades nur dann auftreten, wenn die Nachricht nach Übertragung über eine Kante v mit Nummer i auf die Freigabe einer nachfolgenden Kante w mit Nummer $j > i$ wartet, da diese Kante gerade von einer anderen Nachricht verwendet wird (Verzögerungsbedingung). Zum Auftreten eines Deadlocks wäre es also erforderlich, dass es eine Menge von Nachrichten $N_1, \dots, N_k$ und Netzwerkkanten $n_1, \dots, n_k$ gibt, so dass jede Nachricht N_i für $1 \leq i < k$ gerade Kante n_i für die Übertragung verwendet und auf die Freigabe von Kante n_{i+1} wartet, die gerade von Nachricht N_{i+1} zur Übertragung verwendet wird. Außerdem überträgt N_k gerade über Kante n_k und wartet auf die Freigabe von n_1 durch N_1 . Wenn $n()$ die oben für die Netzwerkkanten eingeführte Numerierung ist, gilt wegen der Verzögerungsbedingung

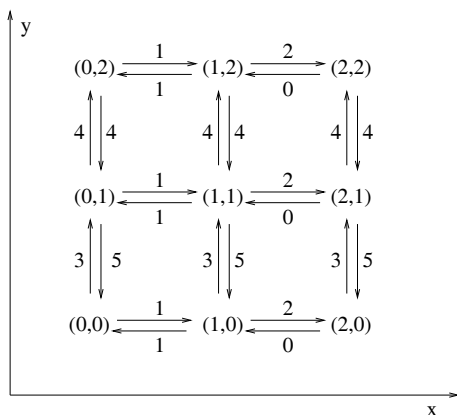
$$n(n_1) < n(n_2) < \dots < n(n_k) < n(n_1).$$

Da dies ein Widerspruch ist, kann kein Deadlock auftreten. Jeder mögliche XY-Routing-Pfad besteht somit aus einer Folge von Kanten mit *aufsteigender* Kantenummerierung. Alle Kanten im Kanalabhängigkeitsgraphen führen zu einer höher nummerierten Verbindung. Es kann somit *keinen* Zyklus im Kantenabhängigkeitsgraphen geben.

Ein ähnliches Vorgehen kann verwendet werden, um die Deadlockfreiheit von E-Cube-Routing zu beweisen, vgl. [33].

Quellenbasiertes Routing. Ein weiterer deterministischer Routingalgorithmus ist das quellenbasierte Routing, bei dem der Sender den gesamten Pfad zur Nachrichtenübertragung auswählt. Für jeden Knoten auf dem Pfad wird der zu wählende Ausgabekanal festgestellt und die Folge der nacheinander zu wählenden Ausgabekanäle $a_0, \dots, a_{n-1}$ wird als Header der eigentlichen Nachricht angefügt. Nachdem die Nachricht einen Knoten passiert hat, wird die Routinginformation im Header der den Knoten verlassenden Nachricht aktualisiert, indem der gerade passierte Ausgabekanal aus dem Pfad entfernt wird.

Tabellenorientiertes Routing. (engl. *table lookup routing*). Beim tabellenorientierten Routing enthält jeder Knoten des Netzwerkes eine Routingtabelle, die für jede Zieladresse den zu wählenden Ausgabekanal bzw. den nächsten Knoten enthält. Kommt eine Nachricht in einem Knoten an, so wird die Zielinformation betrachtet und in der Routingtabelle nachgesehen, wohin die Nachricht weiter zu verschicken ist.

2-dimensionales Gitter mit 3×3 Knoten

Kanalabhängigkeitsgraph

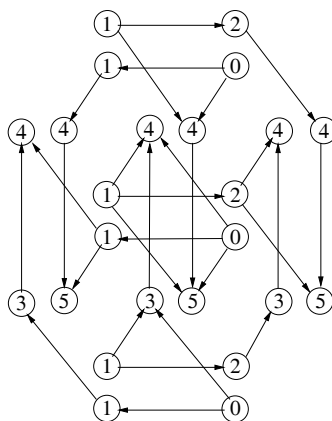
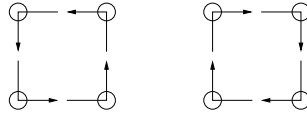


Abb. 2.18. 3×3 -Gitter und zugehöriger Kanalabhängigkeitsgraph bei Verwendung von XY-Routing.

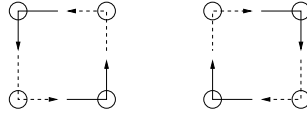
Turn-Modell. Das Turn-Modell (von [60] dargestellt in [111]) versucht Deadlocks durch geschickte Wahl erlaubter Richtungswechsel zu vermeiden. Die Ursache für das Auftreten von Deadlocks besteht darin, dass Nachrichten ihre Übertragungsrichtung so ändern, dass bei ungünstigem Zusammenreffen ein zyklisches Warten entsteht. Deadlocks können vermieden werden, indem gewisse Richtungsänderungen untersagt werden. Ein Beispiel ist das XY-Routing, bei dem alle Richtungsänderungen von vertikaler Richtung in horizontale Richtung ausgeschlossen sind. Von den insgesamt acht möglichen Richtungsänderungen in einem zweidimensionalen Gitter, sind also nur vier Richtungsänderungen erlaubt, vgl. Abbildung 2.19. Diese restlichen vier möglichen Richtungsänderungen erlauben *keinen Zyklus*, schließen Deadlocks also aus, machen allerdings auch adaptives Routing unmöglich. Im Turn-Modell für n -dimensionale Gitter und allgemeine k -fache n -Würfel wird eine minimale Anzahl von Richtungsänderungen ausgewählt, bei deren Ausschluss bei der Wahl eines Routingpfades die Bildung von Zyklen vermieden wird. Konkrete Beispiele sind das West-First-Routing bei zweidimensionalen Gittern oder das P -cube-Routing bei n -dimensionalen Hyperwürfeln.

Beim **West-First-Routing** für zweidimensionale Gitter werden nur zwei der insgesamt acht möglichen Richtungsänderungen ausgeschlossen, und zwar die Richtungsänderungen nach Westen, also nach links, so dass nur noch die Richtungsänderungen, die in Abbildung 2.19 angegeben sind, erlaubt sind. Routingpfade werden so gewählt, dass die Nachricht zunächst nach Westen (d.h. nach links) geschickt wird, bis mindestens die gewünschte x -Koordinate erreicht ist, und dann adaptiv nach Süden (d.h. unten), nach Osten (d.h. rechts) oder Norden (d.h. oben). Beispiele von Routingpfaden sind in Ab-

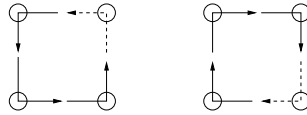
Mögliche Richtungswechsel im zwei-dimensionalen Gitter



Richtungswechsel bei XY-Routing



Richtungswechsel bei West-First-Routing



Erlaubte Richtungswechsel

Nicht erlaubte Richtungswechsel



Abb. 2.19. Illustration der Richtungswechsel beim Turn-Modell im zwei-dimensionalen Gitter mit Darstellung aller Richtungswechsel und der erlaubten Richtungswechsel bei XY-Routing bzw. West-First-Routing.

bildung 2.20 gegeben [111]. West-First-Routing ist deadlockfrei, da Zyklen vermieden werden. Bei der Auswahl von minimalen Routingpfaden ist der Algorithmus nur dann adaptiv, falls das Ziel im Osten (d.h. rechts) liegt. Bei Verwendung nichtminimaler Routingpfade ist der Algorithmus immer adaptiv.

Beim **P-cube-Routing** für den n -dimensionalen Hyperwürfel werden für einen Sender A mit dem n -Bitnamen $\alpha_0 \dots \alpha_{n-1}$ und einen Empfänger B mit dem n -Bitnamen $\beta_0 \dots \beta_{n-1}$ die unterschiedlichen Bits dieser beiden Namen betrachtet. Die Anzahl der unterschiedlichen Bits entspricht der Hammingdistanz von A und B und ist die Mindestlänge eines möglichen Routingpfades. Die Menge $E = \{i \mid \alpha_i \neq \beta_i, i = 0, \dots, n-1\}$ der Positionen der unterschiedlichen Bits wird in zwei Mengen zerlegt und zwar in $E_0 = \{i \in E \mid \alpha_i = 0 \text{ und } \beta_i = 1\}$ und $E_1 = \{i \in E \mid \alpha_i = 1 \text{ und } \beta_i = 0\}$. Das Verschicken einer Nachricht von A nach B wird entsprechend der Mengen in zwei Phasen unterteilt. Zuerst wird die Nachricht über die Dimensionsrichtungen in E_0 geschickt, danach erst über die Dimensionsrichtungen in E_1 .

Virtuelle Kanäle. Insbesondere für minimale adaptive Routingalgorithmen wird das Konzept von *virtuellen Kanälen* verwendet, da für manche Verbindungen mehrere Kanäle zwischen benachbarten Knoten benötigt werden. Da die Realisierungen mehrerer physikalischer Verbindungen zu teuer ist, werden

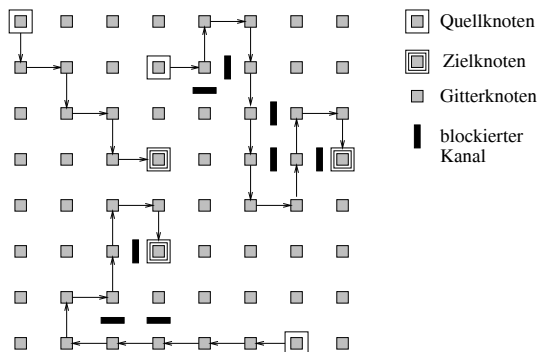


Abb. 2.20. Illustration der Pfadwahl beim West-First-Routing in einem 8×8 -Gitter. Die als blockiert gekennzeichneten Kanäle werden von anderen Nachrichten verwendet und stehen daher nicht für die Nachrichtenübertragung zur Verfügung. Einer der dargestellten Pfade ist minimal, die anderen beiden sind nicht-minimal, da bestimmte Kanäle blockiert sind.

mehrere virtuelle Kanäle eingeführt, die sich eine physikalische Verbindung teilen. Für jeden virtuellen Kanal werden separate Puffer zur Verfügung gestellt. Die Zuteilung der physikalischen Verbindungen zu den virtuellen Verbindungen sollte *fair* erfolgen, d.h. jede virtuelle Verbindung sollte immer wieder genutzt werden können.

Der folgende *minimale adaptive Routingalgorithmus* benutzt virtuelle Kanäle und zerlegt das gegebene Netzwerk in logische Teilnetzwerke. Der Zielknoten einer Nachricht bestimmt, durch welches Teilnetz die Nachricht transportiert wird. Wir demonstrieren die Arbeitsweise für ein zweidimensionales Gitter. Ein zweidimensionales Gitter wird in zwei Teilnetze zerlegt, und zwar in ein $+X$ -Teilnetz und ein $-X$ -Teilnetz, siehe Abbildung 2.21. Jedes Teilnetz enthält alle Knoten, aber nur einen Teil der virtuellen Kanäle. Das $+X$ -Teilnetz enthält in vertikaler Richtung Verbindungen zwischen allen benachbarten Knoten, in horizontaler Richtung aber nur Kanten in positiver Richtung. Das $-X$ -Teilnetz enthält ebenfalls Verbindungen zwischen allen vertikal benachbarten Knoten – was durch Verwendung von virtuellen Kanälen möglich ist – sowie alle horizontalen Kanten in negativer Richtung. Nachrichten von Knoten A mit x -Koordinate x_A nach Knoten B mit x -Koordinate x_B werden im $+X$ -Netz verschickt, wenn $x_A < x_B$ ist. Nachrichten von Knoten A nach B mit $x_A > x_B$ werden im $-X$ -Netz verschickt. Für $x_A = x_B$ kann ein beliebiges Teilnetz verwendet werden. Die genaue Auswahl kann anhand der Auslastung des Netzwerkes getroffen werden. Dieser minimale adaptive Routingalgorithmus ist deadlockfrei [111]. Für andere Topologien wie den Hyperwürfel oder den Torus können mehr zusätzliche Leitungen nötig sein, um Deadlockfreiheit zu gewährleisten, vgl. [111].

Ein *nichtminimaler* adaptiver Routingalgorithmus kann Nachrichten auch über längere Pfade verschicken, falls kein minimaler Pfad zur Verfügung steht.

2-dimensionales Gitter mit virtuellen Kanälen in y-Richtung

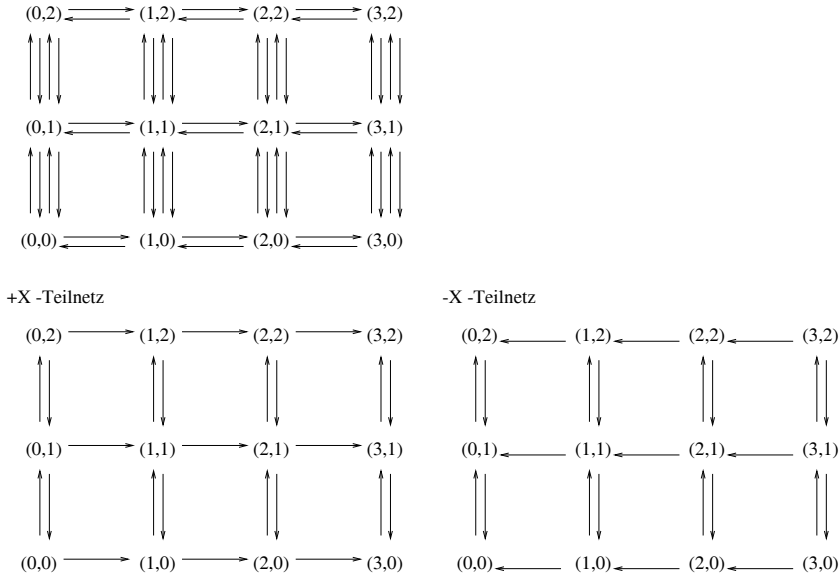


Abb. 2.21. Zerlegung eines zweidimensionalen Gitters mit virtuellen Kanälen in ein +X-Teilnetz und ein -X-Teilnetz für die Anwendung eines minimalen adaptiven Routingalgorithmus.

Der **statische umgekehrt-dimensionsgeordnete Routingalgorithmus** (engl. *dimension reversal routing algorithm*) kann auf beliebige Gittertopologien und k -fache d -Würfel angewendet werden. Der Algorithmus benutzt r Paare von (virtuellen) Kanälen zwischen jedem durch einen physikalischen Kanal miteinander verbundenen Knotenpaar und zerlegt das Netzwerk in r Teilnetzwerke, wobei das i -te Teilnetzwerk für $i = 0, \dots, r-1$ alle Knoten und die i -ten Verbindungen zwischen den Knoten umfasst. Jeder Nachricht wird zusätzlich eine Klasse c zugeordnet, die zu Anfang auf $c = 0$ gesetzt wird und die im Laufe der Nachrichtenübertragung Klassen $c = 1, \dots, r-1$ annehmen kann. Eine Nachricht mit Klasse $c = i$ kann im i -ten Teilnetz in jede Richtung transportiert werden, wobei aber die verschiedenen Dimensionen in aufsteigender Reihenfolge durchlaufen werden müssen. Eine Nachricht kann aber auch entgegen der Dimensionsordnung, d.h. von einem höher-dimensionalen Kanal zu einem niedriger-dimensionalen Kanal transportiert werden. In diesem Fall wird die Klasse der Nachricht um 1 erhöht (umgekehrte Dimensionsordnung). Der Parameter r begrenzt die Möglichkeiten der Dimensionsumkehrung. Ist die maximale Klasse erreicht, so wird der Routing-Pfad entsprechend dem dimensionsgeordneten Routing beendet.

Routing im Omega-Netzwerk. Das in Abschnitt 2.5.4 beschriebene Omega-Netzwerk ermöglicht ein Weiterleiten von Nachrichten mit Hilfe eines ver-

teilten Kontrollschemas, in dem jeder Schalter die Nachricht *ohne* Koordination mit anderen Schaltern weiterleiten kann. Zur Beschreibung des Routingalgorithmus ist es günstig, die n Eingangs- und Ausgangskanäle mit Bitnamen der Länge $\log n$ zu benennen [101]. Zum Weiterleiten einer Nachricht vom Eingangskanal mit Bitnamen α zum Ausgangskanal mit Bitnamen β betrachtet der die Nachricht erhaltende Schalter auf Stufe k , $k = 0, \dots, \log n - 1$, das k -te Bit β_k (von links) des Zielnamens β und wählt den Ausgang für das Weitersenden anhand folgender Regel aus:

1. Ist das k -te Bit $\beta_k = 0$, so wird die Nachricht über den oberen Ausgang des Schalters weitergeleitet.
2. Ist das k -te Bit $\beta_k = 1$, so wird die Nachricht über den unteren Ausgang des Schalters weitergeleitet.

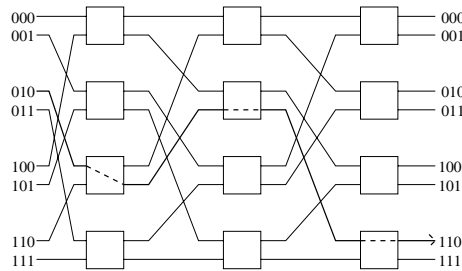


Abb. 2.22. 8×8 Omega-Netzwerk mit Pfad von 010 nach 110 [11].

In Abbildung 2.22 ist der Pfad der Nachrichtenübertragung vom Eingang $\alpha = 010$ zum Ausgang $\beta = 110$ angegeben. Maximal können bis zu n Nachrichten von verschiedenen Eingängen zu verschiedenen Ausgängen parallel zueinander durch das Omega-Netzwerk geschickt werden. Ein Beispiel für eine parallele Nachrichtenübertragung mit $n = 8$ im 8×8 -Omega-Netzwerk ist durch die Permutation

$$\pi^8 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 7 & 3 & 0 & 1 & 2 & 5 & 4 & 6 \end{pmatrix}$$

gegeben, die angibt, dass von Eingang i ($i = 0, \dots, 7$) zum Ausgang $\pi^8(i)$ jeweils eine Nachricht gesendet wird. Die entsprechende parallele Schaltung der 8 Pfade, jeweils von i nach $\pi^8(i)$, ist durch die Schaltereinstellung in Abbildung 2.23 realisiert.

Viele solcher durch Permutation $\pi^8 : \{0, \dots, n-1\} \rightarrow \{0, \dots, n-1\}$ gegebener gewünschter Verbindungen sind jedoch nicht in einem Schritt, also parallel zueinander zu realisieren, da es zu **Konflikten** im Netzwerk kommt. So führen zum Beispiel die beiden Nachrichtenübersendungen von $\alpha_1 = 010$ zu $\beta_1 = 110$ und von $\alpha_2 = 000$ zu $\beta_2 = 111$ in einem 8×8 Omega-Netzwerk

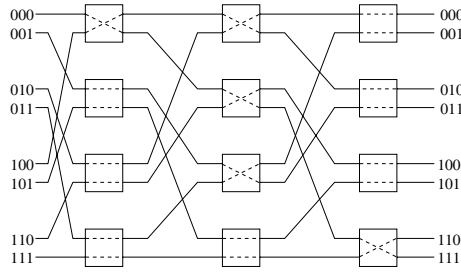


Abb. 2.23. 8×8 Omega-Netzwerk mit Schalterstellungen zur Realisierung von π^8 aus dem Text.

zu einem Konflikt. Konflikte dieser Art können nicht aufgelöst werden, da es zu einem beliebigen Paar (α, β) von Eingabekante und Ausgabekante jeweils nur genau eine mögliche Verbindung gibt und somit kein Ausweichen möglich ist. Netzwerke mit dieser Eigenschaft heißen auch **blockierende Netzwerke**. Konflikte in blockierenden Netzwerken können jedoch durch mehrere Läufe durch das Netzwerk aufgelöst werden. Von den insgesamt $n!$ möglichen Permutationen (bzw. denen durch sie jeweils dargestellten gewünschten n Verbindungen von Eingangskanälen zu Ausgangskanälen) können nur $n^{n/2}$ in einem Durchlauf parallel zueinander, also ohne Konflikte realisiert werden. Denn da es pro Schalter 2 mögliche Schalterstellungen gibt, ergibt sich für die insgesamt $n/2 \cdot \log n$ Schalter des Omega-Netzwerkes eine Anzahl von $2^{n/2 \cdot \log n} = n^{n/2}$ mögliche Schaltungen des Gesamtnetzwerkes, die jeweils einer Realisierung von n parallelen Pfaden entsprechen.

Weitere blockierende Netzwerke sind das Butterfly- oder Banyan-Netzwerk, das Baseline-Netzwerk und das Delta-Netzwerk [101]. Im Unterschied dazu handelt es sich beim Benes-Netzwerk um ein *nicht-blockierendes* Netzwerk, das es ermöglicht, unterschiedliche Verbindungen zwischen einer Eingangskante und einer Ausgangskante herzustellen. Für jede Permutation $\pi : \{0, \dots, n-1\} \rightarrow \{0, \dots, n-1\}$ ist es möglich, eine Schaltung des Benes-Netzwerkes zu finden, die Verbindungen von Eingang i zu Ausgang $\pi(i)$, $i = 0, \dots, n-1$, gleichzeitig realisiert, so dass die n Kommunikationen parallel zueinander stattfinden können. Dies kann durch Induktion über die Dimension k des Netzwerkes bewiesen werden, vgl. dazu [101]. Ein Beispiel für die Realisierung der Permutation

$$\pi^8 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 5 & 3 & 4 & 7 & 0 & 1 & 2 & 6 \end{pmatrix}$$

ist in Abbildung 2.24 gegeben, vgl. [101]. Weitere Details über Routingtechniken in indirekten Netzwerken sind vor allem in [101] zu finden.

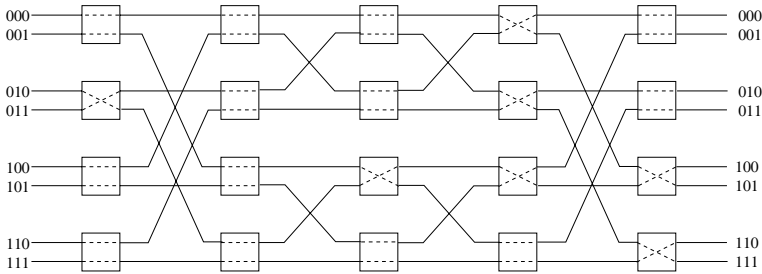


Abb. 2.24. 8×8 Benes-Netzwerk mit Schalterstellungen zur Realisierung von π^8 aus dem Text.

2.6.2 Switching

Eine **Switching-Strategie** oder Switching-Technik legt fest, *wie* eine Nachricht den vom Routingalgorithmus ausgewählten Pfad von einem Sendeknoten zum Zielknoten durchläuft. Genauer gesagt, wird durch eine Switching-Strategie festgelegt

- *ob und wie* eine Nachricht in Stücke, z.B. in Pakete oder *flits* (für engl. *flow control units*), zerlegt wird,
- *wie* der Übertragungspfad vom Sendeknoten zum Zielknoten allokiert wird (vollständig oder teilweise) und
- *wie* Nachrichten (oder Teilstücke von Nachrichten) vom Eingabekanal eines Schalters oder Routers auf den Ausgabekanal gelegt werden. Der Routingalgorithmus legt dann fest, *welcher* Ausgabekanal zu wählen ist.

Die benutzte Switching-Strategie hat einen großen Einfluss auf die Zeit, die für eine Nachrichtenübertragung zwischen zwei Knoten benötigt wird. Bevor wir auf Switching-Strategien und den jeweils benötigten Zeitaufwand eingehen, betrachten wir zunächst den Zeitaufwand, der für eine Nachrichtenübertragung zwischen zwei benachbarten Netzwerkknoten benötigt wird, wenn die Nachrichtenübertragung also über nur *eine* Verbindungsleitung erfolgt.

Nachrichtenübertragung benachbarter Prozessoren. Eine Nachrichtenübertragung zwischen zwei Prozessoren wird durch eine Folge von in Software realisierten Schritten (Protokoll genannt) realisiert. Sind die beiden Prozessoren durch eine bidirektionale Verbindungsleitung miteinander verbunden, so kann das im Folgenden skizzierte Beispielprotokoll verwendet werden. Zum Senden einer Nachricht werden vom sendenden Prozessor folgende Programmschritte ausgeführt:

1. Die Nachricht wird in einen Systempuffer kopiert.
2. Das Betriebssystem berechnet eine **Prüfsumme** (engl. *checksum*), fügt einen **Header** mit dieser Prüfsumme und Informationen zur Nachrichtenübertragung an die Nachricht an und startet einen Timer, der die Zeit misst, die die Nachricht bereits unterwegs ist.

3. Das Betriebssystem sendet die Nachricht zur Netzwerkschnittstelle und veranlasst die hardwaremäßige Übertragung.

Zum Empfangen einer Nachricht werden folgende Programmschritte ausgeführt:

1. Das Betriebssystem kopiert die Nachricht aus der Hardwareschnittstelle zum Netzwerk in einen Systempuffer.
2. Das Betriebssystem berechnet die Prüfsumme der erhaltenen Daten. Stimmt diese mit der beigefügten Prüfsumme überein, sendet der Empfänger eine Empfangsbestätigung (engl. *acknowledgement*) zum Sender. Stimmt die Prüfsumme **nicht** mit der beigefügten Prüfsumme überein, so wird die Nachricht verworfen und es wird angenommen, dass der Sender nach Ablauf einer dem Timer vorgegebenen Zeit die Nachricht nochmals sendet.
3. War die Prüfsumme korrekt, so wird die Nachricht vom Systempuffer in den Adressbereich des Anwendungsprogramms kopiert und dem Anwendungsprogramm wird ein Signal zum Fortfahren gegeben.

Nach dem eigentlichen Senden der Nachricht werden vom sendenden Prozessor folgende weitere Schritte ausgeführt:

1. Bekommt der Sender die Empfangsbestätigung, so wird der Systempuffer mit der Kopie der Nachricht freigegeben.
2. Bekommt der Sender vom Timer die Information, dass die Schranke der Übertragungszeit überschritten wurde, so wird die Nachricht erneut gesendet.

In diesem Protokoll wurde angenommen, dass das Betriebssystem die Nachricht im Systempuffer hält, um sie gegebenenfalls neu zu senden. Wird keine Empfangsbestätigung benötigt, kann ein Sender jedoch erneut eine weitere Nachricht versenden, ohne auf die Ankunft der zuvor gesendeten Nachricht beim Empfänger zu warten. In Protokollen können außer der Zuverlässigkeit in Form der Empfangsbestätigung auch weitere Aspekte berücksichtigt werden, wie etwa die Umkehrung von Bytes beim Versenden zwischen verschiedenartigen Knoten, das Verhindern einer Duplizierung von Nachrichten oder das Füllen des Empfangspuffers für Nachrichten. Das Beispielprotokoll ist ähnlich zum weit verbreiteten UDP-Transportprotokoll [98, 123].

Die Zeit für eine Nachrichtenübertragung setzt sich aus der Zeit für die eigentliche Übertragung der Nachricht über die Verbindungsleitung, also die Zeit im Netzwerk, und der Zeit zusammen, die die Softwareschritte des jeweils verwendeten Protokolls benötigen. Zur Beschreibung dieser Zeit für eine Nachrichtenübertragung, die auch **Latenz** genannt wird, werden die folgenden Maße verwendet:

- Die **Bandbreite** (engl. *bandwidth*) ist die maximale Frequenz, mit der Daten über eine Verbindungsleitung geschickt werden können. Die Einheit ist Bytes/Sekunde.

- Die **Bytetransferzeit** ist die Zeit, die benötigt wird, um ein Byte über die Verbindungsleitung zu schicken. Es gilt:

$$\text{Bytetransferzeit} = \frac{1}{\text{Bandbreite}}.$$

- Die **Übertragungszeit** (engl. *transmission time*) ist die Zeit, die gebraucht wird, um eine Nachricht über eine Verbindungsleitung zu schicken. Es gilt:

$$\text{Übertragungszeit} = \frac{\text{Nachrichtengröße}}{\text{Bandbreite}}.$$

- Die **Signalverzögerungszeit** (engl. *time of flight* oder *channel propagation delay*) bezeichnet die Zeit, die das erste Bit einer Nachricht benötigt, um beim Empfänger anzukommen.
- Die **Transportlatenz** ist die Zeit, die eine Nachricht für die Übertragung im Netzwerk verbringt. Es gilt:

$$\text{Transportlatenz} = \text{Signalverzögerungszeit} + \text{Übertragungszeit}.$$

- Der **Senderoverhead** oder **Startupzeit** ist die Zeit, die der Sender benötigt, um eine Nachricht zum Senden vorzubereiten, umfasst also das Anfügen von Header und Prüfsumme und die Ausführung des Routingalgorithmus.
- Der **Empfängeroverhead** ist die Zeit, die der Empfänger benötigt, um die Softwareschritte für das Empfangen einer Nachricht auszuführen.
- Der **Durchsatz** (engl. *throughput*) wird zur Bezeichnung der Netzwerkbandbreite genutzt, die bei einer bestimmten Anwendung erzielt wird.

Unter Benutzung der obigen Maße setzt sich die gesamte Latenz der Übertragung einer Nachricht folgendermaßen zusammen:

$$\begin{aligned} \text{Latenz} = & \text{Senderoverhead} + \text{Signalverzögerung} \\ & + \frac{\text{Nachrichtengröße}}{\text{Bandbreite}} + \text{Empfängeroverhead}. \end{aligned} \quad (2.1)$$

In einer solchen Formel wird nicht berücksichtigt, dass eine Nachricht evtl. mehrmals verschickt wird bzw. ob Contention oder Congestion im Netzwerk vorliegt. Die Leistungsparameter für eine Nachrichtenübermittlung auf einer Verbindungsleitung sind aus der Sicht des Senders, des Empfängers und des Netzwerkes in Abbildung 2.25 illustriert. Die Formel (2.1) kann vereinfacht werden, indem die konstanten Terme zusammengefasst werden. Es ergibt sich:

$$\text{Latenz} = \text{Overhead} + \frac{\text{Nachrichtengröße}}{\text{Bandbreite}}$$

mit einem konstanten Anteil **Overhead** und einem in der Nachrichtengröße linearen Anteil mit Faktor $\frac{1}{\text{Bandbreite}}$. Mit den Abkürzungen m für die Nachrichtengröße in Bytes, t_S für die den Overhead beschreibende Startupzeit und t_B für die Bytetransferzeit ergibt sich für die Latenz $T(m)$ in Abhängigkeit von der Nachrichtengröße m die Laufzeitformel

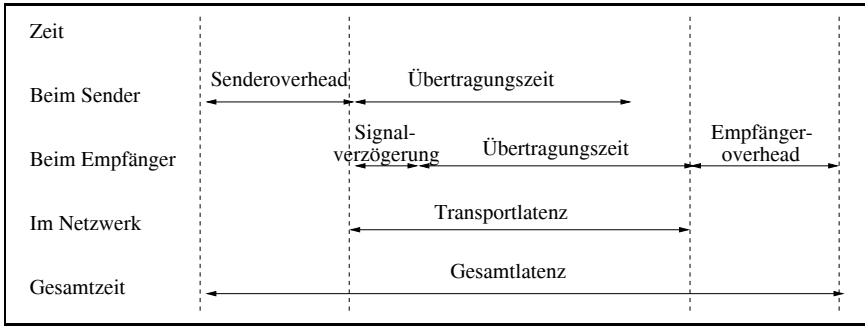


Abb. 2.25. Illustration zu Leistungsmaßen des Einzeltransfers zwischen benachbarten Knoten, siehe [75].

$$T(m) = t_S + t_B \cdot m. \quad (2.2)$$

Diese lineare Beschreibung des zeitlichen Aufwandes gilt für eine Nachrichtenübertragung zwischen zwei durch eine Punkt-zu-Punkt-Verbindung miteinander verbundene Knoten. Liegen zwei Knoten im Netzwerk *nicht* benachbart, so muss eine Nachricht zwischen den beiden Knoten über mehrere Verbindungsleitungen eines Pfades zwischen diesen beiden Knoten geschickt werden. Wie oben bereits erwähnt, kann dies durch verschiedene Switching-Strategien realisiert werden. Bei den Switching-Strategien werden u.a.

- Circuit-Switching,
- Paket-Switching mit Store-und-Forward-Routing,
- Virtuelles Cut-Through Routing und
- Wormhole Routing

unterschieden. Als Grundformen der Switching-Strategien kann man *Circuit-Switching* und *Paket-Switching* (engl. *packet switching*) ansehen [31, 111, 146].

Beim **Circuit-Switching** wird der gesamte Pfad vom Ausgangsknoten bis zum Zielknoten aufgebaut, d.h. die auf dem Pfad liegenden Switches, Prozessoren oder Router werden entsprechend geschaltet und exklusiv der zu übermittelnden Nachricht zur Verfügung gestellt, bis die Nachricht vollständig beim Zielknoten angekommen ist. Intern kann die Nachricht entsprechend der Übertragungsrate in Teilstücke unterteilt werden, und zwar in sogenannte **phits (physical units)**, die die Datenmenge bezeichnen, die pro Takt über eine Verbindung übertragen werden kann, bzw. die kleinste physikalische Einheit, die zusammen übertragen wird. Die Größe der *phits* wird im Wesentlichen durch die Anzahl der Bits bestimmt, die über einen physikalischen Kanal gleichzeitig übertragen werden können, und liegt typischerweise zwischen 1 und 64 Bits. Der Übertragungspfad wird durch das Versenden einer sehr kurzen Nachricht (*probe*) aufgebaut. Danach werden alle *phits* der Nachricht über diesen Pfad übertragen. Die Freigabe des Pfades geschieht

entweder durch das Endstück der Nachricht oder durch eine zurückgesendete Empfangsbestätigung.

Die Kosten für das Versenden der Kontrollnachricht zum Aufbau des Pfades der Länge l vom Sender zum Empfänger benötigt die Zeit $t_c \cdot l$, wobei t_c die Kosten zum Versenden der Kontrollnachricht je Verbindung sind, d.h. $t_c = t_B \cdot m_c$ mit m_c = Größe des Kontrollpaketes. Nach der Reservierung des Pfades braucht die Versendung der eigentlichen Nachricht der Größe m die Zeit $m \cdot t_B$, so dass die Gesamtkosten des Einzeltransfers einer Nachricht auf einem Pfad der Länge l mit Circuit-Switching

$$T_{cs}(m, l) = t_S + t_c \cdot l + t_B \cdot m \quad (2.3)$$

sind. Ist m_c klein gegenüber m , so entspricht dies ungefähr $t_S + t_B \cdot m$, also einer Laufzeitformel, die linear in m und unabhängig von l ist. Die Kosten für einen Einzeltransfer mit Circuit-Switching sind in Abbildung 2.27 a) illustriert.

Bei **Paket-Switching** wird eine Nachricht in eine Folge von Paketen unterteilt, die unabhängig voneinander über das Netzwerk vom Sender zum Empfänger transportiert werden. Bei Verwendung eines adaptiven Routing-Algorithmus können die Pakete einer Nachricht daher über unterschiedliche Pfade transportiert werden. Jedes **Paket** besteht aus drei Teilen, und zwar einem Header, der Routing- und Kontrollinformationen enthält, einem Datenteil, der einen Anteil der Gesamtnachricht enthält und einem Endstück (engl. *trailer*), das typischerweise den Fehlerkontrollcode enthält, siehe Abbildung 2.26. Jedes Paket wird einzeln entsprechend der enthaltenen Routing- oder Zielinformation zum Ziel geschickt. Die Verbindungsleitungen oder Puffer werden jeweils nur von einem Paket belegt.

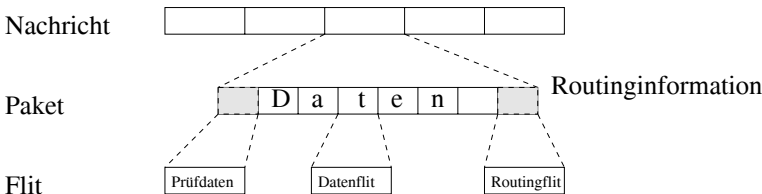


Abb. 2.26. Illustration zur Zerlegung einer Nachricht in Pakete und von Paketen in *flits* (flow control units).

Die Paket-Switching-Strategie kann in verschiedenen Varianten realisiert werden. Paket-Switching mit **Store-and-Forward-Routing** versendet ein gesamtes Paket über je eine Verbindung auf dem für das Paket ausgewählten Pfad zum Empfänger. Jeder Zwischenempfänger, d.h. jeder Knoten auf dem Pfad speichert das gesamte Paket (*store*) bevor es weitergeschickt wird (*forward*). Die Verbindung zwischen zwei Knoten wird freigegeben, sobald das

Paket beim Zwischenempfänger zwischengespeichert wurde. Diese Switching-Strategie wurde für frühe Parallelrechner verwendet und wird teilweise noch von Routern für IP-Pakete in WANs (*wide area networks*) benutzt. Ein Vorteil der Store-and-Forward-Strategie ist eine schnelle Freigabe von Verbindungen, was die Deadlockgefahr verringert und mehr Flexibilität bei hoher Netzbelastung erlaubt. Nachteile sind der hohe Speicherbedarf für die Zwischenpufferung von Paketen sowie eine Kommunikationszeit, die von der Länge der Pfade abhängt, was je nach Topologie und Kommunikationsanforderung zu hohen Kommunikationszeiten führen kann.

Die Kosten zum Versenden eines Paketes über eine Verbindung sind $t_h + t_B \cdot m$, wobei m die Größe des Paketes ist und t_h die konstante Zeit bezeichnet, die an einem Zwischenknoten auf dem Pfad zum Ziel benötigt wird, um z.B. das Paket im Eingangspuffer abzulegen und durch Untersuchung des Headers den nächsten Ausgabekanal auszuwählen. Die Gesamtkosten des Transfers eines Paketes bei Paket-Switching mit Store-and-Forward-Routing auf einem Pfad der Länge l betragen damit

$$T_{sf}(m, l) = t_S + l(t_h + t_B \cdot m). \quad (2.4)$$

Da t_h im Vergleich zu den anderen Größen üblicherweise recht klein ist, ist $T_{sf}(m, l) \approx t_S + l \cdot t_B \cdot m$. Die Kosten für die Zustellung eines Paketes hängen also vom Produkt der Nachrichtengröße m und der Pfadlänge l ab. Eine Illustration der Kosten für einen Einzeltransfer für Paket-Switching mit Store-and-Forward-Routing findet man in Abbildung 2.27 b). Die Kosten für den Transport einer aus mehreren Paketen bestehenden Nachricht vom Sendeknoten zum Empfangsknoten hängen vom verwendeten Routingverfahren ab. Für ein deterministisches Routingverfahren ergibt sich die Transportzeit als die Summe der Transportkosten der einzelnen Pakete, wenn es nicht zu Verzögerungen im Netzwerk kommt. Für adaptive Routingverfahren können sich die Transportzeiten der einzelnen Pakete überlappen, so dass eine geringere Gesamtzeit resultieren kann.

Wenn alle Pakete einer Nachricht den gleichen Übertragungspfad verwenden können, kann die Einführung von **Pipelining** zur Verringerung der Kommunikationszeiten beitragen. Dazu werden die Pakete einer Nachricht so durch das Netzwerk geschickt, dass die Verbindungen auf dem Pfad von aufeinanderfolgenden Paketen überlappend genutzt werden. Ein derartiger Ansatz wird z.T. in software-realisierten Datenkommunikationen in Kommunikationsnetzwerken wie dem Internet benutzt. Bei Pipelining einer Nachricht der Größe m und Paketgröße m_p ergeben sich die Kosten

$$t_S + (m - m_p)t_B + l(t_h + t_B \cdot m_p) \approx t_S + m \cdot t_B + (l - 1)t_B \cdot m_p. \quad (2.5)$$

Dabei ist $l(t_h + t_B \cdot m_p)$ die Zeit, die bis zur Ankunft des ersten Paketes vergeht. Danach kommt in jedem Zeitschritt der Größe $m_p \cdot t_B$ ein weiteres Paket an.

Der Ansatz des gepipelineten Paket-Switching kann mit Hilfe von **Cut-Through-Routing** noch weitergetrieben werden. Die Nachricht wird ent-

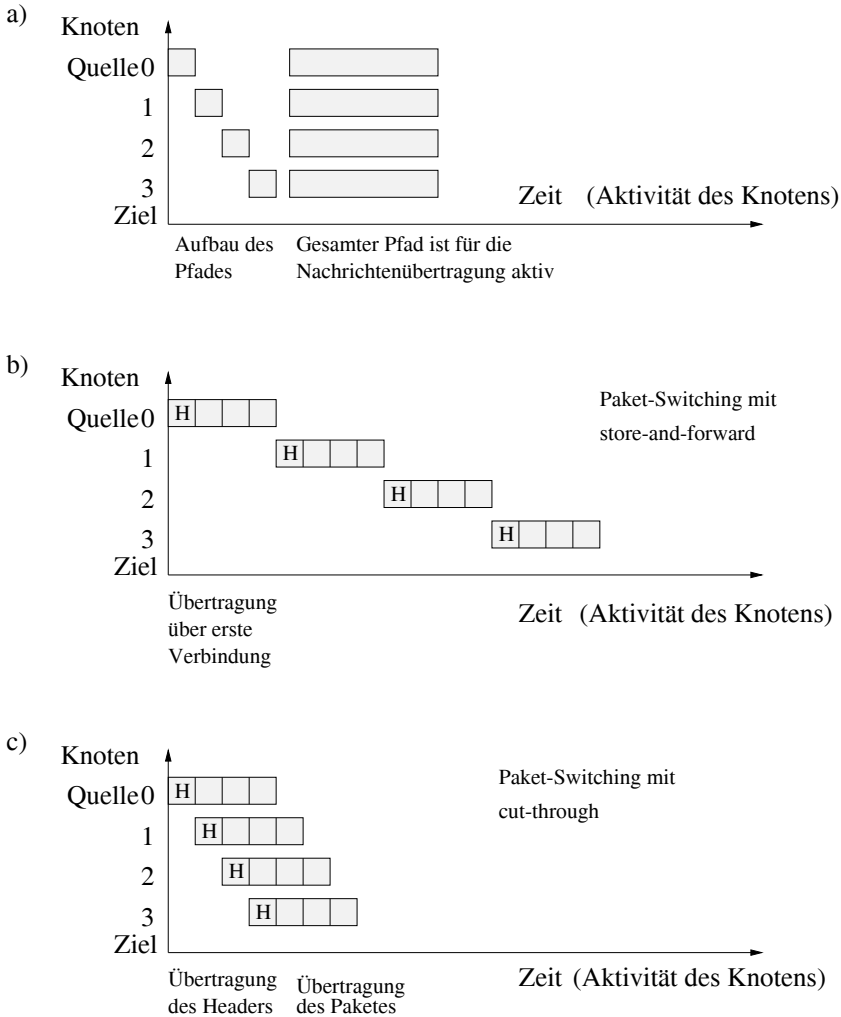


Abb. 2.27. Illustration zur Latenzzeit einer Einzeltransferoperation über einen Pfad der Länge $l = 4$ a) Circuit-Switching, b) Paket-Switching mit store-and-forward und c) Paket-Switching mit cut-through.

sprechend des Paket-Switching-Ansatzes in Pakete unterteilt und jedes einzelne Paket wird pipelineartig durch das Netzwerk geschickt. Die verschiedenen Pakete einer Nachricht können dabei prinzipiell verschiedene Übertragungspfade verwenden. Beim Cut-Through-Routing betrachtet ein auf dem Übertragungspfad liegender Schalter (bzw. Knoten oder Router) die ersten *phits* (*physical units*) des ankommenden Paketes, die den Header mit der Routin-
information enthalten, und trifft daraufhin die Entscheidung, zu welchem

Knoten das Paket weitergeleitet wird. Der Verbindungspfad wird also vom Header eines Paketes aufgebaut. Ist die gewünschte Verbindung frei, so wird der Header weitergeschickt und der Rest des Paketes wird direkt hinterhergeleitet, so dass die *phits* des Paketes pipelineartig auf dem Übertragungspfad verteilt sind. Verbindungen, über die alle *phits* des Paketes einschließlich Endstück vollständig übertragen wurden, werden freigegeben. Je nach Situation im Netzwerk kann also der gesamte Pfad vom Ausgangsknoten bis zum Zielknoten der Übertragung eines Paketes zugeordnet sein.

Sind die Kosten zur Übertragung des Headers auf einer Verbindungsleitung durch t_H gegeben, d.h. $t_H = t_B \cdot m_H$, wobei m_H die Größe des Headers ist, so sind die Kosten zur Übertragung des Headers auf dem gesamten Pfad der Länge l durch $t_H \cdot l$ gegeben. Die Zeit bis zur Ankunft des Paketes der Größe m am Zielknoten *nach* Ankunft des Headers beträgt $t_B \cdot (m - m_H)$. Die Kosten für den Transport eines Paketes betragen bei Verwendung von Paket-Switching mit Cut-Through-Routing auf einem Pfad der Länge l ohne Contention

$$T_{ct}(m, l) = t_S + l \cdot t_H + t_B \cdot (m - m_H) . \quad (2.6)$$

Ist m_H im Vergleich zu der Gesamtgröße m der Nachricht klein, so entspricht dies ungefähr den Kosten $T_{ct}(m, l) \approx t_S + t_B \cdot m$. Verwenden alle Pakete einer Nachricht den gleichen Übertragungspfad und werden die Pakete ebenfalls nach dem Pipelining-Prinzip übertragen, gilt diese Formel auch für die Übertragung einer gesamten Nachricht der Größe m . Die Kosten für den Transport eines Paketes für Paket-Switching mit Cut-Through-Routing sind in Abbildung 2.27 c) illustriert.

Sind außer dem zu übertragenden Paket noch andere Pakete im Netzwerk, so muss Contention, also die Anforderungen einer Verbindung durch mehrere Nachrichten berücksichtigt werden. Ist die nächste gewünschte Verbindung *nicht* frei, so werden bei **virtuellem Cut-Through-Routing** alle *phits* des Paketes im letzten erreichten Knoten aufgesammelt und dort zwischengepuffert. Geschieht dies an jedem Knoten, so kann Cut-Through-Routing zu Store-and-Forward-Routing degenerieren. Bei **partielltem Cut-Through-Routing** können Teile des Paketes weiter übertragen werden, falls die gewünschte Verbindung frei wird, bevor alle *phits* des Paketes in einem Knoten auf dem Pfad zwischengepuffert werden.

Viele derzeitige Parallelrechner benutzen eine Variante des Cut-Through-Routing, die **Wormhole-Routing** oder manchmal auch Hardware-Routing genannt wird, da es durch die Einführung von Routern eine hardwaremäßige Unterstützung gibt, vgl. Abschnitt 2.4.1. Beim Wormhole-Routing werden die Pakete in kleine Einheiten zerlegt, die **flits** (für engl. *flow control units*) genannt werden und deren Größe typischerweise zwischen 1 und 8 Bytes liegt. Die Header-*flits* bestimmen den Weg durch das Netzwerk. Alle anderen *flits* des Paketes folgen pipelinemäßig auf demselben Pfad. Die Zwischenpuffer an den Knoten bzw. den Ein- und/oder Ausgabekanälen der Knoten sind nur für wenige *flits* ausgelegt. Ist eine gewünschte Verbindung nicht frei,

so blockiert der Header bis die Verbindung frei wird. Alle nachfolgenden *flits* werden ebenfalls blockiert und verbleiben in ihrer Position. Im Gegensatz zur oben beschriebenen Form des Cut-Through-Routing werden *flits* also *nicht* bis zur den Header blockierenden Stelle nachgeholt, sondern blockieren einen gesamten Pfad. Dadurch gleicht dieser Ansatz eher dem Circuit-Switching auf Paketebene. Ein Vorteil von Wormhole-Routing ist der geringe Speicherplatzbedarf für die Zwischenspeicherung. Durch die Blockierung ganzer Pfade erhöht sich jedoch wieder die Deadlockgefahr durch zyklisches Warten, vgl. Abbildung 2.28 [111].

Die Deadlockgefahr kann durch geeignete Routing-Algorithmen, z. B. dimensionsgeordnetes Routing, oder die Verwendung virtueller Kanäle vermieden werden.

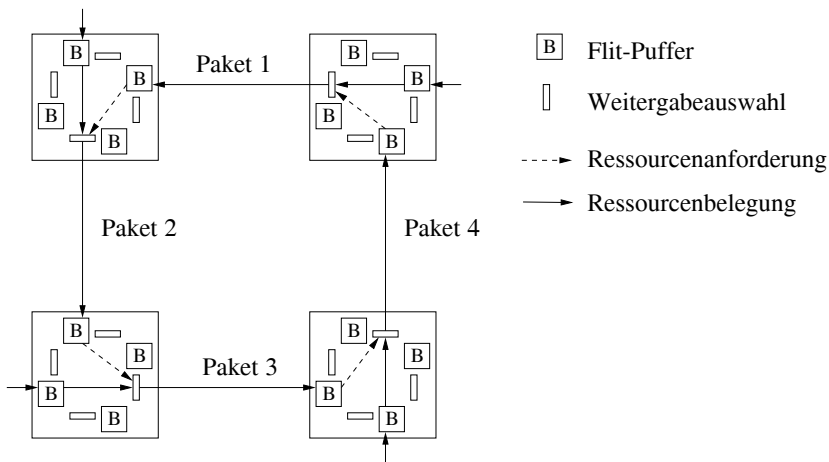


Abb. 2.28. Illustration zur Deadlock-Gefahr beim Wormhole-Routing von vier Paketen über vier Router. Jedes der vier Pakete belegt einen Flit-Puffer und fordert einen Flit-Puffer an, der von einem anderen Paket belegt ist. Es resultiert eine Deadlock-Situation, da keines der Pakete weitergeschickt werden kann.

2.6.3 Flusskontrollmechanismen

Flusskontrollmechanismen (engl. *flow control mechanism*) werden benötigt, wenn sich mehrere Nachrichten im Netzwerk befinden und geregelt werden muss, welchem Paket eine Verbindung oder ein Puffer zur Verfügung gestellt wird. Sind angeforderte Ressourcen bereits von anderen Nachrichten oder Nachrichtenteilen belegt, so entscheidet ein Flusskontrollmechanismus, ob die Nachricht blockiert wird, wo sie sich befindet, in einem Puffer zwischengespeichert wird, auf einem alternativen Pfad weitergeschickt wird oder

einfach weggeworfen wird. Die minimale Einheit, die über eine Verbindung geschickt und akzeptiert bzw. wegen beschränkter Kapazität zurückgewiesen werden kann, wird **flit** (*flow control unit*) genannt. Ein flit kann einem phit entsprechen, aber auch einem ganzen Paket oder einer ganzen Nachricht.

Flusskontrollmechanismen müssen in jeder Art von Netzwerk vorhanden sein und spielen bei Transportprotokollen wie TCP eine wichtige Rolle, vgl. z.B. [98, 123]. Die Netzwerke von Parallelrechnern stellen an die Flusskontrolle aber die besonderen Anforderungen, dass sich sehr viele Nachrichten auf engem Raum konzentrieren und die Nachrichtenübertragung für die korrekte Abarbeitung der Gesamtaufgabe sehr zuverlässig sein muss. Weiter sollten Staus (engl. *congestion*) in den Kanälen vermieden werden und eine schnelle Nachrichtenübertragung gewährleistet sein.

Flusskontrolle bzgl. der Zuordnung von Paketen zu Verbindungen (engl. *link-level flow-control*) regelt die Datenübertragungen über eine Verbindung, also vom Ausgangskanal eines Knotens über die Verbindungsleitung zum Eingangskanal des zweiten Knotens, an dem typischerweise ein kleiner Speicher oder Puffer ankommende Daten aufnehmen kann. Ist dieser Eingangspuffer beim Empfänger gefüllt, so kann die Nachricht nicht angenommen werden und muss beim Sender verbleiben, bis der Puffer wieder Platz bietet. Dieses Zusammenspiel wird durch einen Informationsaustausch mit *Anfrage* des Senders (*request*) und Bestätigung des Empfängers (*acknowledgement*) durchgeführt (*request-acknowledgement handshake*). Der Sender sendet ein Anfrage-Signal, wenn er eine Nachricht senden möchte. Der Empfänger sendet eine Bestätigung, falls die Daten empfangen wurden. Erst danach kann eine weitere Nachricht vom Sender losgeschickt werden.

2.7 Caches und Speicherhierarchien

Ein wesentliches Merkmal der Hardware-Entwicklung der letzten Jahrzehnte ist, wie bereits oben geschildert, das Auseinanderdriften von Prozessorgeschwindigkeit und Speicherzugriffsgeschwindigkeit, was durch ein vergleichsweise geringes Anwachsen der Zugriffsgeschwindigkeit auf DRAM-Chips begründet ist, die für die physikalische Realisierung von Hauptspeichern verwendet werden. Um jedoch trotzdem die Prozessorgeschwindigkeit effektiv nutzen zu können, wurden Speicherhierarchien eingeführt, die aus Speichern verschiedener Größen und Zugriffsgeschwindigkeiten bestehen und deren Ziel die Verringerung der *mittleren* Speicherzugriffszeiten ist. Die einfachste Form einer Speicherhierarchie ist das Einfügen eines einzelnen Caches zwischen Prozessor und Speicher (einstufiger Cache). Ein Cache ist ein relativ kleiner, schneller Speicher mit einer im Vergleich zum Hauptspeicher geringen Speicherzugriffszeit, die meist durch Verwendung von schnellen SRAM-Chips erreicht wird. In den Cache werden entsprechend einer vorgegebenen Nachladestrategie Daten des Hauptspeichers geladen mit dem Ziel, dass sich die zur

Abarbeitung eines Programms benötigten Daten zum Zeitpunkt des Zugriffs in den meisten Fällen im Cache befinden.

Für Multiprozessoren mit lokalen Caches der einzelnen Prozessoren stellt sich die zusätzliche Aufgabe der konsistenten Aufrechterhaltung des gemeinsamen Adressraumes. Typisch sind mittlerweile zwei- oder dreistufige Cache-Speicher für jeden Prozessor. Da viele neuere Multiprozessoren zur Klasse der Rechner mit *virtuell* gemeinsamem Speicher gehören, ist zusätzlich auch der gemeinsame Speicher als Stufe der Speicherhierarchie anzusehen. Dieser Trend der Hardware-Entwicklung in Richtung des virtuell gemeinsamen Speichers ist durch die geringeren Hardwarekosten begründet, die Rechner mit verteiltem Speicher gegenüber Rechnern mit physikalisch gemeinsamem Speicher verursachen. Der gemeinsame Speicher wird softwaremäßig realisiert. Da Caches die grundlegenden Bausteine von Speicherhierarchien darstellen, deren Arbeitsweise die weiterführenden Fragen der Konsistenz des Speichersystems wesentlich beeinflusst, beginnen wir mit einem kurzen Abriss über Caches. Für eine ausführlichere Behandlung verweisen wir auf [31, 71, 75, 121].

2.7.1 Charakteristika von Cache-Speichern

Ein **Cache** ist ein kleiner schneller Speicher, der zwischen Hauptspeicher und Prozessor eingefügt wird. Caches werden häufig durch SRAM-Chips (*Static Random Access Memory*) realisiert, deren Zugriffszeiten deutlich geringer sind als die von DRAM-Chips. Typische Zugriffszeiten sind 0.5-5 *ns* (*ns* = Nanosekunden = $10^{-9}sec$) für SRAM-Chips im Vergleich zu 45-70 *ns* für DRAM-Chips (Angaben von 2005 [121]). Zur Vereinfachung gehen wir im Folgenden zuerst von einem einstufigen Cache aus. Der Cache enthält eine *Kopie* von Teilen der im Hauptspeicher abgelegten Daten. Diese Kopie wird in Form von **Cacheblöcken** (engl. *cache line*), die üblicherweise aus mehreren Worten bestehen, aus dem Speicher in den Cache geladen, vgl. Abbildung 2.29. Die verwendete Blockgröße ist für einen Cache konstant und kann in der Regel während der Ausführung eines Programms nicht variiert werden.

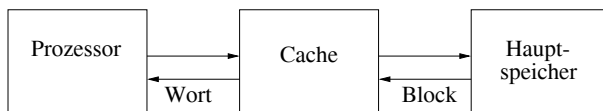


Abb. 2.29. Der Datentransport zwischen Cache und Hauptspeicher findet in Cacheblöcken statt, während der Prozessor auf einzelne Worte aus dem Cache zugreift.

Die Kontrolle des Caches ist vom Prozessor abgekoppelt und wird von einem eigenen Cache-Controller übernommen. Der Prozessor setzt entsprechend der Operanden der auszuführenden Maschinenbefehle Schreib- oder Leseoperationen an das Speichersystem ab und wartet gegebenenfalls, bis

dieses die angeforderten Operanden zur Verfügung stellt. Die Architektur des Speichersystems hat in der Regel keinen Einfluss auf die vom Prozessor abgesetzten Zugriffsoptionen, d.h. der Prozessor braucht keine Kenntnis von der Architektur des Speichersystems zu haben. Nach Empfang einer Zugriffsoption vom Prozessor überprüft der Cache-Controller eines einstufigen Caches, ob das zu lesende Wort im Cache gespeichert ist (**Cachetreffer**, engl. *cache hit*). Wenn dies der Fall ist, wird es vom Cache-Controller aus dem Cache geladen und dem Prozessor zur Verfügung gestellt. Befindet sich das Wort *nicht* im Cache (**Cache-Fehlzugriff**, engl. *cache miss*), wird der Block, in dem sich das Wort befindet, vom Cache-Controller aus dem Hauptspeicher in den Cache geladen. Da der Hauptspeicher relativ hohe Zugriffszeiten hat, dauert das Laden des Blockes wesentlich länger als der Zugriff auf den Cache. Beim Auftreten eines Cache-Fehlzugriffs werden die vom Prozessor angeforderten Operanden also nur verzögert zur Verfügung gestellt. Bei der Abarbeitung eines Programms sollten daher möglichst wenige Cache-Fehlzugriffe auftreten.

Dem Prozessor bleibt die genaue Arbeitsweise des Cache-Controllers verborgen. Er beobachtet nur den Effekt, dass bestimmte Speicherzugriffe länger dauern als andere und er länger auf die angeforderten Operanden warten muss. Diese Entkopplung von Speicherzugriffen und der Ausführung von arithmetisch/logischen Operationen stellt sicher, dass der Prozessor während des Wartens auf die Operanden andere Berechnungen durchführen kann, die von den ausstehenden Operanden unabhängig sind. Dies wird durch die Verwendung mehrerer Funktionseinheiten und durch das Vorladen von Operanden (engl. *operand prefetch*) unterstützt, siehe Abschnitt 2.2. Die beschriebene Entkopplung hat auch den Vorteil, dass Prozessor und Cache-Controller beliebig kombiniert werden können, d.h. ein Prozessor kann in unterschiedlichen Rechnern mit verschiedenen Speichersystemen kombiniert werden, ohne dass eine Adaption des Prozessors erforderlich wäre.

Wegen des beschriebenen Vorgehens beim Laden von Operanden hängt die Effizienz eines Programms wesentlich davon ab, ob viele oder wenige der vom Prozessor abgesetzten Speicherzugriffe vom Cache-Controller aus dem Cache bedient werden können. Wenn viele Speicherzugriffe zum Nachladen von Cacheblöcken führen, wird der Prozessor oft auf Operanden warten müssen und das Programm wird entsprechend langsam abgearbeitet. Da die Nachladestrategie des Caches von der Hardware vorgegeben ist, kann die Effektivität des Caches nur durch die Struktur des Programms beeinflusst werden. Dabei hat insbesondere das von einem gegebenen Programm verursachte Speicherzugriffsverhalten einen großen Einfluss auf seine Effizienz. Die für das Nachladen des Caches relevante Eigenschaft der Speicherzugriffe eines Programms versucht man mit dem Begriff der **Lokalität der Speicherzugriffe** zu fassen. Dabei unterscheidet man zwischen zeitlicher und räumlicher Lokalität:

- Die Speicherzugriffe eines Programms weisen eine hohe **räumliche Lokalität** auf, wenn zu aufeinanderfolgenden Zeitpunkten der Programmausführung auf im Hauptspeicher *räumlich benachbarte* Speicherzellen zugegriffen wird. Für ein Programm mit hoher räumlicher Lokalität tritt relativ oft der Effekt auf, dass nach dem Zugriff auf eine Speicherzelle unmittelbar nachfolgende Speicherzugriffe eine oder mehrere Speicherzellen desselben Cacheblockes adressieren. Nach dem Laden eines Cacheblockes in den Cache werden daher einige der folgenden Speicherzugriffe auf den gleichen Cacheblock zugreifen und es ist kein weiteres Nachladen erforderlich. Die Verwendung von Cacheblöcken, die mehrere Speicherzellen umfassen, basiert auf der Annahme, dass viele Programme eine hohe räumliche Lokalität aufweisen.
- Die Speicherzugriffe eines Programms weisen eine hohe **zeitliche Lokalität** auf, wenn auf *dieselbe* Speicherstelle zu *zeitlich* dicht aufeinanderfolgenden Zeitpunkten der Programmausführung zugegriffen wird. Für ein Programm mit hoher zeitlicher Lokalität tritt relativ oft der Effekt auf, dass nach dem Laden eines Cacheblockes in den Cache auf die einzelnen Speicherzellen dieses Cacheblockes mehrfach zugegriffen wird, bevor der Cacheblock wieder aus dem Cache entfernt wird.

Für ein Programm mit geringer räumlicher Lokalität besteht die Gefahr, dass nach dem Laden eines Cacheblockes nur auf eine seiner Speicherzellen zugegriffen wird, die anderen wurden also unnötigerweise geladen. Für ein Programm mit geringer zeitlicher Lokalität besteht die Gefahr, dass nach dem Laden eines Cacheblockes nur einmal auf eine Speicherzelle zugegriffen wird, bevor der Cacheblock wieder in den Hauptspeicher zurückgeschrieben wird. Verfahren zur Erhöhung der Lokalität der Speicherzugriffe eines Programms sind z.B. in [164] beschrieben.

Wir gehen im Folgenden auf wichtige Charakteristika von Caches näher ein. Wir untersuchen die Größe der Caches und der Cacheblöcke und deren Auswirkung auf das Nachladen von Cacheblöcken, die Abbildung von Speicherworten auf Positionen im Cache, Ersetzungsverfahren bei vollem Cache und Rückschreibestrategien bei Schreibzugriffen durch den Prozessor. Wir untersuchen auch den Einsatz von mehrstufigen Caches.

Cachegröße. Bei Verwendung der gleichen Technologie steigt die Zugriffszeit auf den Cache wegen der Steigerung der Komplexität der Adressierungsschaltung mit der Größe der Caches (leicht) an. Auf der anderen Seite erfordert ein großer Cache weniger Nachladeoperationen als ein kleiner Cache, weil mehr Speicherzellen im Cache abgelegt werden können. Die Größe eines Caches wird auch oft durch die zur Verfügung stehende Chipfläche begrenzt, insbesondere dann, wenn es sich um einen On-Chip-Cache handelt, d.h. wenn der Cache auf der Chipfläche des Prozessors untergebracht wird. Meistens liegt die Größe von Caches erster Stufe zwischen 8K und 128K Speicherworten, wobei ein Speicherwort je nach Rechner aus vier oder acht Bytes besteht.

Wie oben beschrieben, wird beim Auftreten eines Cache-Fehlzugriffs nicht nur das zugegriffene Speicherwort, sondern ein Block von Speicherworten in den Cache geladen. Für die Größe der Cacheblöcke müssen beim Design des Caches zwei Punkte beachtet werden: Zum einen verringert die Verwendung von größeren Blöcken die Anzahl der Blöcke, die in den Cache passen, die geladenen Blöcke werden also schneller ersetzt als bei der Verwendung von kleineren Blöcken. Zum anderen ist es sinnvoll, Blöcke mit mehr als einem Speicherwort zu verwenden, da der Gesamttransfer eines Blockes mit x Worten zwischen Hauptspeicher und Cache schneller durchgeführt werden kann als x Einzeltransporte mit je einem Wort. In der Praxis wird die Größe der Cacheblöcke (engl. *cache line size*) für Caches erster Stufe meist auf vier oder acht Speicherworte festgelegt.

Abbildung von Speicherblöcken auf Cacheblöcke. Daten werden in Form von Blöcken einheitlicher Größe vom Hauptspeicher in sogenannte *Blockrahmen* gleicher Größe des Caches eingelagert. Da der Cache weniger Blöcke als der Hauptspeicher fasst, muss eine Abbildung zwischen Speicherblöcken und Cacheblöcken durchgeführt werden. Dazu können verschiedene Methoden verwendet werden, die die Organisation des Caches wesentlich festlegen und auch die Suche nach Cacheblöcken bestimmen. Dabei spielt der Begriff der **Cache-Assoziativität** eine große Rolle. Die Assoziativität eines Caches legt fest, in wie vielen Blockrahmen ein Speicherblock abgelegt werden kann. Es werden folgende Ansätze verwendet:

- (a) Bei direkt-abgebildeten Caches (engl. *direct-mapped cache*) kann jeder Speicherblock in genau einem Blockrahmen abgelegt werden.
- (b) Bei voll-assoziativen Caches (engl. *associative cache*) kann jeder Speicherblock in einem beliebigen Blockrahmen abgelegt werden.
- (c) Bei mengen-assoziativen Caches (engl. *set-associative cache*) kann jeder Speicherblock in einer festgelegten Anzahl von Blockrahmen abgelegt werden.

Alle drei Abbildungsmechanismen werden im Folgenden kurz vorgestellt. Dabei betrachten wir ein aus einem Hauptspeicher und einem Cache bestehendes Speichersystem. Wir nehmen an, dass der Hauptspeicher $n = 2^s$ Blöcke fasst, die wir mit B_j , $j = 0, \dots, n-1$, bezeichnen. Die Anzahl der Blockrahmen $\bar{B}_i$ im Cachespeicher sei $m = 2^r$. Jeder Speicherblock und Blockrahmen fasse $l = 2^w$ Speicherworte. Da jeder Blockrahmen des Caches zu einem bestimmten Zeitpunkt der Programmausführung verschiedene Speicherblöcke enthalten kann, muss zu jedem Blockrahmen eine Markierung (engl. *tag*) abgespeichert werden, die die Identifikation des abgelegten Speicherblockes erlaubt. Wie diese Markierung verwendet wird, hängt vom benutzten Abbildungsmechanismus ab und wird im Folgenden beschrieben. Als begleitendes Beispiel betrachten wir ein Speichersystem, dessen Cache 64 KBytes groß ist und der Cacheblöcke der Größe 4 Bytes verwendet. Der Cache fasst also $16K = 2^{14}$ Blöcke mit je 4 Bytes, d.h. es ist $r = 14$ und $w = 2$. Der Hauptspeicher ist $16 \text{ MBytes} = 2^{24}$

Bytes groß, d.h. es ist $s = 22$, wobei wir annehmen, dass die Speicherworte ein Byte umfassen.

- (a) **Direkt abgebildeter Cache:** Ein direkt-abgebildeter Cache stellt die einfachste Form der Cache-Organisation dar. Jeder Datenblock B_j des Hauptspeichers wird *genau einem* Blockrahmen $\bar{B}_i$ des Caches zugeordnet, in den er bei Bedarf eingelagert werden kann. Die Abbildungsvorschrift von Blöcken auf Blockrahmen ist z.B. gegeben durch:

B_j wird auf $\bar{B}_i$ abgebildet, falls $i = j \bmod m$ gilt.

In jedem Blockrahmen können also $n/m = 2^{s-r}$ verschiedene Speicherblöcke abgelegt werden. Entsprechend der obigen Abbildung gilt folgende Zuordnung:

Blockrahmen	Speicherblock
0	$0, m, 2m, \dots, 2^s - m$
1	$1, m + 1, 2m + 1, \dots, 2^s - m + 1$
$\vdots$	$\vdots$
$m - 1$	$m - 1, 2m - 1, 3m - 1, \dots, 2^s - 1$

Der Zugriff des Prozessors auf ein Speicherwort erfolgt über dessen Speicheradresse, die sich aus einer Blockadresse und einer Wortadresse zusammensetzt. Die Blockadresse gibt die Adresse des Speicherblockes, der die angegebene Speicheradresse enthält, im Hauptspeicher an. Sie wird von den s signifikantesten, d.h. linken Bits der Speicheradresse gebildet. Die Wortadresse gibt die relative Adresse des angegebenen Speicherwortes bzgl. des Anfanges des zugehörigen Speicherblockes an. Sie wird von den w am wenigsten signifikanten, d.h. rechts liegenden Bits der Speicheradresse gebildet. Bei direkt-abgebildeten Caches identifizieren die r rechten Bits der Blockadresse diejenigen der $m = 2^r$ Blockrahmen, in den der entsprechende Speicherblock gemäß obiger Abbildung eingelagert werden kann. Die $s - r$ verbleibenden Bits können als Markierung (*tag*) interpretiert werden, die angibt, welcher Speicherblock aktuell in einem bestimmten Blockrahmen des Caches enthalten ist. In dem oben angegebenen Beispiel bestehen die Markierungen aus $s - r = 8$ Bits.

Der Speicherzugriff wird in Abbildung 2.30 a) illustriert. Bei jedem Speicherzugriff wird zunächst der Blockrahmen, in dem der zugehörige Speicherblock abgelegt werden muss, durch die r rechten Bits der Blockadresse identifiziert. Anschließend wird die aktuelle Markierung dieses Blockrahmens, die zusammen mit der Position des Blockrahmens den aktuell abgelegten Speicherblock eindeutig identifiziert, mit den $s - r$ linken Bits der Blockadresse verglichen. Stimmen beide Markierungen überein, so handelt es sich um einen *Cachetreffer*, d.h. der zugehörige Speicherblock befindet sich im Cache und der Speicherzugriff kann aus dem Cache bedient werden. Wenn die Markierungen nicht übereinstim-

men, muss der zugehörige Speicherblock in den Cache geladen werden, bevor der Speicherzugriff erfolgen kann.

Direkt abgebildete Caches sind zwar einfach zu realisieren, haben jedoch den Nachteil, dass jeder Speicherblock nur an einer Position im Cache abgelegt werden kann. Bei ungünstiger Speicheradressierung eines Programms besteht daher die Gefahr, dass zwei oft benutzte Speicherblöcke auf den gleichen Blockrahmen abgebildet sein können und so ständig zwischen Hauptspeicher und Cache hin- und hergeschoben werden müssen. Dadurch kann die Laufzeit eines Programms erheblich erhöht werden.

- (b) **Voll-assoziativer Cache:** Beim voll-assoziativen Cache kann jeder Speicherblock in jedem beliebigen Blockrahmen des Caches abgelegt werden, wodurch der Nachteil des häufigen Ein- und Auslagerns von Blöcken behoben wird. Der Speicherzugriff auf ein Wort erfolgt wieder über die aus der Blockadresse (s linken Bits) und der Wortadresse (w rechten Bits) zusammengesetzten Speicheradresse. Als Markierung eines Blockrahmens im Cache muss nun jedoch die gesamte Blockadresse verwendet werden, da jeder Blockrahmen jeden Speicherblock enthalten kann. Bei einem Speicherzugriff müssen also die Markierungen *aller* Blockrahmen im Cache durchsucht werden, um festzustellen, ob sich der entsprechende Block im Cache befindet. Dies wird in Abbildung 2.30 b) veranschaulicht. Der Vorteil von voll-assoziativen Caches liegt in der hohen Flexibilität beim Laden von Speicherblöcken. Der Nachteil liegt zum einen darin, dass die verwendeten Markierungen wesentlich mehr Bits beinhalten als bei direkt-abgebildeten Caches. Im oben eingeführten Beispiel bestehen die Markierungen aus 22 Bits, d.h. für jeden 32-Bit-Speicherblock muss eine 22-Bit-Markierung abgespeichert werden. Ein weiterer Nachteil liegt darin, dass bei jedem Speicherzugriff die Markierungen aller Blockrahmen untersucht werden müssen, was entweder eine sehr komplexe Schaltung erfordert oder zu Verzögerungen bei den Speicherzugriffen führt.
- (c) **Mengen-assoziativer Cache:** Der mengen-assoziative Cache stellt einen Kompromiss zwischen direkt-abgebildeten und voll assoziativen Caches dar. Der Cache wird in v Mengen $S_0, \dots, S_{v-1}$ unterteilt, wobei jede Menge $k = m/v$ Blockrahmen des Caches enthält. Die Idee besteht darin, Speicherblöcke B_j für $j = 0, \dots, n-1$, nicht direkt auf Blockrahmen, sondern auf die eingeführten Mengen von Blockrahmen abzubilden. Innerhalb der zugeordneten Menge kann der Speicherblock beliebig positioniert werden, d.h. jeder Speicherblock kann in k verschiedenen Blockrahmen aufgehoben werden. Die Abbildungsvorschrift von Blöcken auf Mengen von Blockrahmen lautet:

B_j wird auf Menge S_i abgebildet, falls $i = j \bmod v$ gilt.

Der Speicherzugriff auf eine Speicheradresse (bestehend aus Blockadresse und Wortadresse) ist in Abbildung 2.30 c) veranschaulicht. Die $d = \log v$ rechten Bits der Blockadresse geben die Menge S_i an, der der Speicherblock zugeordnet wird. Die linken $s - d$ Bits bilden die Markierung zur

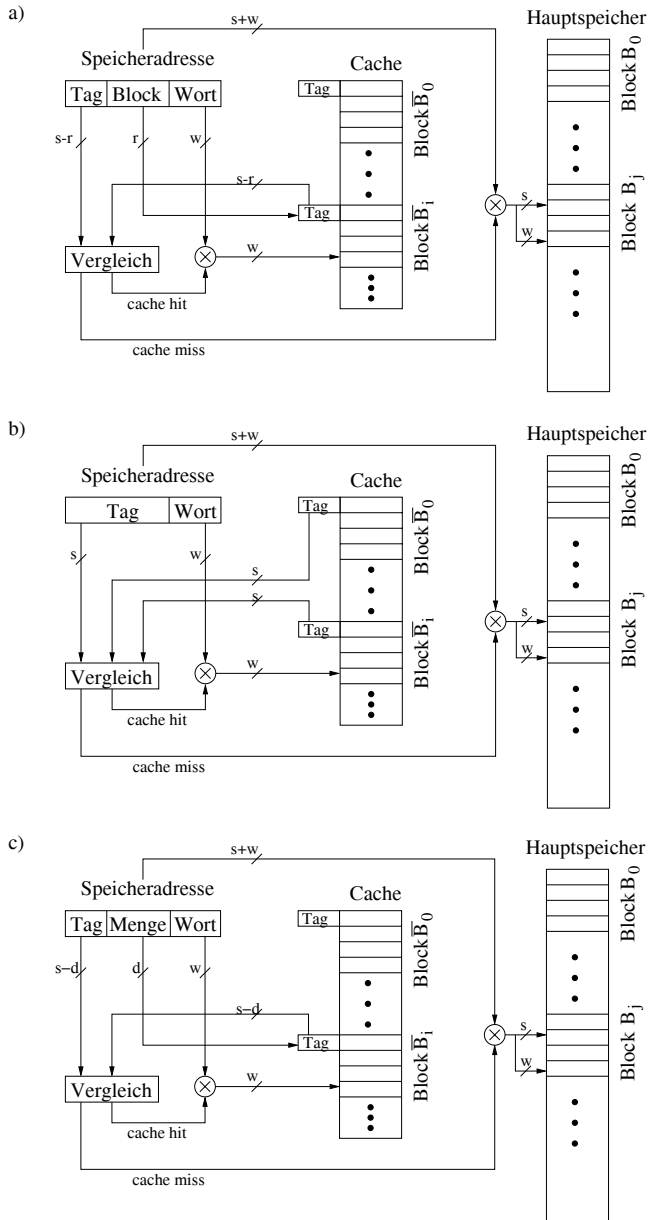


Abb. 2.30. Abbildungsmechanismen von Blöcken des Hauptspeichers auf Blockrahmen des Caches. a) direkt-abgebildeter Cache (oben), b) voll-assoziativer Cache (Mitte), c) mengen-assoziativer Cache (unten).

Identifikation der einzelnen Speicherblöcke in einer Menge. Beim Speicherzugriff wird zunächst die Menge im Cache identifiziert, der der zugehörige Speicherblock zugeordnet wird. Anschließend wird die Markierung des Speicherblockes mit den Markierungen der Blockrahmen *innerhalb* dieser Menge verglichen. Wenn die Markierung mit einer der Markierungen der Blockrahmen übereinstimmt, kann der Speicherzugriff über den Cache bedient werden, ansonsten muss der Speicherblock aus dem Hauptspeicher nachgeladen werden.

Für $v = m$ und $k = 1$ degeneriert der mengen-assoziative Cache zum direkt-abgebildeten Cache. Für $v = 1$ und $k = m$ ergibt sich der voll-assoziative Cache. Häufig verwendete Größen sind $v = m/4$ und $k = 4$ oder $v = m/8$ und $k = 8$. Im ersten Fall spricht man von einem *vier-Wege-assoziativen* Cache (engl. *4-way set-associative cache*), im zweiten Fall von einem *acht-Wege-assoziativen* Cache. Für $k = 4$ entstehen in unseren Beispiel 4K Mengen, für deren Identifikation $d = 12$ Bits verwendet werden. Für die Identifikation der in einer Menge abgelegten Speicherblöcke werden Markierungen mit 10 Bits verwendet.

Blockersetzungsmethoden. Soll ein neuer Speicherblock in den Cache geladen werden, muss evtl. ein anderer Speicherblock aus dem Cache entfernt werden. Für direkt-abgebildete Caches gibt es dabei wie oben beschrieben nur eine Möglichkeit. Bei voll-assoziativen und mengen-assoziativen Caches kann der zu ladende Speicherblock in mehreren Blockrahmen gespeichert werden, d.h. es gibt mehrere Blöcke, die ausgelagert werden könnten. Die Auswahl des auszulagernden Blockes wird gemäß einer Ersetzungsmethode vorgenommen. Die *LRU-Ersetzungsmethode* (*Least-recently-used*) entfernt den Block aus der entsprechenden Blockmenge, der am längsten unreferenziert ist. Zur Realisierung dieser Methode muss im allgemeinen Fall für jeden in einem Blockrahmen abgelegten Speicherblock der Zeitpunkt der letzten Benutzung abgespeichert und bei jedem Zugriff auf diesen Block aktualisiert werden. Dies erfordert zusätzlichen Speicherplatz zur Ablage der Benutzungszeitpunkte und zusätzliche Kontrolllogik zur Verwaltung und Verwendung dieser Benutzungszeitpunkte. Für zwei-Wege-assoziative Caches kann die LRU-Methode jedoch einfacher realisiert werden, indem jeder Blockrahmen jeder (in diesem Fall zweielementigen) Menge ein USE-Bit erhält, das wie folgt verwaltet wird: Wenn auf eine in dem Blockrahmen abgelegte Speicherzelle zugegriffen wird, wird das USE-Bit dieses Blockrahmens auf 1, das USE-Bit des anderen Blockrahmens der Menge auf 0 gesetzt. Dies geschieht bei jedem Speicherzugriff. Damit wurde auf den Blockrahmen, dessen USE-Bit auf 1 steht, zuletzt zugegriffen, d.h. wenn ein Blockrahmen entfernt werden soll, wird der Blockrahmen ausgewählt, dessen USE-Bit auf 0 steht.

Eine Alternative zur LRU-Ersetzungsmethode ist die *LFU-Ersetzungsmethode* (*Least-frequently-used*), die bei Bedarf den Block aus der Blockmenge entfernt, auf den am wenigsten oft zugegriffen wurde. Auch diese Ersetzungsmethode erfordert im allgemeinen Fall einen großen Mehraufwand, da zu je-

dem Block ein Zähler gehalten und bei jedem Speicherzugriff auf diesen Block aktualisiert werden muss. Eine weitere Alternative besteht darin, den zu ersetzenden Block zufällig auszuwählen. Diese Variante hat den Vorteil, dass kein zusätzlicher Verwaltungsaufwand notwendig ist.

Rückschreibestrategien. Bisher haben wir im Wesentlichen die Situation betrachtet, dass Daten aus dem Hauptspeicher gelesen werden und haben den Einsatz von Caches zur Verringerung der mittleren Zugriffszeit untersucht. Wir wenden uns jetzt der Frage zu, was passiert, wenn der Prozessor den Wert eines Speicherwortes, das im Cache aufgehoben wird, verändert, indem er eine entsprechende Schreiboperation an das Speichersystem weiterleitet. Das entsprechende Speicherwort wird auf jeden Fall im Cache aktualisiert, damit der Prozessor beim nächsten Lesezugriff auf dieses Speicherwort vom Speichersystem den aktuellen Wert erhält. Es stellt sich aber die Frage, wann die Kopie des Speicherwortes im Hauptspeicher aktualisiert wird. Diese Kopie kann frühestens nach der Aktualisierung im Cache und muss spätestens bei der Entfernung des entsprechenden Speicherblocks aus dem Cache aktualisiert werden. Der genaue Zeitpunkt und der Vorgang der Aktualisierung wird durch die *Rückschreibestrategie* festgelegt. Die beiden am häufigsten verwendeten Rückschreibestrategien sind die Write-through-Strategie und die Write-back-Strategie:

- (a) **Write-through-Rückschreibestrategie:** Wird ein im Cache befindlicher Speicherblock durch eine Schreiboperation modifiziert, so wird neben dem Eintrag im Cache auch der zugehörige Eintrag im Hauptspeicher aktualisiert, d.h. Schreiboperationen auf den Cache werden in den Hauptspeicher „durchgeschrieben“. Somit enthalten die Speicherblöcke im Cache und die zugehörigen Kopien im Hauptspeicher immer die gleichen Werte. Der Vorteil dieses Ansatzes liegt darin, dass I/O-Geräte, die direkt ohne Zutun des Prozessors auf den Hauptspeicher (DMA, *direct memory access*) zugreifen, stets die aktuellen Werte erhalten. Dieser Vorteil spielt auch bei Multiprozessoren eine große Rolle, da andere Prozessoren beim Zugriff auf den Hauptspeicher *immer* den aktuellen Wert erhalten. Der Nachteil des Ansatzes besteht darin, dass das Aktualisieren eines Wertes im Hauptspeicher im Vergleich zum Aktualisieren im Cache relativ lange braucht. Daher muss der Prozessor möglicherweise warten, bis der Wert zurückgeschrieben wurde (engl. *write stall*). Der Einsatz eines Schreibpuffers, in dem die in den Hauptspeicher zu transportierenden Daten zwischengespeichert werden, kann dieses Warten verhindern [75].
- (b) **Write-back-Rückschreibestrategie:** Eine Schreiboperation auf einen im Cache befindlichen Block wird zunächst *nur* im Cache durchgeführt, d.h. der zugehörige Eintrag im Hauptspeicher wird nicht sofort aktualisiert. Damit können Einträge im Cache aktuellere Werte haben als die zugehörigen Einträge im Hauptspeicher, d.h. die Werte im Hauptspeicher sind u.U. veraltet. Die Aktualisierung des Blocks im Hauptspeicher findet erst statt, wenn der Block im Cache durch einen anderen Block

ersetzt wird. Um festzustellen, ob beim Ersetzen eines Cacheblockes ein Zurückschreiben notwendig ist, wird für jeden Cacheblock ein Bit (*dirty bit*) verwendet, das angibt, ob der Cacheblock seit dem Einlagern in den Cache modifiziert worden ist. Dieses Bit wird beim Laden eines Speicherblockes in den Cache mit 0 initialisiert. Bei der ersten Schreiboperation auf eine Speicherzelle des Blockes wird das Bit auf 1 gesetzt. Bei dieser Strategie werden in der Regel weniger Schreiboperationen auf den Hauptspeicher durchgeführt, da Cacheeinträge mehrfach geschrieben werden können, bevor der zugehörige Speicherblock in den Hauptspeicher zurückgeschrieben wird. Der Hauptspeicher enthält aber evtl. ungültige Werte, so dass ein direkter Zugriff von I/O-Geräten nicht ohne weiteres möglich ist. Dieser Nachteil kann dadurch behoben werden, dass die von I/O-Geräten zugreifbaren Bereiche des Hauptspeichers mit einer besonderen Markierung versehen werden, die besagt, dass diese Teile nicht im Cache aufgehoben werden können. Eine andere Möglichkeit besteht darin, I/O-Operationen nur vom Betriebssystem ausführen zu lassen, so dass dieses bei Bedarf Daten im Hauptspeicher vor der I/O-Operation durch Zugriff auf den Cache aktualisieren kann.

Beindet sich bei einem Schreibzugriff die Zieladresse nicht im Cache (*write miss*), so wird bei den meisten Caches der Speicherblock, der die Zieladresse enthält, zuerst in den Cache geladen und die Modifizierung wird wie oben skizziert durchgeführt (*write-allocate*). Eine weniger oft verwendete Alternative besteht darin, den Speicherblock nur im Hauptspeicher zu modifizieren und nicht in den Cache zu laden (*write no allocate*).

Anzahl der Caches. In der bisherigen Beschreibung haben wir die Arbeitsweise eines einzelnen Caches beschrieben, der zwischen Prozessor und Hauptspeicher geschaltet ist und in dem Daten des auszuführenden Programms abgelegt werden. Ein in dieser Weise verwendeter Cache wird als **Datencache** erster Stufe bezeichnet. Neben den Programmdateien greift ein Prozessor auch auf die Instruktionen des auszuführenden Programms zu, um diese zu dekodieren und die angegebenen Operationen auszuführen. Dabei wird wegen Schleifen im Programm auf einzelne Instruktionen evtl. mehrfach zugegriffen und die Instruktionen müssen mehrfach geladen werden. Obwohl die Instruktionen im gleichen Cache wie die Daten aufgehoben und auf die gleiche Weise verwaltet werden können, verwendet man in der Praxis meistens einen separaten **Instruktionscache**, d.h. die Instruktionen und die Daten eines Programms werden in separaten Caches aufgehoben (*split cache*). Dies erlaubt eine größere Flexibilität beim Design der Caches, da getrennte Daten- und Instruktionscaches entsprechend der Prozessororganisation unterschiedliche Größe und Assoziativität haben und unabhängig voneinander arbeiten können.

In der Praxis werden häufig **mehrstufige Caches**, also mehrere hierarchisch angeordnete Caches, verwendet. Zur Zeit werden, wie in Abbildung 2.31 veranschaulicht, meistens zweistufige Caches verwendet. Für den Einsatz

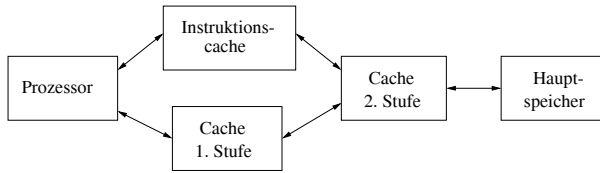


Abb. 2.31. Zweistufige Speicherhierarchie.

in Servern entwickelte Chips wie der Itanium 2 Prozessor von Intel verwenden aber auch (teilweise) Cachehierarchien mit drei Stufen. Die Caches sind meist auf der Chipfläche des Prozessors integriert. Typische Cachegrößen sind 8-128 KBytes für den Cache erster Stufe (L1-Cache), 512 KBytes - 8 MBytes für den Cache zweiter Stufe (L2-Cache) und 512 MBytes bis mehrere GBytes für den Hauptspeicher. Typische Speicherzugriffszeiten sind ein oder einige wenige Maschinenzyklen für den Cache erster Stufe, 10 bis 25 Maschinenzyklen für den Cache zweiter Stufe, 100 bis 1000 Maschinenzyklen für den Hauptspeicher und 10 bis 100 Millionen Maschinenzyklen für eine Festplatte. [121]). Diese Angaben beziehen sich auf 2005.

2.7.2 Cache-Kohärenz

Im letzten Abschnitt haben wir gesehen, dass die Einführung von schnellen Cache-Speichern zwar das Problem des zu langsamen Speicherzugriffs auf den Hauptspeicher löst, dafür aber die zusätzliche Aufgabe aufwirft, dafür zu sorgen, dass sich Veränderungen von Daten im Cache-Speicher auch auf den Hauptspeicher auswirken, und zwar spätestens dann, wenn andere Komponenten (also z.B. I/O-Systeme oder andere Prozessoren) auf den Hauptspeicher zugreifen. Diese anderen Komponenten sollen natürlich auf den *korrekten* Wert zugreifen, also auf den Wert, der zuletzt einer Variablen zugewiesen wurde. Wir werden dieses Problem in diesem Abschnitt näher untersuchen, wobei wir insbesondere Systeme mit mehreren unabhängig voneinander arbeitenden Prozessoren betrachten.

In einem Multiprozessor, in dem jeder Prozessor jeweils einen lokalen Cache besitzt, können Prozessoren gleichzeitig ein und denselben Speicherblock in ihrem lokalen Cache haben. Nach Modifikation derselben Variable in verschiedenen lokalen Caches können die lokalen Caches und der globale Speicher verschiedene, also *inkonsistente* Werte enthalten. Dies widerspricht dem Programmiermodell der gemeinsamen Variablen und kann zu falschen Ergebnissen führen. Diese bei Vorhandensein von lokalen Caches aufkommende Schwierigkeit bei Multiprozessoren wird als *Speicherkohärenz-Problem* oder häufiger als **Cache-Kohärenz-Problem** bezeichnet. Wir illustrieren das Problem an einem einfachen busbasierten System mit drei Prozessoren [31].

Beispiel: Ein busbasiertes SMP-System bestehe aus drei Prozessoren P_1, P_2, P_3 mit jeweils einem lokalen Cache C_1, C_2, C_3 . Die Prozessoren sind über einen zentralen Bus mit dem gemeinsamen Speicher M verbunden. Für die Caches nehmen wir eine Write-Through-Rückschreibestrategie an. Auf eine Variable u im Speicher M mit Wert 5 werden zu aufeinanderfolgenden Zeitpunkten $t_1, \dots, t_4$ die folgenden Operationen angewendet:

Zeitpunkt	Operation
t_1 :	Prozessor P_1 liest Variable u . Der Block, der Variable u enthält, wird daraufhin in den Cache C_1 geladen.
t_2 :	Prozessor P_3 liest Variable u . Der Block, der Variable u enthält, wird daraufhin in den Cache C_3 geladen.
t_3 :	Prozessor P_3 schreibt den Wert 7 in u . Die Veränderung wird aufgrund der Write-Through-Rückschreibestrategie auch im Speicher M vorgenommen.
t_4 :	Prozessor P_1 liest u durch Zugriff auf seinen Cache C_1 .

Der Prozessor P_1 liest also zum Zeitpunkt t_4 den alten Wert 5 statt den neuen Wert 7, was für weitere Berechnungen zu Fehlern führen kann. Dabei wurde angenommen, dass eine *write-through*-Rückschreibestrategie verwendet wird und daher zum Zeitpunkt t_3 der neue Wert 7 direkt in den Speicher zurückgeschrieben wird. Bei Verwendung einer *write-back*-Rückschreibestrategie würde zum Zeitpunkt t_3 der Wert von u im Speicher nicht aktualisiert werden, sondern erst beim Ersetzen des Blockes, in dem sich u befindet. Zum Zeitpunkt t_4 des Beispiels würde P_1 ebenfalls den falschen Wert lesen. $\square$

Um Programme in einem Programmiermodell mit gemeinsamem Adressraum auf Multiprozessoren korrekt ausführen zu können, muss gewährleistet sein, dass bei jeder möglichen Anordnung von Lese- und Schreibzugriffen, die von den einzelnen Prozessoren auf gemeinsamen Variablen durchgeführt werden, jeweils der richtige Wert gelesen wird, egal ob sich der Wert bereits im Cache befindet oder erst geladen werden muss. Das Verhalten eines Speichersystems bei Lese- und Schreibzugriffen von eventuell *verschiedenen* Prozessoren auf die *gleiche* Speicherzelle wird durch den Begriff der **Kohärenz des Speichersystems** beschrieben. Ein Speichersystem ist kohärent, wenn für jede Speicherzelle gilt, dass jede Leseoperation den *letzten* geschriebenen Wert zurückliefert. Da mehrere Prozessoren gleichzeitig oder fast gleichzeitig auf die gleiche Speicherzelle schreibend zugreifen können, ist zunächst zu präzisieren, welches der *zuletzt geschriebene* Wert ist. Als Zeitmaß ist in einem parallelen Programm nicht der Zeitpunkt des physikalischen Lesens oder Beschreibens einer Variable maßgeblich, sondern die Reihenfolge im zugrundeliegenden Programm. Dies wird in nachfolgender Definition berücksichtigt [75].

Ein Speichersystem ist **kohärent**, wenn die folgenden Bedingungen erfüllt sind:

1. Wenn ein Prozessor P die Speicherzelle x zum Zeitpunkt t_1 beschreibt und zum Zeitpunkt $t_2 > t_1$ liest und wenn zwischen den Zeitpunkten t_1 und t_2 kein anderer Prozessor die Speicherzelle x beschreibt, erhält Prozessor P zum Zeitpunkt t_2 *den von ihm geschriebenen* Wert zurück. Dies bedeutet, dass für jeden Prozessor die für ihn geltende Programmreihenfolge der Speicherzugriffe trotz der parallelen Ausführung erhalten bleibt.
2. Wenn ein Prozessor P_1 zum Zeitpunkt t_1 eine Speicherzelle x beschreibt und ein Prozessor P_2 zum Zeitpunkt $t_2 > t_1$ die Speicherzelle x liest, erhält P_2 den von P_1 geschriebenen Wert zurück, wenn zwischen t_1 und t_2 kein anderer Prozessor x beschreibt und wenn $t_2 - t_1$ genügend groß ist. Der neue Wert muss also nach einer *gewissen* Zeit für andere Prozessoren sichtbar sein.
3. Wenn zwei beliebige Prozessoren die gleiche Speicherzelle x beschreiben, werden diese Schreibzugriffe so *sequentialisiert*, dass alle Prozessoren die Schreibzugriffe in der *gleichen* Reihenfolge sehen. Diese Bedingung wird *globale Schreibsequentialisierung* genannt.

Bus-Snooping. In einem busbasierten SMP-System mit lokalen Caches und write-through-Rückschreibstrategie kann die Kohärenz der Caches durch *Bus-Snooping* sichergestellt werden. Diese Methode beruht auf der Eigenschaft eines Busses, dass alle relevanten Speicherzugriffe über den zentralen Bus erfolgen und von den *Cache-Controllern* aller anderen Prozessoren beobachtet werden können. Somit kann jeder Prozessor durch *Überwachung* der über den Bus ausgeführten Speicherzugriffe feststellen, ob durch den Speicherzugriff ein Wert in seinem lokalen Cache (der ja eine Kopie des Wertes im Hauptspeicher ist) aktualisiert werden sollte. Ist dies der Fall, so aktualisiert der beobachtende Prozessor den Wert in seinem lokalen Cache, indem er den an der Datenleitung anliegenden Wert kopiert. Die lokalen Caches enthalten so stets die aktuellen Werte. Wird obiges Beispiel unter Einbeziehung von Bus-Snooping betrachtet, so kann Prozessor P_1 den Schreibzugriff von P_3 beobachten und den Wert von Variable u im lokalen Cache C_1 aktualisieren.

Die Bus-Snooping-Technik beruht auf der Verwendung von Caches mit write-through-Rückschreibstrategie. Deshalb tritt bei der Bus-Snooping-Technik das Problem auf, dass viel Verkehr auf dem zentralen Bus stattfinden kann, da *jede* Schreiboperation über den Bus ausgeführt wird. Dies kann einen erheblichen Nachteil darstellen und zu Engpässen führen, was an folgendem Beispiel deutlich wird [31]. Wir betrachten ein Bussystem mit 2 GHz-Prozessoren, die eine Instruktion pro Zyklus ausführen. Verursachen 15% aller Instruktionen Schreibzugriffe mit 8 Bytes je Schreibzugriff, so erzeugt jeder Prozessor 300 Millionen Schreibzugriffe pro Sekunde. Jeder Prozessor würde also eine Busbandbreite von 2.4 GB/sec benötigen. Ein Bus mit einer Bandbreite von 10 GB/sec könnte dann also maximal vier Prozessoren ohne Auftreten von Staus (*congestion*) versorgen.

Eine Alternative stellt die Benutzung der write-back-Rückschreibestrategie mit *einem geeigneten Protokoll* dar. Wir geben im Folgenden ein Beispiel für ein solches Protokoll an. Für eine ausführlichere Behandlung verweisen wir auf [31].

Write-Back-Invalidierungs-Protokoll (MSI-Protokoll). Das Write-Back-Invalidierungs-Protokoll benutzt drei Zustände, die ein im Cache befindlicher Speicherblock annehmen kann, wobei der gleiche Speicherblock in unterschiedlichen Caches unterschiedlich markiert sein kann:

- M** für *modified* (modifiziert) bedeutet, dass nur der betrachtete Cache die aktuelle Version des Speicherblocks enthält und die Kopien des Blockes im Hauptspeicher und allen anderen Caches *nicht* aktuell sind,
- S** für *shared* (gemeinsam) bedeutet, dass der Speicherblock im unmodifizierten Zustand in einem oder mehreren Caches gespeichert ist und alle Kopien den aktuellen Wert enthalten,
- I** für *invalid* (ungültig) bedeutet, dass der Speicherblock im betrachteten Cache ungültige Werte enthält.

Diese drei Zustände geben dem MSI-Protokoll seinen Namen. Bevor ein Prozessor einen in seinem lokalen Cache befindlichen Speicherblock beschreibt, ihn also modifiziert, werden alle Kopien dieses Blockes in anderen Caches als *ungültig* (I) markiert. Dies geschieht durch eine Operation über den Bus. Der Cacheblock im eigenen Cache wird als modifiziert (M) markiert. Der zugehörige Prozessor kann nun mehrere Schreiboperationen durchführen, ohne dass eine weitere Busoperation nötig ist. Für die Verwaltung des Protokolls werden die drei folgenden Busoperationen bereitgestellt:

- a) **Bus Read (BusRd)**: Die Operation wird durch eine Leseoperation eines Prozessors auf einen Wert ausgelöst, der sich nicht im lokalen Cache befindet. Der zugehörige Cache-Controller fordert durch Angabe einer Hauptspeicheradresse eine Kopie eines Cacheblockes an, die er nicht modifizieren will. Das Speichersystem stellt den betreffenden Block aus dem Hauptspeicher oder einem anderen Cache zur Verfügung.
- b) **Bus Read Exclusive (BusRdEx)**: Die Operation wird durch eine Schreiboperation auf einen Speicherblock ausgelöst, der sich entweder nicht im lokalen Cache befindet oder nicht zum Modifizieren geladen wurde, d.h. nicht mit (M) markiert wurde. Der Cache-Controller fordert durch Angabe der Hauptspeicheradresse eine exklusive Kopie des Speicherblocks an, den er modifizieren will. Das Speichersystem stellt den Block aus dem Hauptspeicher oder einem anderen Cache zur Verfügung. Alle Kopien des Blockes in anderen Caches werden als ungültig (I) gekennzeichnet.
- c) **Write Back (BusWr)**: Der Cache-Controller schreibt einen als modifiziert (M) gekennzeichneten Cacheblock in den Hauptspeicher zurück. Die Operation wird ausgelöst durch das Ersetzen des Cacheblockes.

Der Prozessor selbst führt nur übliche Lese- und Schreiboperationen (**PrRd**, **PrWr**) aus, vgl. Abbildung 2.32 rechts. Der Cache-Controller stellt die vom Prozessor angefragten Speicherworte zur Verfügung, indem er sie entweder aus dem lokalen Cache lädt oder die zugehörigen Speicherblöcke mit Hilfe einer Busoperation besorgt. Die genauen Operationen und Zustandsübergänge sind in Abbildung 2.32 links angegeben.

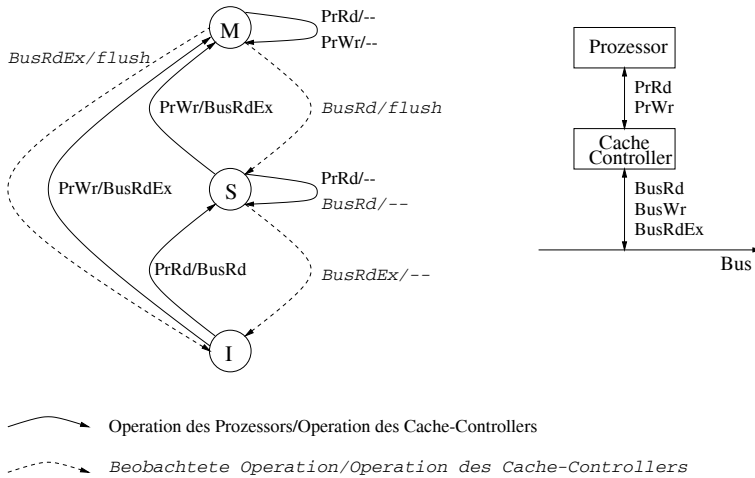


Abb. 2.32. Illustration des MSI-Protokolls: Die möglichen Zustände der Cacheblöcke eines Prozessors sind M (modified), S (shared) und I (invalid). Zustandsübergänge sind durch Pfeile angegeben, die durch Operationen markiert sind. Zustandsübergänge können ausgelöst werden durch: (a) (durchgezogene Pfeile) Operationen des eigenen Prozessors (**PrRd** und **PrWr**). Die entsprechende Busoperation des Cache-Controllers ist hinter dem Schrägstrich angegeben. Wenn keine Busoperation angegeben ist, muss nur ein Zugriff auf den lokalen Cache ausgeführt werden. (b) (gestrichelte Pfeile) Vom Cache-Controller auf dem Bus beobachtete Operationen, die durch andere Prozessoren ausgelöst sind. Die daraufhin erfolgende Operation des Cache-Controllers ist wieder hinter dem Schrägstrich angegeben. *flush* bedeutet hierbei, dass der Cache-Controller den gewünschten Wert auf den Bus legt. Wenn für einen Zustand für eine bestimmte Busoperation keine Kante angegeben ist, ist keine Aktion des Cache-Controllers erforderlich und es findet kein Zustandsübergang statt.

Das Lesen und Schreiben eines mit (M) markierten Cacheblockes kann ohne Busoperation im lokalen Cache vorgenommen werden. Dies gilt auch für das Lesen eines mit (S) markierten Cacheblockes. Zum Schreiben auf einen mit (S) markierten Cacheblock muss der Cache-Controller zuerst mit **BusRdEx** die alleinige Kopie des Cacheblockes erhalten. Die Cache-Controller anderer Prozessoren, die diesen Cacheblock ebenfalls mit (S) markiert in ihrem lokalen Cache haben, beobachten diese Operation auf dem Bus und markieren daraufhin ihre lokale Kopie als ungültig (I). Wenn ein Prozessor einen Speicherblock zu lesen versucht, der nicht in seinem lokalen Cache liegt oder

der dort als ungültig (I) markiert ist, besorgt der zugehörige Cache-Controller durch Ausführen einer **BusRd**-Operation eine Kopie des Speicherblockes und markiert sie im lokalen Cache als shared (S). Wenn ein anderer Prozessor diesen Speicherblock in seinem lokalen Cache mit (M) markiert hat, d.h. wenn er die einzig gültige Kopie dieses Speicherblockes hat, stellt der zugehörige Cache-Controller den Speicherblock auf dem Bus zur Verfügung und markiert seine lokale Kopie als shared (S). Wenn ein Prozessor einen Speicherblock zu schreiben versucht, der nicht in seinem lokalen Cache liegt oder der dort als ungültig (I) markiert ist, besorgt der zugehörige Cache-Controller durch Ausführen einer **BusRdEx**-Operation die alleinige Kopie des Speicherblockes und markiert sie im lokalen Cache als modified (M). Wenn ein anderer Prozessor diesen Speicherblock in seinem lokalen Cache mit (M) markiert hat, stellt der zugehörige Cache-Controller den Speicherblock auf dem Bus zur Verfügung und markiert seine lokale Kopie als ungültig (I).

Der Nachteil des beschriebenen Protokolls besteht darin, dass ein Prozessor, der zunächst ein Datum liest und dann beschreibt, zwei Busoperationen **BusRd** und **BusRdEx** auslöst, und zwar auch dann, wenn kein anderer Prozessor beteiligt ist. Dies trifft auch dann zu, wenn ein einzelner Prozessor ein sequentielles Programm ausführt, was für kleinere SMPs häufig vorkommt. Dieser Nachteil des MSI-Protokolls wird durch die Einführung eines weiteren Zustandes (E) für *exclusive* im sogenannten MESI-Protokoll ausgeglichen. Wenn ein Speicherblock in einem Cache mit

E für *exclusive* (exklusiv) markiert ist, bedeutet dies, dass nur der betrachtete Cache eine Kopie des Blockes enthält und dass diese Kopie *nicht* modifiziert ist, so dass auch der Hauptspeicher die aktuellen Werte dieses Speicherblockes enthält.

Wenn ein Prozessor einen Speicherblock zum Lesen anfordert und kein anderer Prozessor eine Kopie dieses Speicherblocks in seinem lokalen Cache hat, markiert der lesende Prozessor diesen Speicherblock mit (E) statt bisher mit (S), nachdem er ihn aus dem Hauptspeicher über den Bus erhalten hat. Wenn dieser Prozessor den mit (E) markierten Speicherblock zu einem späteren Zeitpunkt beschreiben will, kann er dies tun, nachdem er die Markierung des Blockes lokal von (E) zu (M) geändert hat. In diesem Fall ist also keine Busoperation nötig. Wenn seit dem ersten Lesen durch den betrachteten Prozessor ein anderer Prozessor lesend auf den Speicherblock zugegriffen hätte, wäre der Zustand von (E) zu (S) geändert worden und die für das MSI-Protokoll beschriebenen Aktionen würden ausgeführt. Für eine genauere Beschreibung verweisen wir auf [31]. Varianten des MESI-Protokolls werden in vielen Prozessoren verwendet und spielen auch für Multicore-Prozessoren eine große Rolle. Beispiele sind die Intel Pentium-Prozessoren.

Eine Alternative zu Invalidierungsprotokollen stellen **Write-Back-Update**-Protokolle dar. Bei diesen Protokollen werden nach Aktualisierung eines (mit (M) gekennzeichneten) Cacheblockes auch alle anderen Caches, die

diesen Block ebenfalls enthalten, aktualisiert. Die lokalen Caches enthalten also immer die aktuellen Werte. In der Praxis werden diese Protokolle aber meist nicht benutzt, da sie zu erhöhtem Verkehr auf dem Bus führen.

Cache-Kohärenz in nicht-busbasierten Systemen. Bei nicht-busbasierten Systemen kann Cache-Kohärenz nicht so einfach wie bei busbasierten Systemen realisiert werden, da kein zentrales Medium existiert, über das alle Speicheranfragen laufen. Der einfachste Ansatz besteht darin, *keine* Hardware-Cache-Kohärenz zur Verfügung zu stellen. Um Probleme mit der fehlenden Cache-Kohärenz zu vermeiden, können die lokalen Caches nur Daten aus den jeweils lokalen Speichern aufnehmen. Daten aus den Speichern anderer Prozessoren können nicht per Hardware im lokalen Cache abgelegt werden. Bei häufigen Zugriffen kann ein Aufheben im Cache aber per Software dadurch erreicht werden, dass die Daten in den lokalen Speicher kopiert werden. Dem Vorteil, ohne zusätzliche Hardware auszukommen, steht gegenüber, dass Zugriffe auf den Speicher anderer Prozessoren teuer sind. Bei häufigen Zugriffen muss der Programmierer dafür sorgen, dass die benötigten Daten in den lokalen Speicher kopiert werden, um von dort über den lokalen Cache schneller zugreifbar zu sein.

Die Alternative besteht darin, Hardware-Cache-Kohärenz mit Hilfe eines alternativen Protokolls zur Verfügung zu stellen. Dazu kann ein **Directory-Protokoll** eingesetzt werden. Die Idee besteht darin, ein zentrales Verzeichnis (engl. *directory*) zu verwenden, das den Zustand jedes Speicherblockes enthält. Anstatt den zentralen Bus zu beobachten, kann ein Cache-Controller den Zustand eines Speicherblockes durch ein Nachschauen in dem Verzeichnis erfahren. Dabei kann das Verzeichnis auf die verschiedenen Prozessoren verteilt werden, um zu vermeiden, dass der Zugriff auf das Verzeichnis zu einem Flaschenhals wird. Um dem Leser eine Idee von der Arbeitsweise eines Directory-Protokolls zu geben, beschreiben wir im Folgenden ein einfaches Schema. Für eine ausführlichere Beschreibung verweisen wir auf [31, 75].

Wir betrachten einen Shared-Memory-Rechner mit physikalisch verteiltem Speicher und nehmen an, dass zu jedem lokalen Speicher eine Tabelle (*Directory* genannt) gehalten wird, die zu jedem Speicherblock des lokalen Speichers angibt, in welchem Cache anderer Prozessoren dieser Speicherblock zur Zeit enthalten ist. Für eine Maschine mit p Prozessoren kann ein solches Directory dadurch realisiert werden, dass ein Bitvektor pro Speicherblock gehalten wird, der p *presence*-Bits und eine Anzahl von Statusbits enthält. Jedes der *presence*-Bits gibt für einen bestimmten Prozessor an, ob der zugehörige Cache eine gültige Kopie des Speicherblock enthält (Wert 1) oder nicht (Wert 0). Wir nehmen im Folgenden an, dass nur ein Statusbit (*dirty*-Bit) verwendet wird, das angibt, ob der Hauptspeicher die aktuelle Version des Speicherblockes enthält (Wert 0) oder nicht (Wert 1). Jedes Directory wird von einem eigenen Directory-Controller verwaltet, der auf die über das Netzwerk eintreffenden Anfragen wie im Folgenden beschrieben reagiert. Abbildung 2.33 veranschaulicht die Organisation. In den lokalen Caches sind

die Speicherblöcke wie oben beschrieben mit (M), (S) oder (I) markiert. Die Prozessoren greifen über ihre lokalen Cache-Controller auf die Speicheradressen zu, wobei wir einen globalen Adressraum annehmen.

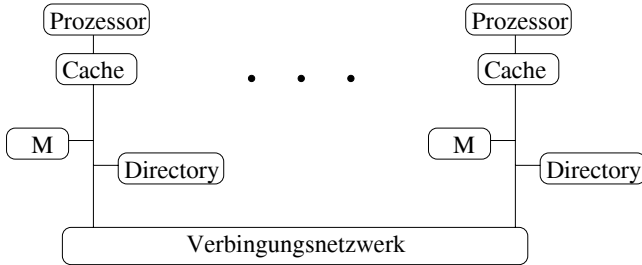


Abb. 2.33. Directory-basierte Cache-Kohärenz.

Bei Auftreten eines Cache-Fehlzugriffs bei einem Prozessor i greift der Cache-Controller von i über das Netzwerk auf das Directory zu, das die Informationen über den Speicherblock enthält. Wenn es sich um einen lokalen Speicherblock handelt, reicht ein lokaler Zugriff aus, ansonsten muss über das Netzwerk auf das entsprechende Directory zugegriffen werden. Wir beschreiben im Folgenden den Fall eines nicht-lokalen Zugriffs. Wenn es sich um einen Lese-Fehlzugriff handelt (engl. *read miss*), reagiert der zugehörige Directory-Controller wie folgt:

- Wenn das *dirty*-Bit des Speicherblockes auf 0 gesetzt ist, liest der Directory-Controller den Speicherblock aus dem zugehörigen Hauptspeicher mit Hilfe eines lokalen Zugriffs und schickt dem anfragenden Cache-Controller dessen Inhalt über das Netzwerk zu. Das *presence*-Bit des zugehörigen Prozessors wird danach auf 1 gesetzt, um zu vermerken, dass dieser Prozessor eine gültige Kopie des Speicherblockes hat.
- Wenn das *dirty*-Bit des Speicherblockes auf 1 gesetzt ist, gibt es genau einen Prozessor, der die aktuelle Version des Speicherblockes enthält, d.h. das *presence*-Bit ist für genau einen Prozessor j gesetzt. Der Directory-Controller schickt über das Netzwerk eine Anfrage an diesen Prozessor. Dessen Cache-Controller setzt den Zustand des Speicherblockes von (M) auf (S) und schickt dessen Inhalt an den ursprünglich anfragenden Prozessor i und an den Directory-Controller des Speicherblockes. Letzterer schreibt den aktuellen Wert in den zugehörigen Hauptspeicher, setzt das *dirty*-Bit auf 0 und das *presence*-Bit von Prozessor i auf 1. Das *presence*-Bit von Prozessor j bleibt auf 1.

Wenn es sich um einen Schreib-Fehlzugriff handelt (engl. *write miss*), reagiert der Directory-Controller wie folgt:

- Wenn das *dirty*-Bit des Speicherblockes auf 0 gesetzt ist, enthält der Hauptspeicher den aktuellen Wert des Speicherblockes. Der Directory-Controller schickt an alle Prozessoren j , deren *presence*-Bit auf 1 gesetzt ist, über das Netzwerk eine Mitteilung, dass deren Kopien als ungültig zu markieren sind. Die *presence*-Bits dieser Prozessoren werden auf 0 gesetzt. Nachdem der Erhalt dieser Mitteilungen von allen zugehörigen Cache-Controller bestätigt wurde, wird der Speicherblock an Prozessor i geschickt, dessen *presence*-Bit und das *dirty*-Bit des Speicherblockes werden auf 1 gesetzt. Nach Erhalt des Speicherblockes setzt der Cache-Controller von i dessen Zustand auf (M).
- Wenn das *dirty*-Bit des Speicherblockes auf 1 gesetzt ist, wird der Speicherblock über das Netzwerk von dem Prozessor j geladen, dessen *presence*-Bit auf 1 gesetzt ist. Dann wird der Speicherblock an Prozessor i geschickt, das *presence*-Bit von j wird auf 0, das *presence*-Bit von i auf 1 gesetzt. Das *dirty*-Bit bleibt auf 1 gesetzt.

Wenn ein Speicherblock im Cache eines Prozessors i ersetzt werden soll, in dem er als einzige Kopie liegt, also mit (M) markiert ist, wird er vom Cache-Controller von i an den zugehörigen Directory-Controller geschickt. Dieser schreibt den Speicherblock mit einer lokalen Operation in den Hauptspeicher zurück, setzt das *dirty*-Bit und das *presence*-Bit von i auf 0. Ein mit (S) markierter Speicherblock kann dagegen ohne Mitteilung an den Directory-Controller ersetzt werden. Eine Mitteilung an den Directory-Controller vermeidet aber, dass bei einem Schreib-Fehlzugriff wie oben beschrieben eine dann unnötige Invalidierungsnachricht an den Prozessor geschickt wird.

2.7.3 Speicherkonsistenz

Speicher- bzw. Cache-Kohärenz liegt vor, wenn jeder Prozessor das *gleiche eindeutige* Bild des Speichers hat, d.h. wenn jeder Prozessor zu jedem Zeitpunkt für jede Variable den gleichen Wert erhält wie alle anderen Prozessoren des Systems (bzw. erhalten würde, falls das Programm einen entsprechenden Zugriff auf die Variable enthalten würde). Die Speicher- oder Cache-Kohärenz sagt allerdings nichts über die Reihenfolge aus, in der die Auswirkungen der Speicheroperationen *sichtbar* werden. **Speicherkonsistenzmodelle** beschäftigen sich mit der Fragestellung, in welcher Reihenfolge die Speicherzugriffsoperationen eines Prozessors von den anderen Prozessoren beobachtet werden. Die verschiedenen Speicherkonsistenzmodelle werden gemäß der folgenden Kriterien charakterisiert.

1. Werden die Speicherzugriffsoperationen der einzelnen Prozessoren in deren Programmreihenfolge ausgeführt?
2. Sehen alle Prozessoren die ausgeführten Speicherzugriffsoperationen in der gleichen Reihenfolge?

Das folgende Beispiel zeigt die Vielfalt der möglichen Ergebnisse eines Programmes für Multiprozessoren, wenn verschiedene Reihenfolgen der Anweisungen der Programme der einzelnen Prozessoren (also Sequentialisierungen des Multiprozessorprogramms) betrachtet werden, siehe auch [85].

Beispiel: Drei Prozessoren P_1, P_2, P_3 führen ein Mehrprozessorprogramm aus, das die gemeinsamen Variablen x_1, x_2, x_3 enthält. Die Variablen x_1, x_2 und x_3 seien mit dem Wert 0 initialisiert. Die Programme der Prozessoren P_1, P_2, P_3 seien folgendermaßen gegeben:

Prozessor	P_1	P_2	P_3
Programm	(1) $x_1 = 1;$ (2) print $x_2, x_3;$	(3) $x_2 = 1;$ (4) print $x_1, x_3;$	(5) $x_3 = 1;$ (6) print $x_1, x_2;$

Nachdem die Prozessoren P_i den Wert x_i mit 1 beschrieben haben, werden die Werte der Variablen $x_j, j = 1, 2, 3, j \neq i$ ausgedruckt, $i = 1, 2, 3$. Die Ausgabe des Multiprozessorprogramms enthält also 6 Ausgabewerte, die jeweils den Wert 0 oder 1 haben können. Insgesamt gibt es $2^6 = 64$ Ausgabekombinationen bestehend aus 0 und 1, wenn jeder Prozessor seine Anweisungen in einer beliebigen Reihenfolge ausführen kann und wenn die Anweisungen der verschiedenen Prozessoren beliebig gemischt werden können. Dabei können verschiedene globale Auswertungsreihenfolgen zur gleichen Ausgabe führen. Führt jeder Prozessor seine Anweisungen in der vorgegebenen Reihenfolge aus, also z.B. Prozessor P_1 erst (1) und dann (2), so ist die Ausgabe 000000 *nicht* möglich, da zunächst ein Beschreiben zumindest einer Variable mit 1 vor einer Ausgabeoperation ausgeführt wird. Eine mögliche Sequentialisierung stellt die Reihenfolge (1), (2), (3), (4), (5), (6), dar. Die zugehörige Ausgabe ist 001011. $\square$

Sequentielles Konsistenzmodell - SC-Modell. Ein häufig verwendetes Speicherkonsistenzmodell ist das Modell der **sequentiellen Konsistenz** (engl. *sequential consistency*) [99], das von den verwendeten Konsistenzmodellen die stärksten Einschränkungen an die Reihenfolge der durchgeführten Speicherzugriffe stellt. Ein Multiprozessorsystem ist sequentiell konsistent, wenn die Speicheroperationen jedes Prozessors in der von seinem Programm vorgegebenen Reihenfolge ausgeführt werden und wenn der Gesamteffekt aller Speicheroperationen aller Prozessoren für alle Prozessoren in der *gleichen* sequentiellen Reihenfolge erscheint, die sich durch Mischung der Reihenfolgen der Speicheroperationen der einzelnen Prozessoren ergibt. Dabei werden die abgesetzten Speicheroperationen als *atomare* Operationen abgearbeitet. Eine Speicheroperation wird als atomar angesehen, wenn der Effekt der Operation für alle Prozessoren sichtbar wird, bevor die nächste Speicheroperation (irgendeines Prozessors des Systems) abgesetzt wird.

Der in der Definition der sequentiellen Konsistenz verwendete Begriff der *Programmreihenfolge* ist im Prinzip nicht exakt festgelegt. So kann u.a. die Reihenfolge der Anweisungen im *Quellprogramm* gemeint sein, oder aber auch die Reihenfolge von Speicheroperationen in einem von einem optimierenden

Compiler erzeugten Maschinenprogramm, das eventuell Umordnungen von Anweisungen zur besseren Auslastung des Prozessors enthält. Wir gehen im Folgenden davon aus, dass das sequentielle Konsistenzmodell sich auf die Reihenfolge im Quellprogramm bezieht, da der Programmierer sich nur an dieser Reihenfolge orientieren kann.

Im sequentiellen Speicherkonsistenzmodell werden also alle Speicheroperationen als atomare Operationen in der Reihenfolge des Quellprogramms ausgeführt und zentral sequentialisiert. Dies ergibt eine *totale Ordnung* der Speicheroperationen eines parallelen Programmes, die für alle Prozessoren des Systems gilt. Im vorherigen Beispiel entspricht die Ausgabe 001011 dem sequentiellen Speicherkonsistenzmodell, aber auch 111111. Die Ausgabe 011001 ist bei sequentieller Konsistenz dagegen nicht möglich.

Die totale Ordnung der Speicheroperationen ist eine stärkere Forderung als bei der im letzten Abschnitt beschriebenen Speicherkohärenz. Die Kohärenz eines Speichersystems verlangte eine Sequentialisierung der Schreiboperationen, d.h. die Ausführung von Schreiboperationen auf die *gleiche* Speicherzelle erscheinen für alle Prozessoren in der gleichen Reihenfolge. Die sequentielle Speicherkonsistenz verlangt hingegen, dass *alle* Schreiboperationen (auf beliebige Speicherzellen) für alle Prozessoren in der gleichen Reihenfolge ausgeführt erscheinen. Das folgende Beispiel zeigt, dass die Atomarität der Schreiboperationen wichtig für die Definition der sequentiellen Konsistenz ist und dass die Sequentialisierung der Schreiboperationen alleine für eine eindeutige Definition nicht ausreicht.

Beispiel: Drei Prozessoren P_1 , P_2 , P_3 arbeiten folgende Programmstücke ab. Die Variablen x_1 und x_2 seien mit 0 vorbesetzt.

Prozessor	P_1	P_2	P_3
Programm	(1) $x_1 = 1$;	(2) while($x_1 == 0$); (3) $x_2 = 1$;	(4) while($x_2 == 0$); (5) print(x_1);

Prozessor P_2 wartet, bis x_1 den Wert 1 erhält, und setzt x_2 dann auf 1; Prozessor P_3 wartet, bis x_2 den Wert 1 annimmt, und gibt dann den Wert von x_1 aus.

Unter Einbehaltung der Atomarität von Schreiboperationen würde die Reihenfolge (1), (2), (3), (4), (5) gelten und Prozessor P_3 würde den Wert 1 für x_1 ausdrucken, da die Schreiboperation (1) von P_1 auch für P_3 sichtbar sein muss, *bevor* Prozessor P_2 die Operation (3) ausführt. Reine Sequentialisierung von Schreibbefehlen einer Variable *ohne* die in der sequentiellen Konsistenz geforderte Atomarität und globale Sequentialisierung würde die Ausführung von (3) vor Sichtbarwerden von (1) für P_3 erlauben und damit die Ausgabe des Wertes 0 für x_1 möglich machen. Um dies zu verdeutlichen, untersuchen wir einen mit einem Directory-Protokoll arbeitenden Parallelrechner, dessen Prozessoren über ein Netzwerk miteinander verbunden sind. Wir nehmen an, dass ein Invalidierungsprotokoll auf Directory-Basis verwendet wird, um die Caches der Prozessoren kohärent zu halten. Weiter nehmen

wir an, dass zu Beginn der Abarbeitung des angegebenen Programmstücks die Variablen x_1 und x_2 mit 0 initialisiert seien und in den lokalen Caches der Prozessoren P_2 und P_3 aufgehoben werden. Die zugehörigen Speicherblöcke seien als shared (S) markiert.

Die Operationen jedes Prozessors werden in Programmreihenfolge ausgeführt und eine Speicheroperation wird erst nach Abschluss der vorangegangenen Operationen des gleichen Prozessors gestartet. Da über die Laufzeit der Nachrichten über das Netzwerk keine Angaben existieren, ist folgende Abarbeitungsreihenfolge möglich:

- 1) P_1 führt die Schreiboperation (1) auf x_1 aus. Da x_1 nicht im Cache von P_1 liegt, tritt ein Schreib-Fehlzugriff (*write miss*) auf, d.h. es erfolgt ein Zugriff auf den Directory-Eintrag zu x_1 und das Losschicken der Invalidierungsnachrichten an P_2 und P_3 .
- 2) P_2 führt die Leseoperation für (2) auf x_1 aus. Wir nehmen an, dass P_2 die Invalidierungsnachricht von P_1 bereits erhalten und den Speicherblock von x_1 bereits als ungültig (I) markiert hat. Daher tritt ein Lese-Fehlzugriff (*read miss*) auf, d.h. P_2 erhält den aktuellen Wert 1 von x_1 über das Netzwerk von P_1 und die Kopie im Hauptspeicher wird ebenfalls aktualisiert. Nachdem P_2 so den aktuellen Wert von x_1 erhalten und die while-Schleife verlassen hat, führt P_2 die Schreiboperation (3) auf x_2 aus. Dabei tritt wegen der Markierung mit (S) ein Schreib-Fehlzugriff (*write miss*) auf, was zum Zugriff auf den Directory-Eintrag zu x_2 führt und das Losschicken von Invalidierungsnachrichten an P_1 und P_3 bewirkt.
- 3) P_3 führt die Leseoperation (4) auf x_2 aus und erhält den aktuellen Wert 1 über das Netzwerk, da die Invalidierungsnachricht von P_2 bereits bei P_3 angekommen ist. Daraufhin führt P_3 die Leseoperation (5) auf x_1 aus und erhält den *alten* Wert 0 für x_1 aus dem lokalen Cache, da die Invalidierungsnachricht von P_1 noch *nicht* angekommen ist.

Das Verhalten bei der Ausführung von Anweisung (5) kann durch unterschiedliche Laufzeiten der Invalidisierungsnachrichten über das Netzwerk aufgelöst werden. Die sequentielle Konsistenz ist verletzt, da die Prozessoren unterschiedliche Schreibreihenfolgen sehen: Prozessor P_2 sieht die Reihenfolge $x_1 = 1, x_2 = 1$ und Prozessor P_3 sieht die Reihenfolge $x_2 = 1, x_1 = 1$ (da der *neue* Wert von x_2 , aber der *alte* Wert von x_1 gelesen wird). $\square$

Die sequentielle Konsistenz kann in einem parallelen System durch folgende *hinreichenden Bedingungen* sichergestellt werden [31, 42, 145]:

- 1) Jeder Prozessor setzt seine Speicheranfragen in seiner Programmreihenfolge ab (d.h. es sind *keine* sogenannten *out-of-order executions* erlaubt, vgl. Abschnitt 2.2).
- 2) Nach dem Absetzen einer Schreiboperation wartet der ausführende Prozessor, bis die Operation abgeschlossen ist, bevor er die nächste Speicheranfrage absetzt. Insbesondere müssen bei Schreiboperationen mit Schreib-

Fehlzugriffen alle Cacheblöcke, die den betreffenden Wert enthalten, als ungültig (I) markiert worden sein.

- 3) Nach dem Absetzen einer Leseoperation wartet der ausführende Prozessor, bis diese Leseoperation und die Schreiboperation, deren Wert diese Leseoperation zurückliefert, vollständig abgeschlossen sind und für *alle* anderen Prozessoren sichtbar sind.

Diese Bedingungen stellen keine Anforderungen an die spezielle Zusammenarbeit der Prozessoren, das Verbindungsnetzwerk oder die Speicherorganisation der Prozessoren. In dem obigen Beispiel bewirkt der Punkt 3) der hinreichenden Bedingungen, dass P_2 nach dem Lesen von x_1 wartet, bis die zugehörige Schreiboperation (1) *vollständig* abgeschlossen ist, bevor die nächste Speicherzugriffsoperation (3) abgesetzt wird. Damit liest Prozessor P_3 sowohl für x_1 als auch für x_2 bei beiden Zugriffen (4) und (5) entweder den alten oder den aktuellen Wert, d.h. die sequentielle Konsistenz ist gewährleistet.

Die sequentielle Konsistenz stellt ein für den Programmierer sehr einfaches Modell dar, birgt aber den Nachteil, dass alle Speicheranfragen atomar und nacheinander bearbeitet werden müssen und die Prozessoren dadurch evtl. recht lange auf den Abschluss der abgesetzten Speicheroperationen warten müssen. Zur Behebung der möglicherweise resultierenden Ineffizienzen wurden weniger strikte Konsistenzmodelle vorgeschlagen, die weiterhin ein intuitiv einfaches Modell der Zusammenarbeit der Prozessoren liefern, aber effizienter implementiert werden können. Wir geben im Folgenden einen kurzen Überblick und verweisen auf [31, 75] für eine ausführlichere Behandlung.

Abgeschwächte Konsistenzmodelle. Das Modell der sequentiellen Konsistenz verlangt, dass die Lese- und Schreib Anfragen, die von einem Prozessor erzeugt werden, die folgende Reihenfolge einhalten:

1. $R \rightarrow R$: Die Lesezugriffe erfolgen in Programmreihenfolge.
2. $R \rightarrow W$: Eine Lese- und eine anschließende Schreiboperation erfolgen in Programmreihenfolge. Handelt es sich um die gleiche Speicheradresse, so ist dies eine *Anti-Abhängigkeit* (engl. *anti-dependence*), in der die Schreiboperation von der Leseoperation abhängt.
3. $W \rightarrow W$: Aufeinanderfolgende Schreibzugriffe erfolgen in Programmreihenfolge. Ist hier die gleiche Speicheradresse angesprochen, so handelt es sich um eine *Ausgabe-Abhängigkeit* (engl. *output dependence*).
4. $W \rightarrow R$: Eine Schreib- und eine anschließende Leseoperation erfolgen in Programmreihenfolge. Bezieht sich dieses auf die gleiche Speicheradresse, so handelt es sich um eine *Fluss-Abhängigkeit* (engl. *true dependence*).

Wenn eine Abhängigkeit zwischen den Lese- und Schreiboperationen besteht, ist die vorgegebene Ausführungsreihenfolge notwendig, um die Semantik des Programmes einzuhalten. Wenn eine solche Abhängigkeit nicht besteht, wird die Ausführungsreihenfolge von dem Modell der sequentiellen Konsistenz verlangt. Abgeschwächte Konsistenzmodelle (engl. *relaxed consistency*) ver-

zichten nun auf einige der oben genannten Reihenfolgen, wenn die Datenabhängigkeiten dies erlauben.

Prozessor-Konsistenzmodelle (engl. *processor consistency*) verzichten auf die Ordnung 4., d.h. auf die Reihenfolge von atomaren Schreib- und Leseoperationen, um so die Latenz der Schreiboperation abzumildern: Obwohl ein Prozessor seine Schreiboperation noch nicht abgeschlossen hat, d.h. der Effekt für andere Prozessoren noch nicht sichtbar ist, kann er nachfolgende Leseoperationen ausführen, wenn es keine Datenabhängigkeiten gibt. Modelle dieser Klasse sind das **TSO-Modell** (*total store ordering*) und das **PC-Modell** (*processor consistency*). Im Unterschied zum TSO-Modell garantiert das PC-Modell *keine* Atomarität der Schreiboperationen. Der Unterschied zwischen sequentieller Konsistenz und dem TSO- oder dem PC-Modell wird im folgenden Beispiel verdeutlicht.

Beispiel: Zwei Prozessoren P_1 und P_2 führen folgende Programmstücke aus, wobei die Variablen x_1 und x_2 jeweils mit 0 initialisiert sind.

Prozessor	P_1	P_2
Programm	(1) $x_1 = 1$; (2) $\text{print}(x_2)$;	(3) $x_2 = 1$; (4) $\text{print}(x_1)$;

Im SC-Modell muss jede mögliche Reihenfolge Anweisung (1) vor Anweisung (2) und Anweisung (3) vor Anweisung (4) ausführen. Dadurch ist die Ausgabe 0 für x_1 und 0 für x_2 *nicht* möglich. Im TSO- und im PC-Modell ist jedoch die Ausgabe von 0 für x_1 und x_2 möglich, da z.B. Anweisung (3) nicht abgeschlossen sein muss, bevor P_1 die Variable x_2 für Anweisung (2) liest. $\square$

Partial-Store-Ordering (PSO)-Modelle verzichten auf die Bedingungen 4. und 3. obiger Liste der Reihenfolgebedingungen für das SC-Modell. In diesen Modellen können also auch Schreiboperationen in einer anderen Reihenfolge abgeschlossen werden als die Reihenfolge im Programm angibt, wenn keine Ausgabe-Abhängigkeit zwischen den Schreiboperationen besteht. Aufeinanderfolgende Schreiboperationen können also überlappt werden, was insbesondere beim Auftreten von Schreib-Fehlzugriffen zu einer schnelleren Abarbeitung führen kann. Wieder illustrieren wir den Unterschied zu den bisher vorgestellten Modellen an einem Beispiel.

Beispiel: Die Variablen x_1 und $flag$ seien mit 0 vorbesetzt. Die Prozessoren P_1 und P_2 führen folgende Programmstücke aus.

Prozessor	P_1	P_2
Programm	(1) $x_1 = 1$; (2) $flag = 1$;	(3) $\text{while}(flag == 0)$; (4) $\text{print}(x_1)$;

Im SC- und im PC- bzw. TSO-Modell ist die Ausgabe des Wertes 0 für x_1 *nicht* möglich. Im PSO-Modell kann die Schreiboperation (2) jedoch *vor* Schreiboperation (1) beendet sein und so die Ausgabe von 0 durch die Lese-

operation auf x_1 in (4) ermöglichen. Diese Ausgabe stimmt nicht unbedingt mit dem intuitiven Verständnis der Arbeitsweise des Programmstückes überein. $\square$

Weak-Ordering-Modelle verzichten zusätzlich auf die Bedingungen (1) und (2), garantieren also keinerlei Fertigstellungsreihenfolge der Operationen. Es werden aber zusätzlich Synchronisationsoperationen bereitgestellt, die sicherstellen, dass

- a) alle Lese- und Schreiboperationen, die in der Programmreihenfolge *vor* der Synchronisationsoperation liegen, fertiggestellt werden, bevor die Synchronisationsoperation ausgeführt wird, und dass
- b) eine Synchronisationsoperation fertiggestellt wird, bevor Lese- und Schreiboperationen ausgeführt werden, die in der Programmreihenfolge *nach* der Synchronisationsoperation stehen.

Die zunehmende Verbreitung von Parallelrechnern hat dazu geführt, dass viele moderne Mikroprozessoren zur Vereinfachung der Integration in Parallelrechner Unterstützung für die Realisierung eines Speicherkonsistenzmodells bereitstellen. Unterschiedliche Hardwarehersteller unterstützen dabei unterschiedliche Speicherkonsistenzmodelle, d.h. es hat sich z.Z. noch keine eindeutige Meinung durchgesetzt, welches der vorgestellten Konsistenzmodelle das beste ist. Sequentielle Konsistenz wird z.B. von SGI im MIPS R10000-Prozessor dadurch unterstützt, dass die Operationen eines Programmes in der Programmreihenfolge fertiggestellt werden, auch wenn in jedem Zyklus mehrere Maschinenbefehle an die Funktionseinheiten abgesetzt werden können. Die Intel Pentium Prozessoren unterstützen ein PC-Modell. Die SPARC-Prozessoren von Sun verwenden das TSO-Modell. Die Alpha-Prozessoren von DEC und die PowerPC-Prozessoren von IBM verwenden ein Weak-Ordering-Modell. Die von den Prozessoren unterstützten Speicherkonsistenzmodelle werden meistens auch von den Parallelrechnern verwendet, die diese Prozessoren als Knoten benutzen. Dies ist z.B. für die Sequent NUMA-Q der Fall, die Pentium-Pro-Prozessoren als Knoten verwenden.

2.8 Parallelität auf Threadebene

Parallelität auf Threadebene kann innerhalb eines Prozessorchips durch geeignete Architekturorganisation realisiert werden. Man spricht in diesem Fall von Threadparallelität auf Chipebene (engl. *Chip Multiprocessing*, CMP). Eine Möglichkeit zum Erreichen von CMP besteht darin, mehrere **Prozessorkerne** (engl. *execution cores*) mit allen Ausführungsressourcen dupliziert auf einen Prozessorchip zu integrieren. Die dadurch resultierenden Prozessoren werden auch als **Multicore-Prozessoren** bezeichnet, siehe Abschnitt 2.1.

Ein alternativer Ansatz besteht darin, mehrere Threads dadurch gleichzeitig auf einem Prozessor zur Ausführung zu bringen, dass der Prozessor je nach Bedarf per Hardware zwischen den zur Verfügung stehenden ausführungsbereiten Threads umschaltet. Dies kann auf verschiedene Weise geschehen [105]. Der Prozessor kann nach fest vorgegebenen Zeitintervallen zwischen den Threads umschalten, d.h. nach Ablauf eines Zeitintervalls wird der nächste Thread zur Ausführung gebracht. Man spricht in diesem Fall von **Zeitscheiben-Multithreading** (engl. *timeslice multithreading*). Zeitscheiben-Multithreading kann dazu führen, dass Zeitscheiben nicht effektiv genutzt werden, wenn z.B. ein Thread auf das Eintreten eines Ereignisses warten muss, bevor seine Zeitscheibe abgelaufen ist, so dass der Prozessor für den Rest der Zeitscheibe keine Berechnungen durchführen kann. Solche unnötigen Wartezeiten können durch den Einsatz von **ereignisbasiertem Multithreading** (engl. *switch-on-event multithreading*) vermieden werden. In diesem Fall kann der Prozessor beim Eintreten von Ereignissen mit langer Wartezeit wie z.B. bei Cache-Fehlzugriffen zu einem anderen ausführungsbereiten Thread umschalten. Ein weiterer Ansatz ist das **simultane Multithreading** (engl. *simultaneous multithreading*, SMT), bei dem mehrere Threads ohne explizites Umschalten ausgeführt werden. Wir gehen im folgenden Abschnitt auf diese Methode, die als **Hyperthreading-Technik** in Prozessoren zum Einsatz kommt, näher ein.

2.8.1 Simultanes Multithreading

Die Hyperthreading-Technologie basiert auf dem Duplizieren des Prozessorbereiches zur Ablage des Prozessorzustandes auf der Chipfläche des Prozessors. Dazu gehören die Benutzer- und Kontrollregister sowie der Interrupt-Controller mit den zugehörigen Registern. Damit erscheint der physikalische Prozessor aus der Sicht des Betriebssystems und des Benutzerprogramms als eine Ansammlung von **logischen Prozessoren**, denen Prozesse oder Threads zur Ausführung zugeordnet werden können. Diese können von einem oder mehreren Anwendungsprogrammen stammen.

Jeder logische Prozessor legt seinen Prozessorzustand in einem separaten Prozessorbereich ab, so dass beim Wechsel zu einem anderen Thread kein aufwendiges Zwischenspeichern des Prozessorzustandes im Speichersystem erforderlich ist. Die logischen Prozessoren teilen sich fast alle Ressourcen des physikalischen Prozessors wie Caches, Funktions- und Kontrolleinheiten und Bussystem. Die Realisierung der Hyperthreading-Technologie erfordert daher nur eine geringfügige Vergrößerung der Chipfläche. Für zwei logische Prozessoren wächst z.B. für einen Intel Xeon Prozessor die erforderliche Chipfläche um weniger als 5% [105, 166]. Die gemeinsamen Ressourcen des Prozessorchips werden den logischen Prozessoren reihum zugeteilt, so dass die logischen Prozessoren simultan zur Ausführung gelangen. Treten bei einem logischen Prozessor Wartezeiten auf, können die Ausführungs-Ressourcen den anderen logischen Prozessoren zugeordnet werden, so dass aus der Sicht des

physikalischen Prozessors eine fortlaufende Nutzung der Ressourcen gewährleistet ist.

Gründe für Wartezeiten eines logischen Prozessors können z.B. Cache-Fehlzugriffe, falsche Sprungvorhersage, Abhängigkeiten zwischen Instruktionen oder Pipeline-Hazards sein. Da auch der Instruktionscache von den logischen Prozessoren geteilt wird, enthält dieser Instruktionen mehrerer logischer Prozessoren. Versuchen logische Prozessoren gleichzeitig eine Instruktion aus dem Instruktionscache in ihr lokales Instruktionsregister zur Weiterverarbeitung zu laden, erhält einer von ihnen per Hardware eine Zugriffserlaubnis. Sollte auch im nächsten Zyklus wieder eine konkurrierende Zugriffsanfrage erfolgen, erhält ein anderer logischer Prozessor eine Zugriffserlaubnis, so dass alle logischen Prozessoren mit Instruktionen versorgt werden.

Untersuchungen zeigen, dass durch die fortlaufende Nutzung der Ressourcen durch zwei logische Prozessoren je nach Anwendungsprogramm Laufzeitverbesserungen zwischen 15% und 30% erreicht werden [105]. Da alle Berechnungsressourcen von den logischen Prozessoren geteilt werden, ist nicht zu erwarten, dass durch den Einsatz von mehr als zwei logischen Prozessoren eine darüberhinausgehende signifikante Laufzeitverbesserung erreicht werden kann. Die Hyperthreading-Technologie wird daher voraussichtlich auf wenige logische Prozessoren beschränkt bleiben und evtl. sogar zugunsten der Multicore-Prozessoren nicht weiter eingesetzt werden. Zum Erreichen einer Laufzeitverbesserung durch den Einsatz der Hyperthreading-Technologie ist es erforderlich, dass das Betriebssystem in der Lage ist, die logischen Prozessoren anzusteuern. Aus Sicht eines Anwendungsprogramms ist es erforderlich, dass für jeden logischen Prozessor ein separater Thread zur Ausführung bereitsteht, d.h. für die Implementierung des Programms müssen Techniken der parallelen Programmierung eingesetzt werden.

2.8.2 Multicore-Prozessoren

Nach dem Gesetz von Moore verdoppelt sich die Anzahl der Transistoren pro Prozessorchip alle 18-24 Monate. Dieser enorme Zuwachs macht es seit vielen Jahren möglich, die Leistung der Prozessoren so stark zu erhöhen, dass ein Rechner spätestens nach 5 Jahren als veraltet gilt und die Kunden in relativ kurzen Abständen einen neuen Rechner kaufen. Die Hardwarehersteller sind daher daran interessiert, die Leistungssteigerung der Prozessoren mit der bisherigen Geschwindigkeit beizubehalten, um einen Einbruch der Verkaufszahlen zu vermeiden.

Wie in Abschnitt 2.1 dargestellt, sind wesentliche Aspekte der Leistungssteigerung die Erhöhung der Taktrate und der interne Einsatz paralleler Abarbeitung von Instruktionen, z.B. durch das Duplizieren von Funktionseinheiten. Die Grenzen beider Entwicklungen sind jedoch abzusehen: Ein weiteres Duplizieren von Funktionseinheiten ist zwar möglich, bringt aber wegen vorhandener Abhängigkeiten zwischen Instruktionen kaum eine weitere Leistungssteigerung. Gegen eine weitere Erhöhung der Taktrate sprechen mehre-

re Gründe [93]: Ein Problem liegt darin, dass die Speicherzugriffsgeschwindigkeit nicht im gleichen Umfang wie die Prozessorgeschwindigkeit zunimmt, was zu einer Erhöhung der Zyklenanzahl pro Speicherzugriff führt. So brauchte z.B. um 1990 ein Intel i486 für einen Zugriff auf den Hauptspeicher zwischen 6 und 8 Maschinenzyklen, während 2006 ein Intel Pentium Prozessor über 220 Zyklen benötigte. Die Speicherzugriffszeiten entwickeln sich daher zum limitierenden Faktor für eine weitere Leistungssteigerung. Zum Zweiten wird die Erhöhung der Transistoranzahl durch eine Erhöhung der Packungsdichte erreicht, mit der aber auch eine Erhöhung der Wärmeentwicklung verbunden ist. Diese wird zunehmend zum Problem, da die notwendige Kühlung entsprechend aufwendiger wird. Zum Dritten wächst mit der Anzahl der Transistoren auch die prozessorinterne Leitungslänge für den Signaltransport, so dass die Signallaufzeit eine wesentliche Rolle spielt, vgl. auch Abschnitt 1.1.

Aus diesen Gründen ist eine Leistungssteigerung im bisherigen Umfang mit den bisherigen Mitteln nicht durchführbar. Stattdessen müssen neue Prozessorarchitekturen eingesetzt werden, wobei die Verwendung mehrerer Prozessorkerne auf einem Prozessorchip schon seit vielen Jahren als die vielversprechendste Technik angesehen wird. Die Idee besteht darin, anstatt eines Prozessorchips mit einer sehr komplexen internen Organisation mehrere Prozessorkerne mit einfacherer Organisation auf den Prozessorchip zu integrieren. Dies hat auch den Vorteil, dass der Stromverbrauch des Prozessorchips dadurch reduziert werden kann, dass vorübergehend ungenutzte Prozessorkerne abgeschaltet werden können [73].

Bei Multicore-Prozessoren werden mehrere Prozessorkerne auf einem Prozessorchip integriert. Jeder Prozessorkern stellt für das Betriebssystem einen separaten logischen Prozessor mit separaten Ausführungsressourcen dar, die getrennt angesteuert werden müssen. Das Betriebssystem kann damit verschiedene Anwendungsprogramme parallel zueinander zur Ausführung bringen. So kann z.B. eine Anzahl von Hintergrundanwendungen wie Viruserkennung, Verschlüsselung und Kompression parallel zu Anwendungsprogrammen des Nutzers ausgeführt werden [128]. Es ist aber mit Techniken der parallelen Programmierung auch möglich, ein rechenzeitintensives Anwendungsprogramm (wie Computerspiele, Bildverarbeitung oder naturwissenschaftliche Simulationsprogramme) auf mehreren Prozessorkernen parallel abzuarbeiten, so dass die Berechnungszeit im Vergleich zu einer Ausführung auf einem Prozessorkern reduziert werden kann. Es ist anzunehmen, dass in Zukunft die Nutzer von Standardprogrammen wie z.B. Computerspielen erwarten, dass diese die Berechnungsressourcen des Prozessorchips effizient ausnutzen, d.h. für die Implementierung der zugehörigen Programme müssen Techniken der parallelen Programmierung eingesetzt werden.

Für die Realisierung von Multicore-Prozessoren gibt es verschiedene Implementierungsvarianten, die sich in der Anzahl der Prozessorkerne, der Größe und Anordnung der Caches, den Zugriffsmöglichkeiten der Prozessorkerne auf die Caches und dem Einsatz von heterogenen Komponenten unter-

scheiden. Für die interne Organisation der Prozessorchips können verschiedene Entwürfe unterschieden werden, die sich in der Anordnung der Prozessorkerne und der Cachespeicher unterscheiden [94]. Dabei können grob drei unterschiedliche Architekturen unterschieden werden, siehe Abbildung 2.34, von denen auch Mischformen auftreten können.

Bei einem *hierarchischen Design* teilen sich mehrere Prozessorkerne mehrere Caches, die in einer baumartigen Konfiguration angeordnet sind, wobei die Größe der Caches von den Blättern zur Wurzel steigt. Die Wurzel repräsentiert die Verbindung zum Hauptspeicher. So kann z.B. jeder Prozessorkern einen separaten L1-Cache haben, sich aber mit anderen Prozessorkernen einen L2-Cache teilen, und alle Prozessorkerne können auf den externen Hauptspeicher zugreifen. Dies ergibt dann eine dreistufige Hierarchie. Dieses Konzept kann auf mehrere Stufen erweitert werden und ist in Abbildung 2.34 (links) für drei Stufen veranschaulicht. Zusätzliche Untersysteme können die Caches einer Stufe miteinander verbinden. Ein hierarchisches Design wird typischerweise für SMP-Konfigurationen verwendet. Ein Beispiel für ein hierarchisches Design ist der IBM Power5 Prozessor, der zwei 64-Bit superskalare Prozessorkerne enthält, von denen jeder zwei logische Prozessoren per Hyperthreading simuliert. Jeder Prozessorkern hat einen separaten L1-Cache (für Daten und Programme getrennt) und teilt sich mit dem anderen Prozessorkern einen L2-Cache (1.8 MB) sowie eine Schnittstelle zu einem externen 36 MB L3-Cache. Andere Prozessoren mit hierarchischem Design sind der Intel Core Duo Prozessor und der Sun Niagara Prozessor.

Bei einem *Pipeline-Design* werden die Daten durch mehrere Prozessorkerne schrittweise weiterverarbeitet, bis sie vom letzten Prozessorkern im Speichersystem abgelegt werden, vgl. Abbildung 2.34 (Mitte). Router-Prozessoren und Grafikchips arbeiten oft nach diesem Prinzip. Ein Beispiel sind die X10 und X11 Prozessoren von Xelerator zur Verarbeitung von Netzwerkpaketen. Der Xelerator X10q enthält z.B. 200 separate VLIW-Prozessorkerne, die in einer logischen linearen Pipeline miteinander verbunden sind. Die Pakete werden dem Prozessor über mehrere Netzwerkschnittstellen zugeführt und dann durch die Prozessorkerne schrittweise verarbeitet, wobei jeder Prozessorkern einen Schritt ausführt. Die X11 Netzwerkprozessoren haben bis zu 800 Pipeline-Prozessorkerne.

Bei einem *netzwerkbasierten Design* sind die Prozessorkerne und ihre lokalen Caches oder Speicher über ein Verbindungsnetzwerk mit den anderen Prozessorkernen des Chips verbunden, vgl. auch Abschnitt 2.5, so dass der gesamte Datentransfer zwischen den Prozessorkernen über das Verbindungsnetzwerk läuft, vgl. Abbildung 2.34 (rechts). Ein netzwerkorientiertes Design wird z.B. für den Intel Teraflop-Chip verwendet.

Das Potential der Multicore-Prozessoren wurde von Hardwareherstellern wie Intel und AMD erkannt und seit 2005 bieten viele Hardwarehersteller Prozessoren mit zwei oder mehr Kernen an. Ab Ende 2006 liefert Intel Quadcore-Prozessoren und ab 2008 wird mit der Auslieferung von Octocore-Prozessoren

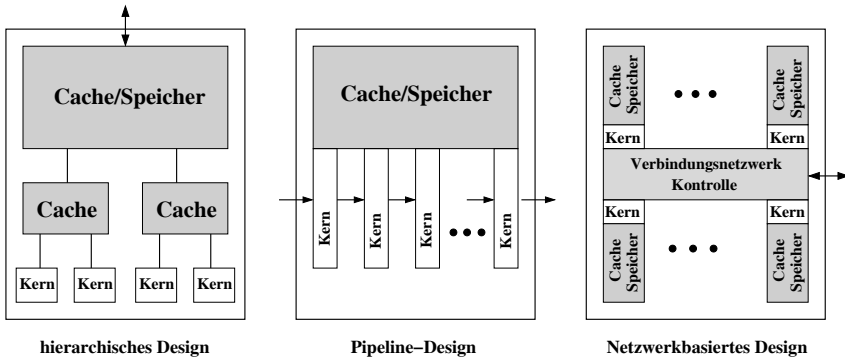


Abb. 2.34. Designmöglichkeiten für Multicore-Chips nach [94].

gerechnet. IBM bietet mit der Cell-Architektur einen Prozessor mit acht spezialisierten Prozessorkernen an, vgl. Abschnitt 2.9.2. Der seit Dezember 2005 ausgelieferte UltraSPARC T1 Niagara Prozessor von Sun hat bis zu acht Prozessorkerne, von denen jeder durch Einsatz von simultanem Multithreading, die von Sun als CoolThreads-Technologie bezeichnet wird, vier Threads simultan verarbeiten kann. Damit kann ein UltraSPARC T1 bis zu 32 Threads simultan ausführen. Das für 2008 angekündigte Nachfolgemodell Niagara II soll bis zu 16 Prozessorkerne enthalten.

Intel untersucht im Rahmen des *Tera-scale Computing Program* die Herausforderungen bei der Herstellung und Programmierung von Prozessoren mit Dutzenden von Prozessorkernen [74]. Diese Initiative beinhaltet auch die Entwicklung eines Teraflop-Prozessors, der 80 Prozessorkerne enthält, die als 8×10 -Gitter angeordnet sind. Jeder Prozessorkern kann Fließkommaoperationen verarbeiten und enthält neben lokalem Cachespeicher auch einen Router zur Realisierung des Datentransfers zwischen den Prozessorkernen und dem Hauptspeicher. Zusätzlich kann ein solcher Prozessor spezialisierte Prozessorkerne für die Verarbeitung von Videodaten, graphischen Berechnungen und zur Verschlüsselung von Daten enthalten. Je nach Einsatzgebiet kann die Anzahl der spezialisierten Prozessorkerne variiert werden.

Ein wesentlicher Bestandteil eines Prozessors mit einer Vielzahl von Prozessorkernen ist ein effizientes Verbindungsnetzwerk auf dem Prozessorchip, das an eine variable Anzahl von Prozessorkernen angepasst werden kann, den Ausfall einzelner Prozessorkerne toleriert und bei Bedarf das Abschalten einzelner Prozessorkerne erlaubt, falls diese für die aktuelle Anwendung nicht benötigt werden. Ein solches Abschalten ist insbesondere zur Reduktion des Stromverbrauchs sinnvoll.

Für eine effiziente Nutzung der Prozessorkerne ist entscheidend, dass die zu verarbeitenden Daten schnell genug zu den Prozessorkernen transportiert werden können, so dass diese nicht auf die Daten warten müssen. Dazu sind

ein leistungsfähiges Speichersystem und I/O-System erforderlich. Das Speichersystem setzt private L1-Caches, auf die nur von jeweils einem Prozessor-kern zugegriffen wird, sowie gemeinsame, evtl. aus mehreren Stufen bestehende L2-Caches ein, die Daten verschiedener Prozessorkerne enthalten. Für einen Prozessorchip mit Dutzenden von Prozessorkernen muss voraussichtlich eine weitere Stufe im Speichersystem eingesetzt werden [74]. Das I/O-System muss in der Lage sein, Hunderte von Gigabytes pro Sekunde auf den Prozessorchip zu bringen. Hier arbeitet z.B. Intel an der Entwicklung geeigneter Systeme.

2.9 Beispiele

Als Beispiel beschreiben wir im Folgenden kurz die Verwendung paralleler Abarbeitung in ausgewählten Prozessoren und parallelen Systemen. Dazu betrachten wir die Architektur des Intel Pentium 4 Prozessors und des IBM Cell-Prozessors und skizzieren die Architektur des IBM Blue Gene/L Parallelrechners.

2.9.1 Architektur des Intel Pentium 4

Als Beispiele für superskalare Prozessoren betrachten wir im Folgenden den Intel Pentium 4. Für eine ausführlichere Darstellung verweisen wir auf [75, 121]. Der Intel Pentium 4 ist ein CISC-Prozessor mit einem RISC-Kern. Die Dekodiereinheit wandelt 80x86-Instruktionen (variabler Länge und stark unterschiedlicher Komplexität) in Mikrobefehle (RISC-Instruktionen) konstanter Länge um, die weiterverarbeitet werden. Im Gegensatz zu den Vorgängern Pentium Pro, Pentium II und Pentium III, die auf der P6-Architektur basierten, benutzt der Pentium 4 die NetBurst-Architektur. Diese besteht aus vier Hauptteilen: dem Speichersystem, dem Front-End, dem Instruktionen-Scheduler und den Funktionseinheiten [159]. Abbildung 2.35 gibt einen Überblick der Prozessorarchitektur.

Das Speichersystem umfasst den L2-Cache, der Daten und Instruktionen enthält, sowie die Kontrolle für den Zugriff auf Daten des Hauptspeichers über den Speicherbus. Von der ersten zur dritten Generation der Pentium 4 Prozessoren wuchs die Größe des L2-Caches von 256KB über 512KB auf 1MB. Der L2-Cache ist 8-Wege-assoziativ, verwendet Cacheblöcke der Größe 128 Bytes und eine Write-back-Rückschreibestrategie. Das Speichersystem enthält weiterhin eine Prefetch-Einheit, die versucht, Daten aus dem Hauptspeicher frühzeitig in den L2-Cache zu laden, um so Wartezeiten zu vermeiden.

Das Front-End lädt 80x86-Instruktionen aus dem L2-Cache und dekodiert sie in der Reihenfolge, in der sie im Programm stehen. Jede Instruktion wird bei der Dekodierung in eine Folge von Mikrobefehlen zerlegt, die in einem separaten Trace-Cache abgelegt werden. Für komplexere 80x86-Instruktionen,

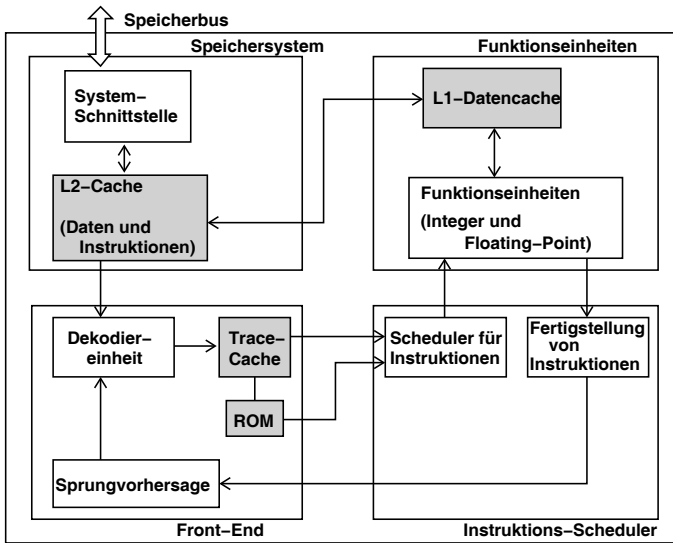


Abb. 2.35. NetBurst-Architektur des Intel Pentium 4.

die mehr als vier Mikrobefehle erfordern, werden die zugehörigen Mikrobefehle über einen ROM-Speicher geladen. Der Trace-Cache, in dem die Mikrobefehle abgelegt werden, ist eine Art Instruktionscache, dessen Blöcke eine dynamisch ausgewählte Folge von Instruktionen enthalten können, so dass der Programmfluss berücksichtigt werden kann. Im Gegensatz zu einem normalen Instruktionscache, dessen Blöcke Instruktionen in der Reihenfolge enthalten müssen, in der sie in den zugehörigen Speicherblöcken stehen, hat ein Trace-Cache den Vorteil einer effektiveren Ausnutzung der Blöcke. Vom Trace-Cache werden die Mikrobefehle an den Instruktions-Scheduler weitergeleitet, der sie je nach Verfügbarkeit an freie Funktionseinheiten weiterleitet. Dabei können die Instruktionen in einer gegenüber dem ursprünglichen Programmcode geänderten Reihenfolge abgesetzt werden.

Um eine korrekte Behandlung von Interrupts sicherzustellen, sorgt eine Fertigstellungseinheit dafür, dass die 80x86-Instruktionen in der Programmreihenfolge fertiggestellt werden.

Abbildung 2.36 veranschaulicht die Datenpfade der NetBurst-Architektur zur Abarbeitung von Instruktionen mit Front-End und Scheduler. Die Dekodiereinheit lädt jeweils 64 Bits aus dem L2-Cache, dekodiert diese und legt die zugehörigen Mikrobefehle im Trace-Cache ab. Trifft die Dekodiereinheit auf einen bedingten Sprung, wird das voraussichtliche Sprungziel über den L1 Sprungzielpuffer (engl. *Branch Target Buffer*, BTB) ermittelt und die Dekodierung wird mit der Sprungziel-Instruktion fortgesetzt. Der Trace-BTB wird für die Sprungvorhersage der Mikrobefehle verwendet. Pro Maschinenzklus werden drei Mikrobefehle aus dem Trace-Cache an die Allokierungseinheit

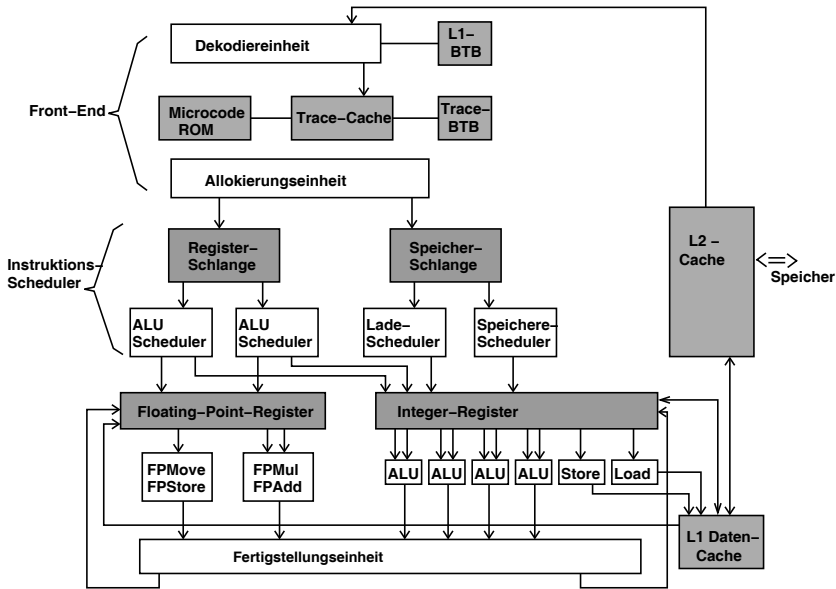


Abb. 2.36. Datenpfade des Intel Pentium 4.

weitergegeben. Diese leitet die Mikrobefehle entsprechend ihres Typs an die Speicherschlange (für Speicherzugriffsbefehle) oder die Registerschlange (für arithmetische-logische Befehle) weiter. Von dort müssen die Mikrobefehle nicht unbedingt in Programmreihenfolge an die Funktionseinheiten weitergegeben werden. Die Weitergabe erfolgt durch vier Schedulingeinheiten für ALU-Instruktionen sowie Lese- und Schreib-Instruktionen. Pro Maschinenzyklus können vier ALU-Instruktionen verarbeitet werden.

Speicherzugriffsoperationen laufen über den L1-Daten-Cache. Dieser ist 8K groß und enthält Integer- und Floating-Point-Daten. Der L1-Daten-Cache ist 4-Wege-assoziativ, verwendet Blöcke der Größe 64 Bytes und eine Write-through-Rückschreibestrategie. Pro Maschinenzyklus kann der L1-Cache eine Lese- und eine Schreiboperation erledigen. Da die 80x86-Instruktionen in der Programmreihenfolge beendet werden sollen, wird eine Schreiboperation in den L1-Cache erst ausgeführt, wenn alle vorangehenden Instruktionen abgeschlossen sind. Das dabei notwendige Zwischenspeichern von Instruktionen erfolgt durch die Fertigstellungseinheit. Tabelle 2.3 vergleicht die Speicherhierarchie des Intel Pentium 4 mit der des AMD Opteron-Prozessors, vgl. auch [121]. Die Angaben beziehen sich auf die Standardvarianten aus dem Jahr 2005. Es werden weitere Varianten mit größeren Caches angeboten, die auch einen auf der Chipfläche integrierten L3-Cache umfassen.

Tabelle 2.3. Charakteristika der Speicherhierarchie des Intel Pentium 4 und des AMD Opteron Prozessors.

Charakteristik	Intel Pentium 4	AMD Opteron
L1-Datencachegröße	8 KB	64 KB
L1-Instruktionscachegröße	96 KB Trace-Cache	64 KB
L1-Assoziativität	4-Wege-assoziativ	2-Wege-assoziativ
L1-Ersetzungsstrategie	approximiertes LRU	LRU
L1-Blockgröße	64 Bytes	64 Bytes
L1-Rückschreibstrategie	Write-through	Write-back
L2-Cache-Organisation	Instr. und Daten	Instr. und Daten
L2-Cachegröße	512 KB	1 MB
L2-Assoziativität	8-Wege-assoziativ	16-Wege-assoziativ
L2-Ersetzungsstrategie	approximiertes LRU	approximiertes LRU
L2-Blockgröße	128 Bytes	64 Bytes
L2-Rückschreibstrategie	Write-back	Write-back

2.9.2 Architektur des Cell-Prozessors

Als Beispiel für die Architektur eines Multicore-Prozessors stellen wir den von IBM in Zusammenarbeit mit Sony und Toshiba entwickelten Cell-Prozessor vor. Der Prozessor wird u.a. von Sony in der Spielekonsole PlayStation 3 eingesetzt, siehe auch [97, 90] für ausführlichere Informationen. Der Cell-Prozessor enthält ein *Power Processing Element* (PPE) und 8 *Single-Instruction Multiple-Datastream* (SIMD) Prozessoren. Das PPE ist ein konventioneller 64-Bit-Mikroprozessor auf der Basis der Power-Architektur von IBM mit relativ einfachem Design: der Prozessor kann pro Takt zwei Instruktionen absetzen und kann simultan zwei unabhängige Threads ausführen. Die einfache Struktur hat den Vorteil, dass trotz hoher Taktrate eine geringe Leistungsaufnahme resultiert. Für den gesamten Prozessor soll bei einer Taktrate von ca. 4 GHz nur eine Leistungsaufnahme von 60-80 Watt erforderlich sein.

Auf der Chipfläche des Cell-Prozessors sind neben dem PPE acht SIMD-Prozessoren integriert, die als SPE (*Synergetic Processing Element*) bezeichnet werden. Jedes SPE ist ein unabhängiger Vektorprozessor mit einem 256KB großen lokalem SRAM-Speicher, der als LS (*Local Store*) bezeichnet wird. Das Laden von Daten in den LS und das Zurückspeichern von Resultaten aus dem LS in den Hauptspeicher muss per Software erfolgen.

Jedes SPE enthält 128 128-Bit-Register, in denen die Operanden von Instruktionen abgelegt werden können. Da auf die Daten in den Registern sehr schnell zugegriffen werden kann, reduziert die große Registeranzahl die Notwendigkeit von Zugriffen auf den LS und führt damit zu einer geringen mittleren Speicherzugriffszeit. Jedes SPE hat vier Floating-Point-Einheiten (32 Bit) und vier Integer-Einheiten. Zählt man eine Multiply-Add-Instruktion als zwei Operationen, kann jedes SPE bei 4 GHz Taktrate pro Sekunde 32 Milliarden Floating-Point-Operationen (32 GFlops) und 32 Milliarden Integer-Operationen (32 GOPS) ausführen. Da ein Cell-Prozessor acht SPE enthält,

führt dies zu einer maximalen Performance von 256 GFlops, wobei die Leistung des PPE noch nicht berücksichtigt wurde. Eine solche Leistung kann allerdings nur bei guter Ausnutzung der LS-Speicher und effizienter Zuordnung von Instruktionen an Funktionseinheiten der SPE erreicht werden. Zu beachten ist auch, dass sich diese Angabe auf 32-Bit Floating-Point-Zahlen bezieht. Der Cell-Prozessor kann durch Zusammenlegen von Funktionseinheiten auch 64-Bit Floating-Point-Zahlen verarbeiten, dies resultiert aber in einer wesentlich geringeren maximalen Performance. Zur Vereinfachung der Steuerung der SPEs und zur Vereinfachung des Scheduling verwenden die SPEs intern keine Hyperthreading-Technik.

Die zentrale Verbindungseinheit des Cell-Prozessors ist ein Bussystem, der sogenannte Element Interconnect Bus (EIB). Dieser besteht aus vier unidirektionalen Ringverbindungen, die jeweils 16 Bytes breit sind und mit der halben Taktrate des Prozessors arbeiten. Zwei der Ringe werden in entgegengesetzter Richtung der anderen beiden Ringe betrieben, so dass die maximale Latenz im schlechtesten Fall durch einen halben Ringdurchlauf bestimmt wird. Für den Transport von Daten zwischen benachbarten Ringelementen können maximal drei Transferoperationen simultan durchgeführt werden, für den Zyklus des Prozessors ergibt dies $16 \cdot 3/2 = 24$ Bytes pro Zyklus. Für die vier Ringverbindungen ergibt dies eine maximale Transferrate von 96 Bytes pro Zyklus, woraus bei einer Taktrate von 4 GHz eine maximale Transferrate von 384 GBytes/Sekunde resultiert.

Abbildung 2.37 zeigt einen schematischen Aufbau des Cell-Prozessors mit den bisher beschriebenen Elementen sowie dem Speichersystem (Memory Interface Controller, MIC) und dem I/O-System (Bus Interface Controller, BIC). Das Speichersystem unterstützt die XDR-Schnittstelle von Rambus. Das I/O-System unterstützt das Rambus RRAC (Redwood Rambus Access Cell). Die Tabelle in Abbildung 2.38 vergleicht die Prozessoren der Spielkonsolen Playstation 2 und 3 von Sony.

Zum Erreichen einer guten Performance ist es wichtig, die SPEs des Cell-Prozessors effizient auszunutzen. Dies kann für spezialisierte Programme, wie z.B. Videospiele, durch direkte Verwendung von SPE-Assembleranweisungen erreicht werden. Da dies für die meisten Anwendungsprogramme zu aufwendig ist, ist eine effektive Compilerunterstützung sowie die Verwendung spezialisierter Programmbibliotheken z.B. zur Verwaltung von Tasksschlangen wichtig, vgl. auch Kapitel 6.

2.9.3 IBM Blue Gene/L Supercomputer (BG/L)

Das Entwurfsziel des IBM BG/L bestand darin, einen Supercomputer mit Prozessoren einfacher Bauart und moderater Taktrate zu entwickeln. Dies führt zu einer niedrigen Leistungsaufnahme und einem hohen Faktor für Performance pro Watt, siehe [54] für einen genauen Überblick.

Der BG/L Computer enthält maximal 65536 Knoten mit je zwei Prozessoren und neun SDRAM-Chips (synchronous dynamic random access memory).

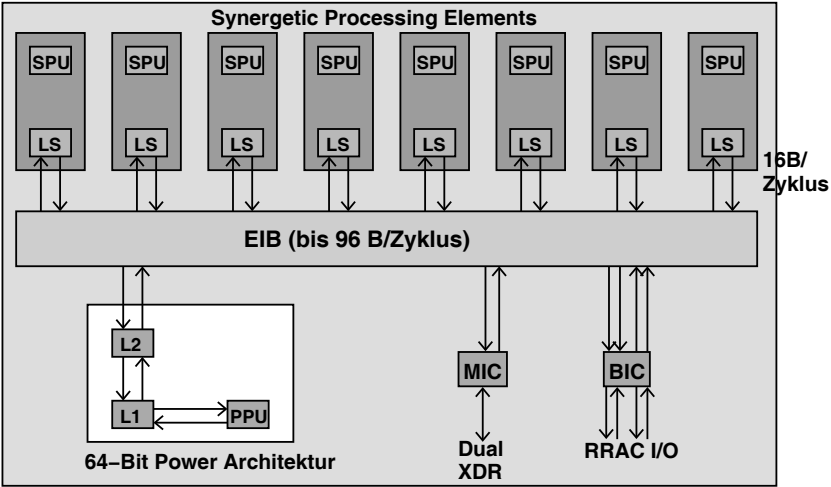


Abb. 2.37. Schematischer Aufbau des Cell-Prozessors.

	Sony Emotion Engine	Cell-Prozessor
CPU-ISA	MIPS64	64-Bit Power
Issue-Rate	dual	dual
Taktfrequenz	300 MHz	ca. 4 GHz
Instruktionspipeline	6 Stufen	21 Stufen
L1 Cache	16KB I-Cache + 8KB D-Cache	32KB I-Cache + 32KB D-Cache
zusätzl. Prozessorspeicher	16KB Skrtach	512KB L2-Cache
Vektoreinheiten	2	8
Vektorregister	32 (128Bit) + 16 (16Bit)	128 (128Bit)
lokaler Vektorspeicher	4K/16KB I-Cache + 4K/16KB D-Cache	256KB unified
Speicherbandbreite	3.2 GB/s max.	ca. 25.6 GB/s max.
Peak FLOPS	6.2 GFLOPS	256 GFLOPS
Anzahl Transistoren	10.5 Millionen	235 Millionen
Leistungsaufnahme	15 Watt	ca. 80 Watt
Chipfläche	240mm ²	235mm ²

Abb. 2.38. Vergleich der Prozessoren der Sony Playstations 2 und 3.

Zusätzlich gibt es I/O-Knoten für den Zugriff auf ein Dateisystem. Für die Realisierung von Interprozess-Kommunikation, I/O und Debugging stehen insgesamt fünf verschiedene Netzwerke zur Verfügung:

- Das 3D-Torus-Netzwerk der Dimension $64 \times 32 \times 32$ in der vollen Ausbaustufe dient zur Übertragung von Punkt-zu-Punkt-Nachrichten. Pro Verbindung stellt dieses Netzwerk eine bidirektionale Übertragungsrate von 1.4 Gbit/s zur Verfügung.

Die Latenzzeit beträgt für jede Verbindung ca. 100 ns. Zwischen zwei Prozessoren müssen in dem 3D-Torus-Netzwerk maximal $32+16+16=64$ Schritte zurückgelegt werden. Da pro Schritt eine Latenzzeit von ca. 100ns anfällt, resultiert eine maximale Latenzzeit von ca. $6.4\mu\text{s}$.

- Das Broadcast-Netzwerk dient der Realisierung globaler Kommunikationsoperationen. Dieses Netzwerk stellt eine Hardwareunterstützung für arithmetische und logische Reduktionsoperationen zur Verfügung. Die Übertragungsrate beträgt 2.8 Gbit/s, die Latenzzeit ist $5\mu\text{s}$.
- Das Barrier-Netzwerk kann für die Realisierung einer globalen Synchronisation verwendet werden. Die Latenzzeit liegt bei weniger als $1.5\mu\text{s}$ und ist damit wesentlich kleiner als die Latenzzeiten des Torus- und des Broadcast-Netzwerks.
- Das Kontrollnetzwerk wird zur Überwachung aller Systemkomponenten, die auch z.B. Temperatursensoren, Lüfter und Netzteile umfassen, durch einen Service-Knoten eingesetzt.
- Das Gigabit-Ethernet-Netzwerk verbindet die I/O-Knoten mit dem externen Dateisystem.

Jeder BG/L-Knoten ist als SoC-Design (System-on-a-Chip) auf einem ASIC-Chip (Application Specific Integrated Circuit) aufgebaut. Jeder Knoten enthält zwei PowerPC 440 Prozessorkerne (PPC 440) und zwei Floating-Point-Kerne mit je zwei gekoppelten Floating-Point-Einheiten (FPU). Dadurch ergibt sich eine maximale Performance von vier Floating-Point-Operationen pro Maschinenzyklus. Jeder PPC 440 ist ein im Vergleich zu anderen Prozessoren wie dem Intel Pentium 4 einfach strukturierter superskalarer Prozessor mit 700 MHz Taktrate. Der Prozessor kann zwei Instruktionen pro Zyklus absetzen und arbeitet mit einer Pipeline der Tiefe 7.

Jeder PPC 440 hat 32 32-Bit-Register und unabhängig 32 KBytes große L1 Instruktions- und Daten-Caches (Blockgröße 32 Bytes, 64-Wege-assoziativ), die im Write-through- und im Write-back-Modus arbeiten können. Der prozessorlokale Bus (PCB) ist 128 Bit breit. Die Memory Management Unit (MMU) verwendet einen Transaction Lookaside Buffer (TLB) mit 64 Einträgen. Abb. 2.39 zeigt den schematischen Aufbau eines BG/L-Knotens mit Netzwerkanbindung und Cachespeichern. Nur die I/O-Knoten sind an das Gigabit-Ethernet-Netzwerk angeschlossen. Jeder Knoten eines BG/L-Systems hat 512 MBytes Speicher. Dieser Speicher wird von den beiden Prozessoren des Knotens geteilt. Bei einer maximalen Knotenanzahl von 65536 ergibt sich eine maximale Speichergröße von 32 TBytes. Abb. 2.40 zeigt einen genaueren Blick auf den PPC 440 mit internen L1-Caches, Speicherverwaltung und CPU-Aufbau.

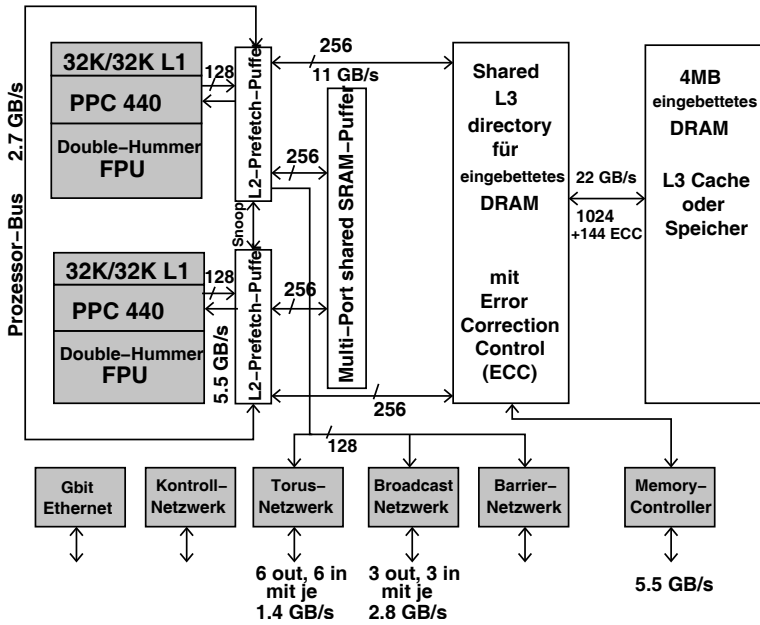


Abb. 2.39. Schematischer Aufbau eines BG/L-Knotens.

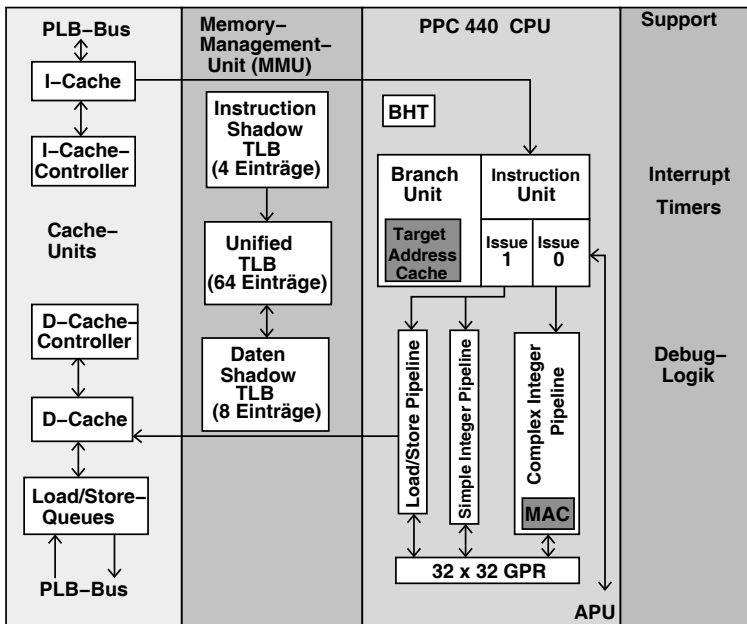


Abb. 2.40. Schematischer Aufbau des PPC 440.

3. Parallele Programmiermodelle

Das Erstellen eines parallelen Programms orientiert sich stark an dem zu benutzenden **parallelen Rechnersystem**. Ein Rechnersystem ist ein allgemeiner Ausdruck für die Gesamtheit von Hardware und Systemsoftware (Betriebssystem, Programmiersprache, Compiler, Laufzeitbibliothek), die dem Programmierer zur Verfügung steht und seine „Sicht“ auf den Rechner bestimmt. Für die gleiche Hardware können durch Verwendung unterschiedlicher Systemsoftwarekomponenten unterschiedliche parallele Rechnersysteme resultieren. Aufgrund der derzeit existierenden Vielfalt paralleler Hardware- und Systemsoftwarekomponenten gibt es eine Vielzahl unterschiedlicher paralleler Rechnersysteme, die jeweils andere Anforderungen hinsichtlich der Nutzung und Effizienz stellen. Dementsprechend können parallele Programme für ein und denselben sequentiellen Algorithmus je nach Anforderung des benutzten parallelen Systems sehr unterschiedlich sein. Um bei der parallelen Programmierung hardwareunabhängige Prinzipien und Programmiermethoden anwenden zu können, wird versucht, anstatt einzelner paralleler Rechnersysteme ganze Klassen von in mancher Hinsicht gleichen Systemen zu betrachten. So werden z.B. Rechnersysteme mit gemeinsamem und Rechnersysteme mit verteiltem Adressraum in jeweils einer Klasse zusammengefasst.

Die Klassifizierung der Rechnersysteme wird durch die Definition von *Modellen* erreicht, die Rechnersysteme auf einem gewissen Abstraktionsniveau beschreiben und die den Entwurf und die Analyse von parallelen Algorithmen oder Programmen erlauben. Die Analyse kann die asymptotische oder approximative Abschätzung der Ausführungszeit eines Programms sein oder auch die Analyse von Programmeigenschaften, wie z.B. die Möglichkeit des Auftretens von Deadlocksituationen, umfassen. Je nach Abstraktionsniveau können bestimmte Details des Rechnersystems in einem Modell unberücksichtigt bleiben, wenn die von diesen Details verursachten Phänomene für die durchzuführende Analyse irrelevant sind oder vernachlässigt werden können. Für die Programmierung steht idealerweise für jede Klasse von Rechnersystemen eine Programmiersprache oder eine portable Laufzeitbibliothek zur Verfügung. Für Rechnersysteme mit gemeinsamem Adressraum stellen etwa Realisierungen des Pthreads-Standards, Java-Threads oder OpenMP eine solche portable Laufzeitbibliothek dar, vgl. Kapitel 6. Für Rechnersysteme mit verteiltem Adressraum liefern MPI- oder PVM-Implementierungen einen

solchen Standard, vgl. die Abschnitte 5.1 und 5.2. Von Modellen für Rechnersysteme wird verlangt, dass sie auf der einen Seite einfach zu handhaben sind, auf der anderen Seite aber einen realen Rechner so genau beschreiben, dass ein nach den Effizienzkriterien des Modells entworfenes paralleles Programm auch auf dem realen Rechner ein effizientes Programm ergibt. Im folgenden Abschnitt 3.1 gehen wir nochmal detailliert auf den Modellbegriff ein.

3.1 Modelle paralleler Rechnersysteme

Für das sequentielle Rechnen ist die von-Neumann-Architektur die Grundlage der Programmierung, vgl. Abschnitt 2.3. Modelle für sequentielle Rechnersysteme unterscheiden sich im Wesentlichen in der Abstraktionsebene der Beschreibung dieser Rechnerarchitektur, nicht aber in der grundlegenden Struktur, was eine weitere Klassifizierung der Modelle nie nötig machte. Für parallele Rechnersysteme hingegen sind eine Vielzahl von Ausprägungen zu betrachten, z.B. bzgl. der Kontroll- oder Speicherorganisation, so dass auf den verschiedenen Abstraktionsebenen weitere Unterscheidungen anhand verschiedener Kriterien vorgenommen werden. Hinsichtlich der betrachteten Abstraktionsebene unterscheidet man zwischen parallelen Maschinenmodellen (engl. *machine models*), Architekturmodellen (engl. *architectural models*), Berechnungsmodellen (engl. *computational models*) und Programmiermodellen (engl. *programming models*) [77]. **Maschinenmodelle** stellen die niedrigste Abstraktionsstufe dar und bestehen aus einer hardwarenahen Beschreibung des Rechners und des Betriebssystems, die z.B. die einzelnen Register und Datenpfade eines Prozessors oder die Eingabe- und Ausgabepuffer und deren Verschaltung innerhalb eines Knotens eines Verbindungsnetzwerkes erfasst. Assemblersprachen nutzen diese Maschinenebene. **Architekturmodelle** stellen Abstraktionen von Maschinenmodellen dar und beschreiben etwa die Topologie des benutzten Verbindungsnetzwerkes, die Speicherorganisation (gemeinsamer oder verteilter Speicher), die Arbeitsweise der Prozessoren (synchrone oder asynchrone Arbeitsweise) oder den Abarbeitungsmodus der Instruktionen (SIMD oder MIMD). In diesem Sinne beschreibt Kapitel 2 die Architektur von parallelen Plattformen auf der Abstraktionsebene von Architekturmodellen.

Ein **Berechnungsmodell** resultiert üblicherweise aus einer Erweiterung eines Architekturmodells, die es ermöglicht, Algorithmen zu entwerfen, die auf dem Rechnersystem ausgeführt werden und möglichst mit Kosten bewertet werden können. Die Kosten beziehen sich meist auf die Ausführungszeit auf einer zugehörigen parallelen Plattform. Ein Berechnungsmodell hat also neben einem operationalen Anteil, der angibt, *welche parallelen Operationen* ausgeführt werden können, einen korrespondierenden analytischen Anteil, der die *zugehörigen Kosten* angibt. Idealerweise sollte ein Berechnungsmodell mit einem Architekturmodell korrespondieren. Für das von-Neumann-Architekturmodell ist etwa das RAM-Modell (*random access machine*) ein

zugehöriges Berechnungsmodell. Das RAM-Modell [6, 143] beschreibt einen sequentiellen Rechner durch einen Speicher und einen Prozessor, der auf diesen Speicher zugreifen kann. Der Speicher besteht aus einer beliebigen Anzahl von Speicherzellen, die jeweils einen beliebigen Wert enthalten können. Der Prozessor führt einen sequentiellen Algorithmus aus, der aus einer Folge von Instruktionen besteht, wobei in jedem Schritt eine einzelne Instruktion ausgeführt werden kann. Das Ausführen einer Instruktion besteht aus dem Laden von Daten aus dem Speicher in interne Register des Prozessors, dem Ausführen einer arithmetischen oder logischen Operation und dem Zurückschreiben eines evtl. errechneten Ergebnisses in den Speicher. Das RAM-Modell wird häufig für theoretische Laufzeitabschätzungen von Algorithmen zugrunde gelegt und ist zumindest zum Ableiten von asymptotischen Aussagen in den meisten Fällen gut geeignet. Dies gilt mit Einschränkungen auch für neuere Rechner und Prozessoren, obwohl diese intern eine wesentlich komplexere Verarbeitung der Instruktionen verwenden, vgl. die Abschnitte 2.2 und 2.7. Wie das RAM-Modell haben parallele Berechnungsmodelle ebenfalls idealisierte Ausprägungen und es existiert keine reale parallele Plattform, die sich genauso verhält, wie es vom Berechnungsmodell beschrieben wird. Ein Beispiel für ein Berechnungsmodell ist das PRAM-Modell, das eine Erweiterung des RAM-Modells darstellt, vgl. Abschnitt 4.5.

Programmiermodelle bilden eine weitere Abstraktionsstufe oberhalb der der Berechnungsmodelle und beschreiben ein paralleles Rechnersystem aus der Sicht einer Programmiersprache oder einer Programmierumgebung. Ein (paralleles) Programmiermodell definiert also eine Sicht des Programmierers auf eine (parallele) Maschine, d.h. es definiert, wie der Programmierer die Maschine ansprechen kann. Die Sicht des Programmierers auf eine (parallele) Maschine wird nicht nur durch das Verhalten der Hardware der Maschine bestimmt, sondern auch, wie bereits erwähnt, durch das verwendete Betriebssystem, den Compiler oder die Laufzeitbibliothek. Daher kann es je nach verwendetem Betriebssystem und Laufzeitbibliothek für eine Hardwarekonfiguration mehrere geeignete Programmiermodelle geben. Dem Programmierer wird ein Programmiermodell üblicherweise in Form einer Programmiersprache und/oder einer integrierten Laufzeitbibliothek zur Verfügung gestellt.

Es gibt eine Reihe von Kriterien, in denen sich parallele Programmiermodelle voneinander unterscheiden bzw. die durch ein paralleles Programmiermodell festgelegt werden. Wir möchten darauf hinweisen, dass die aufgeführten Modellbegriffe, insbesondere die Begriffe Berechnungsmodell und Programmiermodell, hinsichtlich der angesprochenen Abstraktionsebene in der Literatur durchaus unterschiedlich verwendet werden. Beispielsweise umfasst der Begriff Berechnungsmodell oft auch Programmierkonzepte. Unsere Darstellung orientiert sich an der Darstellung in [77].

Kriterien paralleler Programmiermodelle. Die wichtigsten Kriterien paralleler Programmiermodelle spezifizieren:

- welche Art der potentiell in den durchzuführenden Berechnungen enthaltenen Parallelität ausgenutzt werden kann (Instruktionsebene, Anweisungsebene, Prozedurebene oder parallele Schleifen),
- ob und wie der Programmierer diese Parallelität spezifizieren muss (implizit oder explizit parallele Programme),
- in welcher Form der Programmierer die Parallelität spezifizieren muss (z.B. als unabhängige Tasks, die dynamisch in einem Taskpool verwaltet werden oder als Prozesse, die beim Start des Programms erzeugt werden und die miteinander kommunizieren können),
- wie die Abarbeitung der parallelen Einheiten erfolgt (SIMD oder SPMD, synchron oder asynchron),
- in welcher Form der Informationsaustausch zwischen parallelen Teilen erfolgt (durch Kommunikation oder gemeinsame Variablen) und
- welche Möglichkeiten der Synchronisation es gibt.

Für jede realisierte parallele Programmiersprache oder -umgebung sind diese genannten Kriterien eines Programmiermodells auf die ein oder andere Art festgelegt. Die genannten Kriterien sind zum großen Teil unabhängig voneinander und erlauben eine Vielzahl von Kombinationsmöglichkeiten, wobei jede Kombinationsmöglichkeit ein eigenes Programmiermodell darstellt. Das Ziel jedes dieser Programmiermodelle besteht darin, dem Programmierer einen Mechanismus zur Verfügung zu stellen, mit dem er auf einfache Weise effiziente parallele Programme erstellen kann. Dazu muss jedes Programmiermodell gewisse Grundaufgaben unterstützen.

Ein paralleles Programm spezifiziert Berechnungen, die parallel zueinander ausgeführt werden können. Je nach Programmiermodell können dies einzelne Instruktionen sein, die arithmetische oder logische Berechnungen ausführen, oder Anweisungen, die mehrere Instruktionen umfassen können, oder Prozeduren, die beliebig viele Berechnungen beinhalten. Oft werden auch parallele Schleifen zur Verfügung gestellt, deren Iterationen unabhängig voneinander sind und daher parallel zueinander ausgeführt werden können. Abschnitt 3.3 gibt einen Überblick über mögliche Parallelitätsebenen. Die Gemeinsamkeit der Ansätze besteht darin, dass unabhängige **Module** oder **Tasks** spezifiziert werden, die auf den Prozessoren einer parallelen Plattform parallel zueinander ausgeführt werden können. Die Module sollten so auf die Prozessoren abgebildet werden, dass eine effiziente Abarbeitung resultiert. Diese Abbildung muss entweder explizit vom Programmierer vorgenommen werden oder wird von einer Laufzeitbibliothek übernommen. Der Abarbeitung liegt meist ein **Prozess- oder Thread-Konzept** zugrunde, d.h. das parallele Programm besteht aus parallel zueinander ablaufenden Kontrollflüssen, die entweder beim Start des parallelen Programms statisch festgelegt werden oder während der Laufzeit des Programms dynamisch erzeugt werden können. Prozesse können gleichberechtigt sein oder Hierarchien bilden, je nach Abarbeitungs- und Synchronisationsmodus des Programmiermodells. Oft wird ein Prozess einem Prozessor fest zugeordnet, d.h. ein Prozess kann

während seiner Ausführung nicht von einem Prozessor zu einem anderen wechseln. Auf die Zerlegung in Tasks und parallele Abarbeitungskonzepte für Prozessmodelle gehen wir in den Abschnitten 3.2–3.5 ein. Abschnitt 3.6 führt gängige Datenverteilungen für zusammengesetzte Daten wie Vektoren oder Matrizen ein.

Ein wesentliches Klassifizierungsmerkmal für parallele Programmiermodelle ist die *Organisation des Adressraums*, auf dem ein paralleles Programm arbeitet. Dabei unterscheidet man zwischen Modellen mit gemeinsamem und Modellen mit verteiltem Adressraum. Es gibt jedoch auch Mischformen, die Aspekte beider Modelle enthalten und als verteilter gemeinsamer Speicher (engl. *distributed shared memory*, DSM) bezeichnet werden. Die Organisation des Adressraums hat wesentlichen Einfluss auf den Informationsaustausch zwischen Prozessen, wie wir in Abschnitt 3.7 darstellen werden. Bei Modellen mit gemeinsamem Adressraum werden gemeinsame Variablen verwendet, auf die von verschiedenen Prozessen lesend und schreibend zugegriffen werden kann und die daher zum Informationsaustausch genutzt werden können. Bei Modellen mit verteiltem Adressraum nutzt jeder Prozess einen lokalen Speicher, ein gemeinsamer Speicher existiert aber nicht. Die Prozesse haben also keine Möglichkeit, Daten im Adressraum eines anderen Prozesses direkt zu adressieren. Um den Austausch von Informationen zu ermöglichen, gibt es daher zusätzliche Operationen zum Senden und Empfangen von Nachrichten, mit deren Hilfe die zugehörigen Prozesse Daten austauschen können.

3.2 Parallelisierung von Programmen

Der Parallelisierung eines gegebenen Algorithmus oder Programms liegt immer ein paralleles Programmiermodell zugrunde, das, wie wir gesehen haben, die unterschiedlichsten Charakteristika aufweisen kann. So verschieden parallele Programmiermodelle aber auch sein mögen, bei der Parallelisierung fallen grundsätzlich ähnliche Aufgaben an, die wir in diesem Abschnitt skizzieren wollen. In vielen Fällen liegt eine Beschreibung der von einem parallelen Programm durchzuführenden Berechnungen in Form eines sequentiellen Programms oder eines sequentiellen Algorithmus vor. Zur Realisierung des parallelen Programms ist eine **Parallelisierung** erforderlich, die die Daten- und Kontrollabhängigkeiten des sequentiellen Programms berücksichtigt und somit zum gleichen Resultat wie das sequentielle Programm führt. Das Ziel besteht meist darin, die Ausführungszeit des sequentiellen Programms durch die Parallelisierung so weit wie möglich zu reduzieren. Die Parallelisierung kann in mehrere Schritte zerlegt werden, die für einen systematischen Ansatz verwendet werden können:

1. **Zerlegung der durchzuführenden Berechnungen.** Die Berechnungen des Algorithmus werden in parallel ausführbare Einheiten (**Tasks**) zerlegt und die Abhängigkeiten zwischen diesen Tasks werden bestimmt.

Tasks sind die kleinsten Einheiten der Parallelität, die ausgenutzt werden sollen, und können je nach Zielrechner auf verschiedenen Ebenen der Ausführung identifiziert werden (Instruktionsebene, Datenparallelität, Funktionsparallelität), vgl. Abschnitt 3.3. Eine Task ist eine beliebige Folge von Berechnungen, die von einem einzelnen Prozessor ausgeführt wird. Die Abarbeitung einer Task kann Zugriffe auf den gemeinsamen Speicher (bei gemeinsamem Adressraum) oder die Ausführung von Kommunikationsoperationen (bei verteiltem Adressraum) beinhalten. Die Identifikation der Tasks hängt stark von dem zu parallelisierenden Programm ab. Das Ziel der Zerlegungsphase besteht zum einen darin, genügend Potential für eine parallele Abarbeitung zu schaffen, zum anderen sollte die *Granularität* der Tasks, d.h. die Anzahl der von einer Task durchgeführten Berechnungen, an das Kommunikationsverhalten der Zielmaschine angepasst werden.

2. **Zuweisung von Tasks an Prozesse oder Threads.** Ein **Prozess** oder **Thread** ist ein abstrakter Begriff für einen Kontrollfluss, der von einem physikalischen Prozessor ausgeführt wird und der nacheinander verschiedene Tasks ausführen kann. Die Anzahl der Prozesse muss nicht mit der Anzahl der physikalischen Prozessoren übereinstimmen, sondern kann nach den Gegebenheiten des Programms festgelegt werden. Das Ziel der Zuweisung von Tasks an Prozesse besteht darin, jedem Prozess etwa gleich viele Berechnungen zuzuteilen, so dass eine gute **Lastverteilung** oder sogar ein Lastengleichgewicht entsteht. Dabei müssen neben den Berechnungszeiten der Tasks auch Zugriffe auf den gemeinsamen Speicher (bei gemeinsamem Adressraum) bzw. Kommunikation zum Austausch von Daten (bei verteiltem Adressraum) berücksichtigt werden. Wenn zwei Tasks bei verteiltem Adressraum häufig Daten austauschen, ist es sinnvoll, diese Tasks dem gleichen Prozess zuzuordnen, da dadurch Kommunikationsoperationen durch lokale Speicherzugriffe ersetzt werden können. Die Zuordnung von Tasks an Prozesse wird auch als **Scheduling** bezeichnet. Dabei kann man zwischen *statischem Scheduling*, bei dem die Zuteilung beim Start des Programms festgelegt wird, und *dynamischem Scheduling*, bei dem die Zuteilung während der Abarbeitung des Programms erfolgt, unterscheiden. Die Kommunikationsbibliothek MPI beruht z.B. auf einem statischen Scheduling, während PVM, MPI-2 und Thread-Programme auch ein dynamisches Scheduling erlauben, vgl. Kapitel 5 und 6.
3. **Abbildung von Prozessen oder Threads auf physikalische Prozessoren** (auch **Mapping** genannt). Im einfachsten Fall existiert für jeden Prozessor ein Prozess, so dass die Abbildung einfach durchzuführen ist. Gibt es mehr Prozesse oder Threads als Prozessoren, müssen mehrere Prozesse auf einen Prozessor abgebildet werden. Dies kann je nach verwendetem Betriebssystem explizit vom Programm oder durch das Betriebssystem vorgenommen werden. Wenn es weniger Prozesse als Pro-

zessoren gibt, bleiben bestimmte Prozessoren unbeschäftigt. Ein Ziel des Mappings besteht darin, die Prozessoren gleichmäßig auszulasten und gleichzeitig die Kommunikation zwischen den Prozessoren gering zu halten.

Abbildung 3.1 zeigt eine Veranschaulichung der Parallelisierungsschritte.

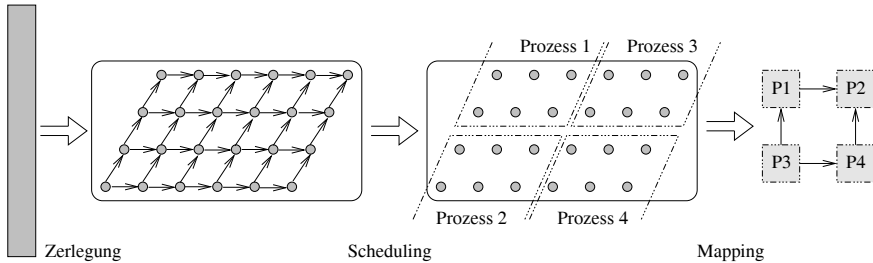


Abb. 3.1. Veranschaulichung der typischen Schritte zur Parallelisierung eines Anwendungsalgorithmus. Der Algorithmus wird in der Zerlegungsphase in Tasks mit gegenseitigen Abhängigkeiten aufgespalten. Diese Tasks werden durch das Scheduling Prozessen zugeordnet, die auf Prozessoren P1, P2, P3 und P4 abgebildet werden.

Unter dem Begriff **Scheduling-Algorithmus** oder **-Verfahren** fasst man allgemein Verfahren zur zeitlichen Planung der Durchführung von Tasks bestimmter Dauer zusammen [18]. Die Planung ist *Nachfolgerrestriktionen*, die durch Abhängigkeiten zwischen Tasks entstehen, und *Kapazitätsrestriktionen*, die durch die endliche Prozessoranzahl verursacht werden, unterworfen. Für die parallele Abarbeitung von Instruktionen, Anweisungen und Schleifen geht man dabei davon aus, dass jede Task von einem Prozessor sequentiell abgearbeitet wird. Für gemischte Programmiermodelle betrachtet man jedoch auch den Fall, dass einzelne Tasks von mehreren Prozessoren parallel abgearbeitet werden können, wodurch sich die Ausführungszeit dieser Tasks entsprechend verkürzt. In diesem Fall spricht man auch von **Multiprozessor-Task-Scheduling**. Das Ziel der Scheduling-Verfahren besteht darin, einen Abarbeitungsplan für die Tasks zu erstellen, der den Nachfolge- und Kapazitätsrestriktionen genügt und der bzgl. einer vorgegebenen Zielfunktion optimal ist. Die am häufigsten verwendete Zielfunktion ist dabei die maximale Fertigstellungszeit (auch Projektdauer, engl. *makespan*, genannt), die die Zeit zwischen dem Start des ersten und der Beendigung der letzten Task eines Programms angibt. Für viele realistische Situationen ist das Problem, einen optimalen Abarbeitungsplan zu finden, NP-vollständig bzw. NP-schwierig. Einen guten Überblick über Scheduling-Verfahren gibt [18].

Oft wird die Anzahl der Prozesse an die Anzahl der verfügbaren Prozessoren angepasst, so dass jeder Prozessor genau einen Prozess ausführt. In diesem Fall orientiert sich das parallele Programm eng an den Gegebenhei-

ten der zur Verfügung stehenden parallelen Plattform. Eine Unterscheidung zwischen Prozess und Prozessor fällt dadurch bei der Parallelisierung weg, so dass in vielen Beschreibungen paralleler Programme nicht zwischen den Begriffen Prozess und Prozessor unterschieden wird.

3.3 Ebenen der Parallelität

Die von einem Programm durchgeführten Berechnungen stellen auf verschiedenen Ebenen (Instruktionsebene, Anweisungsebene, Schleifenebene, Prozedurebene) Möglichkeiten zur parallelen Ausführung von Operationen oder Programmteilen zur Verfügung. Je nach Ebene entstehen dabei Tasks unterschiedlicher **Granularität**. Bestehen die Tasks nur aus einigen wenigen Instruktionen, spricht man von einer feinkörnigen Granularität. Besteht eine Task dagegen aus sehr vielen Instruktionen, spricht man von einer grobkörnigen Granularität. Die Parallelität auf Instruktions- und Anweisungsebene liefert feinkörnige Granularität, Parallelität auf Prozedurebene liefert grobkörnige Granularität und Parallelität auf Schleifenebene liefert meist Tasks mittlerer Granularität. Tasks unterschiedlicher Granularität erfordern üblicherweise auch den Einsatz unterschiedlicher Scheduling-Verfahren zur Ausnutzung des Parallelitätspotentials. Wir werden in diesem Abschnitt einen kurzen Überblick über das verfügbare Parallelitätspotential in Programmen und dessen Ausnutzung in Programmiermodellen geben.

3.3.1 Parallelität auf Instruktionsebene

Bei der Abarbeitung eines Programms können oft mehrere Instruktionen gleichzeitig ausgeführt werden. Dies ist dann der Fall, wenn die Instruktionen unabhängig voneinander sind, d.h. wenn zwischen zwei Instruktionen I_1 und I_2 keine der folgenden **Datenabhängigkeiten** existiert:

- **Fluss-Abhängigkeit** (engl. *true dependence*): I_1 berechnet ein Ergebnis in einem Register, das nachfolgend von I_2 als Operand verwendet wird;
- **Anti-Abhängigkeit** (engl. *anti dependence*): I_1 verwendet ein Register als Operand, das nachfolgend von einer Instruktion I_2 dazu verwendet wird, ein Ergebnis abzulegen;
- **Ausgabe-Abhängigkeit** (engl. *output dependence*): I_1 und I_2 verwenden das gleiche Register zur Ablage ihres Ergebnisses.

Abbildung 3.2 zeigt Beispiele für die verschiedenen Abhängigkeiten [167]. In allen drei Fällen kann ein Vertauschen der ursprünglichen Reihenfolge von I_1 und I_2 bzw. eine parallele Ausführung von I_1 und I_2 zu einem Fehler in der Berechnung führen. Dies gilt für die Fluss-Abhängigkeit, da I_2 evtl. einen alten Wert als Operand verwendet, für die Anti-Abhängigkeit, da I_1 fälschlicherweise einen zu neuen Wert als Operand verwenden kann, und für

die Ausgabe-Abhängigkeit, da nachfolgende Instruktionen evtl. einen falschen Wert aus dem Ergebnisregister verwenden können. Die Abhängigkeiten zwischen Instruktionen können durch Datenabhängigkeitsgraphen veranschaulicht werden. Abbildung 3.3 zeigt ein Beispiel einer Instruktionsfolge und den zugehörigen Graphen.

$I_1: \underline{R_1} \leftarrow R_2 + R_3$	$I_1: R_1 \leftarrow \underline{R_2} + R_3$	$I_1: \underline{R_1} \leftarrow R_2 + R_3$
$I_2: R_5 \leftarrow \underline{R_1} + R_4$	$I_2: \underline{R_2} \leftarrow R_4 + R_5$	$I_2: \underline{R_1} \leftarrow R_4 + R_5$
Fluß-Abhängigkeit	Anti-Abhängigkeit	Ausgabe-Abhängigkeit

Abb. 3.2. Typen von Datenabhängigkeiten zwischen Instruktionen. Für jeden Fall sind zwei Instruktionen angegeben, die den Registern auf der linken Seite einen Wert zuweisen (dargestellt durch einen Pfeil), der sich aus den Registerwerten der rechten Seite und der angegebenen Operation ergibt. Das Register, auf das sich die Abhängigkeit der Instruktionen bezieht, ist jeweils unterstrichen.

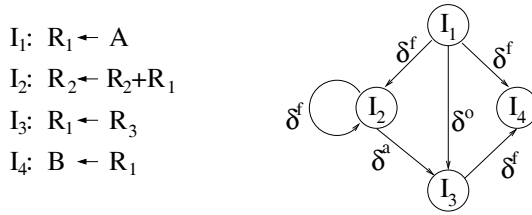


Abb. 3.3. Datenabhängigkeitsgraph zu einer Folge von Instruktionen I_1, I_2, I_3, I_4 . Die Kanten, die Fluss-Abhängigkeiten repräsentieren, sind mit δ^f gekennzeichnet. Anti-Abhängigkeitskanten und Ausgabe-Abhängigkeitskanten sind mit δ^a bzw. δ^o gekennzeichnet. Von I_1 gibt es eine Fluss-Abhängigkeit zu I_2 und I_4 , da beide das Register R_1 als Operanden verwenden. Da I_3 das gleiche Ergebnisregister wie I_1 verwendet, gibt es eine Ausgabe-Abhängigkeit von I_1 nach I_3 . Die restlichen Abhängigkeiten des Datenflussgraphen ergeben sich entsprechend.

Für superskalare Prozessoren kann Parallelität auf Instruktionsebene durch ein dynamisches Scheduling der Instruktionen ausgenutzt werden, vgl. Abschnitt 2.2. Dabei extrahiert ein in Hardware realisierter Instruktionsscheduler aus einem sequentiellen Programm parallel zueinander abarbeitbare Instruktionen, indem er überprüft, ob die oben definierten Abhängigkeiten existieren. Für VLIW-Prozessoren kann Parallelität auf Instruktionsebene durch einen geeigneten Compiler ausgenutzt werden [43], der durch ein statisches Scheduling in einer sequentiellen Instruktionsfolge unabhängige Berechnungen identifiziert und diese so anordnet, dass Funktionseinheiten des Prozessors explizit parallel angesprochen werden. In beiden Fällen liegt ein

sequentielles Programm zu Grunde, d.h. der Programmierer schreibt sein Programm entsprechend eines sequentiellen Programmiermodells.

3.3.2 Datenparallelität

In vielen Programmen werden dieselben Operationen auf unterschiedliche Elemente einer Datenstruktur angewendet. Im einfachsten Fall sind dies die Elemente eines Feldes. Wenn die angewendeten Operationen unabhängig voneinander sind, kann diese verfügbare Parallelität dadurch ausgenutzt werden, dass die zu manipulierenden Elemente der Datenstruktur gleichmäßig auf die Prozessoren verteilt werden, so dass jeder Prozessor die Operation auf den ihm zugeordneten Elementen ausführt. Diese Form der Parallelität wird **Datenparallelität** genannt und ist in vielen Programmen, insbesondere in solchen aus dem wissenschaftlich-technischen Bereich, vorhanden.

Zur Ausnutzung der Datenparallelität wurden sequentielle Programmiersprachen zu **datenparallelen Programmiersprachen** erweitert. Diese verwenden wie sequentielle Programmiersprachen *einen* Kontrollfluss, der aber auch datenparallele Operationen ausführen kann. Dabei wird von jedem Prozessor in jedem Schritt die gleiche Instruktion auf evtl. unterschiedlichen Daten ausgeführt. Dieses Abarbeitungsschema wird analog zum Architekturmodell in Abschnitt 2.3 als SIMD-Modell bezeichnet. Meistens werden datenparallele Operationen nur für Felder zur Verfügung gestellt.

Eine Programmiersprache mit auf Feldern arbeitenden datenparallelen Anweisungen, die auch als **Vektoranweisungen** (engl. *array assignment*) bezeichnet werden, ist FORTRAN 90/95 (F90/95), vgl. auch [44, 164, 109]. Andere Beispiele für datenparallele Programmiersprachen sind C* und Dataparallel C [72], PC++ [16], DINO [137] und High Performance FORTRAN (HPF) [49, 50]. Ein Beispiel für eine Vektoranweisung in FORTRAN 90 ist

$$a(1:n) = b(0:n-1) + c(1:n).$$

Die Berechnungen, die durch diese Anweisung durchgeführt werden, sind identisch zu den Berechnungen der folgenden Schleife:

```
for (i=1:n)
  a(i) = b(i-1) + c(i)
endfor
```

Ähnlich wie in anderen datenparallelen Sprachen ist die Semantik einer Vektoranweisung in FORTRAN 90 so definiert, dass alle auf der rechten Seite auftretenden Felder zugegriffen und die auf der rechten Seite spezifizierten Berechnungen durchgeführt werden, bevor die Zuweisung an das Feld auf der linken Seite der Vektoranweisung erfolgt. Daher ist die Vektoranweisung

$$a(1:n) = a(0:n-1) + a(2:n+1)$$

nicht äquivalent zu der Schleife

```
for (i=1:n)
    a(i) = a(i-1) + a(i+1)
endfor ,
```

da die Vektoranweisung zur Durchführung der Addition die alten Werte für $a(0:n-1)$ und $a(2:n+1)$ verwendet, während die Schleife nur für $a(i+1)$ die alten Werte verwendet. Für $a(i-1)$ wird jedoch jeweils der letzte errechnete Wert benutzt.

Datenparallelität kann auch in MIMD-Modellen ausgenutzt werden. Dies geschieht üblicherweise durch Verwendung eines **SPMD**-Konzeptes (*single program, multiple data*), d.h. es wird *ein* paralleles Programm verwendet, das von allen Prozessoren parallel ausgeführt wird. Dieses Programm wird von den Prozessoren *asynchron* ausgeführt, wobei die Kontrollstruktur des Programms meist so organisiert ist, dass die verschiedenen Prozessoren unterschiedliche Daten des Programms bearbeiten. Dies kann dadurch geschehen, dass jedem Prozessor in Abhängigkeit von seiner Prozessornummer (Prozessor-ID) ein Teil eines Feldes zugeteilt wird, dessen Unter- und Obergrenze in einer privaten Variablen des Prozessors abgelegt wird. Diese sogenannte *Datenverteilung* für Felder betrachten wir in Abschnitt 3.6 näher. Abbildung 3.4 zeigt die Skizze eines nach dieser Methode arbeitenden Programms zur Berechnung des Skalarproduktes zweier Vektoren. Die Bearbeitung unterschiedlicher Daten durch die verschiedenen Prozessoren führt in der Regel dazu, dass die Prozessoren unterschiedliche Kontrollpfade des Programms durchlaufen. Viele in der Praxis verwendete Programme arbeiten nach dem SPMD-Prinzip, da dieses auf der einen Seite das allgemeinere MIMD-Modell handhabbar macht, auf der anderen Seite aber für die meisten Probleme ausdrucksstark genug ist. Fast alle der in den folgenden Kapiteln verwendeten Algorithmen und Programme sind entsprechend dem SPMD-Prinzip strukturiert.

Datenparallelität kann für gemeinsamen oder verteilten Adressraum verwendet werden. Bei einem verteilten Adressraum müssen die Programmdaten so verteilt werden, dass jeder Prozessor auf die Daten, die er verarbeiten soll, in seinem lokalen Speicher direkt zugreifen kann. Der Prozessor wird dann auch als Eigentümer (engl. *owner*) der Daten bezeichnet. Oft bestimmt die Datenverteilung auch die Verteilung der durchzuführenden Berechnungen. Führt jeder Prozessor die Operationen des Programms auf den Daten durch, die er in seinem lokalen Speicher hält, spricht man auch von der **Owner-Computes**-Regel.

3.3.3 Parallelität in Schleifen

Viele Algorithmen führen iterative Berechnungen auf Datenstrukturen aus, die durch Schleifen im Programm ausgedrückt werden. Schleifen sind daher

```

local_size = size/p;
local_lower = me * local_size;
local_upper = (me+1) * local_size - 1;
local_sum = 0.0;

for (i=local_lower; i<=local_upper; i++)
    local_sum += x[i] * y[i];

Reduce(&local_sum, &global_sum, 0, SUM);

```

Abb. 3.4. Programmfragment zur Berechnung des Skalarproduktes zweier Vektoren x und y nach dem SPMD-Prinzip. Alle verwendeten Variablen seien private Variablen des jeweiligen Prozessors, d.h. jeder Prozessor kann einen anderen Wert unter dem Variablennamen ablegen. Dabei ist p die Anzahl der beteiligten Prozessoren und me ist die Nummer des jeweiligen Prozessors, wobei die Prozessoren von Null beginnend aufsteigend numeriert sind. Die beiden Felder x und y der Größe $size$ und die zugehörigen Berechnungen werden blockweise auf die Prozessoren aufgeteilt, wobei die Größe der Datenblöcke in `local_size` berechnet wird, die Untergrenze in `local_lower` und die Obergrenze in `local_upper`. Der Einfachheit halber nehmen wir dabei an, dass $size$ ein Vielfaches von p ist. Jeder Prozessor berechnet in `local_sum` das Teilskalarprodukt des ihm zugeordneten Datenblockes. Die Teilskalarprodukte werden durch Aufruf einer Reduktionsfunktion `Reduce()` bei Prozessor 0 aufsummiert. Verwenden wir ein Programmiermodell mit verteiltem Adressraum, kann dazu in MPI der Aufruf `MPI_Reduce(&local_sum, &global_sum, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD)` verwendet werden, vgl. Abschnitt 5.1.2.

als Programmkonstrukte in jeder (imperativen) Programmiersprache enthalten. Eine Schleife wird *sequentiell* abgearbeitet, d.h. die Ausführung der i -ten Iteration der Schleife wird nicht begonnen, bevor die Ausführung der $(i - 1)$ -ten Iteration vollständig abgeschlossen ist. Wir werden diese `for`-Schleifen in diesem Abschnitt auch als *sequentielle Schleifen* bezeichnen, um sie von den im Folgenden beschriebenen Schleifen mit `forall`, `dopar` und `doall` mit paralleler Abarbeitung zu unterscheiden. Wenn zwischen den Iterationen einer Schleife keine Abhängigkeiten bestehen, können diese in beliebiger Reihenfolge oder parallel zueinander von verschiedenen Prozessoren ausgeführt werden. Eine solche Schleife wird als **parallele Schleife** bezeichnet. Im folgenden beschreiben wir kurz verschiedene Typen von Schleifen für eine parallele Abarbeitung vgl. auch [164, 9].

forall-Schleifen. Eine `forall`-Schleife kann im Schleifenrumpf eine oder mehrere Zuweisungen an Feldelemente enthalten. Wenn eine `forall`-Schleife eine einzelne Zuweisung enthält, ist sie äquivalent zu einer Vektoranweisung,

vgl. Abschnitt 3.3.2, d.h. jede Iteration verwendet die *vor* Ausführung der **forall**-Schleife aktuellen Werte der zugegriffenen Variablen. Die Schleife

```
forall (i = 1:n)
  a(i) = a(i-1) + a(i+1)
endforall
```

ist damit zu der Vektoranweisung

$$a(1:n) = a(0:n-1) + a(2:n+1)$$

in FORTRAN 90/95 äquivalent. Wenn die **forall**-Schleife mehrere Zuweisungen enthält, werden diese *nacheinander* als Vektoranweisungen ausgeführt, indem jede Vektoranweisung vollständig abgearbeitet wird, bevor die Abarbeitung der nächsten beginnt. Das **forall**-Konstrukt existiert in Fortran 95, nicht jedoch in FORTRAN 90 [109].

dopar-Schleifen. Eine **dopar**-Schleife kann ebenfalls eine oder mehrere Zuweisungen enthalten, aber auch weitere Anweisungen oder Schleifen. Die Iterationen der **dopar**-Schleife werden parallel zueinander von verschiedenen Prozessen ausgeführt. Jeder Prozess führt alle Instruktionen der ihm zugeordneten Iterationen sequentiell aus und verwendet dabei die *vor* der Ausführung der **dopar**-Schleife aktuellen Werte der verwendeten Variablen. Damit sind die von einer Iteration ausgeführten Veränderungen von Variablenwerten für die anderen Iterationen *nicht* sichtbar. Nach der Abarbeitung aller Iterationen werden die von den einzelnen Iterationen durchgeführten Modifikationen von Variablen miteinander kombiniert, so dass sich *eine* globale Änderung ergibt. Wenn zwei verschiedene Iterationen die gleiche Variable manipulieren, kann dabei der Wert der einen oder der anderen Iteration übernommen werden, es kann also ein *nichtdeterministisches* Verhalten auftreten.

Die Ergebnisse von **forall**- und **dopar**-Schleifen mit demselben Schleifenrumpf können sich dann unterscheiden, wenn der Schleifenrumpf mehrere Anweisungen enthält.

Beispiel: Wir betrachten die folgenden drei Schleifen.

for (i=1:4)	forall (i=1:4)	dopar (i=1:4)
a(i)=a(i)+1	a(i)=a(i)+1	a(i)=a(i)+1
b(i)=a(i-1)+a(i+1)	b(i)=a(i-1)+a(i+1)	b(i)=a(i-1)+a(i+1)
endfor	endforall	enddopar

In der sequentiellen **for**-Schleife werden zur Berechnung von **b(i)** die von den vorangehenden Iterationen errechneten Werte von **a** verwendet. In der **forall**-Schleife werden zur Berechnung von **b(i)** die Werte von **a** verwendet, die sich durch Ausführung aller Iterationen der Schleife für die erste Anweisung des Schleifenrumpfes ergeben. In der **dopar**-Schleife sind nur die von der gleichen Iteration durchgeführten Manipulationen von **a** sichtbar. Da die Berechnung von **b(i)** den in der gleichen Iteration berechneten Wert **a(i)**

nicht verwendet, werden die vor Betreten der Schleife aktuellen Werte von **a** verwendet. Die folgende Tabelle zeigt ein Berechnungsbeispiel:

Startwerte		nach for-Schleife	nach forall-Schleife	nach dopar-Schleife
a(0)	1			
a(1)	2	b(1) 4	5	4
a(2)	3	b(2) 7	8	6
a(3)	4	b(3) 9	10	8
a(4)	5	b(4) 11	11	10
a(5)	6			

□

Eine **dopar**-Schleife, bei der ein von einer Iteration manipuliertes Feldelement nur in der gleichen Iteration verwendet wird, wird auch als **doall-Schleife** bezeichnet. Die Iterationen einer **doall**-Schleife sind unabhängig voneinander und können parallel zueinander oder in einer beliebigen sequentiellen Reihenfolge ausgeführt werden, ohne dass das Ergebnis der Ausführung sich ändert. Es handelt sich also bei einer **doall**-Schleife um eine *parallele Schleife*. Die einzelnen Iterationen einer **doall**-Schleife können also beliebig auf die ausführenden Prozessoren verteilt werden, ohne dass eine Synchronisation erforderlich wäre. Für allgemeine **dopar**-Schleifen muss für den Fall, dass ein Prozessor mehrere Iterationen ausführt, dafür Sorge getragen werden, dass die Semantik der **dopar**-Schleife eingehalten wird. Dazu muss vermieden werden, dass ein Prozessor die in einer früheren von ihm bearbeiteten Iteration definierten Werte verwendet. Eine Möglichkeit dazu besteht darin, alle Feldoperanden auf der rechten Seite von Zuweisungen, die Konflikte hervorrufen können, in temporären Variablen abzulegen und in der eigentlichen parallelen Schleife dann diese temporären Variablen zu verwenden. Da so alle Konflikte beseitigt sind, können sowohl für die Ablage in den temporären Variablen als auch für die eigentliche parallele Schleife **doall**-Schleifen verwendet werden.

Beispiel: Die folgende **dopar**-Schleife

```
dopar (i=2:n-1)
  a(i) = a(i-1) + a(i+1)
enddopar
```

ist äquivalent zu folgendem Programmsegment

```
doall (i=2:n-1)
  t1(i) = a(i-1)
  t2(i) = a(i+1)
enddoall
doall (i=2:n-1)
  a(i) = t1(i) + t2(i)
```

`enddoall,`

wobei `t1` und `t2` temporäre Felder sind. □

Weiterführende Informationen zu parallelen Schleifen und deren Abarbeitung sowie Transformationen zur Verbesserung einer parallelen Ausführung sind z.B. in [126, 164] enthalten.

3.3.4 Funktionsparallelität

In vielen sequentiellen Programmen sind verschiedene Programmteile unabhängig voneinander und können daher parallel zueinander ausgeführt werden. Bei den Programmteilen kann es sich um einzelne Anweisungen, Grundblöcke (engl. *basic blocks*), unterschiedliche Schleifen oder Funktionsaufrufe handeln. Man spricht daher auch von **Funktions-** oder **Taskparallelität**, wobei man jeden unabhängigen Programmteil als Task auffasst. Zur Ausnutzung der Taskparallelität können die Tasks in einem **Taskgraphen** dargestellt werden, dessen Knoten den Tasks entsprechen und dessen Kanten Abhängigkeiten zwischen Tasks angeben. Je nach Programmiermodell können die einzelnen Knoten des Taskgraphen sequentiell von *einem* Prozessor oder parallel von *mehreren* Prozessoren ausgeführt werden. Im letzteren Fall kann z.B. jede einzelne Task datenparallel abgearbeitet werden, was zu gemischter Task- und Daten-Parallelität führt.

Zur Bestimmung eines Abarbeitungsplanes für einen Taskgraphen auf einer Menge von Prozessoren werden den Knoten die Ausführungszeiten der zugehörigen Tasks zugeordnet. Das Ziel eines Scheduling-Verfahrens besteht darin, einen Abarbeitungsplan zu erstellen, der die Abhängigkeiten zwischen Tasks berücksichtigt und der zu einer minimalen Gesamtausführungszeit führt. Dabei können statische oder dynamische Scheduling-Verfahren verwendet werden. Ein *statisches Scheduling-Verfahren* legt die Zuordnung von Tasks an Prozessoren deterministisch vor der Ausführung des Programms fest. Die Zuordnung basiert auf einer Abschätzung der Ausführungszeiten der beteiligten Tasks, die entweder durch Laufzeitmessungen oder eine Analyse der internen Berechnungsstruktur der Tasks erhalten werden kann, vgl. Abschnitt 4.3. Einen ausführlichen Überblick über statische Scheduling-Verfahren für verschiedene Formen der Abhängigkeit zwischen den beteiligten Tasks findet man in [18]. Wenn die beteiligten Tasks selber parallel abgearbeitet werden können, spricht man auch von *Multiprozessor-Task-Scheduling*. Ein *dynamisches Scheduling-Verfahren* legt die Zuordnung erst während der Ausführung des Programms fest und kann so auf Variationen in den Ausführungszeiten der einzelnen Tasks reagieren. Eine Möglichkeit zur Durchführung eines dynamischen Scheduling-Verfahrens besteht in der Verwendung eines **Taskpools**, in dem ausführbare Tasks abgelegt werden und aus dem die Prozessoren eine neue Task entnehmen können, wenn sie ihre aktuelle Task abgearbeitet haben. Nach der Abarbeitung einer Task werden alle Nachfolger im Taskgraphen, deren Vorgänger vollständig abgear-

beitet sind, ausführbar und in den Taskpool neu aufgenommen. Taskpools werden insbesondere für Rechner mit gemeinsamem Adressraum eingesetzt, da dann der Taskpool einfach im gemeinsamen Speicher gespeichert werden kann. Wir gehen in Kapitel 6 auf die Implementierung eines Taskpools für gemeinsamen Adressraum ein. Informationen zum Aufbau und zum Scheduling von Taskgraphen findet man in [13, 59, 126, 129]. Die Verwaltung von Taskpools wird in [102, 147] näher besprochen. Verschiedene Varianten effizienter Implementierungen werden in [96] vorgestellt. Taskpool-Ansätze für irreguläre Algorithmen werden in [140] besprochen. Die Programmierung mit Multiprozessor-Tasks kann mit Hilfe bibliotheksbasierter Ansätze wie z.B. Tlib [133] unterstützt werden.

Neben der expliziten Organisation eines Programms als Ansammlung von Tasks und deren Verwaltung in einem Taskpool bieten einige Programmiersysteme auch die Möglichkeit, Taskparallelität durch Sprachkonstrukte auszudrücken und die Verwaltung des so angegebenen Grades an Taskparallelität durch den Compiler und das Laufzeitsystem vornehmen zu lassen. Dies hat den Vorteil, dass eine Anpassung an die Details eines speziellen Parallelrechners durch den Compiler vorgenommen werden kann und der Programmierer dadurch portable Programme erstellen kann. Die entsprechenden Sprachen sind oft Erweiterungen von FORTRAN oder High Performance FORTRAN (HPF). Andere Ansätze in dieser Richtung basieren auf der Verwendung von Koordinationssprachen, die die im Algorithmus verfügbare Parallelität in einem Koordinationsprogramm ausdrücken, das die Zusammenarbeit von sequentiellen oder parallelen Berechnungsmodulen spezifiziert und das mit Hilfe eines Übersetzungssystems in ein für eine spezielle Plattform effizientes Programm überführt werden kann. Ansätze in diese Richtung sind TwoL (*Two Level Parallelism*) [131], P3L (*Pisa Parallel Programming Language*) [122], SCL (*Structured Coordination Language*) [34] und PCN (*Program Composition Notation*) [51]. Eine ausführliche Behandlung findet man z.B. in [70].

Weite Teile der Thread-Programmierung basieren auf der Ausnutzung von Funktionsparallelität, da jeder beteiligte Thread unabhängige Funktionsaufrufe abarbeiten kann. Auf die Realisierung von Thread-Parallelität gehen wir in Kapitel 6 genauer ein; dort beschreiben wir mehrere Programmieransätze.

3.4 Explizite und implizite Darstellung der Parallelität

Parallele Programmiermodelle und -systeme können auch anhand der (expliziten oder impliziten) *Darstellung der Parallelität*, also der Zerlegung in Teilaufgaben und dem Vorkommen von Kommunikation und Synchronisation im zugehörigen parallelen Programm, unterschieden werden. Wenn im Programm keine explizite Darstellung erforderlich ist, erleichtert dies die Programmierung, erfordert aber einen sehr fortgeschrittenen Compiler zur Erzeugung eines effizienten Programms. Wenn die Parallelität dagegen explizit vom

Programmierer formuliert werden muss, ist die Programmierarbeit zum Erreichen eines effizienten Programms anspruchsvoll, für die Übersetzung kann aber ein Standardcompiler verwendet werden. Wir werden im Folgenden eine kurze Unterscheidung von Programmiermodellen anhand der expliziten und der impliziten Darstellung der Parallelität geben und verweisen auf [148] für eine ausführlichere Behandlung.

Implizite Parallelität. Die für den Programmierer am einfachsten zu verwendenden Modelle erfordern *keine* explizite Darstellung der Parallelität im Programm, d.h. das Programm ist im Wesentlichen eine Spezifikation der durchzuführenden Berechnungen ohne eine genaue Festlegung der Abarbeitungsreihenfolge oder der Details einer parallelen Ausführung. Der Programmierer kann sich damit auf die Formulierung des sequentiellen Algorithmus beschränken und muss sich nicht um die Organisation der parallelen Abarbeitung kümmern. Von den vielen Ansätzen wollen wir hier zwei Vertreter herausgreifen: parallelisierende Compiler und funktionale Programmiersprachen. Die Idee der **parallelisierenden Compiler** besteht darin, aus einem *imperativen* sequentiellen Programm automatisch ein effizientes paralleles Programm zu erzeugen. Dazu ist es notwendig, dass der Compiler die Abhängigkeiten zwischen den durchzuführenden Berechnungen ermittelt und die Berechnungen den ausführenden Prozessoren so zuordnet, dass eine gute Lastverteilung entsteht, ohne dass die Prozessoren zu oft Daten austauschen müssen, vgl. [126, 164, 9, 5]. Dies ist in vielen praktisch relevanten Fällen eine komplexe Aufgabe und es ist daher nicht verwunderlich, dass parallelisierende Compiler oft noch keine befriedigenden Ergebnisse zeigen, obwohl ein großer Aufwand für die Entwicklung der Compiler betrieben wurde.

Funktionale Programmiersprachen beschreiben die von einem Programm durchzuführenden Berechnungen als Funktionen ggf. höherer Ordnung, d.h. die Funktionen können andere Funktionen als Argumente verwenden und Funktionen als Ergebnis produzieren. Die populärste funktionale Programmiersprache ist Haskell [84, 160, 15]. Das Potential für eine parallele Auswertung eines funktionalen Programms besteht in der parallelen Auswertung der Argumente der Funktionen. Da funktionale Programme keine Seiteneffekte erlauben, können die Argumente von Funktionen ohne Veränderung des Ergebnisses parallel zueinander ausgewertet werden. Die Probleme für eine parallele Ausführung bestehen zum einen darin, dass eine parallele Auswertung der Argumente entweder auf einer oberen Rekursionsebene kein ausreichendes Potential an Parallelität bereitstellt oder umgekehrt auf den unteren Rekursionsebenen einen zu hohen Grad an Parallelität sehr feiner Granularität aufweist, so dass eine effiziente Zuordnung an Prozessoren schwierig ist. Weiter kann es vorkommen, dass in Abhängigkeit von Eingabewerten bestimmte Argumente gar nicht ausgewertet werden müssen.

Explizite Parallelität mit impliziter Zerlegung. Die zweite Klasse paralleler Programmiermodelle umfasst die Modelle, die zwar die vorhandene Parallelität explizit im Programm darstellen, die aber vom Programmierer

nicht verlangen, dass die Berechnungen in Tasks aufgeteilt und Prozessen zugewiesen werden. Damit muss auch keine Kommunikation und Synchronisation explizit dargestellt werden. Der Vorteil für den Compiler liegt gegenüber der ersten Klasse darin, dass der verfügbare Grad an Parallelität angegeben wird und nicht mehr durch eine komplizierte Datenabhängigkeitsanalyse ermittelt werden muss. Die wichtigsten Vertreter dieser Klasse sind parallele Programmiersprachen, die sequentielle Programmiersprachen um *parallele Schleifen* erweitern, deren Iterationen parallel zueinander abgearbeitet werden können, vgl. Abschnitt 3.3.3. Damit wird das Potential an Parallelität angegeben, ohne dass eine Zuordnung von Iterationen an Prozessoren stattfindet. Viele FORTRAN-Erweiterungen beruhen auf dieser Art von Erweiterung. Ein wichtiger Vertreter war High Performance FORTRAN (HPF) [49], das zusätzlich Konstrukte bereitstellte, die die Datenverteilung von Feldern angeben, um dem Compiler die schwierige Aufgabe der Bestimmung einer effizienten Datenverteilung abzunehmen.

Explizite Zerlegung. Die dritte Klasse umfasst parallele Programmiermodelle, die zusätzlich zu einer expliziten Darstellung der Parallelität eine explizite Darstellung der Zerlegung in Tasks im Programm erfordern. Die Zuordnung an Prozessoren und die Kommunikation zwischen den Prozessoren bleibt weiter implizit. Ein Vertreter dieser Klasse ist das BSP-Programmiersmodell, das auf dem in Abschnitt 4.5.2 beschriebenen BSP-Berechnungsmodell beruht, und das durch eine Bibliothek BSPLib realisiert wird [78, 79]. Ein BSP-Programm wird explizit in Threads zerlegt, die Zuordnung von Threads an Prozessoren wird aber von der Bibliothek vorgenommen.

Explizite Zuordnung an Prozessoren. Die vierte Klasse umfasst parallele Programmiermodelle, die zusätzlich zu einer expliziten Zerlegung in Tasks auch eine explizite Zuordnung der Tasks an Prozessoren erfordern. Die erforderliche Kommunikation zwischen den Prozessoren muss vom Programmierer aber nicht explizit angegeben werden. Ein Vertreter dieser Klasse ist die Koordinationssprache Linda [21, 20], die die übliche Punkt-zu-Punkt-Kommunikation zwischen Prozessoren dadurch ersetzt, dass ein gemeinsamer Pool von Daten (*tuple space* genannt) zur Verfügung gestellt wird, in den Prozesse beliebige Daten ablegen und entnehmen können. Der Zugriff auf den Pool erfolgt über drei Operationen:

- **in:** entferne einen Datenwert aus dem Pool,
- **read:** lese einen Datenwert aus dem Pool,
- **out:** lege einen Datenwert im Pool ab.

Die Identifikation der zu entnehmenden Daten erfolgt über die Angabe der Werte eines Teils der Datenfelder, die als Schlüssel interpretiert werden können. Für Rechner mit verteiltem Adressraum müssen die Zugriffsoperationen auf den Pool durch Kommunikationsoperationen zwischen den beteiligten Prozessen ersetzt werden, d.h. wenn im Linda-Programm ein Prozess *A* einen

Datenwert im Pool ablegt, der später von einem Prozess B gelesen oder entfernt wird, muss eine Kommunikation von Prozess A (**send**) zu Prozess B (**recv**) erzeugt werden. Dies führt nicht in allen Fällen zu einer effizienten Implementierung.

Explizite Kommunikation und Synchronisation. Die letzte Klasse umfasst Programmiermodelle, in denen der Programmierer alle Details der parallelen Abarbeitung, also auch die notwendigen Kommunikations- oder Synchronisationsoperationen explizit im Programm angeben muss. Dies hat den Vorteil, dass ein Standardcompiler verwendet werden kann, und dass der Programmierer explizit die parallele Abarbeitung steuern und so ein effizientes paralleles Programm erhalten kann. Gegenüber den anderen Klassen hat der Programmierer aber auch die meiste Arbeit. Vertreter dieser Klasse sind zum einen die Thread-Programmiermodelle, die wir in Kapitel 6 näher besprechen und zum anderen die Message-Passing-Programmiermodelle wie PVM und MPI, auf die wir in Kapitel 5 näher eingehen.

3.5 Strukturierung paralleler Programme

Parallele oder verteilte Programme bestehen aus einer Ansammlung von Tasks, die in Form von Prozessen oder Threads auf verschiedenen Prozessoren ausgeführt werden. Um ein korrektes Programm zu erhalten, müssen die Prozesse koordiniert bzw. strukturiert werden, wozu verschiedene Organisationsformen zur Auswahl stehen. Diese Koordination kann explizit durch den Anwendungsprogrammierer erfolgen oder durch das Laufzeitsystem. In der parallelen Programmierung haben sich eine Reihe von Mustern und Strukturierungsprinzipien für parallele Programme als günstig erwiesen, die wir im Folgenden kurz vorstellen. Weitere Ausführungen sind z.B. in [106] zu finden.

Erzeugung von Prozessen oder Threads. Die Erzeugung von Prozessen/Threads kann statisch oder dynamisch erfolgen. Im statischen Fall wird meist eine feste Anzahl von Prozessen/Threads zu Beginn der Abarbeitung des parallelen Programms erzeugt, die während der gesamten Abarbeitung existieren und erst am Ende des Gesamtprogramms beendet werden. In einem alternativen Ansatz können Prozesse/Threads zu jedem Zeitpunkt der Programmabarbeitung (statisch oder dynamisch) erzeugt und beendet werden. Zu Beginn der Abarbeitung ist meist nur ein einziger Prozess/Thread aktiv. Wir beschreiben im Folgenden verschiedene Organisationsformen der Zusammenarbeit von Prozessen/Threads und beschränken uns dabei auf Prozesse, weisen aber darauf hin, dass die Organisationsformen auch für Threads angewendet werden können.

Das **Fork-Join-Konstrukt** ist eines der einfachsten Konzepte zur Erzeugung von Prozessen [25]. Ein bereits existierender Prozess erzeugt mittels eines Fork-Aufrufs einen Kindprozess, der eine Kopie des Elternprozesses ist und eigene Kopien der Daten besitzt. Beide Prozesse arbeiten also

denselben Programmtext ab bis beide einen Join-Aufruf ausführen. Arbeitet der Kindprozess zuerst den Join-Aufruf ab, so wird er beendet. Arbeitet der Elternprozess zuerst den Join-Aufruf ab, so wartet er, bis auch der Kindprozess diesen Aufruf erreicht und fährt dann mit der Abarbeitung des nachfolgenden Programmtextes fort. Sollen Eltern- und Kindprozess eines Fork-Join-Konstruktes verschiedene Programmtexte abarbeiten, so muss dies durch eine Bedingung bzgl. der Prozessnummer angegeben werden. Das Fork-Join-Konzept kann explizit als Sprachkonstrukt zur Verfügung stehen oder kann im Laufzeitsystem verwendet werden. Meist wird das Konzept in der Programmierung mit gemeinsamen Adressraum verwendet. Die **Spawn**- und **Exit**-Operationen der Message-Passing-Programmierung, also der Programmierung mit verteiltem Adressraum, bewirken im wesentlichen dieselben Aktionen wie die Fork-Join-Operationen. Obwohl das Fork-Join-Konzept sehr einfach ist, erlaubt es durch verschachtelten Aufruf eine beliebige Struktur von paralleler Aktivität. Spezielle Programmiersprachen und -umgebungen haben oft eine spezifische Ausprägung der beschriebenen Erzeugung von Prozessen.

Eine strukturierte Variante der Prozesserzeugung wird durch das gleichzeitige Erzeugen und Beenden mehrerer Prozesse erreicht. Dazu wird das **Parbegin-Parend-Konstrukt** bereitgestellt, das manchmal auch mit dem Namen Cobegin-Coend bezeichnet wird. Zwischen Parbegin- und Parend-Anweisungen werden Prozesse angegeben, etwa durch Angabe von Funktionsaufrufen. Erreicht der ausführende Prozess den Parbegin-Befehl, so werden die durch das Parbegin- und Parend-Paar eingeklammerten Prozesse erzeugt und bearbeitet. Der Programmtext nach dem Parend-Befehl wird erst ausgeführt, wenn alle Prozesse innerhalb des Parbegin-Parend-Konstrukts beendet sind. Die Prozesse innerhalb des Parbegin-Parend-Konstrukts können gleichen oder verschiedenen Programmtext haben. Ob und wie die Prozesse tatsächlich parallel ausgeführt werden, hängt von der zur Verfügung stehenden Hardware und der Implementierung des Konstrukts ab. Die Anzahl und Art der zu erzeugenden Prozesse steht meist statisch fest. Auch für dieses Konstrukt haben spezielle parallele Sprachen oder Umgebungen ihre spezifische Syntax und Ausprägung, wie z.B. in Form von *parallelen Bereichen* (engl. *parallel Sections*).

SPMD und SIMD. Im SIMD- (*single instruction, multiple data*) und SPMD-Programmiermodell (*single program, multiple data*) wird zu Programmbeginn eine feste Anzahl von Prozessen gestartet. Alle Prozesse führen dasselbe Programm aus, das sie auf verschiedene Daten anwenden. Durch Kontrollanweisungen innerhalb des Programmtextes kann jeder Prozess verschiedene Programmteile auswählen und ausführen. Im SIMD-Ansatz werden die einzelnen Instruktionen *synchron* von den Prozessen abgearbeitet, d.h. die verschiedenen Prozesse arbeiten dieselbe Instruktion gleichzeitig ab. Der Ansatz wird auch häufig als *Datenparallelität* im engeren Sinne bezeichnet. Im SPMD-Ansatz können die Prozesse *asynchron* abgearbeitet werden, d.h. zu ei-

nem Zeitpunkt können verschiedene Prozesse verschiedene Programmstellen bearbeiten. Dieser Effekt tritt entweder durch unterschiedliche Ausführungsgeschwindigkeiten der Prozesse oder eine Verzögerung des Kontrollflusses in Abhängigkeit von lokalen Daten der Prozesse auf. Sollen Prozesse synchronisiert werden, so ist dies explizit zu programmieren. Der SPMD-Ansatz ist z.Z. einer der populärsten Ansätze der parallelen Programmierung, insbesondere in der Programmierung mit verteiltem Adressraum und für wissenschaftlich-technische Anwendungen; er ist prinzipiell jedoch nicht auf diesen Bereich beschränkt. Besonders geeignet ist die SPMD-Programmierung für Anwendungsalgorithmen, die auf Feldern arbeiten und bei denen eine Zerlegung der Felder die Grundlage einer Parallelisierung ist, vgl. Abschnitt 3.6. In der SPMD und SIMD-Programmierung sind alle Prozesse gleichberechtigt. Ein anderes Konzept bietet der folgende Ansatz.

Master-Slave oder Master-Worker. Bei diesem Ansatz kontrolliert ein einzelner Prozess die gesamte Arbeit eines Programms. Dieser Prozess wird Masterprozess genannt und entspricht meist dem Hauptprogramm des Anwendungsprogramms. Der Masterprozess erzeugt meist mehrere gleichartige Worker- oder Slaveprozesse, die die eigentlichen Berechnungen ausführen. Diese Workerprozesse können statisch oder dynamisch erzeugt werden. Die Zuteilung von Arbeit an die Workerprozesse kann durch den Masterprozess erfolgen. Die Workerprozesse können aber auch eigenständig Arbeit allokieren. In diesem Fall ist der Masterprozess nur für alle übrigen Koordinationsaufgaben zuständig, wie etwa Initialisierung, Zeitmessung oder Ausgabe.

Pipelining. Der Pipelining-Ansatz beschreibt eine spezielle Form der Zusammenarbeit verschiedener Prozesse. Jeder der Prozesse erwartet Eingaben und erzeugt Ausgabewerte. Die Zusammenarbeit von p Prozessen ist dadurch gegeben, dass Prozess P_k , $k \in \{1, \dots, p-1\}$, nacheinander Daten erzeugt, die vom nachfolgenden Prozess P_{k+1} als Eingabe benötigt werden. Alle Prozesse sind also gleichzeitig aktiv und ein Strom von Daten wird von den Prozessen jeweils an den nächsten weitergegeben. Diese Art des Zusammenarbeitens kann als spezielle Form einer funktionalen Zerlegung betrachtet werden. Die funktionalen Einheiten eines Anwendungsalgorithmus bilden die Prozesse, die durch ihre Datenabhängigkeit nicht nacheinander ausgeführt werden müssen, sondern auf die beschriebene Weise gleichzeitig abgearbeitet werden. Das Pipelining-Konzept kann prinzipiell mit gemeinsamem Adressraum oder mit verteiltem Adressraum realisiert werden.

Client-Server-Modell. Programmierstrukturierungen nach dem Client-Server-Prinzip ähneln dem MPMD-Modell (multiple program, multiple data), stammen ursprünglich aber eher dem verteilten Rechnen und der Unternehmenssoftware, wobei mehrere Client-Rechner mit einem als Server dienenden Mainframe verbunden sind, der etwa Anfragen an eine Datenbank bedient. Parallelität kann hier auf der Server-Seite auftreten, indem entweder mehrere Client-Anfragen parallel zueinander beantwortet werden, oder indem der

Server selbst auf einer parallelen Plattform implementiert ist und interne Parallelität realisiert. Mittlerweile wird das Client-Server Prinzip weiter gefasst und als System aus mehreren Komponenten aufgefasst, die jeweils die Rolle von Clients oder Server haben können, das jeweils eine Komponente (dann ein Client) von einer anderen Komponente (dann ein Server) einen Dienst (Service) erfragen kann. Nach Erledigung des Dienstes geht die Antwort an den Client zurück. Wichtig ist, dass die Komponenten für diese Art der Zusammenarbeit programmiert sind, etwa durch den Einsatz von Middleware, so dass beliebige Komponenten in verschiedenen Programmiermodellen und auf unterschiedlichen Plattformen miteinander kommunizieren können. Das Client-Server-Prinzip ist wichtig für die parallele Programmierung im heterogenen System. Insbesondere wurde das Client-Server-Prinzip in der Grid-Technologie verwendet und wird daher nun auch im Bereich des parallelen wissenschaftlichen Rechnens eingesetzt.

3.6 Datenverteilungen für Felder

Viele Algorithmen insbesondere auch aus dem Bereich des wissenschaftlichen Rechnens basieren auf Vektoren und Matrizen. Daher werden ein- und zweidimensionale Felder ebenso wie höherdimensionale Felder oft als Datenstruktur in den entsprechenden Programmen verwendet. Eine Parallelisierung solcher Algorithmen basiert häufig auf einer Aufteilung der Felder in Teilbereiche und einer Abbildung der Teilbereiche auf die zur Verfügung stehenden Prozessoren. Diese Vorgehensweise und auch die Abbildung selber werden als **Datenverteilung** (Partitionierung) bezeichnet. Im parallelen Algorithmus führen die Prozessoren Berechnungen durch, die mit den ihnen zugeordneten Feldelementen assoziiert sind. Wird ein *Rechner mit verteiltem Speicher* genutzt, so bedeutet dies, dass die Daten des Teilbereiches des Feldes im lokalen Speicher des entsprechenden Prozessors vorhanden sind, andere Prozessoren aber nur über das Netzwerk auf diesen Teilbereich zugreifen können. Eine Datenverteilung ist also die Grundlage für den Entwurf paralleler Algorithmen für Rechner mit verteiltem Speicher. Aber auch für *Rechner mit gemeinsamem Adressraum*, in denen das gesamte Feld allen Prozessoren zur Verfügung steht, ohne dass im Programm explizite Kommunikation nötig wäre, kann die Datenverteilung Grundlage für einen effizienten parallelen Algorithmus sein. Berechnungen werden entsprechend einer Datenverteilung den Prozessoren zugeordnet, wodurch Konflikte beim Datenzugriff auf den gemeinsamen Speicher vermieden werden. Wir betrachten in diesem Abschnitt *reguläre* Datenverteilungen für Felder beliebiger Dimension [36], die sich dadurch auszeichnen, dass die Abbildung der einzelnen Elemente des Feldes auf die jeweiligen Prozessoren des Parallelrechners durch eine in geschlossener Form darstellbare Funktion beschrieben werden kann. Im Folgenden seien $P = \{P_1, \dots, P_p\}$ die Prozessoren der Zielmaschine.

Datenverteilungen für eindimensionale Felder. Für *eindimensionale* Felder sind die gebräuchlichsten Feldverteilungen die blockweise Verteilung und die zyklische Verteilung. Das genaue Aussehen der Abbildung von Feldelementen auf Prozessoren hängt davon ab, ob die Nummerierung der Feldelemente bzw. der Prozessoren bei 0 oder bei 1 beginnt. Wir nehmen im Folgenden an, dass die Nummerierung bei 1 beginnt. Dies verkompliziert die Abbildungsfunktion zwar etwas, ist aber mit der in Kapitel 7 verwendeten Nummerierung konform, die in Anlehnung an die übliche mathematische Beschreibungsweise der entsprechenden Algorithmen gewählt wurde.

Eine **blockweise** Verteilung eines Feldes $v = (v_1, \dots, v_n)$ der Länge n ergibt sich dadurch, dass das Feld in p Blöcke von je $\lceil n/p \rceil$ benachbarten Feldelementen unterteilt wird, wobei Block j für $1 \leq j \leq p$ die Feldelemente $(j-1) \cdot \lceil n/p \rceil + 1, \dots, j \cdot \lceil n/p \rceil$ enthält. Block j wird Prozessor P_j zugeordnet. Wenn n kein Vielfaches von p ist, hat der letzte Block weniger Feldelemente. Für $n = 14$ und $p = 4$ ergibt sich z.B. die Zuordnung

$$\begin{aligned} P_1: & v_1, v_2, v_3, v_4, \\ P_2: & v_5, v_6, v_7, v_8, \\ P_3: & v_9, v_{10}, v_{11}, v_{12}, \\ P_4: & v_{13}, v_{14}. \end{aligned}$$

Eine Alternative besteht darin, den ersten $n \bmod p$ Prozessoren jeweils $\lceil n/p \rceil$ Feldelemente und den verbleibenden Prozessoren jeweils $\lfloor n/p \rfloor$ Feldelemente zuzuteilen. Eine **zyklische** Verteilung eines Feldes ergibt sich dadurch, dass die Feldelemente reihum an die Prozessoren verteilt werden, d.h. für $i = 1, \dots, n$ wird das Feldelement v_i dem Prozessor $P_{(i-1) \bmod p + 1}$ zugeordnet. Prozessor P_j erhält also die Feldelemente $j, j+p, \dots, j+p \cdot (\lceil n/p \rceil - 1)$, falls $j \leq n \bmod p$ bzw. $j, j+p, \dots, j+p \cdot (\lfloor n/p \rfloor - 2)$, falls $n \bmod p < j \leq p$. Für $n = 14$ und $p = 4$ ergibt sich z.B. die Zuordnung

$$\begin{aligned} P_1: & v_1, v_5, v_9, v_{13}, \\ P_2: & v_2, v_6, v_{10}, v_{14}, \\ P_3: & v_3, v_7, v_{11}, \\ P_4: & v_4, v_8, v_{12}. \end{aligned}$$

Die **blockzyklische** Datenverteilung stellt eine Kombination aus blockweiser und zyklischer Verteilung dar, in der benachbarte Feldelemente zu Blöcken zusammengefasst werden, wobei jeder Block eine vorgegebene Größe b hat. Dabei ist üblicherweise $b \ll n/p$. Wenn n kein Vielfaches von p ist, umfasst der letzte Block weniger als b Feldelemente. Die Blöcke werden in zyklischer Weise auf die Prozessoren verteilt. Abbildung 3.5(a) zeigt eine Veranschaulichung der Datenverteilungen für eindimensionale Felder. Gebräuchliche Verteilungen für mehrdimensionale Felder ergeben sich durch Kombination von blockweiser und zyklischer Verteilung in den verschiedenen Dimensionen. Wir untersuchen zuerst den wichtigen Fall der zweidimensionalen Felder.

Datenverteilungen für zweidimensionale Felder. Für *zweidimensionale Felder* (Matrizen) stellen die blockweisen und zyklischen Varianten der **streifenweisen Datenverteilung** eine einfache Verallgemeinerung der Datenverteilungen eindimensionaler Felder dar. Bei der **blockweisen streifenweisen** Datenverteilung werden die Spalten (oder Zeilen) der Matrix in p gleichgroße Blöcke aufeinanderfolgender Spalten (oder Zeilen) unterteilt, und jeder dieser Blöcke wird einem anderen Prozessor zugeordnet. Das Prinzip dieser Datenverteilung entspricht also der blockweisen Verteilung von eindimensionalen Feldern mit dem einzigen Unterschied, dass im zweidimensionalen Fall ganze Spalten (bzw. Zeilen) anstelle von einzelnen Feldelementen zu Blöcken zusammengefasst werden. Ebenso wird bei der **zyklischen streifenweisen** Datenverteilung und der **blockzyklischen streifenweisen** Datenverteilung vorgegangen. Die streifenweisen Datenverteilungen für zweidimensionale Felder werden in Abbildung 3.5(b) illustriert.

Die **schachbrettartigen** Datenverteilungen beziehen beide Dimensionen des zweidimensionalen Feldes ein und teilen dieses in quadratische oder rechteckige Teilstücke auf. Zur Realisierung werden die Prozessoren logisch in einem zweidimensionalen Gitter angeordnet. Wir nehmen im Folgenden an, dass p_1 bzw. p_2 die Anzahl der Zeilen bzw. Spalten des Prozessorgitters ist, also $p_1 \cdot p_2 = p$ gilt. Wir betrachten eine Matrix mit n_1 Zeilen und n_2 Spalten. Bei der **blockweisen schachbrettartigen** Datenverteilung bestimmt die Anzahl der Zeilen bzw. Spalten des Prozessorgitters die Anzahl der Blöcke in den Zeilen bzw. Spalten der aufzuteilenden Matrix. Die Blockgröße in den Zeilen bzw. Spalten der Matrix wird entsprechend festgelegt: Block (i, j) für $1 \leq i \leq p_1$ und $1 \leq j \leq p_2$ enthält alle Matrixelemente (k, l) mit $k = (i - 1) \cdot \lceil n_1/p_1 \rceil + 1, \dots, i \cdot \lceil n_1/p_1 \rceil$ und $l = (j - 1) \cdot \lceil n_2/p_2 \rceil + 1, \dots, j \cdot \lceil n_2/p_2 \rceil$. Block (i, j) wird dem Prozessor an Position (i, j) des Prozessorgitters zugeordnet.

Bei der **zyklischen schachbrettartigen** Datenverteilung werden die einzelnen Matrixelemente reihum den Prozessoren in den Zeilen bzw. Spalten des Prozessorgitters zugeordnet, so dass eine zyklische Zuordnung in beiden Dimensionen resultiert. Matrixelement (k, l) wird dem Prozessor an Position $((k - 1) \bmod p_1 + 1, (l - 1) \bmod p_2 + 1)$ des Prozessorgitters zugeordnet. Wenn n_1 bzw. n_2 Vielfache von p_1 bzw. p_2 sind, erhält der Prozessor an Position (i, j) des Prozessorgitters alle Matrixelemente (k, l) mit $k = i + s \cdot p_1$ und $l = j + t \cdot p_2$ für $0 \leq s < n_1/p_1$ und $0 \leq t < n_2/p_2$. Diese Verteilung kann auch dadurch beschrieben werden, dass Blöcke der Matrix mit jeweils p_1 Zeilen und p_2 Spalten gebildet werden, wobei das Element (i, j) jedes Blockes dem Prozessor an Position (i, j) des Prozessorgitters zugeordnet wird.

Bei der **blockzyklischen Schachbrettaufteilung** werden statt einzelner Matrixelemente rechteckige Blöcke der Größe $b_1 \times b_2$ zyklisch über die Prozessoren verteilt. Matrixelement (m, n) gehört zu Block (k, l) mit $k = \lceil m/b_1 \rceil$ und $l = \lceil n/b_2 \rceil$. Dieser Block wird dem Prozessor an Position $((k - 1) \bmod p_1 + 1, (l - 1) \bmod p_2 + 1)$ des Prozessorgitters zuge-

ordnet. Man beachte, dass die blockzyklischen Schachbrettaufteilungen die zyklischen Schachbrettaufteilungen als Spezialfall $b_1 = b_2 = 1$ enthalten. Blockweise Schachbrettaufteilungen sind durch den Spezialfall $b_1 = n_1/p_1$ und $b_2 = n_2/p_2$ beschrieben. Schachbrettaufteilungen für zweidimensionale Felder werden in Abbildung 3.5(c) illustriert.

Wir werden im Folgenden eine Methode zur Beschreibung von allgemeinen blockzyklischen Verteilungen für Felder beliebiger Dimension vorstellen, die durch Kombination blockweiser und zyklischer Feldverteilungen entstehen.

Datenverteilungen für beliebig-dimensionale Felder. A sei ein d -dimensionales Feld mit Indexmenge $I_A \subset \mathbb{N}^d$, d.h. für $\mathbf{i} = (i_1, \dots, i_d) \in I_A$ ist $A[i_1, \dots, i_d]$ ein Element des Feldes. Wir nehmen $1 \leq i_j \leq n_j$ an, d.h. n_j ist die Anzahl der Elemente in der j -ten Dimension. Für die Festlegung der Verteilung werden die Prozessoren in einem d -dimensionalen Gitter angeordnet, wobei wir annehmen, dass p_i die Anzahl der Prozessoren in der i -ten Dimension angibt. Die Ausdehnungen des Gitters können prinzipiell beliebig festgelegt werden, d.h. die Prozessoren können prinzipiell beliebig auf die Dimensionen des Gitters aufgeteilt werden, es muss aber $p = \prod_{i=1}^d p_i$ gelten. Verschiedene Prozessorgitter resultieren in verschiedenen Feldverteilungen.

Eine Datenverteilung für A wird durch eine **Verteilungsfunktion** $\gamma_A : I_A \subset \mathbb{N}^d \rightarrow 2^P$ beschrieben, wobei 2^P die Potenzmenge der Menge P der Prozessoren bezeichnet. Dabei bedeutet $\gamma_A(\mathbf{i}) \equiv G(\mathbf{i}) \subseteq P$, dass Feldelement $A[i_1, \dots, i_d]$ mit $\mathbf{i} = (i_1, \dots, i_d)$ im Speicher jedes Prozessors aus der Prozessorgruppe $G(\mathbf{i})$ abgelegt wird. Ein Feldelement kann also mehreren Prozessoren zugeordnet sein. Eine Datenverteilung heißt **repliziert**, falls für jedes $\mathbf{i} \in I_A$ gilt, dass $\gamma_A(\mathbf{i}) = P$. Eine Datenverteilung heißt **Einzelverteilung**, falls für alle $\mathbf{i} \in I_A$ gilt, dass $|\gamma_A(\mathbf{i})| = 1$. Die für eine Datenverteilung γ_A von einem Prozessor q abgespeicherten Feldelemente werden durch die Funktion $L(\gamma_A) : P \rightarrow 2^{I_A}$ mit

$$\mathbf{i} \in L(\gamma_A)(q) \quad \text{genau dann, wenn} \quad q \in \gamma_A(\mathbf{i})$$

beschrieben. Im Folgenden betrachten wir Einzelverteilungen.

Die meisten gebräuchlichen Datenverteilungen können durch Verallgemeinerung der oben für den zweidimensionalen Fall beschriebenen blockzyklischen Verteilungen beschrieben werden. Wir nehmen an, dass dabei Blöcke mit b_i Elementen in der i -ten Dimension verwendet werden. Das Feldelement $A[i_1, \dots, i_d]$ wird dem Block $(k_1, \dots, k_d)$ mit $k_j = \lceil i_j/b_j \rceil$ für $1 \leq j \leq d$ zugeordnet. Dieser Block wird dem Prozessor an Position $((k_1 - 1) \bmod p_1 + 1, \dots, (k_d - 1) \bmod p_d + 1)$ zugeordnet. Wir bezeichnen blockzyklische Verteilungen im Folgenden auch als **parametrisierte Datenverteilungen**, da die Datenverteilung durch einen Parametervektor

$$((p_1, b_1), \dots, (p_d, b_d)), \tag{3.1}$$

den wir auch als **Verteilungsvektor** bezeichnen, eindeutig festgelegt wird. Dabei ist p_i die Anzahl der Prozessoren in Dimension i ($1 \leq p_i \leq p$) und b_i

a) blockweise

1	2	3	4	5	6	7	8
P ₁		P ₂		P ₃		P ₄	

zyklisch

1	2	3	4	5	6	7	8
P ₁	P ₂	P ₃	P ₄	P ₁	P ₂	P ₃	P ₄

block-zyklisch

1	2	3	4	5	6	7	8	9	10	11	12
P ₁		P ₂		P ₃		P ₄		P ₁		P ₂	

b) blockweise

	1	2	3	4	5	6	7	8
1								
2	P ₁		P ₂		P ₃		P ₄	
3								
4								

zyklisch

	1	2	3	4	5	6	7	8
1	P ₁	P ₂	P ₃	P ₄	P ₁	P ₂	P ₃	P ₄
2								
3								
4								

block-zyklisch

	1	2	3	4	5	6	7	8	9	10	11	12
1												
2	P ₁		P ₂		P ₃		P ₄		P ₁		P ₂	
3												
4												

c) blockweise

	1	2	3	4	5	6	7	8
1								
2	P ₁				P ₂			
3								
4	P ₃				P ₄			

zyklisch

	1	2	3	4	5	6	7	8
1	P ₁	P ₂	P ₁	P ₂	P ₁	P ₂	P ₁	P ₂
2	P ₃	P ₄	P ₃	P ₄	P ₃	P ₄	P ₃	P ₄
3	P ₁	P ₂	P ₁	P ₂	P ₁	P ₂	P ₁	P ₂
4	P ₃	P ₄	P ₃	P ₄	P ₃	P ₄	P ₃	P ₄

block-zyklisch

	1	2	3	4	5	6	7	8	9	10	11	12
1	P ₁		P ₂		P ₁		P ₂		P ₁		P ₂	
2												
3	P ₃		P ₄		P ₃		P ₄		P ₃		P ₄	
4												

Abb. 3.5. Illustration der Datenverteilungen für Felder: a) für eindimensionale Felder, b) für zweidimensionale Felder mit streifenweiser Datenverteilung und c) für zweidimensionale Felder mit schachbrettartiger Datenverteilung.

ist die Blockgröße in Dimension i ($1 \leq b_i \leq n_i$). Neben beliebigen blockzyklischen Datenverteilungen können durch einen Parametervektor wie im zweidimensionalen Fall auch blockweise und zyklische Datenverteilungen als Spezialfall beschrieben werden. Wir werden parametrisierte Datenverteilungen in Abschnitt 7.1 verwenden, um eine allgemeine parallele Implementierung der Gauß-Elimination zur Lösung von linearen Gleichungssystemen für Rechner mit verteiltem Speicher zu beschreiben, die die für die verschiedenen Datenverteilungen notwendigen Kommunikationsoperationen enthält. Kennt man die Ausführungszeiten der Kommunikationsoperationen für einen gegebenen Parallelrechner, können die Parameter des Verteilungsvektors so bestimmt werden, dass eine minimale Ausführungszeit resultiert.

3.7 Informationsaustausch

Zur Koordination der Zusammenarbeit von Programmteilen eines parallelen Programms muss Information zwischen den Programmteilen ausgetauscht werden. Die Realisierung dieses Informationsaustauschs hängt stark von der Architektur des benutzten parallelen Rechnersystems ab, insbesondere von dessen Speicherorganisation. Im folgenden geben wir einen ersten Überblick über den Informationsaustausch für Rechner mit gemeinsamem Adressraum (Abschnitt 3.7.1) und Rechner mit verteiltem Adressraum (Abschnitt 3.7.2). Als Beispiel geben wir in Abschnitt 3.7.3 eine Parallelisierung einer Matrix-Vektor-Multiplikation für beide Speicherorganisationen an.

3.7.1 Gemeinsame Variablen

Bei Programmiermodellen mit gemeinsamem Adressraum wird die Existenz eines gemeinsamen Speichers angenommen. Jeder Prozess kann neben privaten Daten auch gemeinsame Daten aus dem gemeinsamen Speicher lesen bzw. deren Wert im gemeinsamen Speicher verändern. Gemeinsame Daten werden über **gemeinsame Variablen** angesprochen, auf die wie in imperativen Programmiersprachen zugegriffen werden kann. Die Unterscheidung zwischen gemeinsamen und privaten Variablen von Prozessen erfolgt entweder bei deren Deklaration durch Schlüsselwörter, z.B. **private** oder **shared**, oder durch spezielle Vereinbarungen, die etwa globale Variablen immer als gemeinsame Variablen und lokale Variablen von Prozeduren immer als private Variablen anlegen. Zum koordinierten Zugriff durch mehrere Prozesse auf gemeinsame Variablen stehen Synchronisationsoperationen zur Verfügung, die sicherstellen, dass zum gleichen Zeitpunkt nicht mehrere Prozesse versuchen, die gleiche Variable zu verändern. Bei konkurrierenden Zugriffen auf dieselbe gemeinsame Variable findet eine **Sequentialisierung** statt, d.h. die beteiligten Prozesse führen ihre Operation *nacheinander* aus und nicht gleichzeitig. Kapitel 6 beschäftigt sich mit der Programmierung in einem gemeinsamen

Adressraum und stellt verschiedene Laufzeitbibliotheken wie Pthreads, Java Threads oder OpenMP vor. Wir werden hier bereits einige Grundkonzepte zusammenfassen.

Für Programmiermodelle mit gemeinsamem Adressraum findet der Austausch von Information zwischen den beteiligten Prozessoren über gemeinsame Variablen statt. Will ein Prozessor P_i einem anderen Prozessor P_j Daten übermitteln, so belegt er eine geeignete gemeinsame Variable mit diesen Daten, so dass Prozessor P_j durch Lesen dieser Variablen den übermittelten Wert erhält. Um sicherzustellen, dass Prozessor P_j den Wert der Variablen erst dann liest, wenn Prozessor P_i die Variable mit dem gewünschten Wert belegt hat, muss eine *Synchronisationsoperation* verwendet werden, so dass Prozessor P_i die Variable vor Ausführung der Synchronisationsoperation belegt und Prozessor P_j den Wert nach Ausführung der Synchronisationsoperation liest.

Bei der Verwendung gemeinsamer Variablen muss auch gewährleistet sein, dass nicht verschiedene Prozessoren zum gleichen Zeitpunkt die gleiche gemeinsame Variable zu manipulieren versuchen, da dies zum Auftreten von **zeitkritischen Abläufen** (engl. *race conditions*) führen kann. Als zeitkritischen Ablauf bezeichnet man den Effekt, dass das Ergebnis der Ausführung eines Programmsegments durch mehrere Prozessoren von der relativen Ausführungsgeschwindigkeit der Prozessoren zueinander abhängt, d.h. wenn das Programmsegment zuerst von Prozessor P_i und dann von Prozessor P_j ausgeführt wird, kann ein anderes Ergebnis berechnet werden als wenn das Programmsegment zuerst von Prozessor P_j und dann von Prozessor P_i ausgeführt wird. Das Auftreten von zeitkritischen Abläufen ist meist unerwünscht, da die relative Ausführungsreihenfolge von vielen Faktoren abhängen kann (z.B. der Ausführungsgeschwindigkeit der Prozessoren, dem Auftreten von Interrupts, oder dem Wert von Eingabedaten), die vom Programmierer nur bedingt zu beeinflussen sind. In diesem Zusammenhang spricht man auch von einem **nichtdeterministischen Verhalten**, da für die Ausführungsreihenfolge und das Ergebnis verschiedene Möglichkeiten eintreten können, ohne dass dies vorhergesagt werden kann.

Ein Programmsegment eines Prozesses, in dem Zugriffe auf gemeinsame Variablen vorkommen, die konkurrierend zu den Zugriffen anderer Prozesse geschehen können, so dass inkonsistente Werte der gemeinsamen Variablen auftreten können, heißt **kritischer Bereich** (engl. *critical section*). Eine fehlerfreie Abarbeitung kann dadurch gewährleistet werden, dass sich jeweils nur ein Prozess in einem kritischen Bereich zu einer gemeinsamen Variablen befindet. Diese Vorgehensweise wird **wechselseitiger Ausschluss** (engl. *mutual exclusion*) genannt. Programmiermodelle für einen gemeinsamen Adressraum stellen Operationen und Mechanismen zur Sicherstellung des wechselseitigen Ausschlusses zur Verfügung, die garantieren, dass zu jedem Zeitpunkt nur ein Prozess die Programmstelle ausführt, die auf eine kritische gemeinsame Variable zugreift. Die zugrundeliegenden Mecha-

nismen sind ursprünglich für den Bereich der Multitasking-Betriebssysteme entwickelt worden und wurden für eine Anwendung in der parallelen Programmierung entsprechend angepasst.

Zur Vermeidung des Auftretens von zeitkritischen Abläufen bei Verwendung von gemeinsamen Variablen kann zur Koordination der Prozessoren ein **Sperrmechanismus** zur Verfügung gestellt werden. Dazu wird eine **Sperrvariable** 1 eines speziell vorgegebenen Typs verwendet, die mit Hilfe zweier Funktionen *lock*(1) und *unlock*(1) angesprochen wird. Dabei dient *lock*(1) zur Belegung der Sperrvariablen und *unlock*(1) zu deren Freigabe. Die Vermeidung eines zeitkritischen Ablaufs bei der Abarbeitung eines Programmsegments beruht darauf, dass dem Programmsegment eine Sperrvariable zugeordnet wird und dass jeder Prozessor vor Betreten des Programmsegments *lock*(1) und nach Verlassen des Programmsegments *unlock*(1) aufruft. Nur wenn *jeder* Prozessor diese Vereinbarung einhält, werden zeitkritische Abläufe vermieden.

Der Aufruf *lock*(1) hat den Effekt, dass der aufrufende Prozessor P_i nur dann das dieser Sperrvariablen zugeordnete Programmsegment ausführen kann, wenn gerade kein anderer Prozessor P_j dieses Programmsegment ausführt. Wenn ein anderer Prozessor P_j vorher *lock*(1) aufgerufen hat und die Sperrvariable noch nicht mit *unlock*(1) wieder freigegeben hat, wird Prozessor P_i so lange blockiert, bis Prozessor P_j *unlock*(1) aufruft. Der Aufruf *unlock*(1) bewirkt neben der Freigabe der Sperrvariablen auch das Aufwecken eines anderen bzgl. der Sperrvariablen 1 blockierten Prozessors. Die Verwendung eines Sperrmechanismus führt also zur *Sequentialisierung* der Abarbeitung eines Programmsegments, d.h. es ist sichergestellt, dass jeweils nur ein Prozessor das Programmsegment ausführt. Die Realisierung von Sperrmechanismen in Laufzeitbibliotheken wie Pthreads, Java Threads oder OpenMP werden wir in Kapitel 6 vorstellen.

Ein weiterer Mechanismus zur Realisierung eines wechselseitigen Ausschlusses ist der **Semaphor** [37]. Ein Semaphor ist eine Struktur, die eine Integervariablen s beinhaltet, auf die zwei atomaren Operationen $P(s)$ und $V(s)$ angewendet werden können. Ein binärer Semaphor kann nur die Werte 0 und 1 annehmen. Werden weitere Werte angenommen, spricht man von einem zählenden Semaphor. Die Operation $P(s)$ (oder *wait*(s)) wartet bis der Wert von s größer als 0 ist, dekrementiert den Wert von s anschließend um 1 und erlaubt dann die weitere Ausführung der nachfolgenden Berechnungen des Prozessors. Die Operation $V(s)$ (oder *signal*(s)) inkrementiert den Wert von s um 1. Der genaue Mechanismus der Verwendung von P und V

zum Schutz eines kritischen Bereiches ist nicht streng festgelegt. Eine übliche Form ist:

```
wait(s)
kritischer Bereich
signal(s)
```

Verschiedene Prozesse führen die Operationen P und V auf s aus und koordinieren so den Zugriff der Prozesse auf kritische Bereiche. Führt z.B. Prozess P_i die Operation $wait(s)$ aus um danach seinen kritischen Bereich zu bearbeiten, so wird jeder andere Prozess P_j beim Aufruf von $wait(s)$ am Eintritt in seinen kritischen Bereich so lange gehindert, bis P_i die Operation $signal(s)$ ausführt. Dadurch, dass dem Anwendungsprogrammierer die spezielle Ausgestaltung der Verwendung von Semaphoren überlassen bleibt, ist dieses Konzept relativ fehleranfällig.

Ein abstrakteres Konzept stellt ein **Monitor** dar [80]. Ein Monitor ist ein Sprachkonstrukt, das Daten und Operationen, die auf diese Daten zugreifen, in einer Struktur zusammenfasst. Auf die Daten eines Monitors kann *nur* durch diese Monitoroperationen zugegriffen werden. Da zu jedem Zeitpunkt die Ausführung nur *einer* Monitoroperation erlaubt ist, wird der wechselseitige Ausschluss bzgl. der Daten des Monitors automatisch sichergestellt.

3.7.2 Kommunikationsoperationen

Für Programmiermodelle mit verteiltem Adressraum wird der Austausch von Informationen zwischen den beteiligten Prozessoren durch explizite *Kommunikationsanweisungen* realisiert, die von den Prozessoren während der Abarbeitung des Programms ausgeführt werden. Die Ausführung einer Kommunikationsanweisung bewirkt, dass ein Prozessor Daten erhält, die im Speicher eines anderen Prozessors abgelegt sind. Der Informationsaustausch wird durch das **Versenden von Nachrichten** (engl. *message passing*) realisiert, so dass bei einem Programmiermodell mit verteiltem Adressraum auch von Programmierung mit Nachrichtenübertragung oder -austausch bzw. von **Message-Passing-Programmierung** gesprochen wird.

Für das Versenden einer Nachricht von einem Prozessor zu einem anderen müssen Sende- und Empfangsoperationen als Paar auftreten. Eine Sendeoperation schickt Daten aus dem lokalen Adressraum des ausführenden Prozessors an einen anderen in der Operation angegebenen Prozessor. Eine Empfangsoperation empfängt Daten von einem anderen Prozessor im Adressraum des empfangenden Prozessors. Diese Art des Informationsaustauschs wird auch als *Punkt-zu-Punkt-Kommunikation* bezeichnet. Zusätzlich zur Punkt-zu-Punkt-Kommunikation werden **globale Kommunikationsoperationen** bereitgestellt, die eine Menge von Prozessoren in einen Informationsaustausch einbeziehen. Viele parallele Programme benutzen eine relativ kleine Menge von regelmäßigen *Kommunikationsmustern*, die durch Aufruf entsprechender *Kommunikationsoperationen* ausgeführt werden [14, 89].

Wir werden diese Menge von Kommunikationsoperationen im Folgenden vorstellen und diese in den weiteren Kapiteln dieses Buches zur Beschreibung von parallelen Implementierungen für Rechner mit verteiltem Adressraum nutzen. Dabei legen wir ein aus p identischen Prozessoren $P_1, \dots, P_p$ bestehendes Netzwerk zugrunde und nehmen an, dass jeder Prozessor P_i durch eine eindeutige Prozessornummer i identifiziert wird.

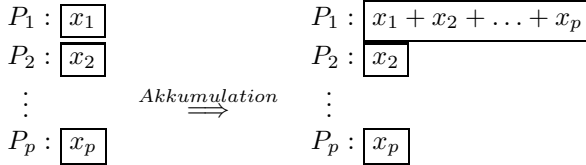
- **Einzeltransfer:** Bei einer Einzeltransferoperation schickt ein Prozessor P_i (Sender) einem anderen Prozessor P_j (Empfänger), $j \neq i$, eine Nachricht. Nur diese beiden Prozessoren sind an der Kommunikationsoperation beteiligt. Zur Durchführung der Operation führt der Sender eine Sendeoperation aus, für die er einen Sendepuffer, in dem die zu verschickende Nachricht abgelegt ist, und die Prozessornummer des Empfängers der Nachricht angibt. Der Empfänger führt eine korrespondierende Empfangsoperation aus, für die er einen Empfangspuffer, in dem die ankommende Nachricht abgelegt werden soll, und die Prozessornummer des Senders angibt. Zu jeder Sendeoperation muss es eine korrespondierende Empfangsoperation und umgekehrt geben, da sonst Deadlocks auftreten können, vgl. z.B. Abschnitt 5.1.1. Einzeltransferoperationen bilden die Grundlage jeder Kommunikationsbibliothek. Prinzipiell können alle Kommunikationsmuster aus Einzeltransferoperationen zusammengesetzt werden, für regelmäßige Kommunikationsmuster ist aber oft der Einsatz von *globalen Kommunikationsoperationen*, an denen alle Prozessoren beteiligt sind, einfacher und effizienter.
- **Einzel-Broadcast:** Bei einer Einzel-Broadcastoperation schickt ein ausgezeichneter Prozessor P_i , der auch als *Wurzel* bezeichnet wird, die *gleiche* Nachricht an alle anderen Prozessoren. Eine Broadcastoperation mit Prozessor P_1 als Wurzel und Nachricht x kann folgendermaßen veranschaulicht werden:



Vor der Ausführung der Operation sind die Daten x im lokalen Adressraum von P_1 , nach der Ausführung der Operation haben alle Prozessoren x in ihren lokalen Adressräumen. Zur Durchführung der Operation führt *jeder* Prozessor eine Broadcastanweisung aus, in der er die Wurzel der Broadcastoperation angibt. Der Wurzelprozessor spezifiziert einen Puffer, aus dem die Broadcastnachricht verschickt werden soll, alle anderen Prozessoren spezifizieren einen Empfangspuffer, in dem die empfangene Broadcastnachricht abgelegt werden soll.

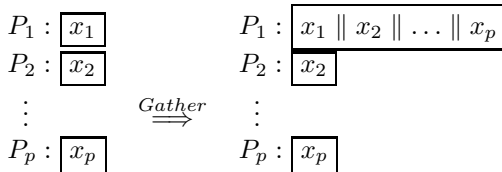
- **Einzel-Akkumulation:** Bei einer Einzel-Akkumulationsoperation schickt jeder Prozessor an einen ausgezeichneten Prozessor P_i , der als Wurzel bezeichnet wird, eine Nachricht mit Daten gleichen Typs. Die einzelnen Nach-

richten werden mit einer vorgegebenen **Reduktionsoperation** (einer assoziativen und kommutativen binären Operation) elementweise miteinander kombiniert, so dass am Wurzelprozessor P_i nur noch *eine* (zusammengesetzte) Nachricht eintrifft. Eine Einzel-Akkumulationsoperation mit einer Addition als Reduktionsoperation und P_1 als Wurzel, zu der Prozessor P_i die Nachricht x_i beiträgt ($i = 1, \dots, p$), kann wie folgt veranschaulicht werden:



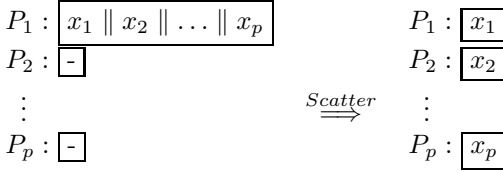
Zur Durchführung einer Einzel-Akkumulationsoperation führt *jeder* Prozessor eine Akkumulationsanweisung aus, in der er die Wurzel der Akkumulationsoperation, die anzuwendende Reduktionsoperation und die von ihm zur Verfügung gestellten Daten spezifiziert. Der Wurzelprozessor gibt zusätzlich einen Akkumulationspuffer an, in dem das Resultat der Akkumulationsoperation aufgesammelt wird.

- **Gather:** Bei einer Gatheroperation schickt jeder Prozessor an einen ausgezeichneten Prozessor P_i (Wurzel) eine Nachricht. Der Wurzelprozessor sammelt die Nachrichten ohne Reduktion auf, d.h. Prozessor P_i erhält p Nachrichten. Die Operation kann wie folgt veranschaulicht werden:



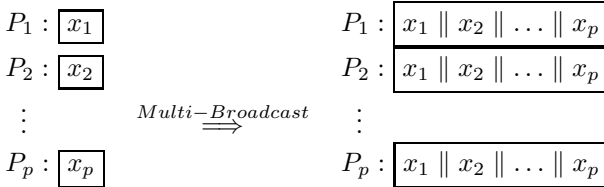
Dabei bezeichnet $\parallel$ die Konkatenation der empfangenen Nachrichten. Zur Durchführung einer Gatheroperation führt *jeder* Prozessor eine Gatheranweisung aus, in der er die Wurzel der Gatheroperation und die von ihm zur Verfügung gestellten Daten angibt. Der Wurzelprozessor spezifiziert zusätzlich einen Puffer zum Aufsammeln der Nachrichten, der groß genug sein muss, um die Nachrichten aller Prozessoren aufzunehmen. Nach Abschluss der Operation enthält dieser Puffer die Nachrichten der beteiligten Prozessoren in der Reihenfolge der Prozessornummern.

- **Scatter:** Bei einer Scatteroperation schickt ein ausgewählter Prozessor P_i (Wurzel) an jeden anderen Prozessor eine evtl. unterschiedliche Nachricht. Die Operation mit Wurzel P_1 kann wie folgt veranschaulicht werden:



Zur Durchführung einer Scatteroperation führt jeder Prozessor eine Scatteranweisung aus, in der er die Wurzel der Scatteroperation und einen Empfangspuffer angibt. Der Wurzelprozessor spezifiziert zusätzlich einen Sendepuffer, in den er vor dem Start der Operation die für die verschiedenen Prozessoren bereitgestellten Daten in der Reihenfolge der Prozessornummern ablegt.

- **Multi-Broadcast:** Bei einer Multi-Broadcastoperation führt jeder Prozessor eine Einzel-Broadcastoperation aus, d.h. jeder Prozessor schickt an jeden anderen Prozessor die *gleiche* Nachricht. Umgekehrt empfängt jeder Prozessor von jedem anderen Prozessor eine Nachricht, wobei unterschiedliche Empfangsknoten vom gleichen Sendeknoten die gleiche Nachricht erhalten. Die Operation kann wie folgt veranschaulicht werden:



Bei einer Multi-Broadcastoperation gibt es im Gegensatz zu den bereits aufgeführten globalen Kommunikationsoperationen keinen ausgezeichneten Wurzelprozessor. Zur Durchführung der Operation führt jeder Prozessor eine Multi-Broadcastanweisung aus, in der er einen Sendepuffer, der die für die anderen Prozessoren zur Verfügung gestellte Nachricht enthält, und einen Empfangspuffer spezifiziert. Nach Abschluss der Operation enthält der Empfangspuffer jedes Prozessors die von den anderen Prozessoren zur Verfügung gestellten Nachrichten (einschließlich seiner eigenen Nachricht) in der Reihenfolge der Prozessornummern. Multi-Broadcastoperationen sind also gut für das Aufsammeln von verteilt berechneten Feldern geeignet, die allen Prozessoren repliziert zur Verfügung gestellt werden sollen.

- **Multi-Akkumulation:** Bei einer Multi-Akkumulationsoperation führt jeder Prozessor eine Einzel-Akkumulation aus, d.h. jeder Prozessor stellt für jeden anderen Prozessor eine evtl. unterschiedliche Nachricht zur Verfügung. Die Nachrichten für den gleichen Empfangsknoten werden mit einer vorgegebenen Reduktionsoperation kombiniert, so dass an jedem Empfangsknoten *eine* (zusammengesetzte) Nachricht eintrifft. Es gibt also keinen ausgezeichneten Wurzelprozessor. Die Operation kann bei Verwendung einer Addition als Reduktionsoperation wie folgt veranschaulicht werden:

$$\begin{array}{ccc}
P_1 : & \boxed{x_{11} \parallel x_{12} \parallel \dots \parallel x_{1p}} & P_1 : \boxed{x_{11} + x_{21} + \dots + x_{p1}} \\
P_2 : & \boxed{x_{21} \parallel x_{22} \parallel \dots \parallel x_{2p}} & P_2 : \boxed{x_{12} + x_{22} + \dots + x_{p2}} \\
\vdots & & \vdots \\
P_p : & \boxed{x_{p1} \parallel x_{p2} \parallel \dots \parallel x_{pp}} & P_p : \boxed{x_{1p} + x_{2p} + \dots + x_{pp}}
\end{array}
\quad \xRightarrow{\text{Multi-Akkumulation}}$$

Dabei bezeichnet x_{ij} die von Prozessor P_i für Prozessor P_j zur Verfügung gestellte Nachricht. Zur Durchführung einer Multi-Akkumulationsoperation führt *jeder* Prozessor eine Multi-Akkumulationsanweisung aus, in der er einen Sendepuffer, einen Empfangspuffer und eine Reduktionsoperation angibt. Im Sendepuffer stellt jeder Prozessor vor dem Start der Operation die für die anderen Prozessoren bestimmten Nachrichten in der Reihenfolge der Prozessornummern zur Verfügung. Der Empfangspuffer jedes Prozessors enthält nach der Beendigung der Operation das durch die zugehörige Akkumulation erhaltene Ergebnis.

- **Gesamtaustausch (total exchange):** Bei einer Gesamtaustauschoperation schickt jeder Prozessor an jeden anderen Prozessor eine evtl. unterschiedliche Nachricht, ohne dass eine Reduktionsoperation angewendet wird, d.h. jeder Prozessor führt eine Scatteroperation durch. Umgekehrt empfängt jeder Prozessor von jedem anderen Prozessor eine evtl. unterschiedliche Nachricht, d.h. jeder Prozessor führt eine Gatheroperation aus. Es gibt also keinen ausgezeichneten Wurzelprozessor. Die Operation kann wie folgt veranschaulicht werden:

$$\begin{array}{ccc}
P_1 : & \boxed{x_{11} \parallel x_{12} \parallel \dots \parallel x_{1p}} & P_1 : \boxed{x_{11} \parallel x_{21} \parallel \dots \parallel x_{p1}} \\
P_2 : & \boxed{x_{21} \parallel x_{22} \parallel \dots \parallel x_{2p}} & P_2 : \boxed{x_{12} \parallel x_{22} \parallel \dots \parallel x_{p2}} \\
\vdots & & \vdots \\
P_p : & \boxed{x_{p1} \parallel x_{p2} \parallel \dots \parallel x_{pp}} & P_p : \boxed{x_{1p} \parallel x_{2p} \parallel \dots \parallel x_{pp}}
\end{array}
\quad \xRightarrow{\text{Gesamtaustausch}}$$

Zur Durchführung einer Gesamtaustauschoperation führt *jeder* Prozessor eine Gesamtaustauschanweisung aus, in der er einen Sendepuffer und einen Empfangspuffer angibt. Der Sendepuffer jedes Prozessors enthält vor dem Start der Operation die für die anderen Prozessoren zur Verfügung gestellten Nachrichten in der Reihenfolge der Prozessornummern. Der Empfangspuffer jedes Prozessors enthält nach der Beendigung der Operation die von den anderen Prozessoren empfangenen Nachrichten, ebenfalls in der Reihenfolge der Prozessornummern.

Auf die Realisierung der Kommunikationsoperationen in verschiedenen Netzwerken und die sich ergebenden Laufzeiten gehen wir in Abschnitt 4.3.1 ein. In Kapitel 5 beschreiben wir die Nutzung von Kommunikationsoperationen in den Laufzeitbibliotheken MPI und PVM.

Dualität von Kommunikationsoperationen. Eine Einzel-Broadcastoperation kann mit Hilfe eines *aufspannenden Baumes* realisiert werden, wobei der sendende Prozessor die Wurzel des Baumes darstellt und die Baumverbindungen Verbindungen im zugrundeliegenden Verbindungsnetzwerk entsprechen. Das Netzwerk wird wie in Abschnitt 2.5.1 als Graph dargestellt. Ein **aufspannender Baum** eines Graphen $G = (V, E)$ ist ein Teilgraph $G' = (V, E')$, der alle Knoten V und eine Teilmenge der Kanten $E' \subseteq E$ enthält und einen Baum darstellt, also ein zusammenhängender Graph ohne Zyklen ist. Die Konstruktion von aufspannenden Bäumen für verschiedene statische Netzwerke wird in Abschnitt 4.3.1 beschrieben.

Für einen gegebenen aufspannenden Baum wird eine *Einzel-Broadcastoperation* durch einen Top-down-Lauf über diesen aufspannenden Baum dadurch realisiert, dass, ausgehend von der Wurzel, jeder Knoten die zu verschickende Nachricht jeweils an alle Kinder weitergibt, sobald die Nachricht bei ihm eingetroffen ist. Dabei können zur gleichen Zeit Nachrichten auf verschiedenen Kanten gesendet werden. Die Kanten des aufspannenden Baumes können in *Stufen* eingeteilt werden, so dass über alle Kanten derselben Stufe gleichzeitig gesendet werden kann. Abbildung 3.6 (links) zeigt einen Baum mit drei Stufen, bei dem von Knoten P_1 aus Nachrichten gesendet werden. Die Stufen sind mit 0, 1, 2 gekennzeichnet.

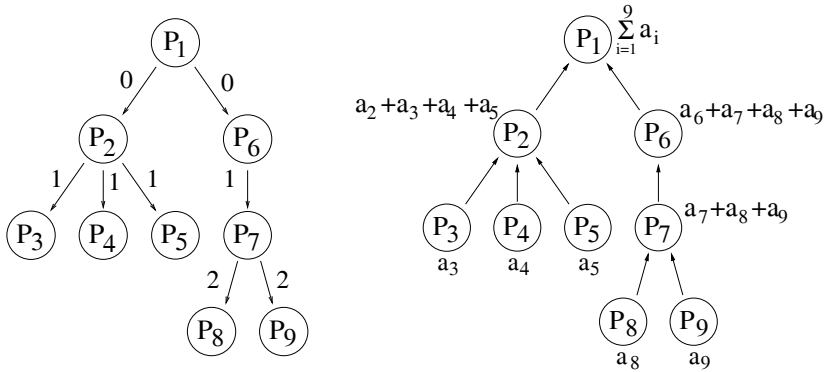


Abb. 3.6. Realisierung einer Einzel-Broadcastoperation mit Hilfe eines aufspannenden Baumes (links). Die Kanten des Baumes sind mit den Sendezeitpunkten (Stufen) annotiert. Realisierung einer Einzel-Akkumulationsoperation mit Hilfe des gleichen aufspannenden Baumes (rechts). Prozessor P_i stellt den Wert a_i zur Verfügung, $i = 1, \dots, 9$. Das Ergebnis wird beim Wurzelprozessor P_1 aufgesammelt.

Ebenso wie eine Einzel-Broadcastoperation kann eine *Einzel-Akkumulationsoperation* mit Hilfe eines aufspannenden Baumes realisiert werden, wobei der aufsammlende Prozessor die Wurzel dieses Baumes ist. Die Reduktion erfolgt dann gemäß der angegebenen Operation an den inneren Knoten und wird durch einen Bottom-up-Lauf über den Baum realisiert, vgl. Abbildung

3.6 (rechts). Jeder Knoten des aufspannenden Baumes empfängt von jedem seiner Kinder (falls vorhanden) eine Nachricht, kombiniert die empfangenen Nachrichten entsprechend der anzuwendenden Reduktionsoperation mit seiner eigenen Nachricht und schickt das Resultat an seinen Elternknoten weiter. Damit wird über jede Kante des aufspannenden Baumes eine Nachricht verschickt, jedoch in umgekehrter Richtung wie beim Einzel-Broadcast. Da die gleichen aufspannenden Bäume verwendet werden können, werden Einzel-Broadcast- und Einzel-Akkumulationsoperation als zueinander *duale* Operationen bezeichnet.

Eine duale Beziehung besteht auch zwischen der Gather- und der Scatteroperation sowie zwischen der Multi-Broadcast- und der Multi-Akkumulationsoperation. Eine *Scatteroperation* kann ebenfalls mit Hilfe eines Top-down-Laufes über einen aufspannenden Baum realisiert werden, wobei jeder Prozessor an seine Kindknoten diejenigen der von seinem Vaterknoten erhaltenen Daten weitergibt, die für Prozessoren im entsprechenden Unterbaum bestimmt sind. Somit sinkt die Anzahl der über eine Kante weitergegebenen Nachrichten auf dem Weg vom Wurzelknoten zu den Blattknoten. Analog kann eine *Gatheroperation* durch einen Bottom-up-Lauf über einen aufspannenden Baum realisiert werden, wobei jeder Knoten an seinen Elternknoten die von seinen Kindknoten erhaltenen Daten und seine eigenen Daten weiterschickt. Somit wächst die Anzahl der von einem Knoten an seinen Elternknoten weitergegebenen Nachrichten auf dem Weg von den Blattknoten zum Wurzelknoten. Auf jedem Pfad zur Wurzel wird dabei die gleiche Gesamtanzahl von Nachrichten wie bei einer Scatteroperation verschickt, aber in umgekehrter Reihenfolge. Daher sind Scatter und Gatheroperationen zueinander duale Operationen. Eine *Multi-Broadcastoperation* kann durch Verwendung von p aufspannenden Bäumen realisiert werden, wobei jeder aufspannende Baum einen anderen Prozessor als Wurzel hat. Im Idealfall treten keine Überlappungen von Verbindungen unterschiedlicher aufspannender Bäume auf, auf denen zur gleichen Zeit gesendet werden soll. Eine *Multi-Akkumulationsoperation* kann ebenfalls durch Verwendung von p aufspannenden Bäumen realisiert werden, wobei wieder die Kommunikationsrichtung im Vergleich zur Multi-Broadcastoperation umgedreht wird. Daher stellen auch Multi-Broadcast- und Multi-Akkumulationsoperation zueinander duale Operationen dar.

Hierarchie von Kommunikationsoperationen. Die beschriebenen Kommunikationsoperationen stehen in einer hierarchischen Beziehung zueinander, die dadurch entsteht, dass die Kommunikationsoperationen sich durch schrittweise *Spezialisierung* aus der allgemeinsten Kommunikationsoperation ergeben. Die allgemeinste der Kommunikationsoperationen ist der Gesamtaustausch, da jeder Prozessor eine evtl. *unterschiedliche* Nachricht an jeden anderen Prozessor schickt. Eine Multi-Broadcastoperation ist ein Spezialfall des Gesamtaustausches, in dem jeder Prozessor an jeden anderen Prozessor die *gleiche* Nachricht schickt, d.h. anstatt p verschiedener Nach-

richten stellt jeder Prozessor nur eine Nachricht zur Verfügung. Eine Multi-Akkumulationsoperation ist ebenfalls ein Spezialfall des Gesamtaustausches, in dem die an einem Prozessor ankommenden Nachrichten auf den Zwischenstufen entsprechend der angegebenen Reduktionsoperation kombiniert werden. Eine Gatheroperation mit Wurzelprozessor P_i ist ein Spezialfall einer Multi-Broadcastoperation, der sich aus der Betrachtung *eines* empfangenden Prozessors P_i der Multi-Broadcastoperation ergibt, der von jedem Prozessor eine andere Nachricht empfängt. Analog ist eine Scatteroperation mit Wurzel P_i ein Spezialfall einer Multi-Akkumulationsoperation, der sich aus der Verwendung einer speziellen Reduktionsoperation ergibt, die nur die von Prozessor P_i kommenden Nachrichten weitergibt. Eine Einzel-Broadcastoperation ist ein Spezialfall einer Scatteroperation, in dem der sendende Prozessor an jeden anderen Prozessor des Netzwerkes die gleiche Nachricht weitergibt, d.h. der Wurzelprozessor stellt anstatt p verschiedener Nachrichten nur eine Nachricht zur Verfügung. Eine Einzel-Akkumulationsoperation ist ein Spezialfall einer Gatheroperation, in dem auf den Zwischenstufen eine Reduktion durchgeführt wird, die dazu führt, dass beim Empfangsknoten nur *eine* (kombinierte) Nachricht ankommt. Ein Einzeltransfer zwischen Prozessor P_i und P_j ist ein Spezialfall einer Einzel-Broadcastoperation mit Wurzel i , in dem nur der Pfad zwischen P_i und P_j relevant ist. Der Einzeltransfer ist auch ein Spezialfall einer Einzel-Akkumulationsoperation mit Wurzelprozessor P_j , in dem eine Reduktionsoperation verwendet wird, die nur die von P_i kommende Nachricht weitergibt. Damit ergibt sich die in Abbildung 3.7 wiedergegebene Hierarchie.

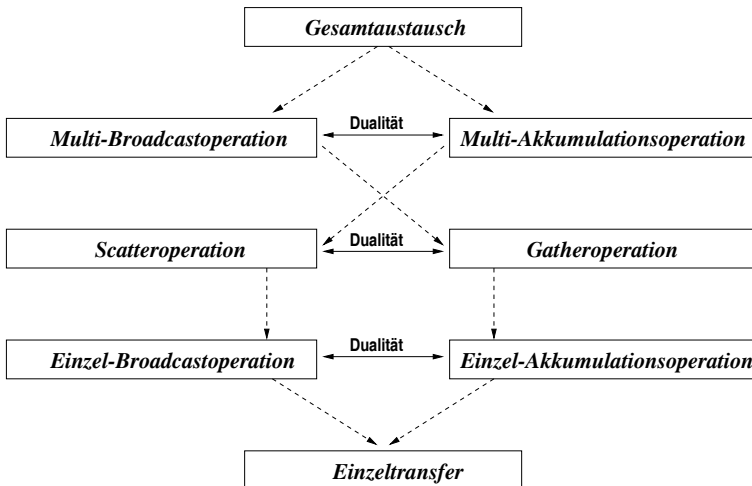


Abb. 3.7. Hierarchie der Kommunikationsoperationen. Die horizontalen Pfeile zeigen Dualitätsbeziehungen, die gestrichelten Pfeile zeigen Spezialisierungen [14].

3.7.3 Parallele Matrix-Vektor-Multiplikation

Matrix-Vektor-Multiplikationen kommen häufig als Basisoperationen in numerischen Algorithmen vor. Wir betrachten die Matrix-Vektor-Multiplikation hier als ein erstes Beispiel für die bei einer parallelen Implementierung auftretenden Fragestellungen. Dazu betrachten wir die Multiplikation einer dicht besetzten $n \times m$ -Matrix $\mathbf{A} \in \mathbb{R}^{n \times m}$ mit einem Vektor $\mathbf{b} \in \mathbb{R}^m$. Zur Verdeutlichung werden wir in diesem Abschnitt Matrizen und Vektoren in Fettdruck und skalare Werte in Normaldruck darstellen. Der sequentielle Algorithmus für die Berechnung des Resultates $\mathbf{c} = (c_1, \dots, c_n) \in \mathbb{R}^n$ mit

$$c_i = \sum_{j=1}^m a_{ij} b_j, \quad i = 1, \dots, n,$$

lässt sich in zwei Formen aufschreiben, die sich in der Reihenfolge der Schleifendurchläufe über i und j unterscheiden. Betrachtet man die Matrix-Vektor-Multiplikation als Berechnung n innerer Produkte der Zeilen $\mathbf{a}_1, \dots, \mathbf{a}_n$ von $\mathbf{A}$ mit dem Vektor $\mathbf{b}$, also als

$$\mathbf{A} \cdot \mathbf{b} = \begin{pmatrix} (\mathbf{a}_1, \mathbf{b}) \\ \vdots \\ (\mathbf{a}_n, \mathbf{b}) \end{pmatrix},$$

wobei $(\mathbf{x}, \mathbf{y}) = \sum_{j=1}^m x_j y_j$ für $\mathbf{x}, \mathbf{y} \in \mathbb{R}^m$ mit $\mathbf{x} = (x_1, \dots, x_m)$ und $\mathbf{y} = (y_1, \dots, y_m)$ das innere Produkt bzw. Skalarprodukt bezeichnet, so entspricht dies folgendem Algorithmus (in C-Notation):

```
for (i=0; i<n; i++) c[i] = 0;
for (i=0; i<n; i++)
    for (j=0; j<m; j++)
        c[i] = c[i] + A[i][j] * b[j];
```

Die Matrix $\mathbf{A}$ entspricht hierbei dem zweidimensionalen Feld `A` und die Vektoren $\mathbf{b}$ und $\mathbf{c}$ entsprechen den eindimensionalen Feldern `b` und `c` im Programm¹. Für jedes $i = 0, \dots, n-1$ besteht der Schleifenrumpf aus einer inneren Schleife über j , die ein inneres Produkt berechnet. Eine andere Sichtweise besteht darin, die Matrix-Vektor-Multiplikation als *Linearkombination* der Spalten $\tilde{\mathbf{a}}_1, \dots, \tilde{\mathbf{a}}_m$ von $\mathbf{A}$ aufzufassen, wobei $\mathbf{b} \in \mathbb{R}^n$ die Koeffizienten dieser Linearkombination enthält, d.h.

$$\mathbf{A} \cdot \mathbf{b} = \sum_{j=1}^m b_j \tilde{\mathbf{a}}_j.$$

¹ Man beachte, dass die Numerierung in der mathematischen Beschreibung wie in der Mathematik üblich bei 1 beginnt. Die Numerierung der Felder in den Programmen beginnt dagegen wie von C vorgegeben bei 0.

Diese Sichtweise wird durch den Algorithmus

```
for (i=0; i<n; i++) c[i] = 0;
for (j=0; j<m; j++)
    for (i=0; i<n; i++)
        c[i] = c[i] + A[i][j] * b[j];
```

ausgedrückt, in dem für jedes $j = 0, \dots, m-1$ die Ausführung des Schleifenrumpfes dem Hinzufügen einer Spalte $\tilde{\mathbf{a}}_j$ zur Linearkombination entspricht. Die beiden sequentiellen Programme ergeben das gleiche Resultat, da die Schleifen über i und j aufgrund fehlender Datenabhängigkeiten vertauscht werden dürfen. Die verschiedenen Sichtweisen der Matrix-Vektor-Multiplikation legen jedoch verschiedene Strategien für eine parallele Implementierung nahe, und zwar

- (a) die Berechnung der n inneren Produkte $(\mathbf{a}_i, \mathbf{b})$ für $i = 1, \dots, n$ wird so auf die Prozessoren verteilt, dass jeder der p Prozessoren ungefähr n/p Skalarprodukte berechnet oder
- (b) die Berechnung der Linearkombination $\sum_{j=1}^m b_j \tilde{\mathbf{a}}_j$ wird auf die Prozessoren verteilt, so dass jeder der p Prozessoren einen Teil der Linearkombination mit ungefähr m/p Spaltenvektoren berechnet.

Wir untersuchen diese beiden Möglichkeiten im Folgenden etwas näher. Dabei nehmen wir zur Vereinfachung an, dass n und m Vielfache von p sind.

Verteilung der Skalarprodukte. Für eine Realisierung auf Rechnern mit verteiltem Adressraum wird die Datenverteilung für $\mathbf{A}$ und $\mathbf{b}$ so gewählt, dass ein Prozessor, der die Berechnung eines inneren Produktes $(\mathbf{a}_i, \mathbf{b})$ mit $i \in \{1, \dots, n\}$ ausführt, die entsprechende Zeile $\mathbf{a}_i$ von $\mathbf{A}$ und den Vektor $\mathbf{b}$ im lokalen Speicher vorliegen hat. Da $\mathbf{b} \in \mathbb{R}^m$ für alle inneren Produkte benötigt wird, wird dieser Vektor repliziert gehalten. Für die Matrix $\mathbf{A}$ ist eine zeilenorientierte streifenweise Datenverteilung sinnvoll. Jeder Prozessor berechnet diejenigen inneren Produkte, für die ihm die zugehörige Zeile von $\mathbf{A}$ in seinem lokalen Speicher zur Verfügung steht.

Wir betrachten zunächst eine *blockweise streifenweise Datenverteilung* für $\mathbf{A}$, d.h. Prozessor P_k , $k = 1, \dots, p$, hat die Zeilen $\mathbf{a}_i$ für $i = n/p \cdot (k-1) + 1, \dots, n/p \cdot k$ in seinem lokalen Speicher und berechnet die Skalarprodukte $(\mathbf{a}_i, \mathbf{b})$. Für die Berechnung werden keine Daten benötigt, die in lokalen Speichern anderer Prozessoren abgelegt sind, d.h. es wird keine Kommunikation ausgeführt. Der Ergebnisvektor $\mathbf{c} = (c_1, \dots, c_n)$ liegt nach Abschluss der lokalen Berechnungen in *blockweiser* Verteilung vor. Da die Matrix-Vektor-Multiplikation häufig als Teilberechnung innerhalb einer größeren Anwendung vorkommt, gibt es oft Anforderungen an die Verteilung des Ergebnisvektors. In einem Iterationsverfahren wird für den Ergebnisvektor $\mathbf{c}$ meist dieselbe Datenverteilung wie für den Vektor $\mathbf{b}$ erwartet, was in unserem Fall die replizierte Verteilung wäre. Eine replizierte Datenverteilung von $\mathbf{c}$ kann erreicht werden, indem jeder Prozessor P_k für $k = 1, \dots, p$ seinen Block

$(c_{n/p \cdot (k-1)+1}, \dots, c_{n/p \cdot k})$ an alle anderen Prozessoren sendet. Dies kann durch eine Multi-Broadcastoperation realisiert werden. Insgesamt hat die parallele Realisierung der Matrix-Vektor-Multiplikation folgende Form:

```

local_n = n/p;
for (i=0; i<local_n; i++) local_c[i] = 0;
for (i=0; i<local_n; i++)
    for (j=0; j<m; j++)
        local_c[i] = local_c[i] + local_A[i][j] * b[j];
multi_broadcast(local_c, local_n, c);

```

Dieses Programm wird von allen Prozessoren P_k mit $k = 1, \dots, p$ im SPMD-Stil abgearbeitet. Die Kommunikationsoperation nach der geschachtelten Schleife bewirkt eine Synchronisation. Das lokale Feld `local_A` der Dimension `local_n` $\times$ `m` eines Prozessors P_k ist jeweils mit einem Zeilenblock der globalen $n \times m$ -Matrix **A** folgendermaßen belegt:

$$\text{local\_A}[i][j] = \mathbf{A}[i + (k - 1) * n/p][j]$$

für $i = 0, \dots, n/p - 1$, $j = 0, \dots, m - 1$ und $k = 1, \dots, p$. Das Ergebnis der Operation wird von jedem Prozessor in seinem lokalen Feld `local_c` der Größe `local_n` abgespeichert. Die Multi-Broadcastoperation wird durch den Aufruf `multi_broadcast(local_c, local_n, c)` realisiert, zu der jeder Prozessor die von ihm in seinem lokalen Feld `local_c` errechneten `local_n` Elemente beiträgt. Nach der Multi-Broadcastoperation gilt:

$$c[i + (k - 1) * n/p] = \text{local\_c}[i]$$

für $i = 0, \dots, n/p - 1$ und $k = 1, \dots, p$. Das globale Feld `c` enthält nach der Berechnung die Werte der lokalen Felder `local_c` der verschiedenen Prozessoren in der Reihenfolge ihrer Prozessornummern, siehe Abbildung 3.8(1).

Bei Verwendung einer *zyklischen streifenweisen Datenverteilung* speichert Prozessor P_k die Zeilen $\mathbf{a}_i$ von **A** mit $i = k + p \cdot (l - 1)$ für $l = 1, \dots, n/p$ in seinem lokalen Speicher und errechnet die zugehörigen Komponenten des Ergebnisvektors, $k = 1, \dots, p$. Die Ablage dieser Zeilen kann wieder in einem Feld `local_A` der Dimension `local_n` $\times$ `m` erfolgen. Nach dem Austausch der verteilt errechneten Komponenten des Ergebnisvektors durch eine Multi-Broadcastoperation müssen diese Komponenten aber entsprechend umgeordnet werden, um die korrekte Reihenfolge der Elemente herzustellen.

Eine Realisierung für Rechner mit gemeinsamem Adressraum kann ähnlich der Realisierung für Rechner mit verteiltem Adressraum erfolgen. Der Unterschied besteht darin, dass für einen gemeinsamen Adressraum keine Verteilung der Daten, sondern nur eine Verteilung der Berechnungen erforderlich ist. Die Berechnungen können analog der Implementierung für verteilten Adressraum verteilt werden, d.h. jeder Prozessor errechnet n/p Komponenten des Ergebnisvektors und verwendet dafür n/p Zeilen der Matrix **A**,

die blockweise oder zyklisch zugeordnet werden können. Da bei der Berechnung der inneren Produkte keine Zugriffskonflikte auf gemeinsame Variablen auftreten, braucht kein Sperrmechanismus verwendet zu werden. Die Berechnungen der inneren Produkte lesen den gemeinsamen Vektor $\mathbf{b}$, beschreiben aber disjunkte Bereiche des gemeinsamen Vektors $\mathbf{c}$. Dies wird dadurch erreicht, dass in der Berechnung die private Variable k verwendet wird, die die Prozessornummer des ausführenden Prozessors enthält $k = 1, \dots, p$. Nach der Berechnung des Ergebnisvektors muss vor einer Weiterverwendung eine Synchronisationsoperation ausgeführt werden, um sicherzustellen, dass alle Komponenten des Ergebnisvektors aktualisiert wurden. Damit ergibt sich folgendes SPMD-Programm für die *blockweise* Verteilung:

```

local_n = n/p;
for (i=0; i<local_n; i++) c[i+(k-1)*local_n] = 0;
for (i=0; i<local_n; i++)
    for (j=0; j<m; j++)
        c[i+(k-1)*local_n] =
            c[i+(k-1)*local_n] + A[i+(k-1)*local_n][j] * b[j];
synch();

```

Verteilung der Linearkombinationen. Wird die Berechnung der Linearkombination $\sum_{j=1}^m b_j \tilde{\mathbf{a}}_j$ auf die Prozessoren eines Rechners mit verteiltem Adressraum aufgeteilt, so legt jeder Prozessor die Spalten von $\mathbf{A}$ in seinem lokalen Speicher ab, die er für die Berechnung des ihm zugeordneten Teils der Linearkombination verwenden muss. Dies entspricht einer spaltenorientierten streifenweisen Datenverteilung von $\mathbf{A}$. Liegt eine *blockweise streifenweise Datenverteilung* von $\mathbf{A}$ vor, so hat Prozessor P_k die Spalten $\tilde{\mathbf{a}}_i$ mit $i = m/p \cdot (k-1) + 1, \dots, m/p \cdot k$ lokal zur Verfügung und berechnet die Teil-Linearkombination

$$\mathbf{d}_k = \sum_{j=m/p \cdot (k-1) + 1}^{m/p \cdot k} b_j \tilde{\mathbf{a}}_j.$$

Dazu benötigt Prozessor P_k nur einen Block der Komponenten des Vektors $\mathbf{b}$. Nach der Berechnung hat jeder Prozessor P_k einen Vektor $\mathbf{d}_k$ der Länge n als Ergebnis. Für die Errechnung der gesamten Linearkombination, die sich durch Addition der Ergebnisse der Teil-Linearkombinationen ergibt, kann eine Akkumulationsoperation mit einer Addition als Reduktionsoperation verwendet werden, zu der jeder Prozessor P_k die von ihm errechnete Teil-Linearkombination $\mathbf{d}_k$ beiträgt. Wenn das Ergebnis repliziert vorliegen soll, muss der aufsammelnde Prozessor dieses anschließend mit einer Einzel-Broadcastoperation an die anderen Prozessoren verteilen, vgl. Abbildung 3.8(2a). Es resultiert das folgende Programm, das von allen Prozessoren im SPMD-Stil auf den lokalen Daten `local_b` und `local_A` abgearbeitet wird.

```

local_m=m/p;
for (i=0; i<n; i++) d[i] = 0;
for (j=0; j<local_m; j++)
    for (i=0 ;i<n; i++)
        d[i] = d[i] + local_b[j] * local_A[i][j];
single_accumulation(d,local_m,c,ADD,1);
single_broadcast(c,1);

```

Die lokalen Felder `local_b` und `local_A` von Prozessor P_k haben die Werte

$$\text{local\_A}[i][j] = A[i][j + (k - 1) * m/p]$$

und

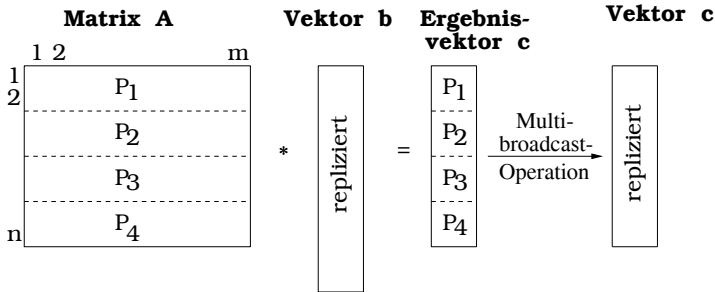
$$\text{local\_b}[j] = b[j + (k - 1) * m/p]$$

für $j = 0, \dots, m/p - 1$, $i = 0, \dots, n - 1$ und $k = 1, \dots, p$. Der Vektor `d` ist ein privater Vektor, den jeder Prozessor zur Berechnung eines Teilergebnisses nutzt. Zur Akkumulation `single_accumulation(d,local_m,c,ADD,1)` trägt jeder Prozessor seinen lokal errechneten Vektor `d` mit `local_m` Komponenten bei. Als Reduktionsoperation wird eine Addition (ADD) verwendet. Prozessor P_1 ist die Wurzel der Akkumulation (Angabe durch den letzten Parameter) und speichert die aktuellen Werte in seinem lokalen Feld `c`. Von dort werden sie mit der nachfolgenden Broadcastanweisung `single_broadcast(c,1)` an alle anderen Prozessoren verteilt.

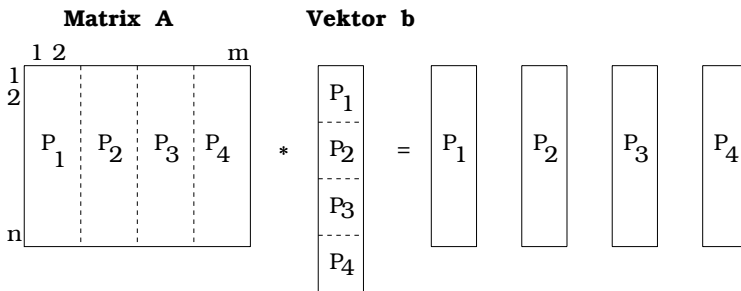
Soll der Ergebnisvektor in einer *blockweisen* Verteilung vorliegen, kann alternativ auch eine Multi-Akkumulationsoperation zum Aufsummeln verwendet werden, bei der jeder Prozessor n/p Komponenten des Ergebnisvektors `c` mit einer Additionsoperation aufsummiert, vgl. Abbildung 3.8(2b). Diese Variante ist insbesondere für den Fall $n = m$ interessant, da in diesem Fall jeder Prozessor genau die gleichen Komponenten des Ergebnisvektors `c` errechnet, die er von `b` abgespeichert hat, d.h. nach Durchführung der Multi-Akkumulationsoperation kann `c` ohne Kommunikation für eine weitere Matrix-Vektor-Multiplikation $A \cdot c$ mit derselben Datenverteilung verwendet werden. Da die Multi-Akkumulationsoperation die duale Operation zur Multi-Broadcastoperation ist, haben diese Variante der parallelen Berechnung der Linearkombination und die parallele Berechnung der Skalarprodukte einen ähnlichen Berechnungs- und Kommunikationsaufwand, worauf wir in Abschnitt 4.4.2 eingehen werden. Die Entscheidung, welche Variante der parallelen Matrix-Vektor-Multiplikation man verwendet, hängt im wesentlichen von der vorliegenden Datenverteilung von A ab.

Für Rechner mit gemeinsamem Adressraum ist die skizzierte Verteilung der Berechnung der Linearkombinationen weniger geeignet als eine Verteilung der Skalarprodukte, da jeder Prozessor zum Errechnen der Teil-Linearkombinationen einen eigenen Vektor verwenden muss, um Zugriffskonflikte zu vermeiden, und da die Addition der Teil-Linearkombinationen bei Fehlen einer Akkumulationsoperation sequenzialisiert werden muss.

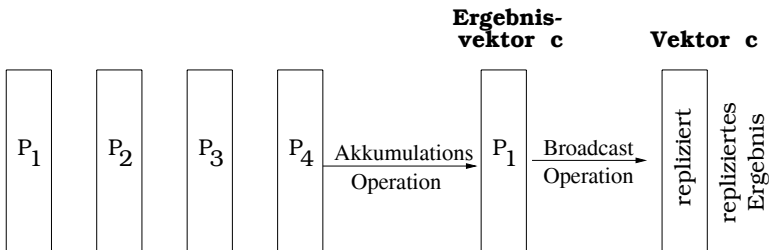
1) Parallele Berechnung innerer Produkte



2) Parallele Berechnung der Linearkombinationen



2a)



2b)

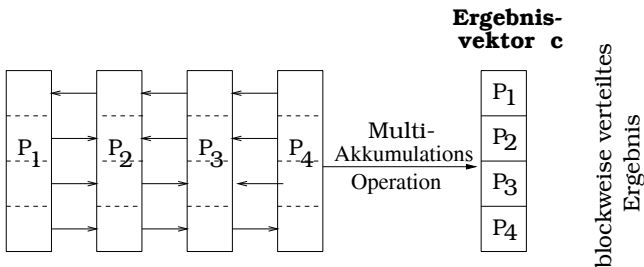


Abb. 3.8. Parallele Matrix-Vektor-Multiplikation mit 1) paralleler Berechnung innerer Produkte mit repliziertem Ergebnis und 2) paralleler Berechnung der Linearkombination für a) repliziertes Ergebnis und b) blockweise verteiltes Ergebnis.

4. Laufzeitanalyse paralleler Programme

Ein wesentlicher Grund für den Einsatz eines Parallelrechners ist eine Reduzierung der Laufzeit von Anwendungsprogrammen. Das Ableiten von Aussagen über die Laufzeit paralleler Programme oder sogar eine Analyse dieser Laufzeit sind daher ein wichtiges Anliegen in der Parallelverarbeitung. Dabei steht zunächst nicht fest, in welcher Weise und mit welchen *Leistungsmaßen* diese Aussagen getroffen werden sollen, da die Ausführung eines parallelen Programmes von vielen Faktoren beeinflusst wird. Diese Faktoren umfassen die Architektur der verwendeten parallelen Plattform, den Compiler, das Betriebssystem, die parallele Programmierungsumgebung einschließlich des zugrundeliegenden Programmiermodells und die Lokalitäts- und Abhängigkeits-Eigenschaften des auszuführenden Programmes. Leistungsmaße können diese Faktoren in unterschiedlich detaillierter Form einbeziehen. Entsprechend sind die unterschiedlichsten Ansätze zur Erfassung und Analyse der Laufzeit von parallelen Programmen verfolgt worden. Diese reichen von theoretischen Kostenmodellen bis zur Analyse von realen, gemessenen Laufzeiten.

In diesem Kapitel betrachten wir Verfahren zur Bewertung und zum Vergleich der Laufzeit paralleler Programme. Dazu fassen wir in Abschnitt 4.1 zunächst verschiedene Verfahren zur Leistungsbewertung *sequentieller Rechnersysteme* zusammen, die vor allem auf eine Bewertung der Architektur von Rechnersystemen ausgerichtet sind. In Abschnitt 4.2 geben wir einen Überblick über *Leistungsmaße*, die für die Bewertung von parallelen Programmen eingesetzt werden und die im wesentlichen darauf abzielen, die Laufzeit der parallelen Programme mit der Laufzeit der zugehörigen sequentiellen Programme zu vergleichen. Die Modellierung der Laufzeit von Kommunikationsoperationen mit Hilfe von *Laufzeitformeln* wird in Abschnitt 4.3 betrachtet. In Abschnitt 4.4 wird die Laufzeit eines Skalarproduktes und einer Matrix-Vektor-Multiplikation analysiert. In Abschnitt 4.5 geben wir einen kurzen Überblick über theoretische Kostenmodelle.

4.1 Leistungsbewertung von Rechnersystemen

Die Leistung (engl. *performance*) eines Rechnersystems ist ein wichtiges, vielleicht das wichtigste Bewertungskriterium. Dabei sind je nach Standpunkt

verschiedene Leistungsbewertungen sinnvoll. Der Benutzer des Rechnersystems ist an möglichst kleinen **Antwortzeiten** (engl. *response time*) interessiert, wobei die Antwortzeit als Zeit zwischen dem Start eines Programmes und dessen Beendigung definiert ist. Der Betreiber eines Rechenzentrums könnte dagegen hauptsächlich an einem möglichst hohen **Durchsatz** (engl. *throughput*) interessiert sein, wobei der Durchsatz die Anzahl der Arbeitseinheiten angibt, die im Mittel pro Zeiteinheit ausgeführt werden.

4.1.1 Bewertung der CPU-Leistung

Wir betrachten in diesem Teilabschnitt ein *sequentielles* Rechnersystem und als Leistungsmaß die *Antwortzeit*, bei der ein Rechnersystem eine umso höhere Leistung hat, je niedriger die Antwortzeiten für eine betrachtete Menge von Programmen sind. Die Antwortzeit bei der Ausführung eines Programmes setzt sich zusammen aus

- der Benutzer-CPU-Zeit, die die CPU bei der Ausführung des Programmes verbringt,
- der System-CPU-Zeit, die die CPU zur Ausführung von Betriebssystemroutinen braucht, die vom Benutzerprogramm aufgerufen werden, und
- der Wartezeit, die für das Warten auf I/O-Geräte und die Zeit zum Ausführen anderer Programme (bei Timesharing) verwendet wird.

Die Antwortzeit beinhaltet also die angegebenen Wartezeiten, die CPU-Zeit beinhaltet diese Wartezeiten nicht. Die verschiedenen Anteile der angegebenen Zeiten an der Antwortzeit erhält man z.B. im Unix-Betriebssystem mit dem `time`-Kommando. Wir vernachlässigen im Folgenden die Wartezeiten, da diese stark von der Auslastung des Rechnersystems abhängen, sowie die System-CPU-Zeit, da diese im wesentlichen von der Implementierung des Betriebssystems abhängt und betrachten nur die Ausführungszeiten, die durch eine direkte Umsetzung von Anweisungen des Benutzerprogramms in Maschinenbefehle (Instruktionen) entstehen [121].

Neben einer effizienten Umsetzung der Anweisungen des Programms in äquivalente Maschinenbefehlssequenzen durch den Compiler spielt bei der Benutzer-CPU-Zeit die Ausführungszeit der einzelnen Maschinenbefehle eine wesentliche Rolle. Einen großen Einfluss auf diese Ausführungszeit hat die **Zykluszeit** (engl. *cycle time* oder *clock cycle time*) des Prozessors, die sich als reziproker Wert der **Taktrate** (engl. *clock rate*) ergibt. So hat z.B. ein mit $2\text{ GHz} = 2 \cdot 10^9 \cdot 1/\text{s}$ getakteter Prozessor eine Zykluszeit von $1/(2 \cdot 10^9)\text{s} = 0.5 \cdot 10^{-9}\text{s} = 0.5\text{ ns}$. Wir werden die Zykluszeit im Folgenden als t_{cycle} bezeichnen. Für eine gegebene Zykluszeit ergibt sich die **Benutzer-CPU-Zeit** eines Programmes A , die wir als $T_{U\_CPU}(A)$ bezeichnen, durch Multiplikation mit der Anzahl $n_{\text{cycle}}(A)$ der CPU-Zyklen, die zur Ausführung aller Maschinenbefehle von Programm A gebraucht werden:

$$T_{U\_CPU}(A) = n_{\text{cycle}}(A) \cdot t_{\text{cycle}} . \quad (4.1)$$

Dabei können unterschiedliche Maschinenbefehle verschiedene Laufzeiten haben. Um eine Beziehung zu der Anzahl der für A ausgeführten Maschinenbefehle herzustellen, führt man die *mittlere Anzahl von CPU-Zyklen* ein, die für einen Maschinenbefehl von A gebraucht wird. Diese Größe wird als **CPI** (**C**lock **C**ycles **P**er **I**nstruction) bezeichnet. Der *CPI*-Wert eines Rechners hängt im allgemeinen vom ausgeführten Programm A ab, so dass sich für verschiedene Programme auf dem gleichen Rechner verschiedene *CPI*-Werte ergeben können. Für die Benutzer-CPU-Zeit ergibt sich damit

$$T_{U\_CPU}(A) = n_{instr}(A) \cdot CPI(A) \cdot t_{cycle}, \quad (4.2)$$

wobei $n_{instr}(A)$ die Anzahl der für A ausgeführten Maschinenbefehle bezeichnet. Diese Anzahl hängt zum einen von der *Architektur* des betrachteten Rechners, d.h. von den Maschinenbefehlen, die dieser Rechner zur Verfügung stellt, ab, da diese die Umsetzung der Konstrukte der verwendeten Programmiersprache in Maschinenprogrammsequenzen bestimmen, und zum anderen vom verwendeten *Compiler*, da dieser die genaue Umsetzung durchführt und dabei zwischen verschiedenen Möglichkeiten der Übersetzung auswählt, die üblicherweise auch zur Ausführung von verschieden vielen Maschinenbefehlen führen. Der *CPI*-Wert hängt für ein gegebenes Programm wesentlich von der internen Realisierung der Maschinenbefehle, d.h. von der Prozessorstruktur und dem Speichersystem, ab. Da unterschiedliche Maschinenbefehle unterschiedliche Ausführungszeiten haben, existiert auch eine Abhängigkeit vom Compiler, da dieser die verwendeten Maschinenbefehle auswählt.

Für einen Rechner, der n Maschinenbefehle $I_1, \dots, I_n$ zur Verfügung stellt, wobei CPI_i die mittlere Anzahl von CPU-Zyklen ist, die zur Ausführung eines Befehls I_i gebraucht wird, ergibt sich für die Anzahl der zur Ausführung eines Programmes A benötigten CPU-Zyklen

$$n_{cycle}(A) = \sum_{i=1}^n n_i(A) \cdot CPI_i, \quad (4.3)$$

wobei wir die Anzahl der Ausführungen von Maschinenbefehl I_i in A mit n_i bezeichnen. Die Gesamtanzahl der für ein Programm A ausgeführten Maschinenbefehle ist hingegen nur dann ein genaues Maß für die Anzahl der benötigten CPU-Zyklen für A , wenn alle Maschinenbefehle gleichviele Zyklen brauchen, d.h. gleiche Werte für CPI_i haben, wie folgendes Beispiel aus [121] zeigt.

Beispiel: Wir betrachten einen Rechner mit drei Instruktionsklassen $\mathcal{I}_1, \mathcal{I}_2, \mathcal{I}_3$, die 1, 2 bzw. 3 Zyklen zur Ausführung brauchen. Für ein Konstrukt einer Programmiersprache gebe es zwei Realisierungsmöglichkeiten, die zur Ausführung von unterschiedlichen Maschinenbefehlen entsprechend der folgenden Tabelle führen:

Realisierung	Instruktionen			Summe der Instruktionen	n_{cycle}
	$\mathcal{I}_1$	$\mathcal{I}_2$	$\mathcal{I}_3$		
1	2	1	2	5	10
2	4	1	1	6	9

Realisierung 2 braucht weniger Zyklen als Realisierung 1, obwohl sie aus mehr Maschinenbefehlen besteht. Für Realisierung 1 ergibt sich ein CPI -Wert von $10/5 = 2$, für Realisierung 2 ein CPI -Wert von $9/6 = 1.5$. $\square$

4.1.2 MIPS und MFLOPS

Ein weiteres in der Praxis verwendetes Maß zur Bestimmung der Leistung eines Rechnersystems ist die für ein Programm A erreichte **MIPS-Rate**, wobei MIPS für **M**illion **I**nstructions **P**er **S**econd (Millionen Instruktionen/Maschinenbefehle pro Sekunde) steht. Mit den oben eingeführten Notationen für die Anzahl der Maschinenbefehle $n_{instr}(A)$ und die Benutzer-CPU-Zeit $T_{U\_CPU}(A)$ ist die MIPS-Rate eines Programmes A definiert durch

$$MIPS(A) = \frac{n_{instr}(A)}{T_{U\_CPU}(A) \cdot 10^6} . \quad (4.4)$$

Mit Hilfe von Gleichung (4.2) lässt sich diese Gleichung umformen zu

$$MIPS(A) = \frac{r_{cycle}}{CPI(A) \cdot 10^6} ,$$

wobei $r_{cycle} = 1/t_{cycle}$ die Taktrate des Rechnersystems ist, die in $1/s$ gemessen wird. Schnellere Rechner haben also höhere MIPS-Raten als langsamere. Man beachte, dass die MIPS-Rate abhängig vom betrachteten Programm ist.

Die Verwendung von MIPS-Raten als Leistungsmaß hat zum einen den Nachteil, dass die MIPS-Rate eines Rechners von dessen Maschinenbefehlssatz abhängt, da mächtigere Maschinenbefehle eine *längere* Ausführungszeit haben, aber nicht berücksichtigt wird, dass bei Verwendung mächtigerer Maschinenbefehle meist *weniger* Instruktionen pro Programm benötigt werden. Dadurch werden Rechner mit primitiven Maschinenbefehlen gegenüber Rechnern mit komplexeren Maschinenbefehlen besser bewertet. Zum anderen stimmen die Aussagen der MIPS-Raten nicht unbedingt mit den korrespondierenden Laufzeiten überein, d.h. beim Vergleich von zwei ausführbaren Programmen A und B auf demselben Rechner X kann Programm B eine höhere MIPS-Rate als A erreichen, also schneller erscheinen, und trotzdem eine höhere Laufzeit als A haben. Dies wird durch folgendes Beispiel verdeutlicht.

Beispiel: Wir betrachten einen Rechner X mit drei Instruktionsklassen $\mathcal{I}_1, \mathcal{I}_2, \mathcal{I}_3$. Jede Instruktion der Klasse $\mathcal{I}_1, \mathcal{I}_2, \mathcal{I}_3$ braucht 1, 2 bzw. 3 Maschinenzyklen zur Ausführung. Der Rechner X habe eine Taktrate von 2 GHz, also eine Zykluszeit von 0.5 ns. Die Übersetzung eines Programmes führt etwa durch Verwendung zweier verschiedener Compiler zu zwei ausführbaren

Programmen A_1 und A_2 , die die in folgender Tabelle angegebenen Anzahlen von Instruktionsausführungen der verschiedenen Instruktionsklassen verursachen.

	$\mathcal{I}_1$	$\mathcal{I}_2$	$\mathcal{I}_3$
A_1	$5 \cdot 10^9$	$1 \cdot 10^9$	$1 \cdot 10^9$
A_2	$10 \cdot 10^9$	$1 \cdot 10^9$	$1 \cdot 10^9$

Für die CPU-Zeit von A_j ergibt sich anhand (4.2) und (4.3) die Formel

$$T_{U\_CPU}(A_j) = \sum_{i=1}^3 n_i(A_j) \cdot CPI_i(A_j) \cdot t_{cycle},$$

wobei $n_i(A_j)$ die in der Tabelle angegebenen Anzahlen der Instruktionsauswertungen und $CPI_i(A_j)$ die für die Instruktionen der Klasse I_i gebrauchten Maschinenzyklen sind, $i = 1, 2, 3$, $j = 1, 2$. Damit hat Programm A_1 eine Laufzeit von 5 s und Programm A_2 eine Laufzeit von 7.5 s. Die MIPS-Raten können anhand von Gleichung (4.4) berechnet werden. Da für Programm A_1 insgesamt $7 \cdot 10^9$ Instruktionen ausgeführt werden, ergibt sich eine MIPS-Rate von 1400 1/s. Für Programm A_2 ergibt sich trotz der höheren Laufzeit eine höhere MIPS-Rate von 1600 1/s. $\square$

Ein weiteres, vor allem im wissenschaftlich-technischen Bereich verwendetes Leistungsmaß ist die für ein Programm A erreichte **MFLOPS-Rate**, wobei MFLOPS für **M**illion **F**loating-**p**oint **O**perations **P**er **S**econd (Millionen Floating-Point-Operationen pro Sekunde) steht. Die MFLOPS-Rate eines Programmes A ist definiert durch

$$MFLOPS(A) = \frac{n_{flp\_op}(A)}{T_{U\_CPU}(A) \cdot 10^6} [1/s], \quad (4.5)$$

wobei $n_{flp\_op}(A)$ die Anzahl der von A ausgeführten Floating-Point-Operationen ist, d.h. es wird nicht die Anzahl der Maschinenbefehle zugrundegelegt, wie dies bei der MIPS-Rate der Fall war, sondern die Anzahl der durch die Instruktionen ausgeführten *Operationen*. Dadurch führt eine unterschiedliche Mächtigkeit der Maschinenbefehle nicht zu einer Verzerrung des Leistungsmaßes. Ein Nachteil der MFLOPS-Raten als Leistungsmaß besteht jedoch darin, dass nicht zwischen den ausgeführten Floating-Point-Operationen unterschieden wird, d.h. teure Operationen wie z.B. Division und Wurzelbildung werden genauso gezählt wie billige Operationen wie z.B. Addition und Multiplikation.

4.1.3 Leistung von Prozessoren mit Cachespeichern

Die Benutzer-CPU-Zeit eines Programmes A wird nach Gleichung (4.1) als Produkt der Anzahl der CPU-Zyklen $n_{cycles}(A)$ und der Zykluszeit des Prozessors t_{cycle} angegeben. Dieser Zusammenhang kann bei Berücksichtigung der Zugriffszeiten auf das Speichersystem verfeinert werden zu

$$T_{U\_CPU}(A) = (n_{cycles}(A) + n_{mm\_cycles}(A)) \cdot t_{cycle}, \quad (4.6)$$

wobei $n_{mm\_cycles}(A)$ die Anzahl der zusätzlichen Maschinenzyklen angibt, die durch solche Speicherzugriffe verursacht werden, die zu einem Nachladen des Caches führen (Cache-Fehlzugriff, engl. *cache miss*), vgl. Abschnitt 2.7. Da Cache-Fehlzugriffe (engl. *cache hit*) keine zusätzlichen Maschinenzyklen verursachen, werden sie durch die bisherige Größe $n_{cycles}(A)$ erfasst. Cache-Fehlzugriffe können durch Lese-Fehlzugriffe (engl. *read miss*) oder Schreib-Fehlzugriffe (engl. *write miss*) verursacht werden, d.h. es ist

$$n_{mm\_cycles}(A) = n_{read\_cycles}(A) + n_{write\_cycles}(A).$$

Dabei gilt z.B.

$$n_{read\_cycles}(A) = n_{read\_op}(A) \cdot r_{read\_miss}(A) \cdot n_{miss\_cycle},$$

wobei $n_{read\_op}(A)$ die Anzahl der Leseoperationen von A und $r_{read\_miss}(A)$ die Lese-Fehlzugriffsrate (engl. *cache miss rate*) von A bezeichnen. n_{miss_cycle} ist die Anzahl der Zyklen, die für das Nachladen des Caches beim Auftreten eines Lese-Fehlzugriffs gebraucht wird (engl. *read miss penalty cycles*). Für $n_{write\_cycles}(A)$ kann ein ähnlicher Ausdruck angegeben werden. Werden Lese- und Schreib-Fehlzugriffe vereinfachend zusammengefasst, ergibt sich also aus (4.6):

$$T_{U\_CPU}(A) = n_{instr}(A) \cdot (CPI(A) + n_{rw\_op}(A) \cdot r_{miss}(A) \cdot n_{miss\_cycle}) \cdot t_{cycle} \quad (4.7)$$

mit der Anzahl $n_{rw\_op}(A)$ der Lese- und Schreiboperationen, der Lese- und Schreib-Fehlzugriffsrate $r_{miss}(A)$ und der Anzahl der Zyklen pro Nachladeoperation n_{miss_cycle} .

Beispiel: Als Beispiel betrachten wir einen Prozessor, für den das Ausführen jeder Instruktion zwei Maschinenzyklen dauert (d.h. es ist $CPI = 2$). Der Prozessor verwendet einen Cache, für den das Nachladen eines Blockes aus dem Hauptspeicher 50 Maschinenzyklen benötigt [75]. Für ein Programm A mit einer Lese- und Schreib-Fehlzugriffsrate von 2% und einer durchschnittlichen Speicherzugriffsrate von 1.33 pro Instruktion, also $n_{rw\_op}(A) = n_{instr}(A) \cdot 1.33$ ergibt sich nach (4.7) :

$$\begin{aligned} T_{U\_CPU}(A) &= n_{instr}(A) (2 + 1.33 \cdot 0.02 \cdot 50) \cdot t_{cycle} \\ &= n_{instr}(A) \cdot 3.33 \cdot t_{cycle} . \end{aligned}$$

Dies kann so interpretiert werden, dass die auftretenden Cache-Fehlzugriffe den "idealen" CPI-Wert von 2 auf den "realen" CPI-Wert von 3.33 erhöhen. Ohne Cache würde aber jeder Speicherzugriff eine Ladeoperation aus dem Hauptspeicher erfordern. Wenn jede Ladeoperation 50 Maschinenzyklen dauert, ergäbe sich so ein CPI-Wert von $2 + 50 \cdot 1.33 = 68.5$.

Bei Verwendung eines doppelt so schnellen Prozessors ohne Änderung des Speichersystems erhöht sich die Zeit für das Nachladen eines Blockes aus dem Hauptspeicher auf 100 Maschinenzyklen. Damit ergibt sich ein CPI-Wert von

$2 + 1.33 \cdot 0.02 \cdot 100 = 4.66$. Wenn t_{cycle} wieder die Zykluszeit des ursprünglichen Prozessors bezeichnet und wenn wir diese Zykluszeit als Zeiteinheit zugrundelegen, ergibt sich für die CPU-Zeit auf dem zweiten Prozessor

$$\tilde{T}_{U\_CPU}(A) = n_{cycles}(A) \cdot 4.66 \cdot t_{cycle}/2,$$

d.h. pro Instruktion werden 2.33 Zeiteinheiten statt 3.33 Zeiteinheiten gebraucht. Die Verdoppelung der Zykluszeit führt also nur zu einer Verringerung der Ausführungszeit des Programmes auf $2.33/3.33 \equiv 70\%$. $\square$

Die Bewertung von Speicherhierarchien kann durch die **mittlere Speicherzugriffszeit** (engl. *average memory access time*) erfolgen [75]. Für die mittlere Lesezugriffszeit $t_{read\_access}(A)$ eines Programmes A gilt:

$$t_{read\_access}(A) = t_{read\_hit} + r_{read\_miss}(A) \cdot t_{read\_miss},$$

wobei t_{read_hit} die Zeit bezeichnet, die für einen Lesezugriff auf den Cache benötigt wird. Der zusätzliche Zeitaufwand für den Zugriff auf den Speicher bei Cache-Fehlzugriffen wird durch das Produkt aus Cache-Fehlzugriffsrate $r_{read\_miss}(A)$ des Programms A und der Zeit für das Nachladen bei Cache-Fehlzugriffen (engl. *read miss penalty time*) t_{read_miss} modelliert. In Formel (4.7) wurde t_{read_miss} aus n_{miss_cycle} und t_{cycle} berechnet. Die Zeit t_{read_hit} für den Lesezugriff auf den Cache war in der Zeit für das Ausführen einer Instruktion enthalten.

Die Zugriffszeit auf den Cache sollte an die Verarbeitungsgeschwindigkeit des Prozessors angepasst werden, da der Prozessor sonst bei jedem Speicherzugriff verzögert wird. Um den Cache (Cache erster Stufe oder L1-Cache) dazu klein genug halten zu können, wird ein zweiter Cache (Cache zweiter Stufe oder L2-Cache) eingesetzt, der groß genug ist, die meisten Zugriffe zu erfüllen. Bei der Leistungsanalyse werden für die mittlere Lesezugriffszeit die Werte des L1-Caches verwendet, d.h. es gilt:

$$t_{read\_access} = t_{read\_hit}^{(L1)} + r_{read\_miss}^{(L1)}(A) \cdot t_{read\_miss}^{(L1)}.$$

Zur Modellierung der Nachladezeit des L1-Caches werden die Zugriffszeit und die Cache-Fehlzugriffsrate des L2-Caches einbezogen. Es gilt

$$t_{read\_miss}^{L1} = t_{read\_hit}^{(L2)} + r_{read\_miss}^{(L2)}(A) \cdot t_{read\_miss}^{(L2)}$$

Hierbei ist die lokale Cache-Fehlzugriffsrate $r_{read\_miss}^{(L1)}(A)$ der Quotient aus der Anzahl der Cache-Fehlzugriffe bzgl. des L1-Caches und der Gesamtanzahl der Lesezugriffe und die lokale Cache-Fehlzugriffsrate $r_{read\_miss}^{(L2)}(A)$ ist der Quotient aus der Anzahl der Cache-Fehlzugriffe bzgl. des L2-Caches und der Anzahl der Cache-Fehlzugriffe bzgl. des L1-Caches. Als eine globale Cache-Fehlzugriffsrate kann man also $r_{read\_miss}^{(L1)}(A) \cdot r_{read\_miss}^{(L2)}(A)$ ansehen.

4.1.4 Benchmarkprogramme

Die Leistung eines Rechners kann stark vom betrachteten Programm abhängen. Für zwei Programme A und B kann der Fall auftreten, dass Programm

A auf einem Rechner X schneller läuft als auf Rechner Y, während B auf Y schneller läuft als auf X. Daher ist es für einen Benutzer wichtig, bei der Auswahl eines Rechners dessen Leistung für die Menge von Programmen zugrunde zu legen, die der Benutzer ausführen will. Idealerweise werden die Programme dabei entsprechend ihrer Ausführungshäufigkeit und Laufzeit gewichtet. Da die auszuführenden Programme oft nicht a priori bekannt sind, wurden **Benchmarkprogramme** entwickelt. Dies sind Testprogramme mit speziellen Charakteristika, deren Ausführungszeit auf verschiedenen Rechnersystemen gemessen werden können, um so eine *standardisierte* Leistungsbewertung zu ermöglichen. Dabei werden folgende Klassen von Benchmarkprogrammen verwendet, die wir nun in der Reihenfolge steigender Aussagefähigkeit auflisten [75].

- **Synthetische Benchmarks** sind (in der Regel kleine) Programme, die aus einer Mischung von Programmanweisungen bestehen, die als repräsentativ für eine große Klasse von realen Programmen angesehen werden. Synthetische Benchmarks führen üblicherweise keine sinnvollen Berechnungen durch, was die Gefahr in sich birgt, dass einige Programmteile durch Compileroptimierungen als redundant erkannt und entfernt werden. Beispiele für synthetische Benchmarks sind *Whetstone* [32, 35], das in FORTRAN vorliegt und die Leistung von Floating-Point-Berechnungen misst, und *Dhrystone* [163], das in C vorliegt und die Integerleistung misst. Die mit den Whetstone- und Dhrystone-Benchmarks ermittelte Leistung wird in der speziellen Einheit *KWhetstone/s* bzw. *KDhrystone/s* angegeben. Der größte Nachteil von synthetischen Benchmarks liegt darin, dass sie das Verhalten von komplexen Programmen nur unvollständig wiedergeben können, da für letztere eine komplexe Wechselwirkung zwischen dem Verhalten des Prozessors und dem des Speichersystems auftreten kann, die die Laufzeit wesentlich beeinflusst, aber durch synthetische Benchmarks nicht erfasst werden kann.
- **Spielzeug-Benchmarks** sind einzelne kleine, aber vollständige Programme, die eine sinnvolle Berechnung ausführen, deren algorithmische Struktur im Vergleich zu realen Programmen aber sehr einfach ist. Beispiele sind Quicksort zum Sortieren von Zahlen oder das Sieb des Erathostenes zum Test auf Primzahl. Der Nachteil von Spielzeug-Benchmarks liegt wie bei synthetischen Benchmarks darin, dass sie das Verhalten von großen Anwendungsprogrammen nur unvollständig widerspiegeln.
- **Programmkern** (*Kernels*) sind relevante Teile von kompletten größeren Programmen, deren Ausführung einen Großteil der Rechenzeit dieser Programme ausmacht. Im Vergleich zu den kompletten Programmen haben Kernels den Vorteil, dass ihr Quelltext oft nur einen Bruchteil des kompletten Quelltextes darstellt. Beispiele für Kernel-Sammlungen sind die *Livermore Loops* (Livermore FORTRAN Kernels, LFK) [108, 45], die 24 aus wissenschaftlich-technischen Anwendungen extrahierte Schleifen enthalten, und *Linpack* [38], das aus einem Teil einer FORTRAN-Bibliothek

mit Verfahren der linearen Algebra besteht. Beide Benchmarks geben die Leistung in MFLOPS an. Der Nachteil von Kernels besteht darin, dass sie zwar für wissenschaftlich-technische Anwendungen einigermaßen aussagekräftig sind, da diese oft eine regelmäßige Struktur aufweisen, aber für Programme aus anderen Bereichen oft zu hohe Leistungswerte angeben.

- **Benchmarks aus kompletten Programmen** fassen zum Teil mehrere für den durchschnittlichen Benutzer relevante Programme zusammen. Dies hat den wesentlichen Vorteil, dass alle Aspekte dieser Programme für die Bewertung berücksichtigt werden. Die Aussagekraft für einen bestimmten Benutzer beruht auf der Annahme, dass die Auswahl der Programme der Benchmarksammlung die Rechnernutzung des Benutzers hinreichend genau beschreibt. Da diese Klasse von Benchmarks die Leistung eines Rechners am besten beschreibt, gehen wir im Folgenden kurz auf ein Beispiel aus dieser Klasse, die SPEC-Benchmarks, ein.

Die am weitesten verbreitete Benchmark-Sammlung sind die **SPEC-Benchmarks**, die von der SPEC-Vereinigung (SPEC: *S*ystem *P*erformance *E*valuation *C*ooperative) verwaltet und aktualisiert werden. Die SPEC-Vereinigung wurde 1988 gemeinsam von HP, DEC, MIPS und Sun mit dem Ziel gegründet, ein standardisiertes Messverfahren für die Leistung von Rechnersystemen zu entwerfen, so dass deren Leistung einfach miteinander verglichen werden kann. Die erste Benchmark-Sammlung, die von SPEC herausgegeben wurde, war SPEC89, es folgten SPEC92, SPEC95, SPEC00 und die zur Zeit aktuelle Sammlung SPEC06. SPEC verwendet die Bezeichnungen CPUxx für SPECxx. Aktuelle Informationen erhält man über www.spec.org.

Die SPEC-Benchmarkprogramme werden dazu verwendet, zwei Leistungsmaße CINT2006 und CFP2006 zu bestimmen, die die mittlere Integer- bzw. Floating-Point-Leistung des getesteten Rechners angeben. Die aktuelle Version SPEC CPU 2006 besteht aus 12 Programmen zur Bestimmung der Integer-Leistung, von denen neun in C und drei in C++ geschrieben sind, und 17 Programmen zur Bestimmung der Floating-Point-Leistung, von denen sechs in Fortran, drei in C, vier in C++ und vier gemischt in C und Fortran geschrieben sind. Die Integer-Programme beinhalten z.B. ein Kompressionsprogramm (bzip2), einen C-Compiler (gcc), ein Videokompressionsprogramm und ein Programm zur Übersetzung von XML-Dokumente in HTML. Die Floating-Point-Programme beinhalten mehrere Simulationsprogramme aus der Physik, aber auch jeweils ein Programm zur Spracherkennung, zum Ray-Tracing computererzeugter Bilder und zur linearen Programmierung.

Die mit Hilfe der SPEC-Benchmarkprogramme ermittelte Leistung wird dabei auf eine Referenzmaschine bezogen. Für SPEC CPU 2006 ist dies eine Sun Ultra Enterprise 2 mit einem 296 MHz UltraSparc II Prozessor. Damit korrespondieren größere CINT2006- und CFP2006-Werte mit einer höheren Leistung des getesteten Rechners. Die CINT2006- und CFP2006-Werte werden getrennt mit Hilfe der Programme zum Test der Integerleistung und der Floating-Point-Leistung ermittelt. Zur Durchführung eines Benchmarktests

und zur Bestimmung der SPEC-Parameter CINT2006- und CFP2006-Werte werden die folgenden drei Schritte ausgeführt:

- a) Jedes der Programme wird mehrfach auf der zu untersuchenden Maschine U ausgeführt. Für jedes der Programme wird eine (mittlere) Ausführungszeit in Sekunden bestimmt.
- b) Die gemessenen Ausführungszeiten werden bzgl. der Referenzmaschine R (Sun Ultra Enterprise 2) normalisiert, indem für jedes Programm die Ausführungszeit auf R durch die Ausführungszeit auf U dividiert und mit 100 multipliziert wird. Damit erhält man für jedes Programm einen Faktor, der angibt, um wie viel schneller das Programm auf U als auf R läuft.
- c) Aus den errechneten Faktoren wird (für Integer und Floating-Point getrennt) ein Gesamtfaktor G durch Berechnung des *geometrischen Mittels* der Einzelfaktoren $E_1, \dots, E_n$ bestimmt. Für n Programme berechnet man

$$G = \sqrt[n]{\prod_{i=1}^n E_i}.$$

Für die Integer-Programme ist $n = 12$, für die Floating-Point-Programme ist $n = 17$. Für die Integer-Programme ist G der gesuchte CINT2006-Wert, für die Floating-Point-Programme ist G der gesuchte CFP2006-Wert.

Eine Alternative zur Verwendung des geometrischen Mittels besteht in der Verwendung des *arithmetischen Mittels* zur Berechnung des Gesamtfaktors

$$G' = \frac{1}{n} \sum_{i=1}^n E_i.$$

Prinzipiell hat die Verwendung des geometrischen Mittels gegenüber dem arithmetischen Mittel jedoch den Vorteil, dass der Vergleich zweier Maschinen unabhängig von der jeweiligen Wahl einer dieser Maschinen als Referenzmaschine ist. Bei Verwendung des arithmetischen Mittels ist dies nicht unbedingt der Fall, wie folgendes Beispiel aus [75] zeigt.

Beispiel: Wir betrachten zwei Programme A und B sowie zwei Rechnersysteme X und Y. Die Ausführungszeit von A sei 1 s auf X und 10 s auf Y, die von B sei 1000 s auf X und 100 s auf Y. Bei Verwendung von X bzw. Y als Referenzmaschine für die jeweils andere Maschine ergeben sich die Einzelfaktoren anhand folgender Tabelle. Die errechneten Gesamtfaktoren unter Verwendung des arithmetischen und geometrischen Mittels sind ebenfalls angegeben.

	Ausführungszeit		Einzelfaktoren (Referenz- maschine Y)		Einzelfaktoren (Referenz- maschine X)	
	X	Y	X	Y	X	Y
Programm A	1 s	10 s	10	1	1	0.1
Programm B	1000 s	100 s	0.1	1	1	10
arithm. Mittel			5.05	1	1	5.05
geom. Mittel			1	1	1	1

Bei Verwendung von X als Referenzmaschine sagt das arithmetische Mittel aus, dass Rechner Y 5.05 mal schneller als Rechner X ist. Bei Verwendung von Y als Referenzmaschine sagt das arithmetische Mittel aus, dass Rechner X 5.05 mal schneller als Rechner Y ist. Dies sind klar widersprüchliche Aussagen. Die Verwendung des geometrischen Mittels sagt in beiden Fällen aus, dass beide Rechner gleichschnell sind.

Der Nachteil der Verwendung des geometrischen Mittels besteht darin, dass der errechnete Wert keine Aussage über die eigentlichen Ausführungszeiten liefert. Im obigen Beispiel braucht die *einmalige* Ausführung von Programm A und B auf X 1001 sec, auf Y 110 sec, d.h. Y ist 9.1 mal schneller als X. Die Aussage, dass beide Rechner gleichschnell sind, ergibt sich, wenn man 100 Ausführungen von A und eine Ausführung von B betrachtet. $\square$

Ausführlicher werden Benchmarkprogramme in [39] oder [81] behandelt, wo sich auch weitere Literatur findet.

4.2 Parallele Leistungsmaße

Ein wesentliches Kriterium für die Güte einer parallelen Implementierung eines Algorithmus ist die Laufzeit auf einem gegebenen Rechner. Die **parallele Laufzeit** $T_p(n)$ eines Programmes ist die Zeit zwischen dem Start der Abarbeitung des parallelen Programmes und der Beendigung der Abarbeitung aller beteiligten Prozessoren und wird meist in Abhängigkeit der Anzahl p der zur Ausführung benutzten Prozessoren und einer Problemgröße n angegeben, die z.B. durch die Größe der Eingabe oder die Anzahl der Gleichungen eines Gleichungssystems gegeben ist. Für Rechner mit physikalisch verteiltem Speicher setzt sich die Laufzeit eines parallelen Programmes zusammen aus:

- der Zeit für die Durchführung von *lokalen Berechnungen*, d.h. für die Durchführung von Berechnungen, die ein Prozessor unter Verwendung von Daten in seinem lokalen Speicher durchführt,
- der Zeit für den *Austausch von Daten* durch Ausführung von Kommunikationsoperationen,
- den *Wartezeiten*, die z.B. wegen ungleicher Verteilung der Last zwischen den Prozessoren entstehen, und
- der Zeit für die *Synchronisation* der ausführenden Prozessoren oder einer Teilmenge der ausführenden Prozessoren.

Diese Aufzählung gibt für die meisten parallelen Programme die Beiträge zur Laufzeit in fallender Reihenfolge des Beitrages zur Gesamtlaufzeit wieder. Für Rechner mit gemeinsamem Speicher setzt sich die parallele Laufzeit ähnlich zusammen mit dem Unterschied, dass die Kommunikationszeiten zum Austausch von Daten durch die Zugriffszeiten auf globale Daten ersetzt werden.

Die *Kosten* eines parallelen Programmes, häufig auch *Arbeit* oder *Prozessor-Zeit-Produkt* genannt, berücksichtigen die Zeit, die *alle* an der Ausführung beteiligten Prozessoren zur Abarbeitung des Programmes verwenden.

Kosten

Die Kosten eines parallelen Programms sind definiert als

$$C_p(n) = T_p(n) \cdot p$$

und sind damit ein Maß für die von allen Prozessoren durchgeführte Arbeit.

Ein paralleles Programm heißt *kostenoptimal*, wenn $C_p(n) = T^*(n)$ gilt, d.h. wenn insgesamt genauso viele Operationen ausgeführt werden wie vom schnellsten sequentiellen Verfahren, das Laufzeit $T^*(n)$ hat. Bei Verwendung asymptotischer Laufzeiten heißt ein paralleles Programm kostenoptimal, falls $T^*(n)/C_p(n) = \Theta(1)$ gilt. Zur Definition der Θ -Notation siehe auch Abschnitt 4.3.1.

Zur Laufzeitanalyse paralleler Programme ist insbesondere ein Vergleich mit einer sequentiellen Implementierung von Interesse, um den Nutzen des Einsatzes der Parallelverarbeitung abschätzen zu können. Für einen solchen Vergleich wird oft der **Speedup**-Begriff als Maß für den relativen Geschwindigkeitsgewinn herangezogen.

Speedup

Der Speedup $S_p(n)$ eines parallelen Programmes mit Laufzeit $T_p(n)$ ist definiert als

$$S_p(n) = \frac{T^*(n)}{T_p(n)},$$

wobei p die Anzahl der Prozessoren zur Lösung des Problems der Größe n bezeichnet. $T^*(n)$ ist die Laufzeit einer optimalen sequentiellen Implementierung zur Lösung desselben Problems.

Der Speedup einer parallelen Implementierung gibt also den relativen Geschwindigkeitsvorteil an, der gegenüber der besten sequentiellen Implementierung durch den Einsatz von Parallelverarbeitung auf p Prozessoren entsteht. Verwendet wird der Speedup-Begriff sowohl für die theoretische Analyse eines Algorithmus mit O -Notation als auch für die Bewertung eines parallelen Programmes.

Theoretisch gilt immer $S_p(n) \leq p$. Denn wäre $S_p(n) > p$, könnte man einen sequentiellen Algorithmus konstruieren, der schneller als der für die Berechnung des Speedups verwendete sequentielle Algorithmus ist. Dieser neue sequentielle Algorithmus wird aus dem parallelen Algorithmus dadurch abgeleitet, dass die Arbeitsschritte der Prozessoren reihum simuliert werden,

d.h. in den ersten p Schritten führt der neue sequentielle Algorithmus den ersten Schritt der p Prozessoren in einer festgelegten Reihenfolge aus, in den nächsten p Schritten wird der zweite Schritt der verschiedenen Prozessoren ausgeführt, usw. Damit führt dieser neue Algorithmus p Mal so viele Schritte wie der parallele Algorithmus aus, d.h. wegen $S_p(n) > p$ hat dieser Algorithmus eine Laufzeit

$$p \cdot T_p(n) = p \cdot \frac{T^*(n)}{S_p(n)} < T^*(n),$$

wodurch ein Widerspruch zu der Annahme entsteht, dass der zur Berechnung des Speedups benutzte sequentielle Algorithmus der schnellste ist.

Mit der oben festgelegten Definition des Speedups können sich in der Praxis Probleme ergeben, da ein Vergleich mit dem schnellsten sequentiellen Algorithmus oft schwierig ist. Mögliche Gründe dafür sind:

- Unter Umständen ist der beste sequentielle Algorithmus nicht bekannt, d.h. es konnte für das gegebene Problem zwar eine untere Schranke für die Laufzeit eines Lösungsverfahrens bestimmt werden, es ist aber bisher noch kein Algorithmus gefunden worden, dessen asymptotische Laufzeit der unteren Schranke entspricht.
- Es gibt zwar einen Algorithmus, dessen asymptotische Laufzeit optimal ist, in der Praxis führen aber andere Algorithmen in Abhängigkeit von der Größe der Eingabe oder bestimmten Charakteristika der Eingabe zu geringeren Laufzeiten. So lohnt sich etwa der Einsatz von balancierten Bäumen zur dynamischen Verwaltung von Datenmengen erst ab einer gewissen Größe der Datenmenge und entsprechend vielen Zugriffsoperationen.
- Es ist zwar bekannt, welcher sequentielle Algorithmus für welche Eingabegrößen zu der geringsten Laufzeit führt, dieser Algorithmus ist aber evtl. nur mit großem Aufwand zu implementieren.

Aus diesen Gründen wird für die Berechnung des Speedups in der Praxis oft statt des schnellsten sequentiellen Algorithmus eine sequentielle Implementierung des Algorithmus verwendet, auf dem die parallele Implementierung beruht.

In der Praxis kann *superlinearer Speedup* auftreten, d.h. der Fall $S_p(n) > p$ ist *nicht* ausgeschlossen. Der Grund dafür liegt meist in Cacheeffekten: Bei vielen parallelen Berechnungen erhält jeder Prozessor einen Teil einer (globalen) Datenstruktur, auf dem er iterativ Berechnungen durchführt, evtl. verbunden mit einem Austausch von Randelementen. Dabei kann der Fall auftreten, dass bei Verwendung eines einzelnen Prozessors die Datenstruktur nicht in dessen Cache passt, so dass bei jeder Iteration ein erneutes Nachladen aus dem Hauptspeicher erforderlich ist, während beim Einsatz mehrerer Prozessoren der einem einzelnen Prozessor zugewiesene Teil der Datenstruktur ganz in dessen Cache passt, so dass ein teures Nachladen entfällt.

Tatsächlich wird in den meisten Fällen nicht einmal ein *linearer Speedup*, d.h. $S_p(n) = p$, erreicht, da für die parallele Implementierung zusätzlicher

Aufwand zur Verwaltung der Parallelität eingesetzt werden muss, insbesondere zum Austausch von Daten durch das Verschicken von Nachrichten, zur Synchronisation der Prozessoren, für Wartezeiten von Prozessoren wegen ungleicher Verteilung der Last oder für im Vergleich zum sequentiellen Algorithmus zusätzlich durchzuführende Berechnungen. Eine große Rolle spielen auch Berechnungen, die aufgrund von Datenabhängigkeiten sequentiell ausgeführt werden müssen, die also nur von einem der p Prozessoren bearbeitet werden können, während die anderen warten. Ein typisches Beispiel sind Eingabe- und Ausgabeoperationen.

Alternativ zum Speedup kann der Begriff der **Effizienz** eines parallelen Programmes benutzt werden, der ein Maß für den Anteil der Laufzeit ist, den ein Prozessor für Berechnungen benötigt, die auch im sequentiellen Programm vorhanden sind. Die Definition bezieht die Kosten des parallelen Programmes ein und ist folgendermaßen gegeben:

Effizienz

Die Effizienz eines parallelen Programms ist definiert als

$$E_p(n) = \frac{T^*(n)}{C_p(n)} = \frac{S_p(n)}{p} = \frac{T^*(n)}{p \cdot T_p(n)}$$

wobei $T^*(n)$ die Laufzeit des besten sequentiellen Algorithmus und $T_p(n)$ die parallele Laufzeit ist.

Liegt kein superlinearer Speedup vor, gilt $E_p(n) \leq 1$. Der ideale Speedup $S_p(n) = p$ entspricht einer Effizienz $E_p(n) = 1$.

Die mögliche Verringerung von Laufzeiten durch eine Parallelisierung sind i.A. begrenzt. So stellt etwa die Anzahl der Prozessoren die theoretisch obere Schranke des Speedups dar. Weitere Begrenzungen liegen im zu parallelisierenden Algorithmus selber begründet, der neben parallelisierbaren Anteilen wegen Datenabhängigkeiten auch inhärent sequentielle Anteile enthalten kann. Der Effekt von Programmteilen, die sequentiell ausgeführt werden müssen, auf den erreichbaren Speedup wird durch das **Amdahlsche Gesetz** quantitativ erfasst [12].

Amdahlsches Gesetz

Wenn bei einer parallelen Implementierung ein (konstanter) Bruchteil f ($0 \leq f \leq 1$) sequentiell ausgeführt werden muss, setzt sich die Laufzeit der parallelen Implementierung aus der Laufzeit $f \cdot T^*(n)$ des sequentiellen Teils und der Laufzeit des parallelen Teils, die mindestens $(1 - f)/p \cdot T^*(n)$ beträgt, zusammen. Für den erreichbaren Speedup gilt damit

$$S_p(n) = \frac{T^*(n)}{f \cdot T^*(n) + \frac{1-f}{p} T^*(n)} = \frac{1}{f + \frac{1-f}{p}} \leq \frac{1}{f}.$$

Bei dieser Berechnung wurde der beste sequentielle Algorithmus verwendet und es wurde angenommen, dass sich der parallel ausführbare Teil perfekt

parallelisieren lässt. Durch ein einfaches Beispiel sieht man, dass nicht parallelisierbare Berechnungsteile einen großen Einfluss auf den erreichbaren Speedup haben. Wenn z.B. 20% eines Programmes sequentiell abgearbeitet werden müssen, beträgt nach Aussage des Amdahlschen Gesetzes der maximal erreichbare Speedup 5, egal wie viele Prozessoren eingesetzt werden. Nicht parallelisierbare Teile müssen insbesondere bei einer großen Anzahl von Prozessoren besonders beachtet werden.

Das Verhalten der Leistung eines parallelen Programmes bei steigender Prozessoranzahl wird durch die **Skalierbarkeit** (engl. *scalability*) erfasst. Die Skalierbarkeit eines parallelen Programmes auf einem gegebenen Parallelrechner ist ein Maß für die Eigenschaft, einen Leistungsgewinn proportional zur Anzahl p der verwendeten Prozessoren zu erreichen. Der Begriff der Skalierbarkeit wird in unterschiedlicher Weise präzisiert, z.B. durch Einbeziehung der Problemgröße n . Eine häufig beobachtete Eigenschaft paralleler Algorithmen ist es, dass für festes n und steigendes p eine Sättigung des Speedups eintritt, dass aber für festes p und steigende Problemgröße n ein höherer Speedup erzielt wird. In diesem Sinne bedeutet Skalierbarkeit, dass die Effizienz eines parallelen Programmes bei gleichzeitigem Ansteigen von Prozessoranzahl p und Problemgröße n konstant gehalten wird. Da es oft ein Ziel der Parallelverarbeitung ist, ein gegebenes Problem nicht schneller, sondern ein größeres Problem in der gleichen Zeit auszuführen, die bisher für das kleinere Problem benötigt wurde, beschreibt diese letzte Definition der Skalierbarkeit eine sinnvolle Eigenschaft.

Das Verhalten des Speedups bei steigender Problemgröße n kann durch das Amdahlsche Gesetz *nicht* erfasst werden. Eine Variante des Amdahlschen Gesetzes ergibt sich, wenn man annimmt, dass der nicht parallelisierbare Teil eines parallelen Programmes nicht einen konstanten Anteil der Gesamtbeurteilung ausmacht, sondern dass dieser Anteil mit der Eingabegröße abnimmt. In diesem Fall kann für jede Anzahl von Prozessoren der gewünschte Speedup $\leq p$ dadurch erreicht werden, dass die Problemgröße entsprechend hoch gesetzt wird.

Formal wird diese Beobachtung durch das **Gustafson-Gesetz** beschrieben, und zwar für den Spezialfall, dass der nicht parallelisierbare Teil eine von der Problemgröße unabhängige *konstante* Laufzeit hat [68]. Bezeichnet τ_f die konstante Zeit für die Ausführung des nicht parallelisierbaren Teils eines parallelen Programms und $\tau_v(n, p)$ die Zeit für die Ausführung des parallelisierbaren Anteils bei Problemgröße n mit p Prozessoren, so ist der *Skalierte Speedup* des Programms durch

$$S_p(n) = \frac{\tau_f + \tau_v(n, 1)}{\tau_f + \tau_v(n, p)}$$

gegeben. Nehmen wir an, dass sich der parallelisierbare Anteil perfekt parallelisieren lässt, so gilt $\tau_v(n, 1) = T^*(1) - \tau_f$ und $\tau_v(n, p) = (T^*(n) - \tau_f)/p$. Für den Speedup folgt

$$S_p(n) = \frac{\tau_f + T^*(n) - \tau_f}{\tau_f + (T^*(n) - \tau_f)/p} = \frac{\frac{\tau_f}{T^*(n) - \tau_f} + 1}{\frac{\tau_f}{T^*(n) - \tau_f} + \frac{1}{p}},$$

und damit

$$\lim_{n \rightarrow \infty} S_p(n) = p,$$

falls $T^*(n)$ streng monoton in n steigt. Dies gilt z.B. für den Fall, dass $\tau_v(n, p) = n^2/p$, was für viele Iterationsverfahren auf zweidimensionalen Gittern gilt:

$$\lim_{n \rightarrow \infty} S_p(n) = \lim_{n \rightarrow \infty} \frac{\tau_f + n^2}{\tau_f + n^2/p} = \lim_{n \rightarrow \infty} \frac{\tau_f/n^2 + 1}{\tau_f/n^2 + 1/p} = p$$

Weitergehend sind Skalierbarkeitsaussagen, die angeben, *wie* die Problemgröße n relativ zur Prozessoranzahl p wachsen muss, um eine konstante Effizienz zu erhalten. Ein Ansatz sind die in [65] eingeführten *Isoeffizienzfunktionen*, die die benötigten Änderungen von n als Funktion von p ausdrücken.

4.3 Modellierung von Laufzeiten

In diesem Abschnitt beschäftigen wir uns mit der theoretischen Laufzeitanalyse paralleler Programme. Da verschiedene parallele Programmvarianten eines Algorithmus, die sich z.B. in der Verteilung der Daten und der Berechnungen und damit auch in den durchzuführenden Kommunikationsoperationen unterscheiden, unterschiedliche parallele Laufzeiten haben können, kann durch eine derartige Vorabanalyse eine günstige Programmvariante mit möglichst geringer Ausführungszeit bestimmt werden. In vielen Fällen reicht ein grober Vergleich aus, um eine Implementierung gegenüber einer anderen zu favorisieren. Oft führen die Prozessoren in verschiedenen parallelen Implementierungen sehr ähnliche oder sogar identische Berechnungen aus, so dass der wesentliche Unterschied in den auszuführenden Kommunikationsoperationen liegt.

Für Rechner mit verteiltem Speicher ist ein solcher Vergleich ohne größere Schwierigkeiten durchführbar, da die Kommunikations- und Synchronisationsoperationen *explizit* im Programm spezifiziert sind. Für Rechner mit gemeinsamem Speicher ist ein solcher Vergleich dagegen wesentlich schwieriger, da die durchgeführten Speicherzugriffe je nach Speicherbelegung unterschiedlich lange dauern, ohne dass dies am Programm erkennbar wäre, da globale Speicherzugriffe als normale Variablenzugriffe im Programm dargestellt sind. Im Folgenden betrachten wir Rechner mit verteiltem Speicher.

Die *Zeit für die Durchführung von lokalen Berechnungen* kann für regelmäßige Anwendungen grob durch die Anzahl der durchzuführenden Operationen abgeschätzt werden. Dabei sind folgende Quellen von Ungenauigkeiten zu beachten:

- Die genaue Anzahl der arithmetischen Operationen kann nicht bestimmt werden, da zur Übersetzungszeit die Grenzen bestimmter Schleifen nicht bekannt sind bzw. da für bestimmte bedingte Anweisungen nicht bekannt ist, welcher Teil mit welcher Häufigkeit betreten wird. Zur Abhilfe werden in der Praxis verschiedene Ansätze verfolgt. Hilfreich sind zusätzliche Angaben des Programmierers, die z.B. in Form von Pragma-Anweisungen für Schleifen die geschätzte Anzahl von Durchläufen und für bedingte Anweisungen die geschätzte Häufigkeit für das Eintreten der Bedingung angeben. Eine andere Möglichkeit besteht im Einsatz von Profilingwerkzeugen, mit deren Hilfe für ähnliche (kleinere) Datensätze typische Werte für die Anzahl der Schleifendurchläufe und die Häufigkeit für das Eintreten der Bedingung bei bedingten Anweisungen erhalten werden können. Aus diesen Angaben für verschiedene Datensätze mit unterschiedlicher Größe können die Werte für größere Datensätze extrapoliert werden.
- Unterschiedliche arithmetische Operationen haben in Abhängigkeit von ihrer Realisierung eine unterschiedliche Ausführungszeit. Größere Unterschiede ergeben sich hier vor allem zwischen Operationen, die direkt in Hardware realisiert sind (wie dies meist für Floating-Point-Addition und -Multiplikation der Fall ist) und Operationen, die durch eine Software-routine realisiert sind (wie dies z.T. für die Division, trigonometrische Funktionen oder Exponentialfunktionen der Fall ist). Als Abhilfe kann eine genauere Differenzierung der ausgeführten Operationen erfolgen, so dass sich die gesamte Berechnungszeit als gewichtete Summe über die einzelnen Operationen ergibt.
- Jeder einzelne Prozessor eines Rechners mit verteiltem Speicher hat üblicherweise eine Speicherhierarchie mit Caches, die evtl. in mehreren Stufen angeordnet sind. Dies führt dazu, dass die Zugriffszeiten auf den lokalen Speicher insbesondere bei unregelmäßigen Anwendungen schwierig zu bestimmen sind. Bei Anwendungen mit regelmäßigem Speicherzugriffsmuster kann oft mit durchschnittlichen Speicherzugriffszeiten gearbeitet werden, ohne das Resultat allzusehr zu verfälschen, vgl. Abschnitt 4.1. Bei anderen Anwendungen muss ein genaueres Modell der Speicherhierarchie verwendet werden.

Die *Zeit für den expliziten Austausch von Daten* zwischen den Prozessoren kann anhand der im Programm enthaltenen Kommunikationsanweisungen abgeschätzt werden, indem für jede Kommunikationsanweisung eine Laufzeitformel genutzt wird. Für eine theoretische Analyse können Formeln für die Beschreibung der asymptotischen Laufzeiten eingesetzt werden, wie wir sie im Folgenden für verschiedene Verbindungsnetzwerke beschreiben werden. In der Praxis werden auch empirisch ermittelte Laufzeitformeln verwendet, die die Laufzeit in Abhängigkeit von der Nachrichtengröße und der Prozessoranzahl darstellen und deren Gestalt vom verwendeten Parallelrechner, der verwendeten Kommunikationsbibliothek und deren Implementierung abhängen.

4.3.1 Realisierung von Kommunikationsoperationen

In diesem Abschnitt betrachten wir die Realisierung der in Abschnitt 3.7.2 vorgestellten globalen Kommunikationsoperationen auf verschiedenen statischen Verbindungsnetzwerken, und zwar dem linearen Feld, dem Ring, dem symmetrischen Gitter, dem Hyperwürfel und dem binären Baum, die in Abschnitt 2.5.1 eingeführt wurden. Durch eine Analyse der Realisierungen globaler Kommunikationsoperationen können Aussagen über die zu erwartenden Laufzeiten in Abhängigkeit von der Prozessoranzahl p und der Nachrichtengröße m gemacht werden. Das Ziel besteht dabei im Ableiten von *asymptotischen Aussagen* für die Übertragungszeit. Unsere Darstellung folgt [14] und wird asymptotische Laufzeiten in der Gesamtanzahl p der Prozessoren eines Netzwerkes angeben, also von der Nachrichtengröße abstrahieren. Für die Analyse machen wir die folgenden Annahmen:

1. Die Kanten der Netzwerke sind **bidirektional**, d.h. auf einer Kante kann zu jedem Zeitpunkt in jede Richtung eine Nachricht unterwegs sein. Bei Verbindungsnetzwerken für Parallelrechner sind die Verbindungskanten fast immer bidirektional realisiert.
2. Jeder Knoten kann auf allen auslaufenden Kanten **simultan senden** (engl. *all-port communication*). Dies kann in der Praxis dadurch realisiert werden, dass für jede auslaufende Kante ein **Ausgangspuffer** mit separatem Controller existiert, der das Verschicken der auslaufenden Nachrichten über die zugehörige Kante kontrolliert. Das simultane Senden wird dann dadurch realisiert, dass die Controller für die verschiedenen Kanten parallel zueinander arbeiten.
3. Jeder Knoten kann auf allen einlaufenden Kanten **simultan empfangen**. Dies kann in der Praxis dadurch realisiert werden, dass für jede einlaufende Kante ein **Eingangspuffer** mit separatem Controller existiert, der den Empfang der über diese Kante ankommenden Nachrichten kontrolliert.
4. Jede Nachricht besteht aus einer Anzahl von Bytes, die **ohne Unterbrechung** nacheinander über die Verbindungskanten geschickt werden.
5. Die Zeit für das Verschicken einer Nachricht setzt sich aus einer **Startzeit** t_S , die unabhängig von der Länge der Nachricht ist, und einer **Übertragungszeit** $m \cdot t_B$, die proportional zur Länge der zu übertragenden Nachricht m ist, zusammen. t_B ist die Zeit für die Übertragung eines einzelnen Bytes (Bytetransferzeit). Damit braucht die Übertragung einer Nachricht mit m Bytes von einem Knoten eines Netzwerkes zu seinem Nachbarknoten die in Abschnitt 2.6.2 hergeleitete Zeit aus Formel (2.2), d.h. $T(m) = t_S + m \cdot t_B$. Wir nehmen an, dass $T(m)$ für alle Kanten eines Netzwerkes gleich ist.
6. Als Switching-Strategie wird Paket-Switching mit **Store-and-Forward-Routing** verwendet, vgl. Abschnitt 2.6.2. Eine Nachricht muss einen Pfad im Netzwerk durchlaufen, der den Sendeknoten mit dem Zielknoten

verbindet. Die Länge des Pfades bestimmt also die Anzahl der erforderlichen Nachrichtenübertragungen, die wir im Folgenden *Schritte* nennen.

Für ein Netzwerk mit gegebenen Parametern t_S und t_B hängt die Gesamtübertragungszeit von einem Sendeknoten zu einem Empfangsknoten somit von der Länge des Übertragungspfades und der Größe der Nachricht ab. Bei der Realisierung globaler Kommunikationsoperationen sind mehrere Nachrichten im Netzwerk zu berücksichtigen, deren Übertragungspfade für eine effiziente Kommunikation geschickt zu planen sind. Daher muss für eine effiziente Realisierung darauf geachtet werden, dass nicht zum gleichen Zeitpunkt zwei verschiedene Nachrichten dieselbe Kante in derselben Richtung benutzen wollen. Dies würde zu einer Kollision auf der Kante führen und damit die Übertragung einer der beiden Nachrichten verzögern.

Bevor wir die Realisierungen der Kommunikationsoperationen beschreiben, geben wir die Notation für asymptotische Laufzeiten an.

Asymptotischen Aussagen. Asymptotische Laufzeiten beschreiben das Wachstumsverhalten für eine steigende Problemgröße, indem sie von Konstanten und von Termen geringerer Steigung abstrahieren, vgl. z.B. [26]. Angegeben werden asymptotische Laufzeiten in Form von Funktionen, die auf $\mathbb{N}$ definiert sind und die die für das Wachstum wesentlichen Terme erfassen. Benutzt werden die O -Notation, die Ω -Notation und die Θ -Notation, die die folgenden Schranken des Wachstums beschreiben und in Abbildung 4.1 illustriert sind. Die *asymptotische obere Schranke* ist durch die O -Notation gegeben:

$$O(g(n)) = \{f(n) \mid \text{es existiert eine positive Konstante } c \text{ und ein } n_0 \in \mathbb{N}, \\ \text{so dass für alle } n \geq n_0 : 0 \leq f(n) \leq cg(n)\}$$

Die *asymptotische untere Schranke* ist durch die Ω -Notation gegeben:

$$\Omega(g(n)) = \{f(n) \mid \text{es existiert eine positive Konstante } c \text{ und ein } n_0 \in \mathbb{N}, \\ \text{so dass für alle } n \geq n_0 : 0 \leq cg(n) \leq f(n)\}$$

Die zusammenfassenden asymptotische Laufzeit wird schließlich durch die Θ -Notation erfasst:

$$\Theta(g(n)) = \{f(n) \mid \text{es existieren positive Konstanten } c_1, c_2 \text{ und ein } n_0 \in \mathbb{N}, \\ \text{so dass für alle } n \geq n_0 : 0 \leq c_1g(n) \leq f(n) \leq c_2g(n)\}$$

Tabelle 4.1 gibt die asymptotischen Laufzeiten für die im Folgenden beschriebenen Realisierungen globaler Kommunikationsoperationen auf verschiedenen Netzwerken an, vgl. [14]. Weitere Herleitungen von Laufzeitaussagen sind z.B. in [89, 65] zu finden. Die verschiedenen Herleitungen variieren insbesondere in den Annahmen über die Architektur. So betrachtet [65] *one-port communication*, d.h. ein Knoten kann jeweils nur über *eine* Kante senden oder empfangen, und gibt Laufzeitformeln in Form geschlossener Funktionen in Abhängigkeit von der Prozessoranzahl p und der Nachrichtengröße m

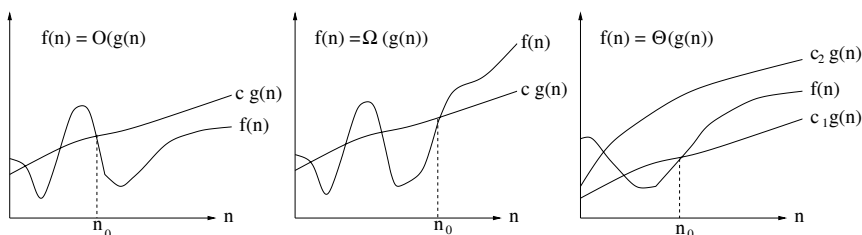


Abb. 4.1. Beispielhafte Illustration der O -, Ω - und Θ -Notation. Die Bezeichnungen entsprechen den gegebenen Definitionen. Das eingezeichnete n_0 ist der minimale Wert, der in der jeweiligen Definition als Wert für n_0 gewählt werden kann.

Tabelle 4.1. Asymptotischer Aufwand für die Realisierung von Kommunikationsoperationen auf verschiedenen Netzwerken in Abhängigkeit von der Prozessoranzahl p . Für ein lineares Feld gelten die gleichen Angaben wie für einen Ring.

Operation	Ring	Baum	Gitter	Hyperwürfel
Einzel-Broadcast	$\theta(p)$	$\theta(\log p)$	$\theta(\sqrt[d]{p})$	$\theta(\log p)$
Scatter	$\theta(p)$	$\theta(p)$	$\theta(p)$	$\theta(p/\log p)$
Multi-Broadcast	$\theta(p)$	$\theta(p)$	$\theta(p)$	$\theta(p/\log p)$
Gesamtaustausch	$\theta(p^2)$	$\theta(p^2)$	$\theta(p^{(d+1)/d})$	$\theta(p)$

an und zwar für die beiden Switching-Strategien Store-und-Forward-Routing und Cut-Through-Switching an.

Für die Analyse nutzen wir die in Abschnitt 3.7.2 hergeleiteten topologieunabhängigen Dualitäts- und Hierarchieeigenschaften zwischen den einzelnen Kommunikationsoperationen, vgl. Abbildung 3.7. Wenn wir für eine Kommunikationsoperation eine Anzahl von Schritten ermittelt haben, wissen wir also, dass aufgrund der Dualitätsrelation die duale Kommunikationsoperation die gleiche Anzahl von Schritten braucht und dadurch realisiert werden kann, dass die Richtung der Nachrichtenübertragungen umgekehrt wird. Eine Spezialisierung einer Kommunikationsoperation braucht zu ihrer Realisierung *höchstens* so viele Schritte wie diese, eine Verallgemeinerung braucht *mindestens* so viele Schritte.

Vollständiger Graph. In einem vollständigen Graphen ist jeder Knoten des Netzwerkes mit jedem anderen durch eine Kante verbunden, auf der gleichzeitiges Senden und Empfangen möglich ist. Ein **Gesamtaustausch** kann also in *einem* Schritt realisiert werden, indem jeder Knoten gleichzeitig alle Nachrichten für die anderen Knoten losschickt. Da ein vollständiger Austausch in einem Schritt realisiert werden kann, können auch alle anderen Kommunikationsoperationen wie *Broadcast*-, *Scatter*- oder *Gatheroperation* in einem Schritt realisiert werden.

Lineares Feld. Ein lineares Feld mit p Knoten kann durch einen Graphen $G = (V, E)$ mit Knotenmenge $V = \{1, \dots, p\}$ und Kantenmenge

$E = \{(i, i + 1) | 1 \leq i < p\}$ beschrieben werden, d.h. jeder Knoten ist mit einem linken und einem rechten Nachbarn durch eine Kante verbunden. Der erste Knoten ist nur mit seinem rechten, der letzte Knoten ist nur mit seinem linken Nachbarn verbunden. Wir betrachten zuerst die Realisierung einer **Einzel-Broadcastoperation**, d.h. ein ausgezeichnete Wurzelknoten stellt eine Nachricht zur Verfügung, die allen anderen Knoten mitgeteilt werden soll. Dies kann dadurch realisiert werden, dass der Wurzelknoten die Nachricht an seinen linken und rechten Nachbarn schickt, und dass ein Knoten eine Nachricht, die er von einem Nachbarn erhält, an den jeweils anderen Nachbarn weitersendet. Die für diese Realisierung notwendige Anzahl von Schritten hängt von der Position des Wurzelknotens im linearen Feld ab. Im besten Fall liegt der Knoten in der Mitte des Feldes, und es werden $\lfloor p/2 \rfloor$ Schritte benötigt. Im schlechtesten Fall liegt der Knoten am Ende des Feldes, und es werden $p - 1$ Schritte benötigt. Da der Durchmesser eines linearen Feldes mit p Knoten $p - 1$ ist, kann eine Einzel-Broadcastoperation auch nicht schneller realisiert werden und es ergibt sich insgesamt die Zeit $\Theta(p)$.

Eine **Multi-Broadcastoperation** kann auf einem linearen Feld in $p - 1$ Schritten realisiert werden. Im ersten Schritt schickt jeder Knoten seine eigene Nachricht an seine beiden Nachbarn, falls diese vorhanden sind. Im k -ten Schritt für $k = 2, \dots, p - 1$ schickt jeder Knoten i mit $k \leq i < p$ die zuvor von seinem linken Nachbarn erhaltene Nachricht von Knoten $i - k + 1$ an seinen rechten Nachbarn $i + 1$ weiter. Gleichzeitig schickt jeder Knoten i mit $2 \leq i \leq p - k + 1$ die zuvor von seinem rechten Nachbarn erhaltene Nachricht von Knoten $i + k - 1$ an seinen linken Nachbarn $i - 1$ weiter. Damit wandern die nach rechts geschickten Nachrichten in jedem Schritt um eine Position nach rechts, die nach links geschickten Nachrichten wandern analog um eine Position nach links. Nach $p - 1$ Schritten sind alle Nachrichten an alle Knoten verteilt. Eine Multi-Broadcastoperation kann also auf einem linearen Feld mit vier Knoten in drei Schritten realisiert werden, vgl. Abbildung 4.2. Dabei bezeichnet p_i die Nachricht, die Knoten i zur Verfügung stellt.

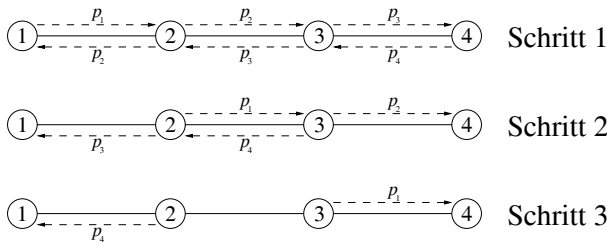


Abb. 4.2. Realisierung einer Multi-Broadcastoperation auf einem linearen Feld.

Da eine Multi-Broadcastoperation in $p - 1$ Schritten realisiert werden kann, können auch **Scatter- und Gatheroperationen** als Spezialisierungen der Multi-Broadcastoperation in höchstens $p - 1$ Schritten realisiert werden.

Da eine Einzel-Broadcastoperation, die ihrerseits eine Spezialisierung einer Scatteroperation ist, vgl. Abbildung 3.7, im schlechtesten Fall ebenfalls $p - 1$ Schritte braucht, können Scatter- und Gatheroperationen im schlechtesten Fall nicht schneller durchgeführt werden. Wenn der Wurzelknoten nicht am Ende des linearen Feldes liegt, kann eine Scatter- oder Gatheroperation in weniger als $p - 1$ Schritten durchgeführt werden. Dazu schickt der Wurzelprozess die für die einzelnen Knoten bestimmten Nachrichten so an seine beiden Nachbarknoten, dass die Nachrichten für weiter entfernt liegende Knoten zuerst geschickt werden. Alle anderen Knoten schicken die von ihrem Nachbarknoten erhaltenen Nachrichten im nächsten Schritt an den jeweils anderen Nachbarknoten weiter.

Zur Bestimmung der Anzahl der Schritte, die für einen **Gesamtaustausch** gebraucht werden, betrachten wir eine Kante $(k, k + 1)$ des linearen Feldes. Diese Kante unterteilt das lineare Feld in zwei Teilmengen mit k und $p - k$ Knoten. Da bei einem vollständigen Austausch jeder Knoten der einen Teilmenge eine Nachricht an einen Knoten der anderen Teilmenge schickt, müssen über die betrachtete Kante $k \cdot (p - k)$ Nachrichten in jede Richtung verschickt werden. Über die mittlere Kante $k = \lfloor p/2 \rfloor$ laufen dabei die meisten Nachrichten. Da über diese Kante $p^2/4$ Nachrichten in jede Richtung geschickt werden müssen, ist dies auch die Minimalanzahl von Schritten zur Realisierung des totalen Austausches. Eine einfache, aber nicht unbedingt die effizienteste Möglichkeit der Realisierung des Gesamtaustausches besteht darin, dass die Knoten des linearen Feldes nacheinander Scatteroperationen ausführen. Jede dieser Scatteroperationen braucht maximal $p - 1$ Schritte, insgesamt resultieren $\theta(p^2)$ Schritte.

Ring. Auf einem Ring kann eine **Einzel-Broadcastoperation** wie beim linearen Feld dadurch realisiert werden, dass der Wurzelknoten die Broadcastnachricht an seine beiden Nachbarn schickt und jeder Knoten die erhaltene Nachricht an seinen anderen Nachbarn weiterschickt. Anders als beim linearen Feld resultiert bei jedem als Wurzelknoten ausgewählten Knoten die gleiche Anzahl $\lfloor p/2 \rfloor$ von Schritten. Da der Durchmesser eines Ringes $\lceil p/2 \rceil$ ist, kann eine Einzel-Broadcastoperation auch nicht schneller realisiert werden.

Eine **Multi-Broadcastoperation** kann wie bei einem linearen Feld realisiert werden, wobei aber anstatt der $p - 1$ Schritte nur $\lfloor p/2 \rfloor$ Schritte erforderlich sind. Im ersten Schritt schickt jeder Knoten seine eigene Nachricht an seine beiden Nachbarn. Im k -ten Schritt für $2 \leq k \leq \lfloor p/2 \rfloor$ schickt jeder Knoten seinem Nachbarn im Uhrzeigersinn die Nachricht, die er im vorangegangenen Schritt von seinem Nachbarn gegen den Uhrzeigersinn erhalten hat und umgekehrt. Nach $\lfloor p/2 \rfloor$ Schritten hat jeder Knoten jede Nachricht erhalten, wobei für gerade p im letzten Schritt jeder Knoten nur eine Nachricht weiterschicken muss. Die Abbildung 4.3 zeigt den resultierenden Nachrichtenfloss für $p = 6$ Knoten, wobei die von Knoten i ausgehende Nachricht mit p_i bezeichnet wird.

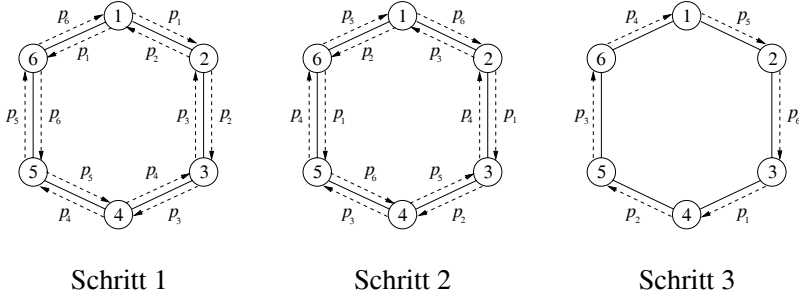


Abb. 4.3. Realisierung einer Multi-Broadcastoperation auf einem Ring.

Die **Scatteroperation** kann als Spezialfall einer Multi-Broadcastoperation realisiert werden. Da sowohl eine Einzel-Broadcastoperation als auch eine Multi-Broadcastoperation wegen des Ringdurchmessers von $\lceil p/2 \rceil$ mindestens $\lfloor p/2 \rfloor$ Schritte braucht, können Scatter- und Gatheroperationen wegen der Dualitäts- und Hierarchiebeziehungen auch nicht schneller realisiert werden. Zur Bestimmung des Aufwandes für einen **Gesamtaustausch** betrachten wir zwei Kanten, die den Ring in je $p/2$ Knoten zerlegen, wobei wir annehmen, dass p gerade ist. Da jeder Knoten der einen Menge eine Nachricht an jeden Knoten der anderen Menge schicken muss, müssen über die beiden Kanten $p^2/4$ Nachrichten verschickt werden. Da die Kanten bidirektional sind, braucht man also mindestens $p^2/8$ Schritte.

Gitter. Auf einem d -dimensionalen symmetrischen Gitter mit insgesamt p Knoten und $\sqrt[d]{p}$ Prozessoren in jeder Dimension benötigt eine **Einzel-Broadcastoperation** die Zeit $\Theta(p^{1/d})$, da der Durchmesser des Gitters $d(p^{1/d} - 1)$ ist. Eine obere Schranke für die Laufzeit einer **Scatteroperation** ist $O(p)$, da dies die Zeit einer Scatteroperation auf einem linearen Feld mit p Prozessoren ist und dies in ein Gitter eingebettet werden kann. Eine Scatteroperation braucht mindestens Zeit $p-1$, da $p-1$ Nachrichten den Wurzelknoten über d Kanten verlassen, also $\lceil \frac{p-1}{d} \rceil$ Schritte benötigen. Mit der gleichen Argumentation ergibt sich, dass auch eine **Multi-Broadcastoperation** die Zeit $\Theta(p)$ braucht.

Für den **Gesamtaustausch** betrachten wir eine gerade Anzahl von Prozessoren und eine $(d-1)$ -dimensionale Ebene, die das Gitter in zwei Teile mit je $p/2$ Knoten unterteilt. Jede Hälfte erhält von der anderen Hälfte $p^2/4$ Nachrichten, die über die Kanten, die die $(d-1)$ -dimensionale Ebene schneidet, gesendet werden müssen. Die Anzahl der Kanten zwischen den Gitterhälften beträgt $(\sqrt[d]{p})^{d-1}$. Es werden also mindestens $p^{\frac{d+1}{d}}$ Schritte benötigt, da $p^2/(4p^{\frac{d-1}{d}}) = 1/(4p^{\frac{d-1-2d}{d}}) = \frac{1}{4}p^{\frac{d+1}{d}}$ gilt.

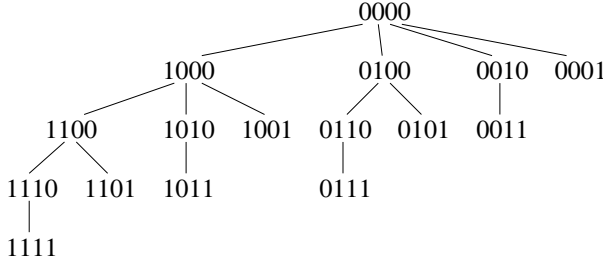
Ein Algorithmus zur Realisierung eines Gesamtaustausches auf einem d -dimensionalen symmetrischen Gitter kann induktiv aus Realisierungen auf niederdimensionalen Gittern aufgebaut werden. Wir zeigen mit Induktion, dass dieser Algorithmus Laufzeit $O(p^{\frac{d+1}{d}})$ hat. Für $d=1$ entspricht das Gitter

einem linearen Feld, für das wir einen Gesamtaustausch in Zeit $O(p^2)$ angegeben haben. Wir nehmen nun an, dass eine Realisierung eines Gesamtaustausches auf einem $(d-1)$ -dimensionalen symmetrischen Gitter mit p Prozessoren in Zeit $O(p^{\frac{d}{d-1}})$ gegeben sei. Der Gesamtaustausch auf dem d -dimensionalen symmetrischen Gitter mit p Prozessoren wird in zwei Phasen realisiert. Dazu wird das Gitter in $\sqrt[d]{p}$ disjunkte Gitter der Dimension $d-1$ zerlegt, indem etwa die letzte Komponente x_d der Knoten $(x_1, \dots, x_d)$ einen festen Wert annimmt. In der ersten Phase werden auf den $(d-1)$ -dimensionalen Gittern Gesamtaustauschoperationen parallel zueinander ausgeführt. Bei den $(d-1)$ -dimensionalen Gittern handelt es sich um symmetrische Gitter mit $\sqrt[d]{p}$ Prozessoren in jeder Dimension, also mit einer Gesamtzahl von $p^{\frac{d-1}{d}}$ Knoten. Nach Annahme kostet ein Gesamtaustausch auf einem Teilgitter also Zeit $O((p^{\frac{d-1}{d}})^{\frac{d}{d-1}}) = O(p)$. Nach der ersten Phase hat jeder Knoten bereits $p^{\frac{d-1}{d}}$ Nachrichten von den anderen Knoten des $(d-1)$ -dimensionalen Teilgitters, in dem er liegt, erhalten. In der zweiten Phase wird die d -te Dimension des Gitters betrachtet. Das d -dimensionale Gitter enthält $p^{\frac{d-1}{d}}$ eindimensionale Teilgitter mit je $\sqrt[d]{p}$ Knoten, wobei jeder dieser Knoten in einem anderen der $(d-1)$ -dimensionalen Teilgitter der ersten Phase liegt und somit $p^{\frac{d-1}{d}}$ Nachrichten erhalten hat. In jedem dieser $p^{\frac{d-1}{d}}$ linearen Felder werden parallel zueinander Gesamtaustauschoperationen durchgeführt. Dies benötigt Zeit $O((\sqrt[d]{p})^2)$. Da $p^{\frac{d-1}{d}}$ Nachrichten von jedem Knoten an alle anderen versendet werden, ergibt dies Zeit $O(p^{\frac{d+1}{d}})$, da $p^{\frac{2}{d}} p^{\frac{d-1}{d}} = p^{\frac{d+1}{d}}$.

4.3.2 Kommunikationsoperationen auf dem Hyperwürfel

Wir betrachten einen Hyperwürfel der Dimension d mit $p = 2^d$ Knoten und benutzen für die Knoten d -Bitnamen $\alpha = \alpha_1 \dots \alpha_d \in \{0, 1\}^d$, wie sie in Abschnitt 2.5.1 eingeführt wurden und benutzen die Voraussetzungen aus Abschnitt 4.3.1. Für die Realisierung einer **Einzel-Broadcastoperation** konstruieren wir einen aufspannenden Baum, dessen Wurzel der Wurzelknoten der Broadcastoperation ist. Wir geben zunächst die Konstruktion für den Fall an, dass der Knoten mit Binärnamen $\alpha = 00 \dots 0 = 0^d$ der Wurzelknoten ist, und leiten dann die Konstruktion eines aufspannenden Baumes für einen beliebigen anderen Wurzelknoten ab.

Ausgehend von der Wurzel $00 \dots 0$ werden die Kinder eines jeden Knotens v dadurch festgelegt, dass die Bits mit Wert 0 im Bitnamen von v , die nach der von links gesehen letzten Eins stehen, nacheinander invertiert werden und so jeweils den Bitnamen eines Kindes angeben. Endet ein Bitnamen mit k Nullen auf der rechten Seite, so hat der entsprechende Knoten also k Kinder. Für $d = 4$ ergibt sich damit der folgende aufspannende Baum:



Der durch die angegebene Konstruktion entstehende Graph hat folgende Eigenschaften. Die Bitrepräsentation eines Kindknotens unterscheidet sich von der eines Elternknotens nach Konstruktion in genau einer Bitposition, d.h. Kind- und Elternknoten sind im Hyperwürfel mit einer Kante verbunden. Die Konstruktion erfasst alle Knoten des Hyperwürfels und erzeugt keine Zyklen, es entsteht also ein aufspannender Baum. Die Bitrepräsentation aller Blätter des aufspannenden Baumes endet mit einer Eins. Der maximale Grad eines Knotens ist d . Da die Bitrepräsentation eines Kindknotens eine Eins mehr hat als die seines Elternknotens, hat ein beliebiger Pfad von der Wurzel zu einem Blatt maximal die Länge d . Der konstruierte aufspannende Baum hat also die Tiefe d .

Für einen beliebigen Knoten z als Wurzelknoten der Einzel-Broadcastoperation wird ein aufspannender Baum T_z mit Wurzel z aus dem soeben konstruierten aufspannenden Baum T_0 mit Wurzel $00 \dots 0$ abgeleitet, indem jeder Knoten x aus T_0 durch $x \oplus z$ ersetzt wird. Mit $\oplus$ bezeichnen wir wieder eine bitweise **xor**-Operation (ausschließendes Oder), d.h.

$$a_1 \dots a_d \oplus b_1 \dots b_d = c_1 \dots c_d \text{ mit } c_i = \begin{cases} 1 & \text{falls } a_i \neq b_i \\ 0 & \text{sonst} \end{cases} \text{ für } 1 \leq i \leq d.$$

Um zu zeigen, dass der resultierende Baum ein aufspannender Baum ist, betrachten wir eine beliebige Kante (v, w) in T_0 . Da sich die Bitrepräsentationen von v und w in genau einer Bitposition unterscheiden, unterscheiden sich auch $v \oplus z$ und $w \oplus z$ in genau dieser einen Bitposition. Daher ist $(v \oplus z, w \oplus z)$ auch eine Kante im Hyperwürfel.

Mit Hilfe des konstruierten aufspannenden Baumes kann eine *Einzel-Broadcastoperation* in $d = \log p$ Schritten dadurch realisiert werden, dass im ersten Schritt der Wurzelknoten die zu verteilende Nachricht an seine Kinder weitergibt, und dass in den darauffolgenden Schritten jeder Knoten die Nachricht, die er im vorangegangenen Schritt erhalten hat, wiederum an seine Kinder weitergibt. Da der Durchmesser des Hyperwürfels d ist, kann die Einzel-Broadcastoperation auch nicht schneller ausgeführt werden und es ergibt sich die Laufzeit $\Theta(\log(p))$.

Wir betrachten nun die Realisierung einer **Multi-Broadcastoperation** auf einem Hyperwürfel. Bei einer Multi-Broadcastoperation muss jeder Knoten $p - 1$ Nachrichten von den anderen Knoten empfangen. Da jedem Knoten $d = \log p$ einlaufende Kanten zur Verfügung stehen, braucht eine Realisierung der Multi-Broadcastoperation mindestens $\lceil (p - 1) / \log p \rceil$ Schritte, da

das aufeinanderfolgende Empfangen der $p - 1$ Nachrichten eines Knotens bereits mindestens $\lceil (p - 1)/\log p \rceil$ Schritte benötigt.

Im Folgenden konstruieren wir eine Realisierung, die genau $\lceil (p - 1)/\log p \rceil$ Schritte braucht und daher ein optimales Verfahren darstellt. Diese Realisierung beruht ebenfalls auf der Konstruktion aufspannender Bäume, wobei für jeden Knoten ein anderer aufspannender Baum verwendet wird. Die Übertragung der Nachrichten über einen einzelnen aufspannenden Baum erfolgt ähnlich wie bei obiger Einzel-Broadcastoperation, d.h. die Nachricht wird in mehreren aufeinanderfolgenden Schritten gleichzeitig an Kinder im Baum geschickt. Die Kanten, über die die Nachricht im i -ten Schritt geschickt werden, werden auch Kanten der i -ten Stufe genannt. Die Idee des Algorithmus besteht darin, die aufspannenden Bäume für die verschiedenen Wurzelknoten so zu konstruieren, dass die korrespondierenden Stufen der verschiedenen aufspannenden Bäume aus *disjunkten* Kantenmengen des Hyperwürfels bestehen, so dass bei der Durchführung des Verschickens von Nachrichten über eine Stufe keine Kollisionen dadurch entstehen, dass die gleiche Netzwerkkante in den zeitlich korrespondierenden Stufen unterschiedlicher Bäume verwendet wird.

Zur Einführung der verwendeten Notation betrachten wir einen aufspannenden Baum T_0 mit Wurzel $00 \dots 0$, der eine Tiefe m habe. Die Menge der Kanten der i -ten Stufe von T_0 , über die bei der Durchführung einer Einzel-Broadcastoperation im i -ten Schritt Nachrichten übertragen werden, bezeichnen wir mit A_i , $1 \leq i \leq m$. Da T_0 ein aufspannender Baum ist, sind die Mengen $A_1, \dots, A_m$ paarweise disjunkt. S_i seien die Startknoten der Kanten in A_i und E_i seien die Endknoten der Kanten in A_i . Dabei ist $S_1 = \{(00 \dots 0)\}$ und $S_i \subset S_1 \cup \bigcup_{k=1}^{i-1} E_k$. Für einen gegebenen aufspannenden Baum T_0 erhalten wir durch Anwendung der **xor**-Operation die Mengen

$$A_i(t) = \{(x \oplus t, y \oplus t) \mid (x, y) \in A_i\} \quad \text{für } 1 \leq i \leq m, \quad (4.8)$$

die wieder aufspannende Bäume T_t mit Wurzel $t \in \{0, 1\}^d$ ergeben.

Unser Ziel ist es, die Mengen $A_1, \dots, A_m$ so zu konstruieren, dass die Mengen $A_i(t)$ für $t \in \{0, 1\}^d$ paarweise *disjunkte* Kantenmengen darstellen und die Übertragung im i -ten Schritt auf allen Mengen $A_i(t)$ für $t \in \{0, 1\}^d$ *simultan ohne Kollisionen* stattfinden kann, wobei $A_i = A_i(0)$ gilt. Um dies zu erreichen, werden die Mengen A_i mit folgender Bedingung definiert:

- Die zu zwei beliebigen Kanten $(x, y) \in A_i$ und $(x', y') \in A_i$ gehörenden Knoten x und x' bzw. y und y' dürfen sich *nicht* in der gleichen Bitposition unterscheiden.

Diese Bedingung wird gefordert, da zwei Kanten, deren Endknoten sich in derselben Bitposition unterscheiden, durch die **xor**-Operation mit einem geeigneten Knoten t aufeinander abgebildet werden können, so dass diese Knoten bei der Konstruktion des Baumes t in der Menge $A_i(t)$ wären und so die Mengen A_i und $A_i(t)$ *nicht* disjunkt wären. Da es nur d verschiedene Bitpositionen gibt, kann jede Menge A_i als Folge dieser Bedingung also höchstens d

Elemente enthalten. Wir konstruieren daher Mengen mit den Größen $|A_i| = d$ für $1 \leq i < m$ und $|A_m| \leq d$. Da die Mengen $A_1, \dots, A_m$ paarweise disjunkt sind und da die Gesamtanzahl der Kanten im aufgespannten Baum $2^d - 1$ beträgt (je eine einlaufende Kante für jeden Knoten außer der Wurzel), gilt

$$\left| \bigcup_{i=1}^m A_i \right| = 2^d - 1$$

und damit wegen $m \cdot d \geq 2^d - 1$

$$m = \left\lceil \frac{2^d - 1}{d} \right\rceil.$$

Wenn es also gelingt, die Mengen $A_i(t)$ kollisionsfrei zu konstruieren, haben wir eine Realisierung einer Multi-Broadcastoperation in Zeit m gefunden.

Als Illustration zeigt Abbildung 4.4 für den Fall $d = 3$ den aufspannenden Baum, der für die Einzel-Broadcastoperation konstruiert wurde. Die Endknoten der beiden Kanten $e_1 = ((010), (011))$ und $e_2 = ((100), (101))$ unterscheiden sich beide von den Startknoten in der ersten Bitposition von rechts. Der durch die **xor**-Operation erzeugte Baum mit Wurzel 110 enthält dieselben Kanten e_1 und e_2 in der zweiten Stufe, was zu einer Kollision führen würde. Die Verschiebung der Kanten in Stufe drei würde zu Kollisionen mit der Stufe drei der Bäume mit Wurzel 010 und 100 führen.

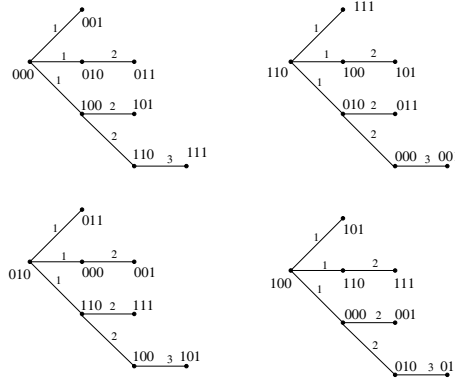


Abb. 4.4. Der aufspannende Baum der Einzel-Broadcastoperation und der durch die **xor**-Operation erzeugte Baum mit Wurzel 110 haben auf Stufe 2 zwei gleiche Kanten (010, 011) und (100, 101). Würde eine der Kanten in die Stufe 3 verschoben werden, so ergäbe dies eine Kollision, da diese Kanten im Baum mit Wurzel 010 bzw. 100 auf Stufe 3 liegen.

Abbildung 4.5 zeigt für den Fall $d = 3$ die Konstruktion der Mengen A_1, A_2, A_3 mit $|A_1| = |A_2| = 3$ und $|A_3| = 1$ und die daraus resultierenden aufspannenden Bäume T_t für $t \in \{0, 1\}^3$. Abbildung 4.5 illustriert, dass die

resultierenden aufspannenden Bäume für $d = 3$ so aufgebaut sind, dass für $i = 1, 2, 3$ über die Kanten $A_i(t)$ für $t \in \{0, 1\}^3$ simultan gesendet werden kann, da keine Kante des Hyperwürfel-Netzwerkes für ein gegebenes i in mehr als einer der 2^3 Mengen $A_i(t)$ auftritt. Damit treten in keinem der $m = \lceil (2^3 - 1)/3 \rceil = 3$ Schritte Kollisionen auf, und die Multi-Broadcastoperation kann in 3 Schritten durchgeführt werden.

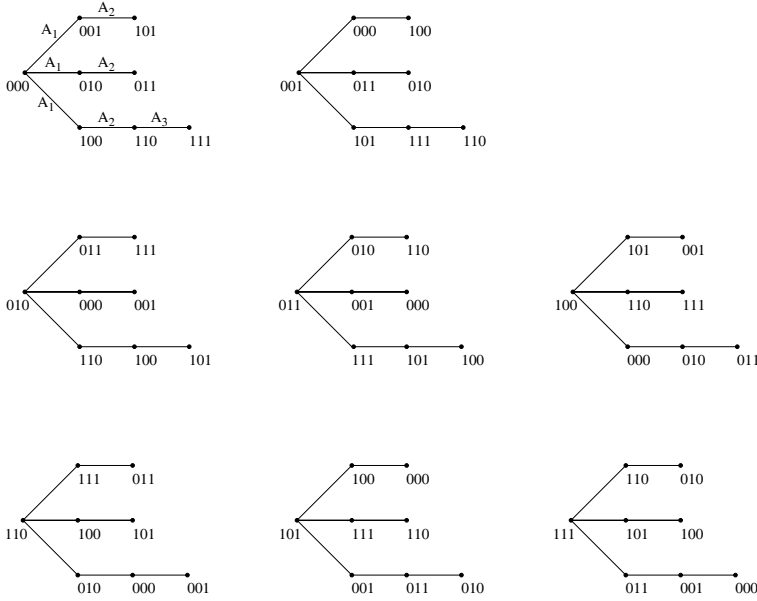


Abb. 4.5. Mengen A_1, A_2, A_3 für die Realisierung einer Multi-Broadcastoperation auf einem d -dimensionalen Hyperwürfel mit $d = 3$ (oben links). Dabei ist $A_1 = \{(000, 001), (000, 010), (000, 100)\}$, $A_2 = \{(001, 101), (010, 011), (100, 110)\}$ und $A_3 = \{(110, 111)\}$. Die weiteren Diagramme zeigen die aufspannenden Bäume für die anderen Wurzelknoten nach Gleichung (4.8).

Zur Konstruktion der Mengen A_i für allgemeines d definieren wir die Mengen

$$N_k = \{t \in \{0, 1\}^d \mid t \text{ hat } k \text{ Einsen und } d - k \text{ Nullen}\} \quad \text{für } 0 \leq k \leq d,$$

d.h. N_k ist die Menge der Knoten des Hyperwürfels, deren Bitrepräsentation genau k Einsen enthält. Dabei gilt

$$|N_k| = \binom{d}{k} = \frac{d!}{k!(d-k)!},$$

und $N_0 = \{(00 \cdots 0)\}$ und $N_d = \{(11 \cdots 1)\}$. Jede der so definierten Mengen N_k wird in disjunkte Teilmengen $R_{k1}, \dots, R_{kn_k}$ zerlegt, wobei in einer der Teilmengen R_{ki} alle Elemente aus N_k enthalten sind, die durch Bitrotation nach links ineinander überführt werden können. Die Teilmengen

$R_{k1}, \dots, R_{kn_k}$ bilden also die Äquivalenzklassen von N_k bezüglich der Relation R mit $(x, y) \in R$, falls $x \in \{0, 1\}^d$ aus $y \in \{0, 1\}^d$ durch Bitrotation nach links erhalten werden kann. Wir legen fest, dass R_{k1} die Menge aller Knoten ist, die aus $(0^{d-k}1^k)$ durch Bitrotation entstehen, wobei $(0^{d-k}1^k)$ entsprechend der Konvention das Wort ist, das von links gesehen zuerst $d - k$ -mal eine 0 und danach k -mal eine 1 hat. Die Zuordnung der Bezeichnungen $R_{k2}, \dots, R_{kn_k}$ an die restlichen Äquivalenzklassen von N_k kann beliebig erfolgen. Aufbauend auf den so definierten Teilmengen ordnen wir jedem Knoten $t \in \{0, 1\}^d$ eine Zahl $n(t) \in \{0, \dots, 2^d - 1\}$ zu, die seiner Position in der Aufzählung der Knoten entsprechend der Reihenfolgen der folgenden Aufzählung der Mengen R_{kj} entspricht:

$$\{\alpha\} R_{11} R_{21} \dots R_{2n_2} \dots R_{k1} \dots R_{kn_k} \dots R_{(d-2)1} \dots R_{(d-2)n_{d-2}} R_{(d-1)1} \{\beta\}, \quad (4.9)$$

mit $\alpha = 00 \dots 0$ und $\beta = 11 \dots 1$. Es gilt z.B. $n(\alpha) = 0$ und $n(\beta) = 2^d - 1$. Jedem Knoten $t \in \{0, 1\}^d$ ordnen wir weiter eine Zahl $m(t)$ zu

$$m(t) = 1 + [(n(t) - 1) \bmod d], \quad (4.10)$$

d.h. den Knoten in der Aufzählung gemäß (4.9) werden in der Reihenfolge ihres Auftretens in der Aufzählung reihum Nummern zwischen 1 und d zugeordnet (d -Nummerierung). Für die Elemente der einzelnen Mengen R_{kj} , $k = 1, \dots, d$, $j = 1, \dots, n_k$, wird eine eindeutige Reihenfolge der Elemente mit folgenden Bedingungen festgelegt.

- Das erste Element t aus jedem R_{kj} sei so gewählt, dass gilt:

$$\text{das Bit an Position } m(t) \text{ von rechts ist eine 1.} \quad (4.11)$$

- Das i -te Element ergibt sich aus dem ersten Element durch Bitrotation um i Stellen nach links. Daher gilt Bedingung (4.11) für jedes Element von R_{kj} .

Für die Mengen R_{k1} , $k = 1, \dots, d$, gilt zusätzlich:

- Das erste Element $t \in R_{k1}$ hat eine 0 an der Bitposition rechts von $m(t)$, d.h. für $m(t) > 1$ ist das Bit an Position $m(t) - 1$ eine 0 und für $m(t) = 1$ ist das erste Bit von links eine 0.
- Diese Eigenschaft gilt für alle anderen Elemente von R_{k1} entsprechend der Reihenfolge durch Bitrotation nach links.

Für den Fall $d = 4$ ergeben sich die nun dargestellten Teilmengen mit der angegebenen internen Reihenfolge, wobei über jedem Knoten t der Wert $m(t)$ angegeben ist:

$$\begin{array}{cccc} N_0 & \overset{0}{(0000)} & & \\ N_1 & \overset{1}{(0001)} & \overset{2}{(0010)} & \overset{3}{(0100)} \quad \overset{4}{(1000)} \\ & \underbrace{\hspace{10em}} & & \\ & R_{11} & & \end{array}$$

$$\begin{array}{lcl}
N_2 & \underbrace{\overset{1}{(0011)} \quad \overset{2}{(0110)} \quad \overset{3}{(1100)} \quad \overset{4}{(1001)}}_{R_{21}} & \underbrace{\overset{1}{(0101)} \quad \overset{2}{(1010)}}_{R_{22}} \\
N_3 & \underbrace{\overset{3}{(1101)} \quad \overset{4}{(1011)} \quad \overset{1}{(0111)} \quad \overset{2}{(1110)}}_{R_{31}} & \\
N_4 & \overset{3}{(1111)} &
\end{array}$$

Unter Benutzung der Nummerierung $n(t)$ definieren wir $m+1$ Knotenmengen $E_0, E_1, \dots, E_m$, die jeweils die Endknoten der zu definierenden Kantenmengen $A_1, \dots, A_m$ bilden sollen, wie folgt:

$$E_0 = \{(00 \cdots 0)\}$$

$$E_i = \{t \in \{0, 1\}^d \mid (i-1)d + 1 \leq n(t) \leq i \cdot d\} \quad \text{für } 1 \leq i < m$$

$$E_m = \{t \in \{0, 1\}^d \mid (m-1)d + 1 \leq n(t) \leq 2^d - 1\} \quad \text{mit } m = \left\lceil \frac{2^d - 1}{d} \right\rceil.$$

Die Kantenmengen $E_1, \dots, E_m$ ergeben sich also aus der Anordnung in $R_{11} \cdots R_{d1}$, indem nacheinander Blöcke mit d Elementen gebildet werden. Außer E_0 und E_m enthält jede Knotenmenge d Elemente. Mit diesen Knotenmengen erfolgt die **Konstruktion der Kantenmengen** A_i , $1 \leq i \leq m$ folgendermaßen:

- Die Kantenmenge A_i ergibt sich für $1 \leq i \leq m$ dadurch, dass man jeden Knoten t aus E_i mit dem Knoten t' verbindet, der sich aus t dadurch ergibt, dass das Bit an Position $m(t)$ von rechts invertiert wird.
- Es gibt einen Spezialfall: Wenn $m(11 \cdots 1) = d$ ist, wird anstatt dem d -ten Bit von rechts das $(d-1)$ -te Bit von rechts invertiert, d.h. statt $((11 \cdots 1), (011 \cdots 1)) \in A_m$ setzen wir $((11 \cdots 1), (1011 \cdots 1)) \in A_m$.

Das nach dieser Vorschrift zu invertierende Bit in t ist nach Konstruktion immer eine 1, so dass die Bitrepräsentation von t' eine 1 weniger hat als die von t , d.h. wenn $t \in N_k$ gilt, so folgt $t' \in N_{k-1}$. Da sich t' aus t durch Invertieren eines einzelnen Bits ergibt, ist (t, t') auch eine Kante des Hyperwürfels. Für $d = 4$ ergeben sich die Knoten- und Kantenmengen in Abbildung 4.6.

Die Definition der Knotenmengen und die Zuordnung der d -Nummerierung $m(t)$ zu den Knoten t nach (4.10) stellt sicher, dass sich für festes i die durch unterschiedliche Kanten aus A_i miteinander verbundenen Knoten in *unterschiedlichen* Bitpositionen voneinander unterscheiden, d.h. für $e_1, e_2 \in A_i$ mit $e_1 = (t_1, t'_1)$, $e_2 = (t_2, t'_2)$ und $e_1 \neq e_2$ unterscheiden sich t_1 und t'_1 in Bitposition $m(t_1)$ und t_2 und t'_2 unterscheiden sich in Bitposition $m(t_2) \neq m(t_1)$. Dies gilt, da E_i aufeinanderfolgende Knoten der Aufzählung $n(t)$ enthält, die nach Konstruktion alle unterschiedliche $m(t)$, $1 \leq m(t) \leq d$, haben.

Es bleibt zu zeigen, dass die Mengen $A_1, \dots, A_m$ einen aufspannenden Baum definieren, über den eine Broadcastnachricht ausgehend von Knoten $(00 \cdots 0)$ in m Schritten übertragen werden kann. Dazu zeigen wir:

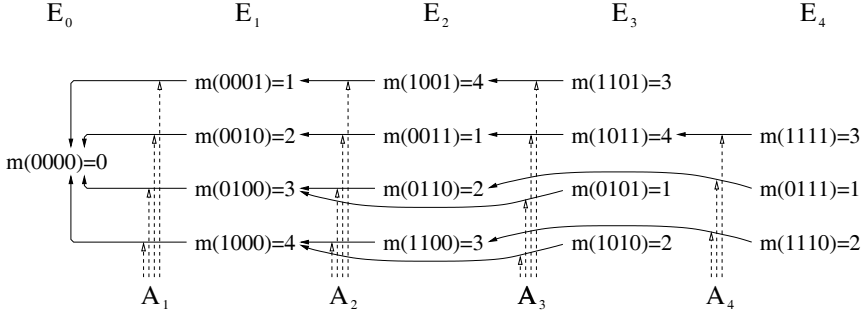


Abb. 4.6. Aufspannender Baum mit Wurzel $00 \dots 0$ für die Multi-Broadcastoperation auf dem Hyperwürfel mit $d=4$. Die Kantenmengen A_i , $i = 1, \dots, 4$, der einzelnen Stufen sind durch gestrichelte Pfeile angegeben.

- Durch Konstruktion der Kantenmengen A_i wird ein Knoten $t \in E_i$ mit einem Knoten $t' \in \bigcup_{k=1}^{i-1} E_k$ verbunden.

Wir betrachten jetzt eine Kante $(t, t') \in A_i$, d.h. es ist $t \in E_i$, und zeigen, dass $t' \in E_k$ für $k < i$ gilt. Da nach Konstruktion die Bitrepräsentation von t' eine Eins weniger hat als die von t , gilt $n(t') < n(t)$. Damit ist $t' \in E_k$ für $k > i$ ausgeschlossen. Um $k < i$ zu zeigen, betrachten wir folgende Fälle:

- Für $t = 11 \dots 1$, d.h. es ist $i = m$, und falls $m(t) = d$ ist, gilt der Spezialfall der Konstruktion der Menge A_m , also $t' \in E_{m-1}$. Der Grund für den Spezialfall liegt darin, dass wegen $m(1011 \dots 1) = d$ im Falle $m(11 \dots 1) = d$ die beiden Knoten $(11 \dots 1)$ und $(1011 \dots 1)$ in E_m bzw. E_{m-1} liegen, dass aber wegen $m(011 \dots 1) = 1$ die Knoten $(11 \dots 1)$ und $(011 \dots 1)$ in der gleichen Knotenmenge E_m lägen.
- Für $t = 11 \dots 1$ und $m(t) < d$ gilt $m(t) = d - k$ für ein $1 \leq k < d$, und nach Konstruktion enthält E_m in diesem Fall $d - k$ Elemente. Der Knoten t' ergibt sich aus t dadurch, dass das Bit an Position $d - k$ von rechts auf Null gesetzt wird. Sei t'' das erste Element in $R_{d-1,1}$. Da $|R_{d-1,1}| = |N_{d-1}| = d$ und $|N_d| = 1$, folgt aus $m(11 \dots 1) = d - k$ auch $m(t'') = d - k$. Daher hat t'' an Position $d - k$ von rechts eine Eins und an Position $(d - k - 1)$ von rechts eine Null. Da das zweite Element von $R_{d-1,1}$ sich aus dem ersten durch eine Bitrotation nach links ergibt, ist t' das zweite Element von $R_{d-1,1}$. Wegen $|E_m| = d - k$ mit $k \geq 1$ gilt also $t' \in E_{m-1}$.

Wir untersuchen jetzt den Fall $t \neq 11 \dots 1$. Um zu zeigen, dass t und t' in unterschiedlichen Knotenmengen liegen, zeigen wir, dass $n(t) - n(t') \geq d$ gilt.

- Für $t \in R_{kn}$ mit $n > 1$ lässt sich dies einfach zeigen. Es ist $t \in N_k$ und $t' \in N_{k-1}$, da t' eine Eins weniger hat als t . Daher liegen in der Liste $R_{11} \dots R_{d-1,1}$ alle Elemente der Klasse R_{k1} zwischen t und t' . R_{k1} ist nach Konstruktion die Äquivalenzklasse von $(0^{d-k}1^k)$ und es gilt $|R_{k1}| = d$. Es folgt also $n(t) - n(t') \geq d$.

- Für $t \in R_{k1}$ ist entweder $t = (0^l 1^k 0^{d-l-k})$ für ein l mit $0 \leq l < d - k$ oder $t = (1^l 0^{d-k} 1^{k-l})$ für ein l mit $0 \leq l < k$. In beiden Fällen muss an Position $m(t)$ von rechts eine Eins stehen, das Bit rechts davon muss eine Null sein. Daher muss im ersten Fall $m(t) = l + k$ sein, d.h. $t' = (0^l 1^{k-1} 0^{d-l-k+1})$ und daher $t' \in R_{k-1,1}$. Im zweiten Fall muss $m(t) = l$ sein und daher $t' = (1^{l-1} 0^{d-k} 1^{k-l})$, also ebenfalls $t' \in R_{k-1,1}$. Da in beiden Fällen $t' \in R_{k-1,1}$ gilt, müssen alle Elemente der Mengen $R_{k-1,2}, \dots, R_{k-1,n_{k-1}}$ zwischen t und t' liegen. Da N_{k-1} genau $\binom{d}{k-1}$ Elemente hat, und da die d Elemente von $R_{k-1,1}$ ausgenommen sind, liegen zwischen t und t' mindestens

$$\binom{d}{k-1} - d$$

Elemente. Wie durch Induktion und Verwendung von $\binom{d}{k-1} = \binom{d-1}{k-1} + \binom{d-1}{k-2}$ gezeigt werden kann, gilt für $2 < k < d$ und $d \geq 5$

$$\binom{d}{k-1} - d \geq d.$$

Für $k = 1, 2$ gilt $R_{11} = E_1$ und $R_{21} = E_2$ für alle d und damit $t' \in E_{k-1}$. Für $d = 3$ und $d = 4$ kann $t' \in E_i$, $i < k$, explizit gezeigt werden, siehe Abbildungen 4.5 und 4.6.

Damit gehören t und t' in jedem Fall unterschiedlichen Knotenmengen an. Insgesamt haben wir gezeigt, dass eine Broadcastnachricht im i -ten Schritt an alle Knoten in E_i weitergegeben werden kann, da bei einer simultanen Übertragung über alle Kanten der Mengen $A_i(t)$ für $t \in \{0, 1\}^d$ keine Kollisionen auftreten. Eine Multi-Broadcastoperation kann also in $m = \lceil (2^d - 1)/d \rceil$ Schritten durchgeführt werden.

Zur Realisierung einer **Scatteroperation** muss der Wurzelknoten an jeden anderen Knoten des Hyperwürfels eine Nachricht schicken. Für die insgesamt $2^d - 1$ zu verschickenden Nachrichten stehen dem Wurzelprozess d auslaufende Kanten zur Verfügung. Somit braucht die Realisierung einer Scatteroperation mindestens $\lceil (2^d - 1)/d \rceil$ Schritte. Da dies die gleiche Anzahl von Schritten wie für eine Multi-Broadcastoperation ist, liefert die Realisierung einer Scatteroperation durch eine Spezialisierung des Algorithmus für eine Multi-Broadcastoperation eine optimale Realisierung. Eine Gatheroperation kann als duale Operation zu einer Scatteroperation analog realisiert werden.

Wir betrachten jetzt die Realisierung eines **Gesamtaustausches**. Eine untere Schranke für die Anzahl der durchzuführenden Schritte erhält man durch Betrachtung einer Zerlegung eines d -dimensionalen Hyperwürfels in zwei $(d - 1)$ -dimensionale Hyperwürfel. Jeder dieser beiden Hyperwürfel enthält $p/2 = 2^{d-1}$ Knoten. Zwischen diesen beiden Hyperwürfeln gibt es 2^{d-1} Kanten. Bei einem vollständigen Austausch muss jeder Knoten des einen Hyperwürfels eine Nachricht an jeden Knoten des anderen Hyperwürfels schicken. Es müssen also $(2^{d-1})^2 = 2^{2d-2}$ Nachrichten über die

2^{d-1} zur Verfügung stehenden Kanten transportiert werden, wozu mindestens $2^{2d-2}/2^{d-1} = 2^{d-1} = p/2$ Schritte benötigt werden.

Wir konstruieren jetzt einen *rekursiv* aufgebauten Algorithmus, der $2^d - 1$ Schritte braucht. Für $d = 1$ besteht der Hyperwürfel aus zwei Knoten, die über eine bidirektionale Kante miteinander verbunden sind. Ein Gesamtaustausch besteht darin, dass die beiden Knoten sich gegenseitig eine Nachricht zuschicken. Dies kann in einem Schritt realisiert werden. Aufbauend auf einem Algorithmus für einen d -dimensionalen Hyperwürfel mit $2^d - 1$ Schritten erhalten wir einen Algorithmus für einen $(d + 1)$ -dimensionalen Hyperwürfel, indem wir diesen in zwei d -dimensionale Hyperwürfel C_1 und C_2 zerlegen. Der Algorithmus führt die folgenden drei Phasen aus:

1. In C_1 und C_2 wird simultan ein Gesamtaustausch vorgenommen, d.h. jeder Knoten in C_1 bzw. C_2 tauscht seine Nachrichten mit jedem anderen Knoten aus C_1 bzw. C_2 aus. Nach Induktionsannahme braucht diese Phase $2^d - 1$ Schritte.
2. Jeder Knoten in C_1 bzw. C_2 schickt seine Nachrichten für die Knoten im jeweils anderen Teilwürfel an den korrespondierenden Knoten in diesem Teilwürfel, mit dem er durch eine Kante verbunden ist. Somit schickt jeder Knoten über eine seiner auslaufenden Kanten 2^d Nachrichten. Da jeder Knoten eine andere Kante für die Übertragung verwendet, erfordert diese Phase insgesamt 2^d Schritte.
3. Die in Phase 2 erhaltenen Nachrichten werden innerhalb von C_1 und C_2 mit Hilfe eines Gesamtaustausches auf den jeweiligen Teilwürfeln verteilt. Der Austausch auf C_1 bzw. C_2 kann wie in Phase 1 simultan erfolgen. Insgesamt braucht Phase 3 daher $2^d - 1$ Schritte.

Phase 1 und 2 können simultan zueinander ausgeführt werden, da unterschiedliche Kanten des Hyperwürfels verwendet werden. Zusammen brauchen Phase 1 und 2 daher 2^d Schritte. Da Phase 3 nach Phase 2 ausgeführt werden muss, braucht der Gesamtalgorithmus $2^d + 2^d - 1 = 2^{d+1} - 1$ Schritte. Insgesamt ergibt sich Zeit $\Theta(p)$.

4.3.3 Kommunikationsoperationen auf einem Baum

Wir betrachten einen vollständigen binären Baum, dessen Knoten die zu verbindenden Prozessoren und dessen Kanten die Kanten des Netzwerkes repräsentieren. Wieder gelten die Voraussetzungen aus Abschnitt 4.3.1. Sei p die Anzahl der Knoten des Baumes. Da der Baum vollständig ist, gilt $p = 2^{t+1} - 1$, wobei t die Tiefe des Baumes ist.

Eine **Einzel-Broadcastoperation** auf einem Baum braucht je nach Wurzelknoten zwischen t und $2t$ Schritten, wobei t Schritte benötigt werden, wenn die Wurzel des Baumes auch Wurzelknoten der Broadcastoperation ist, und $2t$ Schritte, wenn ein Blatt des Baumes Wurzelknoten der Broadcastoperation ist. Im ersten Schritt schickt der Wurzelknoten der Broadcastoperation

die Broadcastnachricht an seine maximal drei Nachbarknoten, in den folgenden Schritten schickt jeder Knoten die im vorangegangenen Schritt empfangene Nachricht an alle Nachbarn weiter, von denen die Nachricht bisher noch nicht empfangen wurde.

Eine **Scatteroperation** kann in maximal $p - 1$ Schritten dadurch realisiert werden, dass der Wurzelknoten der Scatteroperation in jedem Schritt eine Nachricht über eine seiner auslaufenden Kanten abschickt, wobei die Nachrichten entsprechend ihrer Distanz zum jeweiligen Zielknoten geordnet werden, d.h. die Nachrichten für weiter entfernt liegende Knoten werden zuerst abgeschickt. Die Nachrichten werden mit der Nummer des Empfänger-knotens versehen, damit die dazwischenliegenden Knoten eine empfangene Nachricht im nächsten Schritt in die richtige Richtung weiterschicken können. Abbildung 4.7 veranschaulicht die Arbeitsweise des beschriebenen Algorithmus. Da der Wurzelknoten der Scatteroperation in jedem Schritt eine Nachricht abschickt, und da die Nachrichten geeignet geordnet sind, ist die Scatteroperation nach maximal $p - 1$ Schritten abgeschlossen. Je nach Lage des Wurzelknotens der Scatteroperation kann die Anzahl der Schritte dadurch reduziert werden, dass der Wurzelknoten Nachrichten über alle auslaufenden Kanten abschickt. Für den Wurzelknoten des Baumnetzwerkes kann so die Anzahl der Schritte auf $(p - 1)/2$ reduziert werden. Dies ist auch die untere Schranke der Laufzeit. Für Blattknoten müssen dagegen immer $p - 1$ Schritte durchgeführt werden.

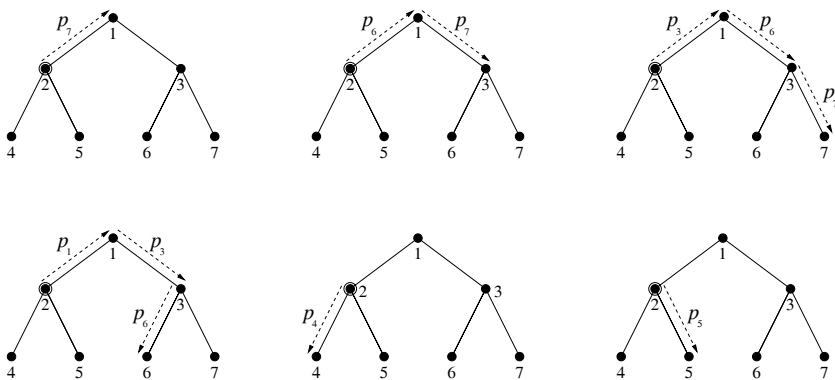


Abb. 4.7. Veranschaulichung der Realisierung einer Scatteroperation auf einem vollständigen, binären Baum mit 7 Knoten als Netzwerk. Der Wurzelknoten der Scatteroperation ist der Knoten mit Nummer 2. Die Nachricht, die Knoten 2 für Knoten i losschickt, wird mit p_i bezeichnet.

Eine **Multi-Broadcastoperation** kann dadurch realisiert werden, dass jeder Knoten im ersten Schritt die von ihm zu verteilende Nachricht an alle seine Nachbarn schickt. In den folgenden Schritten untersucht jeder Knoten i alle auslaufenden Kanten (i, j) . Wenn Knoten i eine Nachricht hat, die noch

nicht an j geschickt wurde, schickt i diese Nachricht über Kante (i, j) . Gibt es mehrere solcher Nachrichten, so wird eine beliebige ausgewählt.

Um zu zeigen, dass durch dieses Vorgehen nach $p - 1$ Schritten alle Nachrichten zugestellt sind, definieren wir für jeden Knoten i die Mengen $R(i, x)$, die alle Nachrichten enthalten, die Knoten i zum Zeitpunkt x bereits empfangen hat, und die Mengen $X(i, j)$, die alle Nachrichten enthalten, die i bereits an seinen Nachbarn j geschickt hat und zuvor nicht von diesem Knoten erhalten hat. Zum Zeitpunkt $x = 0$ wird $R(i, 0)$ mit der eigenen Nachricht von Knoten i initialisiert. Da noch keine Nachrichten verschickt wurden, werden die Mengen durch $X(i, j) = \emptyset$ initialisiert. d.h. $X(i, j)$ enthält alle Nachrichten, die bisher über die Kante (i, j) gesendet wurden.

Zu jedem Zeitpunkt $x \geq 1$ untersucht jeder Knoten i jede auslaufende Kante (i, j) . Wenn i eine noch nicht an j geschickte Nachricht hat, d.h. wenn $R(i, x) \setminus X(i, j) \neq \emptyset$, wählt i eine beliebige Nachricht aus $R(i, x) \setminus X(i, j)$ aus, schickt diese über (i, j) an j und fügt sie in $X(i, j)$ ein. Die zum Zeitpunkt x von Nachbarn ankommenden Nachrichten werden in $R(i, x + 1)$ eingefügt. Kommt eine Nachricht von Nachbar j , so wird sie ebenfalls in die Menge $X(i, j)$ eingefügt. Das Verfahren endet, wenn keiner der Knoten i mehr eine Nachricht hat, die er noch nicht verschickt hat, d.h. wenn für jede auslaufende Kante (i, j) gilt, dass $R(i, x) = X(i, j)$. Da zwischen zwei beliebigen Knoten in einem Baum eindeutige Pfade existieren, wird eine Multi-Broadcastoperation durch diesen lokalen Algorithmus korrekt realisiert.

Zur Berechnung der Anzahl der erforderlichen Schritte definieren wir für jede Kante (i, j) den Wert $b(i, j)$ als die Anzahl der Knoten k , deren Pfad zu Knoten i über die Kante (j, i) führt. Für diese Zahlen $b(i, j)$ gilt für beliebige $q \geq 0$ die folgende Aussage:

- Wenn $b(i, j) \geq q$, wird in Schritt q eine Nachricht über Kante (j, i) an i geschickt.

Intuitiv bedeutet diese Aussage, dass es keine zeitlichen Lücken beim Versenden über eine Kante gibt. Wir zeigen diese Aussage durch Induktion über $b(i, j)$. Für $b(i, j) = 1$ ist j ein Blatt, und i ist Elternknoten eines Blattes. Da im ersten Schritt jeder Knoten seine Nachricht über alle auslaufenden Kanten verschickt, schickt j im ersten Schritt seine Nachricht über Kante (j, i) zu i und damit gilt die Aussage für $b(i, j) = 1$. Zur Durchführung des Induktionsschrittes nehmen wir an, dass die Aussage für alle Knoten i mit Nachbar j und $b(i, j) \leq f$, $f > 1$, gilt und zeigen, dass die Aussage dann auch für alle Knoten i und deren Nachbarn j mit $b(i, j) = f + 1$ gilt. Dazu betrachten wir die Kante $e = (j, i)$. Damit über diese Kante $f + 1$ Nachrichten zu Knoten i laufen können, müssen über die in j einlaufenden Kanten $e' \neq e$ genau f Nachrichten laufen, d.h. es muss gelten

$$\sum_{\substack{(k, j) \\ k \neq i}} b(j, k) = b(i, j) - 1 = f.$$

Wir können die Induktionsannahme also auf die in j einlaufenden Kanten (k, j) mit $k \neq i$ anwenden. Daher wird in jedem Schritt q mit $q \leq b(j, k)$ eine Nachricht über Kante (k, j) an Knoten j gesendet, d.h. Knoten j hat zu jedem Zeitpunkt $q \leq f$ mindestens q Nachrichten über die Kanten (k, j) mit $k \neq i$ erhalten. Da j auch seine eigene Nachricht an i schicken muss, schickt j bei Verwendung des obigen Algorithmus zu jedem Zeitpunkt $1, 2, \dots, b(i, j)$ eine Nachricht über Kante (j, i) an i .

Damit ist gezeigt, dass beim Verschicken der Nachrichten über die Kanten eine zeitlich lückenlose Übertragung entsteht. Für die Anzahl der von Knoten i zum Zeitpunkt q empfangenen Nachrichten gilt damit

$$|R(i, q)| = 1 + \sum_{(j, i)} \min(b(i, j), q),$$

d.h. neben der eigenen Nachricht von Knoten i enthält $R(i, q)$ die bis zum Zeitpunkt q über die einlaufenden Kanten (j, i) eingetroffenen Nachrichten. Dabei können über eine Kante (j, i) maximal $b(i, j)$ Nachrichten einlaufen, da über diese Kante nicht mehr Knoten mit i verbunden sind. Außerdem kann über jede Kante (j, i) in jedem Zeitschritt maximal eine Nachricht eintreffen, d.h. zum Zeitpunkt q können über eine Kante maximal q Nachrichten eingetroffen sein. Weiter gilt

$$Q(i) := \max\{b(i, j) \mid j \text{ ist Nachbar von } i\} \leq p - 1, \quad (4.12)$$

d.h. es können maximal $p - 1$ Knoten über j mit i verbunden sein. Für Zeitpunkte $q \geq b(i, j)$ gilt $\min(b(i, j), q) = b(i, j)$. Da jeden Knoten i über seine Kanten $p - 1$ Nachrichten erreichen müssen, gilt weiter $\sum_{(j, i)} b(i, j) = p - 1$. Für den Zeitpunkt $Q(i)$ folgt damit:

$$|R(i, Q(i))| = 1 + \sum_{(j, i)} b(i, j) = p.$$

Da $Q(i) \leq p - 1$ gilt, hat Knoten i die Nachrichten von allen anderen Knoten nach maximal $p - 1$ Schritten erhalten. Dies gilt für einen beliebigen Knoten i , d.h. nach $p - 1$ Schritten ist die Multi-Broadcastoperation ausgeführt.

Ein **Gesamtaustausch** braucht auf einem Baumnetzwerk $\Theta(p^2)$ Schritte. Eine der beiden von der Wurzel des Netzwerkes ausgehenden Kanten teilt die Knoten des Netzwerkes in zwei Teilmengen mit $\lceil p/2 \rceil$ bzw. $\lfloor p/2 \rfloor$ Knoten. Jeder Knoten der einen muss eine Nachricht an jeden Knoten der anderen Teilmenge schicken, d.h. über die Kante müssen in jede Richtung $\lceil p/2 \rceil \lfloor p/2 \rfloor \approx p^2/4$ Nachrichten geschickt werden.

Eine obere Schranke erhalten wir, indem wir einen Ring auf den Baum abbilden. Eine Abbildung eines Ringes mit p Knoten in einen vollständigen binären Baum kann dadurch erfolgen, dass ausgehend von der Wurzel des Baumes die Knoten von links nach rechts in depth-first-Ordnung durchnummeriert werden. Die Kanten, die die Knoten in Ringnummerierung verbinden, benutzen alle Baumkanten, und zwar einmal in jeder Richtung. Die Realisierung des Gesamtaustausches auf einem Ring in Zeit $O(p^2)$ realisiert auch

einen Gesamtaustausch auf dem vollständigen binären Baum. Eine andere Möglichkeit eine obere Schranke zu erhalten ist, obige Realisierung der Multi-Broadcastoperation auszunutzen.

4.4 Analyse von Laufzeitformeln

Die Laufzeitformel eines parallelen Programms beschreibt dessen Ausführungszeit in Abhängigkeit von

- der **Größe der Eingabe** n und evtl. weiterer Charakteristika der Eingabe wie z.B. vorgegebener Iterationszahlen oder Schleifengrenzen,
- der **Anzahl** der ausführenden Prozessoren p und
- **Kommunikationsparametern**, die für den speziellen Parallelrechner und die verwendete Laufzeitbibliothek spezifisch sind.

Für eine feste Eingabegröße n fällt die *Berechnungszeit* üblicherweise mit einer steigenden Zahl p von eingesetzten Prozessoren, da mit wachsender Prozessorzahl jeder Prozessor einen kleineren Teil der Berechnung ausführen muss. Dies sieht man z.B. anhand einer Matrix-Vektor-Multiplikation $\mathbf{A}\mathbf{b} = \mathbf{c}$. Bei zeilenweiser Verteilung von $\mathbf{A}$ und replizierter Verteilung von $\mathbf{c}$ muss bei einer größeren Anzahl von eingesetzten Prozessoren jeder Prozessor weniger Skalarprodukte berechnen, vgl. Abschnitt 3.7.3. Für festes n und wachsendes p hängt die Entwicklung der *Kommunikationszeit* eines parallelen Programms vom Kommunikationsverhalten des Parallelrechners und den vom Programm verwendeten Kommunikationsoperationen ab:

- Die Kommunikationszeit kann mit wachsendem p zurückgehen. Dies ist z.B. dann der Fall, wenn während der Abarbeitung des Programms jeder Prozessor mit einer festen Anzahl von Nachbarprozessoren Daten per Einzeltransfer austauscht, wobei die Größe der auszutauschenden Datenblöcke mit wachsendem p sinkt. Eine solche Situation liegt vor, wenn bei einer iterativen Berechnung auf einem zweidimensionalen Datengitter die Berechnungen an Gitterpunkten Daten von Nachbarpunkten der vorhergehenden Iteration benötigen. Bei blockweiser Schachbrettaufteilung des Datengitters führt dies zu Kommunikation zwischen Nachbarn im Prozessgitter. Bei wachsendem p sinken die Größen der Blöcke und damit die Anzahl der Gitterpunkte zum Nachbarblock, die versendet werden müssen.
- Die Kommunikationszeit kann mit wachsendem p ansteigen. Dies ist z.B. dann der Fall, wenn die Kommunikationszeit von globalen Kommunikationsoperationen dominiert wird, da deren Ausführungszeit mit p (logarithmisch oder linear) ansteigt.

Im zweiten Fall kann somit die Situation auftreten, dass mit wachsendem p der Anstieg der Kommunikationszeit die Reduktion in der Berechnungszeit

überwiegt, so dass sich für eine feste Eingabegröße ab einer bestimmten Anzahl p_{max} von Prozessoren der Einsatz weiterer Prozessoren nicht mehr lohnt, da die Gesamtlaufzeit des Programms für $p > p_{max}$ ansteigt. Ein solches p_{max} wäre also die Grenze der Skalierbarkeit des betreffenden Programms, bei Verwendung eines Skalierbarkeitsbegriffs ohne Einbeziehung der Problemgröße, vgl. Abschnitt 4.2.

4.4.1 Paralleles Skalarprodukt

Als Beispiel betrachten wir die Berechnung des Skalarproduktes zweier Vektoren $\mathbf{a}, \mathbf{b} \in \mathbb{R}^n$, wobei wir der Einfachheit halber annehmen, dass die Systemgröße n ein Vielfaches der Anzahl der Prozessoren ist, d.h. es ist $n = r \cdot p$ mit $r \in \mathbb{N}$. Weiter nehmen wir an, dass die beiden Vektoren blockweise auf die Prozessoren verteilt sind, d.h. der Prozessor P_k mit $1 \leq k \leq p$ speichert die Elemente a_j und b_j mit $r \cdot (k-1) + 1 \leq j \leq r \cdot k$ und berechnet

$$c_k = \sum_{j=r \cdot (k-1)+1}^{r \cdot k} a_j \cdot b_j .$$

Die von den verschiedenen Prozessoren errechneten Teilergebnisse c_k werden danach mit einer Einzel-Akkumulationsoperation bei einem Prozessor P_i aufgesammelt. Die Laufzeit der Akkumulationsoperation hängt dabei vom verwendeten Netzwerk ab. Wir führen im Folgenden eine netzwerkbasierte Analyse durch, bei der wir annehmen, dass das Verschieben eines Floating-Point-Wertes zwischen zwei im Netzwerk direkt miteinander verbundenen Prozessoren β Zeiteinheiten braucht. Diese Zeit kann sich zusammensetzen aus einer Startupzeit und einer Transferzeit. Weiterhin nehmen wir an, dass die Durchführung einer arithmetischen Operation α Zeiteinheiten benötigt.

Lineares Feld als Verbindungsnetzwerk. Wenn die Prozessoren in einem linearen Feld angeordnet sind, ist der mittlere Knoten des linearen Feldes der beste Knoten zum Aufsammeln des Ergebnisses, da dieser Knoten die maximale Entfernung zwischen dem aufsammelnden Knoten und den versendenden Knoten minimiert, vgl. auch Abschnitt 4.3.1. Die Akkumulationsoperation findet so statt, dass jeder Knoten P_k auf das Eintreffen eines Wertes vom linken (falls $k \leq p/2$) bzw. rechten (falls $k > p/2$) Nachbarn (falls vorhanden) wartet, diesen Wert zu seinem lokalen Teilergebnis addiert und die Summe zum rechten bzw. linken Nachbarn weiterschickt. Damit dauert jedes Weiterleiten $\alpha + \beta$ Zeiteinheiten.

Unter Verwendung des mittleren Knotens $P_{p/2}$ zum Aufsammeln kann die Zeit für die Berechnung der lokalen Teilergebnisse c_k und für das Aufsammeln dieser Teilergebnisse durch die Laufzeitformel

$$T(p, n) = 2 \frac{n}{p} \alpha + \frac{p}{2} (\alpha + \beta). \quad (4.13)$$

beschrieben werden. Dabei ist $2n/p \cdot \alpha$ die Zeit für die Ausführung der lokalen Berechnungen, $p/2(\alpha + \beta)$ ist die Zeit für die Akkumulationsoperation.

Um die Anzahl p der Prozessoren zu bestimmen, die die Laufzeit für eine gegebene, feste Vektorlänge n minimiert, betrachten wir die Laufzeit als Funktion von p , d.h. $T(p) \equiv T(p, n)$, und bilden die erste Ableitung $T'(p)$. Es gilt:

$$T'(p) = -\frac{2n\alpha}{p^2} + \frac{\alpha + \beta}{2},$$

Wenn wir T als reellwertige Funktion in p betrachten, ergibt sich

$$T'(p) = 0 \quad \text{für} \quad p = \pm \sqrt{\frac{4n\alpha}{\alpha + \beta}}.$$

Für $p = +\sqrt{4n\alpha/(\alpha + \beta)}$ gilt $T''(p) > 0$, es liegt also ein Minimum von T vor. Die optimale Anzahl der eingesetzten Prozessoren wächst also mit $\sqrt{n}$. Insbesondere folgt, dass es für $\beta > (4n - 1)\alpha$ besser ist, das Skalarprodukt auf nur einem Prozessor zu berechnen.

Hyperwürfel als Verbindungsnetzwerk. Auf einem Hyperwürfel als Verbindungsnetzwerk kann die Einzel-Akkumulationsoperation in $\log p$ Schritten durchgeführt werden, die von den Blättern des aufgespannenden Baumes zu dessen Wurzel voranschreiten, vgl. Abschnitt 4.3.1. Jeder Schritt besteht aus dem Empfangen eines Wertes von den beiden Kindknoten (falls vorhanden), der Addition dieser Werte zu dem lokalen Teilergebnis und dem Weiter-schicken zum Elternknoten. Da die Sendeoperationen einer Stufe unabhängig voneinander durchgeführt werden können, dauert jeder Schritt $\alpha + \beta$ Zeiteinheiten, wenn wir die Zeit für den Datentransfer jeweils dem sendenden Knoten zuordnen. Die Gesamtzeit für die Berechnung des Skalarproduktes wird damit durch folgende Laufzeitformel

$$T(n, p) = \frac{2n\alpha}{p} + \log p \cdot (\alpha + \beta)$$

beschrieben. Für eine feste Vektorlänge n erhält man die optimale Anzahl von Prozessoren wieder über die Ableitung von $T(p) \equiv T(n, p)$, für die wegen $\log p = \ln p / \ln 2$ unter Verwendung des natürlichen Logarithmus $\ln$ gilt:

$$T'(p) = -\frac{2n\alpha}{p^2} + (\alpha + \beta) \frac{1}{p \ln 2}.$$

Aus $T'(p) = 0$ folgt als notwendige Bedingung für ein Minimum

$$p = \frac{2n\alpha \ln 2}{\alpha + \beta}.$$

Für die zweite Ableitung gilt an dieser Stelle $T''(p) > 0$, es liegt also ein Minimum vor. Für einen Hyperwürfel wächst die optimale Anzahl von einzusetzenden Prozessoren also linear in n , also wesentlich schneller als für ein lineares Feld. Dies ist intuitiv klar, da die Akkumulationsoperation in einem Hyperwürfel schneller ausgeführt werden kann als in einem linearen Feld.

4.4.2 Parallele Matrix-Vektor-Multiplikation

Wir betrachten die Multiplikation einer Matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ mit einem Vektor $\mathbf{b} \in \mathbb{R}^n$. Zur Vereinfachung nehmen wir an, dass die Anzahl der Zeilen der Matrix ein Vielfaches der Anzahl der Prozessoren ist, d.h. $n = r \cdot p$. Zur parallelen Berechnung von $\mathbf{A} \cdot \mathbf{b} = \mathbf{c}$ kann die Matrix $\mathbf{A}$ zeilenweise oder spaltenweise auf die Prozessoren verteilt sein, was entsprechend eine verteilte Berechnung der Skalarprodukte oder eine verteilte Berechnung der Linearkombination nahelegt, vgl. Abschnitt 3.7.3. Wir betrachten zuerst eine *zeilenorientierte streifenweise Datenverteilung*, d.h. wir nehmen an, dass Prozessor P_k die Zeilen i mit $r \cdot (k - 1) + 1 \leq i \leq r \cdot k$ von $\mathbf{A}$ und den gesamten Vektor $\mathbf{b}$ (repliziert) speichert. Prozessor P_k berechnet durch Bildung von r Skalarprodukten die Elemente c_i mit $(k - 1) \cdot r + 1 \leq i \leq k \cdot r$ des Ergebnisvektors $\mathbf{c}$ ohne Kommunikation

$$c_i = \sum_{j=1}^n a_{ij} \cdot b_j .$$

Wenn wir annehmen, dass nach dieser verteilten Berechnung der Ergebnisvektor wieder repliziert vorliegen soll, damit z.B. bei einem Iterationsverfahren für lineare Gleichungssysteme mit dem Ergebnisvektor die nächste Iteration durchgeführt werden kann, muss nach der Berechnung der Teilvektoren eine Multi-Broadcastoperation durchgeführt werden, die jedem Prozessor jeden Teilvektor verfügbar macht. Zu dieser Operation trägt jeder Prozessor r Elemente bei.

Bei einer *spaltenorientierten streifenweisen Datenverteilung* der Matrix speichert jeder Prozessor P_k die Spalten j mit $r \cdot (k - 1) + 1 \leq j \leq r \cdot k$ von $\mathbf{A}$ und die korrespondierenden Elemente des Vektors $\mathbf{b}$. Prozessor P_k berechnet n Teilsummen $d_{k1}, \dots, d_{kn}$ mit

$$d_{kj} = \sum_{l=r \cdot (k-1)+1}^{r \cdot k} a_{jl} b_l ,$$

die dann mit einer Multi-Akkumulationsoperation aufgesammelt werden, wobei eine Addition als Reduktionsoperation verwendet wird. Bei der Akkumulation sammelt Prozessor P_k die Summe der Werte $d_{1j}, \dots, d_{nj}$ für $(k - 1) \cdot r + 1 \leq j \leq k \cdot r$ auf, d.h. jeder Prozessor führt eine Akkumulation mit Blöcken der Größe r durch. Nach Durchführung der Multi-Akkumulationsoperation hat jeder Prozessor die gleichen Elemente des Ergebnisvektors $\mathbf{c}$, die er vom Eingabevektor $\mathbf{b}$ hatte, d.h. auch diese Variante ist für die Durchführung eines Iterationsverfahrens geeignet.

Zum Vergleich der beiden Parallelisierungen und der daraus resultierenden Berechnungen und Kommunikationsoperationen beobachtet man, dass bei beiden Verfahren jeder Prozessor die gleiche Anzahl von lokalen Berechnungen ausführt, nämlich $n \cdot r$ Multiplikationen und die gleiche Anzahl von

Additionen, d.h. bei paralleler Ausführung resultieren $2nr = 2n^2/p$ Operationen. Darüberhinaus sind eine Multi-Broadcastoperation mit r Elementen und eine Multi-Akkumulationsoperation mit r Elementen *duale Operationen*, d.h. auch die Kommunikationszeiten sind asymptotisch betrachtet identisch. Es ist also zu erwarten, dass beide Verfahren zu ähnlichen Gesamtlaufzeiten führen werden.

Zur Berechnung der für ein festes n optimalen Anzahl von Prozessoren nehmen wir wieder an, dass das Ausführen einer arithmetischen Operation α Zeiteinheiten braucht. Darüberhinaus nehmen wir an, dass das Verschicken von r Floating-Point-Werten zwischen zwei im gegebenen Netzwerk direkt miteinander verbundenen Prozessoren $\beta + r \cdot \gamma$ Zeiteinheiten braucht. Eine Überlappung zwischen Berechnung und Kommunikation wird nicht angenommen.

Lineares Feld als Verbindungsnetzwerk. Sind die Prozessoren in einem linearen Feld angeordnet, braucht man zur Durchführung einer Multi-Broadcastoperation (und analog für eine Multi-Akkumulationsoperation) p Schritte, wobei jeder Schritt $\beta + r \cdot \gamma$ Zeiteinheiten braucht. Die Gesamtzeit der Durchführung der Matrix-Vektor-Multiplikation wird also für die beiden oben erwähnten Ablageformeln durch die folgende Laufzeitformel beschrieben

$$T(n, p) = \frac{2n^2}{p} \alpha + p \cdot \left(\beta + \frac{n}{p} \cdot \gamma \right) = \frac{2n^2}{p} \alpha + p \cdot \beta + n \cdot \gamma.$$

Für die Ableitung von $T(p) \equiv T(n, p)$ gilt:

$$T'(p) = -\frac{2n^2 \alpha}{p^2} + \beta.$$

Also ist $T'(p) = 0$ für $p = \sqrt{2\alpha n^2 / \beta} = n \cdot \sqrt{2\alpha / \beta}$. Wegen $T''(p) = 4\alpha n^2 / p^3$ gilt auch $T''(n \sqrt{2\alpha / \beta}) > 0$, d.h. an der Stelle $p = n \sqrt{2\alpha / \beta}$ liegt ein Minimum vor. Die optimale Anzahl von Prozessoren wächst also *linear* in n .

Hyperwürfel als Verbindungsnetzwerk. Wenn ein Hyperwürfel als Verbindungsnetzwerk eingesetzt wird, braucht die Durchführung einer Multi-Broadcastoperation $p / \log p$ Schritte, vgl. Abschnitt 4.3, wobei jeder Schritt $\beta + r \cdot \gamma$ Zeiteinheiten braucht. Die Gesamtlaufzeit der Matrix-Vektor-Multiplikation wird damit durch folgende Laufzeitformel

$$\begin{aligned} T(n, p) &= \frac{2\alpha n^2}{p} + \frac{p}{\log p} (\beta + r \cdot \gamma) \\ &= \frac{2\alpha n^2}{p} + \frac{p}{\log p} \cdot \beta + \frac{\gamma n}{\log p} \end{aligned}$$

beschrieben. Für die Ableitung von $T(p) \equiv T(n, p)$ gilt:

$$T'(p) = -\frac{2\alpha n^2}{p^2} + \frac{\beta}{\log p} - \frac{\beta}{\log^2 p \ln 2} - \frac{\gamma n}{p \cdot \log^2 p \ln 2}.$$

$T'(p) = 0$ wird für

$$-2\alpha n^2 \log^2 p + \beta p^2 \log p - \beta p^2 \frac{1}{\ln 2} - \gamma np \frac{1}{\ln 2}$$

erfüllt. Diese Gleichung lässt sich nicht mehr analytisch lösen, d.h. die optimale Anzahl der einzusetzenden Prozessoren kann nicht durch einen Ausdruck in geschlossener Form angegeben werden. Dies ist eine typische Situation für die Analyse von Laufzeitformeln. Daher geht man zu Approximationen über. Für den Hyperwürfel und andere Netzwerke, in die ein lineares Feld eingebettet werden kann, kann als Laufzeitformel die des linearen Feldes benutzt werden, da die Matrix-Vektor-Multiplikation mindestens so schnell wie auf dem linearen Feld ausgeführt wird. Approximativ gelten also die obigen Analysen auch für den Hyperwürfel.

4.5 Parallele Berechnungsmodelle

Ein Berechnungsmodell eines Rechnersystems beschreibt auf einer von Hardware und Technologie abstrahierenden Ebene, welche Basisoperationen von dem Rechnersystem ausgeführt werden können, wann die damit verbundenen Aktionen stattfinden, wie auf Daten zugegriffen werden kann und wie Daten gespeichert werden können [11]. Berechnungsmodelle werden verwendet, um Algorithmen vor der Realisierung in einer speziellen Programmiersprache und unabhängig vom Einsatz eines speziellen Rechners zu bewerten. Dazu ist es notwendig, ein Modell eines Computers zugrunde zu legen, das von vielen Details spezieller Rechner abstrahiert, aber die wichtigsten Charakteristika einer breiten Klasse von Rechnern erfasst. Dazu gehören insbesondere alle Charakteristika, die signifikanten Einfluss auf die Laufzeit der Algorithmen haben. Zur Bewertung eines Algorithmus wird seine Ausführung auf dem Rechnermodell hinsichtlich des gewählten Bewertungskriteriums untersucht. Dazu gehören insbesondere die Laufzeit und der Speicherplatzverbrauch in Abhängigkeit von der Eingabegröße. Wir geben im Folgenden einen kurzen Überblick über häufig verwendete parallele Berechnungsmodelle, und zwar das PRAM-Modell, das BSP-Modell und das LogP-Modell.

4.5.1 PRAM-Modelle

Für die theoretische Analyse von sequentiellen Algorithmen hat sich das RAM-Modell weitgehend durchgesetzt. Obwohl das RAM-Modell von vielen Details realer Rechner abstrahiert (z.B. von endlicher Speichergröße, evtl. vorhandenen Caches, komplexen Adressierungsarten oder mehreren Funktionseinheiten), sind die mit Hilfe des Modells durchgeführten Laufzeitanalysen in gewissen Grenzen aussagekräftig.

Zur Analyse von parallelen Algorithmen wurde das **PRAM-Modell** (**p**arallel **r**andom **a**ccess **m**achine) als Erweiterung des RAM-Modells eingeführt [48, 87]. Ein PRAM-Rechner besteht aus einer unbeschränkten An-

zahl von RAM-Rechnern (Prozessoren), die von einer globalen Uhr gesteuert synchron zueinander das gleiche Programm ausführen. Neben den lokalen Speichern der Rechner gibt es einen globalen Speicher unbeschränkter Größe, in dem jeder Prozessor auf jede beliebige Speicherzelle in der gleichen Zeit zugreifen kann, die für die Durchführung einer arithmetischen Operation gebraucht wird (uniforme Zugriffszeit). Die Kommunikation zwischen den Prozessoren erfolgt über den globalen Speicher. Da jeder Prozessor auf jede Speicherzelle des globalen Speichers zugreifen kann, können Speicherzugriffskonflikte auftreten, wenn mehrere Prozessoren versuchen, die gleiche Speicherzelle zu lesen oder zu beschreiben. Es gibt mehrere Varianten des PRAM-Modells, die sich in der Behandlung von Speicherzugriffskonflikten unterscheiden. Die *EREW-PRAM* (*exclusive read, exclusive write*) verbietet simultane Zugriffe auf die gleiche Speicherzelle. Die *CREW-PRAM* (*concurrent read, exclusive write*) erlaubt simultane Lesezugriffe, verbietet aber das simultane Beschreiben. Die *ERCW-PRAM* (*exclusive read, concurrent write*) erlaubt das simultane Beschreiben, verbietet aber simultane Lesezugriffe. Die *CRCW-PRAM* (*concurrent read, concurrent write*) erlaubt sowohl simultane Lese- als auch Schreibzugriffe. Bei simultanen Schreibzugriffen muss festgelegt werden, was beim simultanen Schreiben mehrerer Prozessoren auf die gleiche Speicherzelle passiert. Dafür wurden verschiedene Varianten vorgeschlagen: (1) gemeinsames Schreiben ist nur erlaubt, wenn alle Prozessoren den gleichen Wert schreiben; (2) beim gemeinsamen Schreiben gewinnt ein beliebiger Prozessor; (3) beim gemeinsamen Schreiben wird die Summe der Werte der einzelnen Prozessoren in die Speicherzelle geschrieben; (4) den Prozessoren werden Prioritäten zugeordnet und es gewinnt der Prozessor mit der höchsten Priorität.

Die Kosten eines Algorithmus werden als Anzahl der PRAM-Schritte angegeben, wobei jeder PRAM-Schritt aus dem Lesen von Daten aus dem gemeinsamen Speicher, einem Berechnungsschritt und dem Schreiben in den gemeinsamen Speicher besteht. Angegeben werden die Kosten meist als asymptotische Laufzeiten in Abhängigkeit von der Problemgröße. Da die Anzahl der Prozessoren als unbeschränkt angenommen wurde, spielt sie bei der Kostenberechnung also keine Rolle.

Eine PRAM-Modell wurde durch die SB-PRAM als realer Rechner realisiert. Neben den üblichen Lese- und Schreibbefehlen stellt die SB-PRAM Hardwareunterstützung für Multipräfixoperationen zur Verfügung. Wir betrachten eine MPADD-Operation als Beispiel. Die MPADD-Operation arbeitet auf einer Speicherzelle s des gemeinsamen Speichers, die mit dem Wert o vorbesetzt sei. Jeder der an der Operation beteiligten Prozessoren P_i , $i = 1 \dots, n$, stellt einen Wert o_i für die Operation zur Verfügung. Die synchrone Ausführung der Operation bewirkt, daß jeder Prozessor P_j den Wert

$$o + \sum_{i=1}^{j-1} o_i$$

erhält, die Speicherzelle s wird mit dem Wert $o + \sum_{i=1}^n o_i$ besetzt. Jede Multiplikationsoperation erfordert für ihre Ausführung zwei Zyklen, unabhängig von der Anzahl der beteiligten Prozessoren. Diese Operationen können daher für eine effiziente Implementierung von Synchronisationsmechanismen und parallelen Datenstrukturen verwendet werden, auf die verschiedene Prozessoren parallel zueinander zugreifen können, ohne daß dabei zeitkritische Abläufe entstehen [67]. Damit kann auch ein paralleler Taskpool realisiert werden, der die Grundlage einer effizienten Implementierung verschiedener, auch irregulärer Anwendungen bildet [67, 91, 125, 134]. Ein Beispiel ist die in Abschnitt 7.4 beschriebene Cholesky-Zerlegung für dünnbesetzte Matrizen. .

Obwohl das PRAM-Modell für die theoretische Analyse paralleler Algorithmen oft angewendet wird, ist es für die Vorhersage von realistischen Laufzeiten für reale Rechner oft ungeeignet. Der Hauptgrund dafür liegt darin, dass die Annahme der uniformen Zugriffszeit auf den globalen Speicher eine zu vereinfachende Annahme ist, da reale Rechner meist sehr hohe Verzögerungszeiten für Zugriffe auf den globalen Speicher oder die Speicher von anderen Prozessoren haben, während lokale Speicherzugriffe recht schnell ausgeführt werden können. Die für die meisten Rechner vorhandene Speicherhierarchie wird ebenfalls nicht berücksichtigt. Durch diese vereinfachenden Annahmen ist das PRAM-Modell auch nicht in der Lage, Algorithmen mit großer Lokalität gegenüber Algorithmen mit geringer Lokalität positiv zu bewerten. Weitere unrealistische Annahmen sind die synchrone Arbeitsweise der Prozessoren und das Fehlen von Kollisionen beim Zugriff auf den globalen Speicher. Wegen dieser Nachteile sind mehrere Erweiterungen des PRAM-Modells vorgeschlagen worden.

Das Fehlen von Synchronität versucht die in [58] vorgeschlagene *Phasen-PRAM* dadurch nachzubilden, dass die durchgeführten Berechnungen in Phasen eingeteilt werden und die Prozessoren innerhalb einer Phase asynchron arbeiten. Erst am Ende einer Phase wird synchronisiert. Die *Verzögerungs-PRAM* (engl. *delay PRAM*) [120] versucht Verzögerungszeiten der Speicherzugriffe dadurch zu modellieren, dass eine Kommunikationsverzögerung zwischen der Produktion eines Datums durch einen Prozessor und dem Zeitpunkt, zu dem ein anderer Prozessor das Datum benutzen kann, eingeführt wird. Ein ähnlicher Ansatz wird bei der *Local-Memory-PRAM* und *Block-PRAM* [3, 4] verwendet. Bei der Block-PRAM wird ein Zugriff auf den globalen Speicher mit der Zeit $l + b$ bewertet, wobei l die Startupzeit darstellt und b die Größe des adressierten Speicherbereiches angibt. Einen genaueren Überblick über die wichtigsten PRAM-Varianten findet man z.B. in [6, 24].

4.5.2 BSP-Modell

Keines der vorgeschlagenen PRAM-Modelle kann das Verhalten von realen Parallelrechnern für einen breiten Anwendungsbereich zufriedenstellend vorhersagen, z.T. auch deshalb, weil immer wieder Parallelrechner mit neuen Architekturen entwickelt werden. Um zu verhindern, dass die Modellbildung

ständig hinter der Architektur-Entwicklung zurückbleibt, wurde das BSP-Modell (*bulk synchronous parallel*) als Brücke zwischen Softwareentwicklern und Hardwareherstellern vorgeschlagen [161]. Die Idee besteht darin, dass die Architektur von Parallelrechnern dem BSP-Modell entsprechen soll und dass Softwareentwickler sich auf ein vorgegebenes Verhalten der Hardware verlassen können. Damit könnten Hardware- und Softwareentwicklung voneinander entkoppelt werden und entwickelte Softwareprodukte bräuchten nicht ständig zur Erhöhung der Effizienz an neue Hardwaredetails angepasst werden.

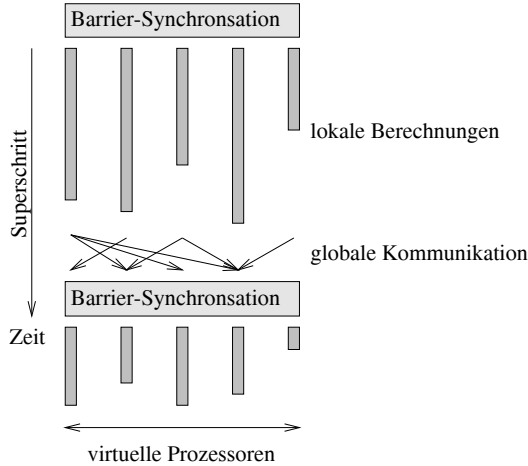


Abb. 4.8. Berechnungen im BSP-Modell werden in Superschritten ausgeführt, wobei jeder Superschritt aus drei Phasen besteht: (1) simultane lokale Berechnungen jedes Prozesses, (2) Kommunikationsoperationen zum Austausch von Daten zwischen Prozessen, (3) eine Barrier-Synchronisation, die die Kommunikationsoperationen abschließt und die versendeten Daten für die empfangenden Prozesse sichtbar macht. Das in der Abbildung dargestellte Kommunikationsmuster der Kommunikationsphase stellt eine 3-Relation dar.

Das BSP-Modell ist eine Abstraktion eines Rechners mit physikalisch verteiltem Speicher, die die stattfindende Kommunikation zu Bündeln zusammenfasst anstatt sie als einzelne Punkt-zu-Punkt-Transfers darzustellen. Ein BSP-Modellrechner besteht aus einer Anzahl von *Berechnungseinheiten* (Prozessoren), von denen jede mit einem Speicher ausgestattet sein kann, einem *Verbindungsnetzwerk* (Router), mit dessen Hilfe Punkt-zu-Punkt-Nachrichten zwischen Berechnungseinheiten versendet werden können, und einem *Synchronisationsmechanismus*, mit dessen Hilfe alle oder eine Teilmenge der Berechnungseinheiten jeweils nach Ablauf von L Zeiteinheiten synchronisiert werden können. Eine Berechnung des BSP-Modellrechners besteht aus einer Folge von **Superschritten**, die in Abbildung 4.8 schematisch dargestellt sind. In jedem Superschritt führt jede Berechnungseinheit lokale Berechnungen durch und kann an Kommunikationsoperationen

(send/receive) teilnehmen. Eine lokale Berechnung kann in einer Zeiteinheit durchgeführt werden. Der Effekt einer Kommunikationsoperation wird erst im nächsten Superschritt wirksam, d.h. die verschickten Daten können erst im nächsten Superschritt vom Empfänger benutzt werden. Am Ende jedes Superschlittes findet eine **Barrier-Synchronisation** statt, die mit Hilfe des Synchronisationsmechanismus durchgeführt wird. Da der Synchronisationsmechanismus maximal alle L Zeiteinheiten synchronisieren kann, dauert ein Superschritt mindestens L Zeiteinheiten. Die Größe von L bestimmt somit die Granularität der Berechnung. Das BSP-Modell sieht vor, dass die Größe von L dynamisch während des Programmlaufes verändert werden kann, obwohl von der Hardware eine Untergrenze für L vorgegeben sein kann. Das Verbindungsnetzwerk bzw. der Router kann in einem Superschritt beliebige **h -Relationen** realisieren. Dabei beschreibt eine h -Relation ein Kommunikationsmuster, in dem jede Berechnungseinheit maximal h Nachrichten versenden oder empfangen kann. Eine Berechnung auf einem BSP-Modellrechner kann durch vier Parameter charakterisiert werden [79]:

- p : die Anzahl der Prozesse (virtuelle Prozessoren), die innerhalb der Superschlittes für die Berechnungen verwendet werden,
- s : die Berechnungsgeschwindigkeit der Berechnungseinheiten, beschrieben durch die Anzahl der Berechnungsschlittes, die eine Berechnungseinheit pro Sekunde durchführen kann, wobei in jedem Berechnungsschritt eine arithmetische Operation mit lokalen Daten ausgeführt werden kann,
- l : die Anzahl der Schlittes, die für die Ausführung einer Barrier-Synchronisation notwendig sind,
- g : die Anzahl der Schlittes, die im Mittel für den Transfer eines Wortes im Rahmen einer h -Relation gebraucht wird.

Der Parameter g wird so bestimmt, dass das Ausführen einer h -Relation mit m Worten pro Nachricht $l \cdot m \cdot g$ Schlittes benötigt. Für einen realen Parallelrechner hängt der Wert von g von der Bisektionsbandbreite des Verbindungsnetzwerkes ab, vgl. S. 35, er wird aber auch vom verwendeten Kommunikationsprotokoll und der Implementierung der verwendeten Kommunikationsbibliothek mitbestimmt. Der Wert von l wird vom Durchmesser des Verbindungsnetzwerkes beeinflusst, hängt aber ebenfalls von der Implementierung der Kommunikationsbibliothek ab. Beide Parameter werden durch geeignete Benchmarkprogramme empirisch bestimmt. Da der Wert von s zur Normalisierung der Werte von l und g verwendet wird, sind nur p , l und g unabhängige Parameter. Alternativ können l und g ebenso wie s als Anzahl der Maschinenzyklen oder in μs angegeben werden.

Die Ausführungszeit eines BSP-Programmes ergibt sich als Summe der Ausführungszeiten der Superschlittes, aus denen das BSP-Programm besteht. Die Ausführungszeit eines Superschlittes $T_{Superschritt}$ ergibt sich als Summe von drei Termen: (1) das Maximum der Dauer w_i der lokalen Berechnungen jedes Prozesses i , (2) die Kosten der globalen Kommunikation zur Realisie-

rung einer h -Relation und (3) die Kosten für die Barrier-Synchronisation zum Abschluss des Superschrittes:

$$T_{\text{Superschritt}} = \max_{\text{Prozesse}} w_i + h \cdot g + l.$$

Das BSP-Modell ist ein Berechnungsmodell, das mehreren Programmiermodellen zugrundegelegt werden kann. Zur Erleichterung der Programmierung innerhalb des BSP-Modells und zur Erstellung von effizienten Programmen wurde eine BSPLib-Bibliothek entwickelt [64, 79], die Operationen zur Initialisierung einer Superschritts, zur Durchführung von Kommunikationsoperationen und zur Teilnahme an Barrier-Synchronisationen bereitstellt.

4.5.3 LogP-Modell

Als Kritikpunkte am BSP-Modell werden in [30] folgende Punkte angeführt: Die Länge der Superschritte muss groß genug sein, um beliebige h -Relationen zu realisieren, d.h. die Granularität kann nicht unter einen bestimmten Wert gesenkt werden. Außerdem sind die innerhalb eines Superschrittes verschickten Nachrichten erst im nächsten Superschritt verfügbar, auch wenn die Übertragungsgeschwindigkeit des Netzwerkes ein Zustellen innerhalb des Superschrittes erlauben würde. Ein weiterer Kritikpunkt besteht darin, dass das BSP-Modell eine zusätzliche Hardware-Unterstützung zur Synchronisation am Ende jedes Superschrittes erwartet, obwohl eine solche Unterstützung auf den meisten existierenden Parallelrechnern nicht zur Verfügung steht. Wegen dieser Kritikpunkte wurde das BSP-Modell zum *LogP-Modell* erweitert [30], das näher an die Hardware heutiger paralleler Maschinen angelehnt ist.

Ebenso wie das BSP-Modell geht das LogP-Modell davon aus, dass ein Parallelrechner aus einer Anzahl von Prozessoren mit lokalem Speicher besteht, die durch Verschicken von Punkt-zu-Punkt-Nachrichten über ein Verbindungsnetzwerk miteinander kommunizieren können, d.h. auch das LogP-Modell ist für die Modellierung von Rechnern mit physikalisch verteiltem Speicher gedacht. Das Kommunikationsverhalten wird durch vier Parameter beschrieben, die dem Modell seinen Namen gegeben haben:

- L (*latency*) ist eine obere Grenze für die Latenz des Netzwerkes, d.h. für die auftretende zeitliche Verzögerung beim Verschicken einer kleinen Nachricht;
- o (*overhead*) beschreibt die Zeit für den Verwaltungsaufwand eines Prozessors beim Abschicken oder Empfangen einer Nachricht, d.h. o ist die Zeit, während der der Prozessor keine anderen Berechnungen durchführen kann;
- g (*gap*) bezeichnet die minimale Zeitspanne, die zwischen dem Senden oder Empfangen aufeinanderfolgender Nachrichten vergehen muss;
- P (*processors*) gibt die Anzahl der Prozessoren der parallelen Maschine an.

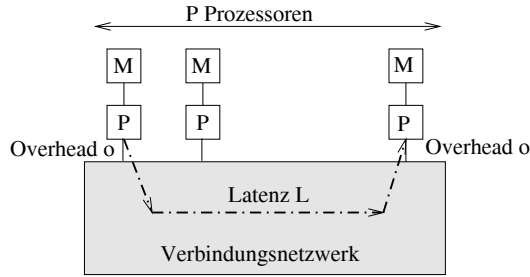


Abb. 4.9. Veranschaulichung der Parameter des LogP-Modells.

Abbildung 4.9 zeigt eine Veranschaulichung der Parameter [29]. Außer P werden alle Parameter entweder in Zeiteinheiten oder Vielfachen des Maschinenzklus gemessen. Vom Netzwerk wird eine endliche Kapazität angenommen: zwischen zwei beliebigen Prozessoren dürfen maximal $\lceil L/g \rceil$ Nachrichten unterwegs sein. Wenn ein Prozessor versucht, eine Nachricht abzusenden, die diese Obergrenze überschreiten würde, wird er blockiert, bis er die Nachricht ohne Limitüberschreitung senden kann. Das LogP-Modell nimmt an, dass kleine Nachrichten verschickt werden, die eine vorgegebene Größe nicht überschreiten. Größere Nachrichten müssen in mehrere kleine Nachrichten zerlegt werden. Die Prozessoren arbeiten asynchron. Die Latenz einer einzelnen Nachricht ist nicht vorhersagbar, ist aber nach oben durch L beschränkt. Dies bedeutet insbesondere, dass nicht ausgeschlossen wird, dass Nachrichten sich überholen können. Die Werte der Parameter L , o und g hängen neben den Hardwareeigenschaften des Netzwerkes von der verwendeten Kommunikationsbibliothek und dem darunterliegenden Kommunikationsprotokoll ab.

Die Laufzeit eines Algorithmus im LogP-Modell wird durch das Maximum der Laufzeiten der einzelnen Prozessoren bestimmt. Als Folgerung aus dem LogP-Modell ergibt sich, dass der Zugriff auf ein Datenelement im Speicher eines anderen Prozessors $2L + 4o$ Zeiteinheiten kostet, wobei jeweils die Hälfte auf den Hin- bzw. Rücktransport entfällt. Eine Folge von n Nachrichten kann in der Zeit $L + 2o + (n - 1)g$ zugestellt werden, vgl. Abbildung 4.10.

Nachteile des LogP-Modells bestehen darin, dass nur kleine Nachrichten vorgesehen sind und dass nur Punkt-zu-Punkt-Nachrichten erlaubt sind. Komplexere Kommunikationsmuster müssen aus Punkt-zu-Punkt-Nachrichten zusammengesetzt werden. Um den Nachteil der Beschränkung auf kurze Nachrichten aufzuheben, wurde das LogP-Modell zum LogGP-Modell erweitert [7], das einen zusätzlichen Parameter G (*Gap per Byte*) enthält, der angibt, welche Zeit bei langen Nachrichten pro Byte beim Verschicken einer Nachricht aufgewendet werden muss. $1/G$ ist die Bandbreite pro Prozessor. Die Zeit für das Verschicken einer Nachricht mit n Byte braucht Zeit $o + (n - 1)G + L + o$, vgl. Abbildung 4.11.

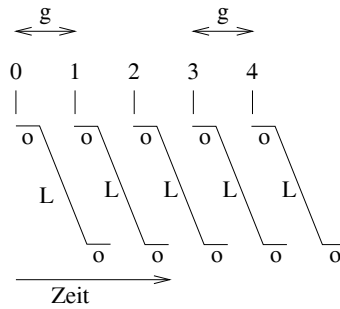


Abb. 4.10. Übertragung einer Nachricht in n Teilnachrichten mit Hilfe des LogP-Modells: Die letzte Teilnachricht wird zum Zeitpunkt $(n-1) \cdot g$ abgeschickt und erreicht das Ziel $2o + L$ Zeiteinheiten später.

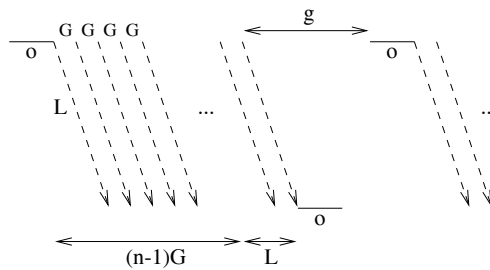


Abb. 4.11. Veranschaulichung der Übertragung einer Nachricht mit n Bytes mit Hilfe des LogGP-Modells: Das letzte Byte der Nachricht wird zum Zeitpunkt $o + (n-1) \cdot G$ abgeschickt und erreicht das Ziel $L + o$ Zeiteinheiten später. Zwischen dem Abschieken des letzten Byte einer Nachricht und dem Start des Abschiekens der nächsten Nachricht müssen mindestens g Zeiteinheiten vergehen.

5. Message-Passing-Programmierung

Das Message-Passing-Programmiermodell ist eine Abstraktion eines Parallelrechners mit verteiltem Speicher, wobei meist *keine* explizite Sicht auf die Topologie des Rechners genutzt wird, um die Portabilität der Programme zu gewährleisten. Ein Message-Passing-Programm besteht aus einer Anzahl von Prozessen mit zugeordneten lokalen Daten. Jeder Prozess kann auf seine lokalen Daten zugreifen und mit anderen Prozessen Informationen durch das explizite Verschicken von Nachrichten austauschen. Im Prinzip kann jeder Prozess ein separates Programm ausführen (MPMD, *multiple program multiple data*). Um die Programmierung zu erleichtern, führt aber im Regelfall jeder Prozess das gleiche Programm aus (SPMD), vgl. Abschnitt 3.4. Dies stellt in der Praxis keine Einschränkung für die Programmierung dar, da jeder Prozess in Abhängigkeit von seiner Prozessnummer einen vollständig unterschiedlichen Programmteil ausführen kann.

Die Prozesse eines Message-Passing-Programms können Daten aus ihren lokalen Speichern untereinander austauschen. Dazu werden Kommunikationsoperationen verwendet, die dem Programmierer in Form einer Programmbibliothek zur Verfügung gestellt werden und zu deren Ausführung die beteiligten Prozesse eine entsprechende Kommunikationsanweisung aufrufen müssen, d.h. alle Kommunikationsoperationen müssen in Message-Passing-Programmen explizit angegeben werden. Üblicherweise umfassen Kommunikationsbibliotheken neben Punkt-zu-Punkt-Kommunikationsoperationen auch globale Kommunikationsoperationen, an denen mehrere Prozesse beteiligt sind und die zur Realisierung regelmäßiger Kommunikationsmuster geeignet sind, wie sie in Abschnitt 3.7.2 vorgestellt wurden. Die Kommunikationsbibliotheken können hersteller- oder hardwarespezifisch sein, meist werden aber portable Bibliotheken verwendet, die eine standardisierte Syntax und Semantik für Kommunikationsanweisungen festlegen und für verschiedene Rechner verfügbar sind.

In diesem Kapitel geben wir einen kurzen Überblick über portable Message-Passing-Bibliotheken. In Abschnitt 5.1 stellen wir die wichtigsten Konzepte von MPI (**M**essage-**P**assing-**I**nterface) vor, in Abschnitt 5.2 beschreiben wir PVM (**P**arallel **V**irtual **M**achine) und in Abschnitt 5.3 gehen wir kurz auf das Prozessmodell und die einseitigen Kommunikationsoperationen von MPI-2 ein, das als Erweiterung von MPI vorgeschlagen wurde. Die offiziell-

len Dokumente zu MPI und MPI-2 erhält man über www.mpi-forum.org. Die unter www.netlib.org/pvm3/book/pvm-book.html erhältliche Html-Dokumentation zu [55] liefert ausführliche Information zu PVM.

5.1 Einführung in MPI

MPI (Message-Passing-Interface) ist ein Standard für Message-Passing-Bibliotheken, der Programm-Schnittstellen für Anweisungen zur Realisierung von Kommunikationsoperationen bereitstellt, wie sie z.B. in Abschnitt 3.7.2 eingeführt wurden. Die Schnittstellen werden für Programme in FORTRAN 77 und in C definiert. Für MPI-2 stehen auch Schnittstellen für C++ zur Verfügung. Die Kommunikationsanweisungen werden in C-Programmen als Funktionen bzw. in FORTRAN-77-Programmen als Subroutinen aufgerufen. Wir beschränken uns im Folgenden auf die C-Schnittstellen. Der MPI-Standard legt die Syntax der Anweisungen und die Semantik der realisierten Operationen fest, also den Effekt, den die Ausführung einer Kommunikationsoperation auf die Daten der beteiligten Prozesse hat. Die genaue Realisierung der Operationen wird aber nicht vorgegeben. Damit können unterschiedliche Implementierungen der Bibliothek für unterschiedliche Hardware-Plattformen intern unterschiedlich realisiert sein. Für den Programmierer wird aber eine einheitliche Schnittstelle zur Verfügung gestellt, um die Portabilität der auf MPI basierenden Programme sicherzustellen. MPI stellt also einen portablen Standard für die Message-Passing-Programmierung bereit. Unter einem MPI-Programm verstehen wir im Folgenden ein C- oder FORTRAN-Programm mit MPI-Aufrufen, das für verschiedene Parallelrechner ohne Änderung des Programmtextes nutzbar ist. Die einzelnen MPI-Implementierungen stellen eine für den jeweiligen Parallelrechner effiziente Realisierungsvariante bereit. Frei verfügbare MPI-Implementierungen sind z.B. MPICH bzw. MPICH2 vom Argonne National Lab und der Mississippi State University (www-unix.mcs.anl.gov/mpi/mpich2), LAM-MPI vom Ohio Supercomputing Center, der Indiana University und der University of Notre Dame (www.lam-mpi.org) sowie die von mehreren Unternehmen und Universitäten unterstützte Open-MPI-Initiative (www.open-mpi.org).

In diesem Abschnitt geben wir einen kurzen Überblick über die Message-Passing-Programmierung mit MPI. Ein MPI-Programm besteht aus Prozessen, die Nachrichten austauschen können. Die Anzahl der Prozesse wird beim Start des Programms festgelegt und kann während des Programmlaufes nicht mehr verändert werden. Zur Laufzeit des Programms ist also (im Gegensatz zu MPI-2 und PVM) *kein dynamisches Erzeugen von Prozessen* möglich. Viele Implementierungen von MPI sind so realisiert oder vorkonfiguriert, dass jeder Prozessor des Parallelrechners genau einen Prozess ausführt und dass jeder Prozess das gleiche Programm im SPMD-Stil abarbeitet. Prinzipiell kann jeder Prozess Daten (von Dateien) einlesen und Daten (auf Dateien) ausgeben, üblicherweise ist aber ein koordiniertes Einlesen und Ausgeben

erforderlich und nur ein ausgewählter Prozess führt die Ein- und Ausgabe durch. MPI-Programme können parametrisiert in der Anzahl p der Prozesse erstellt werden. Beim Aufruf des Programms wird dann die konkrete Anzahl der gewünschten Prozesse angegeben. Der Start eines MPI-Programms wird in vielen Systemen durch die Kommandozeileingabe

```
mpirun -np 4 programmname programmargumente
```

realisiert. Dieser Aufruf bewirkt den Start des Programms `programmname` mit $p = 4$ Prozessen.

Der wesentliche Teil der von MPI zur Verfügung gestellten Kommunikationsoperationen bezieht sich auf den Austausch von Daten zwischen den beteiligten Prozessen. Wir stellen im Folgenden eine Auswahl dieser Kommunikationsoperationen vor, müssen uns aber wegen der Vielzahl der von MPI zur Verfügung gestellten Kommunikationsoperationen auf die wichtigsten beschränken. Für eine vollständige Beschreibung von MPI verweisen wir auf [119, 150, 151], aus denen die folgende Darstellung zusammengestellt ist.

Beschreiben werden wir insbesondere die durch den MPI-Standard festgelegte Semantik der vorgestellten MPI-Operationen. Auf die in den verschiedenen Realisierungen von MPI benutzten Implementierungsvarianten werden wir nur eingehen, wenn dadurch ein unterschiedliches Verhalten von MPI-Operationen verursacht werden kann und das Wissen über die Implementierung daher für das Erstellen von korrekten Programmen wichtig ist. Zur Darstellung der Semantik von MPI-Operationen werden wir die in MPI-Beschreibungen üblichen semantischen Begriffe verwenden, von denen wir nun einige angeben.

blockierend: Eine MPI-Kommunikationsanweisung heißt *blockierend*, falls die Rückkehr der Kontrolle zum aufrufenden Prozess bedeutet, dass alle Ressourcen (z.B. Puffer), die für den Aufruf benötigt wurden, erneut für andere Operationen genutzt werden können. Insbesondere finden alle durch den Aufruf ausgelösten Zustandsveränderungen des aufrufenden Prozesses *vor* der Rückkehr der Kontrolle statt.

nichtblockierend: Eine MPI-Kommunikationsanweisung heißt *nicht-blockierend*, falls die aufgerufene Kommunikationsanweisung die Kontrolle zurückgibt, bevor die durch sie ausgelöste Operation vollständig abgeschlossen ist und bevor eingesetzte Ressourcen (z.B. Puffer) wieder benutzt werden dürfen. Die ausgelöste Operation ist erst dann vollständig abgeschlossen, wenn alle Zustandsänderungen dieser Operation für den die Prozedur aufrufenden Prozess sichtbar sind und alle Ressourcen wieder verwendet werden können.

Die blockierende und nichtblockierende Semantik von Kommunikationsanweisungen beschreibt deren Verhalten aus der Sicht des aufrufenden Prozesses, also aus *lokaler* Sicht. Da an einer Kommunikationsoperation meist zwei oder mehrere Prozesse beteiligt sind, von denen jeder eine geeignete Kommunikationsanweisung ausführt, hat die blockierende oder nichtblockierende

Semantik Auswirkungen auf die Koordination der auszuführenden Operationen. Aus *globaler* Sicht wird das Zusammenspiel der an einer Kommunikationsoperation beteiligten Prozesse durch die Eigenschaften *synchroner* oder *asynchroner* Kommunikation beschrieben.

synchrone Kommunikation: Bei synchroner Kommunikation findet die eigentliche Übertragung einer Nachricht nur statt, wenn Sender und Empfänger zur *gleichen* Zeit an der Kommunikation teilnehmen.

asynchrone Kommunikation: Bei asynchroner Kommunikation kann der Sender Daten versenden, ohne sicher zu sein, dass der Empfänger bereit ist, die Daten zu empfangen.

5.1.1 Einzeltransferoperationen

Alle Kommunikationsoperationen werden in MPI mit Hilfe von **Kommunikatoren** verschickt, wobei ein Kommunikator eine Menge von Prozessen bestimmt, die untereinander Nachrichten austauschen können. Wir gehen im Folgenden bis auf weiteres davon aus, daß der Default-Kommunikator `MPI_COMM_WORLD` verwendet wird, der alle Prozesse eines parallelen Programms umfasst. In Abschnitt 5.1.4 werden wir dann näher auf Kommunikatoren eingehen.

Die einfachste Form des Datenaustausches zwischen Prozessen ist ein **Einzeltransfer**, der auch als **Punkt-zu-Punkt-Kommunikation** bezeichnet wird, da genau zwei Prozesse beteiligt sind, ein Sendeprozess (Sender) und ein Empfangsprozess (Empfänger), die beide eine Kommunikationsanweisung ausführen müssen. Zur Durchführung eines Einzeltransfers führt der Sender folgende Kommunikationsanweisung aus:

```
int MPI_Send(void *smessage,
             int count,
             MPI_Datatype datatype,
             int dest,
             int tag,
             MPI_Comm comm).
```

Dabei bezeichnet

- **smessage** einen Sendepuffer, der die zu sendenden Elemente fortlaufend enthält,
- **count** die Anzahl der zu sendenden Elemente,
- **datatype** den Typ der zu sendenden Elemente, wobei alle Elemente einer Nachricht den gleichen Typ haben müssen,
- **dest** die Nummer des Zielprozesses, der die Daten empfangen soll,
- **tag** eine Markierung der Nachricht, die dem Empfänger die Unterscheidung verschiedener Nachrichten desselben Senders erlaubt und
- **comm** einen Kommunikator, der die Gruppe von Prozessen bezeichnet, die sich Nachrichten zusenden können.

Die Länge einer Nachricht in Bytes ergibt sich aus dem Produkt der Anzahl der zu sendenden Elemente `count` und der Anzahl der von dem angegebenen Datentyp `datatype` belegten Bytes. Die Markierung einer Nachricht sollte ein Wert zwischen 0 und 32767 sein. Zum Empfangen einer Nachricht führt der Empfänger eine korrespondierende Operation

```
int MPI_Recv(void *rmessage,
             int count,
             MPI_Datatype datatype,
             int source,
             int tag,
             MPI_Comm comm,
             MPI_Status *status).
```

aus. Dabei bezeichnet

- `rmessage` einen Empfangspuffer, in den die zu empfangende Nachricht abgelegt werden soll,
- `count` eine Obergrenze für die Anzahl der zu empfangenden Elemente,
- `datatype` den Typ der zu empfangenden Elemente,
- `source` die Nummer des Prozesses, von dem die Nachricht empfangen werden soll,
- `tag` die gewünschte Markierung der zu empfangenden Nachricht,
- `comm` einen Kommunikator und
- `status` eine Datenstruktur, die Informationen über die tatsächlich empfangene Nachricht enthält.

Durch Angabe von `source = MPI_ANY_SOURCE` kann ein Prozess Nachrichten von einem beliebigen anderen Prozess empfangen. Durch Angabe von `tag = MPI_ANY_TAG` kann eine Nachricht mit einer beliebigen Markierung empfangen werden. In beiden Fällen sind die Angaben über die wirklich empfangene Nachricht in der Datenstruktur `status` enthalten, deren Adresse der empfangende Prozess als Parameter von `MPI_Recv()` spezifiziert. Nach dem Aufruf von `MPI_Recv()` enthält diese Struktur die folgende Information:

```
status.MPI_SOURCE spezifiziert den Sender der empfangenen Nachricht,
status.MPI_TAG gibt die Markierung der empfangenen Nachricht an,
status.MPI_ERROR enthält einen Fehlercode.
```

Die tatsächliche Größe der erhaltenen Nachricht erhält man durch den Aufruf

```
int MPI_Get_count (MPI_Status *status,
                  MPI_Datatype datatype,
                  int *count_ptr).
```

wobei `status` ein Zeiger auf die vom zugehörigen `MPI_Recv()`-Aufruf besetzte Datenstruktur ist. Die Funktion liefert die Anzahl der empfangenen Elemente in der Variablen zurück, deren Adresse als Parameter `count_ptr` angegeben ist.

Die vordefinierten Datentypen von MPI und die korrespondierenden Datentypen in C sind in der folgenden Tabelle wiedergegeben:

MPI Datentyp	C-Datentyp
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_PACKED	
MPI_BYTE	

Zu den Datentypen `MPI_PACKED` und `MPI_BYTE` gibt es keine korrespondierenden C-Datentypen. Ein Wert vom Typ `MPI_BYTE` besteht aus einem Byte, das nicht mit einem Character identisch sein muß. Der Typ `MPI_PACKED` wird für spezielle Packprozeduren verwendet.

Ein Nachrichtentransfer wird intern in drei Schritten realisiert:

1. Die Daten werden aus dem Sendepuffer `smessage` in einen Systempuffer kopiert und die zu verschickende Nachricht wird zusammengesetzt, indem ein Kopf (*header*) hinzugefügt wird, der Informationen über den Sender der Nachricht, den Zielprozess, die Markierung und den Kommunikator enthält.
2. Die Daten werden über das Netzwerk des Parallelrechners vom Sender zum Empfänger geschickt.
3. Die Daten werden vom Empfänger `dest` aus dem Systempuffer, in dem die Nachricht empfangen wurde, in den angegebenen Empfangspuffer kopiert.

Bei den Operationen `MPI_Send()` und `MPI_Recv()` handelt es sich um *blockierende, asynchrone* Operationen. Für die `MPI_Recv()`-Operation bedeutet dies, dass diese auch dann gestartet werden kann, wenn die zugehörige `MPI_Send()`-Operation noch nicht gestartet wurde. Die `MPI_Recv()`-Operation *blockiert*, bis der angegebene Empfangspuffer die erwartete Nachricht enthält. Für die beteiligte `MPI_Send()`-Operation bedeutet dies, dass die Operation auch gestartet werden kann, wenn die zugehörige `MPI_Recv()`-Operation noch nicht gestartet wurde. Die `MPI_Send()`-Operation *blockiert*, bis der angegebene Sendepuffer wiederverwendet werden kann. Das tatsächliche Verhalten bei der Blockierung des Senders hängt von der speziellen MPI-Implementierung ab, wobei eine der beiden folgenden Möglichkeiten in vielen Bibliotheken verwendet wird:

- a) Wenn die Nachricht ohne Zwischenspeichern *direkt* aus dem Sendepuffer in den Empfangspuffer eines anderen Prozesses kopiert wird, blockiert

die `MPI_Send()`-Operation, bis die Nachricht *vollständig* in den Empfangspuffer kopiert wurde, was insbesondere bedeutet, dass die zugehörige `MPI_Recv()`-Operation gestartet sein muss.

- b) Wenn die Nachricht in einem *Systempuffer* des Senders zwischengespeichert wird, kann der Sender weiterarbeiten, sobald der Kopiervorgang auf der Senderseite abgeschlossen ist. Das kann auch vor dem Start der zugehörigen `MPI_Recv()`-Operation sein. Der Vorteil liegt also darin, dass der Sender nur kurz blockiert wird. Der Nachteil besteht darin, dass für den Systempuffer zusätzlicher Platz benötigt wird und dass das Kopieren in den Systempuffer zusätzliche Zeit verbraucht.

Beispiel: Abbildung 5.1 zeigt ein einfaches MPI-Programm als Beispiel zur Benutzung von `MPI_Send()` und `MPI_Recv()`, in dem Prozess 0 an Prozess 1 eine Nachricht schickt, vgl. [119]. Alle durch den Programmaufruf

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main (int argc, char *argv[]) {
    int my_rank, p, tag=0;
    char msg [20];
    MPI_Status status;

    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size (MPI_COMM_WORLD, &p);
    if (my_rank == 0) {
        strcpy (msg, "Hello ");
        MPI_Send (msg, strlen(msg)+1, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
    }
    if (my_rank == 1)
        MPI_Recv (msg, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
    MPI_Finalize ();
}
```

Abb. 5.1. Ein einfaches MPI-Programm: Nachrichtenübertragung von Prozess 0 an Prozess 1.

entstehenden Prozesse haben den gleichen Programmtext, können aber auf Grund unterschiedlicher Belegungen der Variablen verschiedene Berechnungen bzw. Kommunikationsoperationen durchführen. Die Variablendeklaration definiert eine Variable `status` vom Typ `MPI_Status`, die in der `MPI_Recv`-Operation verwendet wird. Die erste Anweisung ist `MPI_Init()`, die in jedem MPI-Programm *vor* der ersten MPI-Anweisung stehen muss. Der Aufruf `MPI_Comm_rank (MPI_COMM_WORLD, &my_rank)` liefert jedem beteiligten Prozess seine Prozessnummer bzgl. des angegebenen Kommunikators

`MPI_COMM_WORLD` in der Variable `my_rank` zurück, wobei die Prozessnummern von 0 an aufwärts durchnummeriert sind. Der Aufruf `MPI_Comm_size` (`MPI_COMM_WORLD`, `&p`) liefert die Anzahl der Prozesse des angegebenen Kommunikators in der Variable `p` zurück. Als Kommunikator wird jeweils der vordefinierte Kommunikator `MPI_COMM_WORLD` benutzt. Abhängig von der Belegung von `my_rank` führt das gleiche Programm für die verschiedenen Prozesse zu unterschiedlichen Berechnungen. Prozess 0 führt nach der Abfrage `if (my_rank == 0)` eine Kopier- und eine Sendeoperation aus. Prozess 1 führt nach der Abfrage `if (my_rank == 1)` eine entsprechende Empfangsoperation aus. Als Empfänger bzw. Sender in den Sende- und Empfangsoperationen werden die entsprechenden Werte von `my_rank=1` oder `my_rank=0` eingesetzt. Alle anderen Prozesse springen direkt zum Befehl `MPI_Finalize()`, der jedes MPI-Programm abschließen muss. □

Eine wichtige Eigenschaft von MPI besteht darin, dass durch die MPI-Implementierung sichergestellt wird, dass Nachrichten sich *nicht überholen* können, d.h. wenn ein Sender aufeinanderfolgend zwei Nachrichten zum gleichen Empfänger schickt und beide Nachrichten auf das erste `MPI_Recv()` des Empfängers passen, wird sichergestellt, dass die zuerst gesendete Nachricht auch zuerst empfangen wird. Es ist aber zu beachten, dass die Beteiligung eines dritten Prozesses die Ordnung zerstören kann. Wir betrachten dazu das folgende Programmfragment, vgl. [150]:

```
/* Beispiel zur Nichteinhaltung der gewünschten Empfangsreihenfolge*/
MPI_Comm_rank (comm, &my_rank);
if (my_rank == 0) {
    MPI_Send (sendbuf1, count, MPI_INT, 2, tag, comm);
    MPI_Send (sendbuf2, count, MPI_INT, 1, tag, comm);
}
else if (my_rank == 1) {
    MPI_Recv (recvbuf1, count, MPI_INT, 0, tag, comm, &status);
    MPI_Send (recvbuf1, count, MPI_INT, 2, tag, comm);
}
else if (my_rank == 2) {
    MPI_Recv (recvbuf1, count, MPI_INT, MPI_ANY_SOURCE, tag, comm,
              &status);
    MPI_Recv (recvbuf2, count, MPI_INT, MPI_ANY_SOURCE, tag, comm,
              &status);
}
```

In diesem Programmstück schickt Prozess 0 eine Nachricht an Prozess 2 und danach an Prozess 1. Prozess 1 empfängt die Nachricht von Prozess 0 und schickt sie an Prozess 2 weiter. Prozess 2 empfängt zwei Nachrichten in der Reihenfolge, in der sie ankommen, was durch die Angabe von `MPI_ANY_SOURCE` möglich ist. Da Prozess 0 zuerst eine Nachricht an Prozess 2 schickt und danach erst an Prozess 1 und da die von Prozess 0 zuletzt losgeschickte Nachricht den Umweg über Prozess 1 nimmt, erwartet man, dass die von Prozess 0 zuerst losgeschickte Nachricht diejenige ist, die Prozess 2 mit dem zuerst ausgeführten `MPI_Recv()` empfängt. Dies ist jedoch nicht unbedingt der Fall,

da die von Prozess 0 zuerst losgeschickte Nachricht z.B. durch eine Kollision im Netzwerk verzögert werden kann, während die zweite Nachricht ohne Verzögerung zugestellt wird. Daher kann der Fall auftreten, dass Prozess 2 mit der ersten ausgeführten `MPI_Recv()`-Anweisung die von Prozess 0 zuletzt über Prozess 1 losgeschickte Nachricht empfängt. Der Programmierer kann sich also bei drei und mehr beteiligten Prozessen nicht auf eine Zustellungsreihenfolge verlassen. Eine *sichere Reihenfolge* wird nur dann gewährleistet, wenn Prozess 2 den erwarteten Sender der Nachricht in den `MPI_Recv()`-Operationen angibt.

Ein unvorsichtiger Umgang mit den Operationen zum Senden und Empfangen von Nachrichten kann zum *Deadlock* führen, wie das folgende Beispiel zeigt, in dem die Prozesse 0 und 1 `MPI_Send()`- und `MPI_Recv()`-Operationen ausführen:

```
/* Programmfragment, durch das ein Deadlock erzeugt wird */
MPI_Comm_rank (comm, &my_rank);
if (my_rank == 0) {
    MPI_Recv (recvbuf, count, MPI_INT, 1, tag, comm, &status);
    MPI_Send (sendbuf, count, MPI_INT, 1, tag, comm);
}
else if (my_rank == 1) {
    MPI_Recv (recvbuf, count, MPI_INT, 0, tag, comm, &status);
    MPI_Send (sendbuf, count, MPI_INT, 0, tag, comm);
}
```

Das Problem dieses Programmfragments liegt darin, dass die Prozesse 0 und 1 gegenseitig aufeinander warten: Die `MPI_Send()`-Operation von Prozess 0 kann erst beginnen, wenn die `MPI_Recv()`-Operation von Prozess 0 beendet ist. Die Beendigung dieser `MPI_Recv()`-Operation ist jedoch nur möglich, wenn die `MPI_Send()`-Operation von Prozess 1 ausgeführt wurde. Dies erfordert aber, dass Prozess 1 die vorangehende `MPI_Recv()`-Operation beendet hat, was jedoch nur möglich ist, wenn Prozess 0 die zugehörige `MPI_Send()`-Operation ausgeführt hat. Es kommt also zu einem zyklischen Warten der Prozesse 0 und 1.

Das Auftreten eines Deadlocks kann auch von der Implementierung des Laufzeitsystems von MPI abhängen, wie das folgende Beispiel zeigt:

```
/* Programmfragment, das implementierungsabhängig
   einen Deadlock erzeugt */
MPI_Comm_rank (comm, &my_rank);
if (my_rank == 0) {
    MPI_Send (sendbuf, count, MPI_INT, 1, tag, comm);
    MPI_Recv (recvbuf, count, MPI_INT, 1, tag, comm, &status);
}
else if (my_rank == 1) {
    MPI_Send (sendbuf, count, MPI_INT, 0, tag, comm);
    MPI_Recv (recvbuf, count, MPI_INT, 0, tag, comm, &status);
}
```

In diesem Beispiel läuft die Übertragung korrekt, wenn die von Prozess 0 und 1 abgeschickten Nachrichten jeweils aus dem Sendepuffer `sendbuf` in einen Systempuffer zwischengespeichert werden, so dass die Kontrolle nach dem Kopieren in den Systempuffer an den Sender zurückgegeben werden kann. Existiert kein Systempuffer, so tritt ein Deadlock auf. Keiner der beiden Prozesse kann die zuerst ausgeführte `MPI_Send()`-Operation abschließen, da das korrespondierende `MPI_Recv()` des anderen Prozesses nicht ausgeführt wird. Eine **sichere Implementierung**, die ohne Annahmen über das Verhalten des Laufzeitsystems auskommt, ist die folgende:

```
/* Programmfragment, durch das kein Deadlock erzeugt wird */
MPI_Comm_rank (comm, &myrank);
if (my_rank == 0) {
    MPI_Send (sendbuf, count, MPI_INT, 1, tag, comm);
    MPI_Recv (recvbuf, count, MPI_INT, 1, tag, comm, &status);
}
else if (my_rank == 1) {
    MPI_Recv (recvbuf, count, MPI_INT, 0, tag, comm, &status);
    MPI_Send (sendbuf, count, MPI_INT, 0, tag, comm);
}
```

Ein MPI-Programm wird als **sicher** bezeichnet, wenn seine Korrektheit nicht auf Annahmen über das Vorhandensein von Systempuffern oder die Größe von Systempuffern beruht, d.h. sichere MPI-Programme werden auch ohne Systempuffer korrekt ausgeführt. Wenn mehr als zwei Prozesse sich gegenseitig Nachrichten zusenden, muss zum Erreichen einer sicheren Implementierung genau festgelegt werden, in welcher Reihenfolge die Send- und Empfangsoperationen ausgeführt werden. Als Beispiel betrachten wir p Prozesse, wobei Prozess i , $0 \leq i \leq p-1$, Daten an Prozess $(i+1) \bmod p$ schickt und Daten von Prozess $(i-1) \bmod p$ empfängt, d.h. die Nachrichten werden in einem logischen Ring verschickt. Eine sichere Implementierung erreicht man dadurch, dass die Prozesse mit gerader Nummer zuerst senden und dann empfangen, während die Prozesse mit ungerader Nummer zuerst empfangen und dann senden. Für vier Prozesse ergibt sich damit das folgende Schema:

Zeit	Prozess 0	Prozess 1	Prozess 2	Prozess 3
1	<code>MPI_Send()</code> zu 1	<code>MPI_Recv()</code> von 0	<code>MPI_Send()</code> zu 3	<code>MPI_Recv()</code> von 2
2	<code>MPI_Recv()</code> von 3	<code>MPI_Send()</code> zu 2	<code>MPI_Recv()</code> von 1	<code>MPI_Send()</code> zu 0

Dieses Schema führt auch bei einer ungeraden Anzahl von Prozessen zu einer sicheren Implementierung. Für drei Prozesse ergibt sich z.B. der folgende Ablauf:

Zeit	Prozess 0	Prozess 1	Prozess 2
1	<code>MPI_Send()</code> zu 1	<code>MPI_Recv()</code> von 0	<code>MPI_Send()</code> zu 0
2	<code>MPI_Recv()</code> von 2	<code>MPI_Send()</code> zu 2	-warte-
3		-warte-	<code>MPI_Recv()</code> von 1

Bestimmte Kommunikationsoperationen wie `MPI_Send()` von Prozess 2 können zwar verzögert werden, weil der Empfänger die zugehörige `MPI_Recv()`-Operation erst zu einem späteren Zeitpunkt ausführt, ein Deadlock tritt aber nicht auf.

Für den häufig auftretenden Fall, dass jeder Prozess sowohl Daten empfängt als auch Daten versendet, gibt es einen eigenen Befehl:

```
int MPI_Sendrecv (void *sendbuf,
                  int sendcount,
                  MPI_Datatype sendtype,
                  int dest,
                  int sendtag,
                  void *recvbuf,
                  int recvcount,
                  MPI_Datatype recvtype,
                  int source,
                  int recvtag,
                  MPI_Comm comm,
                  MPI_Status *status)
```

Dabei bezeichnet

- `sendbuf` den Puffer, in dem die zu sendenden Elemente liegen,
- `sendcount` die Anzahl der zu sendenden Elemente,
- `sendtype` den Typ der zu sendenden Elemente,
- `dest` die Nummer des Zielprozesses der Nachricht,
- `sendtag` die Markierung der Nachricht,
- `recvbuf` den Puffer, in dem die zu empfangende Nachricht abgelegt werden soll,
- `recvcount` die maximale Anzahl der zu empfangenden Elemente,
- `recvtype` den Typ der zu empfangenden Elemente,
- `source` die Nummer des Senders der zu empfangenden Nachricht,
- `recvtag` die Markierung der zu empfangenden Nachricht,
- `comm` den verwendeten Kommunikator und
- `status` den Rückgabestatus.

Der Vorteil der Verwendung von `MPI_Sendrecv()` liegt darin, dass der Programmierer in einem System ohne Systempuffer nicht auf die richtige Anordnung der Sende- und Empfangsoperationen achten muss. Wird für jeden Prozess eine `MPI_Sendrecv()`-Operation verwendet, so sorgt das Laufzeitsystem von MPI für eine Realisierung des Nachrichtenaustausches ohne Deadlock. Zu beachten ist, dass der Sendepuffer `sendbuf` und der Empfangspuffer `recvbuf` unterschiedliche, nicht überlappende Speicherbereiche bezeichnen müssen. Es können jedoch unterschiedlich große Nachrichten mit unterschiedlichen Datentypen gesendet und empfangen werden. Wenn Sende- und Empfangspuffer identisch sein sollen, muss die folgende Funktion verwendet werden:

```

int MPI_Sendrecv_replace (void *buffer,
                           int count,
                           MPI_Datatype type,
                           int dest,
                           int sendtag,
                           int source,
                           int recvtag,
                           MPI_Comm comm,
                           MPI_Status *status)

```

Dabei bezeichnet **buffer** den als Sende- und Empfangspuffer verwendeten Puffer. In diesem Fall ist die Anzahl **count** der zu sendenden und der zu empfangenden Elemente und deren Datentyp **type** identisch. Das Laufzeitsystem sorgt für ein eventuelles Zwischenspeichern in Systempuffern.

Die Verwendung von blockierenden Kommunikationsoperationen kann zu einer schlechten Ausnutzung der Systemressourcen führen, da Wartezeiten verursacht werden. Beispielsweise wartet eine blockierende Sendeoperation, bis die zu verschickende Nachricht in einen Sendepuffer kopiert wurde bzw. bis diese beim Empfänger angekommen ist. Da viele Parallelrechner für jeden Knoten eine separate Kommunikationshardware oder sogar einen separaten Kommunikationsprozessor enthalten, braucht der eigentliche Prozessor sich jedoch gar nicht um die Übermittlung der Nachricht zu kümmern und ist damit wegen der Wartezeiten schlecht ausgenutzt. Eine Alternative stellen *nichtblockierende Kommunikationsoperationen* dar, mit deren Hilfe die beschriebenen Wartezeiten vermieden werden können.

Eine **nichtblockierende Sendeoperation** startet den Sendevorgang ohne sicherzustellen, dass nach Abschluss der Operation die Nachricht aus dem Sendepuffer kopiert wurde. Während des eigentlichen Kopier- und Übertragungsvorgangs kann der Prozessor andere Berechnungen ausführen, wenn eine geeignete Kommunikationshardware zur Verfügung steht. Diese Berechnungen sollten den Systempuffer allerdings nicht verändern, bevor nicht sichergestellt ist, dass die Sendeoperation abgeschlossen wurde, die Nachricht also in einen Systempuffer kopiert oder bereits beim Empfänger angekommen ist. In MPI wird eine nichtblockierende Sendeoperation durch folgende Funktion realisiert:

```

int MPI_Isend (void *buffer,
               int count,
               MPI_Datatype type,
               int dest,
               int tag,
               MPI_Comm comm,
               MPI_Request *request).

```

Die Bedeutung der Parameter ist die gleiche wie bei **MPI_Send()**. Der zusätzliche Parameter vom Typ **MPI_Request** bezeichnet eine für den Programmierer nicht direkt zugreifbare Datenstruktur, die zur Identifikation der durchgeführten nichtblockierenden Operation dient und in der vom System Infor-

mationen über den Status der Ausführung der jeweiligen Operation abgelegt werden.

Eine **nichtblockierende Empfangsoperation** startet eine Empfangsoperation, bringt diese aber nicht zum Abschluss, sondern informiert das Laufzeitsystem darüber, dass Daten im Empfangspuffer abgelegt werden können. Die Daten im Empfangspuffer können von den nachfolgenden Operationen aber nicht ohne weiteres benutzt werden, bevor die Empfangsoperation abgeschlossen ist. In MPI wird eine nichtblockierende Empfangsoperation durch folgende Funktion realisiert:

```
int MPI_Irecv (void *buffer,
               int count,
               MPI_Datatype type,
               int source,
               int tag,
               MPI_Comm comm,
               MPI_Request *request).
```

Damit eine Wiederbenutzung der Sende- und Empfangspuffer möglich ist, stellt MPI Funktionen zur Verfügung, mit deren Hilfe überprüft werden kann, ob eine gestartete nichtblockierende Operation bereits abgeschlossen ist bzw. die den ausführenden Prozess blockieren, bis die entsprechende Operation beendet ist. Dabei dient die oben erwähnte Datenstruktur vom Typ `MPI_Request` zur Identifikation der Operation. Ob eine nichtblockierende Operation bereits abgeschlossen ist, kann mit der folgenden Funktion festgestellt werden:

```
int MPI_Test (MPI_Request *request,
              int *flag,
              MPI_Status *status).
```

Wenn die von `request` bezeichnete nichtblockierende Operation bereits beendet ist, ist nach dem Aufruf `flag=1` (true) gesetzt, sonst `flag=0`. Handelt es sich um eine Empfangsoperation und ist diese abgeschlossen, so enthält die Datenstruktur `status` vom Typ `MPI_Status` des Aufrufs von `MPI_Test()` die Informationen, die bei `MPI_Recv()` beschrieben wurden. Bei nicht abgeschlossener Empfangsoperation sind die Einträge von `status` nicht definiert. Wurde mit `MPI_Test()` eine Sendeoperation angesprochen, so sind die Einträge von `status` mit Ausnahme von `status.MPI_ERROR` ebenfalls nicht definiert, vgl. [151]. Die Funktion

```
int MPI_Wait (MPI_Request *request, MPI_Status *status)
```

blockiert den ausführenden Prozess, bis die von `request` bezeichnete Operation vollständig beendet ist. Falls es sich um eine Sendeoperation handelt, kann der auf `MPI_Wait()` folgende Befehl den Sendepuffer überschreiben. Falls `request` eine Empfangsoperation bezeichnet, können die auf `MPI_Wait()` folgenden Befehle die Daten im Empfangspuffer benutzen.

MPI stellt sicher, dass sich auch bei nichtblockierenden Kommunikationsoperationen Nachrichten nicht überholen können. Das Mischen von blockierenden und nichtblockierenden Operationen ist möglich, d.h. mit `MPI_Isend()` gesendete Daten können mit `MPI_Recv()` empfangen werden und umgekehrt.

Beispiel: Als Beispiel zur Verwendung von nichtblockierenden Operationen betrachten wir das Aufsummieren von über mehrere Prozesse verteilter Information, vgl. [119]. Wir nehmen an, dass p Prozesse beteiligt sind, von denen jeder die gleiche Anzahl von Floating-Point-Daten berechnet hat. Jedem dieser Prozesse sollen die Daten von allen anderen Prozessen zur Verfügung gestellt werden. Das Ziel wird in $p - 1$ Schritten realisiert, wozu die Prozesse logisch in einem Ring angeordnet werden. Im ersten Schritt schickt jeder Prozess seine eigenen Daten an seinen Nachfolger im Ring weiter. In den folgenden Schritten schickt jeder Prozess die zuletzt empfangenen Daten an seinen Nachfolger im Ring weiter.

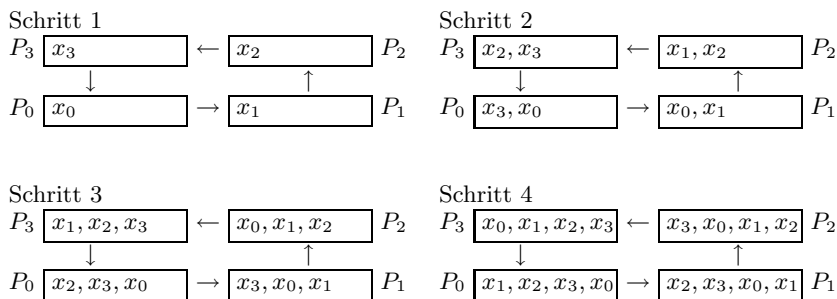


Abb. 5.2. Veranschaulichung der Kommunikationsschritte zum Aufsummieren von Daten mit logischer Ringstruktur für $p = 4$ Prozesse, vgl. [119].

Abbildung 5.2 veranschaulicht die durchzuführenden Schritte für vier Prozesse. Für die nachfolgende Realisierung nehmen wir an, dass jeder Prozess seine lokalen Daten in einem Feld x zur Verfügung stellt und daß die Gesamtdaten in einem Feld y aufgesammelt werden, das p -mal größer als x ist.

Eine Implementierung mit blockierenden Kommunikationsanweisungen ist in Abbildung 5.3 wiedergegeben. Die Größe der lokalen Blöcke wird durch `blocksize` angegeben. Nach dem Kopieren seines lokalen Blockes aus x an die zugehörige Position in y bestimmt jeder Prozess seinen Nachfolger `succ` und seinen Vorgänger `pred` im Ring. Danach wird in $p - 1$ Schritten jeweils der zuletzt in y kopierte Block an den Nachfolger geschickt und es wird ein Block vom Vorgänger in der links daneben liegenden Position empfangen. Man beachte, dass bei dieser Implementierung die Existenz von geeigneten Systempuffern vorausgesetzt wird.

```

void Gather_ring (float x[], int blocksize, float y[])
{
    int i, p, my_rank, succ, pred;
    int send_offset, recv_offset;
    MPI_Status status;

    MPI_Comm_size (MPI_COMM_WORLD, &p);
    MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
    for (i=0; i<blocksize; i++)
        y[i+my_rank * blocksize] = x[i];
    succ = (my_rank+1) % p;
    pred = (my_rank-1+p) % p;
    for (i=0; i<p-1; i++) {
        send_offset = ((my_rank-i+p) % p) * blocksize;
        recv_offset = ((my_rank-i-1+p) % p) * blocksize;
        MPI_Send (y+send_offset, blocksize, MPI_FLOAT, succ, 0,
                  MPI_COMM_WORLD);
        MPI_Recv (y+recv_offset, blocksize, MPI_FLOAT, pred, 0,
                  MPI_COMM_WORLD, &status);
    }
}

```

Abb. 5.3. MPI-Programm zum Aufsummern von über mehrere Prozesse verteilten Datenblöcken. Die Prozesse werden in einem Ring angeordnet. Für die Kommunikation werden *blockierende* Operationen verwendet, so dass Deadlockfreiheit nur bei Verwendung von Systempuffern vorliegt.

Die in Abbildung 5.4 wiedergegebene Implementierung mit *nichtblockierenden* Kommunikationsanweisungen nutzt die nichtblockierenden Operationen zur Überlappung von Kommunikation und lokaler Berechnung. Die Überlappung bezieht sich hier auf die Kommunikation mit Vorgänger **pred** und Nachfolger **succ** und die Berechnung der Positionen des im nächsten Schritt zu sendenden und zu empfangenden Blockes im Puffer **y**, d.h. der Berechnung von **send_offset** und **recv_offset**. Die Sende- und Empfangsoperationen werden mit **MPI_Isend()** bzw. **MPI_Irecv()** gestartet. Nach Rückkehr der Kontrolle werden **send_offset** und **recv_offset** neu berechnet und mit **MPI_Wait()** wird auf die Beendigung der gestarteten Sende- und Empfangsoperationen gewartet. Laut [119] führt die nichtblockierende Variante für Blöcke der Größe 1 für eine Intel Paragon mit 32 Prozessoren zu einer Laufzeit von *4.2ms*, während die blockierende Variante *4.9ms* braucht. Für eine IBM SP2 liegen die Laufzeiten bei *2.5ms* bzw. *3.9ms*. □

Sowohl für blockierende als auch für nichtblockierende Sendeoperationen gibt es verschiedene **Übertragungsmodi**, die die Koordination der Sende- und Empfangsoperationen bestimmen. Die verschiedenen Übertragungsmodi werden über verschiedene Kommunikationsanweisungen angesprochen:

- **Standardmodus:** Die bisher beschriebenen Kommunikationsoperationen bewirken eine Übertragung im Standardmodus. In diesem Modus bestimmt

```

void Gather_ring (float x[], int blocksize, float y[])
{
    int i, p, my_rank, succ, pred;
    int send_offset, recv_offset;
    MPI_Status status;
    MPI_Request send_request, recv_request;

    MPI_Comm_size (MPI_COMM_WORLD, &p);
    MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
    for (i=0; i<blocksize; i++)
        y[i+my_rank * blocksize] = x[i];
    succ = (my_rank+1) % p;
    pred = (my_rank-1+p) % p;
    send_offset = my_rank * blocksize;
    recv_offset = ((my_rank-1+p) % p) * blocksize;
    for (i=0; i<p-1; i++) {
        MPI_Isend (y+send_offset, blocksize, MPI_FLOAT, succ, 0,
                  MPI_COMM_WORLD, &send_request);
        MPI_Irecv (y+recv_offset, blocksize, MPI_FLOAT, pred, 0,
                  MPI_COMM_WORLD, &recv_request);
        send_offset = ((my_rank-i-1+p) % p) * blocksize;
        recv_offset = ((my_rank-i-2+p) % p) * blocksize;
        MPI_Wait (&send_request, &status);
        MPI_Wait (&recv_request, &status);
    }
}

```

Abb. 5.4. MPI-Programm zum Aufsammeln von verteilten Datenblöcken, vgl. Abbildung 5.3. Statt blockierenden Kommunikationsoperationen werden hier nicht-blockierende verwendet.

das Laufzeitsystem, ob Nachrichten in Systempuffern zwischengespeichert werden oder ob sie ohne Zwischenspeicherung direkt dem Empfänger zugestellt werden. Das Laufzeitsystem könnte z.B. festlegen, dass kleine Nachrichten zwischengespeichert werden, große jedoch nicht. Für den Programmierer bedeutet dies, dass er sich nicht darauf verlassen kann, dass eine Zwischenspeicherung vorgenommen wird, d.h. er sollte Programme schreiben, die auch ohne Zwischenspeicherung der Nachrichten korrekt arbeiten. Dies ist insbesondere für die problemlose Portierbarkeit der Programme wichtig.

- **Synchroner Modus:** Anders als im Standardmodus, wo (bei Verwendung von Systempuffern) eine Sendeoperation auch beendet werden kann, wenn die zugehörige Empfangsoperation noch nicht gestartet wurde, wird im synchronen Modus eine Sendeoperation erst dann beendet, wenn die zugehörige Empfangsoperation gestartet und mit dem Empfang von Daten begonnen wurde. Im synchronen Modus impliziert das Ausführen von zusammengehörenden Send- und Empfangsoperationen eine *Synchronisation* zwischen Sender und Empfänger. Eine *blockierende Sendeoperation im synchronen Modus* wird in MPI durch die Kommunikationsanwei-

sung `MPI_Ssend()` ausgeführt. Diese Operation hat die gleichen Parameter mit gleicher Bedeutung wie `MPI_Send()`. Eine *nichtblockierende Sendeoperation im synchronen Modus* wird durch die Kommunikationsanweisung `MPI_Issend()` ausgeführt. Diese Operation hat die gleichen Parameter mit gleicher Bedeutung wie `MPI_Isend()`. Ebenso wie eine nichtblockierende Sendeoperation im Standardmodus kehrt bei einer nichtblockierenden Sendeoperation im synchronen Modus die Kontrolle unmittelbar nach dem Start der Sendeoperation an den aufrufenden Prozess zurück, es findet also **keine Synchronisation** zwischen einem `MPI_Issend()` und dem zugehörigen `MPI_Irecv()` statt. Eine Synchronisation zwischen Sender und Empfänger findet dann statt, wenn der Sender eine `MPI_Wait()`-Operation ausführt. Wird eine `MPI_Wait()`-Operation auf eine Sendeoperation im synchronen Modus angewendet, so wird die Kontrolle erst dann an den Sender zurückgegeben, wenn der Empfänger die zugehörige `MPI_Irecv()`-Operation gestartet hat, d.h. es findet eine Synchronisation mit dem Empfänger statt.

- **Puffermodus:** Der Puffermodus legt fest, dass die Ausführung und die Beendigung einer Sendeoperation *nicht* von nichtlokalen Ereignissen beeinflusst werden darf, wie dies im synchronen Modus der Fall ist und im Standardmodus der Fall sein kann. Wird eine Sendeoperation im Puffermodus gestartet, so wird die Kontrolle auch dann zum aufrufenden Prozess zurückgegeben werden, wenn die zugehörige Empfangsoperation noch nicht gestartet wurde. Der Sender wartet also nicht auf den Start der zugehörigen Empfangsoperation. Wenn die zugehörige Empfangsoperation noch nicht gestartet wurde, muss das Laufzeitsystem die Nachricht in Puffern ablegen. Eine *blockierende Sendeoperation im Puffermodus* wird in MPI durch die Kommunikationsanweisung `MPI_Bsend()` ausgeführt. Diese Anweisung hat die gleichen Parameter mit gleicher Bedeutung wie `MPI_Send()`. Eine *nichtblockierende Sendeoperation im Puffermodus* wird durch die Anweisung `MPI_Ibsend()` gestartet, wobei die Parameter und deren Bedeutung mit den Parametern von `MPI_Isend()` übereinstimmen. Der Puffer für ein eventuelles Zwischenspeichern von Nachrichten *muss vom Programmierer zur Verfügung* gestellt werden. Damit ist der Programmierer auch dafür verantwortlich, dass die Größe des verwendeten Puffers ausreichend ist. Ein Puffer zum Zwischenspeichern der Nachrichten wird mit der Anweisung

```
int MPI_Buffer_attach (void *buffer, int buffersize)
```

zur Verfügung gestellt, wobei `buffersize` die Größe des Puffers `buffer` in Bytes angibt. Ein zuvor allozierter Puffer kann mit der Anweisung

```
int MPI_Buffer_detach (void *buffer, int *buffersize)
```

wieder freigegeben werden. Dabei blockiert der ausführende Prozess, bis alle im freizugebenden Puffer abgelegten Nachrichten zugestellt sind.

Man beachte, dass für die Empfangsoperationen nur der Standardmodus zur Verfügung steht.

5.1.2 Globale Kommunikationsoperationen

Eine *globale* Kommunikationsoperation ist eine Kommunikationsoperation, an der alle Prozesse eines parallelen Programms bzw. einer Teilmenge oder Gruppe dieser Prozesse beteiligt sind. In Abschnitt 3.7.2 haben wir häufig verwendete *globale* Kommunikationsoperationen vorgestellt. Diese globalen Kommunikationsoperationen werden in MPI durch kollektive Kommunikationsanweisungen (engl. *collective communication*) realisiert. Im Folgenden beschreiben wir, welche MPI-Anweisungen für die in Abschnitt 3.7.2 vorgestellten Kommunikationsaufgaben zur Verfügung stehen. Die Namen der Funktionen sind in folgender Tabelle zusammengestellt.

Globale Kommunikationsaufgabe	MPI-Funktion
Broadcastoperation	<code>MPI_Bcast()</code>
Akkumulationsoperation	<code>MPI_Reduce()</code>
Gatheroperation	<code>MPI_Gather()</code>
Scatteroperation	<code>MPI_Scatter()</code>
Multi-Broadcastoperation	<code>MPI_Allgather()</code>
Multi-Akkumulationsoperation	<code>MPI_Allreduce()</code>
Totaler Austausch	<code>MPI_Alltoall()</code>

Broadcastoperation. Bei einer *Broadcastoperation* schickt ein ausgewählter Prozess die gleichen Daten an alle anderen Prozesse einer Gruppe, vgl. Abschnitt 3.7.2. Eine Broadcastoperation wird in MPI mit der folgenden Anweisung ausgeführt:

```
int MPI_Bcast (void *message,
               int count,
               MPI_Datatype type,
               int root,
               MPI_Comm comm).
```

Dabei ist `message` für den Wurzelprozess `root` der Sendepuffer, der die zu verschickenden Daten enthält. Für alle anderen Prozesse bezeichnet `message` den Empfangspuffer. Der Wert `count` bezeichnet die Anzahl der Elemente vom Typ `type`, die vom Wurzelprozess verschickt werden. Wichtig ist, dass es sich bei einer Broadcastanweisung um eine *kollektive* Kommunikationsanweisung handelt, d.h. im Gegensatz zu Einzeltransferanweisungen ruft zur Ausführung einer Broadcastoperation jeder Prozess die gleiche Kommunikationsanweisung `MPI_Bcast()` auf. Jeder Prozess muss dabei den gleichen Prozess als `root` spezifizieren und muss den gleichen Kommunikator `comm` verwenden. Broadcast-Nachrichten können also *nicht* mit `MPI_Recv()` empfangen werden.

In der Parameterliste von `MPI_Bcast()` wird *keine* Markierungsinformation verwendet. Damit gibt es für die empfangenden Prozesse auch keine Möglichkeit, zwischen verschiedenen zu empfangenden Broadcast-Nachrichten zu unterscheiden. Die Semantik für Broadcast-Anweisungen besagt, dass Nachrichten in der gleichen Reihenfolge empfangen werden, in der sie abgesendet werden. Auch wenn die zusammengehörenden Broadcast-Anweisungen

verschiedener Prozesse nicht zur gleichen Zeit ausgeführt werden, wird diese Reihenfolge des Empfangens eingehalten. Abbildung 5.5 zeigt als Beispiel ein Programm, in dem Prozess 0 mit zwei aufeinanderfolgenden Broadcast-Anweisungen Daten *x* und *y* an die Prozesse 1 und 2 schickt, vgl. [119].

```

if (my_rank == 0) {
    MPI_Bcast (&x, 1, MPI_INT, 0, comm);
    MPI_Bcast (&y, 1, MPI_INT, 0, comm);
    local_work ();
}
else if (my_rank == 1) {
    local_work ();
    MPI_Bcast (&y, 1, MPI_INT, 0, comm);
    MPI_Bcast (&x, 1, MPI_INT, 0, comm);
}
else if (my_rank == 2) {
    local_work ();
    MPI_Bcast (&x, 1, MPI_INT, 0, comm);
    MPI_Bcast (&y, 1, MPI_INT, 0, comm);
}

```

Abb. 5.5. Beispiel zur Empfangsreihenfolge bei mehreren Broadcast-Anweisungen.

Prozess 1 führt zunächst lokale Berechnungen `local_work()` aus und speichert dann die zuerst empfangene Broadcast-Nachricht in seiner lokalen Variablen *y*, die zuletzt empfangene in *x*. Prozess 2 verwendet dagegen die gleiche Ablage wie der sendende Prozess 0. Somit wird Prozess 1 die Daten in anderen lokalen Variablen ablegen als Prozess 2. Obwohl keine synchrone Abarbeitung stattfindet, wird die Semantik einer synchronen Abarbeitung eingehalten, was im Beispiel an der umgekehrten Abspeicherung bei Prozess 1 deutlich wird. Voraussetzung für dieses Verhalten ist eine Systempufferung der MPI-Implementierung.

Kollektive Kommunikationsanweisungen sind in MPI immer *blockierend*, d.h. anders als für Einzeltransferanweisungen werden keine nichtblockierenden Kommunikationsanweisungen zur Verfügung gestellt. Der Grund dafür liegt im Wesentlichen darin, dass ansonsten die ohnehin schon große Anzahl von MPI-Anweisungen weiter angewachsen wäre. Aus dem gleichen Grund gibt es für globale Kommunikationsanweisungen nur den Standardmodus als Übertragungsmodus. Ein an einer globalen Kommunikationsoperation beteiligter Prozess kann die Ausführung seiner lokalen Kommunikationsanweisung beenden, sobald *seine* Beteiligung an der Operation abgeschlossen ist. Für den Wurzelprozess bedeutet dies, dass er auch dann mit der Ausführung anderer Operationen beginnen darf, wenn noch nicht alle Prozesse die verteilte Nachricht empfangen haben. Für einen Empfänger bedeutet dies, dass er die Ausführung anderer Operationen beginnen kann, sobald er die Broadcast-Nachricht in seinem lokalen Puffer empfangen hat, auch wenn andere Pro-

zesse die zugehörige `MPI_Bcast()`-Operation noch gar nicht gestartet haben. Somit beinhaltet die Ausführung einer globalen Kommunikationsanweisung nicht unbedingt eine Synchronisation.

Akkumulationsanweisungen. Bei einer *Akkumulationsoperation*, die auch als *globale Reduktionsoperation* bezeichnet wird, stellt jeder beteiligte Prozess Daten zur Verfügung, die mit Hilfe der angegebenen binären Operation verknüpft werden. Das Ergebnis wird von einem ausgezeichneten Wurzelprozess aufgesammelt, vgl. Abschnitt 3.7.2. Eine Akkumulationsoperation wird in MPI mit Hilfe der folgenden Anweisung ausgeführt:

```
int MPI_Reduce (void *sendbuf,
               void *recvbuf,
               int count,
               MPI_Datatype type,
               MPI_Op op,
               int root,
               MPI_Comm comm).
```

Dabei bezeichnet `sendbuf` einen Sendepuffer, in dem jeder beteiligte Prozess seine lokalen Daten für die Durchführung der Akkumulationsoperation zur Verfügung stellt. Das Ergebnis der Akkumulationsoperation unter Verwendung der Operation `op` wird im Empfangspuffer `recvbuf` abgelegt, der vom Wurzelprozess `root` zur Verfügung gestellt werden muss. Der Wert `count` bezeichnet die Anzahl der von jedem Prozess zur Verfügung gestellten Elemente vom Typ `type`. Jeder an der Akkumulationsoperation teilnehmende Prozess des Kommunikators `comm` ruft die Anweisung `MPI_Reduce()` auf, wobei `count`, `type`, `op` und `root` für alle Prozesse die gleichen Werte bezeichnen müssen. Weiterhin ist erforderlich, dass `sendbuf` und `recvbuf` beim Wurzelprozess *unterschiedliche* Datenbereiche bezeichnen.

Die Operation `op` bezeichnet eine assoziative *Reduktionsoperation*. MPI stellt eine Reihe von vordefinierten Reduktionsoperationen vom Typ `MPI_Op` zur Verfügung, die alle auch *kommutativ* sind. Der Programmierer hat aber auch die Möglichkeit, eigene assoziative (aber nicht notwendigerweise kommutative) Reduktionsoperationen zu definieren, worauf wir im Folgenden noch eingehen werden. Als vordefinierte Reduktionsoperationen stehen zur Verfügung:

Darstellung	Operation
<code>MPI_MAX</code>	Maximum
<code>MPI_MIN</code>	Minimum
<code>MPI_SUM</code>	Summe
<code>MPI_PROD</code>	Produkt
<code>MPI_LAND</code>	logisches Und
<code>MPI_BAND</code>	bitweises Und
<code>MPI_LOR</code>	logisches Oder
<code>MPI BOR</code>	bitweises Oder
<code>MPI_LXOR</code>	logisches exklusives Oder
<code>MPI_BXOR</code>	bitweises exklusives Oder
<code>MPI_MAXLOC</code>	maximaler Wert und dessen Index
<code>MPI_MINLOC</code>	minimaler Wert und dessen Index

Die Reduktionsoperationen `MPI_MAXLOC` und `MPI_MINLOC` können dazu verwendet werden, ein globales Maximum bzw. Minimum und einen zusätzlichen Index zu bestimmen. Wir werden diese Operation z.B. in Abschnitt 7.1 dazu verwenden, bei der Gauß-Elimination ein globales Pivotelement und den Prozess, der dieses Pivotelement besitzt, zu bestimmen. In diesem Fall ist der zusätzliche Index also die Nummer des Prozesses. Eine andere Anwendung könnte darin bestehen, in einem verteilt abgelegten Feld das maximale Element und dessen Index zu bestimmen, d.h. der zusätzliche Index wäre ein Feldindex. Die durch `MPI_MAXLOC` definierte Operation ist

$$(u, i) \circ_{max} (v, j) = (w, k)$$

$$\text{wobei } w = \max(u, v) \text{ und } k = \begin{cases} i, & \text{falls } u > v \\ \min(i, j), & \text{falls } u = v \\ j, & \text{falls } u < v \end{cases}.$$

Analog gilt für die von `MPI_MINLOC` definierte Operation:

$$(u, i) \circ_{min} (v, j) = (w, k)$$

$$\text{wobei } w = \min(u, v) \text{ und } k = \begin{cases} i, & \text{falls } u < v \\ \min(i, j), & \text{falls } u = v \\ j, & \text{falls } u > v \end{cases}.$$

Beide Operationen arbeiten also auf *Paaren von Werten*, die aus dem eigentlichen Wert und einem Index bestehen. Folglich muss der in `MPI_Reduce()` angegebene Datentyp `type` ein solches Paar repräsentieren. MPI stellt entsprechende Misch-Datentypen zur Verfügung:

<code>MPI_FLOAT_INT</code>	<code>(float, int)</code>
<code>MPI_DOUBLE_INT</code>	<code>(double, int)</code>
<code>MPI_LONG_INT</code>	<code>(long, int)</code>
<code>MPI_SHORT_INT</code>	<code>(short, int)</code>
<code>MPI_LONG_DOUBLE_INT</code>	<code>(long double, int)</code>
<code>MPI_2INT</code>	<code>(int, int)</code>

Beispiel: Ein Beispiel für die Anwendung einer Akkumulationsoperation mit `MPI_MAXLOC` ist in Abbildung 5.6 wiedergegeben, vgl. [151].

Es wird angenommen, dass jeder Prozess 30 Werte vom Typ `double` in einem Feld `ain` der Länge 30 zur Verfügung stellt. Die Aufgabe besteht darin, für jede der 30 Feldpositionen den maximalen Wert auf den verschiedenen Prozessen und die Nummer des Prozesses, der diesen Wert hat, zu bestimmen. Die Informationen werden bei Prozess 0 aufgesammelt, wobei die maximalen Werte in `aout` und die Prozessnummern in `ind` abgelegt werden sollen. Zur Durchführung des Aufsammelns muss eine Datenstruktur definiert werden, die aus Paaren von `double`- und `int`-Werten besteht. $\square$

Die Definition einer *benutzerdefinierten* Reduktionsoperation kann mit Hilfe der MPI-Anweisung

```

double ain[30], aout[30];
int ind[30];
struct {double val; int rank;} in[30], out[30];
int i, my_rank, root=0;

MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
for (i=0; i<30; i++) {
    in[i].val = ain[i];
    in[i].rank = my_rank;
}
MPI_Reduce(in,out,30,MPI_DOUBLE_INT,MPI_MAXLOC,root,MPI_COMM_WORLD);
if (my_rank == root)
    for (i=0; i<30; i++) {
        aout[i] = out[i].val;
        ind[i] = out[i].rank;
    }

```

Abb. 5.6. Beispiel für die Anwendung von `MPI_Reduce()` mit `MPI_MAXLOC` als Reduktionsoperation.

```

int MPI_Op_create (MPI_User_function *function,
                  int commute,
                  MPI_Op *op)

```

vorgenommen werden. Dabei ist `function` eine vom Programmierer zur Verfügung gestellte Funktion, die vier Parameter definieren muss: `void *in`, `void *out`, `int *len`, `MPI_Datatype *type`. Der Eintrag `commute` gibt an, ob es sich um eine kommutative Operation (`commute = 1`) handelt oder nicht (`commute = 0`). Der Parameter `op` des Aufrufs von `MPI_Op_create()` berechnet eine Operations-Datenstruktur, die später als Parameter an die Operation `MPI_Reduce()` übergeben werden kann.

Beispiel: Als weiteres Beispiel für den Einsatz einer `MPI_Reduce()`-Operation betrachten wir die parallele Berechnung des Skalarproduktes zweier Vektoren x und y der Länge m durch p Prozesse. Beide Vektoren werden in Blöcke der Größe `local_m = m/p` unterteilt, die jeweils einem Prozess lokal zugeordnet sind und in lokalen Vektoren `local_x` und `local_y` der Länge `local_m` abgespeichert sind. Es gilt also für jeden Prozess mit Nummer `my_rank` für $0 \leq j < \text{local\_m}$:

```

local_x[j] = x[j + my_rank * local_m];
local_y[j] = y[j + my_rank * local_m];

```

Abbildung 5.7 zeigt die Berechnung des inneren Produktes. Durch Ausführen des gleichen Programmtextes berechnet jeder der p Prozesse ein inneres Produkt für seine lokalen Teilvektoren und speichert dieses in `local_dot`. Durch die `MPI_Reduce()`-Operation werden diese lokalen Ergebnisse durch `MPI_SUM` addiert. Dazu trägt jeder Prozess sein lokales Ergebnis `local_dot` bei. Das Endergebnis liegt in `dot` bei Prozess 0 vor. □

```

int j, m, p, local_m;
float local_dot, dot;
float local_x[100], local_y[100];
MPI_Status status;

MPI_Init(&argc, &argv);
MPI_Comm_rank( MPI_COMM_WORLD, &my_rank);
MPI_Comm_size( MPI_COMM_WORLD, &p);
if (my_rank == 0) scanf("%d",&m);
local_m = m/p;
local_dot = 0.0;
for (j=0; j < local_m; j++)
    local_dot = local_dot + local_x[j] * local_y[j];
MPI_Reduce(&local_dot, &dot,1, MPI_FLOAT, MPI_SUM,0, MPI_COMM_WORLD);
MPI_Finalize();

```

Abb. 5.7. MPI-Programm zur Berechnung eines parallelen Vektorproduktes.

Gatheranweisung. Bei einer *Gatheroperation* stellt jeder an der Operation beteiligte Prozess Daten zur Verfügung, die (ohne Anwendung einer Reduktionsoperation) am angegebenen Wurzelprozess aufgesammelt werden, vgl. Abschnitt 3.7.2. Damit ist, anders als bei den Broadcast- und Akkumulationsoperationen, die Größe der beim Wurzelprozess aufgesammelten Nachricht größer als die der ursprünglich von jedem Prozess zur Verfügung gestellten Nachricht. Eine Gatheroperation wird in MPI mit Hilfe der folgenden Anweisung realisiert:

```

int MPI_Gather(void *sendbuf,
               int sendcount,
               MPI_Datatype sendtype,
               void *recvbuf,
               int recvcnt,
               MPI_Datatype recvtype,
               int root,
               MPI_Comm comm).

```

Dabei bezeichnet `sendbuf` einen Sendepuffer, in den jeder Prozess `sendcount` Elemente vom Typ `sendtype` zur Verfügung stellt; `recvbuf` bezeichnet den vom Wurzelprozess `root` zur Verfügung gestellten Empfangspuffer, in dem von jedem Prozess `recvcnt` Elemente vom Typ `recvtype` empfangen werden. Die Elemente werden in der Reihenfolge der Nummern der beteiligten Prozesse bzgl. des Kommunikators `comm` abgelegt. Der Effekt der gerade beschriebenen `MPI_Gather()`-Anweisung lässt sich auch dadurch erreichen, dass bei p beteiligten Prozessen jeder Prozess (einschließlich des Wurzelprozesses) die Anweisung

```
MPI_Send (sendbuf, sendcount, sendtype, root, my_rank, comm)
```

ausführt und dass der Wurzelprozess zusätzlich p Empfangsanweisungen

```
MPI_Recv (recvbuf+i*recvcount*extent,
          recvcount, recvtype, i, i, comm, &status)
```

ausführt, wobei *i* die Nummern aller beteiligten Prozesse durchläuft. Die von jedem Eintrag belegte Anzahl von Bytes, die in *extent* angegeben wird, erhält man durch den Aufruf `MPI_Type_extent(recvtype, &extent)`. Für ein korrektes Ausführen von `MPI_Gather()` muss jeder beteiligte Prozess den gleichen Prozess als Wurzelprozess *root* benennen. Zusätzlich muss der gleiche Datentyp und die gleiche Anzahl von Elementen angegeben werden. Abbildung 5.8 zeigt als Beispiel ein Programmfragment, in dem Prozess 0 von jedem anderen Prozess 100 Integerwerte einsammelt, vgl. [151].

```
MPI_Comm comm;
int sendbuf[100], my_rank, root = 0, gsize, *rbuf;
MPI_Comm_rank (comm, &my_rank);
if (my_rank == root) {
    MPI_Comm_size (comm, &gsize);
    rbuf = (int *) malloc (gsize*100*sizeof(int));
}
MPI_Gather(sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

Abb. 5.8. Beispiel für die Anwendung von `MPI_Gather()`.

Es gibt eine Verallgemeinerung `MPI_Gatherv()` von `MPI_Gather()`, die es erlaubt, dass jeder Prozessor eine evtl. *unterschiedliche* Anzahl von Elementen zur Verfügung stellt. Die Parameterliste von `MPI_Gather()` wird dahingehend verändert, dass der Integerparameter *recvcount* durch ein Integerfeld *recvcounts* ersetzt wird, dessen *i*-ter Eintrag festlegt, wie viele Elemente von Prozess *i* zur Verfügung gestellt werden. Zusätzlich muss nach diesem Parameter ein weiteres Integerfeld *displs* gleicher Länge angegeben werden, dessen *i*-ter Eintrag festlegt, an welcher Position des Empfangspuffers der von Prozess *i* empfangene Datenblock abgelegt werden soll. Die beiden Felder *recvcounts* und *displs* müssen nur vom Wurzelprozess mit den entsprechenden Werten belegt werden.

Der Effekt einer `MPI_Gatherv()`-Anweisung lässt sich dadurch beschreiben, dass jeder Prozess die oben beschriebene `MPI_Send()`-Anweisung ausführt und die vom Wurzelprozess ausgeführten Empfangsoperationen zu folgenden Operationen verallgemeinert werden:

```
MPI_Recv(recvbuf+displs[i]*extent, recvcounts[i], recvtype, i, i,
          comm, &status).
```

Für eine korrekte Funktionsweise der `MPI_Gatherv()`-Anweisung muss der von Prozess *i* angegebene Wert *sendcount* mit dem Wert *recvcounts[i]* übereinstimmen, der von der Wurzel angegeben wird. Weiterhin dürfen die vom Wurzelprozess angegebenen Felder *recvcounts* und *displs* nicht so

besetzt sein, dass Einträge des Empfangspuffers mehrfach besetzt werden, es dürfen beim Empfangen also *keine Überlappungen* auftreten. Abbildung 5.9 zeigt als Beispiel für die Anwendung einer `MPI_Gatherv()`-Anweisung eine Verallgemeinerung des Beispiels aus Abbildung 5.8, in der jeder Prozess weiterhin 100 Integerwerte zur Verfügung stellt, in der aber die Ablage im Empfangspuffer so geändert wird, dass zwischen den von den verschiedenen Prozessen eingesammelten Datenblöcken noch freie Datenblöcke liegen, deren Größe durch entsprechende Besetzung des Feldes `displs` variiert werden kann, vgl. [151]. In Abbildung 5.9 bleiben zwischen den Datenblöcken jeweils 10 freie Einträge. Für `stride < 100` tritt ein Fehler auf, da in diesem Fall beim Wurzelprozess Einträge des Empfangspuffers mehrfach beschrieben werden.

```

MPI_Comm comm;
int sbuf[100];
int my_rank, root = 0, gsize, *rbuf, *displs, *rcounts, stride=110;
MPI_Comm_rank (comm, &my_rank);
if (my_rank == root) {
    MPI_Comm_size (comm, &gsize);
    rbuf = (int *) malloc(gsize*stride*sizeof(int));
    displs = (int *) malloc(gsize*sizeof(int));
    rcounts = (int *) malloc(gsize*sizeof(int));
    for (i = 0; i < gsize; i++) {
        displs[i] = i*stride;
        rcounts[i] = 100;
    }
}
MPI_Gatherv(sbuf,100,MPI_INT,rbuf,rcounts,displs,MPI_INT,root,comm);

```

Abb. 5.9. Beispiel für die Anwendung von `MPI_Gatherv()`.

Scatteranweisung. Bei einer *Scatteroperation* stellt ein Wurzelprozess für jeden an der Operation beteiligten Prozess eventuell unterschiedliche Daten zur Verfügung, die durch Ausführen der Operation an die beteiligten Prozesse verteilt werden, vgl. Abschnitt 3.7.2. Eine Scatteroperation wird in MPI mit Hilfe der folgenden Anweisung realisiert:

```

int MPI_Scatter (void *sendbuf,
                int sendcount,
                MPI_Datatype sendtype,
                void *recvbuf,
                int recvcount,
                MPI_Datatype recvtype,
                int root,
                MPI_Comm comm).

```

Dabei bezeichnet `sendbuf` den vom Wurzelprozess `root` zur Verfügung gestellten Sendepuffer, in dem für jeden im Kommunikator `comm` liegenden Pro-

zess **sendcount** Elemente vom Typ **sendtype** zur Verfügung gestellt werden, wobei die Datenblöcke entsprechend der Nummern der Zielprozesse angeordnet sind. Die Zielprozesse empfangen die Datenblöcke im Empfangspuffer **recvbuf**, der auch vom Wurzelprozess zur Verfügung gestellt werden muss. Der Effekt einer **MPI_Scatter()**-Anweisung lässt sich dadurch nachbilden, dass der Wurzelprozess bei p beteiligten Prozessen p Sendeoperationen

```
MPI_Send (sendbuf+i*sendcount*extent, sendcount, sendtype, i, i, comm)
```

für $i = 0, \dots, p-1$ ausführt. Jeder der beteiligten Prozesse, einschließlich des Wurzelprozesses, führt eine Empfangsoperation

```
MPI_Recv (recvbuf, recvcount, recvtype, root, my_rank, comm, &status)
```

aus. Für eine korrekte Abarbeitung der **MPI_Scatter()**-Anweisung muss jeder Prozess die gleiche Wurzel, den gleichen Datentyp und die gleiche Anzahl von Elementen angeben.

Ebenso wie für die Gatheranweisung gibt es in MPI eine Vektorvariante **MPI_Scatterv()** zur Scatteroperation, mit der der Wurzelprozess jedem Prozess Datenblöcke mit *unterschiedlicher* Anzahl von Elementen schicken kann. Dazu wird die Parameterliste von **MPI_Scatter()** so abgeändert, dass der Integerparameter **sendcount** durch ein Integerfeld **sendcounts** ersetzt wird, dessen i -ter Eintrag festlegt, wie viele Elemente der Wurzelprozess an Prozess i schickt, $i = 0, \dots, p-1$. Zusätzlich muss nach diesem Parameter ein weiteres Integerfeld **displs** gleicher Länge übergeben werden, dessen i -ter Eintrag festlegt, von welcher Position des Sendepuffers Daten an Prozess i geschickt werden. Der Effekt dieser Operation kann ebenfalls durch einzelne Sende- und Empfangsoperationen erreicht werden. Dazu wird die vom Wurzelprozess ausgeführte Sendeoperation an Prozess i zu

```
MPI_Send (sendbuf+displs[i]*extent, sendcounts[i], sendtype, i, i, comm)
```

verallgemeinert. Die Empfangsoperationen sind dieselben wie oben angegeben.

Für eine korrekte Funktionsweise der **MPI_Scatterv()**-Anweisung muss der von der Wurzel angegebene Wert von **sendcounts[i]** mit dem von Prozess i angegebenen Wert **recvcount** übereinstimmen. Obwohl es für eine korrekte Funktionsweise nicht unbedingt erforderlich wäre, wird aus Gründen der Symmetrie zu **MPI_Gatherv()** gefordert, dass die Felder **sendcounts** und **displs** so besetzt sind, dass kein Eintrag des Sendepuffers zu mehr als einem Prozess geschickt wird. Das Programmfragment in Abbildung 5.10 beinhaltet eine Scatteroperation, in der Prozess 0 jeweils 100 Elemente an Prozess i schickt, wobei zwischen den verschiedenen Datenblöcken jeweils 10 Einträge liegen, die nicht verschickt werden, vgl. [151].

```

MPI_Comm comm;
int rbuf[100];
int my_rank, root = 0, gsize, *sbuf, *displs, *counts, stride=110;
MPI_Comm_rank (comm, &my_rank);
if (my_rank == root) {
    MPI_Comm_size (comm, &gsize);
    sbuf = (int *) malloc(gsize*stride*sizeof(int));
    displs = (int *) malloc(gsize*sizeof(int));
    counts = (int *) malloc(gsize*sizeof(int));
    for (i=0; i<gsize; i++) {
        displs[i] = i*stride; counts[i]=100;
    }
}
MPI_Scatterv(sbuf, counts, displs, MPI_INT, rbuf, 100, MPI_INT, root, comm);

```

Abb. 5.10. Beispiel für die Anwendung einer `MPI_Scatterv()`-Operation.

Multi-Broadcastanweisung. Bei einer *Multi-Broadcastoperation* stellt jeder beteiligte Prozess Daten für die Operation zur Verfügung, die z.B. Teilergebnisse von verteilt durchgeführten Berechnungen sind. Durch Ausführen der Operation werden die Daten aller beteiligten Prozesse aufgesammelt und jedem der Prozesse zur Verfügung gestellt. Man beachte, dass es *keinen* ausgezeichneten Wurzelprozess gibt, sondern alle beteiligten Prozesse gleichberechtigt an der Operation teilnehmen. Eine Multi-Broadcastoperation wird in MPI durch folgende Anweisung realisiert:

```

int MPI_Allgather (void *sendbuf,
                  int sendcount,
                  MPI_Datatype sendtype,
                  void *recvbuf,
                  int recvcnt,
                  MPI_Datatype recvtype,
                  MPI_Comm comm).

```

Dabei bezeichnet `sendbuf` einen von jedem Prozess zur Verfügung gestellten Sendepuffer, in dem die vom Prozess zur Verfügung gestellten Daten, bestehend aus `sendcount` Elementen vom Typ `sendtype`, abgelegt sind. Jeder Prozess stellt ebenfalls einen Empfangspuffer `recvbuf` zur Verfügung, in dem die von den Prozessen empfangenen Daten in der Reihenfolge der Prozessnummern abgelegt werden. Die Angaben in `sendcount` und `sendtype` entsprechen den Angaben in `recvcnt` und `recvtype`. Im folgenden Beispiel stellt jeder Prozess einen Sendepuffer mit 100 Integerzahlen zur Verfügung, die mit einer Multi-Broadcastoperation jedem Prozess verfügbar gemacht werden:

```

int sbuf[100], gsize, *rbuf;
MPI_Comm_size (comm, &gsize);
rbuf = (int*) malloc (gsize*100*sizeof(int));
MPI_Allgather (sbuf, 100, MPI_INT, rbuf, 100, MPI_INT, comm);

```

Ebenso wie für die `MPI_Gather()`-Anweisung muss bei einer `MPI_Allgather()`-Anweisung jeder Prozess einen Datenblock gleicher Größe mit Elementen gleichen Typs zur Verfügung stellen. Zum Einsammeln von Datenblöcken unterschiedlicher Größe (aber weiterhin gleichen Typs) steht wieder eine Vektorvariante zur Verfügung, die sich durch eine ähnliche Verallgemeinerung wie bei `MPI_Gatherv()` ergibt und durch folgende Anweisung realisiert wird:

```
int MPI_Allgatherv (void *sendbuf,
                    int sendcount,
                    MPI_Datatype sendtype,
                    void *recvbuf,
                    int *recvcounts,
                    int *displs,
                    MPI_Datatype recvttype,
                    MPI_Comm comm).
```

Multi-Akkumulationsanweisung. Bei einer *Multi-Akkumulationsoperation* führt jeder beteiligte Prozess eine (Einzel-)Akkumulationsoperation aus, wobei für jede dieser Akkumulationsoperationen von jedem Prozess unterschiedliche Daten zur Verfügung gestellt werden. MPI stellt zur Realisierung der Multi-Akkumulationsoperation nur eine Kommunikationsanweisung mit *eingeschränkter* Funktionalität zur Verfügung. Die Einschränkung besteht darin, dass jeder Prozess für die verschiedenen Akkumulationsoperationen die gleichen Daten zur Verfügung stellt. Diese eingeschränkte Funktionalität kann wie folgt veranschaulicht werden:

$$\begin{array}{ll}
 P_0 : x_0 & P_0 : x_0 + x_1 + \dots + x_{p-1} \\
 P_1 : x_1 & P_1 : x_0 + x_1 + \dots + x_{p-1} \\
 \vdots & \text{MPI-Akkumulation}(+) : \vdots \\
 P_{p-1} : x_n & P_{p-1} : x_0 + x_1 + \dots + x_{p-1}
 \end{array}$$

Die Einschränkung gegenüber der in Abschnitt 3.7.2 dargestellten Multi-Akkumulationsoperation besteht darin, dass jeder der Prozesse $P_0, \dots, P_{p-1}$ nur ein Datum x_k , $k = 0, \dots, p-1$, zur Verfügung stellt (ausgedrückt durch $P_k : x_k$) und nach Abschluss der Operation die *gleiche* Summe $x_0 + x_1 + \dots + x_{p-1}$ besitzt (ausgedrückt durch $P_k : x_0 + x_1 + \dots + x_{p-1}$). Somit hat eine Multi-Akkumulationsanweisung in MPI den gleichen Effekt wie eine (Einzel-)Akkumulationsanweisung, die von einer Broadcastanweisung gefolgt wird, welche die beim Wurzelprozess aufgesammelten Daten an alle Prozesse verteilt. Die MPI-Anweisung hat folgende Syntax:

```
int MPI_Allreduce (void *sendbuf,
                   void *recvbuf,
                   int count,
                   MPI_Datatype type,
                   MPI_Op op,
                   MPI_Comm comm).
```


Dabei ist `sendbuf` der lokale Puffer, in dem jeder zum Kommunikator `comm` gehörender Prozess seine lokalen Daten zur Verfügung stellt und `recvbuf` ist der lokale Puffer jedes Prozesses, in dem das Resultat aufgesammelt wird. Beide Puffer enthalten `count` Elemente vom Typ `type`. Die anzuwendende Reduktionsoperation wird durch `op` angegeben.

Beispiel: Als Beispiel für die Anwendung einer Multi-Akkumulationsoperation betrachten wir eine Matrix-Vektor-Multiplikation $Ab = c$ einer $n \times m$ -Matrix A mit einem m -dimensionalen Vektor b , deren Ergebnis im n -dimensionalen Vektor c abgelegt wird. Die Matrix A sei *spaltenblockweise* abgelegt, so dass jeder der p Prozesse $local\_m = m/p$ zusammenhängende Spalten lokal abgespeichert hat, wie wir es im Abschnitt 3.6 über Datenverteilungen vorgestellt haben. Entsprechend ist der Vektor b blockweise verteilt. Die parallele Matrix-Vektor-Multiplikation wird wie in Abschnitt 3.7.3 beschrieben durchgeführt, vgl. auch Abbildung 3.8. Abbildung 5.11 zeigt die Skizze eines zugehörigen MPI-Programms. Die lokal abgespeicherten Spaltenblöcke werden im zweidimensionalen Feld `a` abgelegt, das `n` Zeilen und `local_m` Spalten enthält. Jeder Prozess legt die ihm zugeordneten Spalten fortlaufend in diesem Feld ab. Das eindimensionale Feld `local_b` enthält jeweils einen zusammenhängenden Block von b mit der Länge `local_m`. Jeder Prozess berechnet n Teilskalarprodukte, die Teilvektoren der Länge `local_m` abdecken und die mit einer `MPI_Allreduce`-Anweisung zum Ergebnisvektor c aufaddiert werden, der somit allen Prozessen repliziert vorliegt. $\square$

```
int m, local_m, n, p;
float a[MAX_N][MAX_LOC_M], local_b[MAX_LOC_M];
float c[MAX_N], sum[MAX_N];
local_m = m/p;
for (i=0; i<n; i++) {
    sum[i] = 0;
    for (j=0; j<local_m; j++)
        sum[i] = sum[i] + a[i][j]*local_b[j];
}
MPI_Allreduce (sum, c, n, MPI_FLOAT, MPI_SUM, comm);
```

Abb. 5.11. Fragment eines MPI-Programms zur Berechnung einer Matrix-Vektor-Multiplikation mit spaltenblockweiser Abspeicherung der Matrix und unter Verwendung einer `MPI_Allreduce()`-Anweisung.

Gesamtaustausch. Bei einem Gesamtaustausch schickt jeder beteiligte Prozess an jeden anderen beteiligten Prozess eine eventuell unterschiedliche Nachricht, d.h. jeder beteiligte Prozess führt eine Scatteroperation (Sendersicht) bzw. eine Gatheroperation (Empfängersicht) mit jeweils unterschiedlichen Daten aus. Ein Gesamtaustausch wird in MPI mit Hilfe der folgenden Anweisung realisiert:

```
int MPI_Alltoall (void *sendbuf,
                  int sendcount,
                  MPI_Datatype sendtype,
                  void *recvbuf,
                  int recvcount,
                  MPI_Datatype recvtype,
                  MPI_Comm comm).
```

Dabei bezeichnet `sendbuf` einen Sendepuffer, in dem jeder Prozess für jeden anderen Prozess einen Datenblock mit `sendcount` Elementen vom Typ `sendtype` zur Verfügung stellt, wobei die Datenblöcke in der Reihenfolge der Prozessnummern der Zielprozesse abgelegt sind. Im Empfangspuffer `recvbuf` stellt jeder Prozess Platz zum Empfang der von den anderen Prozessen bereitgestellten Datenblöcke zur Verfügung, wobei die Blöcke in der Reihenfolge der Prozessnummern der sendenden Prozesse des Kommunikators `comm` abgelegt werden. Der Effekt eines totalen Austausches kann auch dadurch erreicht werden, dass jeder der beteiligten p Prozesse p Sendeoperationen

```
MPI_Send (sendbuf+i*sendcount*extent, sendcount, sendtype,
          i, my_rank, comm)
```

und p Empfangsoperationen

```
MPI_Recv (recvbuf+i*recvcount*extent, recvcount, recvtype,
          i, i, comm, &status)
```

ausführt, wobei i die Werte zwischen 0 und $p - 1$ durchläuft.

Zur korrekten Arbeitsweise der Operation muss jeder Prozess für jeden anderen Prozess Datenblöcke gleicher Größe zur Verfügung stellen und von jedem anderen Prozess Datenblöcke der gleichen Größe empfangen, d.h. `sendcount` und `recvcount` müssen von allen Prozessen mit dem gleichen Wert besetzt sein. Ebenso müssen `sendtype` und `recvtype` übereinstimmen. Sollen Datenblöcke unterschiedlicher Größe ausgetauscht werden, muss die **Vektorvariante** der Anweisung verwendet werden:

```
int MPI_Alltoallv (void *sendbuf,
                   int *scounts,
                   int *sdispls,
                   MPI_Datatype sendtype,
                   void *recvbuf,
                   int *rcounts,
                   int *rdispls,
                   MPI_Datatype recvtype,
                   MPI_Comm comm).
```

Für einen Prozess i gibt der Eintrag `scounts[j]` an, wie viele Elemente vom Typ `sendtype` Prozess i an Prozess j schickt und der Eintrag `sdispls[j]` gibt die Position dieses Datenblockes im Sendepuffer von Prozess i an. Der Eintrag `rcounts[j]` gibt für Prozess i an, wie viele Elemente vom Typ `recvtype` Prozess i von Prozess j empfängt, `rdispls[j]` gibt an, an welcher

Position im Empfangspuffer von Prozess i die Daten von Prozess j abgelegt werden sollen. Für eine korrekte Funktionsweise von `MPI_Alltoallv()` muss `scounts[j]` für Prozess i den gleichen Wert haben wie `rcounts[i]` für Prozess j . Bei p beteiligten Prozessen kann der Effekt von `MPI_Allgatherv()` auch dadurch erreicht werden, dass jeder Prozess p Sendeanweisungen

```
MPI_Send (sendbuf+sdispls[i]*sextent, counts[i],
          sendtype, i, my_rank, comm)
```

und p Empfangsoperationen

```
MPI_Recv (recvbuf+rdispls[i]*rextent, rcounts[i],
          recvtype, i, i, comm, &status)
```

ausführt, wobei i die Werte zwischen 0 und $p - 1$ durchläuft.

5.1.3 Auftreten von Deadlocks

Bei kollektiven Kommunikationsanweisungen können ähnlich wie bei den Einzeltransferoperationen in Abhängigkeit von der Benutzung von Systempuffern bei der MPI-Implementierung unterschiedliche Verhaltensweisen desselben Programms entstehen.

Bei unvorsichtigem Gebrauch kollektiver Kommunikationsanweisungen kann etwa ein **Deadlock** auftreten, wie wir anhand der `MPI_Bcast()`-Operation erläutern werden, vgl. [151]. Im folgenden Beispiel gehen wir von zwei beteiligten Prozessen aus, die zwei `MPI_Bcast()`-Operationen ausführen, wobei die Prozesse dies aber in unterschiedlicher Reihenfolge tun:

```
switch (my_rank) {
case 0: MPI_Bcast (buf1, count, type, 0, comm);
        MPI_Bcast (buf2, count, type, 1, comm);
        break;
case 1: MPI_Bcast (buf2, count, type, 1, comm);
        MPI_Bcast (buf1, count, type, 0, comm);
}
```

In diesem Programmfragment können zwei Fehler auftreten:

1. Wenn das Laufzeitsystem die beiden ersten `MPI_Bcast()`-Anweisungen jedes Prozesses aufeinander bezieht und als Teil *einer* Broadcastoperation auffasst, wird es einen *Fehler* feststellen, da die Anweisungen unterschiedliche Wurzelprozesse angeben.
2. Wenn das Laufzeitsystem die `MPI_Bcast()`-Anweisungen mit gleicher Wurzel aufeinander bezieht (wie es wahrscheinlich vom Programmierer beabsichtigt ist), tritt ein **Deadlock** auf, wenn keine oder zu kleine Systempuffer verwendet werden.

Der Grund dafür liegt darin, dass globale Kommunikationsanweisungen immer blockierend sind. *Ohne* Systempuffer sind die Anweisungen also synchronisierend, d.h. nach Aufruf der ersten `MPI_Bcast()`-Anweisung

blockiert Prozess 0 so lange, bis Prozess 1 die korrespondierende (zweite) `MPI_Bcast()`-Anweisung aufgerufen hat. Dazu kommt es aber nicht, da Prozess 1 blockiert, bis Prozess 0 die zweite `MPI_Bcast()`-Anweisung aufgerufen hat.

Das Auftreten der Fehler kann dadurch vermieden werden, dass alle beteiligten Prozesse die globalen Kommunikationsanweisungen in der gleichen Reihenfolge ausführen.

Auch beim Mischen von globalen Kommunikations- und Punkt-zu-Punkt-Anweisungen können Deadlocks auftreten, wie das folgende Beispiel zeigt:

```
switch (my_rank) {
case 0: MPI_Bcast (buf1, count, type, 0, comm);
        MPI_Send (buf2, count, type, 1, tag, comm);
        break;
case 1: MPI_Recv (buf2, count, type, 0, tag, comm, &status);
        MPI_Bcast (buf1, count, type, 0, comm);
}
```

Sind keine Systempuffer vorhanden, so tritt hier ein Deadlock auf, da ähnlich wie im ersten Beispiel dieses Abschnitts die Prozesse zyklisch aufeinander warten: Prozess 0 blockiert, bis Prozess 1 die `MPI_Bcast()`-Anweisung ausführt und Prozess 1 blockiert, bis Prozess 0 die `MPI_Send()`-Anweisung ausführt. Ein möglicher Deadlock kann auch hier dadurch vermieden werden, dass jeder Prozess die korrespondierenden globalen Kommunikations- bzw. Punkt-zu-Punkt-Anweisungen in der gleichen Reihenfolge ausführt.

Auch das *Synchronisationsverhalten* kollektiver Kommunikationsanweisungen hängt von der Verwendung oder Nichtverwendung von Systempuffern der MPI-Implementierung ab. Sind *keine* Systempuffer vorhanden, so können kollektive Kommunikationsanweisungen eine **Synchronisation** bewirken. Mit Systempuffern muss dies nicht der Fall sein. Als Beispiel betrachten wir das folgende Programmfragment:

```
switch (my_rank) {
case 0: MPI_Bcast (buf1, count, type, 0, comm);
        MPI_Send (buf2, count, type, 1, tag, comm);
        break;
case 1: MPI_Recv (buf2, count, type, MPI_ANY_SOURCE, tag,
                comm, &status);
        MPI_Bcast (buf1, count, type, 0, comm);
        MPI_Recv (buf2, count, type, MPI_ANY_SOURCE, tag,
                comm, &status);
        break;
case 2: MPI_Send (buf2, count, type, 1, tag, comm);
        MPI_Bcast (buf1, count, type, 0, comm);
}
```

Prozess 0 schickt nach Ausführen der `MPI_Bcast()`-Anweisung eine Nachricht mit `MPI_Send()` an Prozess 1, Prozess 2 schickt vor Ausführen der `MPI_Bcast()`-Anweisung eine Nachricht an Prozess 1. Prozess 1 empfängt

vor und nach Ausführen der `MPI_Bcast()`-Anweisung eine Nachricht mit `MPI_ANY_SOURCE`. Es stellt sich die Frage, welche Nachricht Prozess 1 mit welchem `MPI_Recv()` empfängt. Folgende zwei Abarbeitungsreihenfolgen sind möglich:

1. Prozess 1 empfängt zuerst die Nachricht von Prozess 2:

Prozess ₀	Prozess ₁	Prozess ₂
	<code>MPI_Recv()</code>	<code>MPI_Send()</code>
	$\Leftarrow$	
<code>MPI_Bcast()</code>	<code>MPI_Bcast()</code>	<code>MPI_Bcast()</code>
<code>MPI_Send()</code>	$\Rightarrow$	<code>MPI_Recv()</code>

Diese Abarbeitungsreihenfolge kann auch dann auftreten, wenn keine Systempuffer verwendet werden, d.h. wenn die Aufrufe der Kommunikationsanweisungen synchronisierend sind.

2. Prozess 1 empfängt zuerst die Nachricht von Prozess 0:

Prozess ₀	Prozess ₁	Prozess ₂
<code>MPI_Bcast()</code>		
<code>MPI_Send()</code>	$\Rightarrow$	<code>MPI_Recv()</code>
	<code>MPI_Bcast()</code>	
	<code>MPI_Recv()</code>	$\Leftarrow$
		<code>MPI_Send()</code>
		<code>MPI_Bcast()</code>

Diese Abarbeitungsreihenfolge kann *nur* auftreten, wenn Systempuffer verwendet werden, da ansonsten Prozess 0 seinen Aufruf von `MPI_Bcast()` nicht beenden kann, bevor Prozess 1 seinen korrespondierenden Aufruf gestartet hat.

Damit kann bei Verwendung von Systempuffern ein *nichtdeterministisches* Programm entstehen, das nur dann korrekt ist, wenn die beiden Abarbeitungsreihenfolgen zum gewünschten Ergebnis führen. Da dies meist nicht der Fall ist, sollte diese Situation vermieden werden.

Wie wir gesehen haben, beinhalten globale Kommunikationsanweisungen nur dann eine Synchronisation, wenn das Laufzeitsystem keine Systempuffer zum Zwischenspeichern von Nachrichten verwendet. Der Programmierer sollte die Korrektheit eines Programms daher nicht davon abhängig machen, dass die beteiligten Prozesse durch Ausführung der globalen Kommunikationsanweisungen synchronisiert werden. Eine sichere Synchronisation der Prozesse wird durch die Kommunikationsanweisung

```
MPI_Barrier (MPI_Comm comm)
```

erreicht, die bewirkt, dass alle zum Kommunikator `comm` gehörenden Prozesse blockiert werden, bis jeder andere zu diesem Kommunikator gehörende Prozess diese Funktion ebenfalls aufgerufen hat.

5.1.4 Prozessgruppen und Kommunikatoren

Ein wesentliches Konzept von MPI ist das Bilden von Teilmengen von Prozessen in Form von *Gruppen* oder *Kommunikatoren*. Eine **Prozessgruppe**

oder **Gruppe** ist eine geordnete Menge von Prozessen eines Anwenderprogramms, wobei jedem dieser Prozesse innerhalb der Gruppe eine eindeutig definierte Nummer zugeordnet ist, die als **Rang** (engl. *rank*) bezeichnet wird. Die Nummerierung der Prozesse einer Gruppe beginnt bei 0 und ist fortlaufend. Ein Prozess kann mehreren Gruppen angehören und hat bzgl. dieser Gruppen evtl. verschiedene Nummern. Die Darstellung und die Verwaltung von Gruppen wird vom MPI-System übernommen. Für das Benutzerprogramm sind Gruppen Objekte vom Typ **MPI_Group**, auf die nur über einen **Griff** (engl. *handle*) zugegriffen werden kann, der MPI-intern etwa als Index oder Zeiger realisiert ist. Prozessgruppen können zur Realisierung von taskparallelen Programmen verwendet werden und sind Voraussetzung für den Kommunikationsmechanismus in MPI.

In vielen Situationen ist es nützlich, die Prozesse eines parallelen Programms in disjunkte Teilmengen von Prozessen, also Gruppen, zu unterteilen, die dann unabhängige Aufgaben bearbeiten. Man spricht in diesem Zusammenhang auch von **Taskparallelismus**, vgl. Abschnitt 3.3.4. Obwohl taskparallele Programmteile auch dadurch realisiert werden können, dass die das Programm ausführenden Prozesse in Abhängigkeit von ihrer Prozessnummer verschiedene Prozeduren aufrufen und verschiedene Kommunikationsanweisungen ausführen, kann die Realisierung von Taskparallelismus durch Gruppen wesentlich erleichtert werden, wenn eine Kommunikationsbibliothek Kommunikationsanweisungen zur Unterstützung zur Verfügung stellt.

MPI bietet eine weitgehende Unterstützung von Prozessgruppen, die sich vor allem auch dadurch ausdrückt, dass globale Kommunikationsanweisungen auf vorher definierte Gruppen eingeschränkt werden können. Das ist u.a. wichtig zur Realisierung von Programmbibliotheken, da bei diesen streng zwischen den Kommunikationsanweisungen des Aufrufers und Kommunikationsanweisungen von Funktionen der Programmbibliotheken unterschieden werden muss. So könnte z.B. der Aufruf einer **MPI_Irecv()**-Anweisung unmittelbar vor Aufruf einer Bibliotheksfunktion zu einem Fehler führen, wenn der globale Kommunikator **MPI_COMM_WORLD** verwendet wird und wenn als Quelle **MPI_ANY_SOURCE** und als Markierung **MPI_ANY_TAG** angegeben wird. Ein Fehler kann in dieser Situation dann auftreten, wenn einer der Prozesse, die die Bibliotheksfunktion ausführen, zu Beginn von deren Abarbeitung Daten an den Prozess schickt, der die obige **MPI_Irecv()**-Anweisung ausgeführt hat, denn in diesem Fall könnte dieser Prozess mit der mit **MPI_Irecv()** gestarteten Kommunikationsoperationen evtl. die innerhalb der Bibliotheksfunktion abgeschickte Nachricht empfangen.

Jede Punkt-zu-Punkt-Kommunikation und jede kollektive Kommunikation in MPI findet innerhalb eines **Kommunikationsgebietes** statt. Ein solches globales Kommunikationsgebiet basiert auf einer Prozessgruppe (nutzt also auch deren Prozessnummerierung) und wird auf den Prozessen *lokal* durch **Kommunikatoren** dargestellt. In MPI gehört zu jeder Prozessgruppe auch ein Kommunikator, jeder Kommunikator legt eine Prozessgruppe

fest. Kommunikatoren eines Kommunikationsgebietes haben alle dasselbe Gruppenattribut und einen zusätzlichen Kontext, der MPI-intern z.B. durch einen Namen für das Kommunikationsgebiet dargestellt sein kann. Ein Kommunikator kennt alle anderen Kommunikatoren des Kommunikationsgebietes, was intern zur Durchführung der Kommunikationsoperationen benötigt wird. MPI-intern kann dies etwa durch eine lokale Darstellung der Gruppe als Feld realisiert sein, in dem Eintrag k die Prozessnummer des Prozesses mit $rank = k$ enthält.

Für den Programmierer ist ein **Kommunikator** ein nicht direkt zugreifbares (engl. *opaque*) Datenobjekt vom Typ `MPI_Comm`, das als **Intra-Kommunikator** oder **Inter-Kommunikator** vereinbart werden kann. Intra-Kommunikatoren erlauben die Ausführung beliebiger kollektiver Kommunikationsanweisungen auf *einer* Gruppe von Prozessen. Inter-Kommunikatoren erlauben die Ausführung von Punkt-zu-Punkt-Kommunikation zwischen zwei Prozessgruppen. Im Folgenden beschränken wir uns auf Intra-Kommunikatoren und bezeichnen einen Intra-Kommunikator kurz als Kommunikator.

In den vorangehenden Abschnitten haben wir für Kommunikationsanweisungen den vordefinierten Kommunikator `MPI_COMM_WORLD` verwendet, der alle an der Abarbeitung des parallelen Programms beteiligten Prozesse umfasst. Zur Konstruktion weiterer Prozessgruppen und Kommunikatoren stellt MPI eine Reihe von Operationen zur Verfügung, die immer auf bereits existierenden Gruppen bzw. Kommunikatoren beruhen. Ein Ausgangspunkt kann der vordefinierte Kommunikator `MPI_COMM_WORLD` bzw. die zugehörige Gruppe sein. Die zu einem Kommunikator gehörende Prozessgruppe erhält man mit Hilfe des Aufrufes von

```
int MPI_Comm_group (MPI_Comm comm, MPI_Group *group),
```

wobei `comm` den gegebenen Kommunikator bezeichnet und `group` ein Zeiger auf ein vorher deklariertes Datenobjekt vom Typ `MPI_Group` ist, das durch den Aufruf mit Werten belegt wird. Eine vordefinierte Gruppe ist `MPI_GROUP_EMPTY`, die eine Gruppe ohne Prozesse bezeichnet.

Operationen für Prozessgruppen. MPI stellt Operationen zur Verfügung, die ausgehend von einer existierenden Gruppe den Aufbau von neuen Gruppen gestatten. Dabei kann die vordefinierte leere Gruppe `MPI_GROUP_EMPTY` verwendet werden. Die **Vereinigung** von zwei existierenden Gruppen `group1` und `group2` wird durch die Anweisung

```
int MPI_Group_union (MPI_Group group1,
                    MPI_Group group2,
                    MPI_Group *new_group)
```

erreicht. Dabei wird die Nummerierung in der neuen Gruppe `new_group` so vorgenommen, dass die Prozesse in `group1` ihre Nummerierung aus `group1` beibehalten, während die Prozesse aus `group2`, die nicht in `group1` enthalten sind, die darauffolgenden Nummern fortlaufend erhalten. Die **Schnittmenge** von zwei Gruppen wird durch die Anweisung

```
int MPI_Group_intersection (MPI_Group group1,
                           MPI_Group group2,
                           MPI_Group *new_group)
```

erreicht, wobei in **new_group** die Reihenfolge aus **group1** übernommen wird. Die Prozesse in **new_group** erhalten dabei eine fortlaufende, mit 0 beginnende Nummerierung. Die **Differenz** von zwei Gruppen erreicht man durch die Anweisung

```
int MPI_Group_difference (MPI_Group group1,
                         MPI_Group group2,
                         MPI_Group *new_group),
```

wobei ebenfalls die Reihenfolge aus **group1** übernommen wird. Eine **Untergruppe** einer existierenden Gruppe kann mit der Anweisung

```
int MPI_Group_incl (MPI_Group group,
                   int p,
                   int *ranks,
                   MPI_Group *new_group)
```

gebildet werden, wobei **ranks** ein Integerfeld mit **p** Elementen ist. Der Aufruf dieser Anweisung erzeugt eine neue Gruppe **new_group** mit **p** Prozessen, die von 0 bis **p-1** durchnummeriert sind. Prozess **i** ist der Prozess, der in der vorgegebenen Gruppe **group** die Nummer **ranks[i]** hatte. Für eine korrekte Ausführung der Anweisung ist es erforderlich, dass **group** mindestens **p** Mitglieder hat und dass die Werte **ranks[i]** für $0 \leq i < p$ gültige Prozessnummern in **group** bezeichnen, die voneinander verschieden sind.

Das **Löschen** von Prozessen aus einer Gruppe kann mit der Anweisung

```
int MPI_Group_excl (MPI_Group group,
                   int p,
                   int *ranks,
                   MPI_Group *new_group)
```

erreicht werden. Die Ausführung dieser Anweisung erzeugt eine neue Gruppe **new_group**, die aus der gegebenen Gruppe **group** dadurch entsteht, dass die Prozesse mit den Nummern **ranks[0],...,ranks[p-1]** aus **group** gelöscht werden. Ähnlich wie für **MPI_Group_incl()** müssen alle Einträge **ranks[i]** gültige, unterschiedliche Prozessnummern in **group** bezeichnen.

Da die Datenstruktur vom Typ **MPI_Group** zur Beschreibung von Prozessgruppen vom Programmierer nicht direkt zugreifbar ist, stellt MPI Anweisungen zur Verfügung, mit deren Hilfe Informationen über die Prozessgruppe abgefragt werden können. Die Größe einer Gruppe **group** erhält man durch die Anweisung

```
int MPI_Group_size (MPI_Group group, int *size) ,
```

die die Größe in **size** zurückliefert. Die Nummer des aufrufenden Prozesses in der angegebenen Gruppe **group** erhält man mit der Anweisung:


```
int MPI_Group_rank (MPI_Group group, int *rank).
```

Die Gleichheit von Prozessgruppen kann mit der Anweisung

```
int MPI_Group_compare (MPI_Group group1, MPI_Group group2, int *res)
```

überprüft werden. Dabei besagt der Rückgabewert `res = MPI_IDENT`, dass die beiden angegebenen Gruppen `group1` und `group2` die gleichen Prozesse in der gleichen Reihenfolge enthalten. Der Wert `res = MPI_SIMILAR` besagt, dass die beiden Gruppen zwar die gleichen Prozesse enthalten, dass die Reihenfolge der Prozesse in den Gruppen aber unterschiedlich ist. Der Wert `res = MPI_UNEQUAL` besagt, dass die Gruppen unterschiedliche Prozesse enthalten.

Mit der Anweisung

```
int MPI_Group_free (MPI_Group *group)
```

wird eine Gruppe `group` freigegeben, wobei der Griff auf `MPI_GROUP_NULL` gesetzt wird.

Operationen für Kommunikatoren. Die Erzeugung eines (Intra-)Kommunikators zu einer Gruppe von Prozessen wird durch die Anweisung

```
int MPI_Comm_create (MPI_Comm comm,
                    MPI_Group group,
                    MPI_Comm *new_comm)
```

erreicht. Dabei muss `group` eine Teilmenge der Gruppe sein, die mit dem (alten) Kommunikator `comm` assoziiert ist. Für eine korrekte Ausführung ist es weiterhin erforderlich, dass *jeder* zu `comm` gehörende Prozess den Aufruf von `MPI_Comm_create()` ausführt und dass jeder ausführende Prozess die *gleiche* Gruppe als Argument angibt. Das Resultat des Aufrufes besteht darin, dass jeder zu `group` gehörende Prozess einen Zeiger auf einen neuen Kommunikator `new_comm` zurückerhält. Jeder nicht zu `group` gehörende Prozess erhält `MPI_COMM_NULL` als Rückgabewert in `new_comm`.

Informationen über Kommunikatoren erhält man durch die folgenden MPI-Anweisungen, die als lokale Operationen keine Kommunikation erfordern, also schnell auszuführen sind. Die Größe der mit einem Kommunikator assoziierten Gruppe erhält man durch Ausführung der Anweisung:

```
int MPI_Comm_size (MPI_Comm comm, int *size).
```

Die Größe wird dabei in `size` zurückgeliefert. Die Nummer des aufrufenden Prozesses in der mit einem Kommunikator assoziierten Gruppe erhält man mit Hilfe der Anweisung

```
int MPI_Comm_rank (MPI_Comm comm, int *rank).
```

Diese Anweisung wurde in den bisherigen Beispielen immer mit dem Kommunikator `MPI_COMM_WORLD` aufgerufen. Der Vergleich von Kommunikatoren kann mit Hilfe der Anweisung

```
int MPI_Comm_compare (MPI_Comm comm1, MPI_Comm comm2, int *res)
```

erfolgen. Dabei bedeutet `res = MPI_IDENT`, dass `comm1` und `comm2` dieselbe Kommunikator-Datenstruktur bezeichnen. Der Rückgabewert `res = MPI_CONGRUENT` bedeutet, dass `comm1` und `comm2` Kommunikatoren bezeichnen, deren zugeordnete Gruppen die gleichen Prozesse in der gleichen Reihenfolge enthalten. Ähnlich wie für `MPI_Group_compare()` besagt `res = MPI_SIMILAR`, dass die zugehörigen Gruppen die gleichen Prozesse in unterschiedlicher Reihenfolge enthalten, und `res = MPI_UNEQUAL` bedeutet, dass die Gruppen unterschiedliche Prozesse enthalten.

Zur direkten Konstruktion von Kommunikatoren stellt MPI Anweisungen zum Duplizieren, Löschen und Aufspalten von Kommunikatoren zur Verfügung. Das **Duplizieren** eines Kommunikators wird mit der Anweisung

```
int MPI_Comm_dup (MPI_Comm comm, MPI_Comm *new_comm)
```

erreicht, die einen neuen Kommunikator `new_comm` mit den gleichen Eigenschaften (zugeordnete Gruppe und Topologie) wie `comm` erzeugt, der aber auf einem neuen, unterschiedlichen Kommunikationsgebiet arbeitet. Damit hat der Programmierer z.B. die Möglichkeit, vor dem Aufruf einer parallelen Bibliotheksfunktion durch Duplizieren des Kommunikators die Kommunikation innerhalb der Bibliotheksfunktion vollständig von der Kommunikation außerhalb der Bibliotheksfunktion zu trennen. Zum **Löschen** eines Kommunikators kann die Anweisung

```
int MPI_Comm_free (MPI_Comm *comm)
```

verwendet werden. Der Effekt der Anweisung besteht darin, dass die Datenstruktur des Kommunikators `comm` freigegeben wird, sobald alle Kommunikationsanweisungen, die mit `comm` ausgeführt wurden, abgeschlossen sind. Diese Anweisung kann z.B. verwendet werden, um einen Kommunikator, der für den Aufruf einer Bibliotheksfunktion dupliziert wurde, nach dem Beenden des Aufrufs wieder zu löschen. In diesem Zusammenhang sei auch darauf hingewiesen, dass Kommunikatoren nicht mit einfachen Zuweisungen (`comm2 = comm1`) zugewiesen werden sollten, da in diesem Fall die Anweisung `MPI_Comm_free(comm2)` auch den Kommunikator `comm1` beeinflussen würde, obwohl dies evtl. nicht beabsichtigt ist. Das **Aufspalten** von Kommunikatoren kann mit der Anweisung

```
int MPI_Comm_split (MPI_Comm comm,
                    int color,
                    int key,
                    MPI_Comm *new_comm)
```

erreicht werden. Der Effekt dieser Anweisung besteht darin, dass die dem Kommunikator `comm` zugeordnete Prozessgruppe in so viele disjunkte Untergruppen aufgeteilt wird wie es unterschiedliche Werte von `color` gibt. Dabei enthält eine Untergruppe jeweils alle Prozesse, die den gleichen Wert von

`color` angeben. Die Ordnung der Prozesse innerhalb der Untergruppen wird durch die Werte von `key` bestimmt. Bei gleichen Werten von `key` entscheidet die Reihenfolge in der ursprünglichen Gruppe. Wenn ein Prozess `color = MPI_UNDEFINED` angibt, gehört er keiner der Untergruppen an. Die erzeugten Untergruppen sind für die beteiligten Prozesse nicht direkt über eine Datenstruktur vom Typ `MPI_Group` verfügbar. Stattdessen erhält jeder beteiligte Prozess einen Zeiger auf den Kommunikator der Untergruppe, dem er nach der Operation angehört.

Beispiel: Wir betrachten eine Gruppe mit zehn Prozessen, die die Anweisung `MPI_Comm_split()` ausführen, wobei die Prozesse die Argumentwerte der folgenden Tabelle angeben:

Prozess	a	b	c	d	e	f	g	h	i	j
Rang	0	1	2	3	4	5	6	7	8	9
color	0	$\perp$	3	0	3	0	0	5	3	$\perp$
key	3	1	2	5	1	1	1	2	1	0

Durch Ausführung der Anweisung werden drei Kommunikatorgruppen $\{f, g, a, d\}$, $\{e, i, c\}$ und $\{h\}$ erzeugt, die die Prozesse in der angegebenen Reihenfolge enthalten. Der Eintrag $\perp$ in der Beispieltabelle bezeichnet `color = MPI_UNDEFINED`. $\square$

Die Anweisung `MPI_Comm_split()` kann z.B. dazu verwendet werden, die Ausführung von taskparallelen Berechnungen vorzubereiten. Die erzeugten Kommunikatoren werden dazu gebraucht, die Kommunikation in den taskparallelen Berechnungen der Gruppen durchzuführen, womit dann eine eindeutige Trennung der Kommunikationsoperationen erreicht wird.

5.1.5 Prozesstopologien

Wie bereits erwähnt, hat jeder Prozess einer Prozessgruppe eine eindeutige Prozessnummer bzgl. dieser Gruppe, die zur Kommunikation mit diesem Prozess verwendet werden kann. Obwohl über die Prozessnummer jeder Prozess eindeutig bestimmt werden kann, ist es jedoch oft nützlich, die Prozesse auch über eine andere Bezeichnungsweise ansprechen zu können. Dies ist z.B. dann der Fall, wenn ein Algorithmus auf einem zwei- oder dreidimensionalen virtuellen Gitter arbeitet und jeder einem Gitterpunkt zugeordnete Prozess Daten mit den Nachbarprozessen austauscht. In dieser Situation wäre es nützlich, wenn die Prozesse auch über zwei- oder dreidimensionale Koordinaten angesprochen werden können, weil damit jeder Prozess seine Nachbarprozesse sehr einfach bestimmen kann. Um dies zu unterstützen erlaubt MPI zu jedem (Intra-)Kommunikator die Definition einer **virtuellen Topologie**, die für die Kommunikation innerhalb der zugeordneten Prozessgruppe ver-

wendet werden kann. Mit der folgenden MPI-Anweisung kann ein virtuelles Gitter beliebiger Dimension definiert werden:

```
int MPI_Cart_create (MPI_Comm comm,
                    int ndims,
                    int *dims,
                    int *periods,
                    int reorder,
                    MPI_Comm *new_comm).
```

Dabei bezeichnet `comm` den ursprünglichen Kommunikator ohne Topologie, `ndims` gibt die Anzahl der Dimensionen des zu erzeugenden Gitters an, `dims` bezeichnet ein Feld mit `ndims` Einträgen, wobei `dims[i]` die Gesamtanzahl der Prozessoren in Dimension i angibt.

Die Einträge von `dims` müssen dabei so belegt sein, dass das Produkt aller Einträge der Anzahl der Prozesse entspricht, die in den neuen Kommunikator aufgenommen werden sollen. Insbesondere darf das Produkt der Einträge die Anzahl der Prozesse des ursprünglichen Kommunikators `comm` nicht übersteigen. Das Feld `periods` der Länge `ndims` gibt für jede Dimension an, ob das Gitter in dieser Dimension zyklisch verbunden werden soll (Eintrag 1 oder `true`) oder nicht (Eintrag 0 oder `false`). Wenn `reorder = false` angegeben wird, haben die Prozesse in `new_comm` die gleiche Reihenfolge wie in `comm`. Wird `reorder = true` angegeben, so kann das Laufzeitsystem die Prozesse umordnen, um z.B. eine gute Einbettung des virtuellen Gitters in das physikalische Netzwerk des Parallelrechners zu erreichen.

Beispiel: Sei `comm` ein Kommunikator mit zwölf Prozessen. Mit der Voraussetzung `dims[0] = 3, dims[1] = 4, period[0] = period[1] = 0, reorder = 0` wird durch den Aufruf

```
MPI_Cart_create (comm, 2, dims, period, reorder, &new_comm)
```

ein virtuelles 3×4 -Gitter definiert, in dem die Prozesse entsprechend folgender Darstellung angeordnet sind:

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)

Für jeden Prozess sind seine Nummer und seine kartesischen Koordinaten angegeben. Die kartesischen Koordinaten sind in der Form (Zeile,Spalte) angegeben, entsprechen also der Nummerierung einer Matrix. Die Prozesse werden entsprechend ihrer Nummer *rank* im Kommunikator `comm` zeilenweise in aufsteigender Zeilennummerierung zugeordnet. $\square$

Um den Programmierer bei der Aufteilung der Prozesse auf die verschiedenen Dimensionen zu unterstützen, stellt MPI eine Hilfsfunktion

```
int MPI_Dims_create (int nnodes, int ndims, int *dims)
```

zur Verfügung. Dabei bezeichnet `ndims` die Anzahl der Dimensionen des zu definierenden Gitters, `nnodes` ist die Anzahl der Prozesse im Gitter, `dims` ist ein Feld der Länge `ndims`, in dem für jede Dimension die Anzahl der Prozesse für diese Dimension abgelegt werden. Falls beim Aufruf von `MPI_Dims_create()` der i -te Eintrag von `dims` auf `dims[i] = 0` gesetzt ist, enthält `dims[i]` nach dem Aufruf die Anzahl der Prozesse, die Dimension i zugeordnet werden, $i = 0, \dots, ndims-1$, wobei das MPI-System versucht, jeder Dimension gleichviele Prozesse zuzuordnen, damit ein möglichst quadratisches Gitter entsteht. Falls der Programmierer für eine bestimmte Dimension i die Anzahl der Prozesse vorgeben will, so kann er dies dadurch tun, dass er vor dem Aufruf von `MPI_Dims_create()` den zugehörigen Eintrag von `dims` mit der gewünschten Anzahl von Prozessen vorbelegt. In diesem Fall wird der Aufruf `MPI_Dims_create()` den Eintrag unverändert lassen und die anderen, nicht vorbelegten Einträge von `dims` entsprechend anpassen.

Die Definition eines virtuellen Gitters führt dazu, dass jeder Prozess neben seiner Prozessnummer auch eine Position im Gitter hat, die durch kartesische Koordinaten ausgedrückt wird. Für die Umrechnung zwischen den kartesischen Koordinaten und den Prozessnummern stellt MPI zwei Funktionen zur Verfügung. Die Anweisung

```
int MPI_Cart_rank (MPI_Comm comm, int *coords, int *rank)
```

wandelt die im Feld `coords` zur Verfügung gestellten kartesischen Koordinaten entsprechend des für `comm` definierten virtuellen Gitters in Prozessnummern um. Die Anweisung

```
int MPI_Cart_coords (MPI_Comm comm,
                    int rank,
                    int ndims,
                    int *coords)
```

wandelt umgekehrt die in `rank` zur Verfügung gestellte Prozessnummer in kartesische Koordinaten des virtuellen Gitters um, die im Feld `coords` zurückgeliefert werden. Dabei bezeichnet `ndims` die Anzahl der Dimensionen im virtuellen Gitter, das für `comm` definiert wurde.

Das Ziel der Definition eines virtuellen Gitters liegt in einer vereinfachten Ermittlung der Kommunikationspartner von Prozessen. Die Kommunikation mit Nachbarprozessen in einem Gitter ist ein häufiges in Anwendungsalgorithmen vorkommendes Kommunikationsmuster. MPI stellt eine Anweisung zur Ermittlung der Nachbarprozesse in jeder Dimension des Gitters zur Verfügung:

```
int MPI_Cart_shift (MPI_Comm comm,
                  int dir,
                  int displ,
                  int *rank_source,
                  int *rank_dest).
```

Hierbei bezeichnet `dir` die Dimension, in deren Richtung der Nachbarprozess ermittelt werden soll. Der Eintrag `displ` gibt den gewünschten Abstand an, wobei ein positiver Wert die Nachbarn in Richtung steigender Koordinatenwerte und ein negativer Wert die Nachbarn in Richtung fallender Koordinatenwerte bezeichnet, d.h. `displ = -1` bezeichnet den unmittelbaren Nachbarn, der dem aufrufenden Prozess in der Dimension `dir` vorangeht, `displ = 1` bezeichnet den direkten Nachbarn. Das Resultat des Aufrufs von `MPI_Cart_shift()` besteht darin, dass in `rank_dest` die Nummer des Nachbarprozesses in der angegebenen Dimension und dem angegebenen Abstand zurückgegeben wird. In `rank_source` wird die Nummer des Prozesses zurückgegeben, für den der aufrufende Prozess der Nachbar in der angegebenen Dimension und Abstand ist. Die in `rank_dest` und `rank_source` angegebenen Prozessnummern können dann z.B. als Parameter für einen Aufruf von `MPI_Sendrecv()` verwendet werden.

Beispiel: Als Beispiel betrachten wir zwölf Prozesse in einem 3×4 -Gitter mit zyklischer Verbindung, vgl. [151]. Jeder der Prozesse speichert einen Floating-Point-Wert, den er mit seinen Nachbarprozessen in Dimension 0 austauscht:

```
int coords[2], dims[2], periods[2], source, dest, my_rank, reorder;
MPI_Comm comm_2d;
MPI_Status status;
float a, b;
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
dims[0] = 3; dims[1] = 4;
periods[0] = periods[1] = 1;
reorder = 0;
MPI_Cart_create (MPI_COMM_WORLD, 2, dims, periods, reorder, &comm_2d);
MPI_Cart_coords (comm_2d, my_rank, 2, coords);
MPI_Cart_shift (comm_2d, 0, coords[1], &source, &dest);
a = my_rank;
MPI_Sendrecv (&a, 1, MPI_FLOAT, dest, 0, &b, 1, MPI_FLOAT,
              source, 0, comm_2d, &status);
```

In diesem Beispiel steigt der Abstand der Nachbarn in der 0-ten Dimension mit den Koordinaten in der 1-ten Dimension, da `displs = coord[1]` ist, so dass für die verschiedenen Spalten des Gitters ein unterschiedlicher Austausch stattfindet. Die Operation `MPI_Cart_shift()` bestimmt die für die Operation `MPI_Sendrecv()` benötigten Werte für `dest` und `source`. Im folgenden Diagramm sind diese Werte jeweils als dritte Angabe nach Rang und Gitterposition in der Form `source|dest` angegeben. Für den Prozess mit `rank=5` gilt z.B. `coords[1]=1` und daher `source = 9` (also unterer Nachbar in Dimension 0) und `dest = 1` (also oberer Nachbar in Dimension 0).

0 (0,0) 0 0	1 (0,1) 9 5	2 (0,2) 6 10	3 (0,3) 3 3
4 (1,0) 4 4	5 (1,1) 1 9	6 (1,2) 10 2	7 (1,3) 7 7
8 (2,0) 8 8	9 (2,1) 5 1	10 (2,2) 2 6	11 (2,3) 11 11

□

Wenn für einen Kommunikator ein virtuelles Gitter definiert wurde, kann dieses in Untergitter zerlegt werden. MPI stellt dazu die Anweisung

```
int MPI_Cart_sub (MPI_Comm comm,
                  int *remain_dims,
                  MPI_Comm *new_comm)
```

zur Verfügung. Dabei bezeichnet `comm` den Kommunikator, für den das virtuelle Gitter definiert wurde, und `new_comm` bezeichnet den neuen Kommunikator, für den als Topologie ein Untergitter des ursprünglichen Gitters vereinbart wird. Die Zerlegung in Untergitter wird über das Feld `remain_dims` gesteuert, das für jede Dimension des ursprünglichen Gitters einen Eintrag enthält. Wenn `remain_dims[i] = 1` gesetzt ist, so bedeutet dies, dass die i -te Dimension im Untergitter erhalten bleibt; `remain_dims[i] = 0` besagt, dass die i -te Dimension im Untergitter nicht existiert. Existiert eine Dimension i im Untergitter nicht, so gibt die Größe in Dimension i an, wie viele Untergitter bzgl. dieser Dimension entstanden sind. Wenn jeder der Prozesse des Kommunikators `comm` die Anweisung `MPI_Cart_sub()` ausführt, erhält jeder der Prozesse einen neuen Kommunikator `new_comm` zurück, dem er angehört und auf dem die resultierende Gitterstruktur definiert ist. Die Dimension der Untergitter ergibt sich aus der Anzahl der Dimensionen, für die `remain_dims[i] = 1` gesetzt wurde. Die Anzahl der verschiedenen Kommunikatoren ergibt sich aus dem Produkt der Anzahlen der Prozesse in den Dimensionen, für die `remain_dims[i] = 0` gesetzt wurde.

Beispiel: Wir nehmen an, dass `comm` ein Kommunikator ist, für den ein $(2 \times 3 \times 4)$ -Gitter definiert wurde. Bei der Besetzung `remain_dims = (1, 0, 1)` erzeugt der Aufruf

```
int MPI_Cart_sub (comm_3d, remain_dims, &new_comm)
```

drei verschiedene Kommunikatoren mit jeweils einem (2×4) -Gitter, vgl. Abbildung 5.12. □

Informationen über ein für einen Kommunikator definiertes Gitter können mit den folgenden beiden MPI-Anweisungen erhalten werden. Die Anweisung

```
int MPI_Cartdim_get (MPI_Comm comm, int *ndims)
```

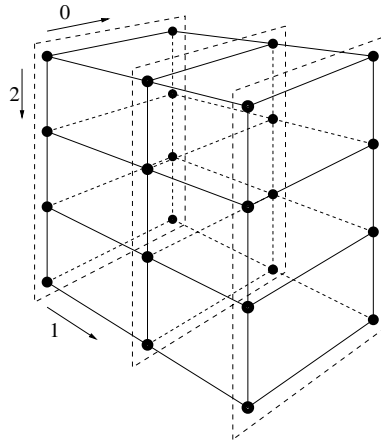


Abb. 5.12. Zerlegung eines $(2 \times 3 \times 4)$ -Gitter in drei (2×4) -Gitter.

liefert in `ndims` die Anzahl der Dimensionen des virtuellen Gitters zurück, das für den Kommunikator `comm` definiert wurde. Die Anweisung

```
int MPI_Cart_get (MPI_Comm comm,
                 int maxdims,
                 int *dims,
                 int *periods,
                 int *coords)
```

liefert die kartesischen Koordinaten des aufrufenden Prozesses bezüglich des für den Kommunikator `comm` definierten Gitters mit `maxdims` Dimensionen, wobei `dims`, `periods` und `coords` Felder mit `maxdims` Einträgen sind. Die Felder `dims` und `periods` haben die gleiche Bedeutung wie für `MPI_Cart_create()`. Das Feld `coords` dient zur Rückgabe der Koordinaten.

5.1.6 Zeitmessung und Abbruch der Ausführung

Zur Messung der Ausführungszeit von Programmteilen stellt MPI die Funktion

```
double MPI_Wtime (void)
```

zur Verfügung, die als Rückgabewert die Zeit in Sekunden liefert, die seit einem gewissen Punkt in der Vergangenheit verstrichen ist. Eine typische Zeitmessung sieht wie folgt aus:

```
start = MPI_Wtime();
part_to_measure();
end = MPI_Wtime();
```


Die von `MPI_Wtime()` zurückgegebene Zeit ist keine Systemzeit, sondern die absolute Zeit zwischen Absetzen und Beenden der Aufgabe, d.h. wenn der ausführende Prozess während der Abarbeitung von `part_to_measure()` unterbrochen wird, so wird diese Zeit mitgezählt.

Die Auflösung der von `MPI_Wtime()` zurückgegebenen Zeit wird durch die Anweisung

```
double MPI_Wtick (void)
```

angegeben, d.h. es wird die Zeit (in Sekunden) zurückgegeben, die zwischen zwei Takten liegt.

Um die Ausführung der Prozesse eines Kommunikators abubrechen, stellt MPI die Funktion

```
int MPI_Abort (MPI_Comm comm, int error_code)
```

zur Verfügung. Die angegebene Fehlernummer `error_code` wird so zurückgeliefert, als ob das Hauptprogramm seine Ausführung mit `return error_code` beendet.

5.2 Einführung in PVM

PVM (*Parallel Virtual Machine*) wurde mit dem Ziel entwickelt, eine einheitliche Programmierungsumgebung für eine Ansammlung von evtl. heterogenen Rechnern zur Verfügung zu stellen, die diese als *eine* virtuelle parallele Maschine erscheinen lässt und die eine einfache verteilte Programmierung erlaubt. Als Programmiersprachen werden C, C++ und FORTRAN unterstützt. Die Rechner der virtuellen Maschine sind üblicherweise mit einem lokalen Netzwerk (LAN, *local area network*) untereinander verbunden, prinzipiell können aber auch beliebige Netzwerke verwendet werden. PVM wird auch für die meisten Parallelrechner mit verteiltem Speicher angeboten. Damit ist es möglich, sehr unterschiedliche Rechner (wie PCs, Workstations, Shared-Memory-Rechner und Distributed-Memory-Rechner) mit evtl. unterschiedlichen Betriebssystemen zu einer virtuellen Maschine zusammenzusetzen. Da für den Datenaustausch zwischen den Rechnern geeignete Kodierungen verwendet werden können, ist es auch zulässig, dass die Rechner einer virtuellen Maschine unterschiedliche Datenrepräsentationen verwenden.

5.2.1 Programmiermodell

Ein PVM-Programm besteht aus einer Menge von parallel ausführbaren Tasks, die Daten untereinander austauschen können und die sich mit Hilfe geeigneter Operationen untereinander synchronisieren können. Die Anzahl der Tasks, die bei der Ausführung eines Programms verwendet wird, muss nicht bereits beim Start des Programms festgelegt werden, sondern kann

in Abhängigkeit von der Größe der Eingabedaten oder der Anzahl der zur Verfügung stehenden Rechner zur Laufzeit des Programms bestimmt werden. Zur Realisierung dieses Konzeptes der dynamischen Taskverwaltung kann jede Task zu beliebigen Zeiten ihrer Ausführung dynamisch andere Tasks erzeugen oder vorher erzeugte Tasks abbrechen. Ebenso können während der Ausführung eines Programms zusätzliche Rechner in die virtuelle Maschine aufgenommen werden. Dementsprechend erlaubt PVM prinzipiell die Ausführung von mehreren Tasks auf demselben Rechner. Bei manchen Implementierungen für Rechner mit verteiltem Speicher wird aber diese Flexibilität durch die Festlegung eingeschränkt, dass auf jedem Prozessor des Rechners nur eine Task ausgeführt wird.

Tasks werden über *Task Identifier* (TID) angesprochen, die vom PVM-System als eindeutige Nummer an neu erzeugte Tasks vergeben werden. Der Programmierer kann Tasks zu *Gruppen* zusammenfassen und den Gruppen selbstgewählte, eindeutige Namen (Zeichenketten) zuordnen. Die Tasks einer Gruppe erhalten innerhalb der Gruppe eine eindeutige, mit 0 beginnende Nummerierung. Eine Task kann mehreren Gruppen angehören, d.h. die Gruppen können sich überlappen.

Die Organisation eines parallelen Programms in Tasks kann nach unterschiedlichen Methoden durchgeführt werden, wobei die Auswahl einer speziellen Organisationsform stark von der gegebenen Anwendung abhängt. Eine häufig verwendete Organisationsform ist das *Master-Slave-Modell*, d.h. es wird eine eigenständige Kontrolltask (*Master*) verwendet, die die Steuerung der anderen Tasks (*Slaves*) übernimmt, vgl. Abschnitt 3.4. Je nach Eingabegröße erzeugt die Kontrolltask mehrere Slave-Tasks, die die eigentliche Berechnung ausführen, und sammelt nach deren Beendigung die errechneten Teilergebnisse ein. Eine andere Möglichkeit ist eine SPMD (*single-program multiple-data*)-Organisation der Tasks, in der jede Task gleichberechtigt das gleiche Programm ausführt, d.h. die Kontrolle der Berechnung wird nicht-hierarchisch über die einzelnen Tasks verteilt. Für manche Anwendungen, die z.B. nach dem Teile-und-Beherrsche-Prinzip arbeiten, ist auch eine baumartige Organisation der Berechnung sinnvoll, in der die Tasks entsprechend der rekursiven Aufrufstruktur der Anwendung abgespalten werden.

Wir werden im Folgenden kurz die PVM-Operationen zur Kontrolle von Tasks, zur Synchronisation von Tasks und zum Austausch von Nachrichten beschreiben. Für eine genaue Beschreibung verweisen wir auf [55].

5.2.2 Prozesskontrolle

Beim Start eines PVM-Programms ist normalerweise genau eine Task aktiv. Neue Tasks können durch Aufruf von `pvm_spawn()` abgespalten werden:

```
int pvm_spawn(char *task,
              char **argv,
              int flag,
              char *where,
```

```
int ntask,
int *tids).
```

Als Argumente werden übergeben:

- der Name (**task**) der als ausführbares Programm vorliegenden Task,
- ein Feld (**argv**) mit den Argumenten des zu startenden Programms, wobei das Ende des Feldes durch NULL angezeigt wird,
- ein Flag (**flag**), das angibt, wie das Abspalten der Task durchgeführt werden soll,
- ein Rechnernamen (**where**), der für bestimmte Besetzungen des Flags angibt, auf welchem Rechner bzw. auf welcher Architektur die neue Task gestartet werden soll,
- die Anzahl (**ntasks**) der neu zu startenden Tasks,
- ein Integerfeld (**tids**) der Größe **ntasks**, das nach erfolgreichem Start der neuen Tasks deren TIDs enthält.

Die wichtigsten Möglichkeiten für die Besetzung des Flags sind:

Wert	symbolische Konstante	Bedeutung
0	PvmTaskDefault	PVM wählt den Rechner für die neuen Tasks
1	PvmTaskHost	where gibt den Zielrechner an
2	PvmTaskArch	where gibt den Namen einer Architektur an
4	PvmTaskDebug	die Tasks werden in einem Debugger gestartet
8	PvmTaskTrace	erzeugt Trace-Daten für die neuen Tasks
16	PvmMppFront	startet Tasks auf einem Frontend-Rechner
32	PvmHostCompl	Komplementmenge zu where wird verwendet

Beispiele für Architekturnamen sind SUN3, SUN4, SUN4SOL2, LINUX, CRAY, CRAY2, KSR1 und I860. Ein Aufruf von `pvm_spawn()` liefert die Anzahl der erfolgreich gestarteten Tasks zurück. Wenn der Rückgabewert kleiner Null ist, ist ein Systemfehler aufgetreten. In diesem Fall ist der genaue Wert des Rückgabewertes ein Fehlercode, der den Typ des aufgetretenen Fehlers angibt. Der Aufruf

```
n = pvm_spawn("prog", (char **) 0, PvmTaskHost, "alpha", 1, &tid[0]);
```

erzeugt z.B. eine neue Task auf dem Rechner mit Hostnamen **alpha**, wobei die Task das Programm **prog** abarbeitet, das keine Parameter braucht.

Eine Task kann ihre Ausführung durch Aufruf von `pvm_exit(void)` beenden. Eine Task kann die Ausführung einer anderen Task mit TID **tid** durch Aufruf von `pvm_kill(int tid)` abbrechen. PVM-Tasks werden über eindeutige Nummern (TID) identifiziert. Jede Task kann ihre eigene TID-Nummer durch Aufruf der Funktion `pvm_mytid()` in Erfahrung bringen. Die TID-Nummer der Task, die die aufrufende Task durch Aufruf von `pvm_spawn()` erzeugt hat, wird durch den Aufruf von `pvm_parent(void)` zurückgeliefert. Falls die aufrufende Task nicht mit `pvm_spawn()` erzeugt wurde, wird `PvmNoParent` zurückgeliefert.

Ein PVM-Programm arbeitet auf einer virtuellen Maschine, deren Konfiguration durch die PVM-Anweisungen `pvm_addhosts()` und `pvm_delhosts()` dynamisch geändert werden kann. Mit Hilfe der Anweisung

```
int pvm_addhosts(char **hosts,
                 int nhost,
                 int *infos)
```

können neue Host-Rechner zur virtuellen Maschine hinzugefügt werden. Dabei bezeichnet `nhost` die Anzahl der hinzuzufügenden Maschinen, `hosts` bezeichnet ein Feld der Länge `nhost`, dessen Einträge die Namen der hinzuzufügenden Maschinen enthalten. Nach dem Aufruf enthält das Feld `infos` eine als Integerzahl kodierte Statusinformation für jeden im Feld `hosts` angegebenen Rechner. Der Rückgabewert des Aufrufs gibt die Anzahl der erfolgreich hinzugefügten Rechner an. Mögliche im Feld `infos` zurückgelieferte Fehlermeldungen sind:

<code>PvmNoHost</code>	falscher Maschinenname
<code>PvmCantStart</code>	PVM-Dämon kann nicht gestartet werden
<code>PvmDupHost</code>	Maschine gehört bereits zur virtuellen Maschine.

Mit Hilfe der Anweisung

```
int pvm_delhosts(char **hosts, int nhost, int *infos)
```

können Rechner aus der virtuellen Maschine entfernt werden. Die Bedeutung der Parameter ist dabei identisch zur Bedeutung der Parameter von `pvm_addhosts()`. Der Rückgabewert des Aufrufs gibt die Anzahl der erfolgreich entfernten Rechner an.

Informationen über die aktuelle Konfiguration der virtuellen Maschine werden durch einen Aufruf von `pvm_config()` zurückgeliefert:

```
int pvm_config(int *nhost, int *narch, struct pvmhostinfo **hostp).
```

Nach diesem Aufruf enthält `nhost` die Anzahl der Rechner in der virtuellen Maschine, `narch` enthält die Anzahl der verschiedenen Rechnertypen, wobei zwei Rechner dann als unterschiedlich angesehen werden, wenn sie verschiedene Datenformate benutzen. Der Parameter `hostp` muß als Zeiger auf ein vom Aufrufer allokiertes Feld der Minimalgröße `nhost` mit Einträgen des vordefinierten Typs `pvmhostinfo` übergeben werden. Nach dem Aufruf enthält jeder Eintrag des Feldes Informationen über einen Rechner der virtuellen Maschine, die durch Untersuchung der Einträge der Datenstruktur `pvmhostinfo` abgefragt werden können, die wie folgt definiert ist:

```
struct pvmhostinfo{int hi_tid;
                   char *hi_name;
                   char *hi_arch;
                   int hi_speed; }.
```

Dabei bezeichnet `hi_tid` die TID des PVM-Dämons auf der entsprechenden Maschine, `hi_name` gibt den Namen der Maschine an, `hi_arch` bezeichnet den Namen der Architektur der Maschine, `hi_speed` gibt die CPU-Geschwindigkeit der Maschine an. Informationen über die Tasks, die zur Zeit

auf der virtuellen Maschine ausgeführt werden, werden durch einen Aufruf `pvm_tasks()` zurückgeliefert:

```
int pvm_tasks(int which,
              int *ntask,
              struct pvmtaskinfo **taskp).
```

Mit dem Parameter `which` kann zwischen verschiedenen Informationen ausgewählt werden: mit 0 werden alle Tasks der virtuellen Maschine ausgewählt, durch Angabe der TID eines PVM-Dämons werden nur die Tasks des Rechners, auf dem der Dämon läuft, ausgewählt. Nach dem Aufruf von `pvm_tasks()` enthält `ntask` die Anzahl der Tasks. Der Parameter `taskp` muss als Zeiger auf ein vom Aufrufer allokirtes Feld der Minimalgröße `ntask` mit Einträgen des vordefinierten Typs `pvmtaskinfo` übergeben werden. Nach dem Aufruf enthält jeder Eintrag des Feldes Informationen über eine Task, die durch Untersuchung der Einträge der Datenstruktur `pvmtaskinfo` abgefragt werden können, die wie folgt definiert ist:

```
struct pvmtaskinfo {int ti_tid;
                   int ti_ptid;
                   int ti_host;
                   int ti_flag;
                   char *ti_a_out;
                   int ti_pid; }.
```

Dabei bezeichnet `ti_tid` die TID der entsprechenden Task, `ti_ptid` bezeichnet die TID des Vaterprozesses, `ti_host` bezeichnet die TID des PVM-Dämons auf der zugehörigen Maschine, `ti_flag` zeigt an, ob die Task auf eine Nachricht wartet, `ti_a_out` gibt den Namen des ausgeführten Programms an, `ti_pid` bezeichnet die Process-ID der Task.

5.2.3 Austausch von Nachrichten

Der Austausch von Nachrichten wird in PVM durch Aufruf mehrerer PVM-Anweisungen für einen Datentransfer realisiert. Zum Versenden einer Nachricht ruft der Sender die folgenden Anweisungen in der angegebenen Reihenfolge auf:

1. Die Initialisierung des Datentransfers wird mit Hilfe der Anweisung

```
int pvm_initsend (int encoding)
```

realisiert, wobei `encoding` die für den Datentransfer verwendete Kodierung angibt. Dabei gibt `PvmDataDefault` an, dass ein sicheres Kodieren der Nachricht verwendet werden soll, für welches unterschiedliche Datenformate der beteiligten Maschinen keine Fehler verursachen können. Die Angabe von `PvmDataRaw` bewirkt, dass für den Datentransfer keine Kodierung vorgenommen werden soll, was nur dann anzuraten ist, wenn

der Programmierer sicher ist, dass die beteiligten Maschinen gleiche Datenformate verwenden. Die Angabe von `PvmDataInPlace` bewirkt, dass das PVM-System die zu verschickende Nachricht nicht in Systempuffern zwischenspeichert, sondern direkt aus der Datenstruktur des Programms an den Empfänger verschickt.

2. Das Einpacken der zu verschickenden Nachricht wird mit Hilfe von Packanweisungen realisiert, wobei für jeden elementaren Datentyp eine eigene Packanweisungen zur Verfügung gestellt wird. Für Integerwerte steht z.B. die Packanweisung

```
int pvm_pkint (int *val, int count, int stride)
```

zur Verfügung, wobei `val` den Datenbereich bezeichnet, der mit der Nachricht verschickt werden soll, `count` bezeichnet die Anzahl der zu verschickenden Elemente, `stride` bezeichnet die Schrittweite, mit der die Elemente aus dem angegebenen Datenbereich entnommen werden sollen. Somit können in eine Nachricht mehrere Datentypen gepackt werden, was einfach dadurch realisiert wird, dass nach der Initialisierung des Datentransfers mehrere Packanweisungen für evtl. verschiedene Datentypen aufgerufen werden.

3. Das eigentliche Verschicken der Nachricht wird mit der Anweisung

```
int pvm_send (int dest, int tag)
```

vorgenommen, wobei `dest` das Ziel des Datentransfers und `tag` die Markierung der Nachricht bezeichnet. Die Anweisung arbeitet asynchron.

Zum Empfang einer Nachricht ruft der empfangende Prozess die folgenden Anweisungen in der angegebenen Reihenfolge auf:

1. Der eigentliche Empfang der Nachricht wird mit der Anweisung

```
int pvm_rcv (int source, int tag)
```

vorgenommen, wobei `source` den Quellprozess der Nachricht und `tag` die Markierung der zu empfangenden Nachricht angibt. Die Besetzung `source = -1` bewirkt, dass eine Nachricht von einem beliebigen Sender empfangen wird und entspricht damit `MPI_ANY_SOURCE`, die Besetzung `tag = -1` bewirkt, dass eine Nachricht mit einer beliebigen Markierung empfangen wird und entspricht damit `MPI_ANY_TAG`. Die Anweisung `pvm_rcv()` ist blockierend.

2. Nach dem Empfang der Nachricht muss diese mit speziellen Anweisungen ausgepackt werden, wobei wieder für jeden elementaren Datentyp eine eigene Auspackanweisungen zur Verfügung steht. Für Integerwerte ist dies z.B. die Anweisung

```
int pvm_upkint (int *val, int count, int stride),
```

wobei die Parameter die gleiche Bedeutung haben wie für die korrespondierenden Packanweisungen mit dem Unterschied, dass sie sich auf die Datenstruktur beziehen, in der die empfangenen Werte abgelegt werden sollen. Die Auspackanweisungen müssen für eine korrekte Ablage in der gleichen Reihenfolge aufgerufen werden, wie die korrespondierenden Packanweisungen für das Einpacken der Nachricht aufgerufen wurden.

Für das Empfangen einer Nachricht steht neben der blockierenden Anweisung `pvm_recv()` auch die nicht-blockierende Anweisung `pvm_nrecv()` mit den gleichen Parametern zur Verfügung. Bei dieser Anweisung zeigt der Rückgabewert 0 an, dass die nachgefragte Nachricht noch nicht vorhanden ist, ein Rückgabewert > 0 bedeutet, dass die Nachricht erfolgreich empfangen wurde, wobei der zurückgegebene Wert eine interne Puffernummer angibt, ein Rückgabewert < 0 bedeutet, dass beim Empfang der Nachricht ein Fehler aufgetreten ist. Für den Test, ob eine bestimmte Nachricht bereits empfangen wurde, steht die Anweisung

```
int pvm_probe (int source, int tag)
```

zur Verfügung, wobei die Rückgabewerte die gleiche Bedeutung wie für `pvm_nrecv()` haben.

Eine Broadcast-Operation kann mit der Anweisung

```
int pvm_mcast (int *tids, int ntasks, int tag)
```

erreicht werden, wobei `ntasks` die Anzahl der Tasks bezeichnet, an die die Nachricht versendet werden soll. Der Parameter `tids` bezeichnet ein Feld der Länge `ntasks`, das die TIDs der Tasks enthält, an die die Nachricht geschickt wird. Vor der Ausführung von `pvm_mcast()` muss wie für eine Einzeltransferoperation der Nachrichtentransfer mit `pvm_initsend()` initialisiert und mit Packanweisungen eingepackt worden sein. Die Empfänger der Broadcast-Nachricht empfangen diese (anders als bei MPI) mit den gleichen Empfangsanweisungen wie sie für Einzeltransferoperationen verwendet werden, d.h. mit `pvm_recv()` oder `pvm_nrecv()`. Die Broadcast-Operation wird auf der Senderseite als nicht-blockierende Operation durchgeführt.

5.2.4 Verwaltung von Prozessgruppen

PVM erlaubt die Verwaltung von Prozessgruppen über eine eigene Bibliothek `libgpvm3.a`. Dabei kann jede Task zu jedem Zeitpunkt eine Gruppe verlassen oder sich einer Gruppe anschließen, ohne die anderen Mitglieder der Gruppe zu informieren. Im Unterschied zu MPI können Tasks auch Nachrichten per Broadcast an Gruppen verschicken, denen sie nicht angehören. Zur Aufnahme einer Task in eine Gruppe führt eine Task die Anweisung

```
int pvm_joingroup (char *group)
```

aus, wobei **group** den Namen der Gruppe angibt, der sich die ausführende Task anschließen will. Der Rückgabewert bezeichnet die Nummer der Task in der angegebenen Gruppe. Ein Rückgabewert kleiner als Null bedeutet, dass ein Fehler aufgetreten ist. Wenn eine Task eine Gruppe verlässt und sich später wieder anschließt, hat sie nicht notwendigerweise dieselbe Nummer in der Gruppe. Zum Verlassen einer Gruppe steht die Anweisung

```
int pvm_lvgroup (char *group)
```

zur Verfügung, wobei der Parameter und der Rückgabewert die gleiche Bedeutung wie für **pvm_joyinggroup()** haben. Als Fehler-Rückgabewerte können **PvmNoGroup** und **PvmNotInGroup** auftreten, wobei ersterer Wert anzeigt, dass die angegebene Gruppe nicht existiert, letzterer Wert gibt an, dass die ausführende Task nicht Mitglied der angegebenen Gruppe ist.

Nach der Definition einer Prozessgruppe können Synchronisations-, Broadcast- und Akkumulationsoperationen auf dieser durchgeführt werden. Eine Barrier-Synchronisation wird durch die Anweisung

```
int pvm_barrier (char *group, int count)
```

erreicht, die den ausführenden Prozess so lange blockiert, bis **count** Mitglieder der angegebenen Prozessgruppe **group** die Synchronisationsoperation aufgerufen haben. Falls **count = -1** übergeben wird, müssen alle Mitglieder von **group** die Synchronisationsoperation aufrufen, um die Blockierung des ausführenden Prozesses aufzuheben. Die Anzahl der Mitglieder einer Gruppe wird durch Aufruf der Funktion

```
int pvm_gsize(char *group)
```

zurückgeliefert. Für ein korrektes Funktionieren von **pvm_barrier()** müssen alle Gruppenmitglieder den gleichen Wert für **count** angeben und es dürfen nicht gleichzeitig dynamische Gruppenänderungen (**pvm_joyinggroup()** oder **pvm_lvgroup()**) stattfinden.

Eine Broadcastoperation kann mit Hilfe der Anweisung

```
int pvm_bcast (char *group, int tag)
```

realisiert werden, wobei die Broadcast-Nachricht nach ihrer Initialisierung und Verpackung durch den Sender durch diese Operation an alle Tasks verschickt wird, die beim Start der Broadcastoperation Mitglieder der angegebenen Gruppe **group** waren, wobei aber der Sender die Nachricht nicht erhält. Der Sender muss auch nicht der angegebenen Gruppe angehören. Die Operation ist nicht-blockierend, d.h. die Kontrolle wird an den Sender zurückgegeben, sobald die Nachricht sicher abgeschickt ist. Der Unterschied zwischen **pvm_mcast()** und **pvm_bcast()** besteht darin, dass **pvm_mcast()** nicht gruppenorientiert eingesetzt werden kann, sondern dass die Operation im Wesentlichen eine abkürzende Schreibweise für das aufeinanderfolgende

Senden der gleichen Nachricht an mehrere Empfänger darstellt. Das folgende Programmfragment realisiert z.B. eine Broadcastoperation auf einer Prozessgruppe `worker`, bei der die Task `root` ein Integerfeld mit 10 Elementen verschickt.

```
if (pvm_mytid() == root) {
    info = pvm_initsend(PvmDataRaw);
    info = pvm_pkint(array,10,1);
    info = pvm_bcast("worker",17);
}
else {
    info = pvm_recv(0,17);
    info = pvm_upkint(array,10,1);
}
```

Eine Akkumulationsoperation kann mit Hilfe der Anweisung

```
int pvm_reduce (void (*f)(),
               void *data,
               int count,
               int datatype,
               int tag,
               char *group,
               int root)
```

realisiert werden. Dabei bezeichnet `f` die anzuwendende Reduktionsoperation, `data` gibt den von jeder beteiligten Task in der Gruppe `group` bereitgestellten Datenbereich an, wobei jeder Datenbereich `count` Elemente vom Typ `datatype` enthält. Die Daten werden bei der Task mit Nummer `root` in der angegebenen Gruppe `group` aufgesammelt. Für die Task `root` gibt `data` den Datenbereich an, in dem nach Abschluss der Operation das Ergebnis steht. Der Eintrag `tag` bezeichnet die Markierung der Nachrichten, die für die Akkumulation verwendet werden. Als vordefinierte Datentypen stehen zur Verfügung:

PVM Datentyp	C Datentyp
PVM_SHORT	short int
PVM_INT	int
PVM_LONG	long int
PVM_FLOAT	float
PVM_DOUBLE	double
PVM_CPLX	komplexe Zahl
PVM_DCPLX	komplexe Zahl doppelter Genauigkeit
PVM_BYTE	

Als vordefinierte Reduktionsoperationen stehen zur Verfügung: `PvmMin` für Minimumoperation, `PvmMax` für Maximumoperation, `PvmSum` für Summenoperation und `PvmProduct` für Produktoperation. Als Reduktionsoperation kann auch eine benutzerdefinierte Funktion `f` verwendet werden, die nach folgendem Prototyp definiert ist:

```
void f (int *datatype,
        void *x,
        void *y,
        int *num,
        int *info).
```

Dabei ist `datatype` der Datentyp der zu reduzierenden Daten, `x` und `y` sind die Datenbereiche der Länge `num`, in denen die Daten stehen, die elementweise reduziert werden sollen, wobei das Ergebnis in `y` abgelegt werden muß. Der Eintrag `info` enthält evtl. Fehlercodes als Rückgabewert.

5.3 Einführung in MPI-2

MPI-2 wurde 1997 als Erweiterung von MPI vorgeschlagen, d.h. der MPI-2-Standard ist eine Obermenge des MPI-Standards, den wir in Abschnitt 5.1 beschrieben haben. Der ursprüngliche MPI-Standard wird im Folgenden als MPI-1 bezeichnet. Da MPI-2 alle Befehle von MPI-1 umfasst, ist jedes korrekte MPI-1-Programm auch ein korrektes MPI-2-Programm. Die wesentlichen von MPI-2 eingeführten Erweiterungen betreffen eine dynamische Prozessverwaltung, einseitige Kommunikationsoperationen und Operationen zur parallelen Ein-/Ausgabe. Außerdem wurden einige der globalen Kommunikationsoperationen erweitert. Wir werden im Folgenden auf die wichtigsten Aspekte der beiden ersten Punkte etwas näher eingehen. Für eine ausführliche Besprechung verweisen wir auf [66].

5.3.1 Prozesserzeugung und -verwaltung

MPI-1 basiert auf einer *statischen* Prozesserzeugung, d.h. die zur Abarbeitung eines parallelen Programms verwendeten Prozesse werden beim *Start* des Programms festgelegt und erzeugt. Während des Programmlaufes können keine Prozesse hinzugefügt oder gelöscht werden. MPI-2 erweitert dieses Prozessmodell in Anlehnung an PVM zu einem dynamischen Modell, das zu jedem Zeitpunkt der Ausführung eines Programms die Erzeugung neuer und das Löschen existierender Prozesse erlaubt. MPI-2 legt dabei nur die Schnittstelle als eine Ansammlung geeignet definierter Funktionen fest, lässt deren Realisierung aber bis auf einige Vorschläge offen. Somit können die Funktionen auf unterschiedlichen Plattformen mit unterschiedlichen Betriebssystemen realisiert werden.

Informations-Datenstruktur. Um eine Interaktion mit einem zugrundeliegenden Betriebssystem zu ermöglichen, haben viele MPI-2-Funktionen ein zusätzliches Argument vom Typ `MPI_Info`, das die Angabe von zusätzlichen, vom Betriebssystem abhängigen Spezifikationen erlaubt. Wenn dies genutzt wird, sind die entstehenden Programme aber nicht mehr uneingeschränkt portabel. Eine Datenstruktur vom Typ `MPI_Info` enthält im Wesentlichen Paare

(**key,value**), wobei in C beide Einträge mit `\0` terminierte Strings vom Typ `char *` sind. Auf mögliche Nutzungen gehen wir im Folgenden noch genauer ein. Die genaue Implementierung der Datenstrukturen vom Typ `MPI_Info` bleibt dem Programmierer verborgen, zu ihrer Manipulation stellt MPI-2 aber eine Anzahl von Funktionen zur Verfügung. Die wichtigsten stellen wir kurz vor. Der Aufruf der Funktion

```
int MPI_Info_create (MPI_Info *info)
```

erzeugt ein neues Objekt vom Typ `MPI_Info`. Der Aufruf der Funktion

```
int MPI_Info_set (MPI_Info info, char *key, char *value)
```

fügt ein neues Paar (**key,value**) zu **info** hinzu und überschreibt damit einen schon vorhandenen Eintrag, wenn bereits ein Eintrag mit dem gleichen Wert von **key** existiert. Der Aufruf der Funktion

```
int MPI_Info_get(MPI_Info info,
                char *key,
                int valuelen,
                char *value,
                int *flag)
```

entnimmt ein vorher abgelegtes Paar (**key,value**) aus **info**. Beim Aufruf gibt der Programmierer den Wert von **key** und die maximal erwünschte Länge **valuelen** von **value** an, als Resultat erhält er den zu **key** gehörenden Eintrag **value**, falls dieser existiert. Falls der nachgefragte Eintrag nicht existiert, wird **flag** auf `false` gesetzt, sonst auf `true`. Der Aufruf der Funktion

```
int MPI_Info_delete(MPI_Info info, char *key)
```

löscht ein Paar (**key,value**) aus **info**, wobei beim Aufruf nur **key** angegeben werden muss.

Prozesserzeugung. Ein neuer Prozess wird mit Hilfe der Funktion

```
int MPI_Comm_spawn (char *command,
                   char *argv[],
                   int maxprocs,
                   MPI_Info info,
                   int root,
                   MPI_Comm comm,
                   MPI_Comm *intercomm,
                   int errcodes[])
```

erzeugt. Dabei bezeichnet **command** den Namen des zu startenden Programms und **argv[]** gibt dessen Kommandozeilenargumente an. Dabei gibt **argv[0]** das erste Kommandozeilenargument und nicht, wie in C üblich, den Programmnamen an. Eine leere Liste von Kommandozeilenargumenten wird durch `MPI_ARGV_NULL` angezeigt. Der Parameter **maxprocs** gibt die Anzahl der zu startenden Prozesse an, **info** stellt die Informations-Datenstruktur dar, die vom Laufzeitsystem für die Erzeugung der Prozesse genutzt wird. Dieser Parameter kann z.B. dazu verwendet werden, den

Pfad des zu startenden Programms oder sogar das zu startende Programm selbst und dessen Kommandozeilenargumente anzugeben. Um ein portables Programm zu erhalten, sollte jedoch `MPI_INFO_NULL` verwendet werden. Der Parameter `root` ist die Nummer des Prozesses, der den neuen Prozess abspaltet. Nur dieser Wurzelprozess stellt die vorangegangenen Argumente zur Verfügung. Die Funktion ist aber als globale Kommunikationsanweisung realisiert, d.h. alle zu dem Kommunikator `comm` gehörenden Prozesse müssen die Funktion `MPI_Comm_spawn()` aufrufen. Der Inter-Kommunikator `intercomm` dient der Kommunikation zwischen dem Kommunikator `comm` des aufrufenden Prozesses und den neu abgespalteten Prozessen. `errcodes[]` ist ein Feld der Länge `maxprocs`, das einen Eintrag für jeden zu erzeugenden Prozess bereitstellt. Wenn ein Prozess erfolgreich erzeugt werden konnte, enthält sein zugehöriger Eintrag `MPI_SUCCESS`, ansonsten wird ein Fehlercode abgelegt. Ein Aufruf von `MPI_Comm_spawn()` startet `maxprocs` identische Kopien des angegebenen Programms und richtet den Inter-Kommunikator `intercomm` ein. Die neu abgespaltenen Prozesse sind in einer anderen Gruppe als der Wurzelprozess. Die abgespaltenen Prozesse haben einen eigenen `MPI_COMM_WORLD`-Kommunikator, der alle abgespaltenen Prozesse umfasst. Der Inter-Kommunikator, der beim Aufruf von `MPI_Comm_spawn()` erzeugt und den aufrufenden Prozessen zurückgeliefert wird, kann von den abgespaltenen Prozessen durch Aufruf der Funktion

```
int MPI_Comm_get_parent(MPI_Comm *parent)
```

erhalten werden, d.h. `parent` zeigt nach diesem Aufruf auf diesen Inter-Kommunikator.

Unterschiedliche MPI-Programme oder MPI-Programme mit unterschiedlichen Kommandozeilenargumenten können durch Aufruf der Funktion

```
int MPI_Comm_spawn_multiple (int count,
                             char *commands[],
                             char **argv[],
                             int maxprocs[],
                             MPI_Info infos[],
                             int root,
                             MPI_Comm comm,
                             MPI_Comm *intercomm,
                             int errcodes[])
```

gestartet werden. Hierbei gibt `count` die Anzahl der zu startenden unterschiedlichen Programme an. Jedes der darauffolgenden vier Argumente ist ein Feld mit `count` Einträgen, die den gleichen Typ und die gleiche Bedeutung wie die korrespondierenden Argumente von `MPI_Comm_spawn()` haben: `commands[]` beinhaltet die Namen der zu startenden Programme, `argv[]` gibt deren Argumente an, `maxprocs[]` beinhaltet für jedes Programm die Anzahl der zu startenden Kopien und `infos[]` enthält für jedes Programm evtl. zusätzliche Angaben. Die anderen Argumente haben die gleiche Bedeutung wie bei `MPI_Comm_spawn()`. Nach Rückkehr der Funktion enthält

`errcodes[]` für jeden zu startenden Prozess einen Eintrag, wobei die in `commands[]` vorgegebene Reihenfolge zwischen den unterschiedlichen Programmen beibehalten wird. Insgesamt enthält `errcodes[]`

$$\sum_{i=0}^{\text{count}-1} \text{maxprocs}[i]$$

Einträge. Es gibt einen Unterschied zwischen aufeinanderfolgenden Aufrufen von `MPI_Comm_spawn()` und einem einmaligen Aufruf von `MPI_Comm_spawn_multiple()`, die beide die gleichen Programme mit den gleichen Argumenten aufrufen. Ein Aufruf von `MPI_Comm_spawn_multiple()` erzeugt *einen* Kommunikator `MPI_COMM_WORLD` für alle abgespaltenen Prozesse, während der mehrfache Aufruf von `MPI_Comm_spawn()` separate Kommunikatoren `MPI_COMM_WORLD` für jede der von den einzelnen Aufrufen abgespaltenen Mengen von Kindprozessen erzeugt.

Das Attribut `MPI_UNIVERSE_SIZE`, das durch `MPI_Init()` initialisiert wird, spezifiziert die maximale Anzahl von Prozessen, die von der laufenden Anwendung gestartet werden können.

5.3.2 Einseitige Kommunikation

MPI-1 stellt globale Kommunikationsoperationen und Einzel-Transferoperationen zur Verfügung. Bei globalen Kommunikationsoperationen ruft jeder dem verwendeten Kommunikator zugehörige Prozess die entsprechende Kommunikationsanweisung auf. An Einzel-Transferoperationen sind jeweils zwei Prozesse beteiligt, die wir bisher als Sender und Empfänger bezeichnet haben. Um Daten aus dem Adressraum des Senders in den Adressraum des Empfängers zu transportieren, muss im einfachsten Fall der Sender eine `MPI_Send()`-Anweisung und der Empfänger eine `MPI_Recv()`-Anweisung ausführen, d.h. sowohl der Sender als auch der Empfänger sind an der Durchführung der Kommunikationsoperation aktiv beteiligt. Aus diesem Grund bezeichnet man diese Form der Kommunikation auch als *zweiseitige* Kommunikation. Da der Sender durch die Positionierung der `MPI_Send()`-Anweisung festlegt, zu welchem Zeitpunkt er die Daten verschickt, und da der Empfänger durch die Positionierung der `MPI_Recv()`-Anweisung und die angegebene Speicheradresse festlegt, wann er die Daten an welche Position in seinem lokalen Speicher übernimmt, findet ein koordinierter Austausch statt.

MPI-2 erlaubt zusätzlich zur zweiseitigen Kommunikation auch *einseitige* Kommunikation, d.h. ein Quellprozess kann auf den Adressraum eines Zielprozesses zugreifen, ohne dass der Zielprozess an der Zugriffsoperation aktiv beteiligt ist. Diese Form der Kommunikation wird auch als RMA (*remote memory access*) bezeichnet und erlaubt eine flexible und dynamische Verteilung der Programmdaten auf die Speicher der einzelnen Prozessoren und flexible Speicherzugriffe der beteiligten Prozesse. Der Programmierer ist aber dafür verantwortlich, dass die Speicherzugriffe koordiniert ablaufen, d.h. er

muss dafür sorgen, dass nicht zum gleichen Zeitpunkt verschiedene Prozesse konkurrierend die Manipulation des gleichen Adreßbereiches versuchen, weil dadurch zeitkritische Abläufe entstehen können. Diese Gefahrenquelle tritt bei zweiseitiger Kommunikation nicht auf und der zusätzliche Aufwand zur Vermeidung von zeitkritischen Abläufen kann als Preis für die erhöhte Flexibilität bei der Verwendung von einseitiger Kommunikation aufgefasst werden.

Fensterobjekte. Um einem Prozess *A* den Zugriff auf einen lokalen Speicherbereich von Prozess *B* mit Hilfe einseitiger Kommunikation zu ermöglichen, muss Prozess *B* diesen Speicherbereich zuerst für den externen Zugriff freigeben. Ein für den externen Zugriff freigegebener Speicherbereich wird auch als *Fenster* (*window*) bezeichnet. Die Freigabe für den externen Zugriff wird durch Ausführen der globalen Kommunikationsanweisung

```
int MPI_Win_create (void *base,
                    MPI_Aint size,
                    int displ_unit,
                    MPI_Info info,
                    MPI_Comm comm,
                    MPI_Win *win)
```

erreicht, die jeder Prozess des Kommunikators *comm* ausführen muss. Dabei ist *base* die Startadresse und *size* die Größe in Bytes des von dem ausführenden Prozess erzeugten Fensters. Jeder beteiligte Prozess kann eine andere Größe angeben. Für die Angabe der Größe wird statt *int* der MPI-Typ *MPI_Aint* verwendet, der zur Darstellung von Speicherbereichsgrößen verwendet wird und die Angabe von Speicherbereichen ermöglicht, die größer als 2^{32} sind. Der Parameter *displ_unit* gibt den Abstand (engl. *displacement*) zwischen benachbarten Einträgen des Fensters an, der bei den einseitigen Speicherzugriffen zugrundegelegt wird. Typische Werte für *displ_unit* sind 1, wenn mit Bytes gearbeitet wird, oder *sizeof (type)*, wenn das Fenster Einträge vom Typ *type* enthält. Die *info*-Datenstruktur kann dazu verwendet werden, zusätzliche Hinweise an das Laufzeitsystem anzugeben. Im Normalfall wird *info* = *MPI_INFO_NULL* angegeben. Der Parameter *comm* gibt den Kommunikator der Prozesse an, die an der *MPI_Win_create()*-Operation teilnehmen. Der Aufruf von *MPI_Win_create()* liefert dem aufrufenden Prozess ein Fenster-Objekt vom Typ *MPI_Win* zurück, das für den Zugriff auf die freigegebenen Speicherbereiche der beteiligten Prozesse verwendet werden kann.

Ein für externe Zugriffe freigegebener Speicherbereich kann wieder geschlossen werden, indem alle beteiligten Prozesse die Funktion

```
int MPI_Win_free (MPI_Win *win)
```

aufrufen und damit das zugehörige Fenster-Objekt freigeben. Voraussetzung für eine erfolgreiche Durchführung dieses Aufrufes ist, dass jeder der beteiligten Prozesse alle Operationen auf dem Fenster *win* beendet hat.

RMA-Operationen. Zum eigentlichen Datentransfer werden von MPI-2 die folgenden drei *nichtblockierende* RMA-Operationen zur Verfügung gestellt: `MPI_Put()` transportiert Daten aus dem Speicher des aufrufenden Prozesses in das Fenster eines anderen Prozesses, `MPI_Get` transportiert Daten aus dem Fenster eines Zielprozesses in den Speicher des aufrufenden Prozesses und `MPI_Accumulate` erlaubt die Akkumulation von Daten im Fenster eines Zielprozesses. Diese Operationen sind nichtblockierend, d.h. der aufrufende Prozess erhält die Kontrolle zurück, ohne dass sichergestellt ist, dass die Operation vollständig abgeschlossen ist. Um die Beendigung einer Operation zu überprüfen, müssen *Synchronisationsoperationen* verwendet werden. Damit steht ein ähnlicher Mechanismus zur Verfügung wie bei nichtblockierenden zweiseitigen Kommunikationsoperationen, vgl. Abschnitt 5.1.1. Der lokale Puffer einer Kommunikationsoperation sollte erst nach dem Aufruf einer nachfolgenden Synchronisationsoperation weiterverwendet werden, damit sichergestellt ist, dass die Kommunikationsoperation vollständig abgeschlossen ist.

Die Ablage eines Datenblocks im Speicher eines anderen Prozesses erfolgt durch Aufruf der Anweisung

```
int MPI_Put (void *origin_addr,
             int origin_count,
             MPI_Datatype origin_type,
             int target_rank,
             MPI_Aint target_displ,
             int target_count,
             MPI_Datatype target_type,
             MPI_Win win)
```

Hierbei ist `origin_addr` die Startadresse des vom aufrufenden Prozess bereitgestellten Datenblocks, `origin_count` ist die Anzahl der Elemente in diesem Datenblock, `origin_type` ist der Typ der Elemente. Der Eintrag `target_rank` gibt die Nummer des Zielprozesses an. Dieser Zielprozess muss in einer vorangegangenen `MPI_Win_create()`-Operation das Fensterobjekt `win` erzeugt haben, und zwar zusammen mit den anderen Prozessen einer Gruppe, der auch der Prozess angehört, der die betrachtete `MPI_Put()`-Anweisung ausführt. Die restlichen Parameter des Aufrufes geben die Position und Größe des Zielpuffers an, der in dem vom Zielprozess zur Verfügung gestellten Fenster liegen muss: `target_displ` gibt den Abstand zwischen dem Anfang des Fensters und dem Anfang des Zielpuffers beim Zielprozess an, `target_count` gibt die Anzahl der Elemente im Zielpuffer an, `target_type` deren Typ. Die Ablage des Datenblockes im Speicher des Zielprozesses erfolgt an der Position `target_addr := window_base + target_displ * displ_unit`, wobei `window_base` die Anfangsadresse des Fensters des Zielprozesses und `displ_unit` der Abstand zwischen benachbarten Einträgen des Fensters angibt, wie sie bei der Initialisierung des Fensters mit Hilfe einer `MPI_Win_create()`-Anweisung durch den Zielprozess angegeben wurden. Die Ausführung der angegebenen `MPI_Put()`-Anweisung durch einen Prozess

`source` hat den gleichen Effekt wie eine zweiseitige Kommunikation, bei der der Quellprozess `source` die Sendeoperation

```
int MPI_Isend (origin_addr, origin_count, origin_type,
               target_rank, tag, comm)
```

und der Zielprozess die Empfangsoperation

```
int MPI_Recv (target_addr, target_count, target_type,
              source, tag, comm, &status)
```

ausführt, wobei der Kommunikator `comm` verwendet wird, der zu der Gruppe von `win` gehört.

Für eine korrekte Durchführung der Operation muss sichergestellt sein, dass der angegebene Zielpuffer in das Fenster des Zielprozesses und dass der vom ausführenden Prozess bereitgestellte Datenblock in den Zielpuffer passt. Die Sendepuffer für verschiedene aufeinanderfolgende `MPI_Put()`-Anweisungen dürfen – anders als bei `MPI_Isend()`-Anweisungen – auch dann überlappen, wenn zwischen diesen Anweisungen keine Synchronisationsanweisungen stattfinden, die die Beendigung der `MPI_Put()`-Anweisungen sicherstellen. Ziel- und Quellprozess der Anweisung dürfen **identisch** sein.

Zum Laden eines Datenblocks aus dem Speicher eines anderen Prozesses stellt MPI-2 die Anweisung

```
int MPI_Get (void *origin_addr,
              int origin_count,
              MPI_Datatype origin_type,
              int target_rank,
              MPI_Aint target_displ,
              int target_count,
              MPI_Datatype target_type,
              MPI_Win win)
```

zur Verfügung. Dabei ist `origin_addr` die Startadresse des Empfangspuffers im lokalen Speicher des ausführenden Prozesses, `origin_count` gibt die Anzahl der Elemente vom Typ `origin_type` an, die im Empfangspuffer abgelegt werden sollen. Wie bei `MPI_Put()` geben `target_rank` und `win` die Nummer des Zielprozesses, der die Daten bereitstellt, und das verwendete Fensterobjekt an. Analog zu `MPI_Put()` geben die restlichen Einträge die Position und Größe des aus dem Fenster des Zielprozesses zu transportierenden Datenblocks an. Dessen Anfangsadresse ergibt sich zu `target_addr := window_base + target_displ * displ_unit`.

Zur Akkumulation von Daten im Speicher eines anderen Prozesses stellt MPI-2 die Anweisung

```
int MPI_Accumulate (void *origin_addr,
                    int origin_count,
                    MPI_Datatype origin_type,
                    int target_rank,
                    MPI_Aint target_displ,
```



```

int target_count,
MPI_Datatype target_type,
MPI_Op op,
MPI_Win win)

```

zur Verfügung. Die Parameter haben dabei die gleiche Bedeutung wie bei einer `MPI_Put()`-Anweisung. Der zusätzliche Parameter `op` gibt die anzuwendende Reduktionsoperation an, wobei die in Abschnitt 5.1.2 bei der Beschreibung von `MPI_Reduce()` angegebenen, vordefinierten Reduktionsoperationen wie `MPI_MAX` oder `MPI_SUM` verwendet werden können. Der Effekt dieser Anweisung besteht darin, dass die angegebene Reduktionsoperation auf die korrespondierenden Einträge des vom ausführenden Prozess bereitgestellten Datenblockes und des angegebenen Zielpuffers angewendet wird und dass das Resultat in den Zielpuffer zurückgeschrieben wird. Damit kann diese Anweisung zur Akkumulation von Werten in einem von einem anderen Prozess zur Verfügung gestellten Zielpuffer verwendet werden. Als Reduktionsoperation dürfen – anders als bei `MPI_Reduce()` – keine benutzerdefinierten Reduktionsoperationen verwendet werden. Es gibt aber eine zusätzliche Reduktionsoperation `MPI_REPLACE`, die eine Ersetzung der Einträge des Zielpuffers bewirkt, ohne dass der frühere Wert des Eintrages berücksichtigt wird. Damit kann `MPI_Put()` als Spezialfall von `MPI_Accumulate` aufgefasst werden, der `MPI_REPLACE` als Reduktionsoperation verwendet.

Für die Ausführung der von verschiedenen Prozessen aufgerufenen einseitigen Kommunikationsanweisungen gelten gewisse Beschränkungen, die zeitkritische Abläufe vermeiden und eine effiziente Implementierung der Kommunikationsoperationen ermöglichen sollen. Jede Speicherzelle eines Fensters darf zu jedem Zeitpunkt der Programmausführung das Ziel maximal einer einseitigen Kommunikationsoperation sein, d.h. eine konkurrierende Manipulation der gleichen Speicherzelle durch unterschiedliche Prozesse ist nicht erlaubt. Eine Ausnahme von dieser Regel stellen Akkumulationsoperationen dar, d.h. mehrere `MPI_Accumulate`-Operationen können zum gleichen Zeitpunkt für die gleiche Speicherzelle aktiv sein. Das Resultat der Berechnung ergibt sich in diesem Fall durch eine beliebige Anordnung der ausgeführten Operationen. Da die vordefinierten Reduktionsoperationen kommutativ sein müssen, ergibt sich dabei stets das gleiche Resultat. Ein Fenster eines Prozesses *P* kann nicht zur gleichen Zeit von einer `MPI_Put()`- oder `MPI_Accumulate()`-Operation eines anderen Prozesses und einer lokalen Schreiboperation von *P* manipuliert werden, auch wenn unterschiedliche Speicherzellen des Fensters angesprochen werden.

Zur Koordination der auf den Fenstern einer Gruppe von Prozessen arbeitenden einseitigen Kommunikationsanweisungen stellt MPI-2 drei verschiedene Synchronisationsmechanismen zur Verfügung, die wir im Folgenden nacheinander beschreiben.

Globale Synchronisation. Der einfachste Synchronisationsmechanismus besteht in einer globalen Synchronisation, die auf der zu einem Fenster

gehörenden Prozessgruppe ausgeführt wird. Diese Form der Synchronisation ist brauchbar für regelmäßige Anwendungen, in denen sich globale Berechnungsphasen mit globalen Kommunikationsphasen abwechseln. Zur Realisierung der globalen Synchronisation stellt MPI-2 die Anweisung

```
int MPI_Win_fence (int assert, MPI_Win win)
```

zur Verfügung. Dabei handelt es sich um eine globale Kommunikationsanweisung, die von allen Prozessen, die zu der Gruppe des angegebenen Fensters `win` gehören, aufgerufen werden muss. Der Aufruf bewirkt, dass ein aufrufender Prozess erst dann die nächste Anweisung ausführt, wenn alle von diesem Prozess ausgehenden und auf dem Fenster `win` arbeitenden einseitigen Kommunikationsanweisungen abgeschlossen sind. Eine nach einer `MPI_Win_fence()`-Anweisung abgesetzte einseitige Kommunikationsanweisung kann erst beendet werden, wenn der Zielprozess seine zugehörige `MPI_Win_fence()`-Anweisung abgeschlossen hat. Die vorgesehene Anwendung für die `MPI_Win_fence()`-Anweisung besteht darin, die Bereiche eines Programms, in denen die einseitigen Kommunikationsanweisungen stattfinden, mit Aufrufen dieser Anweisung zu umschließen, um so Kommunikationsphasen einzurichten, zwischen denen Berechnungsphasen stattfinden können. Der Parameter `assert` kann dazu verwendet werden, Angaben über den Kontext des Aufrufes von `MPI_Win_fence()` zu machen, die vom Laufzeitsystem für Optimierungen verwendet werden können. Der Normalfall besteht darin, keine zusätzlichen Informationen anzugeben, d.h. `assert` auf 0 zu setzen, was wir im Folgenden annehmen.

Beispiel: Wir betrachten die iterative Berechnung einer verteilten Datenstruktur `A`. In jedem Iterationsschritt aktualisiert jeder beteiligte Prozess seinen lokalen Teil der Datenstruktur mit Hilfe einer Funktion `update()` und stellt Teile dieser Datenstruktur seinen Nachbarprozessen zur Verfügung, indem er diese Teile mit `MPI_Put()`-Anweisungen in die Fenster der Nachbarprozesse transportiert. Vor dem Transport werden die Teile in zusammenhängende Puffer kopiert. Wir nehmen an, dass dies durch die Funktion `update_buffer()` durchgeführt wird. Diese Kommunikationsanweisungen werden von `MPI_Win_fence()`-Anweisungen umgeben, um die Kommunikationsphasen der verschiedenen Iterationen voneinander zu trennen. Daraus resultiert das folgende Programmfragment:

```
while (!converged (A)) {
    update(A);
    update_buffer(A, from_buf);
    MPI_Win_fence (0, win);
    for (i=0; i<num_neighbors; i++)
        MPI_Put (&from_buf[i], size[i], MPI_INT, neighbor[i], to_disp[i],
                size[i], MPI_INT, win);
    MPI_Win_fence (0, win);
}
```

Der Iterationsprozess wird von der Funktion `converged()` kontrolliert. □

Teilsynchronisation auf einer Untergruppe. MPI-2 stellt auch Operationen zur Verfügung, mit denen eine Synchronisation auf den zugreifenden Prozess und den Prozess, auf dessen Fenster zugegriffen wird, beschränkt werden kann. Die Synchronisation findet dadurch statt, dass der zugreifende Prozess den Beginn und das Ende einer Zugriffsphase auf die Fenster von anderen Prozessen einer Fenstergruppe, der der zugreifende Prozess ebenfalls angehört, mit `MPI_Win_start()` bzw. `MPI_Win_complete()` anzeigt. Die Prozesse, auf deren Fenster zugegriffen wird, stellen dieses für eine Zugriffsphase zur Verfügung, indem sie deren Beginn mit `MPI_Win_post()` und deren Ende mit `MPI_Win_wait()` anzeigen. Dabei findet zwischen `MPI_Win_start()` und `MPI_Win_post()` eine Synchronisation statt, die sicherstellt, dass die RMA-Zugriffsoperationen, die der zugreifende Prozess nach `MPI_Win_start()` absetzt, erst ausgeführt werden, nachdem der Zielprozess seine `MPI_Win_post()`-Anweisung abgeschlossen hat. Analog stellt die Synchronisation zwischen `MPI_Win_complete()` und `MPI_Win_wait()` sicher, dass die RMA-Zugriffsoperationen beim zugreifenden Prozess beendet werden, bevor der Zielprozess die `MPI_Win_wait()`-Operation beendet.

Bei Verwendung des beschriebenen Synchronisationsmechanismus zeigt ein Prozess, der auf die Fenster von Zielprozessen zugreifen will, den Beginn seiner Zugriffsphase durch den Aufruf der Anweisung

```
int MPI_Win_start (MPI_Group group,
                  int assert,
                  MPI_Win win)
```

an. Dabei gibt `group` die Gruppe der Zielprozesse an, von denen jeder eine zugehörige `MPI_Win_post()`-Anweisung ausführen muss. Der Parameter `win` bezeichnet das Fensterobjekt, auf das zugegriffen werden soll. MPI-2 erlaubt ein blockierendes und ein nichtblockierendes Verhalten von `MPI_Win_start()`.

- Blockierendes Verhalten: `MPI_Win_start()` blockiert, bis alle Zielprozesse ihre zugehörige `MPI_Win_post()`-Anweisung beendet haben.
- Nichtblockierendes Verhalten: Auch wenn der zugreifende Prozess die Anweisung `MPI_Win_start()` ausführt, bevor alle Zielprozesse ihre zugehörige `MPI_Win_post()`-Anweisung beendet haben, muss der zugreifende Prozess nicht blockieren. Er erhält die Kontrolle zurück und kann nachfolgende Zugriffsoperationen wie `MPI_Put()` oder `MPI_Get()` absetzen, wobei diese vom System zwischengespeichert werden, bis die entsprechenden Zielprozesse ihre `MPI_Win_post()`-Anweisung beendet haben.

Das genaue Verhalten hängt von der Implementierung ab. Das Ende einer Zugriffsphase wird vom zugreifenden Prozess durch Aufruf der Anweisung

```
int MPI_Win_complete (MPI_Win win)
```

angezeigt, wobei `win` das Fensterobjekt bezeichnet, auf das in der Zugriffsphase zugegriffen wurde. Zwischen dem Aufruf von `MPI_Win_start()` und

`MPI_Win_complete()` dürfen nur RMA-Zugriffe auf die Fenster `win` von Prozessen stattfinden, die zu der angegebenen Gruppe `group` gehören. Der ausführende Prozess wird blockiert, bis alle zwischen dem vorangehenden `MPI_Win_start()` und `MPI_Win_complete()` liegenden RMA-Zugriffe auf das Fensterobjekt `win` beim *zugreifenden* Prozess beendet sind. Dies bedeutet, dass der zugreifende Prozess z.B. bei einer in der Zugriffsphase liegenden `MPI_Put()`-Anweisung die die Zugriffsphase beendende `MPI_Win_complete()`-Anweisung beendet, sobald der lokale Puffer wiederverwendet werden kann. Dies kann dadurch entstehen, daß die `MPI_Put()`-Anweisung auch beim Zielprozess beendet wurde. Je nach Implementierung des Laufzeitsystems kann aber auch nur eine Zwischenspeicherung in einem Systempuffer stattgefunden haben. In diesem Fall bedeutet die Beendigung der `MPI_Win_complete()`-Anweisung beim zugreifenden Prozess nicht unbedingt, dass die RMA-Zugriffsoperationen auch beim Zielprozess beendet wurden.

Die Freigabe eines lokalen Fensters für RMA-Zugriffe wird dadurch erreicht, dass der Eigentümer des Fensters die Anweisung

```
int MPI_Win_post (MPI_Group group,
                  int assert,
                  MPI_Win win)
```

aufruft. Dabei gibt `group` die Gruppe der Prozesse an, die auf das angegebene Fenster `win` zugreifen dürfen. Jeder dieser Prozesse muss einen zugehörigen Aufruf von `MPI_Win_start()` durchführen. Der Aufruf von `MPI_Win_post()` ist nicht blockierend.

Das Ende einer Zugriffsphase wird von dem Eigentümer eines Fensters `win` durch Ausführung der Anweisung

```
int MPI_Win_wait (MPI_Win win)
```

angegeben. Der Aufruf dieser Anweisung blockiert, bis alle Prozesse aus der im zugehörigen `MPI_Win_post()`-Aufruf angegebenen Gruppe `group` die zugehörige `MPI_Win_complete()`-Anweisung aufgerufen haben. Damit wird sichergestellt, dass diese Prozesse ihre RMA-Zugriffe auf das angegebene Fenster beendet haben. Nach Beendigung des `MPI_Win_post()`-Aufrufes ist sichergestellt, dass alle diese RMA-Zugriffe auf das Fenster des aufrufenden Prozesses beendet sind, d.h. nach Beendigung von `MPI_Win_wait()` können die Einträge des Fensters weiterverwendet werden. Während einer durch `MPI_Win_post()` und `MPI_Win_wait()` angezeigten Zugriffsphase sollte der zugehörige Prozess keine lokalen Operationen auf den Einträgen des zugegriffenen Fensters ausführen, da diese von Zugriffen anderer Prozesse betroffen sein könnten.

Ob die RMA-Zugriffe auf ein Fenster `win` bereits abgeschlossen sind, kann mit Hilfe der Anweisung

```
int MPI_Win_test (MPI_Win win, int *flag)
```

festgestellt werden. Diese Anweisung kann als nichtblockierende Variante von `MPI_Win_wait()` aufgefasst werden. Wenn alle RMA-Zugriffe auf `win` beendet sind, wird `flag = 1` zurückgegeben. In diesem Fall hat `MPI_Win_test()` den gleichen Effekt wie `MPI_Win_wait()` und darf nicht erneut aufgerufen werden. Wenn noch nicht alle RMA-Zugriffe auf `win` abgeschlossen sind, wird `flag = 0` zurückgegeben. In diesem Fall hat der Aufruf keinen weiteren Effekt.

Der beschriebene Synchronisationsmechanismus kann dazu verwendet werden, beliebige Kommunikationsmuster auf einer Gruppe von Prozessen zu realisieren. Das Kommunikationsmuster kann durch einen gerichteten Graphen $G = (V, E)$ beschrieben werden, wobei V die Menge der beteiligten Prozesse bezeichnet und eine Kante $(i, j) \in E$ existiert, falls Prozess i auf das Fenster von Prozess j mit einer RMA-Operation zugreift. Wenn die RMA-Zugriffe auf einem Fenster `win` stattfinden, kann die zugehörige Synchronisation dadurch erreicht werden, dass jeder beteiligte Prozess `MPI_Win_start(target_group, 0, win)` gefolgt von `MPI_Win_post(source_group, 0, win)` aufruft, wobei `source_group = {i; (i, j) ∈ E}` die Menge der zugreifenden Prozesse und `target_group = {j; (i, j) ∈ E}` die Menge der Zielprozesse bezeichnet.

Beispiel: Diese Form der Synchronisation wird auch in dem folgenden Beispiel verwendet, das das vorige Beispiel einer iterativen Berechnung einer verteilten Datenstruktur variiert.

```
while (!converged (A)) {
    update (A);
    update_buffer(A, from_buf);
    MPI_Win_start (target_group, 0, win);
    MPI_Win_post (source_group, 0, win);
    for (i=0; i<num_neighbors; i++)
        MPI_Put (&from_buf[i], size[i], MPI_INT, neighbor[i], to_disp[i],
                size[i], MPI_INT, win);
    MPI_Win_complete (win);
    MPI_Win_wait (win);
}
```

Dabei wird angenommen, dass `source_group` und `target_group` entsprechend dem verwendeten Nachbarschaftskommunikationsmuster von allen beteiligten Prozessen wie gerade beschrieben definiert wurden. Eine Alternative besteht darin, dass jeder Prozess in der Variable `source_group` die Nachbarprozesse, die auf sein lokales Fenster zugreifen können, und in `target_group` die Nachbarprozesse, auf deren lokales Fenster er zugreifen will, angibt. Da jeder Prozess in diesem Fall andere Nachbarn hat, entsteht eine schwächere Form der Synchronisation als bei der gruppenglobalen Definition von `source_group` und `target_group`. $\square$

Sperr-Synchronisation. Zur Simulation eines Shared-Memory-Modells wird von MPI-2 ein Synchronisationsmechanismus zur Verfügung gestellt, bei dem nur der zugreifende Prozess aktiv am Zugriff beteiligt ist. Bei dieser Form der

Synchronisation ist es möglich, dass zwei Prozesse über das Fenster eines dritten Prozesses per RMA-Zugriff miteinander kommunizieren, ohne dass der dritte Prozess aktiv beteiligt ist, indem er z.B. das Fenster freigibt. Um Zugriffskonflikte zu vermeiden, sperrt der zugreifende Prozess während des Zugriffs das zugehörige Fenster, indem er vor dem Zugriff eine Sperre (engl. *lock*) setzt und sie nach dem Zugriff wieder freigibt. Dies wird auch als Sperrmechanismus bezeichnet und findet in Programmiermodellen für einen gemeinsamen Adressraum häufig Anwendung, vgl. Kapitel 6. Zur Realisierung der Synchronisation umgibt der zugreifende Prozess die RMA-Zugriffe auf das Fenster eines anderen Prozesses mit den Anweisungen `MPI_Win_lock()` und `MPI_Win_unlock()`. Die Anweisung

```
int MPI_Win_lock (int lock_type,
                  int rank,
                  int assert,
                  MPI_Win win)
```

startet eine Zugriffsphase auf das Fenster `win` des Zielprozesses mit der Nummer `rank`. Dabei stellt MPI-2 zwei verschiedene Sperrmechanismen zur Verfügung, die durch den Parameter `lock_type` unterschieden werden. Bei einer *exklusiven* Sperre, die durch `lock_type = MPI_LOCK_EXCLUSIVE` spezifiziert wird, wird sichergestellt, dass die nachfolgenden RMA-Zugriffe auf das angegebene Fenster gegen Zugriffe von anderen Prozessen geschützt sind, d.h. es wird sichergestellt, dass der ausführende Prozess während der RMA-Zugriffe das ausschließliche Zugriffsrecht auf das Fenster hat. Diese Form der Sperre sollte verwendet werden, wenn der ausführende Prozess Einträge des Fensters mit `MPI_Put()` verändern will, die auch von anderen Prozessen angesprochen werden könnten. Bei einer *gemeinsamen* Sperre, die durch `lock_type = MPI_LOCK_SHARED` spezifiziert wird, wird sichergestellt, dass die nachfolgenden RMA-Zugriffe auf das angegebene Fenster gegen exklusive Zugriffe von anderen Prozessen geschützt sind, d.h. es wird sichergestellt, dass während der RMA-Zugriffe des ausführenden Prozesses keine anderen Prozesse Einträge des Fensters durch RMA-Zugriffe verändern, die mit einer exklusiven Sperre geschützt sind. Es ist aber erlaubt, dass andere Prozesse RMA-Zugriffe auf das Fenster ausführen, die ebenfalls mit gemeinsamen Sperren geschützt sind. Gemeinsame Sperren sollten verwendet werden, wenn der ausführende Prozess auf die Fenstereinträge nur mit `MPI_Get()` oder `MPI_Accumulate()` zugreift. Wenn ein Prozess Einträge eines lokalen Fensters mit lokalen Operationen verändern oder lesen will, muss er diese Operationen ebenfalls mit einer Sperre schützen, wenn diese Einträge auch von anderen Prozessen zugegriffen werden können. Die Anweisung

```
int MPI_Win_unlock (int rank,
                    MPI_Win win)
```

beendet eine mit `MPI_Win_lock()` begonnene Zugriffsphase auf das Fenster `win` des Zielprozesses mit der Nummer `rank`. Der Aufruf dieser Anweisung

blockiert, bis alle vom ausführenden Prozess auf dem angegebenen Fenster durchgeführten RMA-Zugriffe beim ausführenden Prozess und beim Zielprozess vollständig abgeschlossen sind. Damit ist auch sichergestellt, dass alle Änderungen von Fenstereinträgen nicht nur beim ausführenden Prozess abgesetzt, sondern auch beim Zielprozess durchgeführt worden sind.

Beispiel: Das folgende Beispiel zeigt eine Realisierung des vorigen Beispiels einer iterativen Berechnung einer verteilten Datenstruktur mit Hilfe der beschriebenen Sperr-Synchronisation. Dabei wird jeder der Zugriffe auf das Fenster eines anderen Prozesses mit einer exklusiven Sperre geschützt.

```
while (!converged (A)) {
    update (A);
    update_buffer(A, from_buf);
    MPI_Win_start (target_group, 0, win);
    for (i=0; i<num_neighbors; i++) {
        MPI_Win_lock (MPI_LOCK_EXCLUSIVE, neighbor[i], 0, win);
        MPI_Put (&from_buf[i], size[i], MPI_INT, neighbor[i], to_disp[i],
                 size[i], MPI_INT, win);
        MPI_Win_unlock(neighbor[i], win);
    }
}
```

□

6. Thread-Programmierung

Die parallele Programmierung mit gemeinsamen Variablen (engl. *shared variables*) stellt neben der Message-Passing-Programmierung, siehe Kapitel 5, und der datenparallelen Programmierung eine der am häufigsten benutzten parallelen Programmierparadigmen dar. Die Programmierung mit gemeinsamen Variablen beruht auf einem *Speichermodell*, in dem unabhängige Programmstücke während ihrer Abarbeitung auf einen gemeinsamen Adressraum zugreifen und die dort abgelegten Variablen lesen oder manipulieren können. Im Gegensatz zu dem Modell eines verteilten Adressraumes, das für die Message-Passing-Programmierung verwendet wird, hat also jede an der Abarbeitung eines parallelen Programms beteiligte Ausführungseinheit *direkten* Zugriff auf jede Variable des globalen Adressraumes. Die Ausführungseinheiten werden in diesem Zusammenhang auch als **Threads** bezeichnet. Auf die Unterscheidung zwischen Threads und Prozessen gehen wir in diesem Kapitel genauer ein. Zum Austausch von Informationen zwischen Threads brauchen daher auch keine Kommunikationsoperationen verwendet zu werden. Stattdessen muss der Zugriff auf die gemeinsamen Variablen koordiniert werden, damit nicht verschiedene Threads zum gleichen Zeitpunkt dieselbe Variable zu manipulieren versuchen, da dies zu inkonsistenten Werten führen kann. Programmbibliotheken zur Unterstützung der Programmierung mit gemeinsamen Variablen stellen daher auch vielfältige Mechanismen zur Synchronisation von Threads beim Zugriff auf gemeinsame Variablen zur Verfügung, auf die wir im Laufe dieses Kapitels näher eingehen werden.

Modelle mit gemeinsamen Variablen wurden zuerst im Bereich der Multiuser-Betriebssysteme für Ein-Prozessor-Maschinen eingesetzt, woher auch viele Konzepte und Erfahrungen zur effizienten und sicheren Abarbeitung von interagierenden Prozessen mit gemeinsamem Adressraum stammen, die nun Eingang in die Benutzung von Mehr-Prozessor-Systemen gefunden haben. Die vielfältigen Modelle und Bibliotheken zur Anwendungsprogrammierung mit gemeinsamen Variablen unterscheiden sich vor allem in der Ausprägung der jeweils unabhängig ausgeführten Programmteile und deren mögliche Interaktionen. In diesem Kapitel stellen wir verschiedene häufig verwendete Ansätze vor. Allgemeine Aspekte der Programmierung mit *Threads* werden in Abschnitt 6.1 dargestellt. Abschnitt 6.2.1 stellt die Programmierung mit Pthreads vor, einer standardisierten Schnittstelle zur Anwendungsprogram-

mierung mit Threads, die im POSIX-Standard (**P**ortable **O**perating **S**ystem **I**nterface) seit 1995 definiert ist und von vielen UNIX-Betriebssystemen unterstützt wird. Dabei ist das Threadmodell eine Verallgemeinerung des Prozessmodells in UNIX-Betriebssystemen. Die Programmiersprache Java unterstützt die Verwendung von Threads auf Sprachebene. Die bereitgestellten Sprachkonstrukte und vordefinierte Klassen und Methoden werden in Abschnitt 6.3 dargestellt. *OpenMP* in Abschnitt 6.4 ist ein verbreiteter Standard zur Programmierung mit gemeinsamen Variablen, der auf einem höheren Abstraktionsniveau aufsetzt und insbesondere für den Bereich des wissenschaftlichen Rechnens spezifiziert wurde. In Abschnitt 6.5 beschreiben wir UPC (Unified Parallel C) als Beispiel einer Programmiersprache, die einen verteilten gemeinsamen Speicher auf Sprachebene unterstützt.

6.1 Einführung in die Programmierung mit Threads

Die meisten der heute gebräuchlichen Betriebssysteme sind **Multitasking-Betriebssysteme**, d.h. zum gleichen Zeitpunkt können mehrere Prozesse aktiv sein. Die Prozesse werden von der CPU im Time-Sharing-Verfahren bedient, indem die CPU die Prozesse reihum für kurze *Zeitscheiben* ausführt. Die Ausführung der Prozesse durch die CPU wird vom CPU-Scheduler des Betriebssystems gesteuert.

Das zentrale Ausführungskonzept ist dabei also der Begriff eines Prozesses. Ein **Prozess** ist ein in Ausführung befindliches Programm und umfasst neben dem ausführbaren Programm alle Informationen, die zur Ausführung des Programms erforderlich sind. Dazu gehören die Daten des Programms, die z.B. auf einem Laufzeitstack oder einem Heap aufgehoben werden, die zum Ausführungszeitpunkt aktuellen Registerinhalte und der aktuelle Wert des Programmzählers, der die nächste auszuführende Instruktion des Prozesses angibt. Diese Informationen ändern sich während der Ausführung des Prozesses dynamisch. Ordnet der CPU-Scheduler der CPU einen anderen Prozess zur Ausführung zu, so muss der *Zustand* des suspendierten Prozesses gerettet werden, damit die Ausführung dieses Prozesses zu einem späteren Zeitpunkt mit diesem Zustand fortgesetzt werden kann. Die Zeit für die Durchführung dieses als *Kontextwechsel* bezeichneten Vorganges hängt dabei wesentlich von der Unterstützung der CPU zum Zwischenspeichern von Registerinhalten ab, siehe zum Beispiel [121].

Beim Erzeugen eines Prozesses muss dieser die zu seiner Ausführung erforderlichen Daten erhalten. Im UNIX-Betriebssystem kann ein Prozess mit Hilfe einer **fork**-Anweisung einen neuen Prozess erzeugen. Der neue Kindprozess ist eine identische Kopie des Elternprozesses zum Zeitpunkt des **fork**-Aufrufes, d.h. der Kindprozess arbeitet auf einer *Kopie* des Adressraumes des Elternprozesses und führt das gleiche Programm wie der Elternprozess aus, und zwar ab der der **fork**-Anweisung folgenden Anweisung. Da der Kindprozess eine eigene Prozessnummer hat, kann er in Abhängigkeit von seiner

Prozessnummer andere Anweisungen als der Elternprozess ausführen, vgl. [107] für eine ausführlichere Beschreibung. Eltern- und Kindprozess können über *Sockets*, dem Prozess-Kommunikationsmechanismus in UNIX, miteinander kommunizieren [27].

Bei Ein-Prozessor-Systemen kann der Grund für den Einsatz von mehreren miteinander kommunizierenden Prozessen darin bestehen, eine im Vergleich zu einer Ein-Prozess-Variante schnellere Abarbeitung einer Aufgabe zu erreichen, da geeignet zwischen den Prozessen gewechselt werden kann: Stellt einer der Prozesse eine I/O-Anfrage, auf deren Bearbeitung durch das Betriebssystem er wartet, so kann ein anderer Prozess während der Wartezeit ausgeführt werden. Bei Mehr-Prozessor-Systemen kann eine schnellere Abarbeitung erreicht werden, indem Prozesse auf verschiedene Prozessoren verteilt werden.

Da jeder Prozess einen eigenen Adressraum hat, ist die Erzeugung und Verwaltung von Prozessen je nach Größe des Adressraumes relativ zeitaufwendig. Weiter kann bei häufiger Kommunikation der Austausch von Daten über Sockets einen nicht unerheblichen Anteil der Ausführungszeit ausmachen. Um diese Nachteile zu beseitigen, wurde das Prozessmodell zum **Threadmodell** erweitert. Die Erweiterung besteht darin, dass jeder Prozess anstatt nur aus einem aus mehreren unabhängigen Kontrollflüssen bestehen kann, die während der Abarbeitung des Prozesses entsprechend einem Schedulingverfahren der CPU zur Ausführung zugeteilt werden können. Die Kontrollflüsse eines Prozesses werden als **Threads** bezeichnet. Das Wort Thread wurde gewählt, um anzudeuten, dass eine zusammenhängende, evtl. sehr lange Folge von Instruktionen abgearbeitet wird.

Ein wesentliches Merkmal von Threads besteht darin, dass die verschiedenen Threads *eines* Prozesses sich den Adressraum des Prozesses teilen, d.h. wenn ein Thread einen Wert im Adressraum ablegt, kann ein anderer Thread des gleichen Prozesses ihn unmittelbar darauf lesen. Damit ist der Informationsaustausch zwischen Threads im Vergleich zur Kommunikation der Prozesse über Sockets sehr schnell. Da die Threads eines Prozesses sich einen Adressraum teilen, braucht auch die Erzeugung von Threads wesentlich weniger Zeit als die von Prozessen. Das Kopieren des Adressraumes, das z.B. in UNIX beim Erzeugen von Prozessen mit einer *fork*-Anweisung notwendig ist, entfällt. Das Arbeiten mit mehreren Threads innerhalb eines Prozesses ist also wesentlich flexibler als das Arbeiten mit kooperierenden Prozessen, bietet aber die gleichen Vorteile. Insbesondere ist es möglich, bei Mehr-Prozessor-Systemen die Threads *eines* Prozesses auf verschiedenen Prozessoren auszuführen.

Wegen der beschriebenen Vorteile unterstützen viele derzeitige Betriebssysteme die Erzeugung von mehreren Threads pro Prozess. Viele der verfügbaren Thread-Bibliotheken basieren dabei auf dem IEEE-**POSIX**-Standard, der einen Standard für die Grunddienste eines Betriebssystems definiert und neben einem Standard für die Prozessverwaltung auch Schnittstellen für

die Entwicklung von Thread-Programmen definiert. Dieser Teil des POSIX-Standards POSIX.1c wird als **Pthreads** (für POSIX-Threads) bezeichnet, siehe Abschnitt 6.2, und wird insbesondere von UNIX-Betriebssystemen unterstützt.

Threads können auf Benutzerebene oder auf Betriebssystemebene implementiert werden. *Threads* auf Benutzerebene werden durch die *Thread-Bibliothek* ohne Beteiligung des Betriebssystems verwaltet, d.h. ein Wechsel des von der CPU ausgeführten Threads kann ohne Beteiligung des Betriebssystems erfolgen und ist daher in der Regel wesentlich schneller als der Wechsel bei Betriebssystem-Threads, da letzterer den Aufruf einer Betriebssystem-Routine und damit einen Wechsel zum Betriebssystem erfordert. Der Nachteil von Threads auf Benutzerebene liegt darin, dass das Betriebssystem keine Kenntnis von den Threads hat und nur gesamte Prozesse verwaltet. Wenn *ein* Thread eines Prozesses das Betriebssystem aufruft, um zum Beispiel eine I/O-Operation durchzuführen, wird der CPU-Scheduler des Betriebssystems den gesamten Prozess suspendieren und die CPU einem anderen Prozess zu teilen, da das Betriebssystem nicht weiß, dass innerhalb des Prozesses zu einem anderen Thread umgeschaltet werden kann. Dieser Nachteil besteht bei Betriebssystem-Threads nicht, da das Betriebssystem die Threads direkt verwaltet. Threads können nur dann durch das Betriebssystem verwaltet werden, wenn dieses dafür vorgesehen ist. Dies ist bei den meisten Betriebssystemen der Fall.

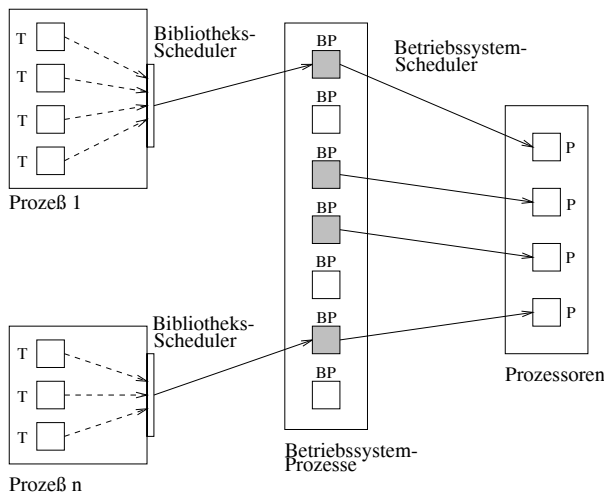


Abb. 6.1. Thread-Verwaltung *ohne* Betriebssystem-Threads. Der Scheduler der Thread-Bibliothek wählt den auszuführenden Thread *T* des Benutzerprozesses aus (N:1-Abbildung). Jedem Benutzerprozess ist *ein* Betriebssystemprozess *BP* zugeordnet. Der Betriebssystem-Scheduler wählt die zu einem bestimmten Zeitpunkt auszuführenden Betriebssystemprozesse aus und bildet sie auf die Prozessoren *P* ab.

Ausführungsmodelle für Threads. Wird eine Thread-Verwaltung durch das Betriebssystem nicht unterstützt, ist die Thread-Bibliothek für das Scheduling der Threads verantwortlich. *Alle* Benutzer-Threads eines Prozesses werden vom Bibliotheks-Scheduler auf *einen* Betriebssystemprozess abgebildet (was auch **N:1-Abbildung** oder *many-to-one mapping* genannt wird). Es existiert also eine eindeutige Abbildung zwischen Benutzerprozessen und Betriebssystemprozessen. Die Verwaltung der Betriebssystemprozesse erfolgt durch das Betriebssystem, vgl. Abbildung 6.1. Falls mehrere Prozessoren für die Ausführung zur Verfügung stehen, kann das Betriebssystem mehrere Betriebssystemprozesse gleichzeitig auf verschiedenen Prozessoren ausführen. Es ist aber bei dieser Organisation nicht möglich, verschiedene Threads *eines* Prozesses auf verschiedenen Prozessoren auszuführen.

Stellt das Betriebssystem eine Thread-Verwaltung zur Verfügung, gibt es für die Abbildung von Benutzer-Threads auf Betriebssystem-Threads zwei Möglichkeiten: Die erste besteht darin, für jeden Benutzer-Thread einen Betriebssystem-Thread zu erzeugen (was auch **1:1-Abbildung** oder *one-to-one mapping* genannt wird, vgl. Abbildung 6.2). Der Betriebssystem-Scheduler wählt den jeweils auszuführenden Betriebssystem-Thread aus und verwaltet bei Mehr-Prozessor-Systemen die Ausführung der Betriebssystem-Threads auf den verschiedenen Prozessoren. Da jeder Benutzer-Thread einem Betriebssystem-Thread zugeordnet ist, muss kein Bibliotheks-Scheduler eingesetzt werden. Wenn genügend Prozessoren zur Verfügung stehen, ist es möglich, dass verschiedene Benutzer-Threads eines Prozesses auf verschiedenen Prozessoren ausgeführt werden.

Die zweite Möglichkeit besteht darin, weiterhin ein zweistufiges Schedulingverfahren anzuwenden, vgl. Abbildung 6.3. Der Scheduler der Thread-Bibliothek ordnet die verschiedenen Threads der verschiedenen Prozesse einer vorgegebenen Menge von Betriebssystem-Threads zu (was auch **N:M-Abbildung** oder *many-to-many mapping* genannt wird), wobei ein Benutzer-Thread zu verschiedenen Zeitpunkten auf verschiedene Betriebssystem-Threads abgebildet werden kann. Je nach Thread-Bibliothek kann der Programmierer den Bibliotheks-Scheduler beeinflussen. Der Betriebssystem-Scheduler bildet die ausführbaren Betriebssystem-Threads auf die zur Verfügung stehenden Prozessoren ab. Das verwendete Schedulingverfahren ist dabei meist auf eine spezielle Hardware-Plattform angepasst, und der Programmierer hat keinen Einfluss auf die Auswahl der zu einem bestimmten Zeitpunkt ausgeführten Betriebssystem-Threads. Diese zweite Möglichkeit der Abbildung von Benutzer-Threads auf Betriebssystem-Threads bietet eine größere Flexibilität als die in Abbildung 6.2 dargestellte feste Zuordnung. Auf der einen Seite kann der Programmierer die Anzahl der Benutzer-Threads an den zu implementierenden Algorithmus anpassen, auf der anderen Seite kann das Betriebssystem die Anzahl der Betriebssystem-Threads so festlegen, dass eine effiziente Verwaltung und Abbildung auf die Prozessoren möglich ist.

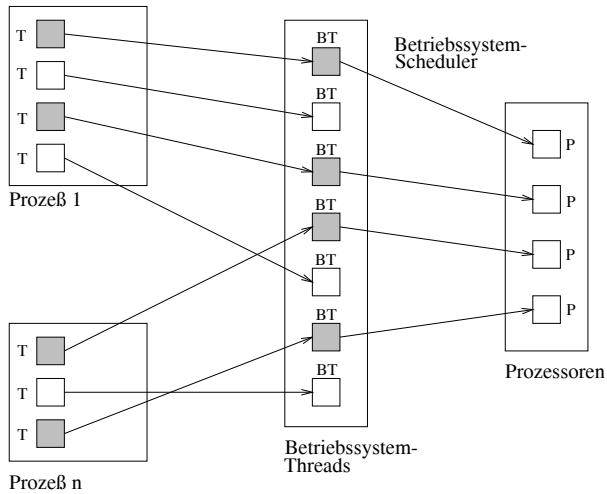


Abb. 6.2. Thread-Verwaltung mit Betriebssystem-Threads. Jeder Benutzer-Thread T wird eindeutig einem Betriebssystem-Thread BT zugeordnet (1:1-Abbildung). Betriebssystem-Threads BT werden vom Betriebssystem-Scheduler den Prozessoren P zur Ausführung zugeordnet.

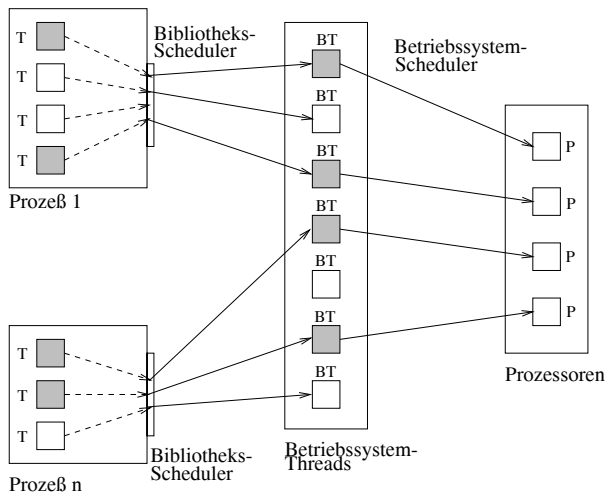


Abb. 6.3. Thread-Verwaltung mit Betriebssystem-Threads und zweistufigem Scheduling. Benutzer-Threads T verschiedener Prozesse werden einer Menge von Betriebssystem-Threads BT zugeordnet (N:M-Abbildung). Der Betriebssystem-Scheduler bildet ausführbare Betriebssystem-Threads auf Prozessoren ab.

Solaris 2. Ein Beispiel für ein Betriebssystem, das die gerade beschriebene zweistufige Abbildung von Benutzer-Threads auf Prozesse unterstützt, ist *Solaris 2* von Sun [103]. Solaris 2 unterstützt Benutzer-Threads über eine Thread-Bibliothek, die auf dem POSIX-Standard beruht und die Funktionen zur Erzeugung und Verwaltung der Threads zur Verfügung stellt. Da die Benutzer-Threads von der Thread-Bibliothek verwaltet werden, hat das Betriebssystem keine Kenntnis über die Anzahl und den Status der Benutzer-Threads. Neben den beschriebenen Betriebssystem-Threads unterstützt Solaris 2 *leichtgewichtige Prozesse* (*lightweight processes*, LWP), die für die Abbildung von Benutzer- auf Betriebssystem-Threads verwendet werden. Jedem LWP ist ein Betriebssystem-Thread zugeordnet. Die Benutzer-Threads werden von der Thread-Bibliothek auf LWPs abgebildet, vgl. Abbildung 6.4. Jeder Benutzerprozess besteht aus einer Anzahl von Benutzer-Threads, die vom Programmierer dynamisch nach den Gegebenheiten des Algorithmus erzeugt werden können, und einer (kleineren) Anzahl von LWPs, die den maximalen Parallelitätsgrad des Benutzerprozesses festlegen, und die von der Thread-Bibliothek in Zusammenarbeit mit dem Betriebssystem so erzeugt und evtl. wieder terminiert werden, dass eine effiziente Ausführung der Benutzer-Threads möglich ist. Bei Mehr-Prozessor-Systemen werden die Betriebssystem-Threads vom Betriebssystem auf die verschiedenen Prozessoren abgebildet. Wenn ein Betriebssystem-Thread blockiert, weil er zum Beispiel auf eine I/O-Operation wartet, werden auch der zugeordnete LWP und der Benutzer-Thread, den dieser ausführt, blockiert. Eine Blockierung des Benutzerprozesses kann in dieser Situation dadurch verhindert werden, dass ein anderer, nicht blockierter LWP des gleichen Benutzerprozesses auf dem gleichen Prozessor zur Ausführung gebracht wird, indem das Betriebssystem einen Wechsel zum zugehörigen Betriebssystem-Thread durchführt. Der Bibliotheks-Scheduler versucht, für jeden Prozess so viele LWPs zu erhalten, dass jederzeit zu einem nichtblockierten LWP umgeschaltet werden kann.

6.2 Programmiermodell und Grundlagen für Pthreads

Der Posix-Threads-Standard ist eine mögliche Realisierung des Threadmodells. In diesem Abschnitt geben wir einen Überblick über die Programmierung mit Pthreads mit den folgenden Unterpunkten. Abschnitt 6.2 beschreibt Grundlagen für die Thread-Programmierung. Abschnitt 6.2.1 gibt einen Überblick über die Prozesskontrolle im Pthreads-Standard. Abschnitt 6.2.2 beschreibt die Möglichkeiten zur Koordination von Threads beim Zugriff auf gemeinsame Variablen. Abschnitt 6.2.3 verwendet die vorgestellten Mechanismen zur Implementierung eines Taskpools, der eine dynamische Ablage und Entnahme von Tasks erlaubt. Abschnitt 6.2.6 stellt fortgeschrittenere Verfahren zur Steuerung von Threads vor, wobei insbesondere Scheduling-Verfahren für Threads besprochen werden. Wir können dabei nur auf den

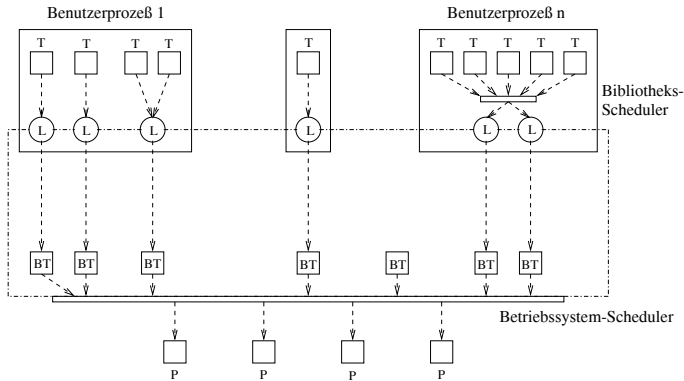


Abb. 6.4. Thread-Verwaltung in Solaris 2. Die Benutzer-Threads T eines Prozesses werden vom Bibliotheks-Scheduler auf leichtgewichtige Prozesse (LWP)s L des Prozesses abgebildet. Jedem LWP ist ein Betriebssystem-Thread BT zugeordnet. Die Betriebssystem-Threads BT werden vom Betriebssystem-Scheduler auf die Prozessoren P abgebildet.

wesentlichen Teil des Pthreads-Standards genauer eingehen. Für eine weiterführende Behandlung verweisen wir auf [19, 92, 103, 112, 127], an denen sich auch die hier vorgestellte Besprechung orientiert.

Wir nehmen im Folgenden an, dass ein zweistufiges Schedulingverfahren gemäß Abbildung 6.3 verwendet wird, da dies den allgemeinsten Fall darstellt. Der Programmierer hat bei diesem Modell die Aufgabe, das Quellprogramm in eine geeignete Anzahl von Benutzer-Threads zu zerlegen, die konkurrierend zueinander ausgeführt werden können.

Wie oben beschrieben wurde, werden Benutzer-Threads vom Bibliotheks-Scheduler auf Betriebssystem-Threads abgebildet und der Betriebssystem-Scheduler bringt die Betriebssystem-Threads bei Mehr-Prozessor-Systemen auf den verschiedenen Prozessoren zur Ausführung. Auf die Arbeitsweise des Bibliotheks-Schedulers hat der Programmierer nur geringen, auf die des Betriebssystem-Schedulers keinen Einfluss. Das Thread-Programmiermodell erlaubt es dem Programmierer also nicht, die Abbildung der von ihm definierten Benutzer-Threads auf die Prozessoren eines Parallelrechners, also das Mapping, selbst mit Hilfe eines in das parallele Programm integrierten Scheduling-Verfahrens abzubilden. Da der Programmierer sich *nicht* um das Scheduling kümmern muss und kann, wird seine Aufgabe auf der einen Seite wesentlich vereinfacht, auf der anderen Seite wird ihm damit auch die Möglichkeit genommen, selbst eine effiziente Abbildung auf die Prozessoren durchzuführen. Evtl. kann damit sogar die Situation auftreten, dass der Programmierer genau weiß, wie die von ihm erzeugten Threads verwaltet und ausgeführt werden sollten, diese Abbildung aber nicht realisieren kann, weil er keinen Einfluss auf das Scheduling hat. Die Übernahme des Scheduling

durch die Thread-Bibliothek beziehungsweise das Betriebssystem ist auch ein wesentlicher Unterschied zu früheren Ansätzen wie z.B. p4, bei denen der Programmierer die Taskpools und deren Verwaltung selbst programmieren musste. In den meisten Fällen kann man davon ausgehen, dass die Übernahme des Scheduling durch das System für den Programmierer ein Vorteil ist, der den Nachteil der fehlenden Einflussnahme überwiegt. In Abschnitt 6.2.6 gehen wir näher auf die Steuerungsmöglichkeiten des Programmierers beim Scheduling ein.

Die Programmierung mit Threads unterliegt keinem strengen Programmiermodell in Bezug auf die Zusammenarbeit und Koordination von Threads. Es haben sich aber einige Strukturierungsarten als geeignet für die Thread-Programmierung herausgebildet. Unter anderem sind folgende Arten der Zusammenarbeit möglich [127]:

- **Master-Slave-Modell:** Es gibt einen ausgezeichneten *Master-Thread*, der für die Steuerung des Programms verantwortlich ist und der für die Durchführung der anfallenden Berechnungen *Slave-Threads* erzeugt und deren Abarbeitung steuert, indem er den Slave-Threads Eingabedaten zur Verfügung stellt, die von diesen errechneten Resultate aufammelt und sich je nach Bedarf um deren Weiterverarbeitung durch evtl. neu erzeugte Slave-Threads kümmert.
- **Worker-Modell:** Bei Verwendung des Worker-Modells besteht ein paralleles Programm aus einer Menge von Worker-Threads, die die Durchführung der anfallenden Berechnungen untereinander koordinieren, *ohne* dass es einen ausgezeichneten Master-Thread gibt. Im Unterschied zum Master-Slave-Modell sind also alle Threads gleichberechtigt, und jeder der Worker-Threads kann bei Bedarf neue Worker-Threads erzeugen. Die so neu erzeugten Worker-Threads nehmen gleichberechtigt an den Berechnungen teil.
- **Pipelining-Modell:** Beim Pipelining-Modell besteht zwischen den beteiligten Threads eine Ein-/Ausgabebeziehung, d.h. die Threads sind logisch in einer durch Datenabhängigkeiten vorgegebenen Reihenfolge angeordnet, und Thread i erhält die Ausgabe von Thread T_{i-1} als Eingabe und produziert als Ausgabe die Eingabe für Thread T_{i+1} , $i = 1, 2, \dots$. Jeder der Threads verarbeitet seine Eingabeelemente in einer sequentiellen Reihenfolge und produziert nach Verarbeitung eines Eingabeelementes ein zugehöriges Element der Ausgabe. Somit können die Threads durch Anwendung des Pipeline-Prinzips trotz der Datenabhängigkeiten parallel zueinander ausgeführt werden. Üblicherweise ist bei Verwendung dieses Modells die Anzahl der beteiligten Threads relativ klein.

Die Threads eines Prozesses teilen sich einen gemeinsamen Adressraum, d.h. die globalen Variablen und alle dynamisch erzeugten Datenobjekte sind von allen erzeugten Threads des Prozesses zugreifbar. Für jeden Thread wird jedoch ein eigener Laufzeitstack gehalten, auf dem die von dem Thread aufgerufenen Funktionen mit ihren lokalen Variablen verwaltet werden. Die auf

dem Laufzeitstack verwalteten, also statisch deklarierten Daten, sind *lokale* Daten des zugehörigen Threads und können von anderen Threads *nicht* zugegriffen werden. Da der Laufzeitstack eines Threads nur so lange existiert, wie der Thread selber, kann ein Thread einen eventuellen Rückgabewert an einen anderen Thread nicht über seinen Laufzeitstack übergeben.

Wir werden in den folgenden Abschnitten den Pthreads-Standard für Thread-Bibliotheken kurz beschreiben. Dabei beschränken wir uns auf die wichtigsten Teile und verweisen auf [19, 92, 103, 112, 127] für eine vollständigere Behandlung. Die von einer Pthreads-Bibliothek verwendeten Datentypen, Schnittstellendefinitionen und Makros sind üblicherweise in der Headerdatei `<pthread.h>` abgelegt, die somit in jedes Pthreads-Programm eingebunden werden muss. Für die Funktionen und Datentypen wird vom Pthreads-Standard eine *Benennungskonvention* eingehalten, die Funktionen in der Form

```
pthread[_<object>]_<operation>()
```

benennt, wobei `<operation>` die auszuführende Operation und `<object>` optional das Objekt beschreibt, auf dem die Operation ausgeführt werden soll. So bezeichnet z.B. `pthread_mutex_init()` eine Funktion, die eine so genannte Mutexvariable initialisiert, das Objekt `<object>` ist also `mutex` und die Operation `<operation>` ist `init`. (Auf Mutexvariablen werden wir später noch eingehen.) Für Funktionen, die Threads manipulieren, also Threads zum Objekt haben, entfällt die Angabe von `<object>`. Die Funktion zur Erzeugung eines Threads wird also z.B. mit `pthread_create()` bezeichnet.

Alle Pthreads-Funktionen liefern den Wert 0 zurück, wenn sie fehlerfrei durchgeführt werden konnten. Wenn bei der Durchführung ein Fehler aufgetreten ist, wird ein Fehlercode aus `<error.h>` zurückgegeben. Diese Headerdatei sollte daher ebenfalls eingebunden werden.

Pthreads-Datentypen bezeichnen ähnlich wie in MPI (vgl. Abschnitt 5.1) grundsätzlich Objekte, deren genaue Implementierung dem Programmierer verborgen bleibt. Datentypen werden mit Namen der Form

```
pthread_<object>_t
```

bezeichnet, wobei `<object>` das Datenobjekt angibt. Eine Mutexvariable wird zum Beispiel durch den Typ `pthread_mutex_t` bezeichnet. Folgende Tabelle enthält Pthreads-Datentypen, die später zum Teil noch näher vorgestellt werden. Wird `_<object>` weggelassen, entsteht der Typ für einen Thread-Namen `pthread_t`.

Pthreads-Datentypen	Bedeutung
<code>pthread_t</code>	Thread-Namen
<code>pthread_mutex_t</code>	Mutexvariable
<code>pthread_cond_t</code>	Bedingungsvariable
<code>pthread_key_t</code>	Zugriffsschlüssel
<code>pthread_attr_t</code>	Thread-Attributobjekt
<code>pthread_mutexattr_t</code>	Mutex-Attributobjekt
<code>pthread_condattr_t</code>	Bedingungsvariablen-Attributobjekt
<code>pthread_once_t</code>	<i>one time initialization</i> Kontrollkontext

6.2.1 Erzeugung und Verwaltung von Pthreads

Beim Start eines Pthreads-Programms ist ein *Haupt-Thread* (*main thread*) aktiv, der die `main()`-Funktion des Programms ausführt. Der Haupt-Thread kann weitere Threads erzeugen, indem er die Funktion

```
int pthread_create (pthread_t *thread,
                  const pthread_attr_t *attr,
                  void *(*start_routine)(void *),
                  void *arg)
```

aufruft. Das erste Argument ist ein Zeiger auf ein Datenobjekt vom Typ `pthread_t`, in dem die Funktion eine Identifikation des erzeugten Threads ablegt, die auch als *Thread-Name* (*thread identifier*, TID) bezeichnet wird und mit der dieser Thread in nachfolgenden Aufrufen von Pthreads-Funktionen angesprochen werden kann. Das zweite Argument ist ein Zeiger auf ein Attributobjekt vom Typ `pthread_attr_t`, mit dessen Hilfe die Attribute des Threads festgelegt werden. Die Angabe von `NULL` bewirkt, dass ein Thread mit den Default-Attributen erzeugt wird. Sollen die Attribute abweichend gesetzt werden, muss die Attributdatenstruktur vor dem Aufruf von `pthread_create()` entsprechend besetzt werden. Die dafür zur Verfügung stehenden Funktionen beschreiben wir in Abschnitt 6.2.6. Das dritte Argument bezeichnet die Funktion `start_routine()`, die der Thread nach seiner Erzeugung ausführt. Die angegebene Funktion sollte ein einziges Argument vom Typ `void*` erwarten und einen Wert des gleichen Typs zurückliefern. Das vierte Argument ist ein Zeiger auf den Argumentwert, mit dem die Funktion `start_routine()` ausgeführt werden soll.

Ein Thread kann seinen Thread-Namen mit Hilfe der Funktion

```
pthread_t pthread_self()
```

in Erfahrung bringen. Da die Implementierung der Thread-Namen vom Typ `pthread_t` dem Programmierer nicht bekannt ist, sollte für den Vergleich von Thread-Namen die Funktion

```
int pthread_equal (pthread_t t1, pthread_t t2)
```

verwendet werden, die den Wert 0 zurückliefert, falls es sich bei `t1` und `t2` nicht um den gleichen Thread handelt.

Die Anzahl der von einem Prozess erzeugbaren Threads ist üblicherweise nach oben begrenzt, die maximal erzeugbare Anzahl von Threads ist dabei systemabhängig. Der POSIX-Standard legt fest, dass mindestens 64 Threads erzeugt werden können müssen. Die Funktion `pthread_create()` liefert den Wert `EAGAIN` zurück, wenn das Limit erreicht ist. Bei den meisten Systemen kann die maximale Anzahl von Threads auch durch den Aufruf

```
maxThreads = sysconf (_SC_THREAD_THREADS_MAX)
```

abgefragt werden. Ein Thread wird dadurch beendet, dass er die auszuführende Startfunktion beendet oder die Bibliotheksfunktion

```
void pthread_exit (void *valuep)
```

aufruft, wobei `valuep` den Rückgabewert bezeichnet, der an den aufrufenden Thread oder einen anderen Thread zurückgegeben wird, wenn dieser mit `pthread_join()` auf die Beendigung des Threads wartet. Wenn ein Thread seine Startfunktion beendet, wird die Funktion `pthread_exit()` *implizit* aufgerufen, und der Rückgabewert der Startfunktion wird zurückgegeben. Da nach dem Aufruf von `pthread_exit()` der aufgerufene Thread und damit auch der von ihm verwendete Laufzeitstack nicht mehr existiert, darf für den Rückgabewert `valuep` keine lokale Variable der Startfunktion oder einer anderen Funktion verwendet werden. Diese werden auf dem Laufzeitstack aufgehoben und können nach dessen Freigabe durch einen anderen Thread überschrieben werden. Stattdessen sollte eine globale oder eine dynamisch allokierte Variable verwendet werden.

Ein Thread kann auf die Beendigung eines anderen Threads warten, indem er die Bibliotheksfunktion

```
int pthread_join (pthread_t thread, void **valuep)
```

aufruft, wobei `thread` die Identifikation des Threads angibt, auf dessen Beendigung gewartet wird. Der aufrufende Thread wird so lange blockiert, bis der angegebene Thread beendet ist. Die Funktion `pthread_join` bietet also die Möglichkeit der *Synchronisation* von Threads. Der Rückgabewert des beendeten Threads `thread` wird dem wartenden Thread in der Variable `valuep` zurückgeliefert. Wenn mehrere Threads auf die Beendigung des gleichen Threads warten, ist nur einer der wartenden Threads erfolgreich. Alle anderen erhalten den Fehlercode `ESRCH` zurück.

Die Pthreads-Bibliothek legt für jeden erzeugten Thread eine interne Datenstruktur an, die die für die Abarbeitung des Threads notwendigen Informationen enthält. Diese Datenstruktur wird von der Bibliothek auch nach Beendigung eines Threads aufgehoben, damit ein anderer Thread eine `pthread_join()`-Operation erfolgreich durchführen kann. Durch den Aufruf der Funktion `pthread_join()` wird auch die interne Datenstruktur des angegebenen Threads freigegeben und kann danach nicht mehr verwendet werden. Wenn der Programmierer weiß, dass keine `pthread_join()`-Operation für einen Thread aufgerufen wird, kann er dies durch Aufruf der Funktion

```
int pthread_detach (pthread_t thread)
```

dem Laufzeitsystem mitteilen, damit dieses die Datenstruktur zur Platzersparnis freigeben kann. Um *mehrere* Argumente an die Startfunktion eines Threads zu übergeben, müssen diese in eine Datenstruktur gepackt werden, deren Adresse an die Funktion übergeben wird. Wenn mehrere Threads die gleiche Funktion mit unterschiedlichen Argumenten ausführen sollen, muss jedem Thread ein Zeiger auf eine *separate* Datenstruktur als Argument der Startfunktion mitgegeben werden, um zu vermeiden, dass Argumentwerte zu früh überschrieben werden oder dass verschiedene Threads ihre Argumentwerte konkurrierend manipulieren.

Beispiel: Abbildung 6.5 zeigt als Beispiel ein Programmfragment zur Durchführung einer Matrixmultiplikation [112]. **MA** und **MB** sind die zu multiplizierenden Matrizen der festen Größe 8×8 . Für jedes zu berechnende Element der Zielformat **MC** wird ein separater Thread mit Thread-Namen **thread[i]** erzeugt. Die Identifikationen der verschiedenen Threads werden im Feld **thread** abgelegt. Jedem Thread wird ein eigener Datenbereich vom Typ **matrix_type_t** mitgegeben, in den neben den Zeigern auf die beteiligten Matrizen die Position des zu berechnenden Eintrages von **MC** eingetragen wird. Jeder der erzeugten Threads führt die gleiche Funktion **thread_mult()** aus, die das Skalarprodukt einer Zeile von **MA** mit einer Spalte von **MB** berechnet. Nach Erzeugung der verschiedenen Threads wartet der Haupt-Thread mit **pthread_join()**, bis alle Threads beendet sind. □

6.2.2 Koordination von Threads

Die Threads eines Prozesses teilen sich einen gemeinsamen Adressraum und können daher konkurrierend auf gemeinsame Variablen zugreifen. Um dabei *zeitkritische Abläufe* (engl. *race conditions*) zu vermeiden, müssen die Zugriffe der beteiligten Threads koordiniert werden. Als wichtigste Hilfsmittel stellen Pthreads-Bibliotheken *Mutexvariablen* und *Bedingungsvariablen* zur Verfügung.

Eine **Mutexvariable** bezeichnet eine Datenstruktur des von der Thread-Bibliothek vorgegebenen Typs **pthread_mutex_t**, die dazu verwendet werden kann, den *wechselseitigen Ausschluss* (engl. *mutual exclusion*) bei der Manipulation gemeinsamer Daten sicherzustellen. Eine Mutexvariable kann zwei Zustände annehmen: gesperrt (*locked*) und ungesperrt (*unlocked*). Um einen wechselseitigen Ausschluss bei der Manipulation einer gemeinsamen Datenstruktur sicherzustellen, müssen die manipulierenden Threads jeweils folgendes Verhalten aufweisen: *Bevor* ein Thread eine Manipulation der gemeinsamen Datenstruktur startet, sperrt er die zugehörige Mutexvariable mit einem speziellen Funktionsaufruf. Wenn ihm dies gelingt, ist er der *Eigentümer* der Mutexvariable und er hat die Kontrolle über sie. Nach Beendigung der Manipulation der gemeinsamen Datenstruktur gibt der Thread die Sperre der Mutexvariable wieder frei.

```

#include<pthread.h>

typedef struct {
    int size, row, column;
    double (*MA)[8], (*MB)[8], (*MC)[8];
} matrix_type_t;

void main() {
    int row, column, size = 8, i;
    double MA[8][8], MB[8][8], MC[8][8];
    matrix_type_t *work;
    pthread_t thread[8*8];
    for (row=0; row<size; row++)
        for (column=0; column<size; column++) {
            work = (matrix_type_t *) malloc (sizeof (matrix_type_t));
            work->size = size;
            work->row = row;
            work->column = column;
            work->MA = MA; work->MB = MB; work->MC = MC;
            pthread_create (&(thread[column + row*8]), NULL,
                           (void *) thread_mult, (void *) work);
        }

    for (i=0; i<size*size; i++)
        pthread_join (thread[i], NULL);
}

void thread_mult (matrix_type_t *work) {
    int i, row = work->row, column = work->column;
    work -> MC[row][column] = 0;
    for (i=0; i < work->size; i++)
        work->MC[row][column] += work->MA[row][i] *
                                work->MB[i][column];
}

```

Abb. 6.5. Thread-Programm zur Realisierung einer Matrixmultiplikation zweier Matrizen MA und MB. Für jeden zu berechnenden Wert der Ergebnismatrix MC wird ein separater Thread erzeugt, der seine Daten in einer separaten Datenstruktur work zur Verfügung gestellt bekommt.

Versucht ein Thread die Kontrolle über eine von einem anderen Thread kontrollierte Mutexvariable zu erhalten, wird er blockiert, bis der andere Thread die Mutexvariable wieder freigegeben hat. Die Thread-Bibliothek stellt also sicher, dass jeweils nur *ein* Thread Kontrolle über eine Mutexvariable hat. Wenn die beschriebene Vereinbarung beim Zugriff auf eine Datenstruktur eingehalten wird, wird dadurch auch eine konkurrierende Manipulation dieser Datenstruktur ausgeschlossen. Sobald jedoch ein Thread die Datenstruktur manipuliert ohne vorher Kontrolle über die Mutexvariable erhalten zu haben, ist ein wechselseitiger Ausschluss *nicht* mehr garantiert.

Die Zuordnung zwischen einer Mutexvariablen und der ihr zugeordneten Datenstruktur erfolgt implizit dadurch, dass der Programmierer die Zugriffe auf die Datenstruktur durch Sperren bzw. Freigabe der Mutexvariablen schützt, eine explizite Zuordnung existiert nicht. Der Programmierer kann die Lesbarkeit eines Programms dadurch erleichtern, dass er die Datenstruktur und die für deren Kontrolle verwendete Mutexvariable in eine gemeinsame Struktur packt. Mutexvariablen haben den Typ `pthread_mutex_t` und können wie alle anderen Variablen deklariert oder dynamisch erzeugt werden. Bevor eine Mutexvariable benutzt werden kann, muss sie durch Aufruf der Funktion

```
int pthread_mutex_init (pthread_mutex_t *mutex,
                        const pthread_mutexattr_t *attr)
```

initialisiert werden. Für `attr = NULL` wird eine Mutexvariable mit den Default-Eigenschaften zur Verfügung gestellt. Durch eine geeignete Vorbesetzung können diese aber auch geändert werden, wie noch ausgeführt wird. Eine statisch deklarierte Mutexvariable `mutex` kann auch durch die Zuweisung

```
mutex = PTHREAD_MUTEX_INITIALIZER
```

mit den Default-Attributen initialisiert werden. Eine initialisierte Mutexvariable kann durch Aufruf der Funktion

```
int pthread_mutex_destroy (pthread_mutex_t *mutex)
```

wieder zerstört werden. Eine Mutexvariable sollte erst dann zerstört werden, wenn kein Thread mehr auf ihre Freigabe wartet. Eine zerstörte Mutexvariable kann durch eine erneute Initialisierung wieder weiterverwendet werden.

Ein Thread erhält die Kontrolle über eine Mutexvariable, indem er diese durch Aufruf der Funktion

```
int pthread_mutex_lock (pthread_mutex_t *mutex)
```

sperrt. Wird die angegebene Mutexvariable `mutex` bereits von einem anderen Thread kontrolliert, so wird der nun die Funktion `pthread_mutex_lock()` aufrufende Thread blockiert, bis der momentane Eigentümer die Mutexvariable wieder freigibt. Wenn mehrere Threads versuchen, die Kontrolle über eine Mutexvariable zu erhalten, werden die auf deren Freigabe wartenden Threads in einer Warteschlange gehalten. Welcher wartende Thread nach der Freigabe der Mutexvariable zuerst die Kontrolle über diese erhält, kann von den Prioritäten der wartenden Threads und dem verwendeten Scheduling-Verfahren abhängen. Die genaue Methode der Zuteilung hängt von der jeweiligen Implementierung der Thread-Bibliothek ab und ist nicht vom Pthreads-Standard vorgegeben. Ein Thread sollte nicht versuchen, Kontrolle über eine Mutexvariable zu erhalten, die er bereits kontrolliert, da dies zu einem Deadlock führen kann. Ein Thread kann eine von ihm kontrollierte Mutexvariable `mutex` durch Aufruf der Funktion

```
int pthread_mutex_unlock (pthread_mutex_t *mutex)
```

wieder freigeben. Nach Ausführen der Funktion ist die Mutexvariable im *unlocked*-Zustand. Wenn zum Zeitpunkt des Aufrufs von `pthread_mutex_unlock()` kein anderer Thread auf die Freigabe der Mutexvariable wartet, hat diese nach dem Aufruf keinen Eigentümer mehr. Wenn andere Threads auf die Freigabe der Mutexvariable warten, wird einer dieser Threads aufgeweckt und Eigentümer der Mutexvariable.

In manchen Situationen ist es sinnvoll, dass ein Thread feststellen kann, ob eine Mutexvariable von einem anderen Thread kontrolliert wird, ohne dass er dadurch blockiert wird. Dazu steht die Funktion

```
int pthread_mutex_trylock (pthread_mutex_t *mutex)
```

zur Verfügung. Beim Aufruf dieser Funktion erhält der aufrufende Thread die Kontrolle über die Mutexvariable `mutex`, wenn diese frei ist. Wenn diese zur Zeit unter der Kontrolle eines anderen Threads steht, liefert der Aufruf `EBUSY` zurück, führt aber *nicht* wie beim Aufruf von `pthread_mutex_lock()` zu einer Blockierung des aufrufenden Threads. Daher kann der aufrufende Thread so lange versuchen, die Kontrolle über die Mutexvariable zu erhalten, bis er erfolgreich ist (*spinlock*).

Beispiel: Abbildung 6.6 zeigt eine Programmskizze, die den Gebrauch einer Mutexvariable zur Sicherstellung des wechselseitigen Ausschlusses beim Zugriff auf eine globale verkettete Liste illustriert [112]. Die Knoten der verketteten Liste haben den Typ `node_t`. Die verkettete Liste wird von einer Mutexvariablen kontrolliert, die mit der von ihr kontrollierten Liste in einen Datentyp `list_t` zusammengefasst wird. Die verkettete Liste wird entsprechend dem Eintrag `index` der Knoten aufsteigend sortiert gehalten. Die Funktion `list_insert()` fügt ein neues Element in die Liste unter Einhaltung dieser Sortierung ein. Vor dem ersten Aufruf von `list_insert()` muss die Funktion `list_init()` im Haupt-Thread aufgerufen werden, so dass die Liste mit der Mutexvariablen initialisiert wird und für alle Threads global zur Verfügung steht. Vor Ausführung der Einfügeoperation versucht der ausführende Thread, die Kontrolle über die für die Liste verwendete Mutexvariable zu erhalten. Nach Beendigung des Einfügens gibt der Thread die Mutexvariable mit `pthread_mutex_unlock()` wieder frei. Damit ist sichergestellt, dass nicht zum gleichen Zeitpunkt verschiedene Threads Teile der Liste manipulieren. Der Zugriff auf die Liste wird also *sequentialisiert*. Da ein Programm die Funktion `list_insert()` benutzen kann, ohne sich um eine evtl. erforderliche Koordination der beteiligten Threads zu kümmern, wird die Funktion als *thread-sicher* (*threadsafe*) bezeichnet. □

Allgemein wird eine Funktion als thread-sicher bezeichnet, wenn sie gleichzeitig von mehreren Threads aufgerufen werden kann, ohne dass die beteiligten Threads zur Vermeidung von zeitkritischen Abläufen zusätzliche Operationen beim Funktionsaufruf ausführen müssen.

```

typedef struct node {
    int index;
    void *data;
    struct node *next;
} node_t;

typedef struct list {
    node_t *first;
    pthread_mutex_t mutex;
} list_t;

int list_init (list_t *listp)
{
    listp->first = NULL;
    pthread_mutex_init (&(listp->mutex), NULL);
    return 1;
}

void list_insert (int newindex, void *newdata, list_t *listp)
{
    node_t *current, *previous, *new;
    int found = FALSE;

    pthread_mutex_lock (&(listp->mutex));
    for (current = previous = listp->first; current != NULL;
         previous = current, current = current->next) {
        if (current->index == newindex) {
            found = TRUE; break; }
        else if (current->index > newindex) break;
    }
    if (!found) {
        new = (node_t *) malloc (sizeof (node_t));
        new->index = newindex;
        new->data = newdata;
        new->next = current;
        if (current == listp->first) listp->first = new;
        else previous->next = new;
    }
    pthread_mutex_unlock (&(listp->mutex));
}

```

Abb. 6.6. Thread-Implementierung einer einfach verketteten Liste. Die Funktion `list_insert()` kann von parallel arbeitenden Threads zum Einfügen von neuen Elementen in eine einfach verkettete Liste verwendet werden. In der vorliegenden Form ist diese Funktion nicht als Startfunktion eines Threads einsetzbar, da sie mehr als ein Argument hat. Um dies zu ändern, muss eine neue Datenstruktur mit den Argumenten von `list_insert()` definiert werden, die als Parameter übergeben wird und aus der am Anfang der Abarbeitung von `list_insert()` die entsprechenden Angaben entnommen werden.

In Abbildung 6.6 wird eine einzelne Mutexvariable zur Kontrolle der gesamten Liste verwendet, die Sperr-Granularität ist also *grobkörnig* (*coarse grain*). Damit kann unabhängig von der Länge der Liste zu einem Zeitpunkt nur eine Operation auf der Liste stattfinden. Eine Alternative könnte darin bestehen, für jedes Element der Liste eine eigene Mutexvariable zu verwenden oder die Liste in Bereiche einzuteilen und jedem Bereich eine Mutexvariable zuzuordnen. Die Sperr-Granularität wäre in diesem Fall *feinkörniger*, und mehrere Threads könnten zum gleichen Zeitpunkt verschiedene Teile der Liste manipulieren, es müssten aber mehr lock/unlock-Operationen als in der Programmskizze in Abbildung 6.6 durchgeführt werden.

Mutexvariablen, die mit den Default-Attributen initialisiert sind, können nur zur Synchronisation der Threads eines Prozesses verwendet werden. Wenn eine Pthreads-Bibliothek `_POSIX_THREAD_PROCESS_SHARED` definiert, kann dies geändert werden. Dazu muss durch Aufruf der Funktion

```
int pthread_mutexattr_init (pthread_mutexattr_t *attr)
```

ein Mutex-Attributobjekt initialisiert werden. Danach kann durch Aufruf der Funktion

```
int pthread_mutexattr_setpshared (pthread_mutexattr_t *attr,  
int pshared)
```

mit `pshared = PTHREAD_PROCESS_SHARED` das Mutex-Attributobjekt so geändert werden, dass eine nachfolgend mit `pthread_mutex_init (mutex, attr)` initialisierte Mutexvariable von den Threads aller Prozesse verwendet werden kann, die Zugriff auf den Datenbereich, der die Mutexvariable enthält, haben.

Gefahr von Deadlocks. Beim gleichzeitigen Sperren mehrerer Mutexvariablen durch mehrere Threads besteht die Gefahr, dass Deadlocks auftreten. Dies kann am Beispiel zweier Threads T_1 und T_2 gezeigt werden, die zwei Mutexvariablen ma und mb wie folgt sperren:

- Thread T_1 sperrt zuerst ma und dann mb ;
- Thread T_2 sperrt zuerst mb und dann ma ;

Falls Thread T_1 nach dem Sperren von ma vom Scheduler unterbrochen wird und danach Thread T_2 mb sperren kann, tritt ein Deadlock auf: T_2 wird beim Versuch, ma zu sperren, blockiert, da ma bereits von T_1 gesperrt ist; wenn T_1 wieder ausgeführt wird, wird T_1 beim Versuch, mb zu sperren, blockiert, da mb bereits von T_2 gesperrt ist. Somit sind beide Threads für immer blockiert, da beide darauf warten, dass der jeweils andere Thread die von ihm blockierte Mutexvariable wieder freigibt, was dieser nicht tun kann, da er blockiert ist.

Das Auftreten von Deadlocks kann durch Verwenden einer festen Sperr-Reihenfolge oder das Verwenden einer Backoff-Strategie vermieden werden, vgl. [19]. Bei Verwendung einer *festen Sperr-Reihenfolge* sperrt jeder beteiligte Thread die betroffenen Mutexvariablen immer in der gleichen Reihenfolge. Bei diesem Vorgehen muss der Thread T_2 die Mutexvariablen in der gleichen

Reihenfolge wie T_1 sperren, so dass auch T_2 zuerst ma und dann mb sperrt. Der oben skizzierte Deadlock kann dann nicht auftreten, da T_2 mb nicht sperren kann, wenn ma bereits von T_1 gesperrt ist. Zum Sperren von mb müsste T_2 zuerst ma sperren. Wenn ma bereits von T_1 gesperrt ist, wird T_2 beim Versuch, ma ebenfalls zu sperren, blockiert und kann daher mb nicht sperren. Die zu verwendende Sperr-Reihenfolge kann im Prinzip beliebig festgelegt werden, es muss jedoch sichergestellt werden, dass die festgelegte Reihenfolge im gesamten Programm verwendet wird. Sollte dies nicht mit der Programmstruktur vereinbar sein, sollte eine Backoff-Strategie verwendet werden.

Bei Verwendung einer **Backoff-Strategie** kann jeder der beteiligten Threads die Mutexvariablen in einer individuellen Reihenfolge sperren. Ein Thread muss sich aber zurückziehen (Backoff), wenn ein Sperrversuch fehlschlägt, d.h. der Thread gibt in diesem Fall alle vorher von ihm bereits erfolgreich gesperrten Mutexvariablen wieder frei. Danach beginnt der Thread den gesamten Sperrvorgang erneut von vorne. Eine Backoff-Strategie wird dadurch realisiert, dass jeder Thread für das Sperren der ersten Mutexvariable `pthread_mutex_lock()`, für das Sperren der folgenden Mutexvariablen `pthread_mutex_trylock()` verwendet. Liefert der Aufruf von `pthread_mutex_trylock()` den Resultatwert `EBUSY` zurück, bedeutet dies, dass die Mutexvariable bereits von einem anderen Thread gesperrt wurde. In diesem Fall gibt der Thread alle vorher erfolgreich gesperrten Mutexvariablen mit `pthread_mutex_unlock()` wieder frei.

Beispiel: Backoff-Strategie (siehe auch Abb. 6.7 und 6.8): Das Vorgehen wird in Abbildung 6.7 am Beispiel zweier Threads `f` und `b` illustriert, die für das Sperren dreier Mutexvariablen `m[0]`, `m[1]` und `m[2]` eine unterschiedliche Sperr-Reihenfolge verwenden, vgl. [19]. Der Thread `f` (forward) sperrt die Mutexvariablen in der Reihenfolge `m[0]`, `m[1]`, `m[2]` mit Hilfe der Funktion `lock_forward()`, der Thread `b` (backward) sperrt die Mutexvariablen in der umgekehrten Reihenfolge `m[2]`, `m[1]`, `m[0]` durch die Funktion `lock_backward()`, vgl. Abbildung 6.8. Beide Threads führen die Sperrversuche 10 Mal aus. Das Hauptprogramm `main()` in Abbildung 6.7 verwendet zwei Steuervariablen, die als Argumente eingelesen werden. Die Steuervariable `backoff` legt fest, ob eine Backoff-Strategie verwendet wird (Wert 1) oder nicht (Wert 0). Bei Verwendung von `backoff = 1` tritt wegen der Verwendung der Backoff-Strategie kein Deadlock auf. Für `backoff = 0` tritt in den meisten Fällen ein Deadlock auf, wenn `f` die Mutexvariable `m[0]` und `b` die Mutexvariable `m[2]` sperren kann.

Je nach Scheduling-Situation kann es jedoch passieren, dass auch ohne Backoff-Strategie kein Deadlock auftritt. Dies geschieht dann, wenn einer der beiden Threads in der Lage ist, alle drei Mutexvariablen zu sperren, bevor der andere Thread an die Reihe kommt. Zur Illustration dieser Abhängigkeit des Auftretens eines Deadlocks von der aktuellen Scheduling-Situation enthält das Beispiel in Abb. 6.7 und 6.8 einen Mechanismus, das Scheduling der Threads zu ändern bzw. den aktuell ausgeführten Thread zu blockieren.

Hierzu wird die zweite Steuervariable `yield_flag` genutzt. Für `yield_flag = 0` versucht jeder Thread wie bisher die Mutexvariablen ohne Unterbrechung zu sperren. Für `yield_flag = 1` ruft jeder Thread nach dem Sperren einer Mutexvariable `sched_yield()` auf und übergibt damit die Kontrolle an einen anderen Thread gleicher Priorität. Damit hat der andere Thread eine Chance, ebenfalls eine Mutexvariable zu sperren. Für `yield_flag = -1` ruft jeder Thread nach Sperren einer Mutexvariable `sleep(1)` auf und wartet damit für eine Sekunde. Auch in diesem Fall kommt der andere Thread an die Reihe und kann eine (andere) Mutexvariable sperren. In beiden Fällen tritt ohne Verwendung einer Backoff-Strategie ein Deadlock auf.

Der Aufruf von `pthread_exit()` im Hauptthread bewirkt, dass der Hauptthread, nicht jedoch der gesamte Prozess terminiert wird. Bei Verwendung eines normalen `return` würde der gesamte Prozess terminiert werden. $\square$

```
#include <pthread.h>
#include <sched.h>
#include <stdlib.h>
#include <stdio.h>
pthread_mutex_t m[3] = {
    PTHREAD_MUTEX_INITIALIZER,
    PTHREAD_MUTEX_INITIALIZER,
    PTHREAD_MUTEX_INITIALIZER };
int backoff = 1; // == 1: mit Backoff-Strategie
int yield_flag = 0; // > 0: Verwende sched_yield, < 0: sleep
int main(int argc, char *argv[]) {
    pthread_t f, b;
    if (argc > 1) backoff = atoi(argv[1]);
    if (argc > 2) yield_flag = atoi(argv[2]);
    pthread_create(&f, NULL, lock_forward, NULL);
    pthread_create(&b, NULL, lock_backward, NULL);
    pthread_exit(NULL); // die beiden anderen Threads laufen weiter
}
```

Abb. 6.7. Steuerprogramm zur Illustration der Arbeitsweise einer Backoff-Strategie.

Eine Backoff-Strategie braucht im Vergleich zur Verwendung einer festen Sperr-Reihenfolge typischerweise mehr Laufzeit, da Threads sich bei einem nicht erfolgreichen Sperrversuch zurückziehen und den Sperrvorgang erneut von vorne starten müssen. Auf der anderen Seite ist eine Backoff-Strategie flexibler als eine feste Sperr-Reihenfolge, da bei ersterer keine feste Sperr-Reihenfolge eingehalten werden muss. Beide Vorgehensweisen können auch gemischt werden.

Bedingungsvariablen. Mutexvariablen werden in erster Linie dazu verwendet, den wechselseitigen Ausschluss beim Zugriff auf globale Datenstrukturen sicherzustellen. Im Prinzip können Mutexvariablen jedoch auch dazu

```

void *lock_forward(void *arg) {
    int iterate, i, status;
    for (iterate = 0; iterate < 10; iterate++) {
        for (i = 0; i < 3; i++) { // Sperr-Reihenfolge vorwärts
            if (i == 0 || !backoff)
                status = pthread_mutex_lock(&m[i]);
            else status = pthread_mutex_trylock(&m[i]);
            if (status == EBUSY)
                for (; i >= 0; i--) pthread_mutex_unlock(&m[i]);
            else printf("forward locker got mutex %d\n", i);
            if (yield_flag) {
                if (yield_flag > 0) sched_yield(); // Wechsel des Threads
                else sleep(1); // Blockieren des ausgeführten Threads
            }
        }
        for (i = 2; i >= 0; i--)
            pthread_mutex_unlock(&m[i]);
        sched_yield(); // Neuer Versuch mit evtl. geänderter Reihenfolge
    }
}

void *lock_backward(void *arg) {
    int iterate, i, status;
    for (iterate = 0; iterate < 10; iterate++) {
        for (i = 2; i >= 0; i--) { // Sperr-Reihenfolge rückwärts
            if (i == 2 || !backoff)
                status = pthread_mutex_lock(&m[i]);
            else status = pthread_mutex_trylock(&m[i]);
            if (status == EBUSY)
                for (; i < 3; i++) pthread_mutex_unlock(&m[i]);
            else printf("backward locker got mutex %d\n", i);
            if (yield_flag)
                if (yield_flag > 0) sched_yield();
                else sleep(1);
        }
        for (i = 0; i < 3; i++)
            pthread_mutex_unlock(&m[i]);
        sched_yield();
    }
}

```

Abb. 6.8. Funktionen `lock_forward` und `lock_backward` zur Backoff-Strategie zur Sperrung der Mutexvariablen in entgegengesetzter Reihenfolge.

verwendet werden, auf das Eintreten einer Bedingung zu warten, die vom Zustand globaler Datenstrukturen abhängt. Dazu verwendet der zugreifende Thread eine oder mehrere Mutexvariablen zum Schutz des Zugriffs auf die globalen Daten und wertet die gewünschte Bedingung von Zeit zu Zeit aus, indem er mit Hilfe der Mutexvariablen geschützt auf die entsprechenden globalen Daten zugreift. Wenn die Bedingung erfüllt ist, kann der Thread die beabsichtigte Operation ausführen. Diese Vorgehensweise hat den Nachteil, dass der auf das Eintreten der Bedingung wartende Thread die Bedingung evtl. sehr oft auswerten muss, bis sie erfüllt ist, und dabei CPU-Zeit verbraucht (*aktives Warten*). Um diesen Nachteil zu beheben, stellt der Pthreads-Standard *Bedingungsvariablen* zur Verfügung.

Eine **Bedingungsvariable** (engl. *condition variable*) ist eine Datenstruktur, die einem Thread erlaubt, auf das Eintreten einer beliebigen Bedingung zu warten. Für Bedingungsvariablen wird ein Signalmechanismus zur Verfügung gestellt, der den wartenden Thread während der Wartezeit blockiert, so dass er keine CPU-Zeit verbraucht, und wieder aufweckt, sobald die angegebene Bedingung erfüllt ist. Um diesen Mechanismus zu verwenden, muss der verwendende Thread neben der Bedingungsvariablen einen *Bedingungsausdruck* angeben, der die Bedingung bezeichnet, auf deren Erfüllung der Thread wartet. Eine Mutexvariable muss die Auswertung des Bedingungsausdrucks schützen. Letzteres ist notwendig, da der Bedingungsausdruck in der Regel auf globale Datenstrukturen zugreift, die von anderen Threads konkurrierend manipuliert werden können.

Bedingungsvariablen haben den Typ `pthread_cond_t`. Nach der Deklaration oder der dynamischen Erzeugung einer Bedingungsvariablen muss diese initialisiert werden, bevor sie weiterverwendet werden kann. Dies kann dynamisch durch Aufruf der Funktion

```
int pthread_cond_init (pthread_cond_t *cond,
                      const pthread_condattr_t *attr)
```

geschehen. Dabei ist `cond` ein Zeiger auf die Bedingungsvariable und `attr` ein Zeiger auf eine Attribut-Datenstruktur für Bedingungsvariablen. Für `attr = NULL` erfolgt eine Initialisierung mit den Default-Attributen. Diese kann auch bei der Deklaration einer Bedingungsvariablen durch Verwendung eines Makros erfolgen:

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER.
```

Diese Form der Initialisierung ist für mit `malloc()` dynamisch erzeugte Bedingungsvariablen *nicht* anwendbar. Eine mit `pthread_cond_init()` dynamisch initialisierte Bedingungsvariable `cond` sollte durch Aufruf der Funktion

```
int pthread_cond_destroy (pthread_cond_t *cond)
```

zerstört werden, wenn sie nicht mehr gebraucht wird, damit das Laufzeitsystem die für die Bedingungsvariable abgelegte Information freigeben kann. Statisch initialisierte Bedingungsvariablen müssen nicht zerstört werden.

Eine Bedingungsvariable muss eindeutig mit einer Mutexvariablen assoziiert sein, und alle Threads, die zur gleichen Zeit auf die Bedingungsvariable warten, müssen die gleiche Mutexvariable verwenden, d.h. für eine Bedingungsvariable dürfen von verschiedenen Threads nicht zur gleichen Zeit verschiedene Mutexvariablen verwendet werden. Eine Mutexvariable kann jedoch verschiedenen Bedingungsvariablen zugeordnet werden. Nach dem Sperren der Mutexvariablen mit `pthread_mutex_lock()` kann ein Thread durch Aufruf der Funktion

```
int pthread_cond_wait (pthread_cond_t *cond,
                      pthread_mutex_t *mutex)
```

auf das Eintreten einer Bedingung warten, die als Bedingung einer umgebenden Kontrollanweisung formuliert werden muss. Dabei bezeichnet `cond` die Bedingungsvariable und `mutex` die assoziierte Mutexvariable. Eine Bedingungsvariable sollte nur mit einer Bedingung assoziiert sein, da sonst die Gefahr von Deadlocks oder zeitkritischen Abläufen vorliegt [19]. Die typische Verwendung hat dabei folgendes Aussehen:

```
pthread_mutex_lock (&mutex);
while (!Bedingung)
    pthread_cond_wait (&cond, &mutex);
pthread_mutex_unlock (&mutex);
```

Die Auswertung der Bedingung wird zusammen mit dem Aufruf von `pthread_cond_wait()` unter dem Schutz der Mutexvariablen `mutex` ausgeführt, um sicherzustellen, dass die Bedingung sich zwischen ihrer Auswertung und dem Aufruf von `pthread_cond_wait()` nicht durch Berechnungen anderer Threads ändert. Daher muss auch gewährleistet sein, dass jeder andere Thread eine Manipulation einer in den Bedingungen auftretenden gemeinsamen Variable mit der gleichen Mutexvariablen schützt.

- Wenn bei der Ausführung des Programmsegments die angegebene Bedingung erfüllt ist, wird die `pthread_cond_wait()`-Funktion *nicht* aufgerufen, und der ausführende Thread arbeitet nach `pthread_mutex_unlock()` das nachfolgende Programm weiter ab.
- Wenn dagegen die Bedingung *nicht* erfüllt ist, wird `pthread_cond_wait()` aufgerufen mit dem Effekt, dass der ausführende Thread *gleichzeitig* die Kontrolle über die Mutexvariable freigibt und so lange bezüglich der Bedingungsvariable blockiert, bis er von einem anderen Thread mit einer `pthread_cond_signal()`-Anweisung aufgeweckt wird, vergleiche unten. Wird der Thread durch diese Anweisung wieder aufgeweckt, versucht er auch automatisch, die Kontrolle über die Mutexvariable `mutex` zurückzuhalten. Hat bereits ein anderer Thread Kontrolle über die Mutexvariable `mutex`, so wird der aufgeweckte Thread unmittelbar nach dem Aufwecken so lange bezüglich der Mutexvariable blockiert, bis er diese kontrolliert.

Erst wenn der aufgeweckte Thread die Mutexvariable kontrolliert, kann er mit der Abarbeitung seines Programms fortfahren, was zunächst in der erneuten Abarbeitung der Bedingung besteht.

Die Programmierung sollte sicherstellen, dass der blockierte Thread nur dann aufgeweckt wird, wenn die angegebene Bedingung erfüllt ist. Trotzdem ist es sinnvoll, die Bedingung nach dem Aufwecken noch einmal zu überprüfen, da ein gleichzeitig aufgeweckter oder zeitgleich arbeitender Thread, der die Kontrolle über die Mutexvariable zuerst erhält, die Bedingung oder in der Bedingung enthaltene gemeinsame Daten modifizieren kann und so die Bedingung nicht mehr erfüllt ist.

Zum Aufwecken von bezüglich einer Bedingungsvariable wartenden Threads stehen die beiden folgenden Funktionen zur Verfügung:

```
int pthread_cond_signal (pthread_cond_t *cond)
int pthread_cond_broadcast (pthread_cond_t *cond).
```

Ein Aufruf von `pthread_cond_signal()` weckt *einen* bezüglich der Bedingungsvariable `cond` wartenden Thread auf, wenn die Bedingung erfüllt ist. Wartet kein Thread, hat der Aufruf keinen Effekt. Warten mehrere Threads, wird ein Thread anhand der Prioritäten der Threads und der verwendeten Scheduling-Strategie ausgewählt. Ein Aufruf von `pthread_cond_broadcast()` weckt *alle* bezüglich der Bedingungsvariablen `cond` wartenden Threads auf. Dabei kann aber höchstens *einer* dieser Threads die Kontrolle über die mit der Bedingungsvariablen assoziierten Mutexvariable erhalten, die anderen bleiben bezüglich der Mutexvariablen blockiert.

Die Funktionen `pthread_cond_signal()` und `pthread_cond_broadcast()` dienen dazu, einen oder mehrere auf das Eintreten einer Bedingung wartende Threads aufzuwecken, wenn die Bedingung erfüllt ist. Zum Test der Bedingung muss der ausführende Thread einen Sperrmechanismus mit der mit der zugehörigen Bedingungsvariable assoziierten Mutexvariable verwenden, damit eine konsistente Auswertung der Bedingung sichergestellt ist. Die Ausführung von `pthread_cond_signal()` und `pthread_cond_broadcast()` muss aber nicht unbedingt mit dem Sperrmechanismus geschützt werden. Ein Aufruf außerhalb des Sperrmechanismus hat den Nachteil, dass ein anderer Thread nach der Freigabe der Mutexvariable Kontrolle über diese erhalten und in der Bedingung auftretende gemeinsame Variablen modifizieren kann. Da der aufgeweckte Thread unmittelbar nach dem Aufwecken die Bedingung erneut überprüft, kann kein Fehler auftreten. Der Vorteil eines Aufrufes außerhalb des Sperrmechanismus liegt darin, dass beim Aufruf die Mutexvariable nicht besetzt ist, so dass der aufgeweckte Thread unmittelbar nach dem Aufwecken die Kontrolle über die Mutexvariable erhalten kann, ohne erneut bezüglich der Mutexvariable zu blockieren.

Als Variante von `pthread_cond_wait()` steht die Funktion

```
int pthread_cond_timedwait (pthread_cond_t *cond,
```

```
pthread_mutex_t *mutex,
const struct timespec *time)
```

zur Verfügung. Der Unterschied zu `pthread_cond_wait()` besteht darin, dass die Blockierung bezüglich der Bedingungsvariable `cond` aufgehoben wird, wenn die in `time` angegebene absolute Zeit abgelaufen ist. In diesem Fall wird die Fehlermeldung `ETIMEDOUT` zurückgeliefert. Die Datenstruktur vom Typ `timespec` ist definiert als

```
struct timespec {
    time_t tv_sec;
    long tv_nsec;
}
```

wobei `tv_sec` die Anzahl der Sekunden und `tv_nsec` die zusätzliche Anzahl von Nanosekunden der verwendeten Zeitscheiben angibt. Der Parameter `time` von `pthread_cond_timedwait()` gibt eine absolute Tageszeit und kein relatives Zeitintervall an. Eine typische Benutzung sieht wie folgt aus:

```
pthread_mutex_t m;
pthread_cond_t c;
struct timespec time;
pthread_mutex_lock (&m);
time.tv_sec = time (NULL) + 10;
time.tv_nsec = 0;
while (!Bedingung)
    if (pthread_cond_timedwait (&c, &m, &time) == ETIMEDOUT)
        timed_out_work();
pthread_mutex_unlock (&m);
```

In diesem Beispiel wartet der ausführende Thread maximal zehn Sekunden auf das Eintreten der Bedingung. Zur Besetzung von `time.tv_sec` wird die Funktion `time` aus `<time.h>` benutzt. (Der Aufruf `time (NULL)` gibt die absolute Zeit in Sekunden zurück, die seit dem 1. Januar 1970 vergangen ist.) Wenn die Bedingung nach zehn Sekunden noch nicht erfüllt ist, wird die Funktion `timed_out_work()` ausgeführt, und die Bedingung wird erneut überprüft.

Erweiterter Sperrmechanismus. Bedingungsvariablen können dazu verwendet werden, komplexe Synchronisationsmechanismen zu implementieren. Als Beispiel hierfür betrachten wir im Folgenden einen **Lese/Schreib-Sperrmechanismus** (*read/write lock*), der als Erweiterung des von Mutexvariablen zur Verfügung gestellten Sperrmechanismus aufgefasst werden kann. Wenn eine gemeinsame Datenstruktur mit einer Mutexvariable geschützt ist, kann zu einem Zeitpunkt jeweils nur ein Thread von der gemeinsamen Datenstruktur lesen bzw. auf die gemeinsame Datenstruktur schreiben. Die Idee des Lese/Schreib-Sperrmechanismus besteht darin, dies dahingehend zu

erweitern, dass zum gleichen Zeitpunkt eine beliebige Anzahl von lesenden Threads zugelassen wird, ein Thread zur Manipulation der Datenstruktur aber das ausschließliche Zugriffsrecht haben muss. Wir werden im Folgenden eine einfache Variante eines solchen modifizierten Sperrmechanismus beschreiben, vgl. auch [113]. Für eine komplexere und effizientere Implementierung verweisen wir auf [19, 92].

Für die Implementierung des erweiterten Sperrmechanismus werden **RW-Sperrvariablen** (*read/write lock variables*) verwendet, die mit Hilfe einer Mutex- und einer Bedingungsvariablen wie folgt definiert werden können:

```
typedef struct rw_lock {
    int num_r, num_w;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
} rw_lock_t;
```

Dabei gibt `num_r` die Anzahl der momentan erteilten Leseberechtigungen und `num_w` die Anzahl der momentan erteilten Schreibberechtigungen an. Letztere hat höchstens den Wert Eins. Die Mutexvariable soll die Zähler der Lese- und Schreibzugriffe schützen. Die Bedingungsvariable regelt den Zugriff auf eine RW-Sperrvariable.

Abbildung 6.9 zeigt Funktionen zur Verwaltung von RW-Sperrvariablen. Die Funktion `rw_lock_init()` dient der Initialisierung einer RW-Sperrvariable. Die Funktion `rw_lock_rlock()` fordert einen Lesezugriff auf die gemeinsame Datenstruktur an. Der Lesezugriff wird nur dann gewährt, wenn kein anderer Thread eine Schreibberechtigung erhalten hat. Sonst wird der anfordernde Thread blockiert, bis die Schreibberechtigung wieder zurückgegeben wird. Die Funktion `rw_lock_wlock()` dient zur Anforderung einer Schreibberechtigung. Diese wird nur gewährt, wenn kein anderer Thread eine Lese- oder Schreibberechtigung erhalten hat. Die Funktion `rw_lock_runlock()` dient der Rückgabe einer Leseberechtigung.

Sinkt durch die Rückgabe einer Leseberechtigung die Anzahl der lesend zugreifenden Threads auf Null, so wird ein evtl. auf eine Schreibberechtigung wartender Thread durch einen Aufruf von `pthread_cond_signal()` aufgeweckt. Die Funktion `rw_lock_wunlock()` dient entsprechend der Rückgabe einer Schreibberechtigung. Da höchstens *ein* schreibender Thread erlaubt ist, hat nach dieser Rückgabe kein Thread mehr eine Schreibberechtigung, und alle auf einen Lesezugriff wartenden Threads können durch `pthread_cond_broadcast()` aufgeweckt werden.

Die in Abbildung 6.9 skizzierte Implementierung von RW-Sperrvariablen gibt Lesezugriffen Vorrang vor Schreibzugriffen: Wenn ein Thread A eine Leseerlaubnis erhalten hat und Thread B auf eine Schreiberlaubnis wartet, erhalten andere Threads auch dann eine Leseerlaubnis, wenn diese nach der Schreiberlaubnis von B beantragt wird. Thread B erhält erst dann eine Schreiberlaubnis, wenn kein Thread mehr eine Leseerlaubnis beantragt hat. Je nach

```

int rw_lock_init (rw_lock_t *rwl) {
    rwl->num_r = rwl->num_w = 0;
    pthread_mutex_init (&(rwl->mutex), NULL);
    pthread_cond_init (&(rwl->cond), NULL);
    return 0;
}

int rw_lock_rlock (rw_lock_t *rwl) {
    pthread_mutex_lock (&(rwl->mutex));
    while (rwl->num_w > 0)
        pthread_cond_wait (&(rwl->cond), &(rwl->mutex));
    rwl->num_r ++;
    pthread_mutex_unlock (&(rwl->mutex));
    return 0;
}

int rw_lock_wlock (rw_lock_t *rwl) {
    pthread_mutex_lock (&(rwl->mutex));
    while ((rwl->num_w > 0) || (rwl->num_r > 0))
        pthread_cond_wait (&(rwl->cond), &(rwl->mutex));
    rwl->num_w ++;
    pthread_mutex_unlock (&(rwl->mutex));
    return 0;
}

int rw_lock_runlock (rw_lock_t *rwl) {
    pthread_mutex_lock (&(rwl->mutex));
    rwl->num_r --;
    if (rwl->num_r == 0) pthread_cond_signal (&(rwl->cond));
    pthread_mutex_unlock (&(rwl->mutex));
    return 0;
}

int rw_lock_wunlock (rw_lock_t *rwl) {
    pthread_mutex_lock (&(rwl->mutex));
    rwl->num_w --;
    pthread_cond_broadcast (&(rwl->cond));
    pthread_mutex_unlock (&(rwl->mutex));
    return 0;
}

```

Abb. 6.9. Funktionen zur Verwaltung von RW-Sperrvariablen (read/write lock variables).

Anwendung ist es sinnvoll, den Schreibzugriffen Vorrang vor Lesezugriffen zu geben, damit die Datenstruktur immer auf dem aktuellsten Stand ist. Wie dies erreicht werden kann, ist in [19] skizziert.

RW-Sperrvariablen können z.B. für die in Abbildung 6.6 skizzierte Verwaltung einer gemeinsamen verketteten Liste verwendet werden, die die

dort verwendete Mutexvariable ersetzt. In der Funktion `list_insert()` wird die Einfügeoperation dann mit `rw_lock_wlock()` und `rw_lock_wunlock()` umgeben. In einer Funktion zur Suche nach einem Eintrag kann dagegen `rw_lock_rlock()` und `rw_lock_runlock()` verwendet werden.

Einmalige Initialisierung. In manchen Situationen soll eine bestimmte Aufgabe, wie z.B. der Aufruf einer Initialisierungsfunktion oder das Öffnen einer Datei, genau einmal ausgeführt werden, wobei man aber wegen nicht vorhersagbarer Ausführungszeiten der beteiligten Threads nicht a priori festlegen will, welcher Thread die Aufgabe ausführen soll. Dieses Problem kann durch Verwendung einer mit Null initialisierten boolschen Variable gelöst werden, die beim Ausführen der durchzuführenden Aufgabe auf Eins gesetzt wird. Der Zugriff auf die boolsche Variable wird mit einer statisch initialisierten Mutexvariable geschützt. Ein Thread führt bei dieser Lösung die gewünschte Aufgabe nur aus, wenn der Wert der boolschen Variable Null ist. Obwohl dies eine gute Lösung ist, stellt der Pthreads-Standard eine zusätzliche Lösungsmöglichkeit zur Verfügung. Dazu wird eine Kontrollvariable vom Typ `pthread_once_t` verwendet werden, die statisch initialisiert werden muss. Für eine Kontrollvariable `once_control` geschieht dies durch

```
pthread_once_t once_control = PTHREAD_ONCE_INIT.
```

Die einmalig auszuführende Aufgabe muss in *eine* Funktion gepackt werden, die keine Parameter hat. Wenn `once_routine()` diese Funktion ist, können die beteiligten Threads durch Aufruf der Funktion

```
pthread_once (pthread_once_t *once_control,  
              void (*once_routine)(void))
```

eine *einmalige* Initialisierung erreichen, auch wenn die Funktion von verschiedenen Threads evtl. mehrfach aufgerufen wird. Wenn beim Aufruf der Funktion `pthread_once()` festgestellt wird, dass noch kein Thread die mitgelieferte Argumentfunktion ausgeführt hat, führt der aufrufende Thread diese Funktion aus. Wenn festgestellt wird, dass die Argumentfunktion bereits von einem anderen Thread beendet wurde, wird die Kontrolle direkt an den ausführenden Thread zurückgegeben. Wenn ein anderer Thread gerade dabei ist, die Argumentfunktion auszuführen, wird die Kontrolle erst zurückgegeben, wenn die Argumentfunktion beendet ist.

6.2.3 Implementierung eines Taskpools

Eine naheliegende Gestaltung eines Thread-Programms besteht darin, für abzuarbeitende Aufgaben oder Funktionen (also allgemein Tasks) jeweils einen Thread zu erzeugen, der genau diese Task abarbeitet und anschließend wieder zerstört wird. Dies kann je nach Anwendung dazu führen, dass sehr viele Threads erzeugt und wieder zerstört werden, was einen nicht unerheblichen Zeitaufwand verursachen kann, da jedesmal die Thread-Bibliothek benötigt

wird. Für viele Anwendungen kann eine effizientere parallele Implementierung mit Hilfe eines **Taskpools** erreicht werden. Die Idee eines Taskpools besteht darin, eine Datenstruktur zu halten, in der die noch abzuarbeitenden Programmteile (Tasks) abgelegt sind. Für die Abarbeitung der Tasks wird eine *fixe* Anzahl von Threads verwendet, die zu Beginn des Programms vom Haupt-Thread erzeugt werden und bis zum Ende des Programms existieren. Für die Threads stellt der Taskpool eine gemeinsame Datenstruktur dar, auf die sie zugreifen und die dort abgelegten Tasks entnehmen und anschließend abarbeiten. Während der Abarbeitung einer Task kann ein Thread neue Tasks erzeugen und diese in den Taskpool einfügen. Die Abarbeitung des parallelen Programms ist beendet, wenn der Taskpool leer ist und jeder Thread seine Tasks abgearbeitet hat.

Der Vorteil dieses Abarbeitungsschemas besteht darin, dass auf der einen Seite nur eine feste Anzahl von Threads erzeugt werden muss, dass also der Aufwand zur Threaderzeugung unabhängig von der Problemgröße ist und relativ gering gehalten werden kann, dass aber auf der anderen Seite Tasks dynamisch erzeugt werden können und dass auch adaptive und irreguläre Anwendungen effizient abgearbeitet werden können. Wir beschreiben im Folgenden eine einfache Implementierung eines Taskpools, vgl. [112]. Weitere Implementierungen sind zum Beispiel in [19, 92] beschrieben.

Abbildung 6.10 zeigt die Datenstruktur für den Taskpool und eine Funktion zur Initialisierung des Taskpools. Der Datentyp `work_t` beschreibt eine einzelne Task des Taskpools. Die Beschreibung besteht aus je einem Zeiger auf die auszuführende Funktion und auf das Argument dieser Funktion. Die einzelnen Tasks sind durch Zeiger `next` in Form einer einfach verketteten Liste miteinander verbunden. Der Datentyp `tpool_t` beschreibt die gesamte Datenstruktur eines Taskpool. Dabei bezeichnet `num_threads` die Anzahl der für die Abarbeitung verwendeten Threads, das Feld `threads` enthält Zeiger auf die abarbeitenden Threads. Die Einträge `max_size` und `current_size` geben die maximale bzw. aktuelle Anzahl von eingetragenen Tasks an. Die Zeiger `head` und `tail` zeigen auf den Anfang bzw. das Ende der Taskliste.

Die Mutexvariable `lock` wird verwendet, um den wechselseitigen Ausschluss bei der Manipulation des Taskpools durch die Threads sicherzustellen. Wenn ein Thread versucht, eine Task aus einem leeren Taskpool zu entnehmen, wird er bezüglich der Bedingungsvariable `not_empty` blockiert. Fügt ein Thread eine Task in einen leeren Taskpool ein, wird ein evtl. bezüglich der Bedingungsvariable `not_empty` blockierter Thread aufgeweckt. Wenn ein Thread versucht, eine Task in einen vollen Taskpool einzufügen, wird er bezüglich der Bedingungsvariable `not_full` blockiert. Entnimmt ein Thread eine Task aus einem vollen Taskpool, wird ein evtl. bezüglich der Bedingungsvariable `not_full` blockierter Thread aufgeweckt.

Die Funktion `tpool_init()` in Abbildung 6.10 initialisiert einen Taskpool, indem sie die Datenstruktur allokiert, mit den als Argument mitgelieferten Werten initialisiert und die zur Abarbeitung vorgesehene Anzahl von

```

typedef struct work {
    void (*routine)();
    void *arg;
    struct work *next;
} work_t;

typedef struct tpool {
    int num_threads, max_size, current_size;
    pthread_t *threads;
    work_t *head, *tail;
    pthread_mutex_t lock;
    pthread_cond_t not_empty, not_full;
} tpool_t;

tpool_t *tpool_init (int num_threads, int max_size) {
    int i;
    tpool_t *tpl;

    tpl = (tpool_t *) malloc (sizeof (tpool_t));
    tpl->num_threads = num_threads;
    tpl->max_size = max_size;
    tpl->current_size = 0;
    tpl->head = tpl->tail = NULL;

    pthread_mutex_init (&(tpl->lock), NULL);
    pthread_cond_init (&(tpl->not_empty), NULL);
    pthread_cond_init (&(tpl->not_full), NULL);
    tpl->threads = (pthread_t *)malloc(sizeof(pthread_t)*num_threads);
    for (i=0; i<num_threads; i++)
        pthread_create (&(tpl->threads[i]), NULL,
                        tpool_thread, (void *) tpl);

    return tpl;
}

```

Abb. 6.10. Implementierung eines Taskpools (Teil 1): Die Datenstruktur `work_t` enthält eine Liste mit abzuarbeitenden Funktionen (Tasks), die Taskpool-Datenstruktur `tpool_t` enthält eine Task-Schlange mit Anfang `head` und Ende `tail` und eine Menge von Threads `threads`. Die Funktion `tpool_init()` dient zur Initialisierung einer Taskpool-Datenstruktur `tpl`.

Threads `tpool->threads[i]`, $i=0, \dots, \text{num\_threads}-1$, erzeugt. Jeder dieser Threads erhält eine Funktion `tpool_thread()` als Startroutine, die einen Taskpool `tpl` als Argument hat. Die in Abbildung 6.11 angegebene Funktion `tpool_thread()` dient der Abarbeitung von im Taskpool abgelegten Tasks. In jedem Durchlauf der Endlosschleife von `tpool_thread()` wird versucht, eine Task vom Anfang der Taskliste des Taskpools zu entnehmen. Wenn der Taskpool zur Zeit leer ist, wird der ausführende Thread bzgl. der Bedingungsvariable `not_empty` blockiert. Sonst wird eine Task `wl` vom Anfang der Taskschlange entnommen. War der Taskpool vor der Entnahme voll, wer-

```

void *tpool_thread (tpool_t *tpl) {
    work_t *wl;

    for( ; ; ) {
        pthread_mutex_lock (&(tpl->lock));
        while (tpl->current_size == 0)
            pthread_cond_wait (&(tpl->not_empty), &(tpl->lock));
        wl = tpl->head;
        tpl->current_size--;
        if (tpl->current_size == 0)
            tpl->head = tpl->tail = NULL;
        else tpl->head = wl->next;
        if (tpl->current_size == tpl->max_size - 1)
            pthread_cond_broadcast (&(tpl->not_full));
        pthread_mutex_unlock (&(tpl->lock));
        (*(wl->routine))(wl->arg);
        free(wl);
    }
}

void tpool_insert (tpool_t *tpl, void (*routine)(), void *arg) {
    work_t *wl;

    pthread_mutex_lock (&(tpl->lock));
    while (tpl->current_size == tpl->max_size)
        pthread_cond_wait (&(tpl->not_full), &(tpl->lock));
    wl = (work_t *) malloc (sizeof (work_t));
    wl->routine = routine;
    wl->arg = arg;
    wl->next = NULL;
    if (tpl->current_size == 0) {
        tpl->tail = tpl->head = wl;
        pthread_cond_signal (&(tpl->not_empty));
    }
    else {
        tpl->tail->next = wl;
        tpl->tail = wl;
    }
    tpl->current_size++;
    pthread_mutex_unlock (&(tpl->lock));
}

```

Abb. 6.11. Implementierung eines Taskpools (Teil 2): Die Funktion `tpool_thread()` dient zur Entnahme und Abarbeitung von Tasks, die Funktion `tpool_insert()` dient zur Ablage von Tasks in einen Taskpool.

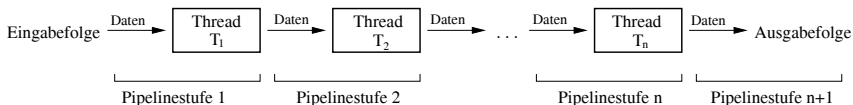
den alle Threads, die blockiert sind, weil sie eine Task abzulegen versuchen, mit einer `pthread_cond_broadcast()`-Anweisung aufgeweckt. Die Zugriffe von `tpool_thread` auf den Taskpool werden durch die Mutexvariable `lock` geschützt. Die Abarbeitung der Funktion `routine` einer entnommenen Task `wl` wird danach ausgeführt. Diese Abarbeitung kann die Erzeugung neuer Tasks beinhalten, die durch die Funktion `tpool_insert` in den Taskpool `tpl` eingetragen werden.

Die Funktion `tpool_insert()` fügt Tasks in den Taskpool ein. Wenn der Taskpool voll ist, wird der ausführende Thread bzgl. der Bedingungsvariable `not_full` blockiert. Ist der Taskpool nicht voll, so wird eine Task mit den entsprechenden Daten belegt und an das Ende der Taskschlange gehängt. War diese vor dem Anhängen leer, wird ein Thread, der bezüglich der Bedingungsvariable `not_empty` blockiert ist, aufgeweckt. Die Manipulationen des Taskpools `tpl` werden wieder durch die Mutexvariable geschützt.

Die skizzierte Implementierung eines Taskpools ist insbesondere für ein Master-Slave-Modell geeignet, in dem ein Master-Thread mit `tpool_init()` die gewünschte Anzahl von Slave-Threads erzeugt, von denen jeder die Funktion `tpool_thread()` abarbeitet. Die zu bearbeitenden Tasks können vom Master-Thread durch Aufruf von `tpool_insert()` in den Taskpool eingetragen werden. Werden bei der Bearbeitung einer Task neue Tasks erzeugt, können diese auch vom ausführenden Slave-Thread eingetragen werden. Die Beendigung der Slave-Threads wird vom Master-Thread übernommen. Dazu werden alle bezüglich der Bedingungsvariablen `not_empty` und `not_full` blockierten Threads aufgeweckt und beendet. Sollte ein Thread gerade eine Task bearbeiten, wird auf die Beendigung der Abarbeitung gewartet bevor der Thread beendet wird.

6.2.4 Parallelität durch Pipelining

Beim Pipeliningmodell wird ein Strom von Daten nacheinander durch eine Folge von Threads $T_1, \dots, T_n$ verarbeitet, wobei jeder Thread T_i bestimmte Operationen auf jedem Element des Datenstroms ausführt und die modifizierten Daten an den Nachfolgethread weitergibt:



Es besteht also eine Ein-/Ausgabe-Beziehung zwischen den Threads: Thread T_i erhält die Ausgabe von Thread T_{i-1} als Eingabe und produziert Daten für Thread T_{i+1} ($1 < i < n$). Thread T_1 liest die Eingabefolge, Thread T_n produziert die Ausgabefolge. Nach einer Anlaufphase können die Threads damit parallel zueinander arbeiten und von mehreren Prozessoren gleichzeitig ausgeführt werden. Das Pipeliningmodell erfordert ein koordiniertes Zusammenarbeiten der beteiligten Threads: Thread T_i kann die Berechnung

der zugehörigen Stufe nur dann durchführen, wenn Thread T_{i-1} Daten zur Verfügung gestellt hat. Außerdem kann T_i die errechneten Daten erst dann an den Thread T_{i+1} der Nachfolgestufe weitergeben, wenn T_{i+1} die Berechnung der zuvor weitergeleiteten Daten abgeschlossen hat, da sonst Daten überschrieben werden können.

```
typedef struct _stage_t { // Pipelinestufe
    pthread_mutex_t m;
    pthread_cond_t avail; // Eingabedaten für diese Stufe verfügbar?
    pthread_cond_t ready; // Stufe fuer neue Daten bereit?
    int data_ready; // != 0, wenn andere Daten bearbeitet werden
    long data; // Daten
    pthread_t thread; // Thread-ID
    struct _stage_t *next;
} stage_t;
typedef struct _pipe_t { // Pipeline
    pthread_mutex_t m;
    stage_t *head, *tail; // erste bzw. letzte Stufe der Pipeline
    int stages; // Anzahl der Stufen der Pipeline
    int active; // Anzahl der aktiven Datenelemente
} pipe_t;
const N_STAGES = 10;
```

Abb. 6.12. Datenstrukturen für die Realisierung einer Pipelineverarbeitung mit Pthreads.

Die Koordination der Threads der Pipelinestufen kann mit Hilfe von Bedingungsvariablen erfolgen. Dies wird im Folgenden an einem einfachen Beispiel demonstriert, in dem eine Folge von Integerwerten schrittweise durch jede Pipelinestufe inkrementiert wird, vgl. [19], d.h. die Pipelinestufen führen jeweils die gleichen Berechnungen aus. Der im Beispiel verwendete Koordinationsmechanismus kann aber auch für den Fall verwendet werden, dass unterschiedliche Pipelinestufen unterschiedliche Berechnungen ausführen.

Für jede Stufe der Pipeline wird eine Datenstruktur vom Typ `stage_t` angelegt, siehe Abbildung 6.12. Diese enthält eine Mutexvariable `m` zur Synchronisation des Zugriffs auf die Stufe und zwei Bedingungsvariablen `avail` und `ready`, die der Synchronisation der Threads aufeinanderfolgender Stufen dienen. Die Bedingungsvariable `avail` wird genutzt, um einem Thread per Signal mitzuteilen, dass Daten für seine Pipelinestufe verfügbar sind und er mit deren Verarbeitung beginnen kann. Ein Thread blockiert bezüglich dieser Bedingungsvariable, falls noch keine Daten seines Vorgängers vorliegen. Die Bedingungsvariable `ready` wird verwendet, um dem Thread der Vorgängerstufe per Signal mitzuteilen, dass er die von ihm errechneten Daten an seine Nachfolgestufe weitergeben kann. Bezüglich dieser Bedingungsvariablen wird der Thread der Vorgängerstufe blockiert, wenn er die von ihm errechneten Daten nicht direkt an die Nachfolgestufe übergeben kann. Am Eintrag

`data_ready` sieht ein Thread, ob Daten für seine Pipelinestufe verfügbar sind (Wert 1) oder nicht (Wert 0). Der Eintrag `data` enthält die zu verarbeitenden Daten, die im Fall dieses einfachen Beispiels aus einem einzelnen Integerwert bestehen. Der Eintrag `thread` bezeichnet den der Pipelinestufe zugehörigen Thread, und `next` ist ein Verweis auf die nächste Pipelinestufe.

Die gesamte Pipeline wird durch die Datenstruktur `pipe_t` repräsentiert. Diese enthält neben einer Mutexvariablen `m` die beiden Zeiger `head` und `tail` auf die erste bzw. letzte Stufe der Pipeline. Die letzte Stufe der Pipeline dient zur Ablage des Endresultats der jeweiligen Berechnung. Auf dieser wird keine Berechnung mehr ausgeführt und daher gibt es keinen assoziierten, aktiven Thread.

```
int pipe_send(stage_t *nstage, long data) {
    //Parameter: Nachfolgestufe und zu übergebende Daten
    pthread_mutex_lock(&nstage->m);
    {
        while (nstage->data_ready)
            pthread_cond_wait(&nstage->ready, &nstage->m);
        nstage->data = data;
        nstage->data_ready = 1;
        pthread_cond_signal(&nstage->avail);
    }
    pthread_mutex_unlock(&nstage->m);
}

void *pipe_stage(void *arg) {
    stage_t *stage = (stage_t*)arg;
    long result_data;
    pthread_mutex_lock(&stage->m);
    {
        for ( ; ; ) {
            while (!stage->data_ready) { // Warten auf Daten
                pthread_cond_wait(&stage->avail, &stage->m);
            }
            // Verarbeiten der Daten und Weiterleiten an die Folgestufe:
            result_data = stage->data + 1; // Berechnung des Resultats
            pipe_send(stage->next, result_data);
            stage->data_ready = 0; // Verarbeitung abgeschlossen
            pthread_cond_signal(&stage->ready);
        }
    }
    pthread_mutex_unlock(&stage->m); //dieses unlock wird nie erreicht
}
```

Abb. 6.13. Weitergabe von Daten an eine Pipelinestufe und Arbeitsweise der Threads zu den Pipelinestufen.

Zur Übergabe von Daten an eine Pipelinestufe dient die Funktion `pipe_send()`, die in Abb. 6.13 abgebildet ist. Diese Funktion wird vom Thread einer Stu-

fe verwendet, um Daten an die Nachfolgestufe, die als Parameter `nstage` spezifiziert ist, zu übergeben. Zur Übergabe der Daten wird zuerst die Mutexvariable `m` der Nachfolgestufe gesperrt. Daten können aber nur dann an die Nachfolgestufe übergeben werden, wenn diese frei ist und zurzeit keine anderen Daten verarbeitet, d.h. wenn `data_ready = 0` ist. Ist dies nicht der Fall, wartet der Thread der Vorgängerstufe bezüglich der Bedingungsvariablen `ready` auf das Eintreten dieser Bedingung. Ist die Nachfolgestufe frei, übergibt der Thread der Stufe die Daten und weckt den evtl. auf Daten wartenden Thread der Nachfolgestufe auf, indem er die Bedingungsvariable `avail` verwendet.

```
int pipe_create(pipe_t *pipe, int stages) { // Erzeugen der Pipeline
    int pi;
    stage_t **link = &pipe->head;
    stage_t *new_stage, *stage;
    pthread_mutex_init(&pipe->m, NULL);
    pipe->stages = stages;
    pipe->active = 0;
    // erzeuge stages+1 Stufen, letzte Stufe für Ergebnis
    for (pi = 0; pi <= stages; pi++) {
        new_stage = (stage_t*)malloc(sizeof(stage_t));
        pthread_mutex_init(&new_stage->m, NULL);
        pthread_cond_init(&new_stage->avail, NULL);
        pthread_cond_init(&new_stage->ready, NULL);
        new_stage->data_ready = 0;
        link = new_stage; // Herstellen der Listenverknüpfung
        link = &new_stage->next;
    }
    link = (stage_t*)NULL;
    pipe->tail = new_stage;
    // Erzeugung eines Threads fuer jede Stufe ausser der letzten
    for (stage = pipe->head; stage->next; stage = stage->next) {
        pthread_create(&stage->thread, NULL, pipe_stage, (void*)stage);
    }
}

int pipe_start(pipe_t *pipe, long v) { // Start Pipelineverarbeitung
    pthread_mutex_lock(&pipe->m);
    {
        pipe->active++;
    }
    pthread_mutex_unlock(&pipe->m);
    pipe_send(pipe->head, v);
}
```

Abb. 6.14. Pthreads-Funktion zur Erzeugung und zum Start der Pipeline.

Jeder an der Pipelineverarbeitung beteiligte Thread führt die ebenfalls in Abbildung 6.13 abgebildete Funktion `pipe_stage()` aus. Diese Funktion

erhält als Argument einen Zeiger auf die Datenstruktur der dem Thread zugeordneten Pipelinestufe. Jeder Thread führt eine Endlosschleife aus, in der er mit Hilfe der Bedingungsvariablen `avail` auf das Eintreffen von zu verarbeitenden Daten wartet. Sobald Daten eingetroffen sind, verarbeitet er diese und gibt das berechnete Resultat mit Hilfe von `pipe_send()` an die nächste Pipelinestufe `stage.next` weiter. Anschließend weckt er den eventuell wartenden Thread der Vorgängerstufe per Signal auf und teilt ihm dadurch mit, dass er wieder Daten zur Verarbeitung weitergeben kann. Die Synchronisation jeweils aufeinanderfolgender Threads erfolgt also dadurch, dass jeder Thread die Bedingungsvariablen `avail` und `ready` seiner zugeordneten Stufe und seiner Nachfolgestufe nutzt. Für die Bedingung wird der Eintrag `data_ready` genutzt, der abwechselnd von einer Thread auf 0 gesetzt wird, falls er weitere Daten entgegennehmen kann, und von seinem Nachfolgethread auf 1 gesetzt wird, falls neue Daten weitergegeben werden. In dem hier dargestellten Beispiel führt jeder der Threads identische Berechnungen aus, d.h. jeder Thread ruft die gleiche Funktion `pipe_stage()` auf. Für komplexere Anwendungen können die Threads auch unterschiedliche Funktionen ausführen, so dass jede Pipelinestufe unterschiedliche Berechnungen durchführt.

Das Erzeugen einer Pipeline mit einer vorgegebenen Anzahl von Stufen erfolgt mit der Funktion `pipe_create()`, vgl. Abbildung 6.14. Diese Funktion erzeugt und initialisiert die Datenstrukturen zur Repräsentation der einzelnen Stufen und startet dann für jede Stufe außer der letzten den zugehörigen Thread, d.h. `pipe_create()` erzeugt `stages+1` Pipelinestufen, für die letzte Pipelinestufe wird aber kein Thread gestartet. Jeder so gestartete Thread führt wie beschrieben die Funktion `pipe_stage()` aus. Die Übergabe eines Wertes an die erste Stufe der Pipeline erfolgt mit Hilfe der Funktion `pipe_start()`, vgl. Abbildung 6.14. Diese gibt den Argumentwert mit Hilfe der Funktion `pipe_send()` an die erste Stufe der Pipeline weiter. Die Funktion wartet nicht auf das Resultat und kehrt stattdessen direkt zum Aufrufer zurück, d.h. die Verarbeitung der Pipeline erfolgt asynchron zu dem Thread, der Werte an die Pipeline zur Verarbeitung weitergibt. Die Synchronisation zwischen diesem Thread und dem Thread zur ersten Pipelinestufe erfolgt innerhalb von `pipe_send()`. Das Entnehmen der errechneten Resultatwerte aus der letzten Stufe der Pipeline erfolgt mit Hilfe der Funktion `pipe_result()`, siehe Abbildung 6.15. Ist zum Zeitpunkt des Aufrufs dieser Funktion kein Wert in der Pipeline vorhanden, wird die Kontrolle mit Rückgabewert 0 direkt an den Aufrufer zurückgegeben. Enthält die Pipeline mindestens einen Wert, was durch `pipe->active > 0` festgestellt wird, wartet der ausführende Thread mit Hilfe der Bedingungsvariablen `avail` der letzten Pipelinestufe darauf, dass der nächste Resultatwert in der letzten Pipelinestufe ankommt. Dies geschieht dadurch, dass der Thread der vorletzten Pipelinestufe den von ihm errechneten Wert mit Hilfe von `pipe_send()` an die letzte Pipelinestufe weitergibt, vgl. `pipe_stage()` in Abbildung 6.13. Dadurch weckt der Thread der vorletzten Pipelinestufe dann den eventuell bezüglich der Bedingungsvari-

```

int pipe_result(pipe_t *pipe, long *result) {
    stage_t *tail = pipe->tail;
    int empty = 0;
    pthread_mutex_lock(&pipe->m);
    {
        if (pipe->active <= 0) empty = 1; // leere Pipeline
        else pipe->active--; // Entnahme eines Wertes
    }
    pthread_mutex_unlock(&pipe->m);
    if (empty) return 0;
    pthread_mutex_lock(&tail->m);
    {
        while (!tail->data_ready) // Warten auf Daten
            pthread_cond_wait(&tail->avail, &tail->m);
        result = tail->data;
        tail->data_ready = 0;
        pthread_cond_signal(&tail->ready);
    }
    pthread_mutex_unlock(&tail->m);
    return 1;
}

int main(int argc, char *argv[]) {
    pipe_t pipe;
    long value, result;
    char line[128];
    pipe_create(&pipe, N_STAGES);
    // Programmabbruch nur bei Einlesefehler oder EOF
    while (fgets(line, sizeof(line), stdin)) {
        if (!*line)
            continue; // leere Eingabe ignorieren
        if (!strcmp(line, "=")) {
            if (pipe_result(&pipe, &result))
                printf("%d\n", result);
            else printf("ERROR: Pipe empty\n");
        }
        else {
            if (sscanf(line, "%ld", &value) < 1)
                printf("ERROR: Not an int\n");
            else pipe_start(&pipe, value);
        }
    }
    return 0;
}

```

Abb. 6.15. Hauptprogramm und Pthreads-Funktion zur Entnahme eines Wertes aus der Pipeline.

riablen `avail` der letzten Pipelinestufe wartenden Thread auf. Dieser eventuell wartende Thread ist der Thread, der mit Hilfe von `pipe_result()` einen Wert aus der Pipeline entnehmen will und auf den nächsten Wert wartet.

Das Hauptprogramm des Beispiels in Abbildung 6.15 erzeugt zuerst mit `pipe_create()` eine Pipeline mit einer festen Anzahl von Stufen. Dann liest das Programm Zeilen mit Zahlen ein und gibt diese an die Pipeline mit Hilfe von `pipe_start()` weiter. Die Synchronisation des Haupt-Threads mit dem Thread der ersten Pipelinestufe erfolgt dabei in `pipe_send()`, das von `pipe_start()` aufgerufen wird. Der Haupt-Thread wartet dabei bezüglich der Bedingungsvariablen `ready` der ersten Pipelinestufe eventuell darauf, dass er den eingelesenen Wert an die erste Pipelinestufe weitergeben kann. Die Eingabe von `' = '` in einer einzelnen Zeile führt dazu, dass der Haupt-Thread mit `pipe_result()` das nächste Resultat aus der Pipeline zu entnehmen versucht, vgl. oben, und dieses, falls vorhanden, ausgibt.

Die Synchronisation zwischen den Pipeline-Threads untereinander und zwischen dem Hauptthread und dem ersten bzw. letzten Pipeline-Thread ist in Abbildung 6.16 für eine Pipeline mit drei Stufen und zwei Threads veranschaulicht. Für jede Stufe sind die wesentlichen Einträge der Datenstruktur vom Typ `stage_t` dargestellt. Die Reihenfolge der von den Pipeline-Threads durchgeführten Zugriffs- und Synchronisationoperationen ergibt sich aus den Anweisungen in `pipe_stage()` und wird durch Nummern veranschaulicht. Die Zugriffs- und Synchronisationoperationen des Hauptthreads ergeben sich aus `pipe_start()` und `pipe_result()`.

6.2.5 Realisierung eines Client-Server-Modells

Bei einer Client-Server-Organisation wird zwischen Client-Threads und Server-Threads unterschieden. Typischerweise gibt es mehrere Server-Threads und mehrere Client-Threads. Die Server-Threads erledigen Anfragen, die von Client-Threads gestellt werden. Die Client-Threads dienen oft als Schnittstelle zu einem Benutzer. Während der Erledigung einer Anfrage durch einen Server-Thread kann der anfragende Client-Thread entweder warten oder kann andere Aufgaben durchführen und das Ergebnis später beim Server-Thread abholen. Im Folgenden wird die Realisierung eines Client-Server-Modells an einem einfachen Beispiel demonstriert.

Beispiel: Mehrere Threads sollen Zeilen von der Standardeingabe lesen und Zeilen auf die Standardausgabe ausgeben. Das Einlesen erfolgt mit Hilfe eines Prompts, der z.B. angibt, welche Eingabe erwartet wird. Der Einsatz von Server-Threads soll sicherstellen, dass die Ausgabe eines Prompts und das Einlesen der zugehörigen Eingabe synchronisiert erfolgt, so dass zwischen Ausgabe eines Prompts und dem Einlesen der zugehörigen Eingabe keine Ausgabe eines anderen Threads erfolgt. Client-Threads geben Anfragen zur Ausgabe eines Prompts oder zum Einlesen einer Eingabe an die Server-Threads weiter. Die Server-Threads werden durch ein spezielles

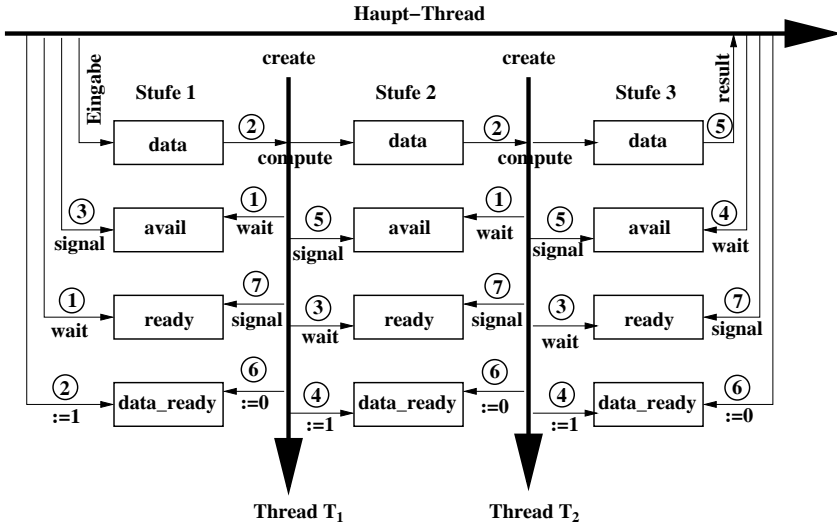


Abb. 6.16. Veranschaulichung der Synchronisation der an der Pipelineverarbeitung beteiligten Threads für den Fall einer Pipeline mit zwei Threads und drei Stufen aus der Sicht der verwendeten Datenstrukturen. Die Nummern beschreiben die Reihenfolge der Ausführung der Synchronisationsschritte, die jeder beteiligte Thread entsprechend der von ihm ausgeführten Pthreads-Funktionen ausführt.

QUIT-Kommando beendet. Abbildung 6.17 enthält die für die Realisierung in Pthreads erforderlichen Datenstrukturen. Die Datenstruktur `_request_t` wird für die Anfragen der Clients zu den Servern verwendet. Der Eintrag `op` spezifiziert die angefragte Operation. Der Eintrag `synchronous` gibt an, ob der Client auf die Beendigung der Anfrage durch den Server wartet (Wert 1) oder nicht (Wert 0). Die Bedingungsvariable `done` dient für den Fall des Wartens zur Synchronisation zwischen Client und Server, d.h. der Client-Thread wartet bezüglich dieser Bedingungsvariable darauf, dass der Server die Anfrage durchgeführt hat. Die Felder `prompt` und `text` dienen zur Ablage eines einzugebenden Prompts bzw. eines vom Server einzulesenden Textes. Die Datenstruktur `_tty_server_t` dient zur Ablage der an einen Server gerichteten Anfragen (Einträge `first` und `last`) in einer FiFo-Liste und enthält die Bedingungsvariable `request`, bezüglich der der Server-Thread blockiert, wenn keine Anfragen zur Verarbeitung vorliegen. Der Eintrag `running` zeigt an, ob der zugehörige Server bereits gestartet wurde (Wert 1) oder nicht (Wert 0). Das im Folgenden skizzierte Programm arbeitet mit einem Server, kann im Prinzip aber auf eine beliebige Anzahl von Servern erweitert werden.

Der Server führt die Funktion `tty_server_routine()` aus, vgl. Abbildung 6.18. So lange keine Anfragen zur Verarbeitung vorliegen, blockiert der Server bezüglich der Bedingungsvariablen `request`. Liegen Anfragen zur Verarbeitung vor, entnimmt der Server die erste Anfrage aus der Liste von Anfragen

```

#define REQ_READ 1
#define REQ_WRITE 2
#define REQ_QUIT 3
#define PROMPT_SIZE 32
#define TEXT_SIZE 128
typedef struct _request_t {
    struct _request_t *next; // verkettete Liste
    int op;
    int synchronous; // 1 gdw. Client wartet auf Server
    int done_flag;
    pthread_cond_t done;
    char prompt[PROMPT_SIZE], text[TEXT_SIZE];
} request_t;
typedef struct _tty_server_t { // Datenstruktur für Server-Kontext
    request_t *first, *last;
    int running; // != 0, wenn der Server läuft
    pthread_mutex_t m;
    pthread_cond_t request;
} tty_server_t;
#define TTY_SERVER_INITIALIZER { NULL, NULL, 0,
    PTHREAD_MUTEX_INITIALIZER,
    PTHREAD_COND_INITIALIZER }
tty_server_t tty_server = TTY_SERVER_INITIALIZER;
int client_threads;
pthread_mutex_t client_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t client_done = PTHREAD_COND_INITIALIZER;
pthread_t server_thread;

```

Abb. 6.17. Datenstrukturen zur Realisierung eines Client-Server-Modells mit Pthreads.

und bearbeitet diese entsprechend der in der Anfrage spezifizierten Operation (`REQ_READ`, `REQ_WRITE` oder `REQ_QUIT`). Für die Operation `REQ_READ` wird der in der Anfrage angegebene Prompt ausgegeben und eine Zeile wird in `text` eingelesen. Für die Operation `REQ_WRITE` wird die in `text` abgelegte Zeile ausgegeben. Die Operation `REQ_QUIT` beendet den Server. Wartet ein Client auf die Erledigung der Anfrage (Eintrag `synchronous`), so ist dieser bis zur Erledigung der Anfrage bezüglich der Bedingungsvariablen `done` blockiert. In diesem Fall weckt der Server-Thread nach Erledigung der Anfrage den blockierten Client-Thread mit `pthread_cond_signal()` wieder auf.

Zur Weitergabe einer Anfrage an den Server verwendet ein Client die Funktion `tty_server_request()`, vgl. Abbildung 6.19. Sollte der Server-Thread noch nicht erzeugt sein, wird er in `tty_server_request` gestartet. Die Funktion legt eine Anfrage-Datenstruktur vom Typ `request_t` und trägt die Angaben zur Anfrage ein. Die Anfrage-Datenstruktur wird in die Liste der Anfragen des Servers eingetragen und der eventuell auf Anfragen wartende Server wird mit `pthread_cond_signal()` aufgeweckt. Soll der Client auf die Beendigung der Bearbeitung der Anfrage durch den Server warten,

```

void *tty_server_routine(void *arg) {
    static pthread_mutex_t prompt_mutex = PTHREAD_MUTEX_INITIALIZER;
    request_t *request;
    int op, len;
    for (;;) {
        pthread_mutex_lock(&tty_server.m);
        {
            while (tty_server.first == NULL)
                pthread_cond_wait(&tty_server.request, &tty_server.m);
            request = tty_server.first;
            tty_server.first = request->next;
            if (tty_server.first == NULL)
                tty_server.last = NULL;
        }
        pthread_mutex_unlock(&tty_server.m);
        switch (request->op) {
            case REQ_READ:
                puts(request->prompt);
                if (fgets(request->text, TEXT_SIZE, stdin) == NULL)
                    request->text[0] = '\0';
                len = strlen(request->text);
                if (len > 0 && request->text[len - 1] == '\n')
                    request->text[len - 1] = '\0';
                break;
            case REQ_WRITE:
                puts(request->text); break;
            default: // auch REQ_QUIT
                break;
        }
        if (request->synchronous) {
            pthread_mutex_lock(&tty_server.m);
            request->done_flag = 1;
            pthread_cond_signal(&request->done);
            pthread_mutex_unlock(&tty_server.m);
        }
        if (request->op == REQ_QUIT) break;
    }
    return NULL;
}

```

Abb. 6.18. Realisierung des Server-Threads zur Verarbeitung von Anfragen mit Pthreads.


```

void tty_server_request(int op, int sync, char *prompt, char *string) {
    request_t *request;
    pthread_mutex_lock(&tty_server.m);
    {
        if (!tty_server.running) {
            pthread_create(&server_thread, NULL, tty_server_routine, NULL);
            tty_server.running = 1;
        }
        request = (request_t *) malloc(sizeof(request_t));
        request->op = op;
        request->synchronous = sync;
        request->next = NULL;
        if (sync) {
            request->done_flag = 0;
            pthread_cond_init(&request->done, NULL);
        }
        if (prompt != NULL) {
            strncpy(request->prompt, prompt, PROMPT_SIZE);
            request->prompt[PROMPT_SIZE - 1] = '\0';
        }
        else request->prompt[0] = '\0';
        if (op == REQ_WRITE && string != NULL) {
            strncpy(request->text, string, TEXT_SIZE);
            request->text[TEXT_SIZE - 1] = '\0';
        }
        else request->text[0] = '\0';
        if (tty_server.first == NULL)
            tty_server.first = tty_server.last = request;
        else {
            tty_server.last->next = request;
            tty_server.last = request;
        }
        pthread_cond_signal(&tty_server.request);
        if (sync) {
            while (!request->done_flag)
                pthread_cond_wait(&request->done, &tty_server.m);
            if (op == REQ_READ)
                strcpy(string, request->text);
            pthread_cond_destroy(&request->done);
            free(request);
        }
    }
    pthread_mutex_unlock(&tty_server.m);
}

```

Abb. 6.19. Weitergabe einer Anfrage an den Server-Thread mit Pthreads.

wird er bezüglich der Bedingungsvariablen `done` der Anfrage-Datenstruktur blockiert, bis der Server ihn nach Erledigung der Anfrage aufweckt. Die Clients führen die Funktion `client_routine()` aus, vgl. Abbildung 6.20. Jeder Client schickt Lese- und Schreib Anfragen mit `tty_server_request()` an den Server, bis der Benutzer den Client beendet. Bei Beendigung des letzten Clients wird der bezüglich der Bedingungsvariablen `client_done` blockierte Haupt-Thread wieder aufgeweckt. Der Haupt-Thread erzeugt die Client-Threads und wartet dann, bis alle Client-Threads wieder terminiert sind. Der Server-Thread wird nicht vom Haupt-Thread, sondern von dem Client-Thread, der die erste Anfrage mit `tty_server_routine()` an den Server weitergibt, gestartet. Nach Terminierung aller Clients wird der Server-Thread vom Haupt-Thread durch Angabe der Anfrage `REQ_QUIT` terminiert. $\square$

6.2.6 Steuerung und Abbruch von Threads

Die Charakteristika eines mit `pthread_create()` erzeugten Threads können über die mitgegebene Attribut-Datenstruktur vom Typ `pthread_attr_t` beeinflusst werden, vgl. Abschnitt 6.2.1. Wird `pthread_create()` mit dem Wert `NULL` als zweitem Parameter aufgerufen, so wird ein Thread mit den Default-Attributen erzeugt, die jedoch für verschiedene Pthreads-Bibliotheken unterschiedlich sein können. Sollen andere Charakteristika gewählt werden, muss *vor* dem Aufruf von `pthread_create()` eine entsprechend besetzte Attributdatenstruktur erzeugt werden. Dazu wird diese zuerst durch einen Aufruf von

```
int pthread_attr_init (pthread_attr_t *attr)
```

erzeugt, wodurch zunächst eine Initialisierung mit den Defaultattributen erfolgt. Zur Änderung der Attributwerte, steht für jedes Attribut eine Funktion zur Abfrage und zum Setzen des Defaultwertes zur Verfügung, wobei aber nicht bei allen Bibliotheken jeder Defaultwert geändert werden kann.

Rückgabewert. Ein wichtiges Attribut, das von allen Pthreads-Bibliotheken unterstützt wird, ist `detachstate` und betrifft das Verhalten des Laufzeitsystems nach Beendigung des Threads. Per Default wird davon ausgegangen, dass der Rückgabewert eines Threads nach seiner Beendigung noch gebraucht wird. Daher wird die für die Verwaltung des Threads angelegte interne Datenstruktur auch nach seiner Beendigung so lange vom Laufzeitsystem aufgehoben, bis ein anderer Thread eine `pthread_join()`-Operation für diesen Thread ausführt, vgl. Abschnitt 6.2.1. Wenn der Programmierer bei der Erzeugung des Threads weiß, dass dessen Rückgabewert nicht mehr gebraucht wird, kann er den Thread so erzeugen, dass die Datenstruktur nach seiner Beendigung direkt freigegeben wird und kein anderer Thread den Rückgabewert abfragen kann. Für das Abfragen bzw. Setzen des zugehörigen Attributs stehen die beiden Funktionen

```

void *client_routine(void *arg) {
    int my_nr = *(int*)arg, loops;
    char prompt[PROMPT_SIZE], string[TEXT_SIZE];
    char format[TEXT_SIZE + 64];
    sprintf(prompt, "Client %d>", my_nr);
    for (;;) {
        tty_server_request(REQ_READ, 1, prompt, string);
        // synchronisierte Eingabe
        if (string[0] == '\0') break; // Programmausstieg
        for (loops = 0; loops < 4; loops++) {
            sprintf(format, "(%d # %d)%s", my_nr, loops, string);
            tty_server_request(REQ_WRITE, 0, NULL, format);
            sleep(1);
        }
    }
    pthread_mutex_lock(&client_mutex);
    client_threads--;
    if (client_threads == 0) pthread_cond_signal(&client_done);
    pthread_mutex_unlock(&client_mutex);
    return NULL;
}

#define N_THREADS 4
int main(int argc, char *argv[]) {
    pthread_t thread;
    int i;
    int args[N_THREADS];
    client_threads = N_THREADS;
    pthread_mutex_lock(&client_mutex);
    {
        for (i = 0; i < N_THREADS; i++) {
            args[i] = i;
            pthread_create(&thread, NULL, client_routine, &args[i]);
        }
        while (client_threads > 0)
            pthread_cond_wait(&client_done, &client_mutex);
    }
    pthread_mutex_unlock(&client_mutex);
    printf("All clients done\n");
    tty_server_request(REQ_QUIT, 1, NULL, NULL);
    return 0;
}

```

Abb. 6.20. Client-Thread und Haupt-Thread für die Realisierung eines Client-Server-Modells mit Pthreads.

```
int pthread_attr_getdetachstate (const pthread_attr_t *attr,
                                int *detachstate)
int pthread_attr_setdetachstate (pthread_attr_t *attr,
                                int detachstate)
```

zur Verfügung. Dabei bedeutet `detachstate = PTHREAD_CREATE_JOINABLE`, dass der Rückgabewert des Threads abgefragt werden kann, `detachstate = PTHREAD_CREATE_DETACHED` bedeutet, dass *kein* Rückgabewert zurückgegeben werden kann.

Stack-Eigenschaften. Je nach Thread-Bibliothek können weitere Thread-Attribute gesetzt werden, die den lokalen Stack eines Threads betreffen, in dem der Thread lokale Variablen von noch nicht vollständig bearbeiteten Funktionen speichert. (Threads eines Prozesses haben einen gemeinsamen Programmspeicher, Datenspeicher und Heap, jedoch einen privaten Laufzeitstack.) Der benötigte Platz für den lokalen Laufzeitstack wird von der Größe der lokalen Variablen und der Anzahl abzuarbeitender Funktionen beeinflusst. Reicht der vorgesehene Platz nicht aus, so kann die Stackgröße erhöht werden. Wenn in der Headerdatei `<unistd.h>` die symbolische Konstante

`_POSIX_THREAD_ATTR_STACKSIZE`

definiert ist, was mit Hilfe von

```
#ifdef _POSIX_THREAD_ATTR_STACKSIZE oder
if (sysconf (_SC_THREAD_ATTR_STACKSIZE) == -1)
```

im Programm abgefragt werden kann, kann die Größe des privaten Stacks eines zu erzeugenden Threads durch Aufruf der Funktionen

```
int pthread_attr_getstacksize (const pthread_attr_t *attr,
                               size_t *stacksize)
int pthread_attr_setstacksize (pthread_attr_t *attr,
                               size_t stacksize)
```

abgefragt bzw. gesetzt werden. Dabei ist `size_t` ein in `<unistd.h>` definierter Datentyp, der meist als `unsigned int` definiert ist, und der die Stackgröße in Bytes angibt. Der Wert von `stacksize` sollte mindestens `PTHREAD_STACK_MIN` Bytes betragen. Wenn in `<unistd.h>` die symbolische Konstante

`_POSIX_THREAD_ATTR_STACKADDR`

definiert ist, kann auch die Startadresse des lokalen Stacks eines Threads durch Aufruf der Funktionen

```
int pthread_attr_getstackaddr (const pthread_attr_t *attr,
                               size_t **stackaddr)
int pthread_attr_setstackaddr (pthread_attr_t *attr,
                               size_t *stackaddr)
```

abgefragt bzw. neu gesetzt werden. Beide Möglichkeiten sollten mit Vorsicht benutzt werden, da sie nicht von allen Pthread-Bibliotheken unterstützt werden und zu nichtportablen Programmen führen.

Nach dem Setzen von Thread-Attributen mit Hilfe der beschriebenen Funktionen kann ein Thread mit den gewünschten Charakteristika dadurch erzeugt werden, dass die erzeugte Attributdatenstruktur als Parameter von `pthread_create()` angegeben wird. Die Charakteristika der dadurch erzeugten Threads werden durch die Besetzung der Attributdatenstruktur zum Zeitpunkt des Aufrufs von `pthread_create()` bestimmt, d.h. eine nachträgliche Veränderung der Attributdatenstruktur beeinflusst die Charakteristika des Threads nicht mehr.

Abbruch von Threads. In manchen Situationen ist es sinnvoll, die Abarbeitung eines Threads von außen zu stoppen, wenn etwa das Ergebnis der von diesem durchgeführten Aufgabe nicht mehr benötigt wird. So können in einer Anwendung, die mehrere Threads zum Durchsuchen einer Datenstruktur einsetzt, alle Threads abgebrochen werden, sobald einer von ihnen den gesuchten Eintrag gefunden hat.

Ein Thread kann einen anderen Thread des gleichen Prozesses durch Aufruf der Funktion

```
int pthread_cancel (pthread_t thread)
```

terminieren, wobei `thread` den zu terminierenden Thread bezeichnet. Der Aufruf dieser Funktion startet den Abbruch des angegebenen Zielthreads `thread` nur, die Kontrolle kehrt aber zum aufrufenden Thread zurück, ohne dass der Zielthread bereits terminiert sein muss. Per Default wird der Zielthread einer `pthread_cancel()`-Operation nicht direkt terminiert, sondern erst, wenn er einen **Abbruchpunkt** (engl. *cancellation point*) erreicht (verzögerter Abbruch). Der Pthreads-Standard legt obligatorische und optionale Abbruchpunkte fest. Alle obligatorischen Abbruchpunkte entsprechen Aufrufen von Funktionen, deren Ausführung zum Blockieren des aufrufenden Threads führen können. Dies sind z.B. `pthread_cond_wait`, `pthread_cond_timedwait`, `open()`, `read()`, `wait()` oder `pthread_join()`, eine vollständige Liste ist z.B. in [92] angegeben. Erreicht der Zielthread eine als Abbruchpunkt festgelegte Operation und blockiert er durch Ausführung der Operation, so wird geprüft, ob eine Abbruchanfrage vorliegt. Liegt eine Abbruchanfrage vor, wird der Zielthread abgebrochen. Der Programmierer kann zusätzliche Abbruchpunkte durch Aufruf der Funktion

```
void pthread_testcancel()
```

definieren. Bei Ausführen dieser Funktion überprüft der ausführende Thread, ob ein anderer Thread eine Abbruchanfrage für ihn gestellt hat. Ist dies der Fall, wird der Thread terminiert. Andernfalls hat der Aufruf der Funktion keinen Effekt.

Ein Thread kann verhindern, dass er von einem anderen Thread abgebrochen werden kann, indem er die Funktion

```
int pthread_setcancelstate (int state, int *oldstate)
```

mit `state = PTHREAD_CANCEL_DISABLE` aufruft. Der vorhergehende Abbruchzustand wird in `*oldstate` abgelegt.

Danach hat der Aufruf einer als Abbruchbedingung festgelegten Operation oder das Ausführen der Funktion `pthread_testcancel()` auch bei ausstehenden Abbruchanfragen keinen Effekt. Ein Abbruch kann durch einen Aufruf von `pthread_setcancelstate()` mit `state = PTHREAD_CANCEL_ENABLE` wieder ermöglicht werden. Erlaubt ein Thread einen Abbruch von außen, kann er durch Aufruf der Funktion

```
int pthread_setcanceltype (int type, int *oldtype)
```

mit `type = PTHREAD_CANCEL_ASYNCHRONOUS` erreichen, dass anstatt einem verzögerten Abbruch, der nur an Abbruchpunkten stattfindet, ein *direkter* Abbruch stattfindet, der direkt nach Eintreffen der Abbruchanfrage durchgeführt wird. Man beachte aber, dass dies gefährlich sein kann, da der zu terminierende Thread beim Eintreffen der Abbruchanfrage gerade eine Manipulation in einem kritischen Bereich durchführen könnte, so dass ein sofortiger Abbruch den zugehörigen Prozess in einem inkonsistenten Zustand hinterlässt, der zu Fehlern bei anderen Threads führen kann. Durch einen Aufruf von `pthread_setcanceltype()` mit `type = PTHREAD_CANCEL_DEFERRED` kann wieder zum verzögerten Abbruch zurückgeschaltet werden.

Cleanup-Stacks. Mit jedem Thread ist ein *Stack von Funktionsaufrufen* (engl. *cleanup stack*) assoziiert, auf den Funktionsaufrufe geschoben werden können, die bei der Terminierung des Threads auszuführen sind. Diese Funktionsaufrufe können benutzt werden, um beim Abbruch eines Threads durch einen anderen Thread vor dem Abbruch einen konsistenten Zustand herzustellen. Dies ist z.B. dann notwendig, wenn zwischen dem Besetzen und der Freigabe einer Mutexvariablen ein Abbruchpunkt liegt. Findet dort ein Abbruch statt, bleibt die Mutexvariable gesperrt, und ein anderer Thread, der auf die Freigabe der Mutexvariable wartet, wird unendlich lange blockiert. Zur Vermeidung dieses Verhaltens kann vor der Besetzung der Mutexvariable ein Funktionsaufruf zur Freigabe der Mutexvariable auf dem Stack abgelegt werden, der dann beim Abbruch des Threads ausgeführt wird. Der zugehörige Funktionsaufruf wird auch als **Cleanup-Handler** bezeichnet. Die Ablage eines Cleanup-Handlers auf dem Cleanup-Stack wird durch Aufruf der Funktion

```
void pthread_cleanup_push (void (*routine) (void *), void *arg)
```

erreicht. Dabei ist `routine` ein Zeiger auf den auszuführenden Cleanup-Handler, und `arg` liefert die aktuellen Parameterwerte für den Cleanup-Handler. Die auf dem Cleanup-Stack abgelegten Cleanup-Handler werden

in umgekehrter Reihenfolge ihrer Ablage, d.h. im LIFO-Prinzip (*last in, first out*), ausgeführt, wenn der Thread beendet oder abgebrochen wird. Ein auf dem Cleanup-Stack abgelegter Cleanup-Handler kann durch Aufruf der Funktion

```
void pthread_cleanup_pop (int execute)
```

wieder vom Stack entfernt werden. Für `execute=0` wird der oberste Cleanup-Handler vom Stack entfernt, ohne dass er ausgeführt wird. Für `execute≠0` wird der entnommene Cleanup-Handler auch ausgeführt. Die Funktionen `pthread_cleanup_push()` und `pthread_cleanup_pop()` sollten immer als Paar innerhalb des gleichen Blockes von Anweisungen benutzt werden, da diese Funktionen als Makros implementiert sein können.

Beispiel: Realisierung eines Semaphor-Mechanismus: Ein (zählender) *Semaphor* ist ein Datentyp, der einen Zähler realisiert, der nichtnegative Integerwerte annehmen kann und der mit zwei Operationen manipuliert werden kann: Eine *Signal*-Operation inkrementiert den Zähler und weckt einen bezüglich des Semaphors blockierten Thread auf, wenn es einen solchen gibt. Eine *Wait*-Operation blockiert bis der Wert des Zählers > 0 ist und dekrementiert den Zähler dann. Zählende Semaphore können zur Zuteilung von beschränkten Ressourcen eingesetzt werden, deren Anzahl durch die Initialisierung des Zählers gegeben ist. *Binäre Semaphore*, die nur die Werte 0 oder 1 annehmen können, können für die Sicherstellung des wechselseitigen Ausschlusses bei kritischen Bereichen verwendet werden.

Abbildung 6.21 zeigt als Beispiel die Verwendung von Cleanup-Handlern zur Realisierung eines Semaphor-Mechanismus mit Hilfe einer Bedingungsvariablen [127]. Die Funktion `AquireSemaphore()` wartet, bis der zugehörige Zähler einen Wert > 0 erhalten hat, um danach den Zähler zu dekrementieren. Die Funktion `ReleaseSemaphore()` inkrementiert den Wert des Zählers und weckt danach einen wartenden Thread mit Hilfe von `pthread_cond_signal()` auf. Die Manipulation der Semaphor-Datenstruktur wird in beiden Funktionen mit Hilfe eines Sperrmechanismus geschützt, um konkurrierende Manipulationen zu vermeiden. Zu Beginn wird jeweils `pthread_mutex_lock()` ausgeführt, am Ende wird durch `pthread_cleanup_pop(1)` die Ausführung der letzten auf dem Cleanup-Stack liegenden Funktion bewirkt, d.h. also von `pthread_mutex_unlock()`. Falls ein Thread bei Ausführung der Funktion `pthread_cond_wait(&(ps->cond), &(ps->mutex))` in `AquireSemaphore()` blockiert, wird die Mutexvariable `ps->mutex` wieder freigegeben. Beim Aufwecken des Threads wird die Mutexvariable automatisch wieder besetzt. Da ein Aufruf von `pthread_cond_wait()` einen Abbruchpunkt darstellt, kann ein Thread abgebrochen werden, während er bezüglich der angegebenen Bedingungsvariablen `ps->cond` wartet. In diesem Fall wird `ps->mutex` vor dem Abbruch wieder besetzt, so dass ein Cleanup-Handler für die Freigabe der Mutexvariable sorgen muss. Dies wird in der Abbildung durch die Funktion `CleanupHandler()` realisiert. □

```

typedef struct Sema {
    pthread_mutex_t mutex;
    pthread_cond_t cond;
    int count;
} SEMA;

void CleanupHandler (void *arg)
{ pthread_mutex_unlock ((pthread_mutex_t *) arg);}

void AquireSemaphore (SEMA *ps)
{
    pthread_mutex_lock (&(ps->mutex));
    pthread_cleanup_push (CleanupHandler, &(ps->mutex));
    while (ps->count == 0)
        pthread_cond_wait (&(ps->cond), &(ps->mutex));
    --ps->count;
    pthread_cleanup_pop (1);
}

void ReleaseSemaphore (SEMA *ps)
{
    pthread_mutex_lock (&(ps->mutex));
    pthread_cleanup_push (CleanupHandler, &(ps->mutex));
    ++ps->count;
    pthread_cond_signal (&(ps->cond));
    pthread_cleanup_pop (1);
}

```

Abb. 6.21. Verwendung von Cleanup-Handlern bei der Implementierung eines Semaphor-Mechanismus. Die Funktion `AquireSemaphore()` realisiert den Zugriff auf den Semaphor, wobei durch Verwendung von `pthread_cond_wait()` sichergestellt wird, dass der Zugriff erst dann stattfindet, wenn der Zählerwert `count` des Semaphors größer als Null ist. Die Funktion `ReleaseSemaphore()` realisiert die Freigabe des Semaphors.

Produzenten-Konsumenten-Threads. Die in Abbildung 6.21 dargestellten Funktionen können zur Synchronisation eines *Produzenten-Konsumenten*-Verhältnisses zwischen Threads verwendet werden, vgl. Abbildung 6.22. Der Produzent legt die von ihm erzeugten Einträge in einem Puffer vorgegebener Länge ab, der Konsument entnimmt Einträge aus dem Puffer und verarbeitet sie weiter. Der Produzent kann nur Einträge im Puffer ablegen, wenn dieser nicht voll ist, der Konsument kann nur Einträge entnehmen, wenn der Puffer nicht leer ist. Um dies zu kontrollieren, werden zwei Semaphore `full` und `empty` verwendet. `full` zählt die Anzahl der belegten Puffereinträge und wird beim Start mit 0 initialisiert, `empty` zählt die Anzahl der freien Puffereinträge und wird mit der Pufferlänge initialisiert. Der Puffer `buffer` wird im Beispiel als Feld der Länge 100 für einen beliebigen Typ `ENTRY` definiert und umfasst die beiden Semaphore `empty` und `full`.

So lange der Puffer nicht voll ist, erzeugt der Produzent Einträge und legt sie mit Hilfe von `produce_item()` im Puffer ab. Bei jeder Ablage wird `empty` durch den Aufruf von `AquireSemaphore()` dekrementiert, `full` wird durch den Aufruf von `ReleaseSemaphore()` inkrementiert. Wenn der Puffer voll ist, wird der Produzent beim Aufruf von `AquireSemaphore()` blockiert. So lange der Puffer nicht leer ist, entnimmt der Konsument mit Hilfe von `consume_item()` Einträge aus dem Puffer und verarbeitet sie weiter. Bei jeder Entnahme wird `full` durch den Aufruf von `AquireSemaphore()` dekrementiert, `empty` wird durch den Aufruf von `ReleaseSemaphore()` inkrementiert. Wenn der Puffer leer ist, wird der Konsument beim Aufruf von `AquireSemaphore()` blockiert.

Die interne Verwaltung des Puffers ist in den Funktionen `produce_item()` und `consume_item()` verborgen. Nachdem ein Produzent einen Eintrag im Puffer abgelegt hat, wird ein bezüglich des Semaphors `full` wartender Konsument durch den Aufruf von `ReleaseSemaphore(&buffer, full)` aufgeweckt, falls es einen solchen wartenden Konsumenten gibt. Nachdem ein Konsument einen Eintrag aus dem Puffer entnommen hat, wird ein bezüglich des Semaphors `empty` wartender Produzent durch den Aufruf `ReleaseSemaphore(&buffer, empty)` aufgeweckt, falls es einen solchen wartenden Produzenten gibt. Das Programm in Abbildung 6.22 arbeitet mit je einem Produzenten- bzw. Konsumenten-Thread, kann aber auf eine beliebige Anzahl von Threads verallgemeinert werden.

6.2.7 Thread-Scheduling

Die vom Programmierer für jeden Prozess definierten Benutzer-Threads werden vom Bibliotheks-Scheduler auf Betriebssystem-Threads abgebildet, die vom Betriebssystem-Scheduler auf den zur Verfügung stehenden Prozessoren zur Ausführung gebracht werden. In vielen Pthreads-Bibliotheken kann der Programmierer die Abbildung der Benutzer-Threads auf Betriebssystem-Threads durch **Scheduling-Attribute** beeinflussen. Dies ist dann der Fall, wenn das Macro `_POSIX_THREAD_PRIORITY_SCHEDULING` in der Systemdatei `<unistd.h>` definiert ist, was zur Laufzeit des Programms durch Verwendung von `sysconf()` mit aktuellem Parameter `_SC_THREAD_PRIORITY_SCHEDULING` festgestellt werden kann. Ist dies der Fall und soll das Scheduling der Benutzer-Threads durch das Setzen von Scheduling-Attributen beeinflusst werden, muss die Datei `<sched.h>` eingebunden werden. Scheduling-Attribute werden in Datenstrukturen vom Typ `struct sched_param` aufgehoben. Der Pthreads-Standard verlangt, dass diese Datenstruktur mindestens den Eintrag

```
int sched_priority;
```

enthält. Mit Hilfe von Scheduling-Attributen kann jedem Thread eine Scheduling-Priorität, eine Scheduling-Methode und ein Scheduling-Bereich zugeordnet werden. Dies kann beim Start eines Threads statisch festgelegt werden,

```

struct linebuf {
    ENTRY line[100];
    SEMA full, empty;
} buffer;

void *Producer(void *arg)
{
    while (1) {
        AquireSemaphore (&buffer.empty);
        produce_item();
        ReleaseSemaphore (&buffer.full);
    }
}

void *Consumer(void *arg)
{
    while (1) {
        AquireSemaphore (&buffer.full);
        consume_item();
        ReleaseSemaphore (&buffer.empty);
    }
}

void CreateSemaphore (SEMA *ps, int count)
{
    ps->count = count;
    pthread_mutex_init (&ps->mutex, NULL);
    pthread_cond_init (&ps->cond, NULL);
}

int main()
{
    pthread_t threadID[2];
    int i;
    void *status;

    CreateSemaphore (&buffer.empty, 100);
    CreateSemaphore (&buffer.full, 0);

    pthread_create (&threadID[0], NULL, Consumer, NULL);
    pthread_create (&threadID[1], NULL, Producer, NULL);

    for (i=0; i<2; i++)
        pthread_join (threadID[i], &status);
}

```

Abb. 6.22. Realisierung von Produzenten- und Konsumenten-Threads mit Hilfe der in Abbildung 6.21 eingeführten Semaphor-Operationen.

aber auch während der Abarbeitung des Threads dynamisch geändert werden. Wir beschreiben im Folgenden die verschiedenen Möglichkeiten näher. Wir beginnen dabei mit der statischen Festlegung der Scheduling-Attribute eines Threads vor dessen Start.

Die **Scheduling-Priorität** eines Threads legt fest, *wie bevorzugt* der Bibliotheks-Scheduler die Abarbeitung des Threads verfolgen soll. Die Priorität eines Threads wird durch einen Integerwert bezeichnet, der zwischen einem minimalen und einem maximalen Prioritätswert liegen muss. Die für eine Thread-Bibliothek vereinbarte minimale bzw. maximale Priorität erhält man durch Aufruf der Funktionen

```
int sched_get_priority_min (int policy)
int sched_get_priority_max (int policy) ,
```

wobei `policy` die verwendete Scheduling-Methode bezeichnet. Die minimale bzw. maximale Priorität werden als Rückgabewert zurückgegeben. Der Bibliotheks-Scheduler hält für jede Prioritätsstufe eine separate Bereitschaftsschlange, die alle ausführbaren Threads dieser Priorität enthält. Wenn der Bibliotheks-Scheduler einen neuen Thread ausführen will, nimmt er diesen aus der nichtleeren Bereitschaftsschlange mit der höchsten Priorität. Enthält diese Schlange mehrere Threads, wird ein Thread anhand der gewählten Scheduling-Methode ausgewählt. Wenn genügend ausführbare Threads hoher Priorität vorhanden sind, kann es vorkommen, dass Threads niedriger Priorität für lange Zeit nicht ausgeführt werden. Durch den Aufruf der Funktionen

```
int pthread_attr_getschedparam (const pthread_attr_t *attr,
                                struct sched_param *param)
int pthread_attr_setschedparam (pthread_attr_t *attr,
                                const struct sched_param *param)
```

kann der in einer Attributdatenstruktur abgelegte Prioritätswert abgefragt bzw. gesetzt werden. Zum Setzen eines Prioritätswertes muss vor Aufruf der Funktion der Eintrag `param->sched_priority` auf den gewünschten Prioritätswert gesetzt werden.

Die **Scheduling-Methode** eines Threads legt fest, *wie lange* ein Thread ausgeführt wird, wenn er vom Bibliotheks-Scheduler zur Ausführung ausgewählt wird. Der Pthreads-Standard stellt drei Scheduling-Methoden zur Verfügung:

- **SCHED_FIFO** (*first in, first out*): Die ausführbaren Threads einer Prioritätsstufe werden in einer FIFO-Schlange aufgehoben. Wenn ein neuer Thread ausgeführt werden soll, wird er vom *Anfang* der FIFO-Schlange mit der höchsten Priorität entnommen und so lange ausgeführt, bis er entweder beendet ist oder blockiert, oder bis ein Thread höherer Priorität ausführungsbereit wird. Im letzten Fall wird die Ausführung des Threads niedrigerer Priorität unterbrochen, der Thread wird an den *Anfang* der

Bereitschaftsschlange seiner Priorität gehängt, und der Thread höherer Priorität wird ausgeführt. Wird ein blockierter Thread wieder ausführbar, wird er ans *Ende* der zugehörigen Bereitschaftsschlange gehängt. Nach dynamischer Änderung der Priorität eines Threads, wird dieser ans *Ende* der Bereitschaftsschlange zur neuen Priorität gehängt.

- **SCHED_RR**(*round-robin*): Die Thread-Verwaltung arbeitet ähnlich wie bei der SCHED_FIFO Methode. Der Unterschied besteht darin, dass für die Abarbeitung der Threads Zeitscheiben verwendet werden, d.h. jedem aufgeführten Thread wird ein vom Bibliotheks-Scheduler festgelegtes Zeitquantum zugeordnet, nach dessen Ablauf er unterbrochen und ans *Ende* der zugehörigen Bereitschaftsschlange gehängt wird. Die Zeitscheiben werden vom Bibliotheks-Scheduler so festgelegt, dass für alle Threads eines Prozesses Zeitscheiben gleicher Länge verwendet werden. Die Länge der für einen Prozess verwendeten Zeitscheiben kann mit Hilfe der Funktion

```
int sched_rr_get_interval (pid_t pid, struct timespec *quantum)
```

abgefragt werden, wobei `pid` den Bezeichner des angefragten Prozesses angibt. Für `pid=0` werden die Informationen für den anfragenden Prozess zurückgeliefert. Die Datenstruktur vom Typ `timespec` ist definiert als

```
struct timespec { time_t tv_sec; long tv_nsec; } .
```

- **SCHED_OTHER**: Der Pthreads-Standard erlaubt eine zusätzliche Scheduling-Strategie, deren Verhalten von der speziellen Implementierung der Thread-Bibliothek abhängig ist. Damit kann die Scheduling-Strategie an das verwendete Betriebssystem angepasst werden. Oft wird eine Strategie verwendet, die die Prioritäten der Threads an deren I/O-Verhalten anpasst, so dass interaktive Threads eine höhere Priorität erhalten als rechenintensive (CPU-gebundene) Threads. Diese Strategie wird bei den meisten Thread-Bibliotheken als Default-Einstellung verwendet, nach der neu gestartete Threads verwaltet werden.

Die für einen Thread zu verwendende Scheduling-Methode wird beim Start des Threads festgelegt. Wenn der Programmierer eine andere Scheduling-Methode als die Default-Einstellung verwenden will, kann er dies dadurch erreichen, dass er eine Attributdatenstruktur entsprechend besetzt und diese für den Aufruf von `pthread_create()` als Argument zur Verfügung stellt. Durch Aufruf der Funktionen

```
int pthread_attr_getschedpolicy (const pthread_attr_t *attr,
                                int *schedpolicy)
int pthread_attr_setschedpolicy (pthread_attr_t *attr,
                                int schedpolicy)
```

kann die in einer Attributdatenstruktur `attr` abgelegte Scheduling-Methode abgefragt bzw. neu gesetzt werden. Auf manchen UNIX-Systemen erfordert das Setzen der Scheduling-Methode jedoch Superuser-Rechte.

Der **Scheduling-Bereich** (engl. *contention scope*) eines Threads legt fest, welche anderen Threads eines Programms beim Scheduling des Threads berücksichtigt werden. Dabei sind zwei Möglichkeiten vorgesehen: Es können die Threads des Prozesses, dem der Thread angehört, berücksichtigt werden (prozesslokales Scheduling), oder es können die Threads aller Prozesse berücksichtigt werden (globales Scheduling). Durch Aufruf der Funktionen

```
int pthread_attr_getscope (const pthread_attr_t *attr,
                          int *contentionscope)
int pthread_attr_setscope (pthread_attr_t *attr,
                          int contentionscope)
```

kann der in einer Attributdatenstruktur abgelegte Scheduling-Bereich abgefragt bzw. gesetzt werden. Dabei bezeichnet `contentionscope = PTHREAD_SCOPE_PROCESS` ein prozesslokales und `contentionscope = PTHREAD_SCOPE_SYSTEM` ein globales Scheduling. Üblicherweise führt ein prozesslokales Scheduling zu besseren Ergebnissen als ein globales, da der Bibliotheks-Scheduler zwischen den Threads eines Prozesses ohne Beteiligung des Betriebssystems wechseln kann, während für einen Wechsel von einem Thread eines Prozesses *A* zu einem Thread eines anderen Prozesses *B* ein Prozesswechsel und damit eine Beteiligung des Betriebssystems notwendig ist [19]. Eine Pthreads-Bibliothek braucht nur einen der beiden Scheduling-Bereiche zu unterstützen. Wenn ein nicht unterstützter Scheduling-Bereich gesetzt werden soll, liefert der Aufruf von `pthread_attr_setscope()` den Wert `ENOTSUP` zurück.

Um nicht bei jeder Erzeugung eines Threads die Scheduling-Attribute neu setzen zu müssen, stellt der Pthreads-Standard die Möglichkeit zur Verfügung, dass beim Start eines neuen Threads die Scheduling-Attribute des Elternthreads, der den Start auslöst, weitervererbt werden. Durch Aufruf der Funktionen

```
int pthread_attr_getinheritsched (const pthread_attr_t *attr,
                                  int *inheritsched)
int pthread_attr_setinheritsched (pthread_attr_t *attr,
                                  int inheritsched)
```

kann der in der Attributdatenstruktur `attr` abgelegte Vererbungsstatus abgefragt bzw. neu gesetzt werden. Dabei bedeutet `inheritsched = PTHREAD_INHERIT_SCHED`, dass ein mit `attr` erzeugter Thread die Scheduling-Attribute des erzeugenden Threads erbt. Dagegen bedeutet `inheritsched = PTHREAD_EXPLICIT_SCHED`, dass die Scheduling-Attribute nicht vererbt werden, d.h. wenn ein erzeugter Thread andere Scheduling-Attribute als vom Default angegeben haben soll, müssen diese vor seiner Erzeugung in der verwendeten

Attributdatenstruktur explizit gesetzt werden. Der Defaultwert für die Vererbung der Scheduling-Attribute ist nicht vom Pthreads-Standard vorgegeben, d.h. er hängt von der verwendeten Thread-Bibliothek ab.

Die Priorität eines Threads und die verwendete Scheduling-Methode kann auch während der Laufzeit des Threads *dynamisch* geändert werden. Durch Aufruf der Funktionen

```
int pthread_getschedparam (pthread_t thread, int *policy,
                           struct sched_param *param)
int pthread_setschedparam (pthread_t thread, int policy,
                           const struct sched_param *param)
```

können die aktuellen Scheduling-Attribute eines Threads `thread` abgefragt bzw. geändert werden. Dabei gibt `policy` die Scheduling-Methode an, `param` enthält den Prioritätswert.

Als Beispiel für die Verwendung von Scheduling-Attributen zeigt Abbildung 6.23 ein Programmfragment, das die Scheduling-Attribute eines Threads vor dessen Erzeugung explizit festlegt. Als Scheduling-Methode wird `SCHED_RR`, als Priorität wird ein mittlerer Prioritätswert verwendet. Damit die festgelegten Scheduling-Attribute auf den erzeugten Thread übertragen werden, wird der Vererbungsstatus auf `PTHREAD_EXPLICIT_SCHED` gesetzt.

6.2.8 Prioritätsinversion

Bei der Synchronisation mehrerer Threads unterschiedlicher Priorität kann es vorkommen, dass bei ungünstiger Anordnung der Synchronisationsoperationen ein Thread höherer Priorität durch einen Thread niedrigerer Priorität am Weiterarbeiten gehindert wird, obwohl er ausführungsbereit ist. Dieses Phänomen wird als Prioritätsinversion bezeichnet, da ein Thread niedriger Priorität ausgeführt wird, obwohl es einen ausführungsbereiten Thread höherer Priorität gibt.

Beispiel: Das Auftreten des Phänomens wird im Folgenden am Beispiel einer Mutexvariablen m und dreier Threads A, B, C mit hoher, mittlerer bzw. niedriger Priorität demonstriert, vgl. [112]. Die Threads führen zu den Zeitpunkten $T_1, \dots, T_6$ folgende Aktionen aus, vgl. auch Abbildung 6.24.

Nach Start des Programms zum Zeitpunkt T_1 wird der Thread C niedriger Priorität zum Zeitpunkt T_2 gestartet und wird ausgeführt. Zum Zeitpunkt T_3 sperrt C die Mutexvariable m mit `pthread_mutex_lock(m)`; da m bisher frei war, ist der Sperrversuch von C erfolgreich und C wird weiter ausgeführt. Zum Zeitpunkt T_4 startet Thread A mit hoher Priorität; da A eine höhere Priorität als C hat, wird C blockiert und A wird ausgeführt. Die Mutexvariable ist weiterhin durch C gesperrt. Zum Zeitpunkt T_5 versucht A die Mutexvariable m mit `pthread_mutex_lock(m)` zu sperren; da m bereits von C gesperrt ist, wird A bzgl. m blockiert; daraufhin wird C wieder ausgeführt. Zum Zeitpunkt T_6 startet Thread B mit mittlerer Priorität. Da B eine höhere Priorität als

```
#include <unistd.h>
#include <pthread.h>
#include <sched.h>

void *thread_routine (void *arg) {}

int main()
{
    pthread_t thread_id;
    pthread_attr_t attr;
    struct sched_param param;
    int policy, status, min_prio, max_prio;

    status = pthread_attr_init (&attr);
    if (sysconf (_SC_THREAD_PRIORITY_SCHEDULING) != -1) {
        status = pthread_attr_getschedpolicy (&attr, &policy);
        status = pthread_attr_getschedparam (&attr, &param);
        printf ("Default: Policy %d, Priority %d \n", policy,
            param.sched_priority);
        status = pthread_attr_setschedpolicy (&attr, SCHED_RR);
        min_prio = sched_get_priority_min (SCHED_RR);
        max_prio = sched_get_priority_max (SCHED_RR);
        param.sched_priority = (min_prio + max_prio)/2;
        status = pthread_attr_setschedparam (&attr, &param);
        status = pthread_attr_setinheritsched (&attr,
            PTHREAD_EXPLICIT_SCHED);
    }
    status = pthread_create (&thread_id, &attr,
        thread_routine, NULL);
    status = pthread_join (thread_id, NULL);
}
```

Abb. 6.23. Verwendung von Scheduling-Attributen zur Festlegung des Scheduling-Verhaltens eines erzeugten Threads.

Zeitpunkt	Ereignis	Thread A hoher Priorität	Thread B mittlerer Priorität	Thread C niedriger Priorität	Mutex- variable <i>m</i>
T ₁	Start	/	/	/	frei
T ₂	Start C	/	/	laufend	frei
T ₃	C sperrt <i>m</i>	/	/	laufend	von C gesperrt
T ₄	Start A	laufend	/	ausführungsbereit	von C gesperrt
T ₅	A sperrt <i>m</i>	blockiert	/	laufend	von C gesperrt
T ₆	Start B	blockiert	laufend	ausführungsbereit	von C gesperrt

Abb. 6.24. Veranschaulichung des Auftretens einer Prioritätsinversion.

C hat, wird C blockiert und B wird ausgeführt. Wenn C die Mutexvariable m vor Zeitpunkt T_6 nicht freigegeben hat und wenn B nicht versucht, m zu sperren, wird B dauerhaft ausgeführt. Dies geschieht, obwohl es einen Thread A höherer Priorität gibt, der jedoch nicht ausgeführt wird, da er auf die Freigabe der Sperre von m durch C wartet. Die Sperre kann aber nicht freigegeben werden, da C nicht ausgeführt wird. Die ungünstige Anordnung der Sperrversuche führt also dazu, dass die CPU dauerhaft B und nicht A zugeteilt wird, obwohl A eine höhere Priorität als B hat. $\square$

Zur Vermeidung des Auftretens einer Prioritätsinversion stellt Pthreads zwei Mechanismen zur Verfügung, die Verwendung einer Prioritäts-Obergrenze (engl. *priority ceiling*) und die Prioritätsvererbung (engl. *priority inheritance*). Beide Mechanismen sind optional, d.h. sie sind nicht notwendigerweise für jede Pthread-Bibliothek verfügbar. Wir beschreiben im Folgenden beide Mechanismen.

Prioritäts-Obergrenze. Der Mechanismus der Prioritäts-Obergrenze steht für eine Pthreads-Implementierung zur Verfügung, wenn in `<unistd.h>` die Konstante

`_POSIX_THREAD_PRIO_PROTECT`

definiert ist. Der Mechanismus der Prioritäts-Obergrenze ordnet einer Mutexvariablen einen Prioritätswert zu. Die Priorität jedes Threads, der die Mutexvariable sperrt, wird für die Zeit der Sperrung automatisch auf den angegebenen Prioritätswert angehoben. Der Thread kann daher während der Sperrung nicht von einem Thread unterbrochen werden, dessen Wert unter dem Prioritätswert der gesperrten Mutexvariablen liegt. Der sperrende Thread kann daher ohne Unterbrechung weiterarbeiten und die Mutexvariable zügig wieder freigeben.

Im obigen Beispiel wird eine Prioritätsinversion vermieden, wenn als Prioritäts-Obergrenze ein Wert gewählt wird, der größer oder gleich dem Prioritätswert von Thread A ist. Im allgemeinen Fall wird Prioritätsinversion vermieden, wenn als Prioritäts-Obergrenze die höchste auftretende Priorität eines Threads verwendet wird.

Zur Verwendung einer Prioritäts-Obergrenze für eine Mutexvariable muss diese entsprechend initialisiert werden. Dazu wird zuerst ein Mutex-Attributobjekt `attr` vom Typ `pthread_mutex_attr_t` deklariert und mit

`pthread_mutexattr_init(&attr)`

initialisiert. Das eingetragene Default-Prioritätsprotokoll des Attributobjektes kann dann mit Hilfe der Funktion

```
int pthread_mutexattr_getprotocol(const pthread_mutex_attr_t
                                *attr, int *prio)
```

abgefragt werden. Diese liefert im Parameter `prio` das eingetragene Protokoll zurück. Folgende Werte sind möglich:

- `PTHREAD_PRIO_NONE`: keiner der beiden Mechanismen wird verwendet, d.h. die Priorität eines Threads ändert sich nicht, wenn er die Mutexvariable sperrt;
- `PTHREAD_PRIO_PROTECT`: der Mechanismus der Prioritäts-Obergrenze wird verwendet;
- `PTHREAD_PRIO_PROTECT`: das Vererben von Prioritäten wird verwendet.

Mit Hilfe der Funktion

```
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,
                                int prio)
```

kann das zu verwendende Prioritätsprotokoll verändert werden, wobei `prio` beim Aufruf einen der gerade angegebenen Werte haben muss.

Bei Verwendung einer Prioritäts-Obergrenze kann der eingetragene Wert der Obergrenze mit Hilfe der Funktionen

```
int pthread_mutexattr_getprioceiling(const pthread_mutexattr_t
                                    *attr, int *prio)
int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,
                                    int prio)
```

abgefragt, bzw. gesetzt werden. Beim Setzen einer Prioritäts-Obergrenze muss der in `prio` angegebene Wert im erlaubten Bereich liegen. Nach Erzeugen und Setzen eines entsprechenden Mutex-Attributobjektes `attr` kann dieses zur Initialisierung einer Mutexvariablen `m` durch

```
pthread_mutex_init (&m, attr)
```

verwendet werden. Für diese wird dann z.B. der Mechanismus der Prioritäts-Obergrenze verwendet.

Prioritätsvererbung. Bei Verwendung der Prioritätsvererbung wird die Priorität eines Threads, der Eigentümer einer Mutexvariablen ist, automatisch angehoben, sobald ein Thread höherer Priorität versucht, die Mutexvariable zu sperren und daher bezüglich der Mutexvariablen blockiert. Das Anheben geschieht so, dass der Eigentümer-Thread die Priorität des blockierten Threads annimmt. Damit hat der Eigentümer-Thread immer die maximale Priorität aller bezüglich der Mutexvariablen blockierten Threads und Prioritätsinversion kann nicht auftreten. Gibt der Eigentümer-Thread die Mutexvariable wieder frei, erhält er seine ursprüngliche Priorität wieder zurück.

Die Methode der Prioritätsvererbung ist anwendbar, wenn in `<unistd.h>` die Konstante `_POSIX_THREAD_PRIO_INHERIT` definiert ist. Die Aktivierung des Mechanismus für eine Mutexvariable geschieht auf die oben beschriebene Weise mit `PTHREAD_PRIO_PROTECT`. Gegenüber der Verwendung einer Prioritäts-Obergrenze hat die Prioritätsvererbung den Vorteil, dass kein Wert für die Obergrenze angegeben werden muss. Eine Prioritätsinversion wird auch bei Threads unbekannter Priorität sicher vermieden. Allerdings ist

die Realisierung der Prioritätsvererbung aufwendiger und führt typischerweise zu einem höheren Laufzeitaufwand als bei Verwendung einer Prioritäts-Obergrenze.

6.2.9 Thread-spezifische Daten

Die Threads eines Prozesses haben einen gemeinsamen Adressraum. Daher können globale und dynamisch allokierte Variablen von jedem Thread eines Prozesses zugegriffen und manipuliert werden. Da aber für jeden Thread ein eigener Stack gehalten wird, auf dem die von diesem Thread durchgeführten Funktionsaufrufe verwaltet werden, sind die innerhalb einer Funktion deklarierten lokalen Variablen nur von dem Thread zugreifbar, der die Funktion aufgerufen hat. Lokale Variablen sind also immer threadlokal. Der Nachteil von lokalen Variablen besteht aber darin, dass ihre Lebensdauer nur die Funktion, in der sie deklariert sind, umfasst. Wenn der Wert einer lokalen Variable während der gesamten Lebensdauer eines Threads zugreifbar sein soll, muss die lokale Variable in der Startfunktion des Threads deklariert werden und allen vom Thread aufgerufenen Funktionen als Parameter übergeben werden. Je nach Anwendung kann dies aber sehr umständlich sein und die Zahl der Parameter unnatürlich aufblähen. Daher stellt der Pthreads-Standard einen zusätzlichen Mechanismus zur Verwaltung von threadlokalen Daten bereit.

Zur Erzeugung von threadlokalen Daten stellen Pthreads-Bibliotheken **Schlüssel** (keys) zur Verfügung, die prozessglobal verwaltet werden, d.h. nach der Erzeugung eines Schlüssels kann jeder Thread des Prozesses auf den Schlüssel zugreifen; jeder Thread kann aber den Schlüsseln threadspezifische Daten zuordnen. Wenn verschiedene Threads dem gleichen Schlüssel unterschiedliche Daten zugeordnet haben, erhält jeder Thread beim Zugriff auf den Schlüssel seine spezifischen Daten zurück. Die entsprechende Verwaltung der Zugriffe und Ablage wird von der Thread-Bibliothek übernommen.

Schlüssel haben in Pthreads-Bibliotheken den Datentyp `pthread_key_t`. Ein Schlüssel kann durch Aufruf der Funktion

```
int pthread_key_create (pthread_key_t *key,
                      void (*destructor)(void *))
```

erzeugt werden. Der Aufruf liefert in `key` einen Zeiger auf den erzeugten Schlüssel zurück. Wenn der Schlüssel von mehreren Threads genutzt werden soll, muss `key` eine globale oder eine dynamisch allokierte Variable sein. Der gleiche Schlüssel darf nur einmal erzeugt werden. Der optionale Funktionsparameter `destructor` kann dazu verwendet werden, einem Schlüssel eine Deallokationsfunktion zuzuordnen, die aufgerufen wird, wenn der Thread normal terminiert.

Wenn keine Deallokation erforderlich ist, sollte `NULL` übergeben werden. Ein Schlüssel kann durch Aufruf der Funktion

```
int pthread_key_delete (pthread_key_t key)
```

wieder gelöscht werden.

Nach der Erzeugung eines Schlüssels ist diesem für jeden Thread der Wert NULL zugeordnet. Jeder Thread kann dem Schlüssel durch Aufruf der Funktion

```
int pthread_setspecific (pthread_key_t key, void *value)
```

einen neuen Wert `value` zuordnen. Üblicherweise wird die Adresse eines dynamisch allokierten Datenobjektes übergeben. Die Übergabe der Adresse einer lokalen Variable sollte vermieden werden, da diese nach Beendigung der zugehörigen Funktion ungültig wird. Der von einem Thread für einen Schlüssel abgelegte Wert kann durch Aufruf der Funktion

```
void *pthread_getspecific (pthread_key_t key)
```

bestimmt werden. Der aufrufende Thread erhält immer den Wert zurück, den er vorher mit `pthread_setspecific()` für den Schlüssel abgelegt hat. Wenn er noch keinen Wert abgelegt hat, erhält er NULL zurück. Dies gilt auch für den Fall, dass ein anderer Thread bereits einen Wert für den Schlüssel abgelegt hat. Wenn ein Thread durch Aufruf von `pthread_setspecific()` für einen Schlüssel einen Wert ablegt, ist ein evtl. vorher von ihm unter dem gleichen Schlüssel abgelegter Wert verloren.

6.3 Java-Threads

Die Entwicklung von aus mehreren Threads bestehenden Programmen wird in der objektorientierten Programmiersprache Java auf Sprachebene unterstützt. Java stellt dazu u.a. Sprachkonstrukte für die synchronisierte Ausführung von Programmbereichen bereit und erlaubt die Erzeugung und Verwaltung von Threads durch Verwendung geeigneter Klassen. Im Folgenden wird die Verwendung von Java-Threads zur Entwicklung paralleler Programme für einen gemeinsamen Adressraum vorgestellt. Der Leser sollte mit den Grundprinzipien objektorientierter Programmierung sowie den Standard-Sprachelementen der Programmiersprache Java bereits vertraut sein. In diesem Abschnitt konzentrieren wir uns auf die Darstellung der Mechanismen zur Entwicklung thread-paralleler Programme und gehen dabei nur auf die wichtigsten Elemente ein. Ausführliche Darstellungen sind z.B. in [115, 100] verfügbar. Für einen Überblick über die Programmiersprache Java verweisen wir auf [46].

6.3.1 Erzeugung von Threads in Java

Jedes Java-Programm besteht aus mindestens einem Thread, dem Haupt-Thread. Dieses ist der Thread, der die `main()`-Methode der Klasse ausführt,

die als Startargument der Java Virtual Machine (JVM) angegeben wird. Weitere Benutzer-Threads werden von diesem Haupt-Thread oder von bereits erzeugten Threads explizit erzeugt und gestartet. Dazu steht die vordefinierte Klasse `Thread` aus `java.lang` zur Verfügung, die zur Repräsentation von Threads verwendet wird und Mechanismen und Methoden zur Erzeugung und Verwaltung von Threads bereitstellt. Das Interface `Runnable` aus `java.lang` repräsentiert den von einem Thread auszuführenden Code. Dieser wird in einer `run()`-Methode zur Verfügung gestellt. Für die Definition einer `run()`-Methode, die von einem Thread asynchron ausgeführt wird, gibt es zwei Möglichkeiten, das Erben von der Klasse `Thread` oder die Implementierung des Interface `Runnable`.

Erben von der Klasse `Thread`. Eine Möglichkeit besteht in der Definition einer neuen Klasse `NewClass`, die von der vordefinierten Klasse `Thread` erbt und in der eine neue Methode `run()` mit den Anweisungen des auszuführenden Threads definiert wird. Da bereits die Klasse `Thread` eine `run()`-Methode enthält, wird diese durch die Definition der neuen `run()`-Methode überschrieben. Zusätzlich enthält die Klasse `Thread` eine Methode `start()`, die einen neuen Thread erzeugt, der dann die Methode `run()` ausführt. Der neu erzeugte Thread wird asynchron zum aufrufenden Thread ausgeführt. Nach Ausführung von `start()` wird die Kontrolle direkt an den aufrufenden Thread zurückgegeben. Dies erfolgt evtl. vor der Beendigung des neu erzeugten Threads, so dass erzeugender und erzeugter Thread asynchron zueinander arbeiten. Der neu erzeugte Thread terminiert, sobald seine `run()`-Methode vollständig abgearbeitet ist.

Dieses Vorgehen ist in Abbildung 6.25 am Beispiel eines Applets illustriert, dessen `init()`-Methode ein Objekt der Klasse `NewClass` erzeugt und dessen `run()`-Methode aktiviert. Zusammenfassend läuft die Erzeugung eines Threads in zwei Schritten:

- (1) Definition einer neuen Klasse `NewClass`, die von der Klasse `Thread` erbt und die eine Methode `run()` definiert, die die Anweisungsfolge, die vom neuen Thread ausgeführt werden soll, enthält;
- (2) Erzeugung eines Objektes der Klasse `NewClass` und Aktivierung der `start()`-Methode.

Bei der gerade beschriebenen Methode zur Erzeugung eines Threads muss die neue Klasse von der Klasse `Thread` erben. Da Java keine Mehrfach-Vererbung zulässt, hat dies den Nachteil, dass die neue Klasse von keiner weiteren Klasse erben kann, was die Entwicklung von Anwendungsprogrammen einschränkt. Dieser Nachteil der fehlenden Mehrfach-Vererbung wird in Java durch die Bereitstellung von Interfaces ausgeglichen, wofür im Falle der Klasse `Thread` das Interface `Runnable` genutzt wird.

Verwendung des Interface `Runnable`. Das Interface `Runnable` enthält eine parameterlose `run()`-Methode:

```

import java.lang.Thread;
public class NewClass extends Thread { // Vererbung
    public void run() {
        // Überschreiben der run()-Methode der Thread-Klasse
        System.out.println("hello from new thread");
    }
}
import java.applet.Applet;
public class NewApplet extends Applet { // Vererbung
    public void init() {
        NewClass nc = new NewClass();
        nc.start(); // start() ruft run() in neuem Thread auf
    }
}

```

Abb. 6.25. Erzeugung eines Threads durch Überschreiben der `run()`-Methode der Klasse `Thread`.

```

public interface Runnable {
    public abstract void run();
}

```

Die vordefinierte Klasse `Thread` implementiert das Interface `Runnable`, d.h. jede von `Thread` abgeleitete Klasse implementiert ebenfalls das Interface `Runnable`. Eine neu erzeugte Klasse `NewClass` kann somit auch direkt das Interface `Runnable` implementieren anstatt von der Klasse `Thread` abgeleitet zu werden. Objekte einer solchen Klasse `NewClass` sind aber keine Threadobjekte, so dass zur Erzeugung eines Threads immer noch ein Objekt der Klasse `Thread` erzeugt werden muss, das allerdings als Parameter ein Objekt der neuen Klasse `NewClass` hat.

Dazu enthält die Klasse `Thread` einen Konstruktor `Thread(Runnable target)`. Bei Verwendung dieses Konstruktors ruft die `start()`-Methode von `Thread` die `run()`-Methode des Parameterobjektes vom Typ `Runnable` auf. Dies wird durch die `run()`-Methode von `Thread` erreicht, die wie folgt definiert ist:

```

public void run() {
    if (target != null) target.run();
}

```

Die `run()`-Methode wird in einem separaten, neu erzeugten Thread asynchron zum aufrufenden Thread ausgeführt. Die Erzeugung eines neuen Threads kann somit in drei Schritten erfolgen:

- (1) Definition einer neuen Klasse `NewClass`, die `Runnable` implementiert und für die eine `run()`-Methode definiert wird, die die von dem neu zu erzeugenden Thread auszuführende Anweisungsfolge enthält;

- (2) Erzeugung eines Objektes der Klasse `Thread` mit Hilfe des Konstruktors `Thread(Runnable target)` und eines Objektes der Klasse `NewClass` und Übergabe dieses Objektes an den `Thread`-Konstruktor;
- (3) Aufruf der `start()`-Methode des `Thread`-Objektes.

Dieses Vorgehen ist in Abbildung 6.26 am Beispiel eines Applets illustriert, vgl. auch Abbildung 6.25. Zur Definition der `run()`-Methode wird eine neue Klasse `NewClass` verwendet. Ein Objekt dieser Klasse wird dem Konstruktor von `Thread` als Parameter übergeben. Bei Nutzung des Interface `Runnable` kann die Definition einer neuen Klasse vermieden werden, indem die `run()`-Methode direkt in der Klasse `NewApplet` definiert wird. Dies ist in Abbildung 6.27 illustriert.

```
import java.lang.Thread;
public class NewClass implements Runnable {
    public void run()
        System.out.println("hello from new thread");
    }
}
import java.applet.Applet;
public class NewApplet extends Applet {
    public void init() {
        NewClass nc = new NewClass();
        Thread th = new Thread(nc);
        th.start(); // start() ruft nc.run() auf
    }
}
```

Abb. 6.26. Erzeugung eines Threads mit Hilfe des Interface `Runnable` und Verwendung einer neuen Klasse `NewClass`.

```
import java.applet.Applet;
public class NewApplet extends Applet implements Runnable {
    public void init() {
        Thread th = new Thread(this);
        th.start(); // start() ruft nc.run() auf
    }
    public void run() {
        System.out.println("hello from new thread");
    }
}
```

Abb. 6.27. Erzeugung eines Threads mit Hilfe des Interface `Runnable` ohne Verwendung einer neuen Klasse.

Weitere Methoden der Klasse Thread. Ein Java-Thread kann auf die Beendigung eines anderen Java-Threads `t` warten, indem er `t.join()` aufruft. Dieser Aufruf bewirkt eine Blockierung des aufrufenden Threads, bis der Thread `t` beendet ist. Die `join()`-Methode wird in drei Varianten zur Verfügung gestellt:

- `void join()`: der aufrufende Thread wird blockiert, bis der angegebene Thread beendet ist;
- `void join(long timeout)`: der aufrufende Thread wird blockiert; die Blockierung wird aufgehoben, sobald der angegebene Thread beendet ist oder wenn die angegebene Zeit `timeout` abgelaufen ist (Angabe in Millisekunden);
- `void join(long timeout, int nanos)`: das Verhalten ist wie für `void join(long timeout)`; der zusätzliche Parameter ermöglicht eine genauere Angabe des Zeitintervalls durch zusätzliche Angabe von Nanosekunden.

Wurde der angegebene Thread noch nicht gestartet, findet bei keiner der `join()`-Varianten eine Blockierung statt.

Die Methode

```
boolean isAlive()
```

der Klasse `Thread` ermöglicht die Abfrage des Ausführungsstatus eines Threads: die Methode liefert `true` zurück, falls der angegebene Thread gestartet wurde, aber noch nicht beendet ist. Weder die `isAlive()`-Methode noch die verschiedenen Varianten der `join`-Methode haben einen Einfluss auf den Thread, der Ziel des Aufrufes ist. Nur der ausführende Thread ist betroffen.

Einem Thread kann ein Name zugewiesen werden, der später wieder genutzt werden kann. Dazu stehen folgende Methoden der `Thread`-Klasse zur Verfügung:

```
void setName (String name);
String getName();
```

Die Zuweisung eines Namens kann auch direkt bei der Erzeugung eines Threads durch Verwendung des Konstruktors `Thread(String name)` erfolgen.

Die `Thread`-Klasse definiert einige statische Methoden, die den aktuell ausgeführten Thread betreffen oder Informationen über das Gesamtprogramm liefern:

```
static Thread currentThread();
static void sleep (long milliseconds);
static void yield();
static int enumerate (Thread[] th_array);
static int activeCount();
```

Da diese Methoden statisch sind, können sie aufgerufen werden, auch wenn kein Objekt der Klasse **Thread** verwendet wird. Der Aufruf der Methode **currentThread()** liefert eine Referenz auf das **Thread**-Objekt des aufrufenden Threads. Diese Referenz kann z.B. dazu verwendet werden, nicht-statische Methoden dieses **Thread**-Objektes aufzurufen. Die Methode **sleep()** blockiert den ausführenden Thread vorübergehend für die angegebene Anzahl von Millisekunden, d.h. der Prozessor kann einem anderen Thread zugeteilt werden. Nach Ablauf des Zeitintervalls wird der Thread wieder ausführungsbereit und kann wieder einem Prozessor zur weiteren Ausführung zugeteilt werden.

Die Methode **yield()** ist ein Hinweis an die Java Virtual Machine (JVM), dass ein anderer ausführungsbereiter Thread gleicher Priorität dem Prozessor zugeteilt werden soll. Wenn ein solcher Thread existiert, kann der Scheduler der JVM diesen zur Ausführung bringen. Die Anwendung von **yield()** ist sinnvoll für JVM-Implementierungen ohne Scheduling mit Zeitscheibenverfahren, wenn Threads langlaufende Berechnungen ohne Blockierungsmöglichkeit ausführen. Die Methode **enumerate()** liefert eine Liste aller **Thread**-Objekte des Programms. Der Rückgabewert gibt die Anzahl der im Parameterfeld **th_array** abgelegten **Thread**-Objekte an. Mit der Methode **activeCount()** kann die Anzahl der **Thread**-Objekte des Programms bestimmt werden. Die Methode kann z.B. verwendet werden, um vor Aufruf von **enumerate()** die erforderliche Größe des Parameterfeldes zu ermitteln.

Beispiel: Abbildung 6.28 zeigt als Beispiel ein Fragment einer Klasse zur Durchführung einer Matrix-Multiplikation mit mehreren Threads. Die zu multiplizierenden Matrizen werden vom Haupt-Thread in **in1** und **in2** mit der statischen Methode **ReadMatrix()** eingelesen. Die Erzeugung der Threads erfolgt durch den Konstruktor der **MatMult**-Klasse, so dass jeder Thread eine Zeile **row** der Ergebnismatrix **out** berechnet. Die zugehörigen Berechnungen werden in der **run()**-Methode spezifiziert. Alle Threads arbeiten auf den vom Haupt-Thread allokierten Matrizen **in1**, **in2** und **out**. Da jeder Thread einen separaten Bereich der Ergebnismatrix **out** beschreibt, ist keine Synchronisation der Threads erforderlich. □

6.3.2 Synchronisation von Java-Threads

Die Threads eines Java-Programms arbeiten auf einem gemeinsamen Adressraum. Wenn auf Variablen durch mehrere Threads zugegriffen werden kann, müssen also zur Vermeidung zeitkritischer Abläufe geeignete Synchronisationsmechanismen angewendet werden. Zur Sicherstellung des wechselseitigen Ausschlusses von Threads beim Zugriff auf gemeinsame Daten stellt Java **synchronized**-Blöcke und -Methoden zur Verfügung. Wird ein Block oder eine Methode als **synchronized** deklariert, ist sichergestellt, dass keine gleichzeitige Ausführung durch zwei Threads erfolgen kann. Eine Datenstruktur kann also dadurch vor konkurrierenden Zugriffen mehrerer Threads geschützt


```

import java.lang.*;
import java.io.*;
class MatMult extends Thread {
    static int in1[] []; static int in2[] []; static int out[] [];
    static int n=3, int row;
    MatMult (int i) {
        row=i;
        this.start();
    }
    public void run() {
        //Berechnung der Zeile row der Resultatmatrix out
        int i,j;
        for(i=0;i<n;i++) {
            out[row][i]=0;
            for (j=0;j<n;j++)
                out[row][i]=out[row][i]+in1[row][j]*in2[j][i];
        }
    }
    public static void ReadMatrix (int in[] []) {
        //Lesen der Eingabematrix
        int i,j;
        BufferedReader br=new BufferedReader(new
            InputStreamReader(System.in));
        System.out.println("Enter the Matrix : ");
        for(i=0;i<n;i++)
            for(j=0;j<n;j++)
                try {
                    in[i][j]=Integer.parseInt(br.readLine());
                } catch(Exception e){ }
    }
    public static void PrintMatrix (int out[] []) {
        //Ausgabe der Ergebnismatrix
        int i,j;
        System.out.println("OUTPUT :");
        for(i=0;i<n;i++)
            for(j=0;j<n;j++)
                System.out.println(out[i][j]);
    }
    public static void main(String args[]) {
        int i,j;
        in1=new int[n][n]; in2=new int[n][n];
        out=new int[n][n];
        ReadMatrix(in1); ReadMatrix(in2);
        MatMult mat[] = new MatMult[n];
        for(i=0;i<n;i++)
            mat[i]=new MatMult(i);
        try {
            for(i=0;i<n;i++)
                mat[i].join();
        } catch(Exception e){ }
        PrintMatrix(out);
    }
}

```

Abb. 6.28. Parallele Matrix-Matrix-Multiplikation in Java.

werden, dass alle Zugriffe auf die Datenstruktur in **synchronized** Methoden oder Blöcken erfolgen.

Die synchronisierte Inkrementierung eines Zählers kann beispielsweise durch folgende Methode `incr()` realisiert werden:

```
public class Counter {
    private int value;
    public synchronized int incr() {
        value = value + 1;
        return value;
    }
}
```

In der JVM wird die Synchronisation dadurch realisiert, dass jedem Java-Objekt implizit eine **Mutexvariable** zugeordnet wird. Jedes Objekt der allgemeinen Klasse `Object` besitzt eine solche implizite Mutexvariable. Da jede Klasse direkt oder indirekt von der Klasse `Object` abgeleitet ist, besitzt somit jedes Objekt eine Mutexvariable. Der Aufruf einer **synchronized**-Methode bezüglich eines Objektes `Ob` hat den folgenden Effekt:

- Beim Start der **synchronized**-Methode durch einen Thread `t` wird die Mutexvariable von `Ob` implizit belegt. Wenn die Mutexvariable bereits von einem anderen Thread belegt ist, wird der ausführende Thread `t` blockiert. Der so blockierte Thread wird wieder ausführungsbereit, wenn die Mutexvariable freigegeben wird. Die aufgerufene **synchronized**-Methode wird nur bei erfolgreicher Sperrung der Mutexvariablen von `Ob` ausgeführt.
- Beim Verlassen der Methode wird die Mutexvariable von `Ob` implizit wieder freigegeben und kann damit von einem anderen Thread gesperrt werden.

Damit kann ein synchronisierter Zugriff auf ein Objekt dadurch realisiert werden, dass alle Methoden, die konkurrierend durch mehrere Threads aufgerufen werden können, als **synchronized** deklariert werden. Zur Sicherstellung des wechselseitigen Ausschlusses ist es wichtig, dass nur über diese Methoden auf das zu schützende Objekt zugegriffen wird. Neben **synchronized**-Methoden können auch **synchronized**-Blöcke verwendet werden. Dies ist dann sinnvoll, wenn nur ein Teil einer Methode auf kritische Daten zugreift, eine synchronisierte Ausführung der gesamten Methode aber nicht notwendig ist. Bei **synchronized**-Blöcken erfolgt die Synchronisation meist bezüglich des Objektes, in dessen Methode der **synchronized**-Block steht. Die obige Methode zur Inkrementierung eines Zählers kann mit Hilfe eines **synchronized**-Blocks folgendermaßen formuliert werden:

```
public int incr() {
    synchronized (this) {
        value = value + 1; return value;
    }
}
```

Synchronisation mit variabler Sperrgranularität. Wir werden im Folgenden die Arbeitsweise der Synchronisationsmechanismen von Java dadurch illustrieren, dass wir schrittweise eine Synchronisationsklasse mit variabler Sperrgranularität entwerfen, vgl. auch [115]. Die neue Klasse `MyMutex` erlaubt die Synchronisation beliebiger Objektzugriffe durch explizites Sperren und Freigeben von Objekten der neuen Synchronisationsklasse, so dass der Sperrmechanismus ähnlich wie Mutexvariablen in Pthreads verwendet werden kann, vgl. Abschnitt 6.2.2. Auch die Synchronisation von Threads, die auf verschiedene Objekte zugreifen, ist mit dieser neuen Klasse möglich. Die Synchronisationsklasse verwendet ein Flag, das anzeigt, welcher Thread aktuell ein Objekt gesperrt hat. Abbildung 6.29 zeigt einen ersten Entwurf der Synchronisationsklasse `MyMutex`.

```
public class MyMutex {
    protected Thread OwnerThread = null;
    public void getMyMutex() {
        while (!tryMyGetMyMutex()) {
            try { Thread.sleep(100); }
            catch (InterruptedException e) { }
        }
    }
    public synchronized boolean tryGetMyMutex() {
        if (OwnerThread == null) {
            OwnerThread = Thread.currentThread();
            return true;
        }
        else return false;
    }
    public synchronized void freeMyMutex() {
        if (OwnerThread == Thread.currentThread())
            OwnerThread = null;
    }
}
```

Abb. 6.29. Synchronisationsklasse mit variabler Sperrgranularität.

Die Methode `getMyMutex()` dient zur Anfrage auf Zuteilung einer Sperre an den aufrufenden Thread. Eine Zuteilung erfolgt, indem das Flag `OwnerThread` auf den aufrufenden Thread `Thread.currentThread()` gesetzt wird. Die `synchronized` Methode `freeMyMutex()` gibt eine vorher zugeteilte Sperre wieder frei, indem das Flag `OwnerThread` wieder auf `null` gesetzt wird. Ist die angefragte Sperre bereits belegt, versucht `getMyMutex()` in festgelegten Zeitintervallen erneut die Sperre zu erhalten. Wichtig ist, dass `getMyMutex()` nicht als `synchronized` deklariert ist, für den Zugriff auf das Flag aber die `synchronized`-Methode `tryGetMyMutex()` verwendet. Dies schützt den kritischen Bereich der Zuteilung der Sperre durch die impli-

zite Mutexvariable des Objektes. Da es nur eine implizite Mutexvariable des Objektes gibt, wird auch der konkurrierende Zugriff auf die **synchronized**-Methoden `tryGetMyMutex()` und `freeMyMutex()` geschützt.

Wäre `getMyMutex()` als **synchronized** deklariert, so würde ein Aufruf von `getMyMutex()` durch einen Thread A zum Sperren der Mutexvariablen des zugehörigen Objektes der Klasse `MyMutex` führen. Wenn das Flag zu diesem Zeitpunkt von einem anderen Thread B gesetzt wäre, könnte B das Flag nicht mit `freeMyMutex()` freigeben, da die Mutexvariable des Objektes durch Thread A gesperrt ist. Dies würde dann zu einem Deadlock führen. Der Aufwand für die zusätzliche Methode `tryGetMyMutex()` kann durch Verwendung eines **synchronized**-Blocks vermieden werden. Die Methode `getMyMutex()` kann dann wie folgt realisiert werden:

```
public void getMyMutex() {
    for ( ; ; ) {
        synchronized(this) {
            if (OwnerThread == null) {
                OwnerThread = Thread.currentThread();
                break;
            }
        }
        try { Thread.sleep(100); }
        catch (InterruptedException e) { }
    }
}
```

Die Synchronisationsklasse `MyMutex` kann zur gezielten Definition von kritischen Bereichen verwendet werden. Dies kann am Beispiel der oben eingeführten Zählerklasse demonstriert werden:

```
public class Counter {
    private int value;
    private MyMutex flag = new MyMutex();
    public int incr() {
        int res;
        flag.getMyMutex();
        value = value + 1;
        res = value;
        flag.freeMyMutex();
        return res;
    }
}
```

Bei Verwendung von **synchronized**-Blöcken kann ein Deadlock auftreten, wenn ein Thread bezüglich mehrerer Objekte synchronisieren muss und unterschiedliche Threads verschiedene Reihenfolgen für das Sperren der Objekte verwenden. Eine Abhilfe besteht entweder in der Verwendung einer Backoff-Strategie oder die Anwendung einer festen Sperr-Reihenfolge, vgl. Abschnitt 6.2.2, Seite 292.

Die beschriebene Realisierung von **synchronized**-Blöcken mit Hilfe der impliziten Mutexvariablen, die jedem Objekt zugeordnet sind, funktioniert für alle Methoden, die bezüglich eines Objektes aktiviert werden. Statische Methoden einer Klasse werden jedoch nicht bezüglich eines speziellen Objektes aktiviert und eine implizite Objekt-Mutexvariable gibt es daher nicht. Nichtsdestotrotz können auch statische Methoden als **synchronized** deklariert werden. Die Synchronisation erfolgt dann über die Mutexvariable des zugehörigen Klassenobjektes der Klasse `java.lang.Class`, das für die Klasse, in der die statische Methode deklariert wurde, automatisch erzeugt wurde. Statische und nicht-statische **synchronized** Methoden einer Klasse verwenden also unterschiedliche Mutexvariablen für die Synchronisation. Eine statische **synchronized**-Methode kann sowohl die Mutexvariable der Klasse als auch die Mutexvariable eines Objektes der Klasse sperren, indem sie eine nicht-statische Methode bzgl. eines Objektes der Klasse aufruft oder ein Objekt der Klasse zur Synchronisation nutzt. Dies wird in Abbildung 6.30 illustriert. Eine **synchronized** nicht-statische Methode kann durch den Aufruf einer **synchronized** statischen Methode ebenfalls neben der Objekt-Mutexvariablen auch die Klassen-Mutexvariable sperren. Für eine Klasse `C1` kann die Synchronisation bzgl. der Klassen-Mutexvariablen auch direkt durch

```
synchronized (C1.class) { /* Rumpf*/ }
```

erfolgen.

```
public class MyStatic {
    public static synchronized void staticMethod(MyStatic obj) {
        // hier wird die Klassen-Sperre verwendet
        obj.nonStaticMethod();
        synchronized(obj) {
            // hier wird zusätzlich die Objekt-Sperre verwendet
        }
    }
    public synchronized void nonStaticMethod() {
        // Verwendung der Objekt-Sperre
    }
}
```

Abb. 6.30. Synchronisation von statischen Methoden.

6.3.3 Signalmechanismus in Java

In manchen Situationen ist es sinnvoll, dass ein Thread auf das Eintreten einer speziellen Bedingung wartet, vgl. auch Seite 294. Sobald die Bedingung erfüllt ist, führt der Thread eine festgelegte Aktion aus. So lange die Bedingung noch nicht erfüllt ist, wartet der Thread darauf, dass ein anderer Thread

durch entsprechende Aktionen das Eintreten der Bedingung herbeiführt. In Pthreads konnten für solche Situationen Bedingungsvariablen eingesetzt werden. Java stellt über die Methoden `wait()` und `notify()`, die in der vordefinierten Klasse `Object` deklariert sind, einen ähnlichen Mechanismus zur Verfügung. Diese Methoden stehen somit für jedes Objekt zur Verfügung, da jedes Objekt direkt oder indirekt von der Klasse `Object` abgeleitet ist. Beide Methoden dürfen aber nur innerhalb eines `synchronized`-Blocks oder einer `synchronized`-Methode aufgerufen werden. Das typische Verwendungsmuster für `wait()` sieht wie folgt aus:

```
synchronized (lockObject) {
    while (!Bedingung) { lockObject.wait(); }
    Aktion;
}
```

Der Aufruf von `wait()` blockiert den aufrufenden Thread so lange, bis er von einem anderen Thread per `notify()` aufgeweckt wird. Diese Blockierung bewirkt auch die Freigabe der impliziten Mutexvariable des Objektes, bezüglich der der Thread synchronisiert. Damit kann diese Mutexvariable von einem anderen Thread gesperrt werden. Ein Aufruf von `notify()` weckt einen bezüglich dem zugehörigen Objekt blockierten Thread auf. Der aufgeweckte Thread wird ausführungsbereit und versucht, wieder die Kontrolle über die implizite Mutexvariable des Objektes zu erhalten. Erst wenn ihm dies gelingt, führt er die beim Eintreten der Bedingung durchzuführende Aktion aus. Wenn dies nicht gelingt, blockiert der Thread bezüglich der Mutexvariablen, bis diese von dem Thread, der sie gesperrt hat, wieder freigegeben wird.

Die Arbeitsweise von `wait()` und `notify()` ähnelt damit stark der Arbeitsweise von `pthread_cond_wait()` und `pthread_cond_signal()`, vgl. Seite 294. Die Implementierung von `wait()` und `notify()` erfolgt mit Hilfe einer impliziten Warteliste, in der für jedes Objekt eine Menge von wartenden Threads gehalten wird. Die Warteliste enthält jeweils die Threads, die zum aktuellen Zeitpunkt durch Aufruf von `wait()` bezüglich dieses Objektes blockiert wurden. Nicht in der Warteliste enthalten sind die Threads, die blockiert wurden, weil sie auf Zuteilung der impliziten Mutexvariable des Objektes warten. Welcher der Threads in der impliziten Warteliste beim Aufruf von `notify()` aufgeweckt wird, wird von der Java-Sprachspezifikation nicht festgelegt. Mit Hilfe der Methode `notifyAll()` werden alle in der Warteliste abgelegten Threads aufgeweckt und werden ausführungsbereit; die analoge Pthreads-Funktion ist `pthread_cond_broadcast()`. Ebenso wie `notify()` muss `notifyAll()` in einem `synchronized`-Block oder -Methode aufgerufen werden.

Der Java-Signalmechanismus kann dazu verwendet werden, die Synchronisationsklasse aus Abbildung 6.29 dahingehend zu verbessern, dass das aktive Warten in der Methode `getMyMutex()` vermieden wird, vgl. Abbildung 6.31.

Hierzu wird ein geschachtelter Sperrmechanismus realisiert, der das mehrfache Sperren des verwendeten Flags durch den gleichen Thread erlaubt. Die Anzahl der Sperren wird durch die Variable `lockCount` erfasst, die mit 0 initialisiert wird und bei jedem Aufruf von `getMyMutex()` bzw. `freeMyMutex()` durch denselben Thread inkrementiert bzw. dekrementiert wird. In Abbildung 6.31 ist die Methode `getMyMutex()` jetzt als `synchronized` deklariert. Wie oben erläutert, hätte dies in Abbildung 6.29 zu einem Deadlock geführt. Hier kann ein solcher Deadlock nicht auftreten, da der Aufruf von `wait()` in `getMyMutex()` die Sperre der impliziten Mutexvariablen wieder freigibt, bevor der Thread suspendiert und in die Warteschlange gehängt wird.

```
public synchronized void getMyMutex() {
    while (!tryGetMyMutex()) {
        try { wait(); }
        catch (InterruptedException e) { }
    }
}

public synchronized boolean tryGetMyMutex() {
    if (OwnerThread == null) {
        OwnerThread = Thread.currentThread();
        lockCount = 1; return true;
    }
    if (OwnerThread == Thread.currentThread()) {
        lockCount ++; return true;
    }
    return false;
}

public synchronized Thread getMutexOwner() {
    return OwnerThread;
}

public synchronized void freeMyMutex() {
    if (OwnerThread == Thread.currentThread()) {
        if (--lockCount == 0) {
            OwnerThread = null;
            notify();
        }
    }
}
```

Abb. 6.31. Realisierung der Synchronisationsklasse `MyMutex` mit Hilfe des Java-Signalmechanismus `wait()` und `notify()`, ohne aktives Warten.

In der Klasse `Object` werden Varianten von `wait()` zur Verfügung gestellt, die die Angabe einer maximalen Wartezeit in Millisekunden bzw. zusätzlichen Nanosekunden erlauben:

```
void wait (long msecs)
void wait (long msecs, int nanos)
```

Diese beiden Varianten haben den gleichen Effekt wie `wait()` ohne Parameter mit dem Unterschied, dass die Blockierung des Threads automatisch aufgehoben wird, sobald das als Parameter angegebene Zeitintervall `msecs` abgelaufen ist. Da diese beiden Varianten ebenfalls in einem `synchronized` Block oder einer `synchronized` Methode stehen müssen, versucht ein wegen des Ablaufs des Zeitintervalls aufgeweckter Thread nach dem Aufwecken zuerst, die Kontrolle über die implizite Mutexvariable des Objektes zu erhalten. Wenn dies nicht gelingt, wird er bezüglich dieser Mutexvariable blockiert. Durch die daraus eventuell resultierende Wartezeit besteht keine Garantie dafür, dass der vorher blockierte Thread nach Ablauf des angegebenen Zeitintervalls tatsächlich wieder zur Ausführung kommt. Es kann keine Obergrenze für die zusätzliche Wartezeit angegeben werden. Bei Verwendung von `wait(msecs)` oder `wait(msecs, nanos)` gibt es für den aufgeweckten Thread keine Möglichkeit festzustellen, ob er durch Ablauf des angegebenen Zeitintervalls oder durch Aufruf von `notify()` durch einen anderen Thread aufgeweckt wurde. Die Aufrufe `wait(0)` bzw. `wait(0,0)` sind äquivalent zum Aufruf `wait()` ohne Parameter.

Ein durch einen Aufruf von `wait()`, `sleep()` oder `join()` blockierter Thread kann auch dadurch wieder aufgeweckt werden, dass er von einem anderen Thread unterbrochen wird. Dazu steht die Methode

```
void interrupt()
```

der Klasse `Thread` zur Verfügung. Durch Aufruf dieser Methode wird der blockierte Thread mit der Ausnahme `InterruptedException` aufgeweckt. Diese kann dann gemäß der üblichen Regeln für die Ausnahmebehandlung verarbeitet werden. Auf einen nicht blockierten Thread `t` hat der Aufruf von `t.interrupt()` den Effekt, dass das Interrupt-Flag des Threads `t` auf `true` gesetzt wird. Ist das Interrupt-Flag eines Threads `t` auf `true` gesetzt, wird bei einem Aufruf von `wait()`, `join()` oder `sleep()` durch diesen Thread direkt die Ausnahme `InterruptedException` ausgelöst. Ein Thread kann seinen eigenen Interrupt-Status durch Aufruf der statischen Methode

```
static boolean interrupted()
```

der Klasse `Thread` überprüfen. Der Interrupt-Status eines beliebigen Threads kann durch Aufruf der nicht-statischen Methode

```
boolean isInterrupted()
```

für das entsprechende Objekt der Klasse `Thread` abgefragt werden. Es ist zu beachten, dass das Unterbrechen eines Threads mit `interrupt()` nicht unbedingt seine Terminierung nach sich zieht. Für die meisten Anwendungen ist es zwar der Normalfall, dass ein mit `interrupt()` unterbrochener Thread nach einer eventuellen Ausnahmebehandlung terminiert. Ein vorher nicht blockierter Thread kann aber trotz gesetztem Interrupt-Flag weiterarbeiten, um dadurch z.B. vor seiner Terminierung einen konsistenten Zustand zu hinterlassen.

Die Methoden

```
static void sleep (long msecs)
static void sleep (long msecs, int nanos)
```

der Klasse `Thread` suspendieren den ausführenden Thread für das angegebene Zeitintervall. Im Unterschied zu `wait()` muss `sleep()` aber nicht in einem `synchronized` Block stehen. Ein Aufruf von `sleep()` hat auch keinen Einfluß auf eine eventuell vom ausführenden Thread gesperrte implizite Mutexvariable eines Objektes. Wenn `sleep()` in einem `synchronized` Block steht, führt der Aufruf von `sleep()` also nicht zur impliziten Freigabe der Mutexvariable des zugehörigen Objektes, und die Mutexvariable bleibt in diesem Fall während der Wartezeit des Threads gesperrt. Nach Ablauf des Zeitintervalls muss der ausführende Thread, im Unterschied zu `wait()`, also auch nicht versuchen, die Kontrolle über die Mutexvariable des Objektes zu erhalten und der ausführende Thread wird direkt ausführungsbereit.

Die Methoden `wait()` und `notify()` sind nicht-statische Methoden der Klasse `Object` und statische Methoden können daher `wait()` und `notify()` nicht direkt aufrufen, da es für statische Methoden keine zugehörige Objektreferenz gibt. Um `wait()` bzw. `notify()` in statischen Methoden verwenden zu können, muss ein zusätzliches Objekt erzeugt werden, bezüglich dem die Synchronisation in Form von `wait()` und `notify()` durchgeführt werden kann. Dies kann ein beliebiges Objekt der Klasse `Object` sein, aber auch das `Class`-Objekt der Klasse, in der die zu synchronisierenden statischen Methoden enthalten sind. Dies ist in Abbildung 6.32 am Beispiel einer Klasse mit zwei statischen Methoden illustriert.

```
public class MyStaticClass {
    public static void staticWait() throws InterruptedException {
        synchronized(MyStaticClass.class) {
            MyStaticClass.class.wait();
        }
    }
    public static void staticNotify() {
        synchronized(MyStaticClass.class) {
            MyStaticClass.class.notify();
        }
    }
}
```

Abb. 6.32. Beispiel zur Synchronisation statischer Methoden mit `wait()` und `notify()`.

6.3.4 Erweiterte Java-Synchronisationsmuster

Die vorgestellten Synchronisationsmechanismen für Java-Threads können dazu verwendet werden, komplexere Synchronisationsmuster zu realisieren, die häufig in parallelen Anwendungsprogrammen eine Rolle spielen.

Barrier-Synchronisation. Dies wird zuerst am Beispiel einer Barrier-Synchronisation gezeigt, vgl. auch [115]. Eine Barrier-Synchronisation realisiert einen festgelegten Synchronisationspunkt, an dem ein Thread wartet, bis alle bzw. eine Teilmenge der Threads des Programms diesen Synchronisationspunkt erreicht haben. Dieses Synchronisationsmuster kann in Form einer Klasse **Barrier** mit Hilfe des **wait()/notify()**-Signal-Mechanismus realisiert werden, vgl. Abbildung 6.33. Die dort definierte **Barrier**-Klasse enthält neben einem Konstruktor, der die Initialisierung eines Barrier-Objektes mit einer Anzahl **n** von Threads vornimmt, die Methode **waitForRest()**, die von jedem an der Barrier-Synchronisation teilnehmenden Thread aufgerufen werden soll. In dieser Methode ruft jeder am Synchronisationspunkt ankommende Thread **wait()** auf und blockiert dadurch bezüglich des **Barrier**-Objektes. Der letzte ankommende Thread weckt alle blockierten Threads mittels **notifyAll()** auf. Objekte der so definierten Klasse **Barrier** können nur einmal benutzt werden, so dass für jede Barrier-Synchronisation ein separates Objekt verwendet werden muss.

```
public class Barrier() {
    private int t2w4;
    private InterruptedException e;
    public Barrier(int n) {
        this.t2w4 = n; }
    public synchronized int waitForRest()
        throws InterruptedException {
        int nThreads = --t2w4;
        if (e != null) throw e;
        if (t2w4 <= 0) {
            notifyAll(); return nThreads; }
        while (t2w4 > 0) {
            if (e != null) throw e;
            try { wait(); }
            catch(InterruptedException e) { this.e = e; notifyAll(); }
        }
        return nThreads;
    }
}
```

Abb. 6.33. Verwendung des **wait()/notify()**-Mechanismus zur Realisierung einer Barrier-Synchronisation.

```

public class ProcessIt implements Runnable {
    String is[];
    Barrier bpStart, bp1, bp2, bpEnd;
    public ProcessIt(String sources[]) {
        is = sources;
        bpStart = new Barrier(sources.length);
        bp1 = new Barrier(sources.length);
        bp2 = new Barrier(sources.length);
        bpEnd = new Barrier(sources.length);
        for (int i = 0; i < sources.length; i++)
            new Thread(this).start();
    }
    public void run() {
        try {
            int i = bpStart.waitForRest();
            doPhase1(is[i]);
            bp1.waitForRest();
            doPhase2(is[i]);
            bp2.waitForRest();
            doPhase3(is[i]);
            bpEnd.waitForRest();
        }
        catch (InterruptedException e)
        {}
    }
    public static void main(String args[]) {
        ProcessIt pi = new ProcessIt(args);
    }
}

```

Abb. 6.34. Beispiel für die Verwendung der Barrier-Klasse

Ein Anwendungsbeispiel für den Einsatz der **Barrier**-Klasse ist in Abbildung 6.34 dargestellt. Das Programm illustriert eine in drei Phasen organisierte Berechnungsfolge, wobei die einzelnen Phasen durch Barrier-Synchronisation voneinander getrennt sind. Jeder der im Konstruktor der Klasse **ProcessIt** erzeugten Threads führt nacheinander alle drei Phasen durch.

Bedingungsvariablen. Der von Java zur Verfügung gestellte Mechanismus mit **wait()** und **notify()** zeigt gewisse Ähnlichkeiten zu dem von Pthreads in Form von Bedingungsvariablen zur Verfügung gestellten Mechanismus, vgl. Abschnitt 6.2.2 Seite 294. Der Unterschied liegt im Wesentlichen darin, dass die für Java vorgenommene Integration in die Klasse **Object** auch eine Bindung an die implizite Mutexvariable des zugehörigen Objektes vornimmt. Dies macht eine explizite Zuordnung einer Bedingungsvariablen an eine Mutexvariable, die für Pthreads notwendig ist, überflüssig und sichert dadurch eine einfache Benutzung. Die feste Zuordnung zwischen Synchronisations- und Signalmechanismus reduziert jedoch auch die Flexibilität der Handhabung, da keine beliebige Kopplung von Mutexvariable und Warteliste eines Objektes möglich ist. Ein Java-Thread muss zum Zeit-

punkt des Aufrufs von `wait()` bzw. `notify()` Eigentümer der Mutexvariablen des Objektes sein, bzgl. dem `wait()` bzw. `notify()` aufgerufen wird. Es ist damit nicht möglich, dieselbe Mutexvariable für die Synchronisation der Wartelisten mehrerer Objekte zu verwenden. Dies wäre jedoch bei der Implementierung von Produzenten- und Konsumenten-Threads mit einem gemeinsamen Datenpuffer nützlich, vgl. z.B. Abbildung 6.36. Der Signalmechanismus von Java kann aber zur Realisierung einer Klasse `CondVar` verwendet werden, die den Bedingungsvariablen-Mechanismus von Pthreads nachbildet, vgl. auch [115, 100]. Abbildung 6.35 zeigt eine Realisierung. Die von `CondVar` zur Verfügung gestellten Methoden `cvWait()`, `cvSignal()` und `cvBroadcast()` bilden den Mechanismus von `pthread_cond_wait()`, `pthread_cond_signal()` und `pthread_cond_broadcast()` nach und erlauben die Verwendung einer beliebigen Mutexvariablen, die als Parameter vom Typ `MyMutex` zur Verfügung gestellt wird, zur Synchronisation. Damit kann eine einzelne Mutexvariable vom Typ `MyMutex` für die Synchronisation mehrerer Bedingungsvariablen vom Typ `CondVar` verwendet werden. Eine Blockierung durch Aufruf von `cvWait()` erfolgt bezüglich des Objektes vom Typ `CondVar`. Die Klasse `CondVar` sichert auch eine einfache Portierung von Pthread-Programmen mit Bedingungsvariablen auf Java-Threads.

Abbildung 6.36 enthält als Beispiel die Realisierung eines Puffermechanismus für Produzenten- und Konsumenten-Threads mit Hilfe der Klasse `CondVar`, vgl. auch [100]. Ein Produzenten-Thread kann mit Hilfe der Methode `put()` Objekte in dem Puffer ablegen, ein Konsumenten-Thread kann mit Hilfe von `take()` Objekte aus dem Puffer entnehmen. Die beiden Bedingungsvariablen `notFull` und `notEmpty` benutzen dieselbe Mutexvariable `mutex` zur Synchronisation.

6.3.5 Thread-Scheduling in Java

Ein Java-Programm besteht typischerweise aus mehreren Threads, die auf einem oder mehreren Prozessoren ausgeführt werden. Die ausführungsbereiten Threads konkurrieren dabei um die Ausführung auf einem freiwerdenden Prozessor. Die jeweilige Zuordnung von Threads an Prozessoren wird vom Scheduler der JVM durchgeführt. Der Programmierer kann die Zuordnung von Threads an Prozessoren dadurch beeinflussen, dass er Threads Prioritäten zuordnet. Die minimalen, maximalen und Default-Prioritäten von Java-Threads sind in statischen Konstanten der Klasse `Thread` festgelegt:

```
public static final int MIN_PRIORITY    //üblicherweise 1
public static final int MAX_PRIORITY    //üblicherweise 10
public static final int NORM_PRIORITY   //üblicherweise 5
```

Dabei entspricht ein *großer* Prioritätswert einer *hohen* Priorität. Der die `main()`-Methode einer Klasse ausführende Hauptthread hat per Default die Priorität `Thread.NORM_PRIORITY`. Ein neu erzeugter Thread hat per Default

```

public class CondVar {
    private MyMutex syncVar; /* verwende Mutex zur Synchronisation */
    public CondVar() {
        this(new MyMutex());
    }
    public CondVar(MyMutex sv) {
        syncVar = sv;
    }
    public void cvWait() throws InterruptedException {
        cvWait(syncVar, 0);
    }
    public void cvWait(MyMutex sv) throws InterruptedException {
        cvWait(sv, 0);
    }
    public void cvWait(int millis) throws InterruptedException {
        cvWait(syncVar, millis);
    }
    public void cvWait(MyMutex sv, int millis)
        throws InterruptedException {
        int i = 0;
        InterruptedException exception;
        synchronized (this) {
            if (sv.getMutexOwner() != Thread.currentThread())
                throw new IllegalMonitorStateException ("thread not owner");
            while (sv.getMutexOwner() == Thread.currentThread()) {
                i++; sv.freeMyMutex();
            }
            try { if (millis == 0) wait(); else wait(millis); }
            catch (InterruptedException e) { exception = e; }
        }
        for (; i > 0; i--) sv.getMyMutex();
        if (exception != null) throw exception;
    }
    public void cvSignal() {
        cvSignal(syncVar);
    }
    public synchronized void cvSignal(MyMutex sv) {
        if (sv.getMutexOwner() != Thread.currentThread())
            throw new IllegalMonitorStateException ("thread not owner");
        notify();
    }
    public void cvBroadcast() {
        cvBroadcast(syncVar);
    }
    public synchronized void cvBroadcast(MyMutex sv) {
        if (sv.getMutexOwner() != Thread.currentThread())
            throw new IllegalMonitorStateException("thread not owner");
        notifyAll();
    }
}

```

Abb. 6.35. Klasse zur Realisierung eines Bedingungsvariablen-Mechanismus mit dem Java-Signalmechanismus.

```

class PThreadsStyleBuffer {
    private final MyMutex mutex = new MyMutex();
    private final CondVar notFull = new CondVar(mutex);
    private final CondVar notEmpty = new CondVar(mutex);
    private int count = 0;
    private int takePtr = 0;
    private int putPtr = 0;
    private final Object[] array;

    public PThreadsStyleBuffer(int capacity) {
        array = new Object[capacity];
    }

    public void put(Object x) throws InterruptedException {
        mutex.getMyMutex();
        try {
            while (count == array.length)
                notFull.cvWait();

            array[putPtr] = x;
            putPtr = (putPtr + 1) % array.length;
            ++count;
            notEmpty.cvSignal();
        }
        finally {
            mutex.freeMyMutex();
        }
    }

    public Object take() throws InterruptedException {
        Object x = null;
        mutex.getMyMutex();
        try {
            while (count == 0);
            notEmpty.cvWait();

            x = array[takePtr];
            array[takePtr] = null;
            takePtr = (takePtr + 1) % array.length;
            --count;
            notFull.cvSignal();
        }
        finally {
            mutex.freeMyMutex();
        }
        return x;
    }
}

```

Abb. 6.36. Realisierung eines Puffermechanismus für Produzenten- und Konsumenten-Threads.

die gleiche Priorität wie der erzeugende Thread. Die aktuelle Priorität eines Threads kann mit Hilfe der Methoden

```
public int getPriority()
public int setPriority(int prio)
```

abgefragt bzw. dynamisch geändert werden.

Gibt es mehr ausführungsbereite Threads als Prozessoren, bringt der Scheduler vorzugsweise Threads mit einer höheren Priorität zur Ausführung. Der exakte Mechanismus zur Auswahl der auszuführenden Threads kann von der speziellen Implementierung der JVM abhängen. Die Programmiersprache Java legt keinen genauen Mechanismus für das Scheduling fest, um die Flexibilität der Realisierung der JVM auf verschiedenen Plattformen und Betriebssystemen nicht zu beeinträchtigen. Der Scheduler der JVM kann z.B. immer den Thread mit der höchsten Priorität zur Ausführung bringen, er kann aber auch einen Alterungsmechanismus integrieren, der sicherstellt, dass auch Threads mit geringerer Priorität ab und zu zur Ausführung kommen.

Da das genaue Scheduling von Threads unterschiedlicher Priorität nicht festgelegt ist, können Prioritäten nicht dazu verwendet werden, Synchronisationsmechanismen zu ersetzen. Prioritäten können nur dazu verwendet werden, die relative Wichtigkeit der verschiedenen Threads auszudrücken, damit im Zweifelsfall zuerst die wichtigsten Threads ausgeführt werden.

Bei Verwendung von Prioritäten kann auch bei Java-Threads das Problem der Prioritätsinversion auftreten, vgl. Abschnitt 6.2.7. Eine Prioritätsinversion tritt auf, wenn ein Thread hoher Priorität blockiert und auf einen Thread niedriger Priorität wartet, weil dieser z.B. eine Mutexvariable gesperrt hat. Der Thread niedriger Priorität kann aber von einem Thread mittlerer Priorität am Weiterarbeiten und an der Freigabe der Mutexvariable gehindert werden mit dem Effekt, dass der Thread hoher Priorität eventuell lange Zeit blockiert. Das Problem der Prioritätsinversion kann, wie in Abschnitt 6.2.7 beschrieben, durch Verwendung von Prioritätsvererbung gelöst werden: wenn ein Thread hoher Priorität blockiert, wird die Priorität des Threads, der das kritische Objekt zur Zeit kontrolliert auf die Priorität des Threads hoher Priorität angehoben. Damit kann kein Thread mittlerer Priorität den Thread hoher Priorität vom Weiterarbeiten abhalten. Viele JVM setzen daher diese Methode ein; dies ist jedoch nicht vom Java-Standard festgelegt.

6.4 OpenMP

OpenMP ist eine Spezifikation von Übersetzerdirektiven, Bibliotheksfunktionen und Umgebungsvariablen, die von einer Gruppe führender Software- und Hardwarehersteller mit dem Ziel entworfen wurde, einen einheitlichen Standard für die Programmierung von Parallelrechnern mit gemeinsamem Adressraum zur Verfügung zu stellen [116]. Unterstützt werden Schnittstellen (API, *application programming interface*) für C, C++ und FORTRAN.

OpenMP erweitert diese sequentiellen Sprachen um Konstrukte zur SPMD-Programmierung, zur Aufteilung von Arbeit, zur Synchronisation und zur Deklaration von gemeinsamen (*shared*) und privaten (*private*) Variablen. Die Auswahl der Konstrukte ist auf den Anwendungsbereich des wissenschaftlichen Rechnens ausgerichtet, obwohl natürlich auch andere Anwendungen in OpenMP realisiert werden können.

Das Programmiermodell von OpenMP basiert auf parallel arbeitenden **Threads**, die nach einem **fork-join**-Prinzip erzeugt und beendet werden. Die Abarbeitung eines mit Hilfe von OpenMP formulierten Programms beginnt mit der Ausführung *eines* sogenannten *Master-Threads*, der das Programm sequentiell ausführt, bis das erste **parallel**-Konstrukt auftritt. Bei Auftreten dieses Konstrukts, das wir weiter unten näher beschreiben, erzeugt der Master-Thread ein *Team von Threads* und wird zum Master des Teams (*fork*). Alle Threads des Teams, zu dem auch der Master selber gehört, führen das auf das **parallel**-Konstrukt folgende Programmstück, den *parallelen Bereich*, parallel zueinander aus, indem entweder alle Threads des Teams den gleichen Programmtext mit evtl. unterschiedlichen privaten Variablen im SPMD-Stil abarbeiten oder indem die Arbeit explizit durch geeignete Konstrukte auf die Threads verteilt wird. Dabei wird ein gemeinsamer Adressraum für das Gesamtprogramm zugrundegelegt, d.h. wenn ein Mitglied des Teams eine Datenstruktur ändert, ist die Änderung nicht nur für die anderen Mitglieder des Teams, sondern für alle anderen Threads des Programms sichtbar. Nach Beendigung der Abarbeitung des parallel auszuführenden Programmstückes werden die Threads des Teams synchronisiert und nur der Master des Teams wird weiter ausgeführt, die anderen Threads werden beendet (*join*).

Mit den zur Verfügung stehenden Mechanismen zur Steuerung der Parallelität können Programme formuliert werden, die sowohl sequentiell als auch parallel ausgeführt werden können. Dabei ist es jedoch auch möglich, Programme zu schreiben, die nur bei einer parallelen Ausführung das gewünschte Ergebnis errechnen. Der Programmierer ist dafür verantwortlich, dass die Programme korrekt arbeiten. Dies gilt auch für die Vermeidung von Konflikten, Deadlocks oder zeitkritischen Abläufen, obwohl die OpenMP-Konstrukte zur Kontrolle der Parallelität auf abstrakterem Niveau ansetzen als etwa die Mechanismen des Pthreads-Standards.

Wir geben im Folgenden einen kurzen Überblick über den OpenMP-Standard und verwenden für die Darstellung die C-Schnittstellen. Weiterführende Informationen können in [116, 22] nachgelesen und über die OpenMPWebseite (<http://www.openmp.org>) erhalten werden.

6.4.1 Steuerung der parallelen Abarbeitung

Die meisten Mechanismen zur Steuerung der parallelen Abarbeitung von Programmteilen werden in OpenMP durch Übersetzer-Direktiven zur Verfügung gestellt, deren Syntax auf den in C und C++ verwendeten **pragma**-Direktiven basiert. Zusätzlich stehen Laufzeitfunktionen zur Steuerung des

Verhaltens der Direktiven zur Verfügung. Eine OpenMP-Direktive hat die Form

```
#pragma omp parallel [Parameter [Parameter] ... ]
```

und muss in einer eigenen Programmzeile stehen. Die Parameter einer Direktive (in OpenMP auch *clause* genannt) können dabei in einer beliebigen Reihenfolge auftreten. Jede Direktive wirkt nur auf die der Direktive direkt folgende Anweisung. Sollen mehrere Anweisungen von der Direktive gesteuert werden, so müssen diese zwischen { und } stehen und so in einem Anweisungsblock zusammengefasst sein. Im folgenden stellen wir einige dieser OpenMP-Direktiven vor.

Paralleler Bereich. Die wichtigste Direktive zur Steuerung der Parallelität ist die `parallel`-Direktive mit der folgenden Syntax:

```
#pragma omp parallel [Parameter [Parameter] ... ]  
Anweisungsblock
```

Diese Direktive bewirkt, dass der angegebene Anweisungsblock parallel ausgeführt wird. Wird die Arbeit nicht explizit verteilt, so führen alle Threads die gleichen Berechnungen mit evtl. unterschiedlichen privaten Daten im SPMD-Stil aus. Der parallel ausgeführte Anweisungsblock wird auch als **paralleler Bereich** bezeichnet. Zur parallelen Abarbeitung wird ein *Team von Threads* erzeugt, dessen Master der die Direktive ausführende Thread ist. Die genaue Anzahl der zu erzeugenden Threads wird, wie wir weiter unten näher beschreiben werden, über Laufzeitfunktionen oder Umgebungsvariablen festgelegt. Nach der Erzeugung des Teams bleibt die Anzahl der Threads, die den Anweisungsblock ausführen, konstant. Für verschiedene parallele Bereiche können jedoch verschiedene Thread-Anzahlen verwendet werden.

Ein paralleler Bereich wird von allen Threads des erzeugten Teams einschließlich des Master-Threads gemeinsam abgearbeitet. Dabei können *gemeinsame* und *private* Variablen der beteiligten Threads über die *Parameter* der `parallel`-Direktive definiert werden. Private Variablen der Threads werden durch den `private`-Parameter der Form

```
private(list_of_variables)
```

spezifiziert, wobei `list_of_variables` eine beliebige Liste von bereits deklarierten Programmvariablen ist. Der Effekt besteht darin, dass auf dem Laufzeitstack jedes Threads des Teams eine uninitialisierte Kopie der angegebenen Variablen angelegt wird, die nur dieser Thread während seiner Ausführung als globale Variable zugreifen und manipulieren kann. Gemeinsame Variablen der Threads eines Teams werden durch den `shared`-Parameter der Form

```
shared(list_of_variables)
```

spezifiziert. Der Effekt besteht darin, dass jeder Thread des Teams beim Lesen oder Beschreiben der angegebenen Variablen auf den gleichen Datenbereich zugreift.

Mit Hilfe des `default`-Parameters kann der Programmierer festlegen, ob die Programmvariablen des `parallel`-Konstrukts per Default gemeinsame oder private Variablen sind. Die Angabe

```
default(shared)
```

bewirkt, dass alle außer den vom `private`-Parameter explizit angegebenen Programmvariablen gemeinsame Variablen der Threads des Teams sind. Die Angabe

```
default(none)
```

bewirkt, dass jede in dem parallelen Bereich verwendete Variable explizit über einen `shared`- oder `private`-Parameter als gemeinsame oder private Variable gekennzeichnet sein muss.

Beispiel: Das Programmfragment in Abbildung 6.37 zeigt die Verwendung einer `parallel`-Direktive zur parallelen Verarbeitung eines Feldes `x`. Für OpenMP-Programme muss die Datei `<omp.h>` eingebunden werden. Eine explizite Initialisierung von OpenMP wird im Programm nicht vorgenommen.

Wir nehmen an, dass die zu verarbeitenden Werte in der Funktion `initialize()` vom Master-Thread eingelesen werden. In der `parallel`-Direktive werden die Variablen `x` und `npoints` als gemeinsame Variable der den parallelen Bereich ausführenden Threads spezifiziert. Die restlichen Variablen `iam`, `np` und `mypoints` sind private Variablen. Zu Beginn der Ausführung des parallelen Bereiches bestimmt jeder beteiligte Thread durch den Aufruf der Funktion `np = omp_get_num_threads()` die Gesamtanzahl der Threads des Teams. Durch den Aufruf der Funktion `omp_get_thread_num()` erhält jeder Thread des Teams eine Nummer zurück, die als Thread-Name dient und im Beispiel in `iam` gespeichert wird. Der Master-Thread hat die Thread-Nummer 0, die Thread-Nummern der anderen Threads liegen fortlaufend zwischen 1 und `np-1`. Jeder der Threads ruft die Funktion `compute_subdomain()` auf, in der die Einträge des Feldes `x` verarbeitet werden. Beim Aufruf von `compute_subdomain()` wird neben dem Feldnamen `x` and dem Thread-Namen `iam` auch die Anzahl `mypoints` der von jedem Thread zu verarbeitenden Feldelemente angegeben. Welche Feldelemente dies speziell sind, ist innerhalb von `compute_subdomain` anzugeben.

□

Nach Abarbeitung eines parallelen Bereiches werden alle Threads des Teams außer dem Master-Thread terminiert. Anschließend führt der Master-Thread die dem parallelen Bereich folgenden Anweisungen alleine aus. Das Ende eines parallelen Bereiches stellt somit einen impliziten Synchronisationspunkt dar.

```

#include <stdio.h>
#include <omp.h>

int npoints, iam, np, mypoints;
double *x;

int main() {
    scanf("%d", &npoints);
    x = (double *) malloc(npoints * sizeof(double));
    initialize();
    #pragma omp parallel shared(x,npoints) private(iam,np,mypoints)
    {
        np = omp_get_num_threads();
        iam = omp_get_thread_num();
        mypoints = npoints / np;
        compute_subdomain(x, iam, mypoints);
    }
}

```

Abb. 6.37. Parallele Verarbeitung einer Datenstruktur mit Hilfe einer OpenMP `parallel`-Direktive.

Prinzipiell können parallele Bereiche geschachtelt werden, d.h. in einem parallelen Bereich kann eine weitere `parallel`-Direktive auftreten. Per Default wird der innere parallele Bereich von einem Team ausgeführt, dem *nur* der Thread angehört, der die innere `parallel`-Direktive ausführt. Der Default kann durch Aufruf der Bibliotheksfunktion

```
void omp_set_nested(int nested)
```

mit `nested != 0` geändert werden. In diesem Fall kann der die geschachtelte `parallel`-Direktive ausführende Thread ein Team mit mehr als einem Thread erzeugen. Die genaue Anzahl der in diesem Fall erzeugten Threads ist implementierungsabhängig.

6.4.2 Parallele Schleife

Innerhalb eines parallelen Bereiches können die durchzuführenden Berechnungen mit Hilfe von speziellen Direktiven zur Verteilung der Arbeit auf die ausführenden Threads verteilt werden. Die wichtigste Direktive zur Verteilung der Arbeit ist die `for`-Direktive mit der folgenden Syntax:

```

#pragma omp for [Parameter [Parameter] ... ]
for (i = lower_bound; i op upper_bound; incr_expr) {
    Schleifenrumpf
}

```

Die **for**-Schleife ist auf solche Schleifen beschränkt, bei denen sichergestellt ist, dass die durch den Schleifenrumpf gegebenen Berechnungen der verschiedenen Iterationen unabhängig voneinander sind und die Gesamtzahl der Iterationen beim Betreten der **for**-Schleife im voraus bestimmt werden kann. Der Effekt der **for**-Direktive besteht darin, dass die einzelnen Iterationen der Schleife auf die den umgebenden parallelen Bereich ausführenden Threads verteilt und unabhängig berechnet werden. Es soll sich also um eine **parallele Schleife** fester Länge handeln. Die Variable **i** bezeichnet eine Integervariable, die im Rumpf der Schleife nicht verändert werden darf und die innerhalb der Schleife als private Variable des die zugehörige Iteration der **for**-Schleife ausführenden Threads behandelt wird. **lower_bound** und **upper_bound** bezeichnen Integerausdrücke, deren Werte durch Ausführung der Schleife nicht geändert werden, **op** bezeichnet einen booleschen Operator, also $\text{op} \in \{ <, <=, >, >= \}$. Der Inkrementierungs-Ausdruck **incr_expr** kann folgende Formen annehmen:

```
++i, i++, --i, i--, i += incr, i -= incr,
i = i + incr, i = incr + i, i = i - incr,
```

wobei **incr** ebenfalls ein schleifenunabhängiger Integerausdruck ist.

Die Aufteilung der Schleifeniterationen auf die ausführenden Threads kann durch den **schedule**-Parameter gesteuert werden. Folgende Steuerungsmöglichkeiten sind vorgesehen:

- **schedule(static, block_size)**. Diese Parameter-Angabe bedeutet, dass eine statische Aufteilung der Iterationen auf die Threads verwendet wird, indem die Iterationen in Blöcken der Größe **block_size** reihum (*round-robin*) auf die Threads verteilt werden. Ist keine Blockgröße angegeben, erhält jeder Thread einen Block fortlaufender Iterationen ungefähr gleicher Größe, d.h. es wird eine *blockweise* Verteilung verwendet.
- **schedule(dynamic, block_size)**. Diese Parameter-Angabe bedeutet, dass eine dynamische Zuteilung von Iterationsblöcken an die Threads vorgenommen wird, d.h. nach Abarbeitung der zugewiesenen Iterationen erhält ein Thread dynamisch einen neuen Block mit **block_size** Iterationen zugeteilt. Ist keine Blockgröße angegeben, werden dynamisch einzelne Iterationen zugeteilt, d.h. es wird die Blockgröße 1 verwendet.
- **schedule(guided, block_size)**. Diese Parameter-Angabe bedeutet, dass ein dynamisches Scheduling mit abnehmender Blockgröße verwendet wird. Für die Angabe **block_size = 1** wird jedem Thread, der seine zugewiesenen Iterationen beendet hat, dynamisch ein neuer Block von Iterationen zugewiesen, deren Größe sich aus dem Quotient der noch nicht bearbeiteten Iterationen und der Anzahl der Threads ergibt, so dass die Blockgröße der zugewiesenen Iterationen linear mit der Anzahl der ausgeführten Iterationen abnimmt. Für die Angabe **block_size = k** mit $k > 1$ nimmt die Blockgröße exponentiell zu k ab, der letzte Block kann jedoch eine kleinere Größe haben. Die Angabe **block_size** gibt also die minimale Blockgröße

an, die (bis auf die eben erwähnte Ausnahme) gewählt werden kann. Ist kein Wert für `block_size` angegeben, wird als Defaultwert 1 verwendet.

- `schedule(runtime)`. Diese Parameter-Angabe bedeutet, dass das Scheduling der Threads zur Laufzeit des Programms festgelegt wird. Dies kann dadurch geschehen, dass vor dem Start des Programms die Umgebungsvariable `OMP_SCHEDULE` durch Angabe von Scheduling-Typ und Blockgröße gesetzt wird, also beispielsweise als

```
setenv OMP_SCHEDULE "dynamic, 4"
setenv OMP_SCHEDULE "guided"
```

Wird dabei keine Blockgröße angegeben, wird der Defaultwert verwendet. Außer für das statische Scheduling (`static`) ist dies `block_size = 1`. Wenn die Umgebungsvariable `OMP_SCHEDULE` nicht gesetzt ist, hängt das verwendete Scheduling von der Implementierung der OpenMP-Bibliothek ab.

Fehlt die Angabe eines `schedule`-Parameters bei der `for`-Direktive, wird ein Default-Schedulingverfahren verwendet, das von der Implementierung der OpenMP-Bibliothek abhängt. Die einer `for`-Direktive zugeordnete parallele Schleife darf nicht durch eine `break`-Anweisung beendet werden. Am Ende der parallelen Schleife findet eine implizite Synchronisation der beteiligten Threads statt, d.h. die der parallelen Schleife folgenden Anweisungen werden erst ausgeführt, wenn alle beteiligten Threads die parallele Schleife beendet haben. Diese Synchronisation kann durch die Angabe eines `nowait`-Parameters in der Parameterliste der `for`-Direktive vermieden werden.

Beispiel: Abbildung 6.38 zeigt als Beispiel für die Anwendung einer `for`-Direktive eine Programmskizze zur Realisierung einer **Matrix-Matrix-Multiplikation** in OpenMP. Der parallele Bereich des Programms besteht aus zwei Phasen, die durch eine implizite Synchronisation voneinander getrennt sind. In der ersten Phase wird die Ergebnismatrix `MC` mit 0 initialisiert, in der zweiten Phase wird die eigentliche Matrix-Multiplikation durchgeführt. Die Aufteilung der Berechnung der Ergebnismatrix auf die auszuführenden Threads erfolgt durch ein statisches Scheduling, wobei jeder Thread einen *Block von Zeilen* initialisiert bzw. berechnet. Da jeder Eintrag und damit auch jede Zeile der Ergebnismatrix gleichen Berechnungsaufwand hat, ist ein solches statisches Scheduling sinnvoll. Bei der Berechnung der Ergebnismatrix `MC` in der zweiten Phase besteht jeder Schleifenrumpf der durch die `for`-Direktive bezeichneten parallelen Schleife aus einer doppelten (sequentiellen) Schleife, wobei die äußere Schleife über die Einträge der jeweiligen Zeile läuft und die innere Schleife zur Berechnung des Skalarproduktes dient. Die Angabe eines `nowait`-Parameters sollte bei der ersten parallelen Schleife vermieden werden, da nicht sichergestellt ist, dass die Verteilung der Zeilen auf die Threads bei beiden Schleifen nach dem gleichen Schema erfolgt, obwohl dies der Normalfall sein sollte. □

```

#include <omp.h>

double MA[100][100], MB[100][100], MC[100][100];
int i, row, col, size = 100;

int main() {
    read_input(MA, MB);
    #pragma omp parallel shared(MA,MB,MC,size) private(row,col,i)
    {
        #pragma omp for schedule(static)
        for (row = 0; row < size; row++) {
            for (col = 0; col < size; col++)
                MC[row][col] = 0.0;
        }
        #pragma omp for schedule(static)
        for (row = 0; row < size; row++) {
            for (col = 0; col < size; col++)
                for (i = 0; i < size; i++)
                    MC[row][col] += MA[row][i] * MB[i][col];
        }
    }
    write_output(MC);
}

```

Abb. 6.38. OpenMP-Programm zur parallelen Berechnung einer Matrix-Multiplikation unter Verwendung eines parallelen Bereiches mit einem Anweisungsblock aus zwei aufeinanderfolgenden parallelen Schleifen.

Das Schachteln von `for`-Direktiven innerhalb eines parallelen Bereiches ist nicht erlaubt. Zur Schachtelung paralleler Schleifen müssen also auch die parallelen Bereiche so geschachtelt werden, dass in jedem parallelen Bereich höchstens eine `for`-Direktive ist.

Beispiel: Eine parallele Matrix-Multiplikation mit einer doppelten parallelen Schleife ist in Abbildung 6.39 skizziert. In dieser Variante wird die Schleife für die Berechnung der Zeilen und die Schleife für die Berechnung der Spalten jeweils parallel ausgeführt, indem der Schleifenrumpf der ersten `for`-Direktive den parallelen Bereich und die `for`-Direktive der zweiten Schleife enthält. Jeder Eintrag der Ergebnismatrix `MC` kann nun von einem anderen Thread berechnet werden. $\square$

Die Programmskizze in Abbildung 6.39 korrespondiert damit zur parallelen Matrix-Multiplikation im Pthreads-Programm in Abbildung 6.5. Ein wesentlicher Unterschied liegt darin, dass im Pthreads-Programm die Steuerung der Threads explizit erfolgt, d.h. alle beteiligten Threads werden im Pthreads-Programm explizit gestartet. Im OpenMP-Programm hingegen werden die Threads nicht explizit gestartet und die genaue Realisierung der verschachtelten parallelen Bereiche wird von der OpenMP-Bibliothek übernommen.

```

#include <omp.h>

double MA[100][100], MB[100][100], MC[100][100];
int i, row, col, size = 100;

int main() {
    read_input(MA, MB);
    #pragma omp parallel private(row,col,i)
    {
        #pragma omp for schedule(static)
        for (row = 0; row < size; row++) {
            #pragma omp parallel shared(MA, MB, MC, size)
            {
                #pragma omp for schedule(static)
                for (col = 0; col < size; col++) {
                    MC[row][col] = 0.0;
                    for (i = 0; i < size; i++)
                        MC[row][col] += MA[row][i] * MB[i][col];
                }
            }
        }
    }
    write_output(MC);
}

```

Abb. 6.39. OpenMP-Programm zur parallelen Berechnung einer Matrix-Multiplikation mit verschachtelter paralleler Schleife.

Wichtig ist hierbei, dass die Korrektheit des parallelen Programms keine spezielle Zuordnung von Aufgaben an Threads erfordert, dass also für verschiedene Zuordnungen eine korrekte Abarbeitung resultiert (thread-sicher).

Ein Vorteil des OpenMP-Ansatzes besteht darin, dass der Programmierer nicht darauf achten muss, dass eine Systemlimitierung, wie z.B. die Anzahl der erzeugbaren Threads, überschritten wird. Für die angegebene Matrixgröße 8×8 arbeitet das Pthreads-Programm in Abbildung 6.5 korrekt, die Verwendung der Matrixgröße 100×100 aus Abbildung 6.39 würde aber zum Start von 10000 Threads führen, was für die meisten Pthreads-Implementierungen die maximale Anzahl der erzeugbaren Threads überschreitet.

6.4.3 Nichtiterative parallele Bereiche

Eine nichtiterative Verteilung der innerhalb eines parallelen Bereiches durchzuführenden Berechnung kann durch Verwendung einer **sections**-Direktive erfolgen, deren Syntax wie folgt definiert ist:

```
#pragma omp sections [Parameter [Parameter] ... ]
{
    [#pragma omp section]
    Anweisungsblock
    [#pragma omp section]
    Anweisungsblock
    :
}
}
```

Innerhalb einer **sections**-Direktive werden durch **section**-Direktiven Abschnitte bezeichnet, die unabhängig voneinander sind und daher parallel zueinander von verschiedenen Threads abgearbeitet werden können. Jeder Abschnitt beginnt mit **#pragma omp section** und kann ein beliebiger Anweisungsblock sein. Für den ersten innerhalb der **sections**-Direktive definierten Anweisungsblock kann die Angabe der **section**-Direktive entfallen. Am Ende einer **sections**-Direktive findet eine implizite Synchronisation statt, die durch die Angabe eines **nowait**-Parameters vermieden werden kann.

Einmalige Ausführung. In manchen Situationen ist es sinnvoll, dass bestimmte Anweisungen innerhalb eines parallelen Bereiches nur einmal ausgeführt werden. Dies gilt z.B. für die Ausgabe von Kontrollmeldungen beim Ablauf eines parallelen Programms. In OpenMP kann dies durch die Verwendung von **single**-Direktiven innerhalb eines parallelen Bereiches erreicht werden. Die Syntax ist:

```
#pragma omp single [Parameter [Parameter] ... ]
Anweisungsblock
```

Der angegebene Anweisungsblock wird nur von *einem* Thread des Teams, das dem parallelen Bereich zugeordnet ist, ausgeführt. Am Ende des Anweisungsblockes findet eine implizite Synchronisation statt, die auch hier durch Angabe eines **nowait**-Parameters verhindert werden kann.

Syntaktische Abkürzungen. Zur Vereinfachung der Schreibweise führt OpenMP Abkürzungen für parallele Bereiche ein, in denen nur eine einzelne **for**- bzw. **sections**-Direktive enthalten ist. Für einen parallelen Bereich mit einer einzelnen **for**-Direktive kann die folgende Abkürzung verwendet werden:

```
#pragma omp parallel for [Parameter [Parameter] ... ]
    for(i = lower_bound; i op upper_bound; incr_expr) {
        Schleifenrumpf
    }
```

Dabei sind als Parameter alle für die **parallel**- und für die **for**-Direktive zugelassenen Parameter erlaubt. Analog kann als Abkürzung für eine einzelne in einem parallelen Bereich enthaltene **sections**-Direktive folgendes Konstrukt verwendet werden:


```
#pragma omp parallel sections [Parameter [Parameter] ... ]
{
    [#pragma omp section]
    Anweisungsblock
    [#pragma omp section]
    Anweisungsblock
    :
}
]
```

Thread-Anzahl. Ein paralleler Bereich wird von einer Anzahl von Threads ausgeführt. Der Programmierer hat die Möglichkeit, diese Anzahl über mehrere Laufzeitfunktionen zu beeinflussen. Mit Hilfe der Funktion

```
void omp_set_dynamic (int dynamic_threads)
```

kann der Programmierer die Anpassung der Thread-Anzahl durch das Laufzeitsystem beeinflussen, wobei die Funktion *außerhalb* eines parallelen Bereiches aufgerufen werden muss. Für `dynamic_threads` $\neq 0$ wird die dynamische Anpassung durch das Laufzeitsystem erlaubt, d.h. das Laufzeitsystem kann die Anzahl der Threads, die für nachfolgende parallele Bereiche verwendet werden, an die Systemgegebenheiten anpassen. Während der Ausführung desselben parallelen Bereiches wird die Anzahl der ausführenden Threads aber stets konstant gehalten. Für `dynamic_threads` = 0 wird die dynamische Anpassung der Thread-Anzahl ausgeschaltet, d.h. das Laufzeitsystem verwendet für nachfolgende parallele Bereiche die derzeit eingestellte Anzahl von Threads. Welche der beiden Varianten den Default darstellt, hängt von der speziellen OpenMP-Bibliothek ab. Der Status der Thread-Anpassung kann durch Aufruf der Funktion

```
int omp_get_dynamic (void)
```

abgefragt werden. Der Aufruf liefert 0 zurück, wenn keine dynamische Anpassung vorgesehen ist. Ansonsten wird ein Wert $\neq 0$ zurückgeliefert.

Der Programmierer kann durch Aufruf der Funktion

```
void omp_set_num_threads (int num_threads)
```

die Anzahl der Threads beeinflussen, die für die Ausführung nachfolgender paralleler Bereiche verwendet werden. Auch dieser Aufruf muss außerhalb eines parallelen Bereiches stattfinden. Der genaue Effekt des Aufrufes hängt davon ab, ob die automatische Thread-Anpassung durch das Laufzeitsystem eingeschaltet ist oder nicht. Wenn die automatische Anpassung eingeschaltet ist, gibt `num_threads` die maximale Anzahl von Threads an, die im Folgenden verwendet wird. Wenn keine automatische Anpassung erlaubt ist, gibt `num_threads` die tatsächliche Anzahl von Threads an, die für alle nachfolgenden parallelen Bereiche verwendet wird.

Wie oben dargestellt wurde, erlaubt OpenMP geschachtelte parallele Bereiche. Die Anzahl der zur Abarbeitung verwendeten Threads eines geschachtelten parallelen Bereiches hängt vom Laufzeitsystem ab, kann jedoch vom Programmierer durch Aufruf der Funktion

```
void omp_set_nested (int nested)
```

beeinflusst werden. Für `nested = 0` wird die Abarbeitung des inneren parallelen Bereiches sequentialisiert und nur von *einem* Thread vorgenommen. Dies ist auch die Default-Einstellung. Für `nested  $\neq$  0` wird eine geschachtelte parallele Abarbeitung erlaubt, d.h. das Laufzeitsystem kann zur Abarbeitung des inneren parallelen Bereiches zusätzliche Threads verwenden. Die genaue Abarbeitung hängt auch hier wieder vom Laufzeitsystem ab und kann auch aus der Abarbeitung durch nur *einen* Thread erfolgen. Durch Aufruf der Funktion

```
int omp_get_nested (void)
```

kann der aktuelle Status zur Behandlung von geschachtelten parallelen Bereichen abgefragt werden.

6.4.4 Koordination von Threads

Ein paralleler Bereich wird in OpenMP-Programmen in der Regel von mehreren Threads ausgeführt, deren Zugriff auf gemeinsame Variablen koordiniert werden muss. Zur Koordination stellt OpenMP mehrere Direktiven zur Verfügung, die innerhalb von parallelen Bereichen verwendet werden können. **Kritische Bereiche**, die zu jedem Zeitpunkt nur von jeweils einem Thread ausgeführt werden dürfen, können durch die **critical**-Direktive mit der folgenden Syntax

```
#pragma omp critical [(name)]  
    Anweisungsblock
```

realisiert werden. Der optional anzugebende Name `name` kann dabei zur Identifikation des kritischen Bereiches verwendet werden. Der Effekt der **critical**-Direktive besteht darin, dass ein Thread beim Erreichen der Direktive so lange wartet, bis kein anderer Thread den Anweisungsblock des kritischen Bereiches ausführt. Erst wenn dies erfüllt ist, führt der Thread den Anweisungsblock aus.

Die Threads eines Teams können mit einer **barrier**-Direktive

```
#pragma omp barrier
```

synchronisiert werden, d.h. erst wenn jeder Thread des Teams diese Direktive erreicht hat, beginnen die Threads des Teams die Abarbeitung der nachfolgenden Anweisungen.

Durch Angabe der **atomic**-Direktive können bestimmte Speicherzugriffe als atomare Operationen durchgeführt werden. Die Syntax dieser Direktive ist

```
#pragma omp atomic
    Zuweisung
```

Die Zuweisung muss dabei eine der folgenden Formen annehmen:

```
x binop= E,
x++, ++x, x--, --x,
```

wobei **x** einen beliebigen Variablenzugriff, **E** einen beliebigen skalaren Ausdruck, der **x** nicht enthält, und **binop** $\in \{+, -, *, /, \&, \wedge, |, <<, >>\}$ einen binären Operator bezeichnet. Der Effekt besteht darin, dass nach Auswertung des Ausdrucks **E** die angegebene Aktualisierung von **x** als atomare Operation erfolgt, d.h. während der Aktualisierung kann kein anderer Thread **x** lesen oder manipulieren. Die Auswertung von **E** erfolgt *nicht* als atomare Operation. Prinzipiell kann der Effekt einer **atomic**-Direktive auch durch eine **critical**-Direktive erreicht werden, die vereinfachte Form der **atomic**-Direktive kann aber evtl. vom Laufzeitsystem für eine effiziente Implementierung ausgenutzt werden. Auch ist es möglich mit der **atomic**-Direktive einzelne Feldelemente anzusprechen, wohingegen die **critical**-Direktive das gesamte Feld schützen würde. Beispiele für die Verwendung von atomaren Operationen sind:

```
extern float a[], *p=a, b; int index[];
#pragma omp atomic
    a[index[i]] += b;
#pragma omp atomic
    p[i] -= 1.0;
```

Mit Hilfe der **flush**-Direktive kann der Programmierer sicherstellen, dass alle Threads eines Teams eine konsistente Sicht auf gemeinsame Datenstrukturen haben. Die Syntax der Direktive ist

```
#pragma omp flush [(list)] ,
```

wobei **list** eine Liste von Variablen bezeichnet, die einen einheitlichen Wert annehmen sollen. Ist in der Liste ein Zeiger angegeben, erhält nur der Zeiger, nicht jedoch die Datenstruktur, auf die er zeigt, einen konsistenten Wert. Wenn keine Variablenliste angegeben ist, erhalten *alle* zugreifbaren Variablen einen konsistenten Wert. Abbildung 6.40 zeigt ein Beispiel nach [116] zur Verwendung einer **flush**-Direktive. Bei folgenden Direktiven wird *implizit* eine **flush**-Direktive ohne Variablenliste ausgeführt:

- **barrier**-Direktive,
- beim Eintritt und beim Verlassen einer **critical**-Direktive,

- beim Verlassen einer `parallel`-Direktive,
- beim Verlassen einer `for`-, `sections`- oder `single`-Direktive, wenn kein `nowait`-Parameter angegeben ist.

```
#pragma omp parallel private (iam, neighbor) shared (work, sync)
{
    iam = omp_get_thread_num();
    sync[iam] = 0;
    #pragma omp barrier
    work[iam] = do_work();
    #pragma omp flush (work)
    sync[iam] = 1;
    #pragma omp flush (sync)
    neighbor = (iam != 0) ? (iam - 1) : (omp_get_num_threads() - 1);
    while (sync[neighbor] == 0) {
        #pragma omp flush (sync)
        { }
    }
    combine (work[iam], work[neighbor]);
}
```

Abb. 6.40. Programmfragment zur Anwendung einer `flush`-Direktive bei der Zusammenarbeit zweier Threads. Jeder der beiden Threads i ($i = 0, 1$) errechnet in `work[i]` einen Wert und macht diesen mit einer `flush`-Direktive dem anderen Thread verfügbar. Die Fertigstellung der Wertberechnung wird im Feld `sync` angezeigt, dessen Inhalt ebenfalls mit einer `flush`-Direktive dem anderen Thread verfügbar gemacht wird. Die Weiterverarbeitung der Werte mit `combine()` geschieht erst, wenn beide Threads ihre Werte errechnet haben.

Die Ausführung eines Anweisungsblocks innerhalb eines parallelen Bereiches nur durch den Master-Thread des ausführenden Teams wird durch die Direktive

```
#pragma omp master
    Anweisungsblock
```

erreicht. Alle anderen Threads des Teams ignorieren den angegebenen Anweisungsblock. Es findet *keine* implizite Synchronisation statt.

Um globale **Reduktionsoperationen** zu ermöglichen, stellt OpenMP für die `parallel`-, `sections`- und `for`-Direktiven einen `reduction`-Parameter mit der Syntax

```
reduction (op: list)
```

zur Verfügung. Dabei bezeichnet $op \in \{+, -, *, /, \&, \wedge, |, \&\&, ||\}$ den anzuwendenden Reduktionsoperator, `list` ist eine mit Kommata getrennte Liste von *Reduktionsvariablen*, die im umgebenden Kontext als gemeinsame Variable

deklariert sein müssen. Der Effekt des Parameters besteht darin, dass bei der Bearbeitung der zugehörigen Direktive für jede der angegebenen Reduktionsvariablen für jeden Thread eine private Kopie der Variablen angelegt wird, die entsprechend der angegebenen Reduktionsoperation mit dem neutralen Element dieser Operation initialisiert wird. Den Reduktionsvariablen können während der Abarbeitung des zugehörigen parallelen Bereiches von den verschiedenen Threads Werte zugewiesen werden, die entsprechend der angegebenen Operation `op` akkumuliert werden. Am Ende der Direktive, für die der `reduction`-Parameter angegeben wurde, werden die (gemeinsamen) Reduktionsvariablen aktualisiert. Dies geschieht dadurch, dass der ursprüngliche Wert der Reduktionsvariablen und die von den Threads während der Abarbeitung der zugehörigen Direktive errechneten Werte der privaten Kopien entsprechend der Reduktionsoperation verknüpft werden. Der so errechnete Wert wird der Reduktionsvariablen als neuer Wert zugewiesen. Typischerweise wird der `reduction`-Parameter zur Akkumulation von Werten verwendet, was in folgendem Beispiel illustriert wird:

Beispiel: Das folgende Programmfragment dient zur Akkumulation von Werten in den Akkumulationsvariablen `a`, `y` und `am`:

```
#pragma omp parallel for reduction (+: a,y) reduction (||: am)
for (i=0; i<n; i++) {
    a += b[i];
    z = sum (y, c[i]);
    am = am || b[i] == c[i];
}
```

Die angegebenen Akkumulations-Operationen werden von den Threads, die den parallelen Bereich ausführen, für verschiedene Iterationen der parallelen Schleife durchgeführt. Dabei kann eine Reduktionsoperation auch in einem Funktionsaufruf ausgeführt werden, wie dies für die Berechnung von `y` der Fall ist. Nach Beendigung der parallelen Schleife werden die von den verschiedenen Threads akkumulierten Werte global in den angegebenen Reduktionsvariablen akkumuliert. $\square$

Für den Zugriff auf gemeinsame Variablen stellt OpenMP einen **Sperrmechanismus** zur Verfügung, der über Laufzeitfunktionen verwaltet werden kann. Dabei unterscheidet OpenMP zwischen *einfachen Sperrvariablen* vom Typ `omp_lock_t` und *schachtelbaren Sperrvariablen* vom Typ `omp_nest_lock_t`. Der Unterschied besteht darin, dass eine einfache Sperrvariable nur einmal belegt werden kann, während eine schachtelbare Sperrvariable vom gleichen Thread mehrfach belegt werden kann. Dazu wird für die schachtelbare Sperrvariable ein Zähler gehalten, der die Anzahl der Belegungen mitzählt. Vor Benutzung einer Sperrvariablen muss diese initialisiert werden, wozu die beiden folgenden Funktionen

```
void omp_init_lock (omp_lock_t *lock)
```

```
void omp_init_nest_lock (omp_nest_lock_t *lock)
```

zur Verfügung stehen. Nach der Initialisierung einer Sperrvariablen ist diese *nicht* belegt. Zur Zerstörung einer initialisierten Sperrvariablen stehen die Funktionen

```
void omp_destroy_lock (omp_lock_t *lock)
void omp_destroy_nest_lock (omp_nest_lock_t *lock)
```

zur Verfügung. Nach Initialisierung einer Sperrvariablen kann diese wie üblich zur Koordination des konkurrierenden Zugriffs auf gemeinsame Daten benutzt werden. Zur Belegung einer Sperrvariablen werden die Funktionen

```
void omp_set_lock (omp_lock_t *lock)
void omp_set_nest_lock (omp_nest_lock_t *lock)
```

verwendet. Beide Funktionen blockieren den ausführenden Thread so lange, bis die angegebene Sperrvariable verfügbar ist. Eine einfache Sperrvariable ist verfügbar, wenn sie von keinem Thread belegt ist. Eine schachtelbare Sperrvariable ist verfügbar, wenn sie entweder von keinem Thread oder vom ausführenden Thread belegt ist. Wenn die angegebene Sperrvariable verfügbar ist, wird sie vom ausführenden Thread belegt. Für schachtelbare Sperrvariablen wird der assoziierte Zähler inkrementiert.

Eine Belegung einer Sperrvariablen kann durch Aufruf der Funktionen

```
void omp_unset_lock (omp_lock_t *lock)
void omp_unset_nest_lock (omp_nest_lock_t *lock)
```

wieder freigegeben werden. Dabei kann nur der Thread, der die Sperrvariable belegt hat, diese auch freigeben. Eine normale Sperrvariable wird durch Aufruf von `omp_unset_lock()` freigegeben. Für eine schachtelbare Sperrvariable dekrementiert der Aufruf `omp_unset_nest_lock()` den zugeordneten Zähler. Wenn der Zähler dadurch den Wert 0 erreicht, wird die Sperrvariable freigegeben.

Soll beim Versuch der Belegung einer von einem anderen Thread belegten Sperrvariablen die Blockierung des ausführenden Threads vermieden werden, können die Funktionen

```
void omp_test_lock (omp_lock_t *lock)
void omp_test_nest_lock (omp_nest_lock_t *lock)
```

verwendet werden. Wenn die angegebene Sperrvariable verfügbar ist, wird sie wie durch die Aufrufe von `omp_set_lock()` bzw. `omp_set_nest_lock()` belegt. Wenn die Sperrvariable nicht verfügbar ist, wird jedoch der aufrufende Thread nicht blockiert. Stattdessen liefern die Funktionsaufrufe den Rückgabewert 0 an den aufrufenden Thread zurück. Bei erfolgreicher Belegung der Sperrvariablen liefert `omp_test_lock()` einen Wert $\neq 0$ zurück,

`omp_test_nest_lock()` liefert den neuen Wert des zugeordneten Zählers zurück.

Beispiel: Abbildung 6.41 zeigt als Beispiel ein Programmfragment, das eine schachtelbare Sperrvariable verwendet, um konkurrierend eine gesamte Datenstruktur `pair` bzw. einen Teil der Datenstruktur `b` zu manipulieren, vgl. [116]. Die Funktion `incr_a()` verändert den Eintrag `a` der Datenstruktur `pair`. Da `incr_a()` nur von `incr_pair()` aufgerufen wird, braucht keine Sperrvariable verwendet werden. Da `incr_b()` hingegen konkurrierend zu `incr_pair()` aufgerufen werden kann, muss die Manipulation mit einem Sperrmechanismus geschützt werden. $\square$

```
#include <omp.h>

typedef struct {
    int a, b;
    omp_nest_lock_t l;
} pair;

void incr_a (pair *p, int a) {
    p->a += a;
}

void incr_b (pair *p, int b) {
    omp_set_nest_lock (&p->l);
    p->b += b;
    omp_unset_nest_lock (&p->l);
}

void incr_pair (pair *p, int a, int b) {
    omp_set_nest_lock (&p->l);
    incr_a (p, a);
    incr_b (p, b);
    omp_unset_nest_lock (&p->l);
}

void f (pair *p) {
    extern int work1(), work2(), work3();
    #pragma omp parallel sections
    {
        #pragma omp section
            incr_pair (p, work1(), work2());
        #pragma omp section
            incr_b (p, work3());
    }
}
```

Abb. 6.41. Programmskizze zur Verwendung einer schachtelbaren Sperrvariablen in OpenMP.

6.5 Unified Parallel C

Unified Parallel C (UPC) ist eine explizite Erweiterung der Programmiersprache C im ISO (International Standard Organization) C99 Standard für den Einsatz im Hochleistungsrechnen auf Parallelrechnern oder Beowulf-Clustern. Das UPC zugrundeliegende Programmiermodell ist das Modell eines verteilten gemeinsamen Speichers (engl. *distributed shared memory*, DSM), das Aspekte der Programmierung mit gemeinsamem Adressraum und der Programmierung mit verteiltem Adressraum verbindet. Dieses Programmiermodell basiert auf einem gemeinsamen, partitionierten Adressraum, wobei Variablen von jedem Thread gelesen oder beschrieben werden können. Jede Variable ist jedoch mit einem Thread assoziiert, auch *affinity* genannt, was bei einer Implementierung auf einem Rechner mit verteiltem Speicher eine tatsächliche physikalische Verteiltheit bedeutet. Der Zugriff auf eine Variable benötigt dann unterschiedliche Zeiten im Sinne eines NUMA Modells, vgl. Abschnitt 2.4. Speicherkonsistenzmechanismen regeln den Zugriff auf diese gemeinsamen Adressbereiche, vgl. Abschnitt 2.7.3.

Parallelität wird in UPC-Programmen explizit im SPMD-Stil ausgedrückt und hat einen beim Programmstart festgelegten Parallelitätsgrad in Form einer bestimmten Anzahl von Threads. Ein Thread und seine assoziierten Daten werden typischerweise auf denselben physikalischen Prozessor oder Knoten abgebildet, wodurch Lokalitätseffekte ausgenutzt werden können. Zusätzlich kann ein Thread private Daten haben, die immer auf demselben Prozessor liegen wie der Thread.

Die UPC-Erweiterungen des ISO C99 Standards beinhalten ein explizites paralleles Ausführungsmodell, einen gemeinsamen Adressraum, Synchronisations-Primitive und Speicherkonsistenzmodelle sowie Operationen für das Speicher-Management. UPC wird durch ein aus Universitäten, Forschungsinstituten und Firmen zusammengesetztes Konsortium entwickelt und unterstützt, darunter die George Washington University (<http://upc.gwu.edu>) und das Lawrence Berkeley National Laboratory (LBNL). Für die Entwicklungen des UPC Standards sind Erfahrungen früherer Erweiterungen des ISO C99 Standards eingeflossen, und zwar AC, Split-C und Parallel C Preprocessor (PCP) [28]. Der Sprachentwurf zielt darauf, das Modell eines gemeinsamen Adressraums mit den Vorteilen der expliziten Datenverteilung und der Leistung der Message-Passing Programmierung zu verbinden. Das dazu benutzte Speichermodell ist ein partitionierter globaler Adressraum (engl. *partitioned global address space*, PGAS) [28]. Weitere Sprachen, die PGAS realisieren sind Co-Array Fortran Language (CAF), eine auf Fortran basierende parallele Sprache, und Titanium, eine auf Java basierende Sprache ähnlich zu UPC.

Eine neue Sprache, die ebenfalls den PGAS-Ansatz verfolgt, ist X10 [23]. X10 ist eine Erweiterung von Java, die für das Hochleistungsrechnen entwickelt wurde und nachrichtenbasierte Protokolle für die Kommunikation zwischen Threads zur Verfügung stellt. X10 wurde von IBM im

Rahmen des DARPA HPCS-Programms (High Productivity Computing Systems) entwickelt. X10 erweitert den PGAS-Ansatz zum GALS-Ansatz (Globally Asynchronous, Locally Synchronous) durch Einführung von logischen Ausführungsorten (engl *places*), wobei Threads eines Ausführungsortes eine lokal synchrone Sicht auf einen gemeinsamen Speicher haben. Threads unterschiedlicher Ausführungsorte werden dagegen asynchron zueinander ausgeführt. Im Rahmen des HPCS-Programms wurde mit der Entwicklung zweier weiterer Sprachen begonnen: Chapel von Cray und Fortress von Sun. Chapel ist eine parallele Sprache, die auf dem Modell eines gemeinsamen Adressraumes basiert. Fortress ist eine an Fortran angelehnte neue objektorientierte Sprache, die versucht, die Programmierung paralleler Systeme durch Verwendung einer exakten mathematischen Notation zu erleichtern [8]. Fortress unterstützt eine parallele Abarbeitung von Programmen z.B. durch parallele Schleifen und die parallele Auswertung von Funktionsargumenten durch unterschiedliche Threads, die sich über atomare Codeblöcke synchronisieren können. Auch die Definition verteilter Datenstrukturen wird unterstützt. Ein weiterer Programmieransatz, der die Programmierung von verteiltem und gemeinsamem Adressraum kombiniert, ist der GA-Ansatz (*Global Arrays*) [114]. Der GA-Ansatz erlaubt dem Programmierer über Bibliotheksaufrufe die Definition von globalen Feldern mit Datenverteilung, auf die alle Berechnungsströme zugreifen können. Eine gemischte Nutzung von GA und MPI ist möglich.

In diesem Abschnitt stellen wir den Ansatz der Sprache UPC genauer vor. Die erste Version 0.9 des UPC Standards stammt aus dem Jahr 1999. Nach weiteren Versionen 1.0 im Jahr 2001 und 1.1 im Jahr 2003 wurde die Version 1.2 im Mai 2005 fertiggestellt (siehe http://upc.lbl.gov/does/user/upc_spec.1.2.pdf). Neben der freien Implementierung des LBNL existieren freie GCC-basierte UPC-Implementierungen für x86, x86-64, SGI IRIX und Cray T3E sowie eine MPI-basierte Referenzimplementierung von Michigan Tech für Linux und Tru64 Betriebssysteme. Hewlett Packard, Cray und IBM bieten ebenfalls UPC Compiler an, darunter IBMUPC für den IBM Parallelrechner BlueGene. Die UPC-Compiler für Linux gibt es auch als GCC UPC (<http://www.intrepid.com/upc>) und Berkeley UPC (<http://upc.nersc.gov>), vgl [83].

6.5.1 UPC Programmiermodell und Benutzung

Die parallel abzuarbeitenden Teilaufgaben werden in UPC Thread genannt. Jeder UPC-Thread hat einen eigenen Adressraum, kann aber auch auf gemeinsame Variablen zugreifen. UPC-Threads entsprechen also nicht den Threads der Pthreads-Bibliothek und werden in einigen Implementierungen auf jeweils separate Prozesse abgebildet. Eine festgelegte Anzahl von Threads arbeitet unabhängig voneinander ein paralleles Programm im SPMD Modus ab. Dies ist vergleichbar mit dem Programmiermodell von MPI oder PVM,

UPC kann jedoch meist nicht zusammen mit MPI genutzt werden, d.h. UPC-Threads können keine privaten Daten mittels MPI-Operationen austauschen. Die explizite Parallelität von UPC unterscheidet sich von der impliziten Parallelität der Threads in OpenMP. Die Anzahl der Threads kann entweder zur Compilezeit oder zur Laufzeit spezifiziert werden und wird in der vordefinierten Programmvariablen `THREADS` abgespeichert. Eine weitere vordefinierte Programmvariable ist `MYTHREAD`, in der der eindeutige Threadindex mit den Werten `0,...,THREADS-1` gespeichert wird. Jedes in C geschriebene Programm ist bereits ein korrektes UPC-Programm. Durch Einfügen des Headers `upc.h` und Übersetzung als UPC-Programm mit p Threads wird ein UPC-Programm mit p Kopien erzeugt. So ergibt das Programm

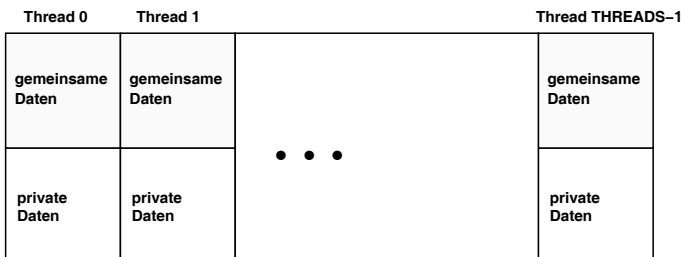
```
#include <upc.h>
#include <stdio.h>
int main (){
    printf("Thread %d von %d: hallo\n", MYTHREAD, THREADS);
}
```

eine Ausgabe, in der jeder der gestarteten Threads seinen eigenen Threadindex und die Gesamtanzahl der Threads ausgibt.

Das Hauptmerkmal des DSM Modells in UPC ist ein Speichermodell mit privaten Variablen (engl. *private variables*) und gemeinsamen Variablen (engl. *shared variables*). Private Variablen sind normale C Variablen, die jeweils im privaten Speicher der Threads abgelegt werden, so dass für eine Variable `int my` eine der Anzahl der Threads entsprechende Anzahl von Kopien angelegt wird. Gemeinsame Variablen werden nur einmal allokiert und werden per Default im Speicher von Thread 0 abgelegt. Die Deklaration gemeinsamer Variablen erfolgt in der Form

```
shared int our;
```

und allokiert die Variable `our` im Speicher von Thread 0, kann aber von allen anderen Threads zugegriffen werden. Dies sollte aus Performance-Gründen eher selten verwendet werden, da der Zugriff über das Verbindungsnetzwerk eines Parallelrechners oder Clusters erfolgen muss. Der globale Adressraum hat das folgende Aussehen:



6.5.2 Gemeinsame Felder

Gemeinsame Felder (engl. *shared arrays*) sind Felder, die im gemeinsamen Adressraum allokiert werden. Die Elemente gemeinsamer Felder werden blockzyklisch ohne Überlappungen auf alle Threads verteilt. Die Blockgröße wird bei der Felddeklaration nach dem Schlüsselwort **shared** angegeben und muss zur Compilezeit auswertbar sein.

Beispiel: Durch die folgende Deklaration

```
shared [3] int x [3*THREADS];
```

wird ein gemeinsames Feld mit Blockgröße 3 eingeführt, so dass jeder Thread genau einen Block erhält. Fehlt die Angabe einer Blockgröße, wird implizit Blockgröße 1 verwendet. Leere Klammern bedeuten eine unbegrenzte Blockgröße; der Effekt besteht darin, dass alle Feldelemente dem Thread 0 zugeordnet werden. Der gleiche Effekt wird durch Angabe der Blockgröße 0 erreicht. Die folgenden drei Deklarationen sind somit äquivalent:

```
shared [12] int x [12];
shared [ ] int x [12];
shared [0] int x [12];
```

□

Durch Angabe der Blockgröße ***** kann eine möglichst gleichmäßige Aufteilung der Elemente eines Feldes auf die Threads in zusammenhängenden Blöcken erreicht werden. Die Errechnung der Blockgrößen erfolgt in diesem Fall durch den Compiler. Für mehrdimensionale Felder wird eine zeilenweise Ablage der Feldelemente zugrundegelegt und die resultierenden Blöcke werden zyklisch auf die Threads verteilt. Jeder Zugriff auf ein Element eines gemeinsamen Feldes im Programm wird vom Compiler in einen entfernten Zugriff (engl. *remote reference*) umgewandelt, der das Element aus dem Speicherbereich des Eigentümer-Threads zugreift. Je nach Organisation des physikalischen Speichers ist dazu Kommunikation erforderlich, d.h. Zugriffe auf gemeinsame Felder können eine lange Zugriffszeit haben. Greift ein Thread auf den ihm zugeordneten Bereich eines gemeinsamen Feldes zu, handelt es sich um einen lokalen Zugriff, der ohne Kommunikation und damit schnell durchführbar ist.

Der Wert der vordefinierten Programmvariablen **THREADS** kann zur Compilezeit durch die Compileroption **fthreads** oder zur Laufzeit, d.h. beim Start des Programms, durch eine Option von **upcrun** festgelegt werden. Erfolgt die Festlegung zur Compilezeit, kann das erzeugte Programm nur mit der bei der Übersetzung angegebenen Threadanzahl gestartet werden. Wird die Programmvariable **THREADS** zur Compilezeit festgelegt, wird diese wie eine normale Konstante behandelt und kann überall dort verwendet werden, wo auch eine Integerkonstante verwendet werden kann. Dies gilt insbesondere auch für die Festlegung der Dimensionen eines gemeinsamen Feldes, d.h. **THREADS**

kann durchaus mehrfach für die Festlegung der Größe von Felddimensionen verwendet werden. Wird die Programmvariable `THREADS` erst zur Laufzeit festgelegt, muss die Größe genau einer Dimension jedes gemeinsamen Feldes mit begrenzter Blockgröße ein Vielfaches von `THREADS` sein, damit der Compiler die Umwandlungsoperationen der Feldzugriffe erzeugen kann. Für Felder mit unbestimmter Blockgröße ist in diesem Fall in keiner Dimension die Verwendung von `THREADS` erlaubt. Das gesamte Feld wird Thread 0 zugeordnet. Wird `THREADS` erst zur Laufzeit des Programms festgelegt und wird `THREADS` in einer Felddimension verwendet, kann die Größe eines gemeinsamen Feldes nicht zur Compilezeit festgelegt werden, da diese von der Anzahl der verwendeten Threads abhängt. In diesem Fall können mit `sizeof(array)` die Größe in Bytes und mit `upc_elemsizeof(array)` die Größe des zugrundeliegenden Datentyps in Bytes bestimmt werden.

Beispiel: Die Deklaration

```
shared [5] int a [4*THREADS] [5];
```

deklariert ein gemeinsames Feld `a` mit zwei Dimensionen, dessen Zeilen zyklisch den Threads zugeordnet werden, so dass jeder Thread insgesamt vier Zeilen erhält. Eine blockweise Verteilung der Zeilen wird durch die Deklaration

```
shared [20] int a [4*THREADS] [5];
```

erreicht. Eine spaltenzyklische Verteilung wird durch die Deklaration

```
shared int a [5] [4*THREADS];
```

realisiert; das Feld hat aber andere Dimensionsgrößen wie in den beiden vorangehenden Deklarationen. Eine blockweise Verteilung der Spalten wird erreicht durch

```
shared [4] int a [5] [4*THREADS];
```

Alle diese Deklarationen sind auch für den Fall zulässig, dass `THREADS` erst zur Laufzeit festgelegt wird. □

6.5.3 Speicherkonsistenzmodelle von UPC

Jeder UPC-Thread hat Zugriff auf einen gemeinsamen Adressraum. Das Speicherkonsistenzmodell legt fest, in welcher Reihenfolge die Zugriffsoperationen von UPC-Threads auf den gemeinsamen Adressraum von den anderen Threads beobachtet werden, vgl. Abschnitt 2.7.3.

UPC stellt zwei Konsistenzmodelle zur Verfügung, die über die Schlüsselwörter `strict` und `relaxed` angesprochen werden. Für das `relaxed` Modell kann das Laufzeitsystem die Speicherzugriffsoperationen eines Threads umordnen, wenn dies das von dem Thread errechnete Ergebnis nicht verändert. Eine Folge von `relaxed` Speicherzugriffen durch einen Thread *A* kann für

einen anderen Thread *B* in einer beliebigen Reihenfolge sichtbar werden. Damit beinhaltet ein Speicherzugriff keinerlei Synchronisation zwischen den Threads; diese muss bei Bedarf durch explizite Synchronisationsoperationen wie `upc_fence` herbeigeführt werden, siehe unten. Im `relaxed` Modell können auch mehrere Threads gleichzeitig auf eine gemeinsame Variable schreiben.

Bei Verwendung des `strict` Modells sind Umordnungen der Speicherzugriffe der Threads nicht erlaubt, d.h. eine Folge von `strict` Speicherzugriffen eines Threads *A* wird für die anderen Threads in der Reihenfolge sichtbar, in der sie von *A* ausgeführt werden. Dieses Verhalten wird dadurch erreicht, dass vor der Ausführung eines `strict` Speicherzugriffs alle vorangehenden Zugriffe des ausführenden Threads auf den gemeinsamen Adressraum abgeschlossen und damit für alle anderen Threads sichtbar werden. Erst dann wird mit der Ausführung des `strict` Speicherzugriffs begonnen.

UPC erlaubt die Steuerung des verwendeten Konsistenzmodells auf verschiedenen Granularitätsstufen: für einzelne `shared` Objekte, für Zugriffe innerhalb eines Blockes von Anweisungen und für ein gesamtes Programm. Die Steuerung auf der Ebene gemeinsamer Objekte erfolgt durch Verwendung der Schlüsselworte `strict` oder `relaxed` in der Objektdeklaration. So deklariert

```
strict shared int x;
```

eine Integervariable `x` im gemeinsamen Adressraum, für die alle Zugriffe `strict` erfolgen. Die Zugriffe innerhalb eines Blockes von Anweisungen können mit Hilfe von `pragma`-Direktiven gesteuert werden, wie sie auch in OpenMP verwendet werden. Die `pragma`-Direktive

```
{
    # pragma upc strict
    ...
}
```

bewirkt, dass alle Zugriffe auf gemeinsame Variablen im Anweisungsblock `strict` erfolgen. Wird eine solche `pragma`-Direktive außerhalb eines Anweisungsblockes verwendet, gilt das angegebene Konsistenzmodell bis zur nächsten `pragma`-Direktive. Für ein gesamtes Programm kann das Speicherkonsistenzmodell durch Einbinden einer der beiden Dateien

```
#include <upc_relaxed.h>
#include <upc_strict.h>
```

festgelegt werden. Die Schlüsselworte `strict` und `relaxed` haben Vorrang vor den `pragma`-Direktiven und diese haben Vorrang vor den Steuerdateien, so dass trotz Festlegung eines Speicherkonsistenzmodells für das gesamte Programm für einzelne Anweisungsblöcke oder Variablen ein abweichendes Speicherkonsistenzmodell vereinbart werden kann.

Die Sichtbarkeit von Modifikationen an gemeinsamen Objekten kann auch mit Hilfe der UPC-Operation

```
upc_fence
```

beeinflusst werden. Das Ausführen dieser Operation bewirkt, dass alle vorangehenden Modifikationen an gemeinsamen Objekten abgeschlossen werden und damit in den Anweisungen nach `upc_fence` für alle Threads sichtbar sind. Die Operation `upc_fence` bewirkt auch, dass der UPC-Compiler keine nach `upc_fence` stehende Referenz auf ein gemeinsames Objekt vor diese Operation stellt.

6.5.4 Zeiger und Felder in UPC

Wie in C können in UPC Zeiger zur Spezifikation kompakter, effizienter Programme verwendet werden. Da das Speichermodell von UPC zwischen privaten und gemeinsamen Daten unterscheidet, können in UPC vier Typen von Zeigern unterschieden werden: private Zeiger auf private Daten, private Zeiger auf gemeinsame Daten, gemeinsame Zeiger auf private Daten und gemeinsame Zeiger auf gemeinsame Daten. Durch Verwendung des Schlüsselwortes `shared` kann zwischen den Typen unterschieden werden:

```
int *p1;           //privater Zeiger auf private Daten
shared int *p2;    //privater Zeiger auf gemeinsame Daten
int * shared p3;   //gemeinsamer Zeiger auf private Daten
shared int * shared p4; //gemeinsamer Zeiger auf gemeinsame Daten
```

Die Angabe von `shared` vor dem Variablennamen des Zeigers bezeichnet einen gemeinsamen Zeiger, die Angabe von `shared` vor dem Typnamen bezeichnet einen Zeiger auf gemeinsame Daten.

Im obigen Beispiel sind `p1` und `p2` private Zeiger, d.h. jeder Thread hat eine private Instanz dieser Zeiger. Von `p3` und `p4` existiert dagegen nur eine, von allen Threads zugreifbare Instanz. Jede Instanz von `p1` kann auf private Daten des jeweiligen Threads, aber auch auf diesem Thread zugeordnete gemeinsame Daten zeigen. Jede Instanz von `p2` kann auf beliebige gemeinsame Daten zeigen. Der Zeiger `p4` kann ebenfalls auf beliebige gemeinsame Daten zeigen. Der gemeinsame Zeiger `p3` wird im gemeinsamen Adressraum von Thread 0 erzeugt und kann nur auf private Daten von Thread 0 zugreifen. Da dies wenig sinnvoll ist, werden gemeinsame Zeiger auf private Daten nicht verwendet.

Zeiger auf gemeinsame Daten werden vom UPC-Laufzeitsystem in einem speziellen Format dargestellt, das aus drei Einträgen besteht:

Thread	Blockadresse	Position

Der Eintrag **Thread** gibt an, welchem Thread die Daten zugeordnet sind, auf die der Zeiger verweist. Der Eintrag **Blockadresse** bezeichnet die Adresse des zugehörigen Datenblockes, **Position** gibt die Position des Datenelements innerhalb des Blocks an. Diese Einträge können innerhalb eines Programms mit Hilfe der folgenden Funktionen abgefragt werden:

```

size_t upc_threadof (shared void *ptr)
size_t upc_addrfieldof (shared void *ptr)
size_t upc_phaseof (shared void *ptr)

```

Argument ist jeweils ein Zeiger auf gemeinsame Daten; Rückgabewert ist der entsprechende Eintrag des Darstellungsformats. Die genaue Festlegung der Einträge des Darstellungsformats ist implementierungsabhängig. Daher kann z.B. nicht davon ausgegangen werden, dass die Blockadresse den gleichen Wert enthält wie ein privater Zeiger auf das erste Element des Blockes.

Zeiger auf gemeinsame Daten können wie in C üblich dazu verwendet werden, auf die Elemente eines Feldes nacheinander zuzugreifen. Dabei ist jedoch zu beachten, dass der Zeiger, der für den Lauf über die Feldelemente verwendet wird, mit der gleichen Blockgröße deklariert ist wie das Feld, über das er läuft, da ansonsten die Standard-Blockgröße 1 verwendet wird. Dies kann an folgendem Beispiel demonstriert werden.

Beispiel: Wir betrachten folgendes Programmstück in UPC:

```

shared [4] int x[4*THREADS], *p1;
shared int *p2, i;
for (p1=x, i=0; i < 4 * THREADS; p1++, i++)
    printf ("x[%d] = %d \n", i, *p1);
for (p2=x, i=0; i < 4 * THREADS; p2++, i++)
    printf ("x[%d] = %d \n", i, *p2);

```

Für vier Threads ergibt sich für das Feld `x` folgende Speicheraufteilung:

Thread 0	Thread 1	Thread 2	Thread 3
x[0]	x[4]	x[8]	x[12]
x[1]	x[5]	x[9]	x[13]
x[2]	x[6]	x[10]	x[14]
x[3]	x[7]	x[11]	x[15]

In der ersten Schleife des Beispiels läuft `p1` in der Reihenfolge `x[0]`, `x[1]`, `x[2]`, ..., `x[15]` über das Feld `x`. In der zweiten Schleife läuft `p2` in der Reihenfolge `x[0]`, `x[4]`, `x[8]`, `x[12]`, `x[1]`, `x[5]`, ... über `x`, da für `p2` die Default-Blockgröße 1 angenommen wird. □

Zeiger auf gemeinsame Daten können in Zeiger auf private Daten umgewandelt werden, indem der Umwandlungsmechanismus von C (engl. *casting*) angewendet wird. Mit dem resultierenden Zeiger auf private Daten kann ein Thread auf die ihm lokal zugeordneten Elemente eines gemeinsamen Feldes zugreifen. Bei der Umwandlung geht aber die in dem Format für Zeiger auf gemeinsame Daten abgelegte Threadinformation verloren. Das Umwandeln eines Zeigers auf private Daten in einen Zeiger auf gemeinsame Daten ist nicht möglich.

6.5.5 Parallele Schleifen in UPC

Neben den Iterationsanweisungen der Programmiersprache C stellt UPC die Anweisung:

```
upc_forall (init; test; loop; affinity) statement;
```

zur Ausführung von parallelen Schleifen bereit, wobei `init`, `test` und `loop` die Bedeutung der Ausdrücke einer `for`-Schleife in C haben. Die Iterationen des Schleifenrumpfes müssen unabhängig voneinander sein, damit die Abarbeitung definiert ist. Der Ausdruck `affinity` ist vom Typ Integer oder Zeiger auf eine gemeinsame Variable und spezifiziert die Iterationen, die durch einen Thread ausgeführt werden sollen. Ist der Ausdruck für `affinity` als Zeiger auf eine gemeinsame Variable gegeben, d.h. als `upc_threadof (affinity)`, dann wird der Schleifenrumpf für alle Iterationen ausgeführt, für die der Wert von MYTHREAD diesem Wert entspricht. Ist der Ausdruck für `affinity` als Integerausdruck gegeben, so wird der Schleifenrumpf für alle Iterationen ausgeführt, für die der Wert von MYTHREAD dem Wert `affinity % THREADS` entspricht.

```
#include <upc_relaxed.h>
#define N 100*THREADS;
shared int a[N][N], x[N], y[N];

int main() {
    int i,j;
    upc_forall (i=0; i < THREADS; i++) {
        y[i] = 0;
        for (j=0; j < N; j++)
            y[i] += x[j] * a[i][j];
    }
}
```

Abb. 6.42. Matrix-Vektor-Multiplikation mit Default-Blockgröße 1.

Beispiel: Der Einfluss der Blockgröße auf die Abarbeitung eines thread-parallelen UPC-Programms lässt sich auch an dem einfachen Beispiel einer Matrix-Vektor-Multiplikation verdeutlichen, vgl. Abb. 6.42. In Abb. 6.42 wird die Default-Blockgröße 1 verwendet. Für die Matrix `a` führt dies dazu, dass die Spalten von `a` zyklisch auf die Threads verteilt werden. Zur Berechnung von `y[i]` muss der berechnende Thread `i%THREADS` auf alle Elemente der `i`-ten Zeile von `a` zugreifen sowie auf alle Elemente von `x`, so dass viele Zugriffe auf Daten anderer Threads resultieren. Die `upc_forall`-Schleife bewirkt, dass jeder Thread die Elemente des Ergebnisvektors `y` berechnet, die ihm zugeordnet sind. Die Elemente von `x` und `y` werden ebenfalls zyklisch auf die Threads verteilt.

Das Programm in Abb. 6.42 kann dahingehend modifiziert werden, dass das Feld **a** mit Blockgröße **N** deklariert wird:

```
shared [N] int a[N][N];
```

Der Rest des Programms bleibt unverändert. Die geänderte Deklaration für **a** bewirkt, dass **a** anstatt wie bisher spaltenzyklisch jetzt zeilenzyklisch auf die Threads verteilt wird. Jeder Thread berechnet weiterhin die ihm zugeordneten Elemente von **y**. Die zur Berechnung von **y[i]** erforderliche **i**-te Zeile von **a** ist jetzt aber vollständig dem berechnenden Thread zugeordnet, so dass wesentlich weniger Zugriffe auf Datenbereiche anderer Threads resultieren. □

Als abschließendes Beispiel enthält Abb. 6.43 eine Programmskizze zur Multiplikation zweier Matrizen **a** und **b**. Die Ergebnismatrix wird in **c** abgelegt. Die Matrizen **a** und **c** werden mit Blockgröße **N·N/THREADS** deklariert, was zu einer blockorientierten Verteilung der Zeilen auf die Threads führt, so dass jeder Thread einen Block mit **N/THREADS** zusammenhängenden Zeilen hat. Für Matrix **b** wird Blockgröße **N/THREADS** angegeben, was zu einer blockorientierten Verteilung der Spalten von **b** auf die Threads führt. Jeder Thread errechnet die ihm zugeordneten Zeilen der Ergebnismatrix **c**. Dazu greift er auf lokal abgelegte Zeilen von **a** und auf lokale und nichtlokale Spalten von **b** zu.

Eine Variation ergibt sich durch geänderte Blockgrößen:

```
shared [N] int a[N][N], c[N][N];
shared      int b[N][N];
```

die für **a** und **c** zu einer zeilenzyklischen Verteilung und für **b** zu einer spaltenzyklischen Verteilung führen.

6.5.6 UPC Synchronisation

UPC enthält keine implizite Synchronisation zwischen Threads, bietet jedoch verschiedene Mechanismen zur expliziten Synchronisation von Threads in Form von Barrier- und Sperranweisungen. Für die globale Barrier-Synchronisation werden zwei Formen bereitgestellt. Die Anweisung **upc_barrier** bewirkt eine Blockierung des ausführenden Threads bis alle anderen Threads ankommen. Neben dieser blockierenden Barrier-Synchronisation gibt es die Möglichkeit einer nichtblockierenden Barrier-Synchronisation durch Verwendung der Anweisungen **upc_notify** und **upc_wait**. Durch die nichtblockierende Anweisung **upc_notify** beginnt die nichtblockierende Barrier-Synchronisation für den ausführenden Thread, die durch die blockierenden Anweisung **upc_wait** abgeschlossen wird. Dazwischen können beliebige Anweisungen durch den Thread ausgeführt werden, die unabhängig von der Barrier-Synchronisation sind. Der Thread blockiert so lange bis alle anderen Threads die **upc_notify** Anweisungen derselben Barrier-Synchronisation erreicht haben. Die Anweisungen zur globalen

```

#include <upc.h>
#include <upc_strict.h>
#define N 8
#define P 8
#define M 8

shared [N*N /THREADS] int a[N][N];
shared [M /THREADS] int b[N][N];
shared [N*N /THREADS] int c[N][N];

int main(void) {
    int i,j,l;
    // array initialization by thread 0
    if (MYTHREAD==0) {
        for (i=0;i<N;i++)
            for (j=0;j<P;j++) a[i][j] = 1;
        for (i=0;i<P;i++)
            for (j=0;j<M;j++) b[i][j] = 2;
    }
    upc_barrier;
    // all threads perform matrix multiplication
    upc_forall (i=0;i<N;i++;&a[i][0])
    // &a[i][0] specifies that this iteration will be executed
    // by the thread that has affinity to element a[i][0]
    for (j=0; j<M; j++) {
        c[i][j] = 0;
        for (l=0; l< P; l++) c[i][j] +=a[i][l]*b[l][j];
        printf("Thread %d has computed c[%d][%d]=%d\n",
               MYTHREAD,i,j,c[i][j]);
    }
    upc_barrier;
    // thread 0 displays results
    if (MYTHREAD == 0) {
        printf("\n\n");
        for(i=0;i<N;i++)
            for(j=0;j<M;j++)
                printf("c[%d][%d]=%d\n",i,j,c[i][j]);
    }
}

```

Abb. 6.43. Matrixmultiplikation in UPC.

Barrier-Synchronisation sind kollektive Operationen, die von allen beteiligten Threads ausgeführt werden. Jeder Thread führt eine Folge von sich abwechselnden `upc_notify` und `upc_wait` Anweisungen aus, so dass eine Folge von Synchronisationsphasen entsteht.

Zum Schutz gemeinsamer Variablen stellt UPC einen Synchronisationsmechanismus durch Sperranweisungen und spezielle Sperrvariablen (engl. *lock variables*) bereit. Der Typ einer Sperrvariable ist `upc_lock_t`. Dieser Typ ist ein vom Programmierer nicht zugreifbarer UPC-Typ. Variablen vom Typ

`upc_lock_t` sind gemeinsame Variablen und können die Zustände *gesperrt* (engl. *locked*) oder *frei* (engl. *unlocked*) haben. Die Anweisung

```
upc_lock_t *upc_all_lock_alloc (void);
```

allokiert dynamisch eine Sperrvariable im Zustand *unlocked* und gibt einen Zeiger auf diese Sperrvariable zurück. Diese Anweisung ist eine kollektive Operation, so dass der zurückgegebene Zeiger für jeden Thread auf dieselbe Sperrvariable zeigt. Die Anweisung

```
upc_lock_t *upc_global_lock_alloc (void);
```

ist eine Funktion, die eine Sperrvariable allokiert (im Zustand *unlocked*) und einen Zeiger auf diese Sperrvariable an den aufrufenden Thread zurückgibt, aber nicht kollektiv ist. Wird diese Anweisung von mehreren Threads ausgeführt, so werden verschiedene Allokationen durchgeführt. Durch die Funktion

```
void upc_lock_free (upc_lock_t *ptr);
```

wird die dynamisch allokierte Sperrvariable, auf die `ptr` zeigt, im Speicher freigegeben.

Zur Programmierung im Stil des gemeinsamen Adressraums können kritische Bereiche durch Sperrvariablen geschützt werden, indem diese geeignet in die Zustände *locked* oder *unlocked* gesetzt werden. Die Funktion

```
void upc_lock (upc_lock_t *ptr);
```

setzt den Zustand der Sperrvariablen, auf den `ptr` zeigt, auf *locked*. Ist diese Sperrvariable bereits durch einen anderen Thread gesperrt, so wartet der aufrufende Thread bis ein anderer Thread die Sperrvariablen wieder freigibt. Ist diese Sperrvariable bereits durch den aufrufenden Thread gesperrt, so ist das Ergebnis nicht definiert. Die Implementierung von `upc_lock()` enthält keine Vorkehrungen zur Erkennung von Deadlocks. Die Vermeidung von Deadlocks liegt vollständig in der Verantwortung des Programmierers. Die Funktion

```
void upc_lock_attempt (upc_lock_t *ptr);
```

bewirkt, dass die Sperrvariable, auf die `ptr` zeigt, auf den Zustand *locked* für den aufrufenden Thread gesetzt wird, falls diese bisher nicht gesperrt war. In diesem Fall wird 1 zurückgegeben, andernfalls 0. Die Funktion

```
void upc_unlock (upc_lock_t *ptr);
```

setzt den Zustand der Sperrvariablen, auf den `ptr` zeigt, auf *unlocked*, falls diese vorher durch den aufrufenden Thread gesperrt war. Andernfalls ist das Ergebnis nicht definiert.

7. Lösung linearer Gleichungssysteme

Das Lösen linearer Gleichungssysteme spielt eine zentrale Rolle im wissenschaftlichen Rechnen, da viele physikalische oder andere naturwissenschaftliche Phänomene durch gewöhnliche und partielle Differentialgleichungen modelliert werden, deren numerische Lösung oft auf lineare Gleichungssysteme zurückgeführt wird. Dies ist etwa der Fall beim Lösen partieller Differentialgleichungen durch Diskretisierungsverfahren. In diesem Kapitel gehen wir auf grundlegende Verfahren zur Lösung von linearen Gleichungssystemen ein und beschreiben deren parallele Realisierung. Dabei stellen wir die Implementierung für Rechner mit verteiltem Adressraum in den Vordergrund.

Im folgenden Kapitel sei $A \in \mathbb{R}^{n \times n}$ eine $(n \times n)$ -Matrix mit reellen Werten und $b \in \mathbb{R}^n$ sei ein n -dimensionaler Vektor. Gesucht ist ein n -dimensionaler Vektor $x \in \mathbb{R}^n$, der das lineare Gleichungssystem

$$Ax = b \tag{7.1}$$

löst. Anders ausgedrückt bedeutet dies, dass der Vektor b als Linearkombination der Spaltenvektoren der Matrix A dargestellt werden soll. Die Koeffizienten dieser Linearkombination sind gerade die Komponenten des gesuchten Vektors x . Das Gleichungssystem $Ax = b$ ist lösbar, wenn A nichtsingulär ist, also die inverse Matrix A^{-1} existiert, welche die Eigenschaft $A \cdot A^{-1} = I$ mit Einheitsmatrix I erfüllt. Äquivalent zu dieser Bedingung ist, dass die Determinante der Matrix A nicht Null ist, siehe z.B. [61] oder [144].

Bei den Lösungsverfahren für lineare Gleichungssysteme wird zwischen direkten und iterativen Verfahren unterschieden. *Direkte* Verfahren berechnen eine (bis auf Rundungsfehler) exakte Lösung in einer vorgegebenen, von der Größe der Matrix abhängigen Anzahl von Berechnungsschritten. *Iterative* Verfahren berechnen eine Näherung der Lösung, indem sie ausgehend von einem Startwert eine Folge von Vektoren erzeugen, wobei jeweils ein Vektor als Ausgangspunkt für die Berechnung des nächsten Vektors genutzt wird. Dadurch wird keine exakte Lösung berechnet, die berechnete Folge der Näherungen konvergiert jedoch asymptotisch gegen diese exakte Lösung. Iterative Verfahren sind oft schneller als direkte Verfahren und sind leichter zu implementieren, insbesondere für Parallelrechner. Sie haben aber den Nachteil, dass sie nur anwendbar sind, wenn das Gleichungssystem bestimmte Bedingungen erfüllt, die vom verwendeten Lösungsverfahren abhängen. Direkte Methoden sind dagegen fast immer anwendbar. Für dünnbesetzte Matrizen,

also bei sehr vielen Matrixeinträgen mit Wert Null, sind iterative Verfahren vorzuziehen, da direkte Verfahren ohne besondere Maßnahmen meist zum Auffüllen der Matrix (engl. *fill-in*) und damit zu vielen zusätzlichen Operationen führen. Bei den direkten Verfahren unterscheidet man zwischen *Orthogonalisierungsverfahren* und *Eliminationsverfahren*. Wir werden hier nur Eliminationsverfahren behandeln. Für eine Darstellung von Orthogonalisierungsverfahren und ihrer parallelen Implementierung verweisen wir auf [61].

Wir beginnen dieses Kapitel in Abschnitt 7.1 mit der Behandlung des Gaußschen Eliminationsverfahrens für allgemeine lineare Gleichungssysteme und beschreiben eine zeilenzyklische und eine gesamtzyklische parallele Implementierung. Da bei der numerischen Behandlung von physikalischen Problemen oft lineare Gleichungssysteme mit einer speziellen Gestalt entstehen, betrachten wir in Abschnitt 7.2 die Anwendung des Gaußschen Eliminationsverfahrens auf Tridiagonalsysteme und Systeme mit allgemeiner, regelmäßiger Bandstruktur. Insbesondere beschreiben wir Ansätze für eine parallele Implementierung mit Hilfe zyklischer Reduktion. Abschnitt 7.3 widmet sich den iterativen Lösungsverfahren für lineare Gleichungssysteme und möglicher paralleler Realisierungen. In Abschnitt 7.5 wird die Methode der konjugierten Gradienten behandelt.

7.1 Gauß-Elimination

7.1.1 Beschreibung der Methode

Wir betrachten das lineare Gleichungssystem $Ax = b$ mit Matrix $A \in \mathbb{R}^{n \times n}$ und Vektor $b \in \mathbb{R}^n$, für das wir in ausgeschriebener Form die Notation

$$\begin{array}{cccc} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n & = & b_1 \\ \vdots & & \vdots \\ a_{i1}x_1 + a_{i2}x_2 + \dots + a_{in}x_n & = & b_i \\ \vdots & & \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n & = & b_n \end{array}$$

benutzen. Die Gauß-Elimination transformiert die Matrix $A^{(1)} := A = (a_{ij})$ und den Vektor $b^{(1)} := b = (b_i)$ schrittweise zu Matrizen $A^{(2)}, \dots, A^{(n)}$ und Vektoren $b^{(2)}, \dots, b^{(n)}$, wobei gilt, dass das lineare Gleichungssystem $A^{(k)}x = b^{(k)}$ für $k = 2, \dots, n$ die gleiche Lösung wie das Originalsystem $Ax = b$ hat. Die Matrix $A^{(k)}$ zeichnet sich dadurch aus, dass in den ersten $k - 1$ Spalten unter den Diagonalelementen nur Nullen stehen, d.h. $A^{(k)}$ hat die Gestalt

$$A^{(k)} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1,k-1} & a_{1k} & \cdots & a_{1n} \\ 0 & a_{22}^{(2)} & \cdots & a_{2,k-1}^{(2)} & a_{2k}^{(2)} & \cdots & a_{2n}^{(2)} \\ \vdots & \ddots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \vdots & & \ddots & a_{k-1,k-1}^{(k-1)} & a_{k-1,k}^{(k-1)} & \cdots & a_{k-1,n}^{(k-1)} \\ \vdots & & & 0 & a_{kk}^{(k)} & \cdots & a_{kn}^{(k)} \\ \vdots & & & \vdots & \vdots & \ddots & \vdots \\ \vdots & & & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & \cdots & 0 & a_{nk}^{(k)} & \cdots & a_{nn}^{(k)} \end{bmatrix}.$$

Die letzte Matrix $A^{(n)}$ hat damit die Form einer oberen Dreiecksmatrix. Die Matrizen $A^{(k)}$ und Vektoren $b^{(k)}$ werden in der *Vorwärtselimination* errechnet. Die Matrix $A^{(k+1)}$ und der Vektor $b^{(k+1)}$ werden aus $A^{(k)}$ und $b^{(k)}$ erzeugt, $k = 1, \dots, n-1$, indem geeignete Vielfache der k -ten Zeile von $A^{(k)}$ bzw. des k -ten Elementes von $b^{(k)}$ zu den Zeilen $k+1, k+2, \dots, n$ bzw. den Elementen $b_{k+1}^{(k)}, b_{k+2}^{(k)}, \dots, b_n^{(k)}$ addiert werden. Dabei sind die Vielfachen so gewählt, dass die Unbekannte x_k aus den Gleichungen $k+1, k+2, \dots, n$ eliminiert wird, d.h. dass die Koeffizienten von x_k in den Gleichungen $k+1, k+2, \dots, n$ Null werden. Dazu werden $n-k$ Eliminationsfaktoren

$$l_{ik} = a_{ik}^{(k)} / a_{kk}^{(k)}, \quad i = k+1, \dots, n, \quad (7.2)$$

berechnet und zur Neuberechnung der Zeilen $k+1, \dots, n$ benutzt, indem das l_{ik} -fache der k -ten Zeile von der i -ten Zeile subtrahiert wird, d.h. man berechnet

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - l_{ik} a_{kj}^{(k)} \quad (7.3)$$

$$b_i^{(k+1)} = b_i^{(k)} - l_{ik} b_k^{(k)} \quad (7.4)$$

für $k < j \leq n$ und $k < i \leq n$. Nach $n-1$ Eliminationsschritten erhält man $A^{(n)}$ als obere Dreiecksmatrix. Die Komponenten des Ergebnisvektors x werden durch *Rückwärtseinsetzen* aus dem gestaffelten Gleichungssystem $A^{(n)}x = b^{(n)}$ in der Reihenfolge $x_n, x_{n-1}, \dots, x_1$ bestimmt, indem für $k = n, n-1, \dots, 1$ die Berechnung

$$x_k = \frac{1}{a_{kk}^{(n)}} \left(b_k^{(n)} - \sum_{j=k+1}^n a_{kj}^{(n)} x_j \right) \quad (7.5)$$

durchgeführt wird. Abbildung 7.1 zeigt eine Programmskizze einer sequentiellen Implementierung. Dabei wird die innere Schleife zur Neuberechnung der Matrixelemente ungefähr $\sum_{k=1}^n k^2 = \frac{1}{6}n(n+1)(2n+1) \approx n^3/3$ Mal durchlaufen, die asymptotische Laufzeit ist also $O(n^3)$.

LR-Zerlegung oder Dreieckszerlegung. Die Matrix A lässt sich als Produkt der oberen Dreiecksmatrix $R := A^{(n)}$ und der aus den errechneten Eliminationsfaktoren bestehenden unteren Dreiecksmatrix

```

double *gauss_sequential (double **a, double *b)
{
    double *x, sum, l[MAX_SIZE];
    int i,j,k,r;

    x = (double *) malloc(n * sizeof(double));
    for (k = 0; k < n-1; k++) {
        r = max_col(a,k);
        if (k != r) exchange_row(a,b,r,k);
        for (i=k+1; i < n; i++) {
            l[i] = a[i][k]/a[k][k];
            for (j=k+1; j < n; j++)
                a[i][j] = a[i][j] - l[i] * a[k][j];
            b[i] = b[i] - l[i] * b[k];
        }
    }
    for (k = n-1; k >= 0; k--)
        sum = 0.0;
        for (j=k+1; j < n; j++)
            sum = sum + a[k][j] * x[j];
        x[k] = 1/a[k][k] * (b[k] - sum);
    }
    return x;
}

```

Abb. 7.1. Programmskizze in C-Notation eines sequentiellen Programmes zur Lösung von $Ax = b$ mit Hilfe der Gauß-Elimination. Die Matrix A wird durch das Feld a realisiert, der Vektor b durch das Feld b . Die Nummerierung im Programm beginnt wieder bei 0. Die Funktionen $\text{max\_col}(a,k)$ und $\text{exchange\_row}(a,b,r,k)$ dienen der Realisierung der Pivotsuche. Die Funktion $\text{max\_col}(a,k)$ liefert den Index r mit $|a_{rk}| = \max_{k \leq s \leq n} (|a_{sk}|)$. Die Funktion $\text{exchange\_row}(a,b,r,k)$ vertauscht die Zeilen r und k von A und die zugehörigen Elemente b_r und b_k der rechten Seite.

$$L = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ l_{21} & 1 & 0 & \dots & 0 \\ l_{31} & l_{32} & 1 & & 0 \\ \vdots & \vdots & \ddots & \ddots & 0 \\ l_{n1} & l_{n2} & l_{n3} & \dots & l_{n,n-1} & 1 \end{bmatrix}$$

darstellen. Diese Darstellung $A = L \cdot R$ wird Dreieckszerlegung oder LR-Zerlegung genannt. Wenn nur die LR-Zerlegung einer Matrix bestimmt werden soll, braucht die rechte Seite b des Gleichungssystems zunächst nicht nach Gleichung (7.4) mittransformiert zu werden. Wegen $Ax = LA^{(n)}x = Ly = b$ mit $y = A^{(n)}x$ kann die Lösung von $Ax = b$ für jedes $b \in \mathbb{R}^n$ nach Bestimmung der Dreieckszerlegung in zwei Schritten durchgeführt werden. Im ersten Schritt wird y durch Lösung des unteren Dreieckssystems $Ly = b$ mit *Vorwärtssubstitution* bestimmt, wobei $y = b^{(n)}$ ist, d.h. die Vorwärtssubstitution entspricht der Berechnung von $b^{(n)}$ nach Gleichung (7.4). Im zweiten

Schritt wird x als Lösung des gestaffelten Gleichungssystems $A^{(n)}x = y$ durch *Rückwärtssubstitution* ermittelt. Die Formulierung als LR-Zerlegung hat also den Vorteil, dass nach Bestimmung von L und $A^{(n)}$ die Lösung des linearen Gleichungssystems für mehrere rechte Seiten mit den obigen zwei Schritten durchgeführt werden kann, ohne dass der gesamte Algorithmus der Gauß-Elimination wiederholt werden muss.

Pivotsuche. Die Berechnung der LR-Zerlegung entspricht der Vorwärtselimination der Gauß-Elimination. Diese Berechnung funktioniert jedoch nur, wenn die Elemente $a_{kk}^{(k)}$ nicht Null sind, d.h. auch wenn $\det A \neq 0$ gilt und das Gleichungssystem lösbar ist, existiert nicht immer eine Zerlegung der Form $A = LR$ von A . Es gibt jedoch immer eine durch Zeilenpermutation aus A entstehende Matrix, für die es eine Dreieckszerlegung gibt, für die also gilt $BA = LR$ mit einer Permutationsmatrix B . Die Bestimmung einer solchen Permutation wird zusammen mit der Vorwärtselimination durchgeführt, indem in jedem Eliminationsschritt ein **Pivotelement** bestimmt wird, das $a_{kk}^{(k)}$ ersetzt. Es kann vorkommen, dass die Pivotelemente zwar von Null verschieden, aber sehr klein sind. In diesem Fall können die errechneten Eliminationsfaktoren l_{ik} sehr groß werden, was zu Ungenauigkeiten bei der Berechnung der Matrixeinträge führen kann, siehe z.B. [154] für eine ausführliche Behandlung. Um auch in diesen Fällen ein gutes Funktionieren der Gauß-Elimination sicherzustellen, verwendet man *Pivotstrategien*. Die am häufigsten verwendete Pivotstrategie ist die **Spaltenpivotsuche**. Diese sucht vor der Berechnung der Zahlen l_{ik} nach Gleichung (7.2) unter den Zahlen $a_{kk}^{(k)} \dots a_{nk}^{(k)}$ der k -ten Spalte die betragsgrößte Zahl $a_{rk}^{(k)}$, $k \leq r \leq n$, und vertauscht im Falle $r \neq k$ die Zeilen k und r der Matrix $A^{(k)}$ und die zugehörigen Einträge $b_k^{(k)}$ und $b_r^{(k)}$. Eine Alternative ist die **Zeilenpivotsuche**, die unter den Zahlen $a_{kk}^{(k)} \dots a_{kn}^{(k)}$ der k -ten Zeile die betragsgrößte Zahl $a_{kr}^{(k)}$ bestimmt und im Falle $r \neq k$ die Spalten k und r vertauscht. Dies entspricht einem Vertauschen der Nummerierung der Unbekannten x_k und x_r .

Sowohl Spaltenpivotstrategie als auch Zeilenpivotstrategie sind Beispiele für *Teilpivotstrategien*. Unter *Totalpivotstrategie* versteht man die Suche nach dem betragsgrößten Element der gesamten Restmatrix $\tilde{A}^{(k)} = (a_{ij}^{(k)})$ mit $k \leq i, j \leq n$. Bei der Anwendung einer Totalpivotstrategie müssen evtl. Zeilen und Spalten vertauscht werden. In der Praxis vermeidet man üblicherweise die Totalpivotstrategie, weil sie eine höhere Berechnungszeit hat als die erwähnten Teilpivotstrategien, und weil sie spezielle Strukturen einer Matrix, wie z.B. die Bandstruktur einer Tridiagonalmatrix, zerstören kann [154]. Bei der Spaltenpivotsuche vermeidet man für eine praktische Realisierung ein aufwendiges Vertauschen der Zeilen. Dies kann durch Verwendung eines Indexvektors geschehen, in dem die Vertauschungen vermerkt werden und der für den Zugriff auf die Zeilen der Matrix verwendet wird [144]. Dies hat zwar den Nachteil, dass alle Feldzugriffe über den Indexvektor laufen, der zusätzliche Zeitaufwand ist aber üblicherweise geringer als die Zeit für ein explizites

Vertauschen der einzelnen Elemente der Zeilen. Eine effizientere Implementierung kann (wenn die verwendete Programmiersprache dies erlaubt) durch Verwendung einer dynamischen Datenstruktur erreicht werden, welche die Matrix in Form einzelner Zeilen der Länge n ablegt, die über ein Feld der Länge n mit Zeigern auf die einzelnen Zeilen adressiert werden. In dieser Datenstruktur kann ein Vertauschen der Zeilen durch ein einfaches Vertauschen von Zeigern erreicht werden, die Matrixelemente können aber wie in einem zweidimensionalen Feld adressiert werden.

7.1.2 Parallele zeilenzyklische Implementierung

Entscheidend für eine parallele Implementierung der Gauß-Elimination ist die Festlegung der Datenverteilung für die Matrix A , da diese die von den einzelnen Prozessoren durchzuführenden Berechnungen und die resultierende Kommunikation stark beeinflusst, vgl. Abschnitt 3.6. Dabei ist eine blockweise Ablage der Zeilen oder Spalten ungeeignet, weil sie zu einer schlechten Lastverteilung führt: Wenn etwa eine blockweise zeilenorientierte Datenverteilung vorliegt, d.h. Prozessor P_q für $1 \leq q \leq p$ die Zeilen $n/p \cdot (q-1) + 1, \dots, n/p \cdot q$ abspeichert, so hat Prozessor P_q nach der Berechnung von $A^{(k)}$ mit $k = q \cdot n/p + 1$ für die restlichen Zwischenmatrizen keine Berechnungen mehr durchzuführen. Ein Ungleichheit der Lastverteilung wird bei einer zyklischen Ablage der Zeilen vermieden, bei der Prozessor P_q für $1 \leq q \leq p$ die Zeilen $q, q+p, q+2p, \dots$ erhält, d.h. alle Zeilen i mit $1 \leq i \leq n$, für die $q = ((i-1) \bmod p) + 1$ gilt.

Wir beschreiben im Folgenden eine parallele Implementierung mit Spaltenpivotsuche und zeilenzyklischer Verteilung von A bzw. zyklischer Verteilung von b [130, 132]. Zur Beschreibung der Arbeitsweise der parallelen Implementierung betrachten wir die Berechnung der Zwischenmatrix $A^{(k+1)}$ und der rechten Seite $b^{(k+1)}$ aus $A^{(k)}$ und $b^{(k)}$ für $1 \leq k \leq n-1$. Für die Vorwärtselimination werden nacheinander die folgenden Schritte durchgeführt:

1. **Bestimmung des lokalen Pivotelementes:** Jeder Prozessor bestimmt unter seinen Elementen der Spalte k in den Zeilen $k, \dots, n$ das betragsgrößte Element durch lokale Vergleiche.
2. **Bestimmung des globalen Pivotelementes:** Die lokalen Pivotelemente werden mit einer *Einzel-Akkumulationsoperation* mit Maximumsreduktion aufgesammelt, d.h. mit Hilfe einer globalen Kommunikationsoperation wird das betragsgrößte Element aller Prozessoren bestimmt und bei einem ausgewählten Wurzelprozessor abgespeichert. Dieser Prozessor teilt den aufgesammelten Wert danach mit Hilfe einer *Einzel-Broadcastoperation* allen anderen Prozessoren mit.
3. **Vertauschen der Pivotzeile:** Sei $a_{rk}^{(k)}$ das Pivotelement. Sei weiter P_q der Prozessor, der Zeile k enthält und sei $P_{q'}$ der Prozessor, der Zeile r enthält. Falls $q \neq q'$ gilt, werden die Zeilen k und r und die korrespondierenden Elemente der rechten Seite mit Hilfe einer Einzeltransferoperation

vertauscht. Falls $q = q'$, aber $k \neq r$ gilt, kann das Vertauschen durch eine lokale Operation von Prozessor P_q durchgeführt werden. Die Elemente b_k und b_r des Vektors b werden entsprechend vertauscht.

4. **Verteilen der Pivotzeile:** Nach Bestimmung des Pivotelementes kann die Pivotzeile an alle Prozessoren verschickt werden, damit diese ihre lokalen Eliminationsoperationen ausführen können. Dazu führt der Prozessor P_q , der Zeile k besitzt, eine Einzel-Broadcastoperation durch, mit deren Hilfe er die Elemente $a_{kk}^{(k)}, \dots, a_{kn}^{(k)}$ von Zeile k und das Element $b_k^{(k)}$ der rechten Seite allen anderen Prozessoren mitteilt.
5. **Berechnung der Eliminationsfaktoren:** Jeder Prozessor berechnet lokal die Eliminationsfaktoren l_{ik} für seine lokalen Zeilen i nach (7.2).
6. **Neuberechnung der Matrixelemente:** Jeder Prozessor berechnet mit seinen lokalen Elementen von $A^{(k)}$ und $b^{(k)}$ die neuen Elemente von $A^{(k+1)}$ und $b^{(k+1)}$ gemäß (7.3) und (7.4).

Das Rückwärtseinsetzen zur Bestimmung der n Komponenten des Ergebnisvektors wird in n Schritten durchgeführt, wobei in Schritt k der Prozessor P_q , der Zeile k abspeichert, den Wert von x_k nach Formel (7.5) berechnet und diesen Wert durch eine Einzel-Broadcastoperation an alle anderen Prozessoren verteilt. Die Phase der Rückwärtssubstitution führt also durch Datenabhängigkeiten zu Sequentialisierungen.

Eine Programmskizze einer Implementierung der Punkte 1.–6. mit MPI-Kommunikationsanweisungen ist in Abbildung 7.2 wiedergegeben, wobei z.T. die in Abbildung 7.1 eingeführten Funktionen verwendet werden. Dabei gehen wir davon aus, dass das Programm im SPMD-Stil von den beteiligten Prozessoren abgearbeitet wird. In der Variable `me` hat jeder Prozessor seine eigene Prozessornummer abgelegt. In Abhängigkeit von `me` und insbesondere der Zuordnung von Zeile k und der Pivotzeile zu Prozessoren werden mit demselben Programmtext unterschiedliche Berechnungen durchgeführt. Die globalen Variablen `n` und `p` enthalten die Größe des Gleichungssystems bzw. die Anzahl der ausführenden Prozessoren. Die Funktion `max_col_loc(a,k)` führt Punkt 1. aus, bestimmt also lokal den Index `r`, für den gilt, dass `a[r][k]` das betragsgrößte der lokalen Elemente in Spalte `k` ab Zeile `k` ist, die der ausführende Prozessor abspeichert. Wenn der ausführende Prozessor ab Zeile `k` keine Elemente von Spalte `k` mehr abspeichert, liefert die Funktion `-1` zurück.

Das oben unter Punkt 2. beschriebene Aufsammeln der lokalen Pivotelemente kann in MPI mit einer `MPI_Allreduce()`-Anweisung realisiert werden, die im wesentlichen eine Einzel-Akkumulationsoperation mit einer nachfolgenden Einzel-Broadcastoperation realisiert. Das global maximale Element wird durch Aufruf dieser Funktion mit der MPI-Reduktionsoperation `MPI_MAXLOC` bestimmt, die auf dem speziellen Datentyp `MPI_DOUBLE_INT`, bestehend aus einem Double- und einem Integer-Wert, ausgeführt wird, vgl. Abschnitt 5.1.2. Im Programm enthält `y` vom Typ `MPI_DOUBLE_INT` in `y.val` ein Spaltenelement und in `y.node` den Prozessor, der die zugehörige Zeile

```

double *gauss_cyclic (double **a, double *b)
{
    double *x, l[MAX_SIZE], *buf;
    int i,j,k,r, tag=42;
    MPI_Status status;
    struct { double val; int node; } z,y;

    x = (double *) malloc(n * sizeof(double));
    buf = (double *) malloc((n+1) * sizeof(double));
    for (k=0 ; k<n-1 ; k++) {
        r = max_col_loc(a,k);
        z.node = me;
        if (r != -1) z.val = fabs(a[r][k]); else z.val = 0.0;
        MPI_Allreduce(&z,&y,1,MPI_DOUBLE_INT,MPI_MAXLOC,MPI_COMM_WORLD);
        if (k % p == y.node) { /* Pivotzeile und Zeile k auf einem Prozessor */
            if (k % p == me) {
                if (a[k][k] != y.val) exchange_row(a,b,r,k);
                copy_row(a,b,k,buf);
            }
        }
        else /* Pivotzeile und Zeile k auf unterschiedlichen Prozessoren */
            if (k % p == me) {
                copy_row(a,b,k,buf);
                MPI_Send(buf+k,n-k+1,MPI_DOUBLE,y.node,tag,
                        MPI_COMM_WORLD);
            }
            else if (y.node == me) {
                MPI_Recv(buf+k,n-k+1,MPI_DOUBLE,MPI_ANY_SOURCE,
                        tag,MPI_COMM_WORLD,&status);
                copy_exchange_row(a,b,r,buf);
            }
        MPI_Bcast(buf+k,n-k+1,MPI_DOUBLE,y.node,MPI_COMM_WORLD);
        if ((k % p != y.node) && (k % p == me)) copy_row(a,b,buf,k);
        i = k+1; while (i % p != me) i++;
        for ( ; i<n; i+=p) {
            l[i] = a[i][k] / buf[k];
            for (j=k+1; j<n; j++)
                a[i][j] = a[i][j] - l[i]*buf[j];
            b[i] = b[i] - l[i]*buf[n];
        }
    }
    for (k=n-1; k>=0; k--) {
        if (k % p == me) {
            sum = 0.0;
            for (j=k+1; j < n; j++) sum = sum + a[k][j] * x[j];
            x[k] = 1/a[k][k] * (b[k] - sum);
            MPI_Bcast(&x[k],1,MPI_DOUBLE,k%p,MPI_COMM_WORLD);
        }
    }
    return x;
}

```

Abb. 7.2. Programmfragment zur Gauß-Elimination mit zeilenzyklischer Verteilung.

besitzt. Die Mitgabe der Prozessornummer bewirkt, dass `y.node` nach dieser Operation auf jedem Prozessor die Nummer des Prozessors enthält, der das maximale Element hat. Wenn dies der Prozessor ist, der Zeile k abspeichert (d.h. es gilt $k\%p == y.node$), werden nach Punkt 3. die Zeilen r und k von A und die Elemente r und k von b von diesem Prozessor lokal vertauscht, falls $r \neq k$ gilt. Man beachte, dass danach Zeile k die Pivotzeile enthält. Ansonsten wird das Vertauschen der Zeilen r und k und der zugehörigen Elemente der rechten Seite b durch eine Einzeltransferoperation, die die Zeile k und das Element b_k vom Prozessor mit der Prozessornummer $k\%p$ zum Prozessor mit der Prozessornummer `y.node` schickt, und einer nachfolgenden Broadcastoperation mit Prozessor `y.node` als Wurzel realisiert. Damit die Zeile k und das Element b_k mit einer einzelnen Transferoperation verschickt werden können, werden die besetzten Elemente der k -ten Zeile von A und b_k vor dem Versenden mit `copy_row(a,b,k,buf)` in einen zusammenhängenden Puffer `buf` kopiert, der als Sendepuffer dient. Nach beiden Fällen des Punktes 3. wird die Pivotzeile auf dem Prozessor mit Nummer `y.node` in den Puffer `buf` kopiert. Dies geschieht mit `copy_row(a,b,k,buf)` für $k\%p == y.node$ und mit `copy_exchange_row(a,b,r,buf)` sonst. Der letztere Aufruf kopiert zusätzlich den vorherigen Inhalt von `buf` in die zugehörigen Einträge von Zeile r und b_r und schließt damit das Vertauschen ab. Der für das Kopieren benutzte Wert von `r` ist dem Prozessor aus seiner lokalen Berechnung von `max_col_loc(a,k)` bekannt. Der Puffer `buf` wird mit einer `MPI_Bcast()`-Anweisung verteilt. Die Aktualisierung der Restmatrix führt jeder Prozessor für die Zeilen durch, die er lokal abspeichert, wobei die Pivotzeile in `buf` benutzt wird. Dabei beginnt Prozessor P_q mit der ersten Zeile $i > k$, für die $i\%p == me$ gilt.

Für eine effiziente parallele Implementierung kann eine verteilte Variante der oben erwähnten Ablage der Matrix verwendet werden, die eine Datenstruktur aus einem Feld mit n Zeigern benutzt, wobei jeder Zeiger auf eine Zeile der Matrix weist. Damit ist ein einfaches Vertauschen von Zeilen durch Umhängen von Zeigern möglich. Dies kann z.B. von den Funktionen `exchange_row()` und `copy_exchange_row()` in Abbildung 7.2 ausgenutzt werden. Jeder Prozessor P_q hält das gesamte Zeigerfeld in seinem lokalen Speicher, hat aber nur die Zeilen i mit $i \bmod p = q$ belegt, d.h. die Zeilen, die er lokal abspeichert, siehe Abbildung 7.3. Alle anderen Einträge des Zeigerfeldes sind mit `NULL` besetzt. Um zu vermeiden, dass jeder Prozessor das gesamte Feld `b` allokieren muss, kann für jede belegte Zeile ein Feld der Länge $n + 1$ allokiert werden, wobei n Einträge für die Matrixelemente der abzulegenden Zeile und ein Eintrag für das korrespondierende Element der rechten Seite verwendet wird. Dies hat auch den Vorteil, dass die Aktualisierung der Elemente der rechten Seite nicht gesondert behandelt werden muss, sondern dadurch erledigt werden kann, dass die Schleife zur Aktualisierung der Matrixelemente statt bis n jetzt bis $n + 1$ läuft. Außerdem kann das Kopieren in einen zusammenhängenden Puffer für $k\%p != y.node$ vor dem Senden ver-

mieden werden und die k -te Zeile $a[k]$ kann direkt als Sendepuffer dienen. Diese Variante ist in Abbildung 7.2 *nicht* realisiert.

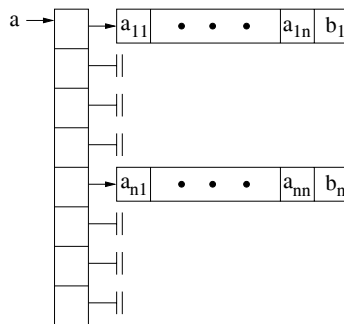


Abb. 7.3. Datenstruktur für die Implementierung der Gauß-Elimination für $n = 8$ und $p = 4$. In diesem Fall speichert jeder Prozessor zwei Zeilen ab. Die Abbildung zeigt die von Prozessor P_1 abgespeicherte Datenstruktur. Jede Zeile hat $n + 1$ Einträge, die sich aus der Zeile der Matrix und dem entsprechenden Element von b zusammensetzen.

7.1.3 Parallele gesamtyyklische Implementierung

Wir beschreiben im Folgenden eine parallele Implementierung der Gauß-Elimination mit einer beliebigen blockzyklischen schachbrettartigen Datenverteilung der Matrix A , die sich durch die parametrisierten Datenverteilungen aus Abschnitt 3.6 beschreiben lässt und die beliebige zeilen- oder spaltenzyklische Datenverteilungen als Spezialfall enthält. Diese zyklische Datenverteilung nennen wir in diesem Abschnitt zur Unterscheidung von der zeilenzyklischen Datenverteilung des letzten Teilabschnitts und zur Abkürzung auch **gesamtyyklische** Datenverteilung. Gemäß Abschnitt 3.6 wird eine parametrisierte Datenverteilung für eine Matrix durch einen Verteilungsvektor

$$((p_1, b_1), (p_2, b_2)) \quad (7.6)$$

mit $p_1 \cdot p_2 = p$ beschrieben. Dabei ist p_1 die Anzahl der Prozessoren entlang der Zeilen der Matrix, p_2 ist die Anzahl der Prozessoren entlang der Spalten. Die Zahlen b_1 und b_2 geben die für die Zeilen bzw. Spalten verwendeten Blockgrößen an. Für die Angabe der Verteilungsfunktion zum Verteilungsvektor (7.6) werden die Prozessoren durch Angabe einer Gitterfunktion $\mathcal{G} : P \rightarrow \mathbb{N}^2$ logisch in einem Gitter der Dimension $p_1 \times p_2$ angeordnet. Diese Anordnung definiert p_1 Zeilengruppen

$$R_q = \{Q \in P \mid \mathcal{G}(Q) = (q, \cdot)\}$$

mit $|R_q| = p_2$ für $1 \leq q \leq p_1$ und p_2 Spaltengruppen

$$C_q = \{Q \in P \mid \mathcal{G}(Q) = (\cdot, q)\}$$

mit $|C_q| = p_1$ für $1 \leq q \leq p_2$. Sowohl die Zeilengruppen als auch die Spaltengruppen bilden jeweils eine Partitionierung der Menge der Prozessoren P , d.h. es gilt

$$\bigcup_{q=1}^{p_1} R_q = \bigcup_{q=1}^{p_2} C_q = P$$

und $R_q \cap R_{q'} = \emptyset = C_q \cap C_{q'}$ für $q \neq q'$. Wenn man für A die zum Verteilungsvektor (7.6) gehörende Datenverteilung verwendet, ist Zeile i von A in den lokalen Adreßräumen der Prozessoren einer einzelnen Zeilengruppe abgespeichert, die wir mit $Ro(i)$ bezeichnen. Diese Zeilengruppe entspricht der Zeilengruppe R_k mit $k = \left\lfloor \frac{i-1}{b_1} \right\rfloor \bmod p_1 + 1$. Analog ist Spalte j von A in den Prozessoren einer einzelnen Spaltengruppe $Co(j)$ abgespeichert, die identisch zur Spaltengruppe C_k mit $k = \left\lfloor \frac{j-1}{b_2} \right\rfloor \bmod p_2 + 1$ ist.

Beispiel: Als Beispiel betrachten wir eine 12×12 Matrix, die auf 4 Prozessoren $\{P_1, P_2, P_3, P_4\}$ verteilt werden soll, d.h. es ist $n = 12$ und $p = 4$. Wir betrachten den Verteilungsvektor $((p_1, b_1), (p_2, b_2)) = ((2, 2), (2, 3))$, d.h. die Prozessoren werden durch die Gitterfunktion $\mathcal{G}$ logisch in einem 2×2 Gitter angeordnet, wobei wir annehmen, dass die Dimensionen des Gitters von 1 an durchnummeriert sind. Wegen $p_1 = p_2 = 2$ gibt es zwei Zeilengruppen R_1 und R_2 und zwei Spaltengruppen C_1 und C_2 mit jeweils zwei Elementen:

$$R_1 = \{Q \in P \mid \mathcal{G}(Q) = (1, j), j = 1, 2\}$$

$$R_2 = \{Q \in P \mid \mathcal{G}(Q) = (2, j), j = 1, 2\}$$

$$C_1 = \{Q \in P \mid \mathcal{G}(Q) = (j, 1), j = 1, 2\}$$

$$C_2 = \{Q \in P \mid \mathcal{G}(Q) = (j, 2), j = 1, 2\}.$$

Wegen $b_1 = 2$ und $b_2 = 3$ sind jedem Prozessor Blöcke der Größe 2×3 zugeordnet. Die Zuordnung der Blöcke an die Prozessoren entsprechend des angegebenen Verteilungsvektors ist in Abbildung 7.4 angegeben, in der die Prozessoren durch ihre Gitterdarstellung gegeben sind. So ist z.B. Zeile 5 auf die Prozessoren der Zeilengruppe R_1 verteilt, Spalte 7 ist auf die Prozessoren der Spaltengruppe C_1 verteilt. $\square$

Unter der Annahme, daß n ein Vielfaches von p_1 , p_2 , b_1 und b_2 ist, speichert jeder Prozessor elementare Blöcke der Größe $b_1 \times b_2$. Weiter bilden $p_1 \cdot p_2$ dieser elementaren Blöcke einen **Superblock**, von dem jeder der Prozessoren genau einen elementaren Block abspeichert. Insgesamt gibt es in den Zeilen $\lceil n/(p_1 b_1) \rceil$ Superblöcke, in den Spalten gibt es $\lceil n/(p_2 b_2) \rceil$ Superblöcke. Wenn wir annehmen, dass n zusätzlich zu p_1 , p_2 , b_1 und b_2 auch ein Vielfaches von $p_1 b_1$ und $p_2 b_2$ ist, sind alle elementaren Blöcke und alle Superblöcke vollständig. Wenn n kein Vielfaches von p_1 , p_2 , b_1 und b_2 ist, sind die elementaren Blöcke und die Superblöcke am rechten und unteren Rand der Matrix evtl. nicht vollständig.

1	(1,1)	(1,2)	(1,1)	(1,2)
2	(2,1)	(2,2)	(2,1)	(2,2)
3	(1,1)	(1,2)	(1,1)	(1,2)
4	(2,1)	(2,2)	(2,1)	(2,2)
5	(1,1)	(1,2)	(1,1)	(1,2)
6	(2,1)	(2,2)	(2,1)	(2,2)
7	(1,1)	(1,2)	(1,1)	(1,2)
8	(2,1)	(2,2)	(2,1)	(2,2)
9	(1,1)	(1,2)	(1,1)	(1,2)
10	(2,1)	(2,2)	(2,1)	(2,2)
11	(1,1)	(1,2)	(1,1)	(1,2)
12	(2,1)	(2,2)	(2,1)	(2,2)
	1	2	3	4

Abb. 7.4. Illustration der gesamtzyklischen Verteilung einer 12×12 Matrix.

Mit Hilfe der angegebenen Verteilungsvektoren lassen sich auch blockweise sowie zeilen- und spaltenzyklische Verteilungen beschreiben: Mit der Besetzung $p_1 = p$, $p_2 = 1$, $b_1 = 1$ und $b_2 = n$ wird eine zeilenzyklische Verteilung beschrieben wie sie in Abschnitt 7.1.2 verwendet wurde. Für $p_1 = 1$, $p_2 = p$, $b_1 = n$ und $b_2 = 1$ wird eine spaltenzyklische Verteilung beschrieben. Für $p_1 \cdot b_1 = n = p_2 \cdot b_2$ gibt es in jeder der beiden Dimensionen genau einen Superblock, d.h. es wird eine blockweise Verteilung beschrieben.

Für die im Folgenden vorgestellte parallele Implementierung der Gauß-Elimination sei eine gesamtzyklische Verteilung der Matrix A durch einen beliebigen Verteilungsvektor (7.6) gegeben. Jede gültige Besetzungen der Parameter des Verteilungsvektors stellt eine spezielle parallele Variante dar. Die veränderte Datenverteilung der Matrix bewirkt ein im Vergleich zu der in Abschnitt 7.1.2 angegebenen Implementierung verändertes Kommunikationsverhalten. Zur Verdeutlichung der Arbeitsweise der parallelen gesamtzyklischen Implementierung betrachten wir wieder eine Zwischenmatrix $A^{(k)}$. Zur Berechnung der nächsten Matrix $A^{(k+1)}$ werden nacheinander die folgenden Schritte durchgeführt, die in Anlehnung an die Implementierungsbeschreibung aus Abschnitt 7.1.2 formuliert und nummeriert sind:

1. **Bestimmung des lokalen Pivotelementes:** Die Spalte k der Matrix ist auf die Prozessoren einer Spaltengruppe $Co(k)$ verteilt. Daher berechnet jeder Prozessor in $Co(k)$ das betragsgrößte Element seines Teils der Spalte k .
2. **Bestimmung des globalen Pivotelementes:** Die Prozessoren in $Co(k)$ führen eine *gruppenorientierte* Einzel-Akkumulationsoperation durch, zu der jeder Prozessor in $Co(k)$ sein in Schritt (1) gefundenes betragsgrößtes Element beiträgt. Als Reduktionsoperation wird eine Ma-

ximumsoperation verwendet, die auch die Nummer der Pivotzeile mit übergibt. Die akkumulierten Werte werden von dem Prozessor $P_{\bar{q}}$ aufgesammelt, der das Element $a_{kk}^{(k)}$ enthält. Nach diesem Schritt kennt Prozessor $P_{\bar{q}}$ den Wert $a_{rk}^{(k)}$ und die Nummer der Pivotzeile. Diese Information kann er mit einem Einzel-Broadcast *allen* anderen Prozessoren mitteilen.

3. **Vertauschen der Pivotzeile:** Sei $a_{rk}^{(k)}$ das Pivotelement. Die Pivotzeile r ist über die Prozessoren einer Zeilengruppe $Ro(r)$ verteilt, Zeile k ist über die Prozessoren einer evtl. anderen Zeilengruppe $Ro(k)$ verteilt. Falls $Ro(r) = Ro(k)$ gilt, vertauschen die Prozessoren dieser Zeilengruppe die Elemente von Zeile r und k lokal. Falls $Ro(r) \neq Ro(k)$ ist, schickt jeder Prozessor in $Ro(k)$ seinen Teil der Zeile k an den korrespondierenden Prozessor in $Ro(r)$, d.h. an den Prozessor in $Ro(r)$, mit dem er in der gleichen Spaltengruppe ist, und umgekehrt.
4. **Verteilen der Pivotzeile:** Die Pivotzeile r ist nach Schritt 3. unter den Prozessoren von $Ro(k)$ aufgeteilt. Zur Neuberechnung der Feldelemente muss jedes Element $a_{rs}^{(k)}$, $s = k, \dots, n$, der Pivotzeile an alle Prozessoren geschickt werden, die einen Teil der Spalte s besitzen, d.h. an alle Prozessoren in $Co(s)$. Dies wird dadurch realisiert, dass jeder Prozessor in Zeilengruppe $Ro(k)$ eine *gruppenorientierte* Einzel-Broadcastoperation an alle Prozessoren seiner Spaltengruppe durchführt und dabei seinen Teil der Pivotzeile verschickt, den er in Schritt 3. erhalten hat.
5. **Berechnung der Eliminationsfaktoren:** Die Prozessoren in der Spaltengruppe $Co(k)$ berechnen lokal die Eliminationsfaktoren l_{ik} für ihre Elemente i von Spalte k nach (7.2).
- 5a. **Verteilen der Eliminationsfaktoren:** Da die Elemente von Zeile i auf die Prozessoren der Zeilengruppe $Ro(i)$ verteilt sind, muss der errechnete Eliminationsfaktor l_{ik} den anderen Prozessoren in $Ro(i)$ zugänglich gemacht werden. Dazu führt jeder Prozessor in der Spaltengruppe $Co(k)$ eine *gruppenorientierte* Einzel-Broadcastoperation durch, in der er die von ihm errechneten lokalen Eliminationsfaktoren an die anderen Prozessoren seiner Zeilengruppe $Ro(i)$ verteilt.
6. **Neuberechnung der Matrixelemente:** Jeder Prozessor berechnet mit seinen lokalen Elementen von $A^{(k)}$ und $b^{(k)}$ die neuen Elemente von $A^{(k+1)}$ und $b^{(k+1)}$ gemäß (7.3) und (7.4).

Das Rückwärtseinsetzen zur Bestimmung der n Komponenten des Ergebnisvektors wird wieder in n Schritten durchgeführt, wobei Schritt k aus den folgenden Teilschritten besteht:

1. Jeder Prozessor der Zeilengruppe $Ro(k)$ berechnet die Teilsumme von $\sum_{j=k+1}^n a_{kj}^{(n)} x_j$, die seine lokalen Elemente der Zeile k enthält.
2. Der Gesamtwert der Summe $\sum_{j=k+1}^n a_{kj}^{(n)} x_j$ wird von den Prozessoren in $Ro(k)$ durch eine *gruppenorientierte* Einzel-Akkumulationsoperation bei Prozessor $P_{\bar{q}}$ aufgesammelt, der das Element $a_{kk}^{(n)}$ abspeichert. Dabei wird eine Addition als Reduktionsoperation verwendet.

3. Prozessor P_q berechnet den Wert von x_k nach Formel (7.5).
4. Prozessor P_q verteilt den Wert von x_k an alle anderen Prozessoren durch eine Broadcastoperation.

Ein MPI-Pseudoprogramm der Implementierung ist in Abbildung 7.5 wiedergegeben, wobei wir wieder davon ausgehen, dass die Prozessoren das angegebene Programm im SPMD-Stil ausführen. Die Berechnungen entsprechen denen in Abbildung 7.2. Die Funktionen $\text{Co}(\mathbf{k})$ und $\text{Ro}(\mathbf{k})$ bezeichnen die Spalten- bzw. die Zeilengruppen, auf die Spalte $\mathbf{k}$ bzw. Zeile $\mathbf{k}$ verteilt sind. Die Funktion $\text{member}(\mathbf{me}, \text{Co}(\mathbf{k}))$ bestimmt, ob der ausführende Prozessor $\mathbf{me}$ in der Gruppe $\text{Co}(\mathbf{k})$ enthalten ist. Die Funktion $\text{grp\_leader}()$ bestimmt den ersten Prozessor einer Gruppe. Die Funktionen $\text{Cop}(\mathbf{q})$ und $\text{Rop}(\mathbf{q})$ bestimmen die Spaltengruppe bzw. die Zeilengruppe, der ein Prozessor $\mathbf{q}$ angehört. Die Funktion $\text{rank}(\mathbf{q}, \mathbf{G})$ bestimmt den lokalen Rang eines Prozessors $\mathbf{q}$ in einer Zeilen- oder Spaltengruppe $\mathbf{G}$. Zur Bestimmung des globalen Pivotelementes wird wieder eine MPI_MAXLOC -Reduktionsoperation verwendet. Im Gegensatz zu Abbildung 7.2 wird statt der Nummer des Prozessors aber die Nummer der Zeile mit dem lokal maximalen Element zur Verfügung gestellt. Der Grund dafür liegt darin, dass die Prozessoren, die Teile der Pivotzeile besitzen, wissen müssen, welche ihrer lokal abgelegten Zeilenstücke zur Pivotzeile gehört, damit sie diese für die Kommunikationsoperationen verwenden können. Da aber nur die Prozessoren in $\text{Co}(\mathbf{k})$ die Pivotzeile bestimmen, kennen alle anderen Prozessoren die Nummer der Pivotzeile nicht, wenn wir wie in Abbildung 7.2 die Prozessornummer mitakkumulieren. Die gewählte Lösung schafft Abhilfe, da nach dem globalen $\text{MPI\_Bcast}()$ nach Bestimmung der Pivotzeile jeder Prozessor deren Nummer kennt. Eine Alternative wäre eine separate Broadcastoperation auf der Zeilengruppe $\text{Ro}(\mathbf{r})$.

Wenn die Pivotzeile und Zeile $\mathbf{k}$ auf der gleichen Zeilengruppe abgelegt sind (d.h. wenn $\text{Ro}(\mathbf{k}) == \text{Ro}(\mathbf{r})$ gilt), vertauschen die Prozessoren dieser Gruppe ihre lokalen Elemente der Zeilen $\mathbf{k}$ und $\mathbf{r}$ durch $\text{exchange\_row\_loc}()$ und kopieren danach ihre lokalen Elemente der Pivotzeile (jetzt Zeile $\mathbf{k}$) mit Hilfe von $\text{copy\_row\_loc}()$ in den Puffer $\mathbf{buf}$. Dabei werden nur die Elemente ab Spalte $\mathbf{k}$ einschließlich beachtet. Die Funktionen $\text{compute\_partner}(\text{Ro}(\mathbf{r}), \mathbf{me})$ berechnet den Kommunikationspartner des aufrufenden Prozessors $\mathbf{me}$, d.h. den Prozessor $\mathbf{q} \in \text{Ro}(\mathbf{r})$, der in der gleichen Spaltengruppe wie $\mathbf{me}$ ist. Die Funktion $\text{compute\_size}(\mathbf{n}, \mathbf{k}, \text{Ro}(\mathbf{k}))$ berechnet die Anzahl der Elemente der Pivotzeile, die der aufrufende Prozessor im aktuellen Schritt ab Spalte $\mathbf{k}$ abspeichert. Diese Anzahl hängt von der Größe der Zeilengruppe $\text{Ro}(\mathbf{k})$, der verwendeten Blockgröße und der Position von $\mathbf{k}$ ab. Diese Funktion wird auch zur Berechnung der Anzahl der auszutauschenden Eliminationsfaktoren verwendet. Mit Hilfe der Funktion $\text{compute\_elim\_fact\_loc}()$ errechnet jeder Prozessor die Eliminationsfaktoren l_{ik} für alle Elemente a_{ik} , die er in seinem lokalen Speicher hält, und speichert diese in einen Puffer $\mathbf{elim\_buf}$ ab, aus dem er sie dann an alle Prozessoren der gleichen Zeilengruppe per Einzel-Broadcast mit dem zu $\text{Rop}(\mathbf{q})$

```

double * gauss_double_cyclic (double **a, double *b)
{
    double *x, *buf, *elim_buf;
    int i,j,k,r,q, ql, size, buf_size, elim_size, psz;
    struct { double val; int pvtline; } z,y;
    MPI_Status status;

    x = (double *) malloc(n * sizeof(double));
    buf = (double *) malloc((n+1) * sizeof(double));
    elim_buf = (double *) malloc((n+1) * sizeof(double));
    for (k=0; k<n-1; k++) {
        if (member(me, Co(k))) {
            r = max_col_loc(a,k);
            z.pvtline = r; z.val = fabs(a[r][k]);
            MPI_Allreduce(&z,&y,1,MPI_DOUBLE_INT,MPI_MAXLOC,comm(Co(k)));
        }
        MPI_Bcast(&y,1,MPI_DOUBLE_INT,grp_leader(Co(k)),MPI_COMM_WORLD);
        r = y.pvtline;
        if (Ro(k) == Ro(r)) {
            /*Pivotzeile und Zeile k in einer Zeilengruppe */
            if (member(me, Ro(k))) {
                if (r != k) exchange_row_loc(a,b,r,k);
                copy_row_loc(a,b,k,buf); } }
        else /* Pivotzeile und Zeile k in unterschiedl. Zeilengruppen */
            if (member(me, Ro(k))) {
                copy_row_loc(a,b,k,buf);
                q = compute_partner(Ro(r),me);
                psz = compute_size(n,k,Ro(k));
                MPI_Send(buf+k,psz,MPI_DOUBLE,q,tag,MPI_COMM_WORLD); }
            else if (member(me,Ro(r))) {
                /* ausführender Prozessor enthält Teile der Pivotzeile */
                q = compute_partner(Ro(k),me);
                psz = compute_size(n,k,Ro(r));
                MPI_Recv(buf+k,psz,MPI_DOUBLE,q,tag,MPI_COMM_WORLD,&status);
                exchange_row_loc(a,b,r,buf);
            }
    }
    for (q=0; q<p; q++)
        if (member(q,Ro(r)) && member(me,Cop(q))) {
            ql = rank(q,Cop(q)); buf_size = compute_size(n,k,Ro(k));
            MPI_Bcast(buf+k,buf_size,MPI_DOUBLE,ql,comm(Cop(q)));}
    if ((Ro(k) != Ro(r)) && (member(me,Ro(k))))
        copy_row_loc(a,b,buf,k);
    if (member(me,Co(k))) elim_buf = compute_elim_fact_loc(a,b,k,buf);
    for (q=0; q<p; q++)
        if (member(q,Co(k)) && member(me,Rop(q))) {
            ql = rank(q,Rop(q)); elim_size = compute_size(n,k,Co(k));
            MPI_Bcast(elim_buf,elim_size,MPI_DOUBLE,ql,comm(Rop(q))); }
        compute_local_entries(a,b,k,elim_buf,buf); }
    backward_substitution(a,b,x);
    return x;
}

```

Abb. 7.5. Programmfragment zur Gauß-Elimination mit gesamtzyklischer Verteilung.

gehörenden Kommunikator `comm(Rop(q))` verschickt. Der Rang der Wurzel `q` der Broadcastoperation im Kommunikator der jeweiligen Zeilen- oder Spaltengruppe `G` wird mit Hilfe der Funktion `rank(q,G)` bestimmt. Die Neuberechnung der Matrixelemente durch `compute_local_entries()` und das Rückwärtseinsetzen durch `backward_substitution()` verläuft ähnlich wie in Abbildung 7.2. Der wesentliche Unterschied zum Programm in Abbildung 7.2 besteht darin, daß zusätzliche Kommunikationsanweisungen in den Zeilen- und Spaltengruppen erforderlich sind und dass bis auf eine `MPI_Bcast()`-Operation alle globalen Kommunikationsanweisungen gruppenbezogen sind.

7.1.4 Laufzeitanalyse der gesamtzyklischen Implementierung

Nachdem wir uns im letzten Abschnitt mit der Frage beschäftigt haben, wie eine parallele Implementierung unter Verwendung einer allgemeinen gesamtzyklischen Datenverteilung zu gestalten ist, wenden wir uns jetzt der Frage zu, wie die Parameter der gesamtzyklischen Datenverteilung zu besetzen sind, um eine parallele Implementierung mit einer möglichst geringen Laufzeit zu erhalten. Dabei umfasst die beschriebene gesamtzyklische Datenverteilung verschiedene Datenverteilungen wie zeilenzyklische, spaltenzyklische und gesamtzyklische Datenverteilungen mit verschiedenen Blockgrößen und verschiedener Verteilung der Prozessoren auf die Zeilen und Spalten der Matrix als Spezialfall. Prinzipiell kann die Frage nach der besten Besetzung der Parameter der gesamtzyklischen Datenverteilung dadurch geklärt werden, dass die in Abbildung 7.5 skizzierte Implementierung für verschiedene Besetzungen der Parameter realisiert wird und dass die verschiedenen Implementierungen für einen gegebenen Parallelrechner durch Laufzeittests miteinander verglichen werden. Um einen aussagekräftigen Vergleich zu erhalten, sollte der Vergleich für Eingabematrizen verschiedener Größe durchgeführt werden. Bei einer Portierung der Implementierungen auf einen anderen Parallelrechner müssen die Laufzeittests wiederholt werden, da das Kommunikationsverhalten des Parallelrechners die resultierende Laufzeit stark beeinflussen kann. Insgesamt liefert der Vergleich mit Hilfe von Laufzeittests zwar verlässliche Ergebnisse, ist aber sehr aufwendig. Daher stellen wir in diesem Abschnitt eine alternative Methode zum Vergleich der verschiedenen Datenverteilungen vor, die an die in Abschnitt 4.4 dargestellte Analysetechnik anschließt.

Grundlage des Vergleichs ist die Beschreibung der Laufzeit des zu analysierenden Programms durch Laufzeitformeln mit maschinenspezifischen Parametern. Die Laufzeitformel zur Programmskizze in Abbildung 7.5 enthält wegen der allgemeinen Form der Datenverteilung zusätzlich die Parameter p_1 , p_2 , b_1 und b_2 , die die verwendete Datenverteilung beschreiben. Die Idee des Ansatzes besteht darin, durch Analyse der resultierenden Laufzeitformel diese Parameter so zu besetzen, dass die Laufzeitformel eine möglichst geringe Laufzeit angibt. Dabei steht nicht a priori fest, ob für verschiedene Eingabegrößen verschiedene Parameterwerte zu minimalen Laufzeiten führen. Für die

Laufzeitanalyse betrachten wir im Folgenden das Gaußsche Eliminationsverfahren *ohne* Rückwärtseinsetzen und Pivotsuche, also die Punkte 4., 5., 5a. und 6. der Berechnungsschritte im letzten Teilabschnitt. Prinzipiell könnten auch Pivotsuche und Rückwärtseinsetzen in die Analyse einbezogen werden, wir haben hier aber darauf verzichtet, um die resultierenden Laufzeitformeln überschaubar zu halten. Eine wichtige Voraussetzung für die Anwendbarkeit des beschriebenen Verfahrens besteht darin, dass die Laufzeitformeln die wirkliche Laufzeit auf einem gegebenen Parallelrechner möglichst genau beschreiben. Dies ist für viele Rechner mit verteiltem Speicher der Fall.

Im Folgenden sei für einen Prozessor P_q

$$I_q = \{(i, j) \in \{1 \dots n\} \times \{1 \dots n\} \mid P_q \text{ hat } a_{ij} \text{ als lokales Element}\}$$

die Menge der Indizes der Matrixelemente, die Prozessor P_q in seinem lokalen Speicher ablegt. Die Zeit für die Ausführung einer Einzel-Broadcastoperation mit p beteiligten Prozessoren für Nachrichten mit m **double**-Werten wird mit $T_{sb}(p, m)$ bezeichnet. Wenn wir annehmen, dass die Kommunikationsoperationen auf den disjunkten Zeilen- bzw. Spaltengruppen unabhängig und parallel zueinander ausgeführt werden können, wird die Laufzeit der Eliminationsschritte der LR-Zerlegung (also der Schritte 4., 5., 5a. und 6. des Abschnitts 7.1.3) durch die Laufzeitformel

$$\begin{aligned} T(n, p) = \sum_{k=1}^{n-1} \{ & \max_{q \in Ro(k)} T_{sb}(\#Cop(q), \#\{(k, j) \in I_q \mid j \geq k\}) \\ & + \max_{q \in Co(k)} \#\{(i, k) \in I_q \mid i > k\} \cdot t_{op} \\ & + \max_{q \in Co(k)} T_{sb}(\#Rop(q), \#\{(i, k) \in I_q \mid i > k\}) \\ & + \max_{q \in P} (\#\{(i, k) \in I_q \mid i > k\} \cdot \#\{(k, j) \in I_q \mid j > k\}) \cdot 2t_{op} \} \end{aligned} \quad (7.7)$$

beschrieben. Für eine beliebige Menge X bezeichnet dabei $\#X$ die Anzahl der Elemente in X . Der erste Term ist die Laufzeit der Einzel-Broadcastoperation für Zeile k , der zweite Term beschreibt die Laufzeit für die lokale Bestimmung der Eliminationsfaktoren, der dritte Term modelliert die Verteilung der errechneten Eliminationsfaktoren über die Zeilengruppen nach (7.2), der vierte Term beschreibt die lokale Neuberechnung der Matrixelemente nach (7.4). Der Parameter t_{op} gibt die durchschnittliche Ausführungszeit einer arithmetischen Operation auf dem verwendeten Parallelrechner an, vgl. Abschnitt 4.3. Für die Pivotsuche, die wir im Folgenden nicht weiter betrachten, würde zusätzlich die Zeit

$$\begin{aligned} T_{pivot}(n, p) = \sum_{k=1}^{n-1} \{ & \max_{q \in Co(k)} \#\{(i, k) \in I_q \mid i > k\} \cdot t_{op} \\ & + T_{m\_acc}(\#Co(k), 1) \\ & + T_{sb}(p, 1) \} \end{aligned}$$

$$+T_{s2s}(\#\{(k, j) \in I_q | j \geq k\})\}$$

gebraucht, wobei die einzelnen Terme in der Reihenfolge ihres Auftretens die Bestimmung der lokalen Pivotelemente, deren Aufsammeln in der Spaltengruppe, das Verteilen in den Zeilengruppen und den Austausch der Pivotzeile mit der aktuellen Zeile k beschreiben. Wir werden im Folgenden die in der Laufzeitformel (7.7) für die LR-Zerlegung auftretenden Größen so abschätzen, dass sie nur die Parameter der Datenverteilung, die Eingabegröße n und die Laufvariable k der Summation enthalten. Für die Anzahl der Elemente, die Prozessor P_q von Zeile k ab Spalte k besitzt, gilt:

$$\#\{(k, j) \in I_q | j \geq k\} \leq \left\lceil \frac{n-k+1}{p_2 b_2} \right\rceil b_2 \leq \left(\frac{n-k+1}{p_2 b_2} + 1 \right) b_2 = \frac{n-k+1}{p_2} + b_2. \quad (7.8)$$

Dabei ist $\left\lceil \frac{n-k+1}{p_2 b_2} \right\rceil$ die Anzahl der Superblöcke, die in den Spalten der Restmatrix ab Zeile und Spalte k vorhanden sind, b_2 ist die Anzahl der Spaltenelemente, die jeder Prozessor aus $Ro(k)$ in einem voll besetzten Superblock besitzt. Analog gilt für die Anzahl der Eliminationsfaktoren, die ein Prozessor $q \in Co(k)$ berechnet und an die Prozessoren seiner Zeilengruppe verteilt:

$$\#\{(i, k) \in I_q | i > k\} \leq \left\lceil \frac{n-k}{p_1 b_1} \right\rceil b_1 \leq \left(\frac{n-k}{p_1 b_1} + 1 \right) b_1 = \frac{n-k}{p_1} + b_1. \quad (7.9)$$

Wieder ist $\left\lceil \frac{n-k}{p_1 b_1} \right\rceil$ die Anzahl der Superblöcke, die in den Zeilen der Restmatrix vorhanden sind, b_1 ist die Anzahl der Zeilenelemente, die jeder Prozessor aus $Co(k)$ in einem voll besetzten Superblock besitzt. Damit kann die Laufzeitformel (7.7) durch

$$\begin{aligned} T(n, p) \approx \sum_{k=1}^{n-1} & \left(T_{sb}(p_1, \frac{n-k+1}{p_2} + b_2) \right. \\ & + \left(\frac{n-k}{p_1} + b_1 \right) \cdot t_{op} + T_{sb}(p_2, \frac{n-k}{p_1} + b_1) \\ & \left. + \left(\frac{n-k}{p_1} + b_1 \right) \left(\frac{n-k}{p_2} + b_2 \right) \cdot 2t_{op} \right) \end{aligned}$$

approximiert werden. Für die weitere Analyse müssen wir für die Broadcastoperation eine Laufzeit zugrundelegen. Dazu nehmen wir an, dass die Laufzeit einer Einzel-Broadcastoperation durch die folgende Formel beschrieben wird, die für viele Netzwerke und Parallelrechner zumindest in erster Näherung gilt:

$$T_{sb}(p, m) = \log p \cdot (\tau + m \cdot t_c),$$

wobei τ die Startupzeit für eine Nachrichtenübertragung, t_c die Transferzeit des Netzwerkes und m die Nachrichtengröße ist. Damit gilt wegen

$$\sum_{k=1}^{n-1} (n-k+1) = \sum_{k=2}^n k = \left(\sum_{k=1}^n k \right) - 1 = \frac{n(n+1)}{2} - 1$$

für die Gesamtausführungszeit der Broadcastoperationen zum Verteilen von Zeile k :

$$\begin{aligned}
 & \sum_{k=1}^{n-1} T_{sb}(p_1, \frac{n-k+1}{p_2} + b_2) \\
 &= \sum_{k=1}^{n-1} \log p_1 \left(\left(\frac{n-k+1}{p_2} + b_2 \right) t_c + \tau \right) \\
 &= \log p_1 \left(\left(\frac{n(n+1)}{2} - 1 \right) \frac{1}{p_2} t_c + (n-1)b_2 t_c + (n-1)\tau \right).
 \end{aligned}$$

Für die weitere Berechnung wird die allgemeine Beziehung

$$\sum_{k=1}^{n-1} \frac{n-k}{p_1} \cdot \frac{n-k}{p_2} = \frac{1}{p} \sum_{k=1}^{n-1} k^2 = \frac{1}{p} \frac{n(n-1)(2n-1)}{6}$$

benutzt. Insgesamt erhält man für die LR-Zerlegung die Laufzeitformel

$$\begin{aligned}
 T(n, p) &= \log p_1 \left(\left(\frac{n(n+1)}{2} - 1 \right) \frac{t_c}{p_2} + (n-1)b_2 t_c + (n-1)\tau \right) \\
 &+ \left(\frac{n(n-1)}{2} \frac{1}{p_1} + (n-1)b_1 \right) t_{op} \\
 &+ \log p_2 \left(\left(\frac{n(n-1)}{2} \frac{t_c}{p_1} \right) + (n-1)b_1 t_c + (n-1)\tau \right) \\
 &+ \left(\frac{n(n-1)(2n-1)}{6p} + \frac{n(n-1)}{2} \left(\frac{b_2}{p_1} + \frac{b_1}{p_2} \right) + (n-1)b_1 b_2 \right) 2t_{op}.
 \end{aligned}$$

Dabei gilt für die Blockgrößen $1 \leq b_i \leq n/p_i$ für $i = 1, 2$. Da in der Laufzeitformel die Blockgrößen b_1 und b_2 überall nur als multiplikativer Faktor in positiven Termen auftreten, wird der Wert von $T(n, p)$ minimal für $b_1 = b_2 = 1$. Zur Festlegung der Werte der Parameter p_1 und p_2 zerlegen wir $T(n, p)$ in drei Teile:

$$T(n, p) = T_C(n, p) + T_S(n, p) + T_A(n, p).$$

Dabei ist $T_C(n, p)$ ein konstanter Teil, der unabhängig von p_1 und p_2 ist, $T_S(n, p)$ ist ein symmetrischer Teil, der eine symmetrische Funktion in p_1 und p_2 ist, $T_A(n, p)$ ist der verbleibende asymmetrische Teil. Unter Verwendung der Blockgrößen $b_1 = b_2 = 1$ ergibt sich für die einzelnen Teile:

$$\begin{aligned}
 T_C(n, p) &= \frac{n(n-1)(2n-1)}{3p} \cdot t_{op} + (n-1) \cdot 3t_{op} \\
 &\quad + (\log p_1 + \log p_2) ((n-1)\tau + (n-1)t_c) \\
 T_S(n, p) &= \frac{n(n-1)}{2} \left(\frac{\log p_1}{p_2} + \frac{\log p_2}{p_1} \right) t_c + \frac{n(n-1)}{2} \left(\frac{1}{p_1} + \frac{1}{p_2} \right) 2t_{op}
 \end{aligned}$$

$$T_A(n, p) = \frac{n(n-1)}{2} \frac{1}{p_1} t_{op} + \frac{t_c}{p_2} (n-1) \log p_1$$

Dabei ist $T_C(n, p)$ wegen $\log p_1 + \log p_2 = \log(p_1 p_2) = \log p$ unabhängig von p_1 und p_2 und hat als Konstante keinen Einfluß auf die Belegung von p_1 und p_2 und kann im Folgenden ignoriert werden. Für die folgende Analyse betrachten wir den symmetrischen Teil $T_S(n, p)$ und vernachlässigen den asymmetrischen Teil $T_A(n, p)$. Dies ist dann gerechtfertigt, wenn t_c im Vergleich zu t_{op} groß ist, was für viele Parallelrechner der Fall ist. Wir suchen eine Belegung von p_1 und p_2 , die den Wert von $T_S(n, p)$ minimiert. Wegen $p_1 \cdot p_2 = p$ ist $p_2 = p/p_1$. Damit können wir $T_S(n, p)$ darstellen als

$$T_S(n, p) = \frac{n(n-1)}{2} \left(\frac{p_1 \log p_1}{p} + \frac{\log p - \log p_1}{p_1} \right) t_c + \frac{n(n-1)}{2} \left(\frac{1}{p_1} + \frac{p_1}{p} \right) 2t_{op}.$$

Wenn wir $T_S(n, p)$ als Funktion von p_1 auffassen, nimmt diese Funktion ein Minimum an, wenn die Ableitung der Funktion Null wird. Wegen $\log_2 p = \ln p / \ln 2$, wobei $\ln$ den natürlichen Logarithmus bezeichnet, gilt für die Ableitung

$$\begin{aligned} \frac{dT(p_1)}{dp_1} &= \frac{n(n-1)}{2} \left(\frac{1}{p \cdot \ln 2} + \frac{\log p_1}{p} - \frac{\log p}{p_1^2} + \frac{\log p_1}{p_1^2} - \frac{1}{p_1^2 \cdot \ln 2} \right) t_c \\ &\quad + \frac{n(n-1)}{2} \left(\frac{1}{p} - \frac{1}{p_1^2} \right) 2t_{op}. \end{aligned}$$

Die Gleichung $dT(p_1)/dp_1 = 0$ ist analytisch nicht lösbar. Wenn man aber p_1 als reellen Wert auffasst, gilt für $p_1 = \sqrt{p}$:

$$\begin{aligned} \frac{1}{p} - \frac{1}{p_1^2} &= \frac{1}{p} - \frac{1}{p} = 0 \\ \frac{1}{p \ln 2} - \frac{1}{p_1^2 \ln 2} &= 0 \\ \frac{\log p_1}{p} - \frac{\log p}{p_1^2} + \frac{\log p_1}{p_1^2} &= 0. \end{aligned}$$

Also nimmt $dT(p_1)/dp_1$ für $p_1 = \sqrt{p}$ den Wert 0 an. Die zweite Ableitung von $T(p_1)$ ist für $p_1 = \sqrt{p}$ größer Null, d.h. der symmetrische Teil wird für $p_1 = p_2 = \sqrt{p}$ minimal. Unter Vernachlässigung des asymmetrischen Teils bedeutet dies, dass insgesamt die Laufzeitformel einen minimalen Wert annimmt für $p_1 = p_2 = \sqrt{p}$ und $b_1 = b_2 = 1$. Das Ergebnis der Analyse besagt also, dass für die Datenverteilung eine gesamtzyklische Verteilung mit Blockgröße 1 in beiden Dimensionen und gleicher Anzahl von Prozessoren in den Zeilen und Spalten der Matrix empfohlen wird. In der Praxis müssen dabei die Werte von p_1 und p_2 so mit Integerwerten belegt werden, dass ihr Produkt p ergibt.

7.2 Direkte Verfahren für Gleichungssysteme mit Bandstruktur

Bei der numerischen Behandlung von partiellen Differentialgleichungen mit Diskretisierungsverfahren entstehen oft große lineare Gleichungssysteme mit regelmäßiger Bandstruktur. Dies sind dünnbesetzte Gleichungssysteme, in deren Koeffizientenmatrix nur die Diagonale und einige Nebendiagonalen mit Einträgen ungleich Null besetzt sind. Im ersten Teilabschnitt werden wir als Motivation zunächst die Diskretisierung der zweidimensionalen Poisson-Gleichung betrachten und das zugehörige Gleichungssystem herleiten. Danach wenden wir uns in Abschnitt 7.2.2 der Lösung von Tridiagonalsystemen zu und zeigen, wie mit Hilfe der Technik des rekursiven Verdoppelns und der zyklischen Reduktion effiziente parallele Implementierungen erzeugt werden können. Die Verfahren werden in Abschnitt 7.2.3 für allgemeine Bandsysteme verallgemeinert und in dieser Form auf das Modellproblem aus Abschnitt 7.2.1 angewendet.

7.2.1 Diskretisierung der Poisson-Gleichung

Sind große lineare Gleichungssysteme zu lösen, so resultieren diese häufig aus Diskretisierungen elliptischer partieller Differentialgleichungen mit Randbedingungen. Wir betrachten im Folgenden die **Poisson-Gleichung** als Prototyp einer solchen partiellen Differentialgleichung (engl. *partial differential equation*, PDE). Die Poisson-Gleichung wird oft als **Modellproblem** bezeichnet, da die Gleichung zwar sehr einfach ist, ihre Lösung aber prinzipiell sehr ähnlich zu der Lösung komplizierterer Differentialgleichungen verläuft und sie daher häufig als Anwendungsbeispiel für numerische Lösungsverfahren und ihre Parallelisierungen benutzt wird, vgl. etwa [52, 69, 155]. Im zweidimensionalen Fall lautet die Poisson-Gleichung

$$-\Delta u(x, y) = f(x, y) \quad \text{für alle } (x, y) \in \Omega \quad (7.10)$$

in einem Gebiet $\Omega \subset \mathbb{R}^2$. Dabei bezeichnet u die gesuchte Lösungsfunktion, f ist eine vorgegebene, auf dem abgeschlossenen Gebiet Ω stetige Funktion und Δ ist der **Laplace-Operator**, der im zweidimensionalen Fall als

$$\Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$$

definiert ist, wobei $\partial/\partial x$ und $\partial/\partial y$ die partiellen Ableitungen nach x bzw. y bezeichnen. Analog sind $\partial^2/\partial x^2$ und $\partial^2/\partial y^2$ die zweiten partiellen Ableitungen nach x bzw. y . Ausgeschrieben lautet Gleichung (7.10) damit

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = f(x, y).$$

Wir betrachten Gleichung (7.10) auf dem Einheitsquadrat $\Omega = (0, 1) \times (0, 1)$ und nehmen an, dass eine **Dirichlet**-Randbedingung gegeben ist, d.h. es soll die Randbedingung

$$u(x, y) = \varphi(x, y) \quad \text{für alle } (x, y) \in \partial\Omega \quad (7.11)$$

gelten, wobei $\partial\Omega$ den Rand von Ω bezeichnet, also in unserem Fall $\partial\Omega = \{(x, y) \mid 0 \leq x \leq 1, y = 0 \text{ oder } y = 1\} \cup \{(x, y) \mid 0 \leq y \leq 1, x = 0 \text{ oder } x = 1\}$, und φ eine vorgegebene Funktion ist. Abbildung 7.6 zeigt eine Veranschaulichung. Indem ihre Werte auf dem Rand des Definitionsbereiches vorgegeben werden, wird die gesuchte Lösungsfunktion durch die Dirichlet-Randbedingung eindeutig definiert.

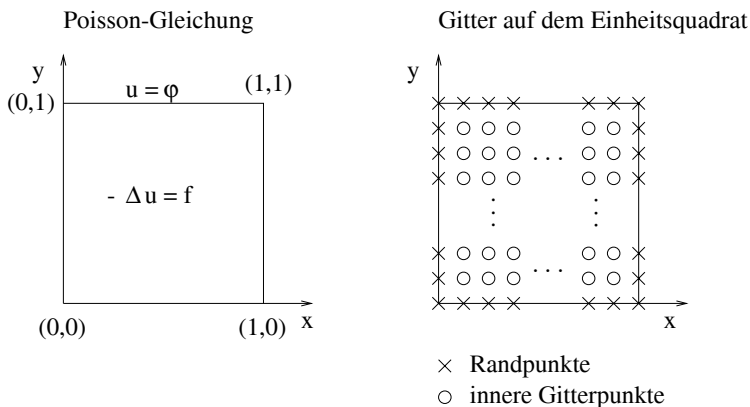


Abb. 7.6. Poisson-Gleichung mit Dirichlet-Randbedingung auf dem Einheitsquadrat $\Omega = (0, 1) \times (0, 1)$. Zur numerischen Lösung der Poisson-Gleichung wird auf das Einheitsquadrat ein äquidistantes Gitter mit Gitterpunktabstand $1/(N + 1)$ gelegt. Das Gitter hat N^2 innere Gitterpunkte. Auf den Randpunkten gilt die vorgegebene Randbedingung.

Ein Beispiel für eine Poisson-Gleichung tritt z.B. in der Elektrostatik auf. Für das von einem Ladungsfeld ϱ verursachte Potential u gilt

$$\Delta u = -\frac{\varrho}{\varepsilon_0},$$

wobei $\varrho = \varrho(x, y)$ eine ortsabhängige Ladungsdichte bezeichnet [86].

Zur numerischen Lösung von $-\Delta u(x, y) = f(x, y)$ verwenden wir die Methode der **finiten Differenzen**. Dazu wird der Definitionsbereich der gesuchten Lösungsfunktion $\Omega \cup \partial\Omega$ in x - und y -Richtung mit einem in Abbildung 7.6 illustrierten Gitter überzogen, das in beiden Richtungen aus der gleichen Anzahl $N + 2$ von Gitterpunkten besteht, die einen äquidistanten Abstand $h = \frac{1}{N+1}$ haben. Die Gitterpunkte sind

$$(x_i, y_j) = (ih, jh) \quad \text{für } i, j = 0, 1, \dots, N + 1.$$

Speziell gilt also $x_0 = y_0 = 0$ und $x_{N+1} = y_{N+1} = 1$. Die gesuchte Funktion u wird in den Gitterpunkten (x_i, y_j) bestimmt. Gesucht sind also die Werte

$u_{ij} := u(x_i, y_j)$ für $i, j = 0, 1, \dots, N+1$. Für innere Punkte des Gitters (x_i, y_j) mit $i, j = 1, \dots, N$ sind die Werte u_{ij} aus der Poisson-Gleichung (7.10) zu bestimmen, indem ein Gleichungssystem für N^2 Unbekannte aufgestellt wird. Dazu führen wir eine **Taylor-Entwicklung** um den Punkt (x_i, y_j) durch. Die Entwicklung in Richtung der x -Koordinate ergibt:

$$\begin{aligned} u(x_i + h, y_j) &= u(x_i, y_j) + h \cdot u_x(x_i, y_j) + \frac{h^2}{2} u_{xx}(x_i, y_j) \\ &\quad + \frac{h^3}{6} u_{xxx}(x_i, y_j) + O(h^4), \\ u(x_i - h, y_j) &= u(x_i, y_j) - h \cdot u_x(x_i, y_j) + \frac{h^2}{2} u_{xx}(x_i, y_j) \\ &\quad - \frac{h^3}{6} u_{xxx}(x_i, y_j) + O(h^4), \end{aligned}$$

wobei u_x die partielle Ableitung von u nach x bezeichnet, d.h. $u_x = \partial u / \partial x$ und entsprechend $u_{xx} = \partial^2 u / \partial x^2$. Durch Addition der beiden Gleichungen erhält man:

$$u(x_i + h, y_j) + u(x_i - h, y_j) = 2u(x_i, y_j) + h^2 u_{xx}(x_i, y_j) + O(h^4).$$

Eine analoge Vorgehensweise in y -Richtung führt auf die Gleichung

$$u(x_i, y_j + h) + u(x_i, y_j - h) = 2u(x_i, y_j) + h^2 u_{yy}(x_i, y_j) + O(h^4).$$

Die Auflösung der beiden Gleichungen nach $u_{xx}(x_i, y_j)$ bzw. $u_{yy}(x_i, y_j)$ führt unter Vernachlässigung der Terme höherer Ordnung auf eine Näherung des Laplace-Operators an den Gitterpunkten

$$\Delta u(x_i, y_j) = -\frac{1}{h^2} (4u_{ij} - u_{i+1,j} - u_{i-1,j} - u_{i,j+1} - u_{i,j-1}).$$

Die Näherung für $\Delta u(x_i, y_j)$ bezieht also außer dem Wert u_{ij} am Gitterpunkt (x_i, y_j) die Werte von u an den vier Nachbargitterpunkten ein, siehe Abbildung 7.7. Diese Nachbarschaftsbeziehung wird auch als **Fünfpunkt-Stern** (engl. *five-point stencil*) bezeichnet.

Durch Einsetzen in Gleichung (7.10) erhält man mit der Abkürzung $f_{ij} := f(x_i, y_j)$ für die Funktionswerte von f an den Gitterpunkten die **Fünfpunkt-Gleichung**

$$\frac{1}{h^2} (4u_{ij} - u_{i+1,j} - u_{i-1,j} - u_{i,j+1} - u_{i,j-1}) = f_{ij} \quad (7.12)$$

für $1 \leq i, j \leq N$, die die N^2 Unbekannten u_{ij} miteinander in Beziehung setzt. Für die Randpunkte gilt wegen der Randbedingung (7.11)

$$u_{ij} = \varphi(x_i, y_j) \quad (7.13)$$

für $i = 0, N+1$ und $j = 0, \dots, N+1$ bzw. $j = 0, N+1$ und $i = 0, \dots, N+1$. Für die Indizes $i = 1, i = N, j = 1$ oder $j = N$ liegen die Punkte (x_i, y_j) am Rand des Bereichs der inneren Punkte des Gitters. Die Fünfpunkt-Formel

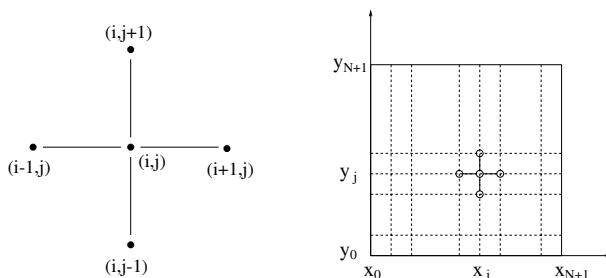


Abb. 7.7. Fünfpunkt-Stern der Diskretisierung des Laplace-Operators mit finiten Differenzen. Die Berechnung an einem Gitterpunkt bezieht Werte an den vier Nachbar-gitterpunkten in Richtung Norden, Süden, Westen und Osten ein.

greift somit auf einen bzw. (für die Eckpunkte des inneren Bereiches) zwei der vorgegebenen konstanten Randwerte zu. Für die in Gleichung (7.12) angesprochenen Nachbarpunkte von u_{ij} , die auf dem Rand liegen, werden die entsprechenden Werte von u durch die vorgegebenen Werte der Randbedingung nach Gleichung (7.13) ersetzt und in die konstanten rechten Seiten von (7.12) einbezogen. Für den Eckpunkt (x_1, y_1) des inneren Gitterbereiches ergibt sich z.B. die Gleichung

$$\frac{1}{h^2}(4u_{11} - u_{21} - u_{12}) = f_{ij} + \frac{1}{h^2}\varphi(0, y_1) + \frac{1}{h^2}\varphi(x_1, 0) .$$

Die Fünfpunkt-Gleichung (7.12) mit ihren N^2 Unbekannten und N^2 Gleichungen lässt sich als lineares Gleichungssystem mit einer Koeffizientenmatrix $A \in \mathbb{R}^{N \times N}$ schreiben. Für eine Matrixformulierung $Az = d$ des Gleichungssystems (7.12) unter Einbeziehung der Randbedingung (7.13) werden die N^2 Unbekannten u_{ij} für $i, j = 1, \dots, N$ zunächst in einem Vektor z der Länge $n = N^2$ angeordnet, und zwar in der Form

$$z = (u_{11}, u_{21}, \dots, u_{N1}, u_{12}, u_{22}, \dots, u_{N2}, \dots, u_{1N}, u_{2N}, \dots, u_{NN}) ,$$

d.h. die Gitterpunkte $(u_{ij})_{i,j=1,\dots,N}$ werden *zeilenweise* (in Richtung der x -Koordinate) angeordnet, siehe Abbildung 7.8. Die Abbildungsvorschrift ist

$$z_k := u_{ij} \text{ mit } k = i + (j - 1)N \text{ für } i, j = 1, \dots, N .$$

Die Fünfpunkt-Formel hat mit Verwendung des Vektors z die Gestalt

$$\begin{aligned} \frac{1}{h^2} (4z_{i+(j-1)N} - z_{i+1+(j-1)N} - z_{i-1+(j-1)N} - z_{i+jN} - z_{i+(j-2)N}) \\ = d_{i+(j-1)N} \end{aligned}$$

mit $d_{i+(j-1)N} = f_{ij}$ bzw.

$$\frac{1}{h^2} (4z_k - z_{k+1} - z_{k-1} - z_{k+N} - z_{k-N}) = d_k \quad (7.14)$$

für $k = 1, \dots, n$ mit $k = i + (j-1)N$ und einem Vektor d auf der rechten Seite, der f_{ij} und wie oben erläutert die konstanten Anteile des Randes umfasst. Die Besetzung dieses Vektors d und der Koeffizientenmatrix A ist in folgendem Algorithmus beschrieben [69]. Dabei sind die beiden Schleifen über i und j für den Lauf über das zweidimensionale Diskretisierungsgitter zuständig. Jedem Punkt (i, j) des Diskretisierungsgitters entspricht eine Zeile der Koeffizientenmatrix $A = (a_{ij})_{i,j=1,\dots,N^2}$. Bei jeder Ausführung des Rumpfes der Schleife werden also die Einträge *einer* Zeile der Matrix gesetzt. Sind keine Randpunkte des inneren Bereichs angesprochen, so enthält die k -te Zeile nach (7.14) fünf Einträge, die nicht Null sind, und zwar in den Spaltenpositionen $k, k+1, k-1, k+N, k-N$. Der Algorithmus behandelt auch die Sonderfälle der Randzugriffe, also die Fälle $i = 1, i = N, j = 1$ oder $j = N$.

```
/* Algorithmus zur Besetzung der Matrix A und des Vektors d */
Setze alle Elemente der Matrix A auf 0 ;
for (j = 1; j <= N; j++)
  for (i = 1; i <= N; i++) {
    /* Besetzung von d_k und Zeile k von A mit k = i + (j-1)N */
    k = i + (j-1) * N;
    a_{k,k} = 4/h^2;
    d_k = f_{ij};
    if (i > 1) a_{k,k-1} = -1/h^2 else d_k = d_k + 1/h^2 * phi(0, y_j);
    if (i < N) a_{k,k+1} = -1/h^2 else d_k = d_k + 1/h^2 * phi(1, y_j);
    if (j > 1) a_{k,k-N} = -1/h^2 else d_k = d_k + 1/h^2 * phi(x_i, 0);
    if (j < N) a_{k,k+N} = -1/h^2 else d_k = d_k + 1/h^2 * phi(x_i, 1);
  }
```

Abbildung 7.8 zeigt das Diskretisierungsgitter und die Besetzungsstruktur der zugehörigen Koeffizientenmatrix. Das lineare Gleichungssystem (7.14) hat in Matrixdarstellung die Gestalt:

$$\frac{1}{h^2} \begin{pmatrix} B & -I & & 0 \\ -I & B & & \\ & & \ddots & \ddots \\ & & \ddots & \ddots & -I \\ 0 & & & -I & B \end{pmatrix} \cdot z = d. \quad (7.15)$$

Dabei ist I die $N \times N$ -Einheitsmatrix, bei der nur die Diagonalelemente mit dem Wert 1 besetzt sind, alle anderen Elemente sind 0. B stellt die folgende Matrix dar:

$$B = \begin{pmatrix} 4 & -1 & & 0 \\ -1 & 4 & & \\ & & \ddots & \ddots \\ & & \ddots & \ddots & -1 \\ 0 & & & -1 & 4 \end{pmatrix}. \quad (7.16)$$

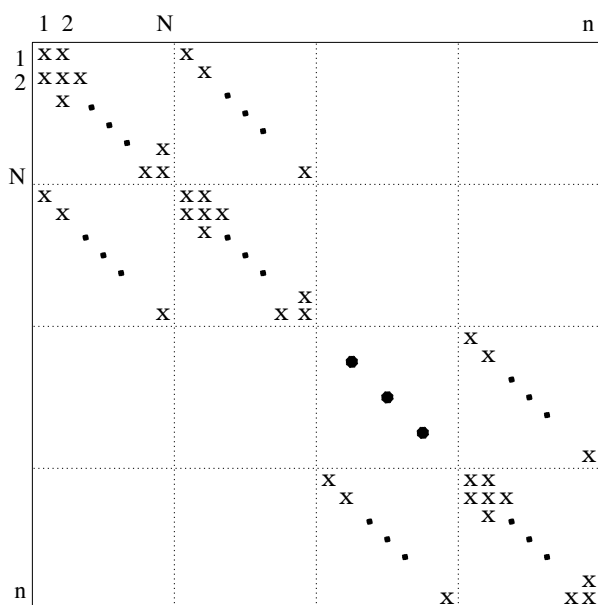
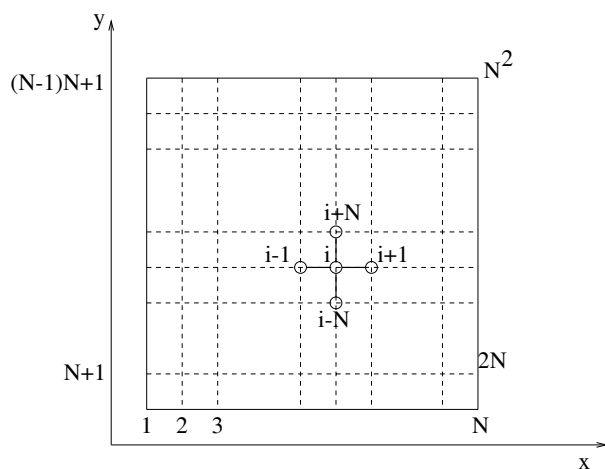


Abb. 7.8. Rechteckiges Gitter in der x - y -Ebene der Größe $N \times N$ und $n \times n$ -Matrix mit $n = N^2$ des zugehörigen linearen Gleichungssystems für die Fünfpunkt-Gleichung. Die Belegung der Matrix mit Nichtnullelementen entspricht der Adjazenzmatrix des Gitters. Umgekehrt kann das Gitter als Adjazenzgraph der Nichtnullstruktur der Matrix aufgefasst werden.

Insgesamt stellt (7.12) bzw. (7.14) somit ein Gleichungssystem dar, dessen Koeffizientenmatrix die Hauptdiagonale und die beiden Nebendiagonalen besetzt hat, aber auch zwei weitere Diagonalen, die N Positionen von der Hauptdiagonalen entfernt sind. Somit hat das Gleichungssystem eine spezielle Bandstruktur, die bei der Lösung des Gleichungssystems ausgenutzt werden kann, da die Nulleinträge bei der Lösung nicht berücksichtigt werden müssen. Wir betrachten im Folgenden Lösungsverfahren für Gleichungssysteme mit Bandstruktur und beginnen unsere Betrachtungen mit Tridiagonalsystemen, d.h. mit Gleichungssystemen, bei denen nur die Hauptdiagonale und die beiden Nebendiagonalen besetzt sind. Ein solches Tridiagonalsystem entsteht z.B. bei der Anwendung des oben beschriebenen Diskretisierungsverfahrens auf eine eindimensionale Poisson-Gleichung.

7.2.2 Lösung von Tridiagonalsystemen

Sei $A = (a_{ij})_{i,j=1,\dots,n} \in \mathbb{R}^{n \times n}$ eine Matrix und sei $r \in \mathbb{N}$ eine Zahl mit $r \leq n$. A heißt **Bandmatrix** mit halber **Bandbreite** r , falls

$$a_{ij} = 0 \text{ für } |i - j| > r .$$

Falls $r = 1$ ist, heißt A **Tridiagonalmatrix**. Wir betrachten zuerst ein lineares Gleichungssystem $Ax = y$ mit einer Tridiagonalmatrix A . Die Anwendung der Gauß-Elimination (ohne Pivotsuche) bewirkt, dass auf Stufe k der Vorwärtselimination die folgenden Schritte durchzuführen sind, vgl. Abschnitt 7.1:

1. Setze $l_{ik} := a_{ik}^{(k)} / a_{kk}^{(k)}$ für $i = k + 1, \dots, n$.
2. Subtrahiere das l_{ik} -fache der k -ten Zeile von den Zeilen $i = k + 1, \dots, n$, d.h. berechne

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - l_{ik} \cdot a_{kj}^{(k)} \quad \text{für } k \leq j \leq n \text{ und } k < i \leq n .$$

Die rechte Seite y des Gleichungssystems wird analog verändert.

Wegen der speziellen Struktur der Tridiagonalmatrix A sind alle Einträge $a_{ik} = 0$ für $i \geq k+2$. Für die Anwendung der Gauß-Elimination bedeutet dies, dass in jeder Stufe nur ein Eliminationsfaktor $l_{k+1} := l_{k+1,k}$ zu berechnen ist und dass nur eine Zeilensubtraktion durchgeführt werden muss. Für diese Zeile ist nur ein Wert neu zu berechnen. Die Werte von Matrix $A \in \mathbb{R}^{n \times n}$ seien für die weitere Formulierung wie folgt bezeichnet:

$$A = \begin{pmatrix} b_1 & c_1 & & & 0 \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & \ddots & \\ & & \ddots & \ddots & c_{n-1} \\ 0 & & & a_n & b_n \end{pmatrix} . \quad (7.17)$$

Mit $u_1 = b_1$ werden in Stufe $k + 1$ mit $k = 1, \dots, n - 1$ also folgende Berechnungen durchgeführt:

$$\begin{aligned} l_{k+1} &= a_{k+1}/u_k \\ u_{k+1} &= b_{k+1} - l_{k+1} \cdot c_k . \end{aligned} \quad (7.18)$$

Nach $n - 1$ Schritten entsteht eine LR-Zerlegung $A = LR$ von Matrix (7.17) mit

$$L = \begin{pmatrix} 1 & & & 0 \\ l_2 & 1 & & \\ & \ddots & \ddots & \\ 0 & & l_n & 1 \end{pmatrix} \quad \text{und} \quad R = \begin{pmatrix} u_1 & c_1 & & 0 \\ & \ddots & \ddots & \\ & & u_{n-1} & c_{n-1} \\ 0 & & & u_n \end{pmatrix} .$$

Die rechte Seite y des Gleichungssystems wird entsprechend der angegebenen Schritte mittransformiert, d.h. in Schritt $k + 1$ wird ein Wert

$$\tilde{y}_{k+1} = y_{k+1} - l_{k+1} \cdot \tilde{y}_k$$

berechnet. Die Lösung x von $Ax = y$ ergibt sich aus der unteren Dreiecksmatrix R durch Rückwärtseinsetzen, indem ausgehend von $x_n = \tilde{y}_n/u_n$ nacheinander aus Gleichung $u_i x_i + c_i x_{i+1} = \tilde{y}_i$ die Werte

$$x_i = \frac{\tilde{y}_i}{u_i} - \frac{c_i}{u_i} x_{i+1} \quad \text{für } i = n - 1, \dots, 1$$

errechnet werden. Insgesamt ist damit der Aufwand für die Durchführung der Berechnung der Lösung linear in der Größe n des Gleichungssystems. Der Nachteil dieser Vorgehensweise besteht darin, dass die Eliminationsphase (7.18) Datenabhängigkeiten enthält, die eine sequentielle Durchführung jedes Schrittes erfordern: l_{k+1} hängt von u_k ab, u_{k+1} hängt von l_{k+1} ab. Die Berechnung der Lösung durch Ausführung der Eliminationsphase und des Rückwärtseinsetzens muss also sequentiell erfolgen, da die vorhandenen Datenabhängigkeiten eine parallele Ausführung verhindern.

Parallele Verfahren zur direkten Lösung von linearen Gleichungssystemen mit Tridiagonalmatrix ergeben sich durch die Techniken des **rekursiven Verdoppelns** oder der **zyklischen Reduktion**. Beide Verfahren betrachten die einzelnen Gleichungen des Gleichungssystems $Ax = y$ mit einer Matrix A der Gestalt (7.17), also

$$\begin{aligned} b_1 x_1 + c_1 x_2 &= y_1 \\ a_i x_{i-1} + b_i x_i + c_i x_{i+1} &= y_i \quad \text{für } i = 2, \dots, n - 1 \\ a_n x_{n-1} + b_n x_n &= y_n \end{aligned}$$

und basieren auf dem Ansatz, die Variablen x_{i-1} und x_{i+1} aus der i -ten Gleichung durch Einsetzen von Vielfachen der Gleichungen $i - 1$ und $i + 1$ in Gleichung i zu eliminieren. Dies führt auf ein neues Gleichungssystem für die gesuchte Lösung $x = (x_1, \dots, x_n)$, dessen Koeffizientenmatrix verschobene Diagonalen aufweist. Wird dieser Trick wiederholt auf jede Gleichung

angewandt, so entsteht eine Matrix, in der nur die Diagonalelemente besetzt sind und aus der daher die Lösung x direkt abgelesen werden kann. Im Folgenden stellen wir diese Verfahren unter dem Namen *rekursives Verdoppeln* ausführlicher vor. Die *zyklische Reduktion* ist eine Variation des rekursiven Verdoppelns, in der ebenfalls in mehreren Schritten Variablen mit Hilfe der benachbarten Gleichungen eliminiert werden, dies jedoch jeweils nur auf die Hälfte der vorherigen Gleichungen angewandt wird und so weniger Berechnungen als beim rekursiven Verdoppeln durchgeführt werden. Es wird jedoch eine abschließende Substitutionsphase zur Berechnung der Lösung x benötigt.

Bevor wir beide Verfahren genauer vorstellen, möchten wir darauf hinweisen, dass die Begriffe *rekursives Verdoppeln* und *zyklische Reduktion* in der Literatur sehr unterschiedlich verwendet werden. In unserer Bezeichnungsweise folgen wir der Sicht, dass zyklische Reduktion und rekursives Verdoppeln zwei unterschiedliche Realisierungen der gleichen numerischen Idee sind, vgl. [40, 53], also des Eliminierens von Variablen durch Verwendung benachbarter Gleichungen. Diese Idee kann allgemein für **Rekurrenzgleichungen** formuliert werden. Eine Rekurrenzgleichung ist eine rekursive Berechnungsvorschrift für Elemente einer Folge, in der die Berechnung eines Elementes von anderen Elementen abhängt, die vorher oder nachher nach der *gleichen* Vorschrift berechnet werden. So bilden z.B. die beiden Gleichungen (7.18) eine Rekurrenzgleichung. In der folgenden Darstellung werden rekursives Verdoppeln und zyklische Reduktion für das Verfahren von Hockney und Golub verwendet [82, 117]. Häufig wird das Verfahren von Hockney und Golub jedoch insgesamt als *zyklische Reduktion* bezeichnet, vgl. [52, 101]. Die Bezeichnung *rekursives Verdoppeln* hingegen wird auch für das Verfahren von Stone benutzt [157]. Dieses Verfahren nutzt die Ideen des rekursiven Verdoppelns und der zyklischer Reduktion für die Rekurrenzgleichungen, die bei Anwendung der Gauß-Elimination auf lineare Gleichungssysteme mit Tridiagonalmatrix (7.17) entstehen, also Gleichungen (7.18). Darstellungen dieser Verfahren findet man in [53] oder [162] (hier unter dem Namen *full recursive doubling*). In unserer Darstellung folgen wir im Wesentlichen [53]. Eine parallele Realisierung wurde in [10] vorgestellt.

Zur Durchführung der Technik des **rekursiven Verdoppelns** betrachten wir für $i = 3, 4, \dots, n - 2$ drei aufeinanderfolgende Gleichungen $i - 1, i, i + 1$ des Gleichungssystems $Ax = y$ mit Matrix A aus (7.17).

$$\begin{array}{llll} a_{i-1}x_{i-2} & + & b_{i-1}x_{i-1} & + & c_{i-1}x_i & & = & y_{i-1} \\ & & a_i x_{i-1} & + & b_i x_i & + & c_i x_{i+1} & = & y_i \\ & & & & a_{i+1}x_i & + & b_{i+1}x_{i+1} & + & c_{i+1}x_{i+2} & = & y_{i+1} \end{array}$$

Wir benutzen Gleichung $i - 1$ zur Elimination von x_{i-1} aus der i -ten Gleichung und Gleichung $i + 1$ zur Elimination von x_{i+1} aus der i -ten Gleichung. Aus den Gleichungen $i - 1$ und $i + 1$ folgt:

$$\begin{aligned}
 x_{i-1} &= \frac{y_{i-1}}{b_{i-1}} - \frac{a_{i-1}}{b_{i-1}} x_{i-2} - \frac{c_{i-1}}{b_{i-1}} x_i, \\
 x_{i+1} &= \frac{y_{i+1}}{b_{i+1}} - \frac{a_{i+1}}{b_{i+1}} x_i - \frac{c_{i+1}}{b_{i+1}} x_{i+2}.
 \end{aligned}$$

Durch Einsetzen dieser Werte in die i -te Gleichung erhält man eine Gleichung, die die Unbekannten x_{i-2} , x_i und x_{i+2} enthält. Die Gleichung lautet

$$a_i^{(1)} x_{i-2} + b_i^{(1)} x_i + c_i^{(1)} x_{i+2} = y_i^{(1)} \quad (7.19)$$

und hat die Koeffizienten

$$\begin{aligned}
 a_i^{(1)} &= \alpha_i^{(1)} \cdot a_{i-1} \\
 b_i^{(1)} &= b_i + \alpha_i^{(1)} c_{i-1} + \beta_i^{(1)} \cdot a_{i+1} \\
 c_i^{(1)} &= \beta_i^{(1)} c_{i+1} \\
 y_i^{(1)} &= y_i + \alpha_i^{(1)} \cdot y_{i-1} + \beta_i^{(1)} y_{i+1}
 \end{aligned} \quad (7.20)$$

mit

$$\begin{aligned}
 \alpha_i^{(1)} &:= -a_i/b_{i-1}, \\
 \beta_i^{(1)} &:= -c_i/b_{i+1}.
 \end{aligned}$$

Die Fälle $i = 1, 2, n-1, n$ müssen gesondert behandelt werden. Nach Elimination der Unbekannten x_{i-1} und x_{i+1} (falls vorhanden) aus der i -ten Gleichung ergibt sich Gleichung (7.19) auch für diese Fälle, wobei folgende Koeffizientenbelegung gilt:

$$\begin{aligned}
 b_1^{(1)} &= b_1 + \beta_1^{(1)} a_2 & y_1^{(1)} &= y_1 + \beta_1^{(1)} y_2 \\
 b_n^{(1)} &= b_n + \alpha_n^{(1)} \cdot c_{n-1} & y_n^{(1)} &= b_n + \alpha_n^{(1)} \cdot y_{n-1} \\
 a_1^{(1)} &= a_2^{(1)} = 0 \text{ und } c_{n-1}^{(1)} = c_n^{(1)} = 0.
 \end{aligned}$$

Die Werte $a_{n-1}^{(1)}$, $a_n^{(1)}$, $b_2^{(1)}$, $b_{n-1}^{(1)}$, $c_1^{(1)}$, $c_2^{(1)}$, $y_2^{(1)}$ und $y_{n-1}^{(1)}$ erfüllen die Gleichungen (7.20) mit entsprechender Besetzung von i . Gleichung (7.19) stellt ein lineares Gleichungssystem $A^{(1)}x = y^{(1)}$ mit der Koeffizientenmatrix

$$A^{(1)} = \begin{pmatrix} b_1^{(1)} & 0 & c_1^{(1)} & & 0 \\ 0 & b_2^{(1)} & 0 & c_2^{(1)} & \\ a_3^{(1)} & 0 & b_3^{(1)} & \ddots & \ddots \\ & a_4^{(1)} & \ddots & \ddots & \ddots & c_{n-2}^{(1)} \\ & & \ddots & \ddots & \ddots & 0 \\ 0 & & & a_n^{(1)} & 0 & b_n^{(1)} \end{pmatrix}$$

dar. Im Vergleich zur ursprünglichen Koeffizientenmatrix sind die Seitendiagonalen um eine Position nach außen geschoben. Bei einer erneuten Anwendung der beschriebenen Vorgehensweise auf $A^{(1)}x = y^{(1)}$ betrachtet man für

$i = 5, 6, \dots, n-4$ die Gleichungen $i-2, i, i+2$ und benutzt Gleichung $i-2$ zur Elimination von x_{i-2} aus Gleichung i und Gleichung $i+2$ zur Elimination von x_{i+2} aus Gleichung i . Damit erhält man eine neue Gleichung i , die die Unbekannten x_{i-4}, x_i und x_{i+4} enthält, und zwar:

$$a_i^{(2)}x_{i-4} + b_i^{(2)}x_i + c_i^{(2)}x_{i+4} = y_i^{(2)}.$$

Diese Gleichung stellt wieder ein lineares Gleichungssystem $A^{(2)}x = y^{(2)}$ dar, in dessen Koeffizientenmatrix die Seitendiagonalen im Vergleich zur ursprünglichen Koeffizientenmatrix um drei Positionen nach außen geschoben sind, d.h. $A^{(2)}$ hat das Aussehen:

$$A^{(2)} = \begin{pmatrix} b_1^{(2)} & 0 & 0 & 0 & c_1^{(2)} & & 0 \\ 0 & b_2^{(2)} & & & c_2^{(2)} & & \\ 0 & & \ddots & & & \ddots & \\ 0 & & & \ddots & & & c_{n-4}^{(2)} \\ a_5^{(2)} & & & & \ddots & & 0 \\ & a_6^{(2)} & & & & \ddots & 0 \\ & & \ddots & & & & \ddots & 0 \\ 0 & & & a_n^{(2)} & 0 & 0 & 0 & b_n^{(2)} \end{pmatrix}.$$

Wird das Verfahren entsprechend fortgesetzt, so bewirkt der k -te Schritt der Methode, dass die Seitendiagonalen im Vergleich zur ursprünglichen Koeffizientenmatrix um $2^k - 1$ Positionen weiter von der Hauptdiagonalen weggeschoben sind, d.h. es sind nur die beiden 2^k -ten Nebendiagonalen besetzt. Um dies zu erreichen werden im k -ten Schritt der Methode mit $k = 1, \dots, \lceil \log n \rceil$ die Gleichungen $i - 2^{k-1}, i, i + 2^{k-1}$ betrachtet, also:

$$\begin{aligned} a_{i-2^{k-1}}^{(k-1)}x_{i-2^k} + b_{i-2^{k-1}}^{(k-1)}x_{i-2^{k-1}} + c_{i-2^{k-1}}^{(k-1)}x_i &= y_{i-2^{k-1}}^{(k-1)} \\ a_i^{(k-1)}x_{i-2^{k-1}} + b_i^{(k-1)}x_i + c_i^{(k-1)}x_{i+2^{k-1}} &= y_i^{(k-1)} \\ a_{i+2^{k-1}}^{(k-1)}x_i + b_{i+2^{k-1}}^{(k-1)}x_{i+2^{k-1}} + c_{i+2^{k-1}}^{(k-1)}x_{i+2^k} &= y_{i+2^{k-1}}^{(k-1)} \end{aligned}$$

Gleichung $i - 2^{k-1}$ wird dazu verwendet, aus Gleichung i die Unbekannte $x_{i-2^{k-1}}$ zu eliminieren. Analog wird Gleichung $i + 2^{k-1}$ dazu verwendet, aus Gleichung i die Unbekannte $x_{i+2^{k-1}}$ zu eliminieren. Die Durchführung der Elimination wird durch die Berechnung der Koeffizienten

$$\begin{aligned} a_i^{(k)} &= \alpha_i^{(k)} \cdot a_{i-2^{k-1}}^{(k-1)} \quad \text{für } i = 2^k + 1, \dots, n, \text{ und } a_i^{(k)} = 0 \text{ sonst,} \\ c_i^{(k)} &= \beta_i^{(k)} \cdot c_{i+2^{k-1}}^{(k-1)} \quad \text{für } i = 1, \dots, n - 2^k, \text{ und } c_i^{(k)} = 0 \text{ sonst,} \\ b_i^{(k)} &= \alpha_i^{(k)} \cdot c_{i-2^{k-1}}^{(k-1)} + b_i^{(k-1)} + \beta_i^{(k)} \cdot a_{i+2^{k-1}}^{(k-1)} \quad \text{für } i = 1, \dots, n, \\ y_i^{(k)} &= \alpha_i^{(k)} \cdot y_{i-2^{k-1}}^{(k-1)} + y_i^{(k-1)} + \beta_i^{(k)} \cdot y_{i+2^{k-1}}^{(k-1)} \quad \text{für } i = 1, \dots, n, \end{aligned} \quad (7.21)$$

realisiert, wobei

$$\begin{aligned}\alpha_i^{(k)} &:= -a_i^{(k-1)} / b_{i-2^{k-1}}^{(k-1)} & \text{für } i = 2^{k-1} + 1, \dots, n, \\ \beta_i^{(k)} &:= -c_i^{(k-1)} / b_{i+2^{k-1}}^{(k-1)} & \text{für } i = 1, \dots, n - 2^{k-1},\end{aligned}\quad (7.22)$$

gesetzt wird. Die modifizierte Gleichung i erhält man durch Multiplikation von Gleichung $i - 2^{k-1}$ aus Schritt $k-1$ mit $\alpha_i^{(k)}$, Multiplikation von Gleichung $i + 2^{k-1}$ aus Schritt $k-1$ mit $\beta_i^{(k)}$ und Addition der resultierenden beiden Gleichungen zu Gleichung i . Für die resultierende modifizierte Gleichung i erhält man so die Gleichung

$$a_i^{(k)} x_{i-2^k} + b_i^{(k)} x_i + c_i^{(k)} x_{i+2^k} = y_i^{(k)} \quad (7.23)$$

mit den obigen Werten für die Koeffizienten aus (7.21). Die für den Fall $k = 1$ angegebenen Koeffizienten aus (7.20) stimmen mit der gerade beschriebenen allgemeinen Form überein. Als Vorbesetzungen für den gesamten Algorithmus des rekursiven Verdoppelns werden die folgenden Werte für $k = 0$ verwendet:

$$\begin{aligned}a_i^{(0)} &= a_i & \text{für } i = 2, \dots, n, \\ b_i^{(0)} &= b_i & \text{für } i = 1, \dots, n, \\ c_i^{(0)} &= c_i & \text{für } i = 1, \dots, n-1, \\ y_i^{(0)} &= y_i & \text{für } i = 1, \dots, n.\end{aligned}$$

Weiter nehmen wir an, dass für $k = 0, \dots, \lceil \log n \rceil$ und $i \in \mathbb{Z} \setminus \{1, \dots, n\}$

$$\begin{aligned}a_i^{(k)} = c_i^{(k)} = y_i^{(k)} &= 0, \\ b_i^{(k)} &= 1, \\ x_i &= 0,\end{aligned}$$

gilt. Weiterhin sei $a_1^{(0)} = 0$ und $c_n^{(0)} = 0$. Die Anwendung des gerade beschriebenen Verfahrens liefert nach $N = \lceil \log n \rceil$ Schritten das System

$$A^{(N)} x = y^{(N)},$$

wobei in $A^{(N)}$ nur die Hauptdiagonale besetzt ist, d.h.

$$A^{(N)} = \text{diag}(b_1^{(N)}, \dots, b_n^{(N)}).$$

Aus der Matrix $A^{(N)}$ können die Komponenten des Lösungsvektors x also direkt bestimmt werden. Es ist

$$x_i = y_i^{(N)} / b_i^{(N)} \quad \text{für } i = 1, 2, \dots, n.$$

Damit kann das Verfahren des rekursiven Verdoppelns wie folgt zusammengefasst werden:

1. Berechnung der Zahlen $a_i^{(k)}$, $b_i^{(k)}$, $c_i^{(k)}$ und $y_i^{(k)}$ für $k = 1, \dots, \lceil \log n \rceil$ und $i = 1, \dots, n$.
2. Berechnung von $x_i = y_i^{(N)} / b_i^{(N)}$ für $i = 1, \dots, n$ mit $N = \lceil \log n \rceil$.

Für die Bestimmung von $A^{(N)}$ sind $N = \lceil \log n \rceil$ Schritte erforderlich, wobei jeder Schritt die Neuberechnung von $O(n)$ Werten erfordert. Die Berechnung des Lösungsvektors aus $A^{(N)}$ erfordert $O(n)$ Operationen. Im sequentiellen Fall ist die asymptotische Laufzeit des Verfahrens also $O(n \cdot \log n)$ im Vergleich zur asymptotischen Laufzeit $O(n)$ des Originalverfahrens, das auf dem Gaußschen Eliminationsverfahren beruht. Der Vorteil des rekursiven Verdoppelns liegt darin, dass die Berechnung der Werte eines Schrittes unabhängig voneinander sind und daher parallel zueinander ausgeführt werden können. Das gleiche gilt für die Berechnung der Komponenten des Lösungsvektors. Abbildung 7.9 veranschaulicht die Abhängigkeiten zwischen den in den verschiedenen Stufen errechneten Werten.

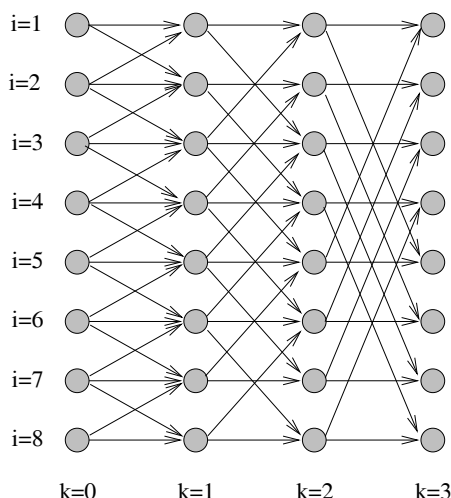


Abb. 7.9. Veranschaulichung der Abhängigkeiten zwischen benachbarten Berechnungsstufen des rekursiven Verdoppelns für 3 Berechnungsstufen und 8 Gleichungen in Form eines Abhängigkeitsgraphen. Die Werte einer Berechnungsstufe sind als Spalte dargestellt. Spalte k enthält einen Knoten für jede Gleichung i , der alle Koeffizienten von Gleichung i nach Schritt k repräsentiert. Die Spalte $k = 0$ steht für die Einträge der ursprünglichen Matrix. Eine Kante von einem Knoten i in Stufe k zu einem Knoten j in Stufe $k + 1$ bedeutet, dass zur Berechnung einer der Koeffizienten von Gleichung j in Stufe $k + 1$ mindestens ein Koeffizient der Gleichung i aus Stufe k verwendet wird.

Das Verfahren des rekursiven Verdoppelns bietet zwar einen hohen Grad an Parallelität, erkaufte sich diesen aber durch eine große *rechnerische Redundanz*. Wir betrachten daher im Folgenden eine Weiterentwicklung des Verfahrens, das mit einer linearen Laufzeit auskommt und unter dem Namen **zyklische Reduktion** bekannt ist. Der wesentliche Unterschied zum rekursiven Verdoppeln besteht darin, dass *nicht* alle Zahlen $a_i^{(k)}$, $b_i^{(k)}$, $c_i^{(k)}$ und $y_i^{(k)}$ berechnet werden, sondern dass die Anzahl der berechneten Zahlen in

jedem Schritt halbiert wird, siehe auch Abbildung 7.10. Das Verfahren der zyklischen Reduktion umfasst die Schritte:

1. Für $k = 1, \dots, \lfloor \log n \rfloor$ Berechnung der Zahlen $a_i^{(k)}$, $b_i^{(k)}$, $c_i^{(k)}$ und $y_i^{(k)}$ für $i = 2^k, \dots, n$ mit Schrittweite 2^k . Damit verringert sich die Anzahl der zur Verfügung stehenden Gleichungen (7.23) in jedem Schritt um die Hälfte. Nach Schritt $k = \lfloor \log n \rfloor$ bleibt nur noch eine Gleichung übrig, nämlich die für $i = 2^N$ mit $N = \lfloor \log n \rfloor$.
2. Für $k = \lfloor \log n \rfloor, \dots, 0$ Berechnung der Komponenten x_i aus Gleichung (7.23) für $i = 2^k, \dots, n$ mit Schrittweite 2^{k+1} durch:

$$x_i = \frac{y_i^{(k)} - a_i^{(k)} \cdot x_{i-2^k} - c_i^{(k)} x_{i+2^k}}{b_i^{(k)}}. \quad (7.24)$$

Durch dieses Vorgehen werden insgesamt

$$\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \dots + \frac{n}{2^N} = n \cdot \sum_{i=1}^{\lfloor \log n \rfloor} \frac{1}{2^i} \leq n$$

Knoten berechnet, wobei $N = \lfloor \log n \rfloor$ ist. Die sequentielle Laufzeit des Verfahrens ist somit linear in der Anzahl der Gleichungen.

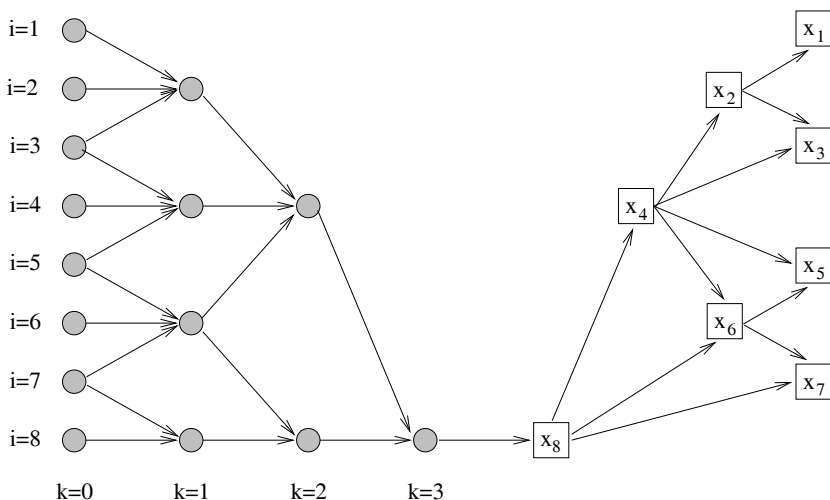


Abb. 7.10. Veranschaulichung der Abhängigkeiten zwischen benachbarten Berechnungsstufen der zyklischen Reduktion für 3 Berechnungsstufen und 8 Gleichungen in Form eines Abhängigkeitsgraphen analog zu Abbildung 7.9. Die ersten 4 Spalten stellen die Berechnungen der Koeffizienten dar, die zusätzlichen 4 Spalten stellen die Berechnungen der Elemente des Lösungsvektors x dar, die vom zweiten Schritt des Verfahrens der zyklischen Reduktion nach Gleichung (7.24) durchgeführt werden.

Da die Berechnung der Zahlen $\alpha_i^{(k)}$, $\beta_i^{(k)}$ und x_i Divisionen durch Zahlen $b_i^{(k)}$ beinhalten, sind die Verfahren des rekursiven Verdoppeln und der zyklischen Reduktion nur durchführbar, wenn *alle* Zahlen $b_i^{(k)}$ ungleich Null sind. Diese Bedingung ist auch dann nicht immer erfüllt, wenn die Gauß-Elimination ohne Pivotsuche durchführbar ist. Für eine Durchführbarkeit des rekursiven Verdoppeln müssen weitere Bedingungen an die Matrix A gestellt werden. Eine solche ist z.B., dass A symmetrisch und positiv definit ist oder dass A diagonal dominant ist, siehe etwa [53] oder [101] (hier unter dem Namen odd-even reduction). (Eine Matrix A ist symmetrisch, falls $A = A^T$ und positiv definit, falls $x^T A x > 0$ für alle x . Eine Matrix A ist diagonal dominant, falls die Absolutwerte der Diagonalelemente jeweils die Summe der Absolutwerte der entsprechenden Zeile ohne das Diagonalelement übersteigen.)

Parallele Realisierung der zyklischen Reduktion. Wir betrachten im Folgenden die parallele Implementierung des Verfahrens der zyklischen Reduktion auf p Prozessoren. Zur Vereinfachung nehmen wir an, dass die Anzahl der Gleichungen n ein Vielfaches von p ist, d.h. $n = p \cdot q$ für ein $q \in \mathbb{N}$. Die nichtleeren Diagonalen der Matrix A werden einzeln blockweise abgespeichert, so dass eine zeilenweise Abspeicherung von A entsteht. Wegen $n = p \cdot q$ speichert jeder Prozessor einen Zeilenblock der Größe q . Prozessor P_i erhält dabei die Zeilen $(i-1)q+1, \dots, i \cdot q$ von A für $1 \leq i \leq p$. Wir nehmen zur weiteren Vereinfachung an, dass q eine Zweierpotenz ist, d.h. $q = 2^Q$ für ein $Q \in \mathbb{N}$. Die parallele Implementierung führt die folgenden drei Phasen durch:

1. Phase: (Parallele Reduktionsphase der zyklischen Reduktion in $\log q$ Schritten.) Jeder Prozessor berechnet die ersten $Q = \log q$ Stufen der zyklischen Reduktion, d.h. Prozessor P_i berechnet für $k = 1, \dots, Q$ die Zahlen $a_j^{(k)}, b_j^{(k)}, c_j^{(k)}, y_j^{(k)}$ für $j = (i-1) \cdot q + 2^k, \dots, i \cdot q$ mit Schrittweite 2^k . Nach jeder Stufe muss Prozessor P_i Zahlen von Prozessor P_{i-1} (falls $i > 1$) und Prozessor P_{i+1} (falls $i < p$) empfangen, die diese in der vorausgehenden Stufe berechnet haben. Da jeder Prozessor Blöcke der Größe q besitzt, braucht wegen der Wahl von Q kein Datenaustausch mit anderen Prozessoren stattzufinden. Da jeweils 4 Koeffizienten ($a_j^{(k)}, b_j^{(k)}, c_j^{(k)}$ und $y_j^{(k)}$) berechnet werden, ist die Anzahl der mit den Nachbarprozessoren ausgetauschten Daten ein Vielfaches von 4. Bei der zyklischen Reduktion muss in jeder Stufe nur ein Datenblock von jedem Nachbarprozessor empfangen werden. Insgesamt empfängt und sendet jeder Prozessor also maximal $2Q$ Nachrichten mit jeweils 4 Elementen.

2. Phase: (Paralleles rekursives Verdoppeln für ein Tridiagonalsystem der Größe p .) Nach der ersten Phase liegen Prozessor P_i die Koeffizienten der i -ten Gleichung des folgenden p -dimensionalen Gleichungssystems mit Triagonalgestalt vor:

$$\tilde{a}_i \tilde{x}_{i-1} + \tilde{b}_i \tilde{x}_i + \tilde{c}_i \tilde{x}_{i+1} = \tilde{y}_i \quad \text{für } i = 1, \dots, p$$

mit

$$\left. \begin{aligned} \tilde{a}_i &= a_{i \cdot q}^{(Q)} \\ \tilde{b}_i &= b_{i \cdot q}^{(Q)} \\ \tilde{c}_i &= c_{i \cdot q}^{(Q)} \\ \tilde{y}_i &= y_{i \cdot q}^{(Q)} \\ \tilde{x}_i &= x_{i \cdot q} \end{aligned} \right\} \quad \text{für } i = 1, \dots, p.$$

Wir lösen dieses p -dimensionale Gleichungssystem mit *rekursivem Verdoppeln*, wobei jeder Prozessor eine Zeile bearbeitet. Prozessor P_i berechnet in jedem der $\lceil \log p \rceil$ Schritte $k = 1, \dots, \lceil \log p \rceil$ die vier Koeffizienten

$$\tilde{a}_i^{(k)}, \tilde{b}_i^{(k)}, \tilde{c}_i^{(k)}, \tilde{y}_i^{(k)}.$$

Zu deren Berechnung benötigt Prozessor P_i die Koeffizienten

$$\tilde{a}_j^{(k-1)}, \tilde{b}_j^{(k-1)}, \tilde{c}_j^{(k-1)}, \tilde{y}_j^{(k-1)}$$

der vorangegangenen Stufe $k-1$ für $j \in \{i-2^{k-1}, i+2^{k-1}\}$, vgl. Berechnungsschema (7.21). Diese Koeffizienten sind von den Prozessoren $P_{i-2^{k-1}}$ und $P_{i+2^{k-1}}$ berechnet worden. Somit erfordert jede der $\lceil \log p \rceil$ Stufen eine Kommunikation mit zwei Prozessoren, wobei in jeder Kommunikationsoperation 4 Zahlen ausgetauscht werden. Nach $N' = \lceil \log p \rceil$ Schritten berechnet Prozessor i den Wert

$$\tilde{x}_i = \tilde{y}_i^{(N')} / \tilde{b}_i^{(N')}.$$

3. Phase: (Parallele Substitutionsphase der zyklischen Reduktion.) Nach Berechnung von $\tilde{x}_i = x_{i \cdot q}$ in der zweiten Phase berechnet jeder Prozessor P_i , $i = 1, \dots, p$, in dieser Phase die Werte x_j mit $j = (i-1)q+1, \dots, iq-1$. Diese Berechnung wird entsprechend der Substitution der zyklischen Reduktion in mehreren Schritten nach Gleichung (7.24) durchgeführt. Dazu führen alle Prozessoren die Berechnungsschritte $k = Q-1, \dots, 0$ aus und berechnen in Schritt k die Elemente x_j , $j = 2^k, \dots, n$, mit Schrittweite 2^{k+1} . Davon berechnet jeder einzelne Prozessor P_i diejenigen Elemente x_j mit $j \bmod q + 1 = i$. Für die Berechnung benötigt Prozessor P_i die Werte $\tilde{x}_{i-1} = x_{(i-1)q}$ und $\tilde{x}_{i+1} = x_{(i+1)q}$ von seinen Nachbarprozessoren P_{i-1} und P_{i+1} (falls vorhanden). Abbildung 7.11 zeigt den Fall $p = 2$ und $n = 8$.

Laufzeit der parallelen zyklischen Reduktion

1. Phase: In der ersten Phase werden

$$Q = \log q = \log \frac{n}{p} = \log n - \log p$$

Stufen ausgeführt. In Stufe k mit $1 \leq k \leq Q$ berechnet jeder Prozessor maximal $4q/2^k$ Zahlen, da jeder Prozessor mit q Zeilen startet, jeweils 4 Zahlen berechnet und in jeder Stufe die Anzahl der berechneten Koeffizienten halbiert wird. Der Berechnungsaufwand für den ersten Schritt beträgt also

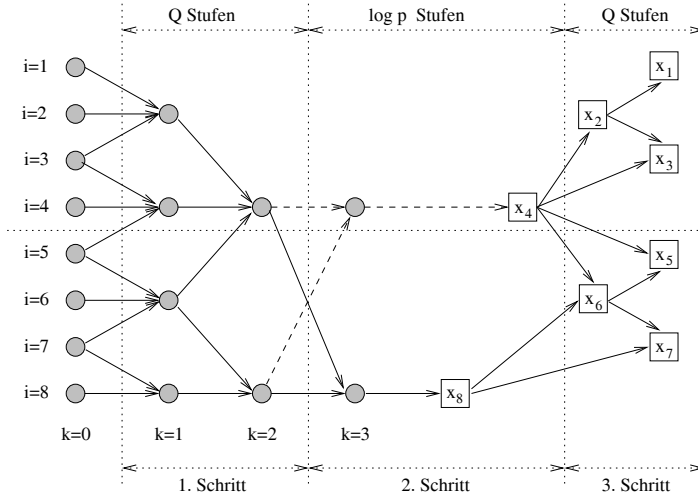


Abb. 7.11. Veranschaulichung der parallelen Implementierung der zyklischen Reduktion mit $n = 8$ Gleichungen für $p = 2$ Prozessoren. Jeder Prozessor erhält $q = 4$ Gleichungen und es ist $Q = 2$. Die erste und die dritte Phase bestehen jeweils aus $\log q = 2$ Stufen. Die zweite Phase besteht aus einer Stufe, da $\log p = 1$. Man beachte, dass im Vergleich zu Abbildung 7.10 durch die Verwendung des rekursiven Verdoppelns in der zweiten Phase mehr Knoten berechnet werden.

$$T_1(n, p) = 4t_{op} \cdot \sum_{k=1}^Q \frac{4q}{2^k} \leq 16 \frac{n}{p} \cdot t_{op}.$$

Darüberhinaus werden in der ersten Phase von jedem Prozessor $2Q$ Nachrichten mit je 4 Elementen durch parallel ausgeführte Einzel-Transfers gesendet und empfangen. Daher erfordert der erste Schritt die Kommunikationszeit

$$C_1(n, p) = 2Q \cdot t_{s2s}(4) = 2 \cdot \log \frac{n}{p} \cdot t_{s2s}(4).$$

Dabei haben wir angenommen, dass die zwischen verschiedenen Nachbarprozessoren ablaufenden Einzel-Transferoperationen gleichzeitig ausgeführt werden können.

2. Phase: In der zweiten Phase werden $\lceil \log p \rceil$ Berechnungsstufen ausgeführt. In jeder Stufe ist jeder Prozessor für die Berechnung einer Zeile verantwortlich und berechnet folglich vier Koeffizienten. Für jeden Koeffizienten werden vier Operationen benötigt. Damit gilt für den Berechnungsaufwand

$$T_2(n, p) = 16 \lceil \log p \rceil \cdot t_{op} + t_{op},$$

wobei eine weitere Operation für die Berechnung von $\tilde{x}_i$ durch Prozessor P_i berücksichtigt ist. In jeder Stufe erhält jeder Prozessor Daten von zwei anderen Prozessoren, so dass für die Kommunikationszeit der zweiten Phase gilt:

$$C_2(n, p) = 2 \lceil \log p \rceil \cdot t_{s2s}(4) .$$

3. Phase: In jeder Stufe k der dritten Phase mit $k = 0, \dots, Q - 1$ berechnet jeder Prozessor 2^k Komponenten des Lösungsvektors nach Gleichung (7.24). Für jede Komponente sind dabei nach (7.24) 5 Operationen nötig. Insgesamt berechnet jeder Prozessor also $\sum_{k=0}^{Q-1} 2^k = 2^Q - 1 = q - 1$ Komponenten. Dabei ist zu beachten, dass eine Komponente bereits im zweiten Schritt berechnet wurde. Für den Berechnungsaufwand gilt damit:

$$T_3(n, p) = 5 \cdot (q - 1) \cdot t_{op} = 5 \cdot \left(\frac{n}{p} - 1 \right) \cdot t_{op} .$$

In der dritten Phase empfängt jeder Prozessor nur je einen Wert von jedem seiner beiden Nachbarprozessoren. Für die Kommunikationszeit der dritten Phase ergibt sich also

$$C_3(n, p) = 2 \cdot t_{s2s}(1) .$$

Damit ergibt sich folgender Gesamtberechnungsaufwand:

$$\begin{aligned} T(n, p) &= \left(16 \frac{n}{p} + 16 \cdot \lceil \log p \rceil + 5 \frac{n}{p} - 4 \right) \cdot t_{op} \\ &\simeq \left(21 \frac{n}{p} + 16 \cdot \log p \right) \cdot t_{op} . \end{aligned}$$

Für den Kommunikationsaufwand ergibt sich wegen $\log(n/p) = \log n - \log p$ folgender Gesamtaufwand

$$\begin{aligned} C(n, p) &= \left\lceil 2 \cdot \log \frac{n}{p} + 2 \lceil \log p \rceil \right\rceil t_{s2s}(4) + 2 \cdot t_{s2s}(1) \\ &\simeq 2 \cdot \log n \cdot t_{s2s}(4) + 2 \cdot t_{s2s}(1) . \end{aligned}$$

Damit kann man zusammenfassend feststellen, dass sich die Berechnungszeit bis auf einen kleinen zusätzlichen Term von $16 \cdot \log p$ ideal auf die Prozessoren verteilt. Die Kommunikationszeit steigt logarithmisch in der Anzahl der Zeilen der zu bearbeitenden Matrix, während die Berechnungszeit linear ansteigt. Für große n wird die Berechnungszeit also die Kommunikationszeit überwiegen. Ab welcher Größe dies der Fall sein wird, hängt von den Kosten eines Einzeltransfers auf dem betrachteten Parallelrechner ab.

7.2.3 Verallgemeinerung auf beliebige Bandmatrizen

Das Ziel dieses Abschnitts besteht darin, die im letzten Abschnitt besprochene Methode der zyklischen Reduktion auf Gleichungssysteme $Ax = y$ zu verallgemeinern, deren Koeffizientenmatrix $A \in \mathbb{R}^{n \times n}$ eine Bandmatrix mit halber Bandbreite $r > 1$ ist, vgl. [53]. Wir nehmen an, dass n ein Vielfaches der Bandbreite r ist, d.h. $n = s \cdot r$. Im Normalfall ist r klein, d.h. $r \ll n$. Für $i = 1, \dots, s$ führen wir die Indexbereiche

$$I_i = \{j \in \mathbb{N} \mid (i-1)r < j \leq ir\}$$

ein, mit deren Hilfe wir Teilmatrizen $A_i^{(0)}, B_i^{(0)}, C_i^{(0)} \in \mathbb{R}^{r \times r}$ von A mit

$$\begin{aligned} A_i^{(0)} &= (a_{lm})_{l \in I_i, m \in I_{i-1}} \quad \text{für } i = 2, \dots, s, \\ B_i^{(0)} &= (a_{lm})_{l \in I_i, m \in I_i} \quad \text{für } i = 1, \dots, s, \\ C_i^{(0)} &= (a_{lm})_{l \in I_i, m \in I_{i+1}} \quad \text{für } i = 1, \dots, s-1, \end{aligned}$$

und Vektoren $X_i, Y_i^{(0)} \in \mathbb{R}^r$ mit

$$X_i = (x_l)_{l \in I_i} \quad \text{und} \quad Y_i^{(0)} = (y_l)_{l \in I_i} \quad \text{für } i = 1, \dots, s$$

definieren. Damit lässt sich $Ax = y$ als **Block-Tridiagonalsystem** schreiben:

$$\begin{pmatrix} B_1^{(0)} & C_1^{(0)} & & & 0 \\ A_2^{(0)} & B_2^{(0)} & C_2^{(0)} & & \\ & \ddots & \ddots & \ddots & \\ & & A_{s-1}^{(0)} & B_{s-1}^{(0)} & C_{s-1}^{(0)} \\ 0 & & & A_s^{(0)} & B_s^{(0)} \end{pmatrix} \begin{pmatrix} X_1 \\ X_2 \\ \vdots \\ X_{s-1} \\ X_s \end{pmatrix} = \begin{pmatrix} Y_1^{(0)} \\ Y_2^{(0)} \\ \vdots \\ Y_{s-1}^{(0)} \\ Y_s^{(0)} \end{pmatrix}$$

Da A eine Bandmatrix mit Bandbreite r ist, haben die Matrizen $A_i^{(0)}$ obere und die Matrizen $C_i^{(0)}$ untere Dreiecksgestalt. Die Methode der zyklischen Reduktion wird auf dieses Block-Tridiagonalsystem übertragen, indem die Eliminationsschritte aus Abschnitt 7.2.2 auf Matrixgleichungen angewendet werden. Im ersten Schritt betrachten wir drei aufeinanderfolgende Matrixgleichungen $i-1, i, i+1$ für $i = 3, 4, \dots, s-2$

$$\begin{aligned} A_{i-1}^{(0)} X_{i-2} + B_{i-1}^{(0)} X_{i-1} + C_{i-1}^{(0)} X_i &= Y_{i-1}^{(0)} \\ A_i^{(0)} X_{i-1} + B_i^{(0)} X_i + C_i^{(0)} X_{i+1} &= Y_i^{(0)} \\ A_{i+1}^{(0)} X_i + B_{i+1}^{(0)} X_{i+1} + C_{i+1}^{(0)} X_{i+2} &= Y_{i+1}^{(0)} \end{aligned}$$

und benutzen die $(i-1)$ te Gleichung zur Elimination von Teilvektor X_{i-1} aus Gleichung i und die $(i+1)$ te Gleichung zur Elimination von Teilvektor X_{i+1} aus Gleichung i . Für die Durchführung des k -ten Schrittes dieses Verfahrens setzen wir

$$A_1^{(0)} := 0 \in \mathbb{R}^{r \times r}, \quad C_s^{(0)} := 0 \in \mathbb{R}^{r \times r}$$

und für $k = 0, \dots, \lceil \log s \rceil$ und $i \in \mathbb{Z} \setminus \{1, \dots, s\}$

$$\begin{aligned} A_i^{(k)} &= C_i^{(k)} := 0 \in \mathbb{R}^{r \times r}, \\ B_i^{(k)} &:= I \in \mathbb{R}^{r \times r}, \\ Y_i^{(k)} &:= 0 \in \mathbb{R}^r. \end{aligned}$$

Analog zur Vorgehensweise im letzten Abschnitt werden für $k = 1, \dots, \lceil \log s \rceil$ mit Hilfe der Matrizen

$$\alpha_i^{(k)} := -A_i^{(k-1)} \left(B_{i-2^{k-1}}^{(k-1)} \right)^{-1},$$

$$\beta_i^{(k)} := -C_i^{(k-1)} \left(B_{i+2^{k-1}}^{(k-1)} \right)^{-1},$$

die Matrizen

$$\begin{aligned} A_i^{(k)} &= \alpha_i^{(k)} \cdot A_{i-2^{k-1}}^{(k-1)} \\ C_i^{(k)} &= \beta_i^{(k)} \cdot C_{i+2^{k-1}}^{(k-1)} \\ B_i^{(k)} &= \alpha_i^{(k)} C_{i-2^{k-1}}^{(k-1)} + B_i^{(k-1)} + \beta_i^{(k)} A_{i+2^{k-1}}^{(k-1)} \end{aligned} \quad (7.25)$$

und der Vektor

$$Y_i^{(k)} = \alpha_i^{(k)} Y_{i-2^{k-1}}^{(k-1)} + Y_i^{(k-1)} + \beta_i^{(k)} Y_{i+2^{k-1}}^{(k-1)} \quad (7.26)$$

berechnet. Für $k = 0, \dots, \lceil \log s \rceil$ gelten somit die Gleichungen

$$A_i^{(k)} X_{i-2^k} + B_i^{(k)} X_i + C_i^{(k)} X_{i+2^k} = Y_i^{(k)} \quad (7.27)$$

für $i = 1, \dots, s$. Damit kann das Verfahren der zyklischen Reduktion für allgemeine Bandsysteme wie folgt durchgeführt werden:

1. Für $k = 1, \dots, \lceil \log s \rceil$ Berechnung der Matrizen $A_i^{(k)}, B_i^{(k)}, C_i^{(k)}$ und der Vektoren $Y_i^{(k)}$ für $i = 2^k, \dots, s$ mit Schrittweite 2^k nach (7.25) und (7.26).
2. Für $k = \lceil \log s \rceil, \dots, 0$ Berechnung der Teilvektoren X_i für $i = 2^k, \dots, s$ mit Schrittweite 2^{k+1} durch Lösung des linearen Gleichungssystems (7.27), also

$$B_i^{(k)} X_i = Y_i^{(k)} - A_i^{(k)} X_{i-2^k} - C_i^{(k)} X_{i+2^k}.$$

Zur Berechnung der $\alpha_i^{(k)}$ und $\beta_i^{(k)}$ ist eine Matrixinversion notwendig, deren Durchführung $O(r^3)$ Berechnungen erfordert, der Berechnungsaufwand jeder Stufe steigt also kubisch mit der Bandbreite r . Insgesamt werden in Schritt (1) $O(s) = O(n/r)$ Teilmatrizen berechnet, der sequentielle Gesamtberechnungsaufwand ist also $O(nr^2)$. Im 2. Schritt werden insgesamt $O(s) = O(n/r)$ lineare Gleichungssysteme gelöst, wobei bei Verwendung eines direkten Verfahrens für jedes Gleichungssystem ein Berechnungsaufwand von $O(r^3)$ notwendig ist, der Gesamtberechnungsaufwand ist also ebenfalls $O(nr^2)$.

Für die parallele Implementierung kann das für Tridiagonalsysteme beschriebene Verfahren verwendet werden. Der Unterschied in der Implementierung besteht darin, dass einfache arithmetische Operationen durch Operationen auf Matrizen ersetzt werden, dass also die von den einzelnen Prozessoren durchzuführenden Berechnungen aufwendiger werden. Zwischen den einzelnen Prozessoren werden nicht mehr einzelne Zahlen, sondern $r \times r$ -Matrizen ausgetauscht, also $O(r^2)$ Zahlen. Der Aufwand für lokale Berechnungen ist

wegen der Matrixoperationen $O(r^3)$, d.h. der lokale Berechnungsaufwand steigt mit der Bandbreite stärker als die Größe der ausgetauschten Nachrichten. Für $p \ll s$ ist eine effiziente parallele Implementierung zu erwarten.

7.2.4 Anwendung auf die Poisson-Gleichung

Im letzten Abschnitt haben wir gesehen, wie das Verfahren der zyklischen Reduktion auf allgemeine Bandmatrizen verallgemeinert werden kann. Bei der in Abschnitt 7.2.1 beschriebenen Diskretisierung der zweidimensionalen Poisson-Gleichung entsteht eine solche Bandmatrix, deren halbe Bandbreite der Anzahl N der Diskretisierungspunkte in x - bzw. in y -Richtung entspricht, bei der aber nur vier Nebendiagonalen besetzt sind, vgl. Abbildung 7.8. Die entstehende Bandmatrix ist eine $n \times n$ -Matrix mit $n = N^2$. Bei Anwendung des Gaußschen Eliminationsverfahrens bleibt die Bandstruktur der Originalmatrix *nicht* erhalten, da bei Anwendung der Vorwärtselimination zur Elimination der unteren Nebendiagonalen zusätzliche Nichtnulleinträge zwischen den beiden äußeren Nebendiagonalen erzeugt werden. Dadurch entsteht schnell ein Berechnungsaufwand, der dem allgemeiner Bandsysteme mit Bandbreite N entspricht. Die spezielle Gestalt der Koeffizientenmatrix wird also nicht ausgenutzt. Wir betrachten nun die Anwendung der Methode der zyklischen Reduktion auf das aus der Diskretisierung der Poisson-Gleichung entstehende Gleichungssystem $Az = d$ der Gestalt (7.15) und (7.16) und fassen die Koeffizientenmatrix A als Block-Tridiagonalsystem auf, wobei die entstehenden Teilmatrizen eine spezielle Gestalt haben.

Mit den Bezeichnungen aus (7.15) und (7.16) gilt:

$$B_i^{(0)} := \frac{1}{h^2} B \quad \text{für } i = 1, \dots, N,$$

$$A_i^{(0)} := -\frac{1}{h^2} I \quad \text{und} \quad C_i^{(0)} := -\frac{1}{h^2} I \quad \text{für } i = 1, \dots, N.$$

Der Vektor $d \in \mathbb{R}^n$ besteht aus N Teilvektoren $D_j \in \mathbb{R}^n$

$$d = \begin{pmatrix} D_1 \\ \vdots \\ D_N \end{pmatrix} \quad \text{mit} \quad D_j = \begin{pmatrix} d_{(j-1)N+1} \\ \vdots \\ d_{jN} \end{pmatrix}.$$

Analog besteht der Lösungsvektor z aus N Teilvektoren der Länge N , d.h.

$$z = \begin{pmatrix} Z_1 \\ \vdots \\ Z_N \end{pmatrix} \quad \text{mit} \quad Z_j = \begin{pmatrix} z_{(j-1)N+1} \\ \vdots \\ z_{jN} \end{pmatrix}.$$

Mit den Vereinbarungen

$$\begin{aligned}
B^{(0)} &:= B, \\
D_j^{(0)} &:= D_j \quad \text{für } j = 1, \dots, N, \\
D_j^{(k)} &:= 0 \quad \text{für } k = 0, \dots, \lfloor \log N \rfloor, j \in \mathbb{Z} \setminus \{1, \dots, N\}, \\
Z_j &:= 0 \quad \text{für } j \in \mathbb{Z} \setminus \{1, \dots, N\},
\end{aligned}$$

ergibt die Anwendung des Verfahrens der zyklischen Reduktion, dass für $k = 1, \dots, \lfloor \log N \rfloor$ die Matrizen $B^{(k)} \in \mathbb{R}^{N \times N}$ und die Vektoren $D_j^{(k)} \in \mathbb{R}^N$ für $j = 1, \dots, N$ wie folgt berechnet werden:

$$\begin{aligned}
B^{(k)} &= (B^{(k-1)})^2 - 2I, \\
D_j^{(k)} &= D_{j-2^{k-1}}^{(k-1)} + B^{(k-1)} D_j^{(k-1)} + D_{j+2^{k-1}}^{(k-1)}. \tag{7.28}
\end{aligned}$$

Für $k = 0, \dots, \lfloor \log N \rfloor$ hat Gleichung (7.27) die spezielle Gestalt:

$$-Z_{j-2^k} + B^{(k)} Z_j - Z_{j+2^k} = D_j^{(k)} \quad \text{für } j = 1, \dots, n. \tag{7.29}$$

Die Gleichungen (7.28) und (7.29) realisieren die zyklische Reduktion, was man mit vollständiger Induktion sieht. Für $k = 0$ stellt Gleichung (7.29) das ursprüngliche Gleichungssystem $Az = d$ dar. Für $0 < k < \lfloor \log N \rfloor$ ergibt sich die Gleichung für $k + 1$, indem man für k und festes $j \in \{1, \dots, N\}$ die Gleichungen

$$\begin{aligned}
-Z_{j-2^{k+1}} + B^{(k)} Z_{j-2^k} - Z_j &= D_{j-2^k}^{(k)} \\
-Z_{j-2^k} + B^{(k)} Z_j - Z_{j+2^k} &= D_j^{(k)} \\
-Z_j + B^{(k)} Z_{j+2^k} - Z_{j+2^{k+1}} &= D_{j+2^k}^{(k)}
\end{aligned} \tag{7.30}$$

betrachtet. Die Multiplikation von (7.29) von links mit $B^{(k)}$ liefert die Gleichung

$$-B^{(k)} Z_{j-2^k} + B^{(k)} B^{(k)} Z_j - B^{(k)} Z_{j+2^k} = B^{(k)} D_j^{(k)}. \tag{7.31}$$

Die Addition der Gleichungen in der ersten Zeile von (7.30), der dritten Zeile von (7.30) und (7.31) liefert die Gleichung

$$-Z_{j-2^{k+1}} - Z_j + B^{(k)} B^{(k)} Z_j - Z_j - Z_{j+2^{k+1}} = D_{j-2^k}^{(k)} + B^{(k)} D_j^{(k)} + D_{j+2^k}^{(k)},$$

die mit den Formeln (7.28) der oben angegebenen Gleichung (7.29) entspricht. Damit kann die zyklische Reduktion für das Gleichungssystem der diskretisierten Poisson-Gleichung wie folgt durchgeführt werden:

1. Für $k = 1, \dots, \lfloor \log N \rfloor$ Berechnung der Matrizen $B^{(k)}$ und der Vektoren $D_j^{(k)}$ für $j = 2^k, \dots, N$ mit Schrittweite 2^k nach (7.28).
2. Berechnung der Lösung des linearen Gleichungssystems

$$B^{(k)} Z_j = D_j^{(k)} + Z_{j-2^k} + Z_{j+2^k}$$

für $k = \lfloor \log N \rfloor, \dots, 0$ und $j = 2^k, \dots, N$ mit Schrittweite 2^{k+1} unter Benutzung der Matrizen und Vektoren aus dem ersten Schritt.

Dabei werden im ersten Schritt $\lfloor \log N \rfloor$ Matrizen und $O(N)$ Teilvektoren berechnet, wobei die Berechnung jeder Matrix eine Matrixmultiplikation mit Aufwand $O(N^3)$ erfordert. Die Berechnung jedes Teilvektors erfordert die Durchführung einer Matrix-Vektor-Multiplikation mit Aufwand $O(N^2)$. Insgesamt hat der erste Schritt den Berechnungsaufwand $O(N^3 \log N)$. Im zweiten Schritt müssen $O(N)$ lineare Gleichungssysteme gelöst werden. Dabei hat die Lösung jedes Gleichungssystems einen Aufwand $O(N^3)$, wenn die spezielle Gestalt der Matrizen $B^{(k)}$ nicht ausgenutzt wird. In [53] wird gezeigt, wie dieser Berechnungsaufwand durch Ausnutzung der Gestalt der Matrizen $B^{(k)}$ reduziert werden kann. Für die parallele Implementierung kann das im vorletzten Abschnitt skizzierte Vorgehen verwendet werden, d.h. es wird im Wesentlichen die Implementierung für Tridiagonalsysteme übernommen.

7.3 Klassische Iterationsverfahren

In diesem Abschnitt geben wir einen Überblick über klassische **Iterationsverfahren** zur Lösung linearer Gleichungssysteme, dem Jacobi-Verfahren, dem Gauß-Seidel-Verfahren und dem SOR-Verfahren (für engl. *successive overrelaxation*), und betrachten deren mögliche Realisierung auf Parallelrechnern. Direkte Lösungsmethoden wie die in Abschnitt 7.1 vorgestellte Gauß-Elimination beruhen meist auf einer Umformung oder Faktorisierung der das Gleichungssystem beschreibenden Matrix. Bei dünnbesetzten Matrizen kann dies zu einer Auffüllung mit Nichtnullelementen und so zu einer dichter Besetzung der Matrix führen, was den Rechenaufwand und die Abspeicherungsmethode stark beeinflusst. Umgehen lässt sich dies entweder durch Anpassung der benutzten direkten Methode auf die spezielle Gestalt der Matrix, z.B. die Gestalt einer Bandmatrix wie in Abschnitt 7.2, oder aber durch die Anwendung eines iterativen Verfahrens.

Iterative Verfahren zur Lösung eines linearen Gleichungssystems $Ax = b$ mit gegebener $(n \times n)$ -Matrix $A \in \mathbb{R}^{n \times n}$ und gegebenem Vektor $b \in \mathbb{R}^n$ berechnen eine Folge von Approximationsvektoren $\{x^{(k)}\}_{k=1,2,\dots}$, die gegen die Lösung $x^* \in \mathbb{R}^n$ konvergieren. Die Berechnung eines Approximationsvektors enthält jeweils eine Matrix-Vektor-Multiplikation mit der *Iterationsmatrix* des speziellen Problems. Die Matrix A des Gleichungssystems geht nur in diese Matrix-Vektor-Multiplikation über die in jedem Iterationsschritt benutzte Iterationsmatrix ein. Ein wesentliches Kriterium zur Bewertung von Iterationsverfahren ist die Konvergenzgeschwindigkeit. Einfache iterative Verfahren sind das Jacobi- und das Gauß-Seidel-Verfahren, die historisch auch als „Relaxationsverfahren“ bezeichnet werden, da die Berechnung einer neuen Approximation auf der Kombination von Elementen des letzten Approximationsvektors beruht. Obwohl diese Relaxationsverfahren abhängig von der Problemstellung und dem gewählten Verfahren schneller als direkte Lösungsmethoden sein können, sind diese Verfahren oft zu langsam für den praktischen Einsatz. Eine bessere Konvergenz als diese Verfahren weist

z.B. das SOR-Verfahren auf. In Kombination mit Verfahren wie dem konjugierten Gradientenverfahren oder dem Mehrgitterverfahren haben Relaxationsverfahren jedoch immer noch praktische Bedeutung. Zudem bieten iterative Lösungsverfahren eine gute Einführung in iterative Methoden und zeigen nicht zuletzt charakteristische Datenabhängigkeitsstrukturen auf, die wesentlichen Einfluß auf eine parallele Realisierung haben. Im Folgenden beschreiben wir Relaxationsverfahren nach einer Darstellung von [17], siehe auch [61, 155]. Parallelisierungen werden u.a. in [52, 53, 62, 141] betrachtet.

7.3.1 Beschreibung iterativer Verfahren

Klassische Iterationsverfahren zur Lösung von linearen Gleichungssystemen $A \in \mathbb{R}^{n \times n}$ basieren auf einer Zerlegung der Matrix A in

$$A = M - N, \quad \text{mit } M, N \in \mathbb{R}^{n \times n},$$

wobei M eine nichtsinguläre Matrix ist, für die die inverse Matrix M^{-1} leicht zu berechnen ist, also z.B. eine Diagonalmatrix. Die gesuchte Lösung x^* des Gleichungssystems $Ax = b$ erfüllt die Gleichung

$$Mx^* = Nx^* + b.$$

Diese Gleichung induziert eine Iterationsvorschrift $Mx^{(k+1)} = Nx^{(k)} + b$, $k = 0, 1, \dots$, die typischerweise in folgender Form geschrieben wird

$$x^{(k+1)} = Cx^{(k)} + d \tag{7.32}$$

mit $C := M^{-1}N$ und $d := M^{-1}b$. Das Iterationsverfahren heißt *konvergent* gegen x^* , falls ein Vektor $x^* \in \mathbb{R}^n$ existiert, so dass unabhängig von der Wahl des Startvektors $x^{(0)} \in \mathbb{R}^n$ die Folge $\{x^{(k)}\}_{k=1,2,\dots}$ gegen x^* konvergiert, d.h. $\lim_{k \rightarrow \infty} x^{(k)} = x^*$ bzw. $\lim_{k \rightarrow \infty} \|x^{(k)} - x^*\| = 0$. Im Falle der Konvergenz existiert ein *eindeutiger* Vektor x^* mit $x^* = Cx^* + d$. Subtrahiert man diese letzte Gleichung von Gleichung (7.32), so folgt durch Induktion $x^{(k)} - x^* = C^k(x^{(0)} - x^*)$, wobei C^k die k -fache Multiplikation der Matrix C bezeichne. Die Konvergenz von Iterationsvorschrift (7.32) ist also äquivalent zu

$$\lim_{k \rightarrow \infty} C^k = 0.$$

Ein Satz der linearen Algebra stellt einen Zusammenhang zwischen diesem Konvergenzkriterium und dem Spektralradius $\rho(C)$ der Iterationsmatrix C her. Der Spektralradius einer Matrix bezeichnet den betragsmäßig größten Eigenwert λ , d.h. $\rho(C) = \max_{\lambda \in EW} |\lambda|$ mit $EW = \{\lambda \mid Cv = \lambda v, v \neq 0\}$. Es gilt die Äquivalenz der folgenden Aussagen [155]:

- (1) Iteration (7.32) konvergiert für jedes $x^{(0)} \in \mathbb{R}^n$,
- (2) $\lim_{k \rightarrow \infty} C^k = 0$,
- (3) $\rho(C) < 1$.

Das **Jacobi-Verfahren** (oder **Gesamtschritt-Verfahren**) basiert auf der Zerlegung $A = D - L - R$, ($D, L, R \in \mathbb{R}^{n \times n}$) der Matrix A , wobei D die Diagonale, $-L$ die untere und $-R$ die obere Dreiecksmatrix (jeweils ohne Diagonale) mit den entsprechenden Elementen von A besetzt hat. Alle übrigen Einträge sind Null. Die Zerlegung wird in der Form

$$Dx^{(k+1)} = (L + R)x^{(k)} + b$$

für einen Iterationsschritt genutzt, führt also auf ein Iterationsverfahren mit Iterationsmatrix $C_G := D^{-1}(L + R)$ bzw.

$$C_G = (c_{ij})_{i,j=1,\dots,n} \text{ mit } c_{ij} = \begin{cases} -a_{ij}/a_{ii} & \text{für } j \neq i, \\ 0 & \text{sonst.} \end{cases}$$

Für den Nachweis der Konvergenz wird diese Matrixschreibweise benutzt. Für die praktische Ausführung hingegen ist die folgende Komponentenschreibweise für einen Iterationsschritt günstiger

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1, j \neq i}^n a_{ij} x_j^{(k)} \right), \quad i = 1, \dots, n. \quad (7.33)$$

Bei sequentieller Berechnung der Komponenten in der Reihenfolge $x_1^{(k+1)}, \dots, x_n^{(k+1)}$ nach (7.33) liegen die Werte $x_1^{(k+1)}, \dots, x_{i-1}^{(k+1)}$ bei der Berechnung von $x_i^{(k+1)}$ schon vor. Diese Information wird durch das Gauß-Seidel-Verfahren ausgenutzt.

Das **Gauß-Seidel-Verfahren** (oder **Einzelschritt-Verfahren**) basiert auf der gleichen Zerlegung der Matrix A in $A = D - L - R$ wie das Jacobi-Verfahren, nutzt diese Zerlegung für einen Iterationsschritt jedoch in der Form

$$(D - L)x^{(k+1)} = Rx^{(k)} + b.$$

Die Iterationsmatrix ist also $C_E := (D - L)^{-1}R$. Ein Iterationsschritt in Komponentenschreibweise berechnet

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^n a_{ij} x_j^{(k)} \right), \quad i = 1, \dots, n. \quad (7.34)$$

Konvergenzkriterium. Für das Jacobi- und das Gauß-Seidel-Verfahren wird häufig das folgende, leichter überprüfbare Konvergenzkriterium verwendet, das auf Eigenschaften und Gestalt der Matrix A beruht: Erfüllt die $(n \times n)$ -Matrix A das **starke Zeilensummenkriterium** (bzw. ist A stark diagonaldominant), d.h.

$$|a_{ii}| > \sum_{j=1, j \neq i}^n |a_{ij}|, \quad i = 1, \dots, n,$$

dann konvergieren Jacobi- und Gauß-Seidel-Verfahren. Eine größere Dominanz der Diagonalelemente ergibt häufig eine bessere Konvergenz der Verfahren. Konvergiert eines der beiden Verfahren überhaupt, so konvergiert das Gauß-Seidel-Verfahren in vielen praktischen Fällen schneller als das Jacobi-Verfahren, dies kann aber immer noch sehr langsam sein. Zur Beschleunigung der Konvergenz werden zusätzlich **Relaxationsparameter** $\omega \in \mathbb{R}$ eingeführt.

Das **JOR-Verfahren**, auch relaxiertes Jacobi-Verfahren genannt (engl. *Jacobi overrelaxation*), basiert auf einer Zerlegung $A = \frac{1}{\omega}D - L - R - \frac{1-\omega}{\omega}D$ der Matrix A und benutzt in Komponentenschreibweise die Iterationsvorschrift

$$x_i^{(k+1)} = \frac{\omega}{a_{ii}} \left(b_i - \sum_{j=1, j \neq i}^n a_{ij} x_j^{(k)} \right) + (1 - \omega) x_i^{(k)}, \quad i = 1, \dots, n. \quad (7.35)$$

Das **SOR-Verfahren**, auch relaxiertes Gauß-Seidel-Verfahren genannt (engl. *successive overrelaxation*), erhöht die Konvergenzgeschwindigkeit des Gauß-Seidel-Verfahrens durch Einführung eines Relaxationsparameters ω , der zur Kombination von alten und neuem Iterationsvektor genutzt wird. Dazu wird der nach der Gauß-Seidel-Iterationsvorschrift berechnete neue Iterationsvektor als Zwischenergebnis $\hat{x}^{(k+1)}$ betrachtet. Der endgültige neue Iterationsvektor wird aus diesem Wert und dem letzten Iterationsvektor berechnet

$$\hat{x}_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^n a_{ij} x_j^{(k)} \right), \quad i = 1, \dots, n, \quad (7.36)$$

$$x_i^{(k+1)} = x_i^{(k)} + \omega (\hat{x}_i^{(k+1)} - x_i^{(k)}), \quad i = 1, \dots, n. \quad (7.37)$$

Setzt man die erste dieser Gleichungen in die zweite Gleichung ein, so ergibt sich folgende Iterationsvorschrift in Komponentenschreibweise

$$x_i^{(k+1)} = \frac{\omega}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^n a_{ij} x_j^{(k)} \right) + (1 - \omega) x_i^{(k)} \quad (7.38)$$

für $i = 1, \dots, n$. Die zugehörige Zerlegung der Matrix A ist $A = \frac{1}{\omega}D - L - R - \frac{1-\omega}{\omega}D$ und ein Iterationsschritt des SOR-Verfahrens in Matrixschreibweise hat die Gestalt

$$(D - \omega L)x^{(k+1)} = (1 - \omega)Dx^{(k)} + \omega Rx^{(k)} + \omega b.$$

Die Konvergenz hängt von den Eigenschaften von A und der Größe des Parameters ω ab. Es gilt z.B.: Ist A symmetrisch und positiv definit und liegt ω im offenen Intervall $(0, 2)$, so konvergiert das SOR-Verfahren für jeden Startvektor $x^{(0)}$. Für weitere Ergebnisse verweisen wir auf [17, 53, 61, 155].

Implementierung durch Matrix-Operationen. Die allgemeine Vorschrift für Iterationsverfahren der Form (7.32) besteht in jedem Iterationsschritt zur Berechnung von $x^{(k+1)}$ aus

- einer Matrix-Vektor-Multiplikation mit der Iterationsmatrix C und
- einer Vektor-Vektor-Addition des Ergebnisses dieser Multiplikation mit dem Vektor d .

Im Einzelfall wird jedoch die spezielle Gestalt der Iterationsmatrix, also C_E für das Jacobi-Verfahren und C_G für das Gauß-Seidel-Verfahren bzw. die Varianten für die relaxierten Verfahren, ausgenutzt. Im Falle des Jacobi-Verfahrens ist also

- eine Matrix-Vektor-Multiplikation von $L + R$ mit $x^{(k)}$,
- eine Vektor-Vektor-Addition des Ergebnisses mit b und
- eine Matrix-Vektor-Multiplikation mit D^{-1} (wobei D als Diagonalmatrix leicht zu invertieren ist)

durchzuführen. In der praktischen sequentiellen Realisierung wird die Formel (7.33) verwendet und es werden nacheinander die Komponenten $x_i^{(k+1)}$, $i = 1, \dots, n$, berechnet. Der gesamte vorherige Approximationsvektor $x^{(k)}$ muss für alle Berechnungen des Iterationsschrittes zur Verfügung stehen.

Betrachtet man das Gauß-Seidel-Verfahren, also die Iterationsmatrix $C_G = (D - L)^{-1}R$, so sind folgende Operationen durchzuführen:

- die Matrix-Vektor-Multiplikation $Rx^{(k)}$ mit oberer Dreiecksmatrix R ,
- eine Vektor-Vektor-Addition mit b und
- das Lösen eines Gleichungssystems mit unterer Dreiecksmatrix $(D - L)$.

Wieder wird in der praktischen sequentiellen Realisierung die komponentenweise Vorschrift (7.34) verwendet. Da innerhalb eines Iterationsschrittes jeweils die neuesten zur Verfügung stehenden Komponenten genutzt werden, kann bei Berechnung von $x_i^{(k+1)}$ der vorherige Wert $x_i^{(k)}$ direkt überschrieben werden.

Das Iterationsverfahren wird abgebrochen, wenn der zuletzt berechnete Iterationsvektor um weniger als eine zuvor festgelegte *Toleranzgrenze* ε von der tatsächlichen Lösung abweicht. Da diese Lösung natürlicherweise nicht bekannt ist, wird in der praktischen Realisierung der *relative Fehler* als Abbruchkriterium genutzt, d.h. es wird nach jedem Iterationsschritt überprüft, ob

$$\|x^{(k+1)} - x^{(k)}\| \leq \varepsilon \|x^{(k+1)}\| \quad (7.39)$$

gilt, wobei $\|\cdot\|$ eine Vektornorm bezeichnet, z.B. $\|x\|_\infty = \max_{i=1,\dots,n} |x_i|$ oder $\|x\|_2 = (\sum_{i=1}^n |x_i|^2)^{\frac{1}{2}}$.

7.3.2 Parallele Realisierung des Jacobi-Verfahrens

Nach Berechnungsvorschrift (7.33) für einen Iterationsschritt des Jacobi-Verfahrens sind die Berechnungen der verschiedenen Komponenten $x_i^{(k+1)}$ des Vektors $x^{(k+1)}$ für $i = 1, \dots, n$ unabhängig voneinander und können parallel zueinander ausgeführt werden. Jeder Iterationsschritt kann also maximal von $p = n$ Prozessoren parallel bearbeitet werden. Der berechnete neue Wert $x_i^{(k+1)}$ ist bei Rechnern mit verteiltem Speicher zunächst nur auf dem berechnenden Prozessor verfügbar. Da aber zur Berechnung einer Komponente eines Iterationsvektors $x_i^{(k+1)}$ jeweils der gesamte vorherige Iterationsvektor $x^{(k)}$ benötigt wird, müssen die Komponenten auf allen Prozessoren lokal zur Verfügung gestellt werden, d.h. es muss eine replizierte Verteilung verwendet werden. Dies kann z.B. durch eine Multi-Broadcastoperation im Anschluss an die Berechnung realisiert werden.

Eine andere Sichtweise besteht darin, die Jacobi-Iteration wie im letzten Unterabschnitt ausgeführt als aus Matrix-Vektor-Multiplikation und Vektor-Vektor-Addition zusammengesetzt zu betrachten. Es können also die in Abschnitt 3.7.3 ausgeführten Parallelisierungen benutzt werden, wobei wir die Iterationsmatrix C nicht explizit aufbauen, sondern entsprechend Iterationsvorschrift (7.35) die Matrix A ohne die Diagonale nutzen. Die Verteilung der Berechnung der Komponenten von $x^{(k+1)}$ gemäß (7.33) auf die einzelnen Prozessoren entspricht der *Parallelisierung der inneren Produkte* der Matrix-Vektor-Multiplikation aus Abschnitt 3.7.3. Die zusätzliche Vektor-Vektor-Addition des Jacobi-Verfahrens kann nach der Multi-Broadcastoperation auf allen Prozessoren stattfinden oder sie kann vor die Multi-Broadcastoperation gezogen und verteilt über die Prozessoren ausgeführt werden. Entsprechend kann auch die *Parallelisierung der Linearkombinationen* der Matrix-Vektor-Multiplikation aus Abschnitt 3.7.3 benutzt werden. Hierbei findet die Vektor-Vektor-Addition *nach* der Akkumulationsoperation statt. In beiden Varianten muss sichergestellt werden, dass der gesamte neu berechnete Iterationsvektor $x^{(k+1)}$ vor dem nächsten Iterationsschritt jedem Prozessor lokal zur Verfügung steht.

Abbildung 7.12 zeigt die Skizze einer Funktion zur Realisierung einer parallelen Implementierung eines Jacobi-Verfahrens unter Einbeziehung von MPI-Kommunikationsoperationen [119]. Dabei wird zur Vereinfachung angenommen, dass die Größe n des zu lösenden Gleichungssystems ein Vielfaches der Anzahl p der Prozessoren ist. Es wird eine zeilenorientierte blockweise Verteilung der Iterationsmatrix verwendet. Die Iterationsmatrix sei beim Start der Funktion entsprechend auf die Prozessoren verteilt, d.h. jeder Prozessor speichert lokal in der Variablen `local_A` n/p Zeilen der Iterationsmatrix. Der Vektor b der rechten Seite sei in `local_b` entsprechend abgelegt. MPI entsprechend startet die Numerierung der Prozessoren bei 0. Daher speichert der Prozessor mit Prozessornummer `me` mit $0 \leq \text{me} < p$ die Zeilen `me · n/p + 1, ..., (me + 1) · n/p` von A in `local_A` und entsprechende Komponenten von b in `local_b`. Für die Durchführung der Iteration werden zwei

```

int Parallel_jacobi(int n, int p, int max_it, float tol)
{
    int i_local, i_global, j, i;
    int n_local, it_num;
    float x_temp1[GLOB_MAX], x_temp2[GLOB_MAX], local_x[GLOB_MAX];
    float *x_old, *x_new, *temp;

    n_local = n/p; /* lokale Blockgroesse */
    MPI_Allgather(local_b, n_local, MPI_FLOAT, x_temp1, n_local,
                  MPI_FLOAT, MPI_COMM_WORLD);
    x_new = x_temp1;
    x_old = x_temp2;
    it_num = 0;
    do {
        it_num ++;
        temp = x_new; x_new = x_old; x_old = temp;
        for (i_local = 0; i_local < n_local; i_local++) {
            i_global = i_local + me * n_local;
            local_x[i_local] = local_b[i_local];
            for (j = 0; j < i_global; j++)
                local_x[i_local] = local_x[i_local] -
                                   local_A[i_local][j] * x_old[j];
            for (j = i_global+1; j < n; j++)
                local_x[i_local] = local_x[i_local] -
                                   local_A[i_local][j] * x_old[j];
            local_x[i_local] = local_x[i_local] / local_A[i_local][i_global];
        }
        MPI_Allgather(local_x, n_local, MPI_FLOAT, x_new, n_local,
                      MPI_FLOAT, MPI_COMM_WORLD);
    } while ((it_num < max_it) && (distance(x_old,x_new,n) >= tol));
    output(x_new,global_x);
    if (distance(x_old, x_new, n) < tol ) return 1;
    else return 0;
}

```

Abb. 7.12. Skizze eines C-Programms zur Implementierung des Jacobi-Verfahrens mit MPI-Kommunikationsoperationen. Die Felder `local_x`, `local_b` und `local_A` seien global deklariert. Das Feld `local_A` ist als Feld der Dimension $n_{\text{local}} \times n$ deklariert und nicht in dem in Abbildung 7.3 dargestellten Abspeicherungsformat. Daher unterscheidet sich die Adressierung der Matrixelemente von der Adressierung im sequentiellen Fall.

lokale Felder `x_old` und `x_new` verwendet, in denen der vorangegangene und der aktuelle Iterationsvektor repliziert abgelegt werden. `GLOB_MAX` ist eine symbolische Konstante, die die maximale Größe des zu lösenden Gleichungssystems angibt. Die zu den lokal abgelegten Zeilen von A gehörenden Komponenten des Iterationsvektors werden in `local_x` entsprechend Gleichung (7.33) berechnet. Danach werden die verteilt errechneten Komponenten durch eine `MPI_Allgather()`-Anweisung, die eine Multi-Broadcastoperation realisiert, aufgesammelt, so dass jedem Prozessor der gesamte Iterationsvektor

in `x_new` zur Verfügung steht. Die Iteration stoppt, wenn entweder eine vorgegebene maximale Anzahl `max_it` von Iterationsschritten erreicht ist oder wenn die Veränderung des Iterationsvektors eine vorgegebene Toleranzgrenze `tol` unterschreitet. Die Funktion `distance()` realisiert die Berechnung einer Maximumnorm. Die Funktion `output(x_new, global_x)` besetzt die globale Variable `global_x` mit dem Resultatvektor und gibt das Ergebnis aus.

7.3.3 Parallele Realisierung des Gauß-Seidel-Verfahrens

Die Iterationsvorschrift (7.34) für einen Iterationsschritt des Gauß-Seidel-Verfahrens enthält Datenabhängigkeiten: die Berechnung von $x_i^{(k+1)}$ benutzt die im gleichen Iterationsschritt berechneten Werte $x_1^{(k+1)}, \dots, x_{i-1}^{(k+1)}$. Daher müssen die Komponenten $x_i^{(k+1)}$, $i = 1, \dots, n$, des zu berechnenden neuen Iterationsvektors *nacheinander* berechnet werden. Da Berechnungsschritt (7.34) für jedes $i \in \{1, \dots, n\}$ ein inneres Produkt von $(x_1^{(k+1)}, \dots, x_{i-1}^{(k+1)}, 0, x_{i+1}^{(k)}, \dots, x_n^{(k)})$ mit der i -ten Zeile von A enthält, bedeutet dies, dass diese inneren Produkte *nacheinander* berechnet werden müssen. Parallelität ist also nur *innerhalb* der Berechnung eines inneren Produktes zu erzielen, indem die Berechnung von Teilsummen auf verschiedene Prozessoren verteilt wird. Dazu wird eine spaltenorientierte Block-Datenverteilung für die Matrix A gewählt, wobei wir wieder annehmen, dass n ein Vielfaches der Anzahl p der Prozessoren ist. Die Komponenten von Iterationsvektoren sind in der gleichen Weise auf die Prozessoren verteilt wie die Spalten von A . Ein Prozessor P_q mit $1 \leq q \leq p$ berechnet den Anteil am inneren Produkt, für den er die Komponenten von A und des Iterationsvektors besitzt. Für Prozessor P_q ist dies die Teilsumme

$$s_{qi} = \sum_{\substack{j=(q-1) \cdot n/p + 1 \\ j < i}}^{q \cdot n/p} a_{ij} x_j^{(k+1)} + \sum_{\substack{j=(q-1) \cdot n/p + 1 \\ j > i}}^{q \cdot n/p} a_{ij} x_j^{(k)}.$$

Die von den verschiedenen Prozessoren errechneten Teilsummen s_{qi} werden durch eine Einzel-Akkumulation mit einer Addition als Reduktionsoperation zum Wert $x_i^{(k+1)}$ zusammengefasst. Da der Iterationsvektor blockweise verteilt werden soll, wird die Akkumulationsoperation so gewählt, dass das Ergebnis $x_i^{(k+1)}$ auf dem Prozessor P_q zusammengefasst wird, dem die i -te Komponente zugeordnet ist, d.h. x_i wird auf Prozessor q mit $q = \lceil i/(n/p) \rceil$ akkumuliert. Da sich das SOR-Verfahren vom Gauß-Seidel-Verfahren nur in der Verwendung eines Relaxationsparameters unterscheidet, können die für eine parallele Implementierung des Gauß-Seidel-Verfahrens getroffenen Aussagen direkt auf das SOR-Verfahren übertragen werden. Abbildung 7.13 zeigt ein Pseudocodeprogramm für die skizzierte parallele Realisierung des Gauß-Seidel-Verfahrens. Da immer nur die zuletzt berechnete Iterierte einer Komponente benutzt wird, kann $x_i^{(k)}$ direkt mit $x_i^{(k+1)}$ überschrieben werden,

sobald $x_i^{(k+1)}$ berechnet ist. Es wird daher nur ein Vektor x zur Ablage des Iterationsvektors benutzt. Zur Berechnung des Fehlers wird eine Maximumnorm verwendet.

```

n_local = n/p;
do {
    delta_x = 0.0;
    for (i = 0; i < n; i++) {
        s_k = 0.0;
        for (j = 0; j < n_local; j++)
            if (j + me * n_local != i)
                s_k = s_k + local_A[i][j] * x[j];
        root = i/n_local;
        i_local = i % n_local;
        MPI_Reduce(&s_k, &x[i_local], 1, MPI_FLOAT, MPI_SUM, root,
                  MPI_COMM_WORLD);
        if (me == root) {
            x_new = (b[i_local] - x[i_local]) / local_A[i][i_local];
            delta_x = max(delta_x, abs(x[i_local] - x_new));
            x[i_local] = x_new;
        }
    }
    MPI_Allreduce(&delta_x, &global_delta, 1, MPI_FLOAT,
                  MPI_MAX, MPI_COMM_WORLD);
} while(global_delta > tol);

```

Abb. 7.13. Programmfragment eines parallelen Gauß-Seidel-Verfahrens zur Lösung eines linearen Gleichungssystems $Ax = b$ für **dichtbesetzte** Matrizen A . Da die Berechnungen der einzelnen Komponenten eines Iterationsvektors *nacheinander* erfolgen müssen, kann nur Parallelität innerhalb der Berechnung einer Komponente nach (7.34) ausgenutzt werden. Matrix A ist spaltenblockweise verteilt. Die Vektoren x und b sind entsprechend blockweise verteilt. Für den Konvergenztest berechnet jeder Prozessor in `delta_x` die maximale Änderung seiner lokalen Komponenten. In `global_delta` wird die maximale Änderung aller Komponenten mit einer `MPI_Allreduce()`-Anweisung aufgesammelt, so dass jeder Prozessor entscheiden kann, ob bereits Konvergenz eingetreten ist.

Gauß-Seidel-Verfahren für dünnbesetzte Matrizen. Die in (7.34) enthaltenen Datenabhängigkeiten schränken das Parallelitätspotential des Gauß-Seidel- und des SOR-Verfahrens stark ein, so dass nur für große Gleichungssysteme ein annehmbarer Speedup zu erwarten ist. Die Verfahren besitzen bessere Parallelisierungsmöglichkeiten, falls die Matrix $A \in \mathbb{R}^{n \times n}$ des zu lösenden Gleichungssystems $Ax = b$ dünnbesetzt ist und dadurch weniger Datenabhängigkeiten vorhanden sind. Ist $A = (a_{ij})_{i,j=1,\dots,n}$ dünnbesetzt, so werden zur Berechnung von $x_i^{(k+1)}$ *nicht* alle Elemente $x_j^{(k+1)}$, $j < i$, gebraucht: Ist $a_{ij} = 0$, dann hängt die Berechnung von $x_i^{(k+1)}$ *nicht* von $x_j^{(k+1)}$, $j < i$, ab. Die Nullelemente der Matrix A schaffen also zusätzliche Parallelisierungsmöglichkeiten für die Verfahren.

Im folgenden betrachten wir eine dünnbesetzte Matrix mit Bandstruktur, wie sie bei der Diskretisierung der Poisson-Gleichung (7.10) entsteht, vgl. Abschnitt 7.2.1. Zur Berechnung von $x_i^{(k+1)}$ wird die i -te Zeile der in Abbildung 7.8 dargestellten Bandmatrix benötigt. Diese Matrix hat Einträge $a_{ij} \neq 0$ nur für $j = i - \sqrt{n}, i - 1, i, i + 1, i + \sqrt{n}$ und die Berechnung von $x_i^{(k+1)}$ nach (7.34) reduziert sich auf

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - a_{i, i-\sqrt{n}} \cdot x_{i-\sqrt{n}}^{(k+1)} - a_{i, i-1} \cdot x_{i-1}^{(k+1)} - a_{i, i+1} \cdot x_{i+1}^{(k)} - a_{i, i+\sqrt{n}} \cdot x_{i+\sqrt{n}}^{(k)} \right), \quad i = 1, \dots, n. \quad (7.40)$$

Es müssen also die Komponenten $x_{i-\sqrt{n}}^{(k+1)}$ und $x_{i-1}^{(k+1)}$ berechnet sein, bevor $x_i^{(k+1)}$ berechnet werden kann. Die bestehenden Datenabhängigkeiten von $x_i^{(k+1)}$, $i = 1, \dots, n$, zu $x_j^{(k+1)}$, $j < i$, sind auf der zugehörigen Gitterstruktur in Abbildung 7.14 a) für den allgemeinen Fall eines rechteckigen Gitters dargestellt. Wie in Abschnitt 7.2.1 beschrieben wurde, korrespondiert eine Komponente x_i^* des Lösungsvektors und damit auch des Iterationsvektors $x_i^{(k+1)}$ zu Gitterpunkt i , $i = 1, \dots, n$. Die Berechnung von $x_i^{(k+1)}$ hängt von allen Gitterpunktberechnungen $x_j^{(k+1)}$ ab, die links *und* oberhalb des Punktes i liegen. Berechnungen an Gitterpunkten $j > i$, die rechts *und* unterhalb des Punktes i liegen, müssen hingegen auf die Berechnungen von $x_i^{(k+1)}$ warten.

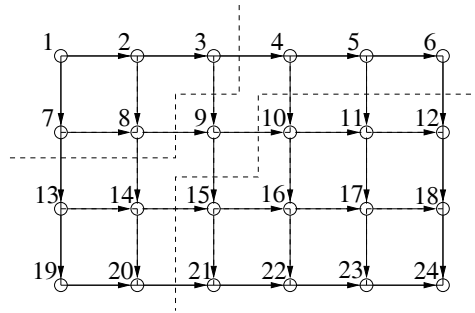
Insbesondere sind die Berechnungen der Komponenten $x_i^{(k+1)}$, die in von links unten nach rechts oben verlaufenden Diagonalen liegen, unabhängig voneinander und können parallel zueinander ausgeführt werden. Die Diagonalen unabhängiger Berechnungen sind in Abbildung 7.14 b) für ein rechteckiges Gitter angegeben. Der resultierende Parallelitätsgrad pro Diagonale ist maximal $\sqrt{n}$, so daß maximal $p = \sqrt{n}$ Prozessoren parallel zueinander Berechnungen ausführen können. Die Berechnung der Werte $x_i^{(k+1)}$ für $i = 1, \dots, n$ kann so auf zwei ineinander geschachtelte Schleifen verteilt werden, dass die unabhängigen Berechnungen jeweils einem inneren Schleifendurchlauf entsprechen.

Für ein quadratisches Gitter der Größe $\sqrt{n} \times \sqrt{n}$ wie in Abbildung 7.8 und eine zeilenweise Numerierung der Gitterpunkte beginnend in der oberen linken Ecke wie in Abbildung 7.14 gibt es $2\sqrt{n} - 1$ Diagonalen, und zwar $\sqrt{n}$ Diagonalen, von denen jede einen anderen Gitterpunkt der ersten Gitterzeile enthält, und weitere $\sqrt{n} - 1$ Diagonalen, die jeweils einen Gitterpunkt der rechten Spalte enthalten. Die ersten $\sqrt{n}$ Diagonalen enthalten für $l = 1, \dots, \sqrt{n}$ die l Gitterpunkte x_i mit

$$i = l + j \cdot (\sqrt{n} - 1) \quad \text{für} \quad 0 \leq j < l.$$

Die restlichen $\sqrt{n} - 1$ Diagonalen enthalten für $l = 2, \dots, \sqrt{n}$ die $\sqrt{n} - l + 1$ Gitterpunkte x_i mit

a) Berechnungsreihenfolgen des SOR-Verfahrens



b) Gitter in x-y-Ebene mit Berechnungsphasen

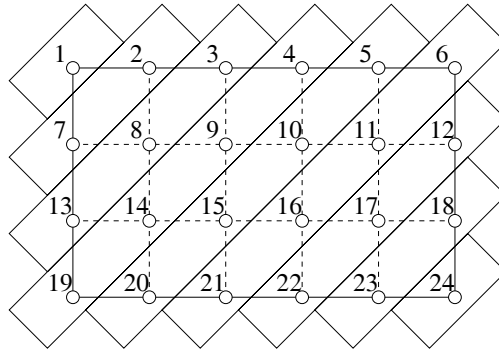


Abb. 7.14. Datenabhängigkeiten im Gauß-Seidel- und SOR-Verfahren für ein rechteckiges Gitter in der x-y-Ebene der Größe 6×4 . a) Die Datenabhängigkeiten der Berechnungen einzelner Komponenten sind durch Pfeile in der Gitterstruktur angegeben. Für Gitterpunkt 9 ist beispielhaft der Bereich gekennzeichnet, der vorher berechnet sein muss, sowie der Bereich, der von der Berechnung bei Gitterpunkt 9 abhängt. b) Die Datenabhängigkeiten erlauben die Unterteilung der Berechnungen an den Gitterpunkten in *Berechnungsphasen*, die Diagonalen im Gitter entsprechen. Die Berechnungen innerhalb einer dieser Phasen sind unabhängig voneinander und können parallel zueinander ausgeführt werden. Die Länge einer Diagonalen entspricht dem Grad der Parallelität.

$$i = l \cdot \sqrt{n} + j \cdot (\sqrt{n} - 1) \quad \text{für} \quad 0 \leq j \leq \sqrt{n} - l.$$

Eine parallele Realisierung auf einem Rechner mit gemeinsamem Adressraum könnte also so aussehen, dass die innere Schleife auf maximal $\sqrt{n}$ Prozessoren verteilt wird. Eine Realisierung auf einem Rechner mit verteiltem Speicher benötigt eine Verteilung des Iterationsvektors x und der rechten Seite b , die zur Abarbeitung der Diagonalen des Gitters korrespondiert. Die Elemente a_{ij} der Matrix A werden so verteilt, dass sie für die Berechnung von $x_i^{(k+1)}$ nach Formel (7.40) lokal zur Verfügung stehen.

Abbildung 7.15 zeigt ein Programmfragment einer parallelen Implementierung des Gauß-Seidel-Verfahrens für die Poisson-Gleichung, das die beschriebene Parallelität innerhalb der Diagonalen ausnutzt. Das Programmfragment verwendet zur Ablage der Koeffizienten eine Matrix, vgl. Abbildung 7.8. Für Gitter mit vielen Gitterpunkten, die zu großen Gleichungssystemen führen, ist es üblich, ein spezielles Abspeicherungsschema für dünnbesetzte Matrizen zu verwenden, das nur die Nichtnullelemente abspeichert. Ein solches Abspeicherungsschema werden wir in Abschnitt 7.4 benutzen. Sind wie im Fall der Poisson-Gleichung die Koeffizienten einer Zeile unabhängig von der speziellen Zeilennummer, können die Koeffizienten auch direkt ins Programm übernommen werden. Dies hat den Vorteil, dass keine Matrix abgespeichert und damit auch keine Feldzugriffe durchgeführt werden müssen, ergibt aber ein Programm, das nur für das spezielle Problem anwendbar ist. Eine Umstellung auf ein anderes Problem erfordert eine Änderung der Koeffizienten und damit eine Änderung des Programms.

Das Programmfragment in Abbildung 7.15 arbeitet die Diagonalen des Gitters in einer äußeren Schleife nacheinander ab, während die innere Schleife über die einzelnen Elemente der Diagonalen läuft. Entsprechend der obigen Beschreibung werden die Diagonalen bis einschließlich der Hauptdiagonalen einerseits und die restlichen Diagonalen andererseits in zwei getrennten äußeren Schleifen behandelt. Die Iterationen der inneren Schleife sind unabhängig voneinander und können über die Prozessoren verteilt werden. In Abbildung 7.15 wird dabei von einer zyklischen Verteilung der Elemente einer Diagonalen auf die Prozessoren ausgegangen und insgesamt eine zeilenzyklische Verteilung der Gitterzeilen auf die Prozessoren erreicht. Nach der Berechnung innerhalb einer Diagonalen werden die neu errechneten Elemente entsprechend der zyklischen Verteilung jeweils an den nächsten Prozessor gesendet, was durch die Funktion `collect_elements()` ausgeführt wird. Für Rechner mit gemeinsamem Speicher findet keine Kommunikation statt, es wird wegen der Datenabhängigkeiten zwischen aufeinanderfolgenden Diagonalberechnungen jedoch eine Synchronisation am Ende jeder inneren Schleife benötigt. Die Details zur Feststellung der Konvergenz sind im Programmfragment nicht enthalten. Zur Berechnung der Maximumnorm von $x^{(k+1)} - x^{(k)}$ kann eine ähnliche Methode wie in Abbildung 7.13 verwendet werden. Das Abspeicherungsformat von A ist gemäß Abbildung 7.3 festgelegt. Damit entspricht die Adressierung der Feldelemente der im sequentiellen Fall verwendeten Adressierung.

7.3.4 Rot-Schwarz-Anordnung

Eine Möglichkeit zur Erhöhung des Parallelitätsgrades des Gauß-Seidel- oder SOR-Verfahrens zur Lösung von *dünnbesetzten* Gleichungssystemen ist eine *Umordnung* der Gleichungen des zu lösenden Gleichungssystems. Ziel der Umordnung ist es, dass sich für einen Iterationsschritt mehr Datenunabhängigkeiten bei der Berechnung der einzelnen Komponenten ergeben. Ein bekanntes

```

sqn = sqrt(n);
do {
    for (l = 1; l <= sqn; l++) {
        for (j = me; j < l; j+=p) {
            i = l + j * (sqn-1) - 1; /* starte Nummerierung bei 0 */
            x[i] = 0;
            if (i-sqn >= 0) x[i] = x[i] - a[i][i-sqn] * x[i-sqn];
            if (i > 0) x[i] = x[i] - a[i][i-1] * x[i-1];
            if (i+1 < n) x[i] = x[i] - a[i][i+1] * x[i+1];
            if (i+sqn < n) x[i] = x[i] - a[i][i+sqn] * x[i+sqn];
            x[i] = (x[i] + b[i]) / a[i][i];
        }
        collect_elements(x,l);
    }
    for (l = 2; l <= sqn; l++) {
        for (j = me -1 +1; j <= sqn -1; j+=p) {
            if (j >= 0) {
                i = l * sqn + j * (sqn-1) - 1;
                x[i] = 0;
                if (i-sqn >= 0) x[i] = x[i] - a[i][i-sqn] * x[i-sqn];
                if (i > 0) x[i] = x[i] - a[i][i-1] * x[i-1];
                if (i+1 < n) x[i] = x[i] - a[i][i+1] * x[i+1];
                if (i+sqn < n) x[i] = x[i] - a[i][i+sqn] * x[i+sqn];
                x[i] = (x[i] + b[i]) / a[i][i];
            }
        }
        collect_elements(x,l);
    }
} while(convergence_test() < tol);

```

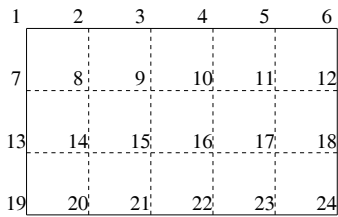
Abb. 7.15. Pseudocodeprogramm eines parallelen Gauß-Seidel-Verfahrens zur Lösung eines linearen Gleichungssystems $Ax = b$ für *dünnbesetzte* Matrizen A , die von der Fünfpunkt-Formel für das Poisson-Problem stammen. Die Berechnungsreihenfolge erfolgt in Diagonalen über das zugehörige zweidimensionale Gitter, vgl. Abbildung 7.14.

Anordnungsschema stellt die **Schachbrett-Nummerierung** (engl. *checker-board*) oder **Rot-Schwarz-Anordnung** (engl. *red-black-ordering*) dar. Angewendet wird die Schachbrett-Nummerierung bei linearen Gleichungssystemen, die von einer Gitterstruktur herrühren, wie etwa dem Modellproblem, vgl. Abschnitt 7.2.1, Abbildung 7.8.

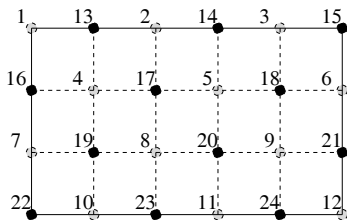
Die Punkte des Gitters werden zunächst in zwei Klassen Rot und Schwarz so aufgeteilt, dass rote Gitterpunkte *nur* schwarze Nachbarpunkte haben und umgekehrt. Die Gitterpunkte beider Klassen werden getrennt zeilenweise durchnummeriert, also je Zeile von links nach rechts und bezüglich der Zeilen von oben nach unten. Sind insgesamt n Gitterpunkte vorhanden, so haben die roten Gitterpunkte die Bezeichnungen $1, \dots, n_R$ und die schwarzen Gitterpunkte die Bezeichnungen $n_R + 1, \dots, n_R + n_S$, wobei n_R die Anzahl der roten und n_S die Anzahl der schwarzen Gitterpunkte ist, $n = n_R + n_S$.

7. Lösung linearer Gleichungssysteme

a) Gitter in x-y-Ebene mit zeilenweiser Numerierung



b) Gitter in x-y-Ebene mit Schachbrett-Numerierung



c) Matrix des linearen Gleichungssystems zur 5-Punkte-Formel bei Schachbrett-Numerierung

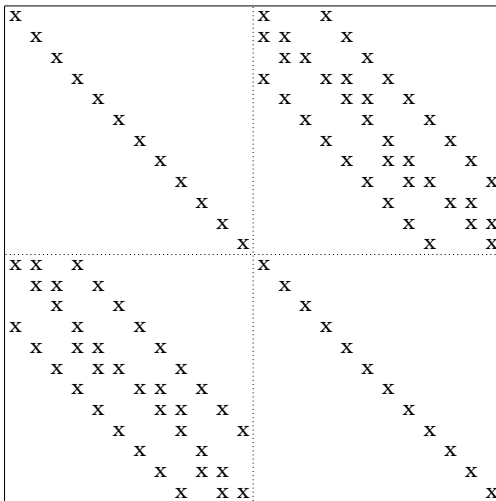


Abb. 7.16. Rechteckiges Gitter der x-y-Ebene der Größe 6×4 mit a) zeilenweiser Numerierung, b) Schachbrett-Numerierung bzw. Rot-Schwarz-Anordnung und c) Matrix des zugehörigen linearen Gleichungssystems für die Fünfpunkt-Formel bei Verwendung einer Schachbrett-Nummerierung.

Abbildung 7.16 b) zeigt diese Nummerierung. Die den Gitterpunkten zugeordneten Unbekannten erhalten die gleiche Nummerierung wie die Gitterpunkte, so dass die ersten n_R Unbekannten $\hat{x}_1, \dots, \hat{x}_{n_R}$ den roten Gitterpunkten und die letzten n_S Unbekannten $\hat{x}_{n_R+1}, \dots, \hat{x}_{n_R+n_S}$ den schwarzen Gitterpunkten zugeordnet sind. Das der Schachbrett-Nummerierung entsprechende Gleichungssystem $\hat{A}\hat{x} = \hat{b}$ für das Poisson-Problem hat die Gestalt

$$\hat{A} \cdot \hat{x} = \begin{pmatrix} D_R & F \\ E & D_S \end{pmatrix} \cdot \begin{pmatrix} \hat{x}_R \\ \hat{x}_S \end{pmatrix} = \begin{pmatrix} \hat{b}_1 \\ \hat{b}_2 \end{pmatrix}, \quad (7.41)$$

wobei $\hat{x}_R$ die ersten n_R Unbekannten und $\hat{x}_S$ die restlichen n_S Unbekannten bezeichne. Die rechte Seite $\hat{b}$ ist entsprechend in Teilstücke $\hat{b}_1$ der Länge n_R und $\hat{b}_2$ der Länge n_S aufgeteilt. Die Matrix $\hat{A}$ besteht aus vier Blöcken $D_R \in \mathbb{R}^{n_R \times n_R}$, $D_S \in \mathbb{R}^{n_S \times n_S}$, $E \in \mathbb{R}^{n_S \times n_R}$ und $F \in \mathbb{R}^{n_R \times n_S}$. Die Teilmatrizen D_R und D_S sind Diagonalmatrizen, die Teilmatrizen E und F sind dünn besetzte Bandmatrizen. Abbildung 7.16 c) zeigt die Matrix $\hat{A}$ für das Poisson-Problem mit der Schachbrett-Nummerierung des Gitters aus Abbildung 7.16 b). Die Diagonalgestalt von D_R (bzw. D_S) zeigt, dass ein roter (bzw. schwarzer) Gitterpunkt *keine* Abhängigkeiten zu anderen roten (bzw. schwarzen) Gitterpunkten hat. Eine Zeile i der Matrix F drückt die Interaktionen zwischen den rot gefärbten Unbekannten $\hat{x}_i$ ($i < n_R$) und den schwarz gefärbten Unbekannten $\hat{x}_j$, $j \in \{n_R+1, \dots, n_R+n_S\}$ aus. Analog drückt Matrix E die Interaktionen von schwarz gefärbten Unbekannten zu rot gefärbten aus.

Das neu erhaltene Gleichungssystem $\hat{A}\hat{x} = \hat{b}$ kann mit Hilfe einer Permutationsabbildung zum ursprünglichen Gleichungssystem $Ax = b$ für das Poisson-Problem, das bei zeilenweiser Nummerierung entstanden ist, in Beziehung gesetzt werden. Sei $\pi : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$ die Permutation, die Nummer i aus der zeilenweisen Nummerierung auf die Nummer $\pi(i)$ der Schachbrett-Nummerierung abbildet. Es gilt

$$x_i = \hat{x}_{\pi(i)}, \quad b_i = \hat{b}_{\pi(i)}, \quad i = 1, \dots, n \quad \text{bzw.} \quad x = P\hat{x} \quad \text{und} \quad b = P\hat{b}$$

mit Permutationsmatrix $P = (P_{ij})_{i,j=1,\dots,n}$, $P_{ij} = \begin{cases} 1 & \text{falls } j = \pi(i) \\ 0 & \text{sonst} \end{cases}$. Für

die Matrix A gilt $\hat{A} = P^T A P$. Da $P^T = P^{-1}$ für Permutationsmatrizen gilt, ergibt sich $\hat{A}\hat{x} = P^T A P P^T x = P^T b = \hat{b}$.

Gauß-Seidel-Verfahren mit Rot-Schwarz-Anordnung. Die Lösung des Gleichungssystems (7.41) mit dem Gauß-Seidel-Verfahren basiert auf der Zerlegung der Matrix $\hat{A} = \hat{D} - \hat{L} - \hat{U}$ mit $\hat{D}, \hat{L}, \hat{U} \in \mathbb{R}^{n \times n}$,

$$\hat{D} = \begin{pmatrix} D_R & 0 \\ 0 & D_S \end{pmatrix}, \quad \hat{L} = \begin{pmatrix} 0 & 0 \\ -E & 0 \end{pmatrix}, \quad \hat{U} = \begin{pmatrix} 0 & -F \\ 0 & 0 \end{pmatrix},$$

wobei $\hat{D}$ eine Diagonalmatrix, $\hat{L}$ eine untere und $\hat{U}$ eine obere Dreiecksmatrix darstellt. 0 bezeichnet eine Matrix, in der alle Einträge 0 sind. Ein Schritt des Gauß-Seidel-Verfahrens (7.34) hat damit die Form:

$$\begin{pmatrix} D_R & 0 \\ E & D_S \end{pmatrix} \cdot \begin{pmatrix} x_R^{(k+1)} \\ x_S^{(k+1)} \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} - \begin{pmatrix} 0 & F \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} x_R^{(k)} \\ x_S^{(k)} \end{pmatrix} \quad (7.42)$$

für $k = 1, 2, \dots$. Hierbei ist der Iterationsvektor $x^{(k+1)}$ analog zur Schreibweise im Gleichungssystem (7.41) in zwei Teilvektoren $x_R^{(k+1)}$ und $x_S^{(k+1)}$ für die rot bzw. schwarz eingefärbten Unbekannten aufgeteilt. (Zur Vereinfachung der Notation benutzen wir nun x_R anstatt $\hat{x}_R$.) Die Umformung von Iterationsvorschrift (7.42) in Vektorschreibweise für $x_R^{(k+1)}$ und $x_S^{(k+1)}$ ergibt

$$D_R \cdot x_R^{(k+1)} = b_1 - F \cdot x_S^{(k)} \quad \text{für } k = 1, 2, \dots, \quad (7.43)$$

$$D_S \cdot x_S^{(k+1)} = b_2 - E \cdot x_R^{(k+1)} \quad \text{für } k = 1, 2, \dots. \quad (7.44)$$

In dieser Form wird die Entkoppelung der roten und schwarzen Teilvektoren $x_R^{(k+1)}$ und $x_S^{(k+1)}$ offensichtlich. In Gleichung (7.43) wird der neue Iterationsvektor $x_R^{(k+1)}$ *nur* in Abhängigkeit vom alten schwarz gefärbten Iterationsvektor $x_S^{(k)}$ berechnet. In Gleichung (7.44) wird der neue schwarz gefärbte Iterationsvektor $x_S^{(k+1)}$ *nur* in Abhängigkeit vom neuen rot gefärbten Iterationsvektor $x_R^{(k+1)}$ berechnet. Innerhalb der beiden Gleichungen gibt es *keine* Abhängigkeit der Berechnungen der Einzelkomponenten von $x_R^{(k+1)}$ bzw. $x_S^{(k+1)}$ untereinander. Der potentielle Parallelitätsgrad entspricht damit in beiden Gleichungen dem des Jacobi-Verfahrens. In jedem Iterationsschritt kann jede Komponente von $x_R^{(k+1)}$ gemäß (7.43) unabhängig berechnet werden, da der gesamte Vektor $x_R^{(k)}$ ja vorliegt. Danach kann jede Komponente von $x_S^{(k+1)}$ gemäß (7.44) unabhängig berechnet werden, da nun der Anteil $x_R^{(k+1)}$ der neuen Iteration vorliegt. Ein Iterationsschritt des Gauß-Seidel-Verfahrens mit Rot-Schwarz-Anordnung zur Berechnung von $x_R^{(k+1)}$ und $x_S^{(k+1)}$ nach (7.43) und (7.44) hat in Komponentenschreibweise die Form:

$$\begin{aligned} \left(x_R^{(k+1)}\right)_i &= \frac{1}{\hat{a}_{ii}} \left(\hat{b}_i - \sum_{j \in N(i)} \hat{a}_{ij} \cdot (x_S^{(k)})_j \right), \quad i = 1, \dots, n_R, \\ \left(x_S^{(k+1)}\right)_i &= \frac{1}{\hat{a}_{i+n_R, i+n_R}} \left(\hat{b}_{i+n_R} - \sum_{j \in N(i)} \hat{a}_{i+n_R, j} \cdot (x_R^{(k+1)})_j \right), \quad i = 1, \dots, n_S. \end{aligned}$$

Hierbei bezeichnet $N(i)$ die Menge der maximal vier Indizes der Nachbargitterpunkte des zu Index i gehörenden Gitterpunktes. Nach Konstruktion der Rot-Schwarz-Anordnung enthält die Menge $N(i)$ eines rot gefärbten Gitterpunktes nur Indizes für schwarz eingefärbte Gitterpunkte und umgekehrt.

In einer Realisierung auf einem Rechner mit gemeinsamem Adressraum kann die Berechnung auf maximal $p = n_R$ bzw. $p = n_S$ Prozessoren ausgeführt werden, da keine Zugriffskonflikte entstehen. Die Realisierung auf einem Rechner mit verteiltem Speicher benötigt für eine verteilte Berechnung eine Verteilung der Vektoren $x_S^{(k)}$ und $x_R^{(k)}$ auf die lokalen Speicher der

Prozessoren und dadurch explizite Kommunikation der verteilt berechneten Komponenten. Wie in der diagonalen Berechnungsweise des SOR-Verfahrens bietet es sich auch hier an, eine Aufteilung der Berechnung gemäß der Gitterstruktur vorzunehmen. Wird ein Gitterpunkt i des Gitters einem Prozessor P_k zugeordnet, so ist dieser Prozessor für die Aktualisierung der zugehörigen Unbekannten x_i bzw. der Approximationen verantwortlich. Für die systematische Zuordnung von Gitterpunkten zu Prozessoren wird üblicherweise eine ein- oder zweidimensionale Blockstruktur auf dem Gitter gewählt. Wir betrachten zunächst eine eindimensionale Aufteilung eines quadratischen Gitters mit n Gitterpunkten, wobei n eine Quadratzahl sei. In einer zeilenorientierten Aufteilung der Gitterpunkte für p Prozessoren werden die Gitterzeilen in zusammenhängende Blöcke der Größe $\sqrt{n}/p$ unterteilt, die jeweils einem Prozessor P_k , $k \in \{1, \dots, p\}$, zugeordnet werden. Benutzen wir nun die zeilenweise Nummerierung der Rot-Schwarz-Anordnung, so werden einem Prozessor damit $\frac{1}{2} \frac{n}{p}$ rot und $\frac{1}{2} \frac{n}{p}$ schwarz eingefärbte Gitterpunkte zugeordnet. (Wir nehmen hier wieder eine ganzzahlige Teilbarkeit von $\sqrt{n}$ durch p an.) Prozessor P_k werden also die Gitterpunkte

$$(k-1) \cdot \frac{n_R}{p} + 1, \dots, k \cdot \frac{n_R}{p} \quad \text{für } k = 1, \dots, p \quad \text{und} \\ (k-1) \cdot \frac{n_S}{p} + 1 + n_R, \dots, k \cdot \frac{n_S}{p} + n_R \quad \text{für } k = 1, \dots, p$$

zugeordnet. Ein SPMD-Programm für das Gauss-Seidel-Verfahren mit Rot-Schwarz-Anordnung hat dann die in Abbildung 7.17 angegebende Gestalt. Dabei wird wieder die in Abbildung 7.3 verwendete Allokierung für Matrix A verwendet, die eine bequeme Adressierung der Matrixelemente erlaubt. Nach der verteilten Berechnung der Approximationen zu den rot bzw. schwarz gefärbten Gitterpunkten werden diese durch Aufruf der Funktion `collect_elements()`, die im Wesentlichen eine Multi-Broadcastoperation durchführt, allen Prozessoren repliziert zur Verfügung gestellt.

SOR-Verfahren mit Rot-Schwarz-Numerierung. Ein SOR-Iterationsschritt mit Parameter w für das Gleichungssystem (7.41) kann aus dem Gauß-Seidel-Iterationsschritt (7.43) und (7.44) gewonnen werden, indem die aus dem Gauß-Seidel-Schritt berechneten Vektoren nach Formel (7.37) aktualisiert werden. Insgesamt hat ein SOR-Iterationsschritt also die Form

$$\begin{aligned} \tilde{x}_R^{(k+1)} &= D_R^{-1} \cdot b_1 - D_R^{-1} \cdot F \cdot x_S^{(k)}, \\ \tilde{x}_S^{(k+1)} &= D_S^{-1} \cdot b_2 - D_S^{-1} \cdot E \cdot x_R^{(k+1)}, \\ x_R^{(k+1)} &= x_R^{(k)} + \omega \left(\tilde{x}_R^{(k+1)} - x_R^{(k)} \right), \\ x_S^{(k+1)} &= x_S^{(k)} + \omega \left(\tilde{x}_S^{(k+1)} - x_S^{(k)} \right), \quad k = 1, 2, \dots \end{aligned} \tag{7.45}$$

```

local_nr = nr/p; local_ns = ns/p;
do {
    mestarttr = me * local_nr;
    for (i= mestarttr; i < mestarttr + local_nr; i++) {
        xr[i] = 0;
        for (j ∈ N(i))
            xr[i] = xr[i] - a[i][j] * xs[j];
        xr[i] = (xr[i]+b[i]) / a[i][i] ;
    }
    collect_elements(xr);
    mestarts = me * local_ns + nr;
    for (i= mestarts; i < mestarts + local_ns; i++) {
        xs[i] = 0;
        for (j ∈ N(i))
            xs[i] = xs[i] - a[i+nr][j] * xr[j];
        xs[i] = (xs[i] + b[i+nr]) / a[i+nr][i+nr];
    }
    collect_elements(xs);
} while (convergence_test());

```

Abb. 7.17. Programmskizze zu einer parallelen Realisierung des Gauß-Seidel-Verfahrens mit Rot-Schwarz-Anordnung. Die Unbekannten zu den rot bzw. schwarz gefärbten Gitterpunkten werden in den Feldern xr bzw. xs errechnet. Die Prozessornummer des ausführenden Prozessors sei in me abgelegt.

Die entsprechende Zerlegung der Matrix $\hat{A}$ ist $\hat{A} = \frac{1}{\omega}\hat{D} - \hat{L} - \hat{U} - \frac{1-\omega}{\omega}\hat{D}$ mit obigen Matrizen $\hat{D}, \hat{L}, \hat{U}$. In der Matrixschreibweise mit Blockmatrizen hat ein SOR-Iterationsschritt dann die Form

$$\begin{pmatrix} D_R & 0 \\ \omega E & D_S \end{pmatrix} \cdot \begin{pmatrix} x_R^{(k+1)} \\ x_S^{(k+1)} \end{pmatrix} \quad (7.46)$$

$$= (1 - \omega) \begin{pmatrix} D_R & 0 \\ 0 & D_S \end{pmatrix} \cdot \begin{pmatrix} x_R^{(k)} \\ x_S^{(k)} \end{pmatrix} - \omega \begin{pmatrix} 0 & F \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} x_R^{(k)} \\ x_S^{(k)} \end{pmatrix} + \omega \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

Die Iterationsmatrix für das SOR-Verfahren ist also

$$\hat{S}_\omega = \left(\frac{1}{\omega} \hat{D} - \hat{L} \right)^{-1} \left(\frac{1-\omega}{\omega} \hat{D} + \hat{U} \right).$$

Für die praktische Implementierung wird man wieder die Komponentenschreibweise des SOR-Iterationsschrittes heranziehen. Für den Nachweis der Konvergenz wird jedoch die SOR-Iterationsmatrix $\hat{S}_\omega$ betrachtet. Es ist also zu zeigen, dass für den Spektralradius $\rho(\hat{S}_\omega) < 1$ für $\omega \in (0, 2)$ gilt. Wie bereits erwähnt wurde, gilt zwar $P^T A P = \hat{A}$, die Matrix $P^T S_\omega P$ ist jedoch *nicht* identisch zu S_ω , wobei S_ω die Iterationsmatrix des SOR-Verfahrens für

das Problem $Ax = b$ bezeichne. Aussagen zur Konvergenz sind z.B. in [53] zu finden.

Die bisherige Darstellung des Gauß-Seidel- und SOR-Verfahrens nutzte die Indizes $i = 1, \dots, n$ des zu lösenden Gleichungssystems $\hat{A}\hat{x} = \hat{b}$. Wie diese Darstellung auch gezeigt hat, basiert die Idee der Schachbrett-Nummerierung aber im Wesentlichen darauf, dass die Rot-Schwarz-Anordnung auf der zugrundeliegenden zweidimensionalen Gitterstruktur beruht und deren Nachbarschaftsbeziehungen entkoppelt. Die Parallelisierung des Gauß-Seidel-Verfahrens verwendete eine solche Zerlegung der Gitterstruktur (und nicht der Matrix), um so Nachbarschaftsbeziehungen zwischen Gitterpunkten zu nutzen. Für diesen Ansatz ist es oft einfacher, für das parallele Programm die zweidimensionalen Indizes der Gitterstruktur zu nutzen. Abbildung 7.18 zeigt eine Programmskizze für eine parallele SOR-Realisierung mit Rot-Schwarz-Nummerierung, die die Indizes verwendet, wie sie in Abschnitt 7.2.1 für die Diskretisierung der Poisson-Gleichung eingeführt wurden. Dabei werden im Gegensatz zur Programmskizze in Abbildung 7.17 die Koeffizienten nicht in einer Matrix abgelegt, sondern direkt als Konstanten ins Programm aufgenommen. Das zweidimensionale Feld `x` dient zur Aufnahme der Unbekannten. Jedem Gitterpunkt ist als zusätzliches Attribut eine Farbe (rot oder schwarz) zugeordnet, deren Wert durch Aufruf der Funktionen `is_red()` bzw. `is_black()` abgefragt werden kann. Die zusätzliche Abfrage nach der Farbe der bearbeiteten Gitterpunkte verteuert diese Implementierung gegenüber der Implementierung aus Abbildung 7.17. Analog zu Abschnitt 7.2.1 bezeichnet `f[i][j]` die Diskretisierung der vorgegebenen Funktion f gemäß der definierten Gitterstruktur, vgl. z.B. Gleichung (7.12), und entspricht der rechten Seite des Gleichungssystems. Der Aufruf der Funktion `exchange_red_borders()` bzw. `exchange_black_borders()` tauscht die roten bzw. schwarzen Gitterpunkte an den Rändern der den einzelnen Prozessoren zugeordneten Bereichen zwischen benachbarten Prozessoren aus.

7.4 Cholesky-Faktorisierung für dünnbesetzte Matrizen

Viele der in der Praxis auftretenden linearen Gleichungssysteme sind groß, haben aber eine dünnbesetzte Koeffizientenmatrix, d.h. viele Einträge der Koeffizientenmatrix sind Null. Wenn die Besetzungsstruktur der Koeffizientenmatrix eine regelmäßige Bandstruktur aufweist, können die in Abschnitt 7.2.2 besprochenen Verfahren des rekursiven Verdoppels und der zyklischen Reduktion verwendet werden. In diesem Fall brauchen nur die besetzten Diagonalen abgespeichert zu werden. Wenn die Nichtnulleinträge aber unregelmäßig verteilt sind, muss ein allgemeines Abspeicherungsschema und ein anderes paralleles Berechnungsverfahren eingesetzt werden. In diesem Abschnitt untersuchen wir die Cholesky-Zerlegung einer dünnbesetzten Matrix als Beispiel. In Abschnitt 7.4.1 beschreiben wir den sequentiellen Algorithmus und verschiedene Varianten der Realisierung, die sich für dünnbesetzte


```

do {
  for ((i,j) ∈ myregion) {
    if (is_red(i,j))
      x[i][j] = omega/4 * (h*h*f[i][j] + x[i][j-1] + x[i][j+1]
        + x[i-1][j] + x[i+1][j]) + (1- omega) * x[i][j];
  }
  exchange_red_borders(x);
  for ((i,j) ∈ myregion) {
    if (is_black(i,j))
      x[i][j] = omega/4 * (h*h*f[i][j] + x[i][j-1] + x[i][j+1]
        + x[i-1][j] + x[i+1][j]) + (1- omega) * x[i][j];
  }
  exchange_black_borders(x);
} while (convergence_test());

```

Abb. 7.18. Programmskizze eines parallelen Gauß-Seidel-Verfahrens für Rot-Schwarz-Anordnung mit zweidimensionaler Gitternummerierung. Die einem Prozessor zugeordneten Gitterpunkte sind in myregion abgelegt.

Matrizen ergeben. In Abschnitt 7.4.2 beschreiben wir Abspeicherungsschemata für dünnbesetzte Matrizen und in Abschnitt 7.4.3 untersuchen wir die Realisierung der verschiedenen Varianten aus Abschnitt 7.4.1 für einen Parallelrechner mit gemeinsamen Adressraum.

7.4.1 Sequentieller Algorithmus

Ebenso wie das Gaußsche Eliminationsverfahren ist die Cholesky-Zerlegung ein *direktes* Verfahren zur Lösung von linearen Gleichungssystemen $Ax = b$. Voraussetzung für die Anwendung des Verfahrens ist, dass die Koeffizientenmatrix $A = (a_{ij}) \in \mathbb{R}^{n \times n}$ des linearen Gleichungssystems *symmetrisch* und *positiv-definit* ist, d.h. dass $a_{ij} = a_{ji}$ ist und dass für alle $x \in \mathbb{R}^n$ mit $x \neq 0$ gilt, dass $x^T Ax > 0$ ist. In diesem Fall existiert für $A \in \mathbb{R}^{n \times n}$ eine eindeutige Dreieckszerlegung

$$A = L \cdot L^T,$$

siehe z.B. [57], wobei $L \in \mathbb{R}^{n \times n}$ eine untere Dreiecksmatrix mit positiven Diagonalelementen ist. L^T bezeichnet die transponierte Matrix von L . Mit der gegebenen Dreieckszerlegung kann das lineare Gleichungssystem $Ax = b$ in zwei Schritten gelöst werden. Im ersten Schritt wird das untere Dreieckssystem $Ly = b$ durch Vorwärtseinsetzen gelöst. Im zweiten Schritt wird das obere Dreieckssystem $L^T x = y$ durch Rückwärtseinsetzen gelöst. Dabei ist $x \in \mathbb{R}^n$ wegen $Ly = LL^T x = Ax = b$ die Lösung des ursprünglichen Gleichungssystems. Wegen der speziellen Gestalt von L und L^T können die Gleichungssysteme $Ly = b$ und $L^T x = y$ in Zeit $O(n^2)$ gelöst werden, vgl. Abschnitt 7.1. Die Lösung des Originalsystems mit Hilfe des Gaußschen Eliminationsverfahrens erfordert dagegen Aufwand $O(n^3)$. Wir beschreiben

im Folgenden kurz, wie die Cholesky-Zerlegung von A berechnet werden kann. Für eine ausführliche Behandlung verweisen wir auf [57].

Die Matrix $L = (l_{ij})_{\substack{i=0,\dots,n-1 \\ j=0,\dots,i}}$ kann spaltenweise aus $A = (a_{ij})_{i,j=0,\dots,n-1}$ berechnet werden, indem die Spalten von L nacheinander von links nach rechts mit folgenden Algorithmus erzeugt werden:

```

for (j=0; j<n; j++) {
     $l_{jj} = \sqrt{a_{jj} - \sum_{k=0}^{j-1} l_{jk}^2};$ 
    for (i=j+1; i<n; i++)
         $l_{ij} = \frac{1}{l_{jj}} \left( a_{ij} - \sum_{k=0}^{j-1} l_{jk} l_{ik} \right);$ 
}

```

Zur Bestimmung der Spalte j von L wird also zuerst das Diagonalelement l_{jj} der Spalte bestimmt, wozu die vorher bestimmten Einträge der Zeile j von L verwendet werden. Danach werden die restlichen Einträge von Spalte j bestimmt, wobei zur Berechnung von l_{ij} Zeile j von L und die bisher berechneten Einträge von Zeile i von L verwendet werden, vgl. Abbildung 7.19 zur Illustration. Die Errechnung jedes Elements von L erfordert $O(n)$ Berechnungen. Somit erfordert die Errechnung der gesamten Matrix L $O(n^3)$ Berechnungen [154].

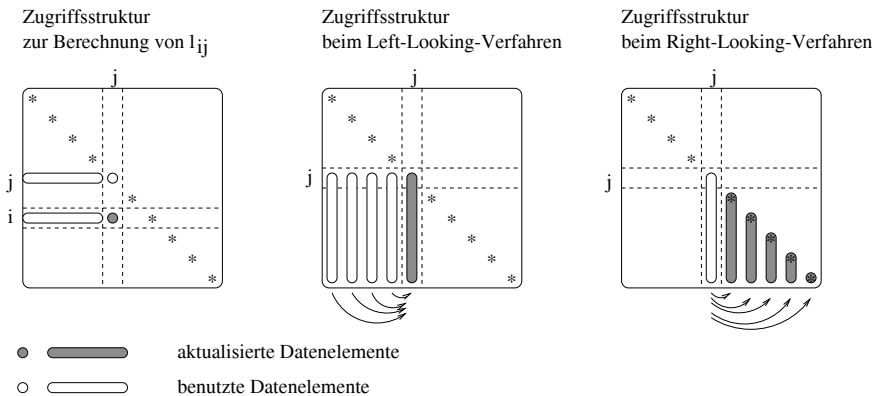


Abb. 7.19. Zugriffsstruktur für die Berechnung von L bei Verwendung des Grundalgorithmus (links), des Left-Looking-Algorithmus (mitte) und des Right-Looking-Algorithmus (rechts).

Der beschriebene Algorithmus macht keinen Gebrauch davon, dass bestimmte Einträge der Eingabematrix A Null sein können, obwohl die Berücksichtigung dieser Tatsache viel Rechenzeit sparen kann. Zur Berücksichtigung

der Besetzungsstruktur von A muss A in einer Datenstruktur abgelegt werden, die nur die Nichtnullelemente abspeichert. Wir gehen in Abschnitt 7.4.2 auf solche Datenstrukturen ein. Dabei ist zu beachten, dass bei der Berechnung von L üblicherweise **Fill-in-Effekte** auftreten, d.h. L hat Nichtnull-einträge an Positionen, an denen A Nulleinträge hatte. Das Auftreten von Fill-in-Effekten kann dadurch verhindert oder erheblich reduziert werden, dass die Unbekannten und die Gleichungen des linearen Gleichungssystems umgeordnet werden. Das Problem, die Fill-in-Effekte durch Umordnen zu minimieren, ist NP-vollständig [165]. Effiziente Heuristiken sind in [57] besprochen. Unter Beachtung der Fill-in-Effekte besteht die Lösung des linearen Gleichungssystems $Ax = b$ mit Hilfe der Cholesky-Zerlegung aus den folgenden 4 Schritten:

1. **Umordnen der Gleichungen** zur Reduktion der Fill-in-Effekte: suche eine Permutationsmatrix $P \in \mathbb{R}^{n \times n}$ mit Hilfe einer geeigneten Heuristik. Das modifizierte Gleichungssystem lautet $(PAP^T)(Px) = Pb$.
2. **Symbolische Faktorisierung**: bestimme die resultierende Besetzungsstruktur von L und allokiere die Datenstruktur zur Ablage der Nichtnullelemente von L entsprechend, siehe [57].
3. **Numerische Faktorisierung**: faktorisiere die modifizierte Matrix PAP^T in LL^T .
4. **Lösung der Dreieckssysteme**: Löse die Dreieckssysteme $Ly = Pb$ und $L^T z = y$. Die Lösung des Originalsystems ist $x = P^T z$.

Von diesen vier Schritten wird die bei weitem am meiste Rechenzeit von der numerischen Faktorisierung gebraucht. Dieser Schritt ist daher auch der wichtigste für eine parallele Ausführung. Wir konzentrieren uns im Folgenden auf die numerische Faktorisierung und nehmen an, dass die Eingabematrix bereits umgeordnet ist. Für die Behandlung der restlichen Punkte verweisen wir auf [57, 165]. Die Besetzungsstruktur der Matrix L kann durch folgende Mengen beschrieben werden [110]

$$Struct(L_{*j}) = \{k > j \mid l_{kj} \neq 0\},$$

$$Struct(L_{i*}) = \{k < i \mid l_{ik} \neq 0\}.$$

$Struct(L_{*j})$ enthält die Zeilenindizes aller Nichtnullelemente von Spalte j , wobei die Diagonalelemente nicht berücksichtigt werden. $Struct(L_{i*})$ enthält die Spaltenindizes aller Nichtnullelemente in Zeile i . Diese Mengen können dazu verwendet werden, die Manipulation der Spalten zur Berechnung von L durch folgende Prozeduren zu beschreiben [110, 138]:

```

cmod(j,k) =
  for each  $i \in Struct(L_{*k})$  with  $i \geq j$  :
     $a_{ij} = a_{ij} - l_{jk}l_{ik}$  ;
cdiv(j) =
   $l_{jj} = \sqrt{a_{jj}}$  ;
  for each  $i \in Struct(L_{*j})$  :
     $l_{ij} = a_{ij}/l_{jj}$  ;

```

Die Prozedur $\text{cmod}(j,k)$ subtrahiert das l_{jk} -fache der bereits errechneten Spalte k von Spalte j . Dabei werden nur die Nichtnullelemente von Spalte k berücksichtigt, da nur diese einen Effekt hervorrufen. Die Einträge a_{ij} der ursprünglichen Matrix werden zur Akkumulation verwendet. Die Prozedur $\text{cdiv}(j)$ berechnet die Quadratwurzel des Diagonaleintrags von Spalte j und dividiert alle Nichtnullelemente dieser Spalte durch diesen Wert. Mit Hilfe dieser Prozeduren kann die Berechnung der Einträge von Spalte j von L dadurch erfolgen, dass zuerst $\text{cmod}(j,k)$ für jedes $k \in Struct(L_{*j})$ aufgerufen wird, d.h. das l_{jk} -fache jeder dieser Spalten wird von Spalte j subtrahiert. Alle anderen Spalten $k \notin Struct(L_{*j})$ brauchen nicht berücksichtigt zu werden, da für diese Spalten wegen $l_{jk} = 0$ kein Beitrag entsteht. Danach wird $\text{cdiv}(j)$ aufgerufen, d.h. l_{jj} wird berechnet und alle Nichtnullelemente erhalten ihren endgültigen Wert. Die Spalten von L werden durch diese Methode von links nach rechts berechnet. Der resultierende Algorithmus wird als **Left-Looking-Algorithmus** bezeichnet, da zur Berechnung einer Spalte j von L alle Spalten *links* von j zur Berechnung verwendet werden.

```

left_cholesky =
  for  $j = 0, \dots, n-1$  {
    for each  $k \in Struct(L_{*j})$ :
      cmod(j,k);
    cdiv(j);
  }

```

Abbildung 7.19 (mitte) zeigt eine Veranschaulichung der Zugriffsstruktur. Das Zugriffsverhalten auf die Spalten ist derart, dass aufeinanderfolgende $\text{cmod}()$ -Operationen dieselbe Zielspalte j manipulieren.

Der Left-Looking-Algorithmus berechnet die Spalten von L in einer strikten Reihenfolge von links nach rechts, d.h. beim Start der Berechnung von Spalte j sind alle Spalten links von j bereits vollständig berechnet. Ein alternatives Berechnungsschema besteht darin, nach der vollständigen Berechnung einer Spalte j alle Spalten k rechts von j , die von j abhängen, durch Aufruf von $\text{cmod}(k,j)$ zu manipulieren, d.h. das l_{kj} -fache von Spalte j von k zu subtrahieren. Wegen $l_{kj} = 0$ für $k \notin Struct(L_{*j})$ müssen nach Berechnung von Spalte j nur die Spalten $k \in Struct(L_{*j})$ manipuliert werden. Die Spalten werden weiterhin von links nach rechts berechnet, die vorgezogenen Aufrufe

von `cmod()` bewirken aber, dass zur vollständigen Berechnung einer Spalte j nur `cdiv( $j$ )` aufgerufen werden muss, wenn bereits alle Spalten links von j vollständig berechnet sind. Es resultiert das folgende Berechnungsschema:

```

right_cholesky =
  for  $j = 0, \dots, n - 1$  {
    cdiv( $j$ );
    for each  $k \in Struct(L_{*j})$ :
      cmod( $k, j$ );
  }
```

Dieses Berechnungsschema wird als **Right-Looking-Algorithmus** bezeichnet, da nach Berechnung einer Spalte j alle Spalten rechts von j mit dieser Spalte manipuliert werden, vgl. Abbildung 7.19 (rechts). Das Zugriffsverhalten auf die Spalten ist derart, dass aufeinanderfolgende `cmod()`-Operationen unterschiedliche Zielspalten mit derselben Ausgangsspalte manipulieren.

In beiden Berechnungsschemata wird die gleiche Anzahl von `cmod()`- und `cdiv()`-Operationen ausgeführt: Jeder Nichtnulleintrag l_{jk} führt zu einem Aufruf von `cmod()`. Beim Left-Looking-Algorithmus ist dies der Aufruf von `cmod( $j, k$ )` bei der Berechnung von Spalte j . Beim Right-Looking-Algorithmus ist es der Aufruf von `cmod( $j, k$ )` zur Manipulation von Spalte $j \in Struct(L_{*k})$ nach der vollständigen Berechnung von Spalte k . Für jede Spalte wird in beiden Schemata eine `cdiv()`-Operation durchgeführt, um die Berechnung der Spalte abzuschließen.

Ein weiteres Berechnungsschema zur Berechnung von L erhält man, wenn man ausnutzt, dass bestimmte benachbarte Spalten eine ähnliche Besetzungsstruktur haben. Für eine genaue Beschreibung führen wir das Konzept von **Superknoten** ein [110, 138]. Eine Menge $I(p) = \{p, p+1, \dots, p+q-1\}$ von aufeinanderfolgenden Spalten von L heißt Superknoten, wenn für alle i mit $p \leq i \leq p+q-1$ gilt, dass

$$Struct(L_{*i}) = Struct(L_{*(p+q-1)}) \cup \{i+1, \dots, p+q-1\}.$$

Ein Superknoten von q benachbarten Spalten, der in Spalte p startet, hat also oberhalb von Zeile $p+q-1$ einschließlich einen dichtbesetzten Dreiecksblock, d.h. alle Einträge sind $\neq 0$. Unterhalb von Zeile $p+q-1$ haben die q Spalten eine identische Besetzungsstruktur, d.h. jede Spalte hat ihre Nichtnulleinträge in den gleichen Zeilen wie die anderen Spalten des Superblocks.

Abbildung 7.20 (links) zeigt ein Beispiel. Da die Spalten eines Superblocks gleiche Besetzungsstruktur haben, modifiziert jede dieser Spalten die gleichen Zielspalten rechts des Superblocks. Damit kann die Modifikation der Spalten so beschrieben werden, dass die Spalten der Superblöcke zusammengefasst werden. Um dies zu beschreiben, bezeichnen wir für einen Superknoten $I(p) = \{p, \dots, p+q-1\}$ mit $first(I) = p$ die erste Spalte von I und mit $last(I) = p+$

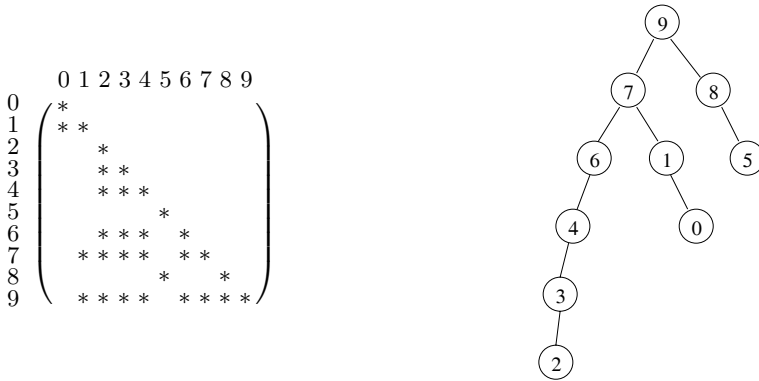


Abb. 7.20. Matrix L mit Superknoten $I(0) = \{0\}$, $I(1) = \{1\}$, $I(2) = \{2, 3, 4\}$, $I(5) = \{5\}$, $I(6) = \{6, 7\}$, $I(8) = \{8, 9\}$ und zugehörigem Eliminationsbaum.

$q - 1$ die letzte Spalte von I . Zusätzlich zu den oben eingeführten Prozeduren `cmod()` und `cdiv()` verwenden wir eine weitere Prozedur

```

smod( $j, J$ ) =
     $r = \min\{j - 1, \text{last}(J)\}$ ;
    for  $k = \text{first}(J), \dots, r$ 
        cmod( $j, k$ );

```

die die Modifikationen einer Spalte j mit allen Spalten eines Superknoten J beschreibt. Dabei sind zwei Fälle zu unterscheiden: Wenn die Spalte j zum Superknoten J gehört, wird j mit den Spalten von J modifiziert, die links von j stehen. Wenn j nicht zu J gehört, wird j mit allen Spalten von J modifiziert. Mit Hilfe von `smod()` kann die Berechnung der Spalten von L durch das folgende Berechnungsschema beschrieben werden, das als **Right-Looking-Superknoten-Algorithmus** bezeichnet wird:

```

supernode_cholesky =
    for each supernode  $J$  do from left to right {
        cdiv( $\text{first}(J)$ );
        for  $j = \text{first}(J) + 1, \dots, \text{last}(J)$  {
            smod( $j, J$ );
            cdiv( $j$ );
        }
        for  $k \in \text{Struct}(L_{*(\text{last}(J))})$ 
            smod( $k, J$ );
    }

```

Dieses Berechnungsschema berechnet die Spalten von L weiterhin von links nach rechts. Die Berechnung der Spalten eines Superknotens wird aber zu-

sammengefasst. Da ein Right-Looking-Schema verwendet wird, kann die Berechnung der ersten Spalte eines Superknotens J durch eine `cdiv()`-Operation abgeschlossen werden, wenn alle Superknoten links von J bereits berechnet sind. Das Berechnungsschema berechnet die einzelnen Spalten eines Superknotens durch ein Left-Looking-Schema, d.h. nachdem alle Superknoten links von J berechnet und nachdem wegen des Right-Looking-Schemas zwischen Superknoten alle Spalten von J mit allen Superknoten links von J modifiziert worden sind, wird jede Spalte $j \in J$ dadurch berechnet, dass sie zuerst mit allen Spalten von J , die links von j liegen, modifiziert wird. Dies wird durch den Aufruf `smod( $j, J$ )` erreicht. Danach wird die Berechnung von j durch einen `cdiv( $j$ )`-Aufruf abgeschlossen. Nach Berechnung aller Spalten eines Superknotens J werden alle Spalten k rechts von J , die von den Spalten von J abhängen, durch den Aufruf `smod( $k, J$ )` mit allen Spalten von J modifiziert. Anstatt des beschriebenen Left-Looking-Schemas innerhalb der Superknoten könnte auch ein Right-Looking-Schema verwendet werden.

Der Vorteil der auf Superknoten basierten Algorithmen liegt darin, dass sie einen *hohen Lokalitätsgrad* aufweisen, so dass sie für Speicherhierarchien besonders effizient ausgeführt werden können. Wegen der Verwendung des Right-Looking-Schemas zwischen Superknoten sind beim Start der Bearbeitung eines Superknotens J alle Spalten von J bereits mit allen Spalten links von J modifiziert. Für die Vervollständigung der Berechnung der Spalten von J wird wegen der Verwendung des Left-Looking-Algorithmus innerhalb der Superknoten ausschließlich auf die Spalten von J zugegriffen. Dies ist ein Unterschied zum normalen Right-Looking-Algorithmus, bei dem nach Berechnung einer Spalte von J auch Spalten außerhalb von J modifiziert würden. Während der Berechnung der Spalten von J würden also unter Umständen wesentlich mehr Zielspalten adressiert, die dann evtl. nicht mehr in den Cache passen. Nach Berechnung der Spalten von J werden die von $j \in J$ abhängigen Spalten *nacheinander* mit den Spalten von J modifiziert. Dabei wird jede abhängige Spalte rechts von J mit *allen* Spalten von J modifiziert, bevor die Modifikation der nächsten abhängigen Spalten rechts von J begonnen wird. Neben den Spalten von J muss also nur jeweils eine Spalte im Cache aufgehoben werden. Der Unterschied zum normalen Left-Looking-Algorithmus besteht darin, dass die abhängigen Spalten rechts von J nur mit den Spalten von J modifiziert werden anstatt mit weiteren Spalten links von J , d.h. im Vergleich zum normalen Left-Looking-Algorithmus müssen weniger Spalten im Cache aufgehoben werden.

7.4.2 Abspeicherungsschemata für dünnbesetzte Matrizen

Da die meisten Einträge von dünnbesetzten Matrizen Null sind, werden für dünnbesetzte Matrizen Abspeicherungsschemata benutzt, die nur die Nicht-nullelemente abspeichern. Dabei muss die implizite Zuordnung von Elementen an Zeilen und Spalten, die bei der Verwendung von zweidimensionalen

Feldern gegeben ist, aufgehoben werden. Neben den Werten der Nichtnullelemente muss daher auch deren Zeilen- und Spaltenposition *explizit* abgespeichert werden, d.h. neben dem Speicherplatz zur Ablage der eigentlichen Elemente erfordern diese Abspeicherungsschemata zusätzlichen Overhead-Speicherplatz, ihr Einsatz ist daher nur für dünnbesetzte Matrizen sinnvoll.

Wir beschreiben im Folgenden ein Abspeicherungsschema, das insbesondere zur Ablage einer unteren Dreiecksmatrix L geeignet ist, wie sie bei der in Abschnitt 7.4.1 beschriebenen Cholesky-Zerlegung entsteht [56, 102]. Neben der Besetzungsstruktur der abzuspeichernden Matrix nutzt das Abspeicherungsschema auch die Tatsache aus, dass die Zeilen eines Superknotens eine ähnliche Besetzungsstruktur haben. Wir beschreiben zuerst eine vereinfachte Variante des Abspeicherungsschemas ohne Ausnutzung der Superknoten und erläutern dann die Modifikation, die zur Ausnutzung der Superknoten erforderlich ist. Für die Beschreibung nehmen wir an, dass eine Matrix $L \in \mathbb{R}^{n \times n}$ abgelegt werden soll und dass nz die Anzahl der Nichtnulleinträge (engl. *nonzeros*) von L ist. Zur Ablage von L werden zwei Felder **Nonzero** und **Row** der Länge nz und drei Felder **StartColumn**, **StartRow** und **Supernode** der Länge n verwendet. Das Feld **Nonzero** enthält die Werte der Nichtnulleinträge, wobei diese spaltenweise von links nach rechts geordnet eingetragen sind. Das Feld **StartColumn** gibt an, welchen Spalten die in **Nonzero** abgespeicherten Elemente angehören: **StartColumn**[j] gibt an, an welcher Position in Feld **Nonzero** das erste Nichtnullelement von Spalte j abgespeichert ist, d.h. **Nonzero**[**StartColumn**[j]] enthält dieses Element. Wegen der spaltenweisen Ablage enthält **Nonzero**[**StartColumn**[$j + 1$] - 1] das letzte Nichtnullelement von Spalte j . Die restlichen Nichtnullelemente von Spalte j sind zwischen diesen Positionen abgespeichert. Insgesamt werden **StartColumn**[$j + 1$] - **StartColumn**[j] Nichtnulleinträge von Spalte j in **Nonzero** abgespeichert.

Das Feld **Row** enthält die Zeilenindizes der korrespondierenden Elemente in **Nonzero**. Ohne Berücksichtigung der Superknoten enthält **Row**[r] für $r = 0, \dots, nz - 1$ den Zeilenindex des Nichtnulleintrages in **Nonzero**[r]. Entsprechend der Abspeicherung in **Nonzero** sind die Zeilenindizes der Nichtnulleinträge einer Spalte j an aufeinanderfolgenden Positionen in **Row** abgespeichert. Die ähnliche Besetzungsstruktur der Spalten von Superknoten kann dazu verwendet werden, die Anzahl der in **Row** abgespeicherten Zeilenindizes zu reduzieren. Da für die Spalten eines Superknotens unterhalb des dichtbesetzten Dreiecksblocks die gleichen Zeilenindizes besetzt sind, brauchen diese eigentlich nur einmal abgespeichert zu werden. Um dies zu realisieren, wird zusätzlich das Feld **StartRow** verwendet. Der Eintrag **StartRow**[j] enthält dazu den Index von **Row**, in dem der Zeilenindex des ersten Nichtnulleintrages von Spalte j abgelegt ist, d.h. **Row**[**StartRow**[j]] = j , da l_{jj} der erste Nichtnulleintrag von Spalte j ist. Das zusätzliche Feld **StartRow** kann dazu verwendet werden, die strikte Korrespondenz der Einträge von **Nonzero** und **Row** zugunsten einer kompakteren Abspeicherung aufzugeben. Wenn j die er-

ste Spalte eines Superknotens $I(j) = \{j, j+1, \dots, j+k-1\}$ ist, so hat Spalte $j+l$ für $1 \leq l \leq k$ die gleiche Besetzungsstruktur wie Spalte j ab Zeile $j+l$, d.h. $\text{Row}[\text{StartRow}[j]+l]$ enthält den Zeilenindex des ersten Eintrags von Spalte $j+l$. Da dies der Diagonaleintrag ist, ist $\text{Row}[\text{StartRow}[j]+l] = j+l$. Die folgenden Einträge enthalten die Zeilenindizes der restlichen Einträge von Spalte $j+l$. Die Zeilenindizes von Spalte $j+l$ befinden sich also in $\text{Row}[\text{StartRow}[j]+l], \dots, \text{Row}[\text{StartRow}[j] + \text{StartColumn}[j+1] - \text{StartColumn}[j]]$. Wir können daher $\text{StartRow}[j+l] = \text{StartRow}[j] + l$ setzen und brauchen in Row nur die Zeilenindizes der ersten Spalten jedes Superknotens abzulegen. Auf die Zeilenindizes einer Spalte j kann damit durch

$$\text{Struct}(L_{*j}) =$$

$$\{\text{Row}[\text{StartRow}[j] + i] \mid 1 \leq i \leq \text{StartColumn}[j+1] - \text{StartColumn}[j]\}.$$

zugegriffen werden. Abbildung 7.21 zeigt eine Illustration des Abspeicherungsschemas. Zusätzlich zu den abgebildeten Feldern gibt es ein Feld **Supernode**, das zur Verwaltung der Superknoten verwendet wird. Wenn eine Spalte j die erste Spalte eines Superknotens J ist, wird die Anzahl der Spalten von J in $\text{Supernode}[j]$ abgespeichert.

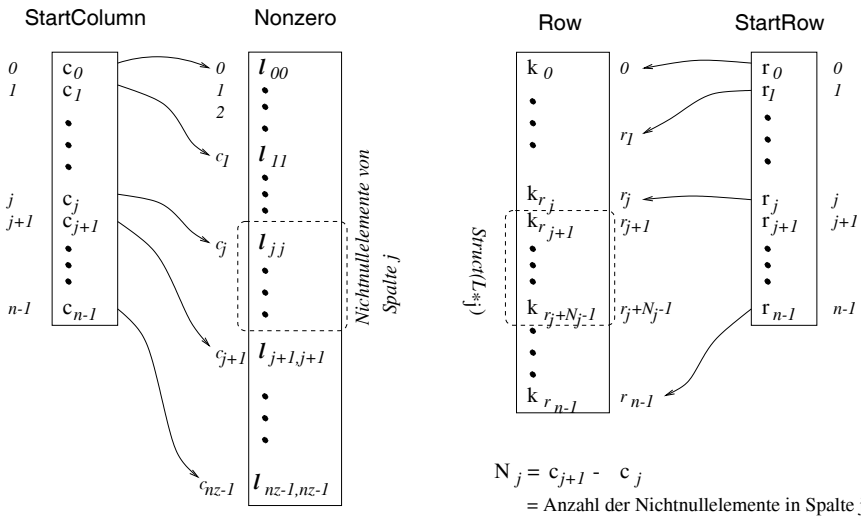


Abb. 7.21. Komprimiertes Abspeicherungsschema für eine dünnbesetzte untere Dreiecksmatrix L . Nonzero enthält die Nichtnullelemente von Matrix L , die Positionen der jeweils ersten Elemente einer Spalte sind in StartColumn abgelegt. Zusätzlich enthält das Feld Row die Zeilenindizes der Elemente in Nonzero, die Zeilenanfänge sind in StartRow abgelegt. Bei Vorhandensein von Superknoten ist Row kompakter mit überlappenden Zeilenbereichen abgespeichert.

7.4.3 Implementierung für gemeinsamen Adressraum

Wir untersuchen im Folgenden die parallele Implementierung der verschiedenen Varianten, die wir in Abschnitt 7.4.1 vorgestellt haben [135]. Dafür betrachten wir einen Parallelrechner mit einem gemeinsamen Adressraum.

Für dichtbesetzte Matrizen können die Einträge einer Spalte j erst dann errechnet werden, wenn alle Spalten links von j bereits berechnet worden sind. Daher müssen die Spalten in einer strikt sequentiellen Reihenfolge von links nach rechts berechnet werden. Für dünnbesetzte Matrizen kann durch das Fehlen von Einträgen der Fall auftreten, dass von dieser sequentiellen Reihenfolge abgewichen werden kann bzw. dass Spalten parallel zueinander berechnet werden können, wenn keine Abhängigkeiten zwischen ihnen bestehen. Daher ist es sinnvoll, die Abhängigkeiten zwischen den Spalten zu veranschaulichen. Um dies zu tun, definiert man für jede Spalte j

$$\text{parent}(j) = \min\{i \mid i \in \text{Struct}(L_{*j})\} ,$$

d.h. $\text{parent}(j)$ ist der Zeilenindex des ersten Elements von Spalte j , das nicht auf der Diagonalen steht. Dabei nehmen wir $\text{Struct}(L_{*j}) \neq \emptyset$ an. Für $\text{Struct}(L_{*j}) = \emptyset$ setzen wir $\text{parent}(j) = j$. Die Bedeutung von $\text{parent}(j)$ liegt darin, dass $\text{parent}(j)$ die erste Spalte $i > j$ angibt, die von j abhängt. Eine dazwischenliegende Spalte l , $j < l < i$, hängt nicht von j ab, da wegen $j \notin \text{Struct}(L_{l*})$ keine Operation $\text{cmod}(l, j)$ durchgeführt werden muss. Aufbauend auf der Definition von $\text{parent}(j)$ definieren wir

$$\text{children}(i) = \{j < i \mid \text{parent}(j) = i\} ,$$

d.h. $\text{children}(i)$ enthält alle Spalten j , deren erstes Element, das nicht auf der Diagonalen steht, in Zeile i steht. Aufbauend auf der Definition von $\text{parent}()$ können wir einen gerichteten Graphen $G = (V, E)$ mit $V = \{0, \dots, n-1\}$ definieren, der für jede Spalte einen Knoten enthält. Es gibt eine Kante $(i, j) \in E$ genau dann, wenn $i = \text{parent}(j)$ und $i \neq j$, d.h. $(i, j) \in E$ bedeutet, dass j zur Berechnung von i , aber nicht zur Berechnung der Spalten l mit $j < l < i$ gebraucht wird. Es lässt sich zeigen, dass der so definierte Graph für *irreduzible* Matrizen ein *Baum* ist, den man als **Eliminationsbaum** bezeichnet [104, 110]. (Eine Matrix heißt *reduzibel*, falls sie durch Permutation, also Zeilen- und Spaltenvertauschung, auf Block-Dreiecksgestalt gebracht werden kann. Kann eine solche Gestalt nicht erzeugt werden, so heißt eine Matrix *irreduzibel*.) Die Bedeutung des Eliminationsbaumes liegt darin, dass er die Datenabhängigkeit zwischen den Spalten genau aufzeigt: zur Berechnung einer Spalte j werden alle Spalten gebraucht, die im Teilbaum des Eliminationsbaumes mit Wurzel j auftreten. Wir bezeichnen den Unterbaum mit Wurzel j im Folgenden als $G[j]$. Abbildung 7.22 zeigt ein Beispiel eines Eliminationsbaumes.

Die Bedeutung des Eliminationsbaumes für die parallele Implementierung der Cholesky-Faktorisierung liegt darin, dass er die Reihenfolge der Berechnung der Spalten festlegt und damit auch angibt, welche Spalten unabhängig

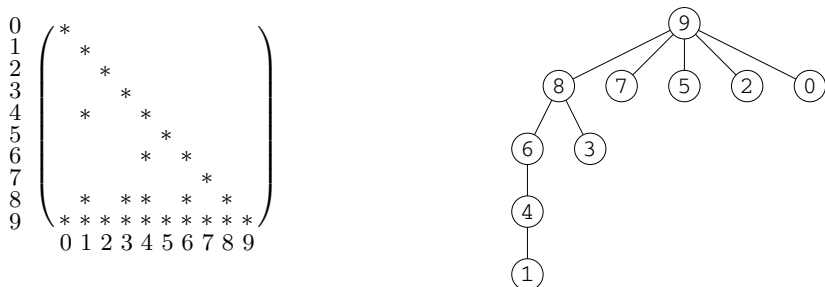


Abb. 7.22. Dünnbesetzte Matrix mit zugehörigem Eliminationsbaum.

voneinander sind und daher parallel zueinander berechnet werden können. Spalte j hängt nur von den Spalten in $G[j]$ ab. Daher können die Spalten i und j genau dann parallel zueinander berechnet werden, wenn $G[i]$ und $G[j]$ disjunkte Teilbäume des Eliminationsbaumes sind. Insbesondere können alle Blätter des Eliminationsbaumes parallel zueinander berechnet werden, je nach Matrix muss die Berechnung also nicht notwendigerweise bei Spalte 0 beginnen. Das Aussehen des Eliminationsbaumes bestimmt also wesentlich den verfügbaren Grad an Parallelität, der durch die Nulleinträge der Matrix zusätzlich gegeben ist. Als grober Anhaltspunkt kann gelten, dass der verfügbare Grad an Parallelität sich zur Tiefe des Eliminationsbaumes umgekehrt proportional verhält, d.h. ein tiefer Eliminationsbaum bietet weniger Parallelität als ein weniger tiefer Eliminationsbaum.

Left-Looking-Algorithmus

Für die parallele Ausführung des Left-Looking-Algorithmus werden n Spalten-tasks $\text{Tcol}(0), \dots, \text{Tcol}(n-1)$ verwendet, wobei $\text{Tcol}(j)$ für die Berechnung von Spalte j zuständig ist, d.h. $\text{Tcol}(j)$ beinhaltet die Ausführung der Operationen $\text{cmod}(j, k)$ für alle $k \in \text{Struct}(L_{j*})$ und der Operation $\text{cdiv}(j)$. Die Tasks werden in einem Taskpool verwaltet, der *zentral* gehalten wird und auf den jeder Prozessor zugreifen kann. Dies erlaubt eine dynamische Verwaltung der Tasks, d.h. die Prozessoren greifen auf den Taskpool zu, wenn sie ihre aktuelle Task abgearbeitet haben. Diese dynamische Verwaltung hat gegenüber einer statischen Verteilung der Spalten auf die verfügbaren Prozessoren den Vorteil, dass die unterschiedliche Ausführungszeit der Tasks, die durch die unterschiedliche Besetzungsstruktur der Spalten verursacht wird, berücksichtigt wird.

Da die verschiedenen Prozessoren konkurrierend auf den Taskpool zugreifen können, müssen die Zugriffe zur Vermeidung zeitkritischer Abläufe mit einem Lock-Mechanismus voreinander geschützt werden, vgl. Abschnitt 3.7.1, so dass sichergestellt ist, dass zu jedem Zeitpunkt jeweils nur ein Prozessor

auf den Taskpool zugreift. Diese Sequentialisierung des Zugriffs birgt für eine große Anzahl von Prozessoren die Gefahr, dass der Zugriff auf den Taskpool zum Flaschenhals wird. Diese Gefahr besteht insbesondere dann, wenn die Ausführungszeit der einzelnen Task gering ist, so dass die Prozessoren häufig auf den Taskpool zugreifen müssen. Eine Alternative besteht in der Verwendung einer dezentralen Task-Schlange mit *task stealing*.

Für das Einfügen der Spaltentasks in den Taskpool gibt es beim Left-Looking-Algorithmus mehrere Möglichkeiten. Die einfachste Möglichkeit besteht darin, eine Spaltentask $\text{Tcol}(j)$ erst dann in den Taskpool einzufügen, wenn alle Spaltentasks $\text{Tcol}(k)$ mit $k \in \text{Struct}(L_{j*})$ vollständig ausgeführt sind. Damit kann ein Prozessor, wenn er eine Task aus dem Taskpool entnimmt, diese vollständig ausführen, ohne auf die Beendigung anderer Tasks zu warten. Bei dieser Variante kann der Taskpool mit den Blättern des Eliminationsbaumes initialisiert werden. Wegen der rein sequentiellen Abarbeitung von voneinander abhängigen Tasks ist der Parallelitätsgrad zu jedem Zeitpunkt durch die Anzahl der unabhängigen Knoten des Eliminationsbaumes beschränkt.

Der Parallelitätsgrad kann wesentlich erhöht werden, wenn die Berechnung der Spaltentasks $\text{Tcol}(j)$ bereits gestartet werden kann, *bevor* alle Spaltentasks $\text{Tcol}(k)$ mit $k \in \text{Struct}(L_{j*})$ ausgeführt sind. In diesem Fall können zwar nicht alle Operationen $\text{cmod}(j, k)$ sofort ausgeführt werden, der ausführende Prozessor kann aber bereits mit der Ausführung beginnen, indem er die Operation $\text{cmod}(j, k)$ für alle $k \in \text{Struct}(L_{j*})$ ausführt, deren zugehörige Spaltentask bereits ausgeführt ist. Während der Ausführung von $\text{Tcol}(j)$ muss ein Prozessor evtl. jedoch auf die Fertigstellung weiterer Tasks warten. Zur Organisation des Wartens kann man für jede Spalte j eine Datenstruktur S_j halten, die alle Spalten $k \in \text{Struct}(L_{j*})$ enthält, für die die Operationen $\text{cmod}(j, k)$ bereits ausgeführt werden kann. Wenn ein Prozessor die Ausführung einer Task $\text{Tcol}(k)$ beendet, indem er $\text{cdiv}(k)$ ausführt, schreibt er die Nummer k der fertig berechneten Spalte in die Datenstruktur jeder Spalte j mit $j \in \text{Struct}(L_{*k})$. Da dabei verschiedene Prozessoren zum gleichen Zeitpunkt auf die Datenstruktur S_j der gleichen Spalte j zugreifen können, müssen die Datenstrukturen wieder mit einem Lock-Mechanismus geschützt werden. Zum Ausführen einer Spaltentask $\text{Tcol}(j)$ entnimmt der ausführende Prozessor aus S_j nacheinander die darin abgelegten Spaltennummern k und führt die Operation $\text{cmod}(j, k)$ aus. Wenn die Datenstruktur leer ist, muss der Prozessor warten, bis ein anderer Prozessor eine neue Spaltennummer in S_j ablegt. Wenn $|\text{Struct}(L_{j*})|$ Spaltennummern entnommen wurden, kann $\text{Tcol}(j)$ durch die Operation $\text{cdiv}(j)$ beendet werden.

Abbildung 7.23 zeigt die zugehörige Implementierung. Der zentrale Taskpool ist implizit über eine parallele Schleife realisiert, wobei die verwendete Multiplikationsoperation MPADD sicherstellt, dass jeder Prozessor einen anderen Wert der Zählvariablen c in j erhält, auch wenn mehrere Prozessoren gleichzeitig die Schleife betreten. Der Rumpf der **while**-Schleife für einen

```

parallel_left_cholesky =
    c = 0;
    while ((j = MPADD(c, 1)) < n) {
        for (i = 0; i < |Struct(Lj*)|; i++) {
            while (Sj empty) wait();
            k = pop(Sj);
            cmod(j, k);
        }
        cdiv(j);
        for (i ∈ Struct(Lj*)) : push(j, Si);
    }

```

Abb. 7.23. Paralleler Left-Looking-Algorithmus mit Verwendung der gemeinsamen Datenstrukturen $S_1, \dots, S_n$, die das Warten der Spaltentasks organisieren. Die Funktion `parallel_left_cholesky()` wird von jedem der beteiligten Prozessoren ausgeführt.

Wert j entspricht der Spaltentask $\text{Tcol}(j)$. Die Datenstrukturen $S_1, \dots, S_n$ sind jeweils als Stack realisiert, aus denen durch `pop(Sj)` Werte entnommen und auf die durch `push(i, Sj)` Werte abgelegt werden können.

Der Taskpool kann bei der gerade beschriebenen Variante mit allen Spaltentask so initialisiert werden, dass die Spalten von links nach rechts bearbeitet werden. Eine Alternative besteht darin, für die Initialisierung den Eliminationsbaum zu verwenden, d.h. die Knoten des Eliminationsbaumes werden so in den Taskpool eingefügt, dass die Kindknoten vor ihren Elternknoten stehen. Damit greifen die Prozessoren zuerst auf die Blätter des Eliminationsbaumes zu, so dass die zugehörigen Spaltentasks ohne Warten ausgeführt werden können.

Right-Looking-Algorithmus

Die parallele Implementierung des Right-Looking-Algorithmus basiert ebenfalls auf der Benutzung von Spaltentasks, die jedoch anders als beim Left-Looking-Algorithmus definiert sind. Eine für den Right-Looking-Algorithmus verwendete Spaltentask $\text{Tcol}(j)$ beinhaltet die Ausführung der Operation `cdiv(j)` und der Operationen `cmod(k, j)` für alle $k \in \text{Struct}(L_{*j})$, d.h. $\text{Tcol}(j)$ beinhaltet die Fertigstellung von Spalte j und die Modifikation aller Spalten $k > j$, die von Spalte j abhängen. Der Taskpool wird mit den Spaltentasks initialisiert, die den Blättern des Eliminationsbaumes entsprechen. Eine Task $\text{Tcol}(j)$, deren zugehörige Spalte nicht Blatt des Eliminationsbaumes ist, wird in den Taskpool eingefügt, sobald alle Operation `cmod(j, k)` für $k \in \text{Struct}(L_{j*})$ ausgeführt sind, d.h. sobald die Operation `cdiv(j)` ausgeführt werden kann. Zur Ausführung der `cmod(k, j)`-Operationen von $\text{Tcol}(j)$ müssen im Gegensatz zur parallelen Implementierung des Left-Looking-Algorithmus die Zielspalten k der Operation durch einen Lock-Mechanismus geschützt werden, da zum gleichen Zeitpunkt verschiedene Prozessoren `cmod()`-Operationen mit derselben Zielspalte ausführen können.

```

parallel_right_cholesky =
    c = 0;
    initialize_task_pool(TP);
    while ((j = MPADD(c, 1)) < n) {
        while (!filled_pool(TP, j)) wait();
        get_column(j);
        cdiv(j);
        for (k ∈ Struct(L*j)) {
            lock(k); cmod(k, j); unlock(k);
            if (MPADD(ck, 1) + 1 == |Struct(Lk*)|) add_column(k);
        }
    }

```

Abb. 7.24. Paralleler Right-Looking-Algorithmus. Die Spaltentasks werden in einem Taskpool TP gehalten, in den sie durch `add_column()` eingetragen und durch `get_column()` entnommen werden. Die Funktion `initialize_task_pool()` legt den Taskpool an und fügt die den Blättern des Eliminationsbaums entsprechenden Tasks ein. Die Bedingung der `while`-Schleife dient der Zuordnung von Indices j an Prozessoren, deren korrespondierende Task sie aus dem Taskpool entnehmen, sobald sie durch den Aufruf von `filled_pool(TP, j)` feststellen, dass dieser Eintrag besetzt ist.

Abbildung 7.24 zeigt eine parallele Implementierung des Right-Looking-Algorithmus. Das Einfügen der Task in den Taskpool kann dadurch kontrolliert werden, dass für jede Spalte j ein Zähler c_j gehalten wird, der mit 0 initialisiert wird und dessen Wert nach jeder Ausführung einer `cmod(j, *)`-Operation inkrementiert wird. Da diese Inkrementierung durch den Prozessor, der die `cmod(j, *)`-Operation ausführt, vorgenommen werden muss und da mehrere Prozessoren zur gleichen Zeit `cmod()`-Operationen mit der gleichen Zielspalte ausführen können, muss auch der Zugriff auf den Zähler c_j durch einen Lock-Mechanismus geschützt werden. Eine Task `Tcol(j)` wird dann in den Taskpool eingefügt, wenn der Zähler c_j den Wert $|Struct(L_{j*})|$ erreicht hat.

Der Unterschied zwischen der gerade beschriebenen parallelen Implementierung des Right-Looking-Algorithmus und der oben beschriebenen parallelen Implementierung des Left-Looking-Algorithmus liegt in der Ausführungsreihenfolge der `cmod()`-Operationen und darin, dass unterschiedliche Prozessoren diese Operationen ausführen. Bei der beschriebenen Left-Looking-Implementierung wird die Ausführung der Operation `cmod(j, k)` durch den Prozessor initiiert, der `Tcol(k)` berechnet. Diese Initiierung geschieht dadurch, dass dieser Prozessor den Spaltenindex k nach Beendigung von `Tcol(k)` in die Datenstruktur S_j von Spalte j einträgt. Die eigentliche Operation `cmod(j, k)` wird aber von dem Prozessor durchgeführt, der die Spaltentask `Tcol(j)` ausführt. Die Durchführung der Operation braucht dabei der Initiierung nicht unmittelbar zu folgen. Bei der Right-Looking-Implementierung wird die Operation `cmod(j, k)` von dem Prozessor, der `Tcol(k)` ausführt, nicht

nur initiiert, sondern direkt ausgeführt. Ein Warten auf andere Prozessoren tritt bei der Right-Looking-Implementierung nicht auf.

Superknoten-Algorithmus

Die parallele Implementierung des Superknoten-Algorithmus basiert auf der Verwendung von **fundamentalen Superknoten**. Ein Superknoten $I(p) = \{p, p+1, \dots, p+q-1\}$ wird als fundamental bezeichnet, wenn für jedes i mit $0 \leq i \leq q-2$ gilt, dass $\text{children}(p+i+1) = \{p+i\}$ ist, d.h. Knoten $p+i$ ist das einzige Kind von Knoten $p+i+1$ im Eliminationsbaum. In Abbildung 7.20 ist $I(2) = \{2, 3, 4\}$ ein fundamentaler Superknoten während $I(6) = \{6, 7\}$ und $I(8) = \{8, 9\}$ nicht fundamental sind. Der Vorteil eines fundamentalen Superknotens liegt darin, dass *alle* Spalten eines solchen Superknotens berechnet werden können, sobald die *erste* Spalte berechnet werden kann. Für die Berechnung der folgenden Spalten des Superknotens muss also nicht auf die Berechnung von Spalten außerhalb des Superknotens gewartet werden. Ein aus einer einzelnen Spalte bestehender Superknoten kann als fundamental angesehen werden. Wir gehen im Folgenden davon aus, dass alle vorliegenden Superknoten fundamental sind, was durch Aufspalten nicht-fundamentaler Superknoten erreicht werden kann.

Die parallele Implementierung des Superknoten-Algorithmus basiert auf Superknotentasks. Ein Superknoten J mit $0 \leq J \leq N$ wird dabei durch eine Superknotentask $\text{Tsup}(J)$ berechnet, wobei $\text{Tsup}(J)$ die Ausführung der Operationen $\text{smod}(j, J)$ und $\text{cdiv}(j)$ für alle $j \in J$ von links nach rechts und die Ausführung der Operationen $\text{smod}(k, J)$ für alle $k \in \text{Struct}(L_{*(\text{last}(J))})$ beinhaltet. Dabei bezeichnet $\text{last}(J)$ die letzte Spalte von J . $\text{Tsup}(J)$ vervollständigt also die Berechnung der Spalten von J von links nach rechts und führt die Modifikationen von Spalte $k > \text{last}(J)$, die von Spalten von J abhängen, entsprechend dem in Abschnitt 7.4.1 beschriebenen Right-Looking-Superknoten-Algorithmus durch. Da fundamentale Superknoten verwendet werden, treten für den Prozessor, der $\text{Tsup}(J)$ ausführt, keine Wartezeiten auf. Die ausführbaren Superknotentasks werden in einem zentralen Taskpool gehalten. Der Taskpool wird mit den Superknoten initialisiert, deren erste Spalte ein Blatt des Eliminationsbaumes ist. Eine Task $\text{Tsup}(K)$ wird dann in den Taskpool eingefügt, wenn sie ausführbar ist, d.h. nachdem alle Modifikationen $\text{smod}(k, J)$ auf den Spalten $k \in K$ durch Tasks $\text{Tsup}(J)$ mit $J < K$ durchgeführt worden sind. Um das Eintreffen dieses Ereignisses zu überprüfen, halten wir wie bei der parallelen Implementierung des Right-Looking-Algorithmus für jede Spalte j einen Zähler c_j , der mit 0 initialisiert ist, und der bei jeder Modifikation von j inkrementiert wird. Eine Superknotentask $\text{Tsup}(J)$ wird ausführbar, sobald der Zähler c_j der ersten Spalte $j \in J$ den Wert $|\text{Struct}(L_{j*})|$ erreicht. Auch hier muss sowohl die Inkrementierung der Zähler als auch die Manipulation der Zielspalten mit Hilfe eines Lock-Mechanismus geschützt werden. Abbildung 7.25 skizziert die resultierende Implementierung.

```

parallel_supernode_cholesky =
    c = 0;
    initialize_task_pool(TP);
    while ((J = MPADD(c, 1)) < N) {
        while (!filled_pool(TP, J)) wait();
        get_supernode(J);
        cdiv(first(J));
        for (j = first(J) + 1; j ≤ last(J); j++) {
            smod(j, J);
            cdiv(j);
        }
        for (k ∈ Struct(L*(last(J)))) {
            lock(k); smod(k, J); unlock(k);
            MPADD(ck, 1);
            if ((k == first(K)) && (ck + 1 == |Struct(Lk*)|))
                add_supernode(K);
        }
    }

```

Abb. 7.25. Paralleler Superknoten-Algorithmus. Die Verwaltung der N Superknoten erfolgt in einem Taskpool analog zu Abbildung 7.24.

Diskussion der Varianten

Bei einem Vergleich der beschriebenen Varianten müssen mehrere Punkte beachtet werden. Alle Varianten verwenden für die Realisierung einer dynamischen Verteilung der Spalten auf die Prozessoren einen zentralen Taskpool, der mit einem Lock-Mechanismus vor konkurrierenden Zugriffen geschützt werden muss. Dabei werden bei dem Superknoten-Algorithmus in der Regel weniger Tasks als bei den anderen Varianten verwendet, da benachbarte Spalten oft zu (fundamentalen) Superknoten zusammengefasst werden. Die Tasks haben daher eine längere Ausführungszeit und die ausführenden Prozessoren greifen weniger häufig auf den Taskpool zu. Damit ist die Wahrscheinlichkeit für Zugriffskonflikte, die mit der Anzahl der beteiligten Prozessoren steigt, geringer als bei den anderen Varianten. Bei der Right-Looking- und der Superknoten-Implementierung werden die verwendeten Tasks erst dann in den Taskpool eingefügt, wenn sie ohne Warten vollständig ausgeführt werden können. Damit kann der Fall auftreten, dass Prozessoren auf einen leeren Taskpool zugreifen und warten müssen, bis durch Abarbeitung einer anderen Task eine neue Task ausführbar wird. Da beim Left-Looking-Algorithmus der Taskpool mit allen Spaltentasks initialisiert wird, kann ein Zugriff auf einen leeren Taskpool nur am Ende der gesamten Berechnung auftreten. Während der Abarbeitung einer Task $Tcol(j)$ können jedoch Wartezeiten auftreten, wenn andere Tasks, von denen $Tcol(j)$ abhängt, noch nicht vollständig ausgeführt worden sind.

Neben dem Taskpool erfordern die verschiedenen Varianten zusätzlichen Aufwand für die Taskverwaltung. Beim Left-Looking-Algorithmus ist dies

die Datenstruktur S_j , die für jede Spalte j gehalten werden muss. Beim Right-Looking- und Superknoten-Algorithmus muss für jede Spalte ein Zähler gehalten werden. In beiden Fällen muss ein Lock-Mechanismus verwendet werden, um mögliche konkurrierende Zugriffe auf die gemeinsamen Daten zu koordinieren. Die resultierende Sequentialisierung der Zugriffe kann dazu führen, dass ein Prozessor warten muss, bis ein oder mehrere andere Prozessoren ihre Zugriffe beendet haben. Die Wahrscheinlichkeit für Zugriffskonflikte steigt wieder mit der Anzahl der ausführenden Prozessoren. Da das Einfügen von Spalten in die Datenstruktur S_j in der Regel aufwendiger ist als das Inkrementieren eines Zählers und da die eingefügten Spalten auch wieder entnommen werden müssen, wobei auch hierfür ein Lock-Mechanismus verwendet werden muss, ist die Taskverwaltung für die Left-Looking-Variante aufwendiger als für die anderen beiden Varianten. Da bei gleicher Zugriffshäufigkeit die Wahrscheinlichkeit für Zugriffskonflikte mit der Dauer der einzelnen Zugriffe ansteigt, sind bei der Left-Looking-Variante mit steigender Anzahl der beteiligten Prozessoren mehr Zugriffskonflikte als für die anderen beiden Varianten zu erwarten.

Beim Left-Looking-Algorithmus wird die Berechnung einer Spalte vollständig von *einem* Prozessor vorgenommen, für die Durchführung der `cmod()`-Operationen und der `cdiv()`-Operationen muss also *kein* Lock-Mechanismus verwendet werden. Bei den anderen beiden Varianten muss dagegen ein Lock-Mechanismus verwendet werden, da zu einem Zeitpunkt mehrere Prozessoren dieselbe Zielspalte ansprechen können. Die Wahrscheinlichkeit, dass dabei Zugriffskonflikte auftreten können, steigt in der Regel mit der Anzahl der beteiligten Prozessoren und sinkt mit der Anzahl der zu berechnenden Spalten.

Somit hat jede der betrachteten Varianten Vor- und Nachteile. Welche der beschriebenen Faktoren letztendlich überwiegt, hängt stark von den Gegebenheiten des verwendeten Parallelrechners wie z.B. dem Aufwand des Lock-Mechanismus und der Implementierung der verwendeten Datenstrukturen ab. In [135] wurden die beschriebenen Varianten für die SB-PRAM, einem Parallelrechner mit gemeinsamem globalen Speicher, miteinander verglichen. Da dieser Rechner Multipräfixoperationen auf Integerwerten zur Verfügung stellt, können die Zugriffskonflikte auf den Taskpool und für die zusätzliche Taskverwaltung vermieden werden. Da die Spalten aber Floating-Point-Werte beinhalten, muss bei der Right-Looking- und der Superknoten-Variante ein Lock-Mechanismus verwendet werden, um die `cmod()`-Operationen zu koordinieren. Da dies bei der Left-Looking-Variante nicht erforderlich ist, erreicht diese Variante in diesem Fall auch die besten Ergebnisse. Für andere Parallelrechner könnte der Vergleich aber vollkommen anders aussehen.

7.5 Methode der konjugierten Gradienten

Die in [76] entwickelte Methode der konjugierten Gradienten (engl. *conjugate gradient method*, daher wird der Name **CG-Verfahren** verwendet) ist ein

Lösungsverfahren für lineare Gleichungssysteme $Ax = b$ mit symmetrischer und positiv definiter Koeffizientenmatrix $A \in \mathbb{R}^{n \times n^1}$. Beim CG-Verfahren handelt es sich um ein iteratives Verfahren, in dem die Lösung x^* des zu lösenden Gleichungssystems schrittweise aufgebaut wird. Bei exakter Rechnung liegt die Lösung x^* nach n Schritten vor und das CG-Verfahren wird daher auch als direktes Verfahren bezeichnet. Aufgrund von Rundungsfehlern ist die n -te Approximation aber oft noch zu ungenau, so dass in der Praxis weitere Schritte durchgeführt werden müssen und die Methode war daher zunächst gegenüber anderen direkten Verfahren nicht konkurrenzfähig. In [136] wurde jedoch gezeigt, dass das CG-Verfahren insbesondere für dünnbesetzte Matrizen in *weniger* als n Iterationsschritten zu einer genügend genauen Lösung gelangt und das CG-Verfahren und seine Varianten sind heute weit verbreitete Methode zur Lösung linearer Gleichungssysteme. Angewendet wird das CG-Verfahren meist als *präkonditioniertes CG-Verfahren* (engl. *preconditioned CG method*), bei dem das CG-Verfahren mit einem Präkonditionierer kombiniert wird, siehe [141]. Parallele Realisierungen werden u.a. in [62, 117, 118, 141] vorgestellt, ein Überblick ist in [142] zu finden. In diesem Abschnitt werden wir das CG-Verfahren herleiten und den Basisalgorithmus vorstellen. Unsere Darstellung folgt dabei [17, 61, 155]. Anschließend stellen wir Parallelisierungsmöglichkeiten vor.

7.5.1 Beschreibung der Methode

Der Ausgangspunkt für die Herleitung des CG-Verfahrens ist folgende Aussage über die Äquivalenz der Minimierung einer Funktion und der Lösung eines linearen Gleichungssystems. Die Lösung x^* des linearen Gleichungssystems $Ax = b$, $A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$, entspricht dem Minimum der Funktion $\Phi : M \subset \mathbb{R}^n \rightarrow \mathbb{R}$ mit

$$\Phi(x) = \frac{1}{2} x^T A x - b^T x, \quad (7.47)$$

falls A positiv definit und symmetrisch ist. Diese Aussage ergibt sich durch Differenzieren von $\Phi(x)$, siehe z.B. [61]. Eine einfache Strategie zur Minimierung der Funktion (7.47) ist die Anwendung der **Methode des steilsten Anstieges**, die ausgehend von einem Punkt $x_c \in \mathbb{R}^n$ für die Funktion $\Phi : \mathbb{R}^n \rightarrow \mathbb{R}$ nacheinander die beiden folgenden Schritte ausführt, vgl. [17].

- a) Berechnung des negativen Gradienten $d_c \in \mathbb{R}^n$ in x_c

$$d_c = - \text{grad } \Phi(x_c) = - \left(\frac{\partial}{\partial x_1} \Phi(x_c), \dots, \frac{\partial}{\partial x_n} \Phi(x_c) \right) = b - A x_c$$

¹ Eine Matrix A heißt *positiv definit*, falls $x^T A x > 0$ für alle $x \in \mathbb{R}^n$, $x \neq 0$, gilt, wobei x^T den zu x transponierten Vektor bezeichnet. Eine Matrix A ist *symmetrisch*, falls $A = A^T$ gilt, wobei hier A^T die zu A transponierte Matrix bezeichnet.

b) Bestimmung des Minimums der Funktion Φ auf der Menge

$$\{x_c + td_c \mid t \geq 0\} \cap M,$$

die eine Linie im $\mathbb{R}^n$ beschreibt (Liniensuche). Dazu wird $x_c + td_c$ in (7.47) eingesetzt. Unter Ausnutzung von $d_c = b - Ax_c$ und der Symmetrie von A ergibt sich

$$\Phi(x_c + td_c) = \Phi(x_c) - td_c^T d_c + \frac{1}{2} t^2 d_c^T A d_c. \quad (7.48)$$

Für diese Funktion wird das Minimum $t \in \mathbb{R}$ durch Ableiten nach t bestimmt. Das Minimum $t \in \mathbb{R}$ dieser Funktion wird für

$$t_c = \frac{d_c^T d_c}{d_c^T A d_c} \quad (7.49)$$

angenommen.

Werden die Schritte a) und b) der Methode des steilsten Anstieges nacheinander auf Vektoren $x_k, k = 0, 1, 2, \dots$, mit $x_0 \in \mathbb{R}^n$ und $x_{k+1} = x_k + t_k d_k$ angewandt, so ist die Folge $(\Phi(x_k))_{k=0,1,2,\dots}$ monoton fallend. Dies ergibt sich durch Einsetzen von (7.49) in (7.48). Die Folge konvergiert gegen das gesuchte Minimum, die Konvergenz ist jedoch möglicherweise sehr langsam [61].

Der Grundgedanke des **CG-Verfahrens** ist nun, bei der Berechnung der Folge $\{x_k\}_{k=0,1,2,\dots}$ möglichst Richtungen d_k zu wählen, die entsprechend des gestellten Problems nicht „fast parallel“ laufen, sondern „orthogonal“ zueinander sind. Dazu benötigt man zunächst eine genaue Definition dessen, was „orthogonal bzgl. eines Problems“ heißt. Dies wird durch den Begriff der **konjugierten** bzw. **A-orthogonalen Vektoren** erfasst. Für eine symmetrische, nichtsinguläre Matrix A heißen zwei Vektoren $x, y \in \mathbb{R}^n$ *konjugiert* oder *A-orthogonal*, falls $x^T A y = 0$ gilt. Ist A positiv definit, so sind k paarweise konjugierte Vektoren $d_0, \dots, d_{k-1}$ (mit $d_i \neq 0, i = 0, \dots, k-1$ und $k \leq n$) linear unabhängig [17]. Die gesuchte Lösung x^* von $Ax = b$ kann also als Linearkombination bzgl. einer Basis $d_0, \dots, d_{n-1}$ konjugierter Vektoren dargestellt werden, d.h. als

$$x^* = \sum_{k=0}^{n-1} t_k d_k. \quad (7.50)$$

Durch Anwendung der Orthogonalitätsbedingung der gewählten Basis ergibt sich $d_k^T A x^* = \sum_{k=0}^{n-1} d_k^T A t_k d_k = t_k d_k^T A d_k$, woraus für die Koeffizienten t_k die Darstellung

$$t_k = \frac{d_k^T A x^*}{d_k^T A d_k} = \frac{d_k^T b}{d_k^T A d_k}$$

folgt. Ist eine Basis aus konjugierten Vektoren gewählt, so können die Koeffizienten $t_k, k = 0, \dots, n-1$, also direkt aus der rechten Seite $b \in \mathbb{R}^n$ des gegebenen linearen Gleichungssystems berechnet werden.

Beim CG-Verfahren wird die gesuchte Lösung x^* *rekursiv* berechnet, indem für einen beliebigen Startvektor $x_0 \in \mathbb{R}^n$ nacheinander die Vektoren

$$x_{k+1} = x_k + \alpha_k d_k, \quad k = 1, 2, \dots, \quad \text{mit} \quad (7.51)$$

$$\alpha_k = \frac{-g_k^T d_k}{d_k^T A d_k} \quad \text{und} \quad g_k = A x_k - b \quad (7.52)$$

bestimmt werden. Diese Berechnung beruht auf einer Darstellung

$$x^* = x_0 + \sum_{i=0}^{n-1} \alpha_i d_i, \quad (7.53)$$

der gesuchten Lösung als Summe aus Startvektor x_0 und noch zu bestimmendem Rest $\sum_{i=0}^{n-1} \alpha_i d_i$. Durch (7.51) und (7.52) wird iterativ der noch fehlende Anteil $\sum_{i=0}^{n-1} \alpha_i d_i$ bestimmt, indem nacheinander die Koeffizienten α_i berechnet und $\alpha_i d_i$ hinzugefügt wird. Nach (höchstens) n Schritten wird die Lösung x^* erzeugt, was man folgendermaßen sieht. Die Koeffizienten α_i von (7.53) haben die Gestalt

$$\alpha_i = \frac{d_i^T A(x^* - x_0)}{d_i^T A d_i}.$$

Da nach (7.51) $x_k = x_0 + \sum_{i < k} \alpha_i d_i$ gilt, x_k sich also aus zu d_k konjugierten Vektoren zusammensetzt, gilt $d_k^T A(x_k - x_0) = d_k^T A \sum_{i < k} \alpha_i d_i = 0$ und es folgt

$$\alpha_k = \frac{d_k^T A(x^* - x_k)}{d_k^T A d_k} + \frac{d_k^T A(x_k - x_0)}{d_k^T A d_k} = \frac{d_k^T (b - A x_k)}{d_k^T A d_k}. \quad (7.54)$$

Dies wird aber gerade durch (7.52) berechnet und die gesuchte Lösung ist spätestens durch $x^* = x_n$ gegeben. Werden nicht alle Richtungen benötigt, so gilt bereits für ein k ($k < n$) die Gleichung $g_k = A x_k - b = 0$ und $x^* = x_k$ ist die gesuchte Lösung.

Algorithmus der Methode der konjugierten Gradienten. Bei den üblichen Algorithmen zur Implementierung der Methode der konjugierten Gradienten werden die Richtungen $d_0, \dots, d_{n-1}$, *nicht* vorher gewählt, sondern aus dem jeweils aktuellen Gradienten g_k durch Addition einer Korrektur ermittelt. Der Basisalgorithmus der Methode der konjugierten Gradienten ist in Abbildung 7.26 angegeben. Die Herleitung der Methode der konjugierten Gradienten führt zu

$$\alpha_k = -\frac{g_k^T d_k}{d_k^T A d_k} \quad \text{und} \quad \beta_k = -\frac{g_{k+1}^T A d_k}{d_k^T A d_k}.$$

Aus numerischen Gründen der Stabilität werden die äquivalenten Berechnungen der Größen im Algorithmus gewählt [17, 62].

7.5.2 Parallelisierung des CG-Verfahrens

Der Algorithmus des CG-Verfahrens in Abbildung 7.26 setzt sich in jedem Iterationsschritt aus Basisoperationen zusammen und zwar

Wähle $x_0 \in \mathbb{R}^n$
 Setze $d_0 = -g_0 = b - Ax_0$
 Solange $\|g_k\| > \epsilon$ gilt, berechne für $k = 0, 1, 2, \dots$

- (1) $\omega_k = Ad_k$
- (2) $\alpha_k = \frac{g_k^T g_k}{d_k^T \omega_k}$
- (3) $x_{k+1} = x_k + \alpha_k d_k$
- (4) $g_{k+1} = g_k + \alpha_k \omega_k$
- (5) $\beta_k = \frac{g_{k+1}^T g_{k+1}}{g_k^T g_k}$
- (6) $d_{k+1} = -g_{k+1} + \beta_k d_k$

Abb. 7.26. Algorithmus der Methode der konjugierten Gradienten (CG-Verfahren). (1) und (2) berechnen α_k nach (7.52). Der Vektor ω_k wird für das Zwischenergebnis der Berechnung Ad_k verwendet. (3) entspricht der Berechnung (7.51). (4) berechnet g_{k+1} für den nächsten Iterationsschritt nach (7.52), wofür die rekursive Form $g_{k+1} = Ax_{k+1} - b = A(x_k + \alpha_k d_k) - b = g_k + A\alpha_k d_k$. g_{k+1} entspricht dem Fehler der derzeitigen Näherung x_k zur exakten Lösung. In (5) und (6) wird der nächste Vektor d_{k+1} der Basis aus konjugierten Gradienten berechnet.

- der Matrix-Vektor-Multiplikation Ad_k in (1),
- zwei inneren Produkten $g_k^T g_k$ und $d_k^T \omega_k$ in (2),
- zwei sogenannten axpy-Operationen $x_k + \alpha_k d_k$ und $g_k + \alpha_k \omega_k$ (wobei axpy für die Operation $a x$ plus y steht) in (3) und (4),
- dem inneren Produkt $g_{k+1}^T g_{k+1}$ in (5) und
- der axpy-Operation $-g_{k+1} + \beta_k d_k$ in (6).

Da das Ergebnis des inneren Produktes $g_k^T g_k$ sowohl in Iterationsschritt k als auch im nächsten Iterationschritt $k+1$ benötigt wird, kann die Berechnung eines inneren Produktes pro Iterationsschritt eingespart werden. Damit sind nur noch eine Matrix-Vektor-Multiplikation und zwei innere Produkte pro Iterationschritt auszuführen.

Eine Möglichkeit der effizienten Implementierung von Algorithmen der linearen Algebra ist die Bereitstellung von effizienten Routinen für Basisoperationen wie der axpy-Operation oder der Matrix-Vektor-Multiplikation und dem Skalarprodukt. Eine solche Menge von Basisroutinen wurde durch die **BLAS** (*Basic Linear Algebra Subroutines*) Bibliothek zur Verfügung gestellt. Die Operationen der BLAS Bibliothek sind relativ einfache Operationen, die als Basisoperationen in Algorithmen der linearen Algebra vorkommen, darunter axpy-Operationen verschiedener Typen. Die Operation beschreibt, dass ein Skalar a mit einem Vektor x multipliziert und das Ergebnis zum Vektor y addiert wird. Mögliche Präfixe sind s und d für *simple precision* und *double precision*. Algorithmen können aus einzelnen BLAS-Operationen aufgebaut werden. Eine einführende Beschreibung zu BLAS wird z.B. in [40, 52] gegeben. Alle im CG-Verfahren vorkommenden Operationen gehören zu diesen Basisoperationen. Eine mögliche Parallelisierungsweise für Algorithmen der linearen Algebra besteht darin, die Operationen der BLAS-Bibliothek

zu parallelisieren und einen parallelen Algorithmus durch das Zusammensetzen der parallelen BLAS-Operationen zu konstruieren. Das CG-Verfahren ist aufgrund seiner Zusammensetzung aus solchen Basisoperationen ein idealer Kandidat für eine solche Vorgehensweise.

Im Folgenden gehen wir auf eine parallele Realisierung des CG-Verfahrens ein, die auf den Parallelisierungen der Matrix-Vektor-Operationen und des inneren Produktes aus Kapitel 3 basiert. Diese Parallelisierungen beruhen auf einer Datenverteilung der Matrix A und der beteiligten Vektoren. Für eine Parallelisierung der Methode der konjugierten Gradienten auf einem Rechner mit verteiltem Speicher ist es nun wichtig, dass diese Verteilungen der Daten zusammenpassen, so dass keine zusätzliche Kommunikation nötig wird. Die potentielle Gefahr zusätzlicher Kommunikation wird im Abhängigkeitsgraphen der Basisoperationen des CG-Verfahrens deutlich, der in Abbildung 7.27 dargestellt ist.

Abbildung 7.27 zeigt die Datenabhängigkeiten zwischen den Berechnungsschritten (1)–(6) des CG-Algorithmus aus Abbildung 7.26. Datenabhängigkeiten beziehen sich nicht nur auf einzelne Werte, sondern können auch bzgl. mehrdimensionaler Datenstrukturen wie Vektoren oder Matrizen auftreten. Die Berechnungen und deren Abhängigkeiten werden in einem gerichteten azyklischen Graphen (auch DAG für engl. *directed acyclic graph* genannt) dargestellt. Jeder Knoten des Graphen entspricht der Berechnung einer Basisoperation. In Abbildung 7.27 sind die Knoten mit Nummern bezeichnet, die den in 7.26 gegebenen Berechnungen entsprechen. Eingehende Kanten entsprechen den benötigten Daten für die Berechnung. Eine ausgehende Kante entspricht dem berechneten Datum, also der Variablen auf der linken Seite der Berechnung. Dargestellt ist die Berechnungsfolge eines Iterationsschrittes k . Die Datenabhängigkeiten zum $(k-1)$ -ten und $(k+1)$ -ten Iterationsschritt sind durch gestrichelte Linien skizziert. So benötigt Knoten (2) das Zwischenergebnis ω_k aus Berechnung (1) und d_k und γ_k aus dem vorangegangenen Iterationsschritt $k-1$. Der Abhängigkeitsgraph beschreibt mögliche Berechnungsreihenfolgen der Berechnungen (1)–(6). Die Matrix-Vektor-Multiplikation (1) muss beendet sein bevor Berechnung (2) beginnt und diese muss wiederum beendet sein bevor Berechnung (3) bzw. Berechnung (4) beginnen kann. Berechnung (3) kann allerdings gleichzeitig zu den Berechnungen (4)–(6) stattfinden. Eine korrekte Berechnungsreihenfolge ist also (1),(2),(3),(4),(5),(6), aber auch (1),(2),(4),(5),(6),(3).

Neben möglicher Berechnungsreihenfolgen gibt der Abhängigkeitsgraph aber auch wichtige Informationen über notwendige Kommunikation in Realisierungen für Rechner mit verteiltem Speicher an. Datenabhängigkeiten können in Form von Skalaren vorkommen, wie bei den Größen α_k und β_k , oder in Form von zusammengesetzten Datenobjekten, wie den Vektoren ω_k oder g_k . Wird ein Vektor verteilt durch mehrere Prozessoren berechnet, so liegt er entsprechend in einer bestimmten Datenverteilung vor. Wird dieser Vektor von einer nachfolgenden Operation in einer anderen Datenverteilung

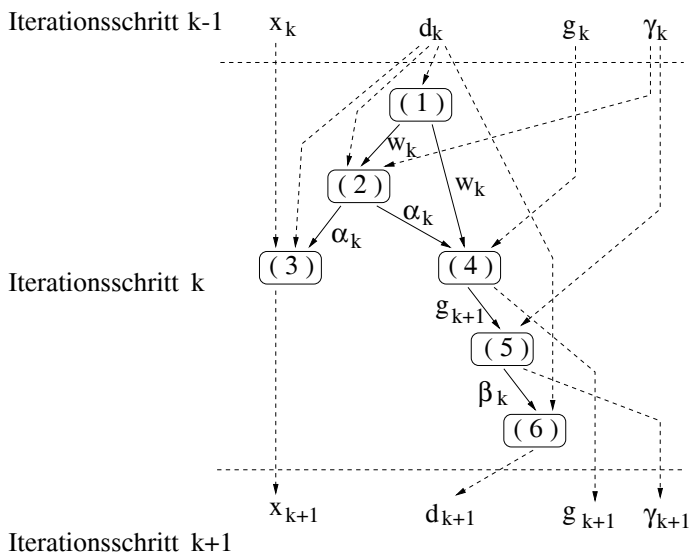


Abb. 7.27. Datenabhängigkeiten zwischen den Berechnungen (1)–(6) des CG-Verfahrens aus Abbildung 7.26. Die Knoten repräsentieren jeweils eine der Berechnungen des Iterationsschrittes k . Eingehende Kanten sind mit den benötigten Daten und ausgehende Kanten sind mit dem berechneten Datum annotiert. Sind zwei Knoten mit einer gerichteten Kante verbunden, so bedeutet dies, dass das von einem Knoten mit ausgehender Kante berechnete Datum in der Berechnung des Knotens mit eingehender Kante benötigt wird. Die Datenabhängigkeiten zum vorhergehenden Iterationsschritt $k-1$ und zum nachfolgenden Iterationsschritt $k+1$ sind mit gestrichelten Kanten gekennzeichnet. Die Bezeichnungen der Variablen entsprechen denen im Algorithmus 7.26. Zusätzlich wurde die Bezeichnung γ_k für das Zwischenergebnis $\gamma_k = g_k^T g_k$ eingeführt, das in (5) mitberechnet wird und zur Berechnung von α_k und β_k benutzt wird.

benötigt, so muss eine *Datenumverteilung* vorgenommen werden. Ebenso tritt dieses Problem bereits bei Skalaren auf. Wird ein Skalar auf einem Prozessor berechnet, aber anschließend von allen Prozessoren benötigt, so muss der Wert über explizite Kommunikation zur Verfügung gestellt werden.

Da Kommunikationsoperationen teure Operationen sind, versucht man, die Datenverteilung von Eingabewerten und berechneten Werten so zu gestalten, dass möglichst wenig zusätzliche Kommunikation für Umverteilungen benötigt wird. Dabei sind insbesondere Iterationsgrenzen zu beachten, an denen *Invarianten von Datenverteilungen* vorliegen sollten, die im nachfolgenden Iterationsschritt benötigt werden, damit jeder Iterationsschritt nach dem gleichen parallelen Schema ablaufen kann. Eine solche Parallelisierung ist z.B. die folgende, die eine zeilenblockweise Verteilung der Matrix A und eine blockweise Verteilung der Vektoren d_k , ω_k , g_k , x_k und r_k verwendet und die in jedem Iterationsschritt die folgenden Berechnungen ausführt:

- (0) Vor Durchführung der eigentlichen Berechnungen wird der Vektor d_k durch eine Multi-Broadcastoperation von einer blockweisen Verteilung, in der er in der vorangegangenen Iteration berechnet wurde, in eine replizierte Verteilung überführt.
- (1) Die Matrix-Vektor-Multiplikation $\omega_k = Ad_k$ wird wie in Abschnitt 3.7.3 beschrieben entsprechend der zeilenblockweisen Datenverteilung von A verteilt errechnet. Da d_k repliziert vorliegt, ist keine Kommunikation erforderlich. Das Ergebnis ω_k liegt nach der Berechnung in blockweiser Verteilung vor.
- (2) Die inneren Produkte $g_k^T g_k$ und $d_k^T \omega_k$ werden verteilt errechnet, indem jeder Prozessor für seine lokalen Komponenten der beteiligten Vektoren Teilsummen berechnet. Diese werden dann mit Hilfe einer Einzel-Akkumulationsoperation, für die eine Addition als Reduktionsoperation verwendet wird, bei Prozessor P_1 aufgesammelt. Dieser Prozessor errechnet dann mit den ausgesammelten Werten den Wert von α_k und stellt diesen Wert durch eine Einzel-Broadcastoperation allen anderen Prozessoren zur Verfügung.
- (3) Da jeder Prozessor den Wert von α_k kennt, kann die saxpy-Operation $x_{k+1} = x_k + \alpha_k d_k$ verteilt ohne Kommunikation berechnet werden, indem jeder Prozessor die erforderlichen arithmetischen Operationen für seine lokalen Komponenten der beteiligten Vektoren x_k und d_k durchführt. Das Ergebnis x_{k+1} liegt in der gleichen Verteilung wie x_k , also blockweise vor.
- (4) Die saxpy-Operation $g_{k+1} = g_k + \alpha_k \omega_k$ wird analog errechnet. Das Ergebnis g_{k+1} liegt blockweise vor.
- (5) Die inneren Produkte $g_{k+1}^T g_{k+1}$ und $g_k^T g_k$ werden analog zu (2) verteilt errechnet und mit einer Einzel-Akkumulationsoperation bei Prozessor P_1 aufgesammelt. Prozessor P_1 errechnet β_k und schickt diesen Wert mit einer Einzel-Broadcastoperation an alle anderen Prozessoren.
- (6) Die saxpy-Operation $d_{k+1} = -g_{k+1} + \beta_k d_k$ wird analog (3) verteilt errechnet. Das Ergebnis d_{k+1} liegt blockweise vor.

Die Laufzeit eines Iterationsschrittes des CG-Verfahrens setzt sich aus der Laufzeit der beschriebenen Schritte zusammen. Wir gehen dabei von Vektoren der Größe n und von p Prozessoren aus. Die parallele Berechnung einer saxpy-Operation erfordert Laufzeit

$$T_{saxpy} = 2 \cdot \frac{n}{p} \cdot t_{op} ,$$

da jeder Prozessor n/p Komponenten des Ergebnisvektors berechnet und da für jede Komponente eine Multiplikation und eine Addition benötigt werden. Die Laufzeit für die Errechnung eines inneren Produktes ist

$$T_{ip} = 2 \cdot \left(\frac{n}{p} - 1 \right) \cdot t_{op} + T_{acc}(+)(p, 1) + T_{sb}(p, 1) ,$$

wobei $T_{acc}(op)(p, x)$ die Laufzeit einer Einzel-Akkumulation mit Reduktionsoperation op auf p Prozessoren und Nachrichten der Größe x bezeich-

ne. Die Berechnung des Teil-Skalarproduktes mit n/p Komponenten erfordert n/p Multiplikationen und $n/p - 1$ Additionen. Für die Matrix-Vektor-Multiplikation ergibt sich die Laufzeit

$$T_{MVM} = 2 \cdot \frac{n^2}{p} \cdot t_{op} ,$$

da jeder Prozessor n/p innere Produkte berechnet. Die Gesamtlaufzeit einer Iteration des konjugierten Gradientenverfahren beträgt also

$$T_{CG} = T_{mb}(p, \frac{n}{p}) + T_{MVM} + 2 \cdot T_{ip} + 3 \cdot T_{saxpy} ,$$

wobei $T_{mb}(p, x)$ die Ausführungszeit einer Multi-Broadcastoperation auf p Prozessoren und Nachrichten der Größe x bezeichne.

Literatur

1. F. Abolhassan, J. Keller, and W.J. Paul. On the Cost-Effectiveness of PRAMs. In *Proc. 3rd IEEE Symposium on Parallel and Distributed Processing*, pages 2–9, 1991.
2. A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiawicz, Beng-Hong Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proc. 22nd Int. Symp. on Computer Architecture (ISCA'95)*, pages 2–13. ACM, 1995.
3. A. Aggarwal, A.K. Chandra, and M. Snir. On Communication Latency in PRAM Computations. In *Proc. 1989 ACM Symposium on Parallel Algorithms and Architectures (SPAA'89)*, pages 11–21, 1989.
4. A. Aggarwal, A.K. Chandra, and M. Snir. Communication Complexity of PRAMs. *Theoretical Computer Science*, 71:3–28, 1990.
5. A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques & Tools*. Pearson-Addison Wesley, 2007.
6. S.G. Akl. *Parallel Computation – Models and Methods*. Prentice Hall, 1997.
7. A. Alexandrov, M. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model – One Step Closer towards a Realistic Model for Parallel Computation. In *Proc. 7th ACM Symposium on Parallel Algorithms and Architectures (SPAA'95)*, pages 95–105, Santa Barbara, California, July 1995.
8. E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, S. Ryu, G.L. Steele Jr., and S. Tobin-Hochstadt. The Fortress Language Specification. Technical report, Sun Microsystems, 2006.
9. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002.
10. S. Allmann, T. Rauber, and G. Rünger. Cyclic Reduction on Distributed Shared Memory Machines. In *Proc. 9th Euromicro Workshop on Parallel and Distributed Processing*, pages 290–297, Mantova, Italien, 2001. IEEE Computer Society Press.
11. G.S. Almasi and A. Gottlieb. *Highly Parallel Computing*. Benjamin Cummings, 1994.
12. G. Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computer Capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485, 1967.
13. C.J. Beckmann and C. Polychronopoulos. Microarchitecture Support for Dynamic Scheduling of Acyclic Task Graphs. Technical Report CSRD 1207, University of Illinois, 1992.
14. D.P. Bertsekas and J.N. Tsitsiklis. *Parallel and Distributed Computation*. Prentice Hall, 1989.
15. R. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 1998.

16. F. Bodin, P. Beckmann, D.B. Gannon, S. Narayana, and S. Yang. Distributed C++: Basic Ideas for an Object Parallel Language. In *Proc. Supercomputing'91 Conference*, pages 273–282, 1991.
17. D. Braess. *Finite Elemente*. Springer, 1997.
18. P. Brucker. *Scheduling Algorithms*. 4. Auflage, Springer, 2004.
19. D. R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.
20. N. Carriero and D. Gelernter. Linda in Context. *Commun. ACM*, 32(4):444–458, 1989.
21. N. Carriero and D. Gelernter. *How to Write Parallel Programs*. MIT Press, 1990.
22. R. Chandra, L. Dagum, D. Koher, D. Maydan, J. McDonald, and R. Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2001.
23. P. Charles, C. Donowa, K. Ebcioglu, C. Grothoff, A. Kielstra, C. von Praun, V. Saraswat, and V. Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *Proceedings of the OOPSLA'05*. ACM, 2005.
24. A. Chin. Complexity Models for All-Purpose Parallel Computation. In *Lectures on Parallel Computation*, chapter 14. Cambridge University Press, 1993.
25. M.E. Conway. A Multiprocessor System Design. In *Proc. AFIPS 1963 Fall Joint Computer Conference*, volume 24, pages 139–146. NewYork: Spartan Books, 1963.
26. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, 2001.
27. G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley, 4th edition, 2005.
28. D.E. Culler, A.C. Arpaci-Dusseau, S.C. Goldstein, A. Krishnamurthy, S. Lumetta, T. van Eicken, and K.A. Yelick. Parallel programming in Split-C. In *Proceedings of Supercomputing*, pages 262–273, 1993.
29. D.E. Culler, A.C. Dusseau, R.P. Martin, and K.E. Schauser. Fast Parallel Sorting under LogP: from Theory to Practice. In *Portability and Performance for Parallel Processing*, pages 71–98. Wiley, 1994.
30. D.E. Culler, R. Karp, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. *Proc. 4th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP'93)*, pages 1–12, 1993.
31. D.E. Culler, J.P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware Software Approach*. Morgan Kaufmann, 1999.
32. H.J. Curnov and B.A. Wichmann. A Synthetic Benchmark. *The Computer Journal*, 19(1):43–49, 1976.
33. W.J. Dally and C.L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Transactions on Computers*, 36(5):547–553, 1987.
34. J. Darlington, Y. Guo, H.W. To, and J. Yang. Functional Skeletons for Parallel Coordination. In *Proceedings of the EURO-PAR'95*, Lecture Notes in Computer Science 966, pages 55–66. Springer, 1995.
35. DEC. The Whetstone Performance. Technical Report, Digital Equipment Corporation, 1986.
36. A. Dierstein, R. Hayer, and T. Rauber. The ADDAP System on the iPSC/860: Automatic Data Distribution and Parallelization. *Journal of Parallel and Distributed Computing*, 32(1):1–10, 1996.
37. E.W. Dijkstra. Cooperating Sequential Processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, 1968.

38. J. Dongarra. Performance of various Computers using Standard Linear Equations Software in Fortran Environment. Technical Report CS-89-85, Computer Science Dept., University of Tennessee, Knoxville, 1990.
39. J. Dongarra and W. Gentzsch, editors. *Computer Benchmarks*. Elsevier, North Holland, 1993.
40. J.J. Dongarra, I.S. Duff, D.C. Sorenson, and Henk A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*. SIAM, 1993.
41. J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks - An Engineering Approach*. Morgan Kaufmann, 2003.
42. M. Dubois, C. Scheurich, and F. Briggs. Memory Access Buffering in Multiprocessors. In *Proc. 13th Int. Symp. on Computer Architecture (ISCA'86)*, pages 434–442. ACM, 1986.
43. J.R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. MIT Press, 1986.
44. T. Ellis, I. Phillips, and T. Lahey. *Fortran90 Programming*. Addison-Wesley, 1994.
45. J.T. Feo. An Analysis of the Computational and Parallel Complexity of the Livermore Loops. *Parallel Computing*, 7:163–185, 1988.
46. D. Flanagan. *Java in a Nutshell*. O'Reilly, 2005.
47. M.J. Flynn. Some Computer Organizations and their Effectiveness. *IEEE Transactions on Computers*, 21(9):948–960, 1972.
48. S. Fortune and J. Wyllie. Parallelism in Random Access Machines. In *Proc. 10th ACM Symposium on Theory of Computing*, pages 114–118, 1978.
49. High Performance Fortran Forum. High Performance Fortran Language Specification. *Scientific Programming*, 2(1):1–165, 1993.
50. I. Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995.
51. I. Foster. Compositional Parallel Programming Languages. *ACM Transactions on Programming Languages and Systems*, 18(4):454–476, 1996.
52. T. L. Freeman and C. Phillips. *Parallel Numerical Algorithms*. Prentice Hall, 1992.
53. A. Frommer. *Lösung linearer Gleichungssysteme auf Parallelrechnern*. Vieweg, 1990.
54. A. Gara, M.A. Blumrich, D. Chen, G. Chiu, P. Coteus, M.E. Giampapa, R.A. Haring, P. Heidelberger, D. Hoenike, G.V. Kopcsay, T.A. Liebsch, M. Ohnmacht, B.D. Steinmacher-Burow, T. Takken, and P. Vranas. Overview of the Blue Gene/L System Architecture. *IBM Journal of Research and Development*, 49(2/3):195–212, 2005.
55. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM Parallel Virtual Machine: A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA, 1996. Zugehörige Webseite: www.netlib.org/pvm3/book/pvm_book.html.
56. A. George, J. Liu, and E. Ng. User's Guide for SPARSPAK: Waterloo Sparse Linear Equations Package. Technical Report CS-78-30, Department of Computer Science, University of Waterloo, 1980.
57. A. George and J. W-H Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, 1981.
58. P.B. Gibbons. A More Practical PRAM Model. In *Proc. 1989 ACM Symposium on Parallel Algorithms and Architectures (SPAA'89)*, pages 158–168, 1989.
59. M.B. Girkar and C. Polychronopoulos. Automatic Extraction of Functional Parallelism from Ordinary Programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):166–178, 1992.
60. C.J. Glass and L.M. Li. The Turn Model for Adaptive Routing. In *Proc. 19th Int. Symp. on Computer Architecture (ISCA'92)*, pages 278–287. ACM, 1992.

61. G. Golub and Ch. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 3rd. edition, 1996.
62. G. Golub and J. Ortega. *Scientific Computing*. Academic Press, 1993.
63. A. Gottlieb, R. Grishman, C. Kruskal, K. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer – Designing an MIMD Shared Memory Parallel Computer. *IEEE Transactions on Computers*, 32(2):175–189, February 1983.
64. M.W. Goudreau, J.M. Hill, K. Lang, W.F. McColl, S.D. Rao, D.C. Stefanescu, T. Suel, and T. Tsantilas. A proposal for a BSP Worldwide standard. Technical Report, BSP Worldwide, www.bsp-worldwide.org, 1996.
65. A. Grame, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Programming*. Addison Wesley, 2003.
66. W. Gropp, S. Huss-Ledermann, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI: The Complete Reference, Vol.2: The MPI Extensions*. MIT Press, Cambridge, MA, 1998. Zugehörige Webseite: mitpress.mit.edu/book-home.tcl?isbn=0262571234.
67. T. Grün, T. Rauber, and J. Röhrig. Support for Efficient Programming on the SB-PRAM. *International Journal of Parallel Programming*, 26(3):209–240, 1998.
68. J.L. Gustafson. Reevaluating Amdahl’s law. *Commun. ACM*, 31(5):532–533, 1988.
69. W. Hackbusch. *Iterative Lösung großer schwachbesetzter Gleichungssysteme*. Teubner, 1993.
70. K. Hammond and G. Michaelson, editors. *Research Directions in Parallel Functional Programming*. Springer, 1999.
71. J. Handy. *The Cache Memory Book*. Academic Press, 2nd edition, 1998.
72. P.J. Hatcher and M.J. Quinn. *Data-Parallel Programming*. MIT Press, 1991.
73. J. Held and J. Bautista ans S. Koehl. From a Few Cores to Many – A Tera-Scale Computing Research Overview. Intel White Paper, Intel, 2006.
74. J. Held, J. Bautista, and S. Koehl. From a Few Cores to Many: A Tera-scale Computing Research Overview, 2006.
75. J.L. Hennessy and D.A. Patterson. *Computer Architecture — A Quantitative Approach*. Morgan Kaufmann, 3rd edition, 2003.
76. M.R. Hestenes and E. Stiefel. Methods of Conjugate Gradients for Solving Linear Systems. *J. Res. Nat. Bur. Stand*, 49:409–436, 1952.
77. Heywood, T. and Ranka, S. A Practical Hierarchical Model of Parallel Computation. *Journal of Parallel and Distributed Computing*, 16:212–249, 1992.
78. J.M.D. Hill, B. McColl, D.C. Stefanescu, M.W. Goudreau, K. Lang, S.B. Rao, T. Suel, T. Tsantilas, and R. Bisseling. BSPlib The BSB Programming Library. Technical Report TR-29-97, Oxford University, May 1997.
79. M. Hill, W. McColl, and D. Skillicorn. Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.
80. C.A.R. Hoare. Monitors: An Operating Systems Structuring Concept. *Commun. ACM*, 17(10):549–557, 1974.
81. R.W. Hockney. *The Science of Computer Benchmarking*. SIAM, 1996.
82. R.W. Hockney and C.R. Jesshope. *Parallel Computers*. Adam Hilgner Ltd., 1981.
83. F. Hoffmann. Unified Parallel C 101, 5/5/2006.
84. P. Hudak and J. Fasel. A Gentle Introduction to Haskell. *ACM SIGPLAN Notices*, 27, No.5, May 1992.
85. K. Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill, 1993.
86. J.D. Jackson. *Klassische Elektrodynamik*. de Gruyter, 2nd edition, 1983.
87. J. Jájá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

88. M. Johnson. *Superscalar Microprocessor Design*. Prentice Hall, 1991.
89. S. Johnsson and C. Ho. Optimum Broadcasting and Personalized Communication in Hypercubes. *IEEE Transactions on Computers*, 38(9):1249–1268, 1989.
90. J.A. Kahle, M.N. Day, H.P. Hofstee, C.R. Johns, T.R. Maeurer, and D. Shippy. Introduction to the Cell Multiprocessor. *IBM Journal of Research and Development*, September 2005.
91. J. Keller, Th. Rauber, and B. Rederlechner. Conservative Circuit Simulation on Shared-Memory Multiprocessors. In *Proc. 10th Workshop on Parallel and Distributed Simulation (PADS'96)*, pages 126–134. ACM, 1996.
92. St. Kleiman, D. Shah, and B. Smaalders. *Programming with Threads*. Prentice Hall, 1996.
93. G. Koch. Discovering Multi-Core: Extending the Benefits of Moore's Law. Intel White Paper, TechnologyIntel Magazine, 2005.
94. P.M. Kogge. An Exploitation of the Technology Space for Multi-Core Memory/Logic Chips for Highly Scalable Parallel Systems. In *Proceedings of the Innovative Architecture for Future Generation High-Performance Processors and Systems*. IEEE, 2005.
95. Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded Sparc processor. *IEEE Micro*, 25(2):21–29, March/April 2005.
96. M. Korch and T. Rauber. A Comparison of Task Pools for Dynamic Load Balancing of Irregular Algorithms. *Concurrency and Computation: Practice and Experience*, 16:1–47, 2004.
97. K. Krewell. Cell moves into the Limelight. Micorprocessor Report, Reed Business Information, February 2005. www.MPRonline.com.
98. J. Kurose and K. Ross. *Computer Networking, 3. Auflage*. Addison Wesley, 2005.
99. L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.
100. D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison Wesley, 1999.
101. F.T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, 1992.
102. D.E. Lenoski and W. Weber. *Scalable Shared-Memory Multiprocessing*. Morgan Kaufmann, 1995.
103. B. Lewis and D. J. Berg. *Multithreaded Programming with Pthreads*. Prentice Hall, 1998.
104. J. Liu. Computational Models and Task Scheduling for Parallel Sparse Cholesky Factorization. *Parallel Computing*, 3:327–342, 1986.
105. D.T. Marr, F. Binus, D.L. Hill, G. Hinton, D.A. Konfaty, J.A. Miller, and M. Upton. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, 6(1):4–15, 2002.
106. T. Mattson, B. Sandor, and B. Massingill. *Pattern for Parallel Programming*. Pearson – Addison Wesley, 2005.
107. M.K. McKusick, K. Bostic, M.J. Karels, and J.S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley, 1996.
108. F. McMahon. The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, Livermore, 1986.
109. M. Metcalf and J. Reid. *Fortran 90/95 explained*. Oxford University Press, 2002.

110. E.G. Ng and B.W. Peyton. A Supernodal Cholesky Factorization Algorithm for Shared-Memory Multiprocessors. Technical Report, Oak Ridge National Laboratory, 1991.
111. L.M. Ni and P.K. McKinley. A Survey of Wormhole Routing Techniques in Direct Networks. *IEEE Computer*, pages 62–76, February 1993.
112. B. Nichols, D. Buttlar, and J. Proulx Farrell. *Pthreads Programming*. O'Reilly & Associates, 1997.
113. M.A. Nichols, H.J. Siegel, and H.G. Dietz. Data Management and Control-Flow Aspects of an SIMD/SPMD Parallel Language/Compiler. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):222–234, 1993.
114. J. Nieplocha, J. Ju, M.K. Krishnan, B. Palmer, and V. Tipparaja. The Global Arrays User's Manual. Technical report, Pacific Northwest Laboratory, 2002.
115. S. Oaks and H. Wong. *Java Threads*. 3. Auflage, O'Reilly, 2004.
116. *OpenMP Application Program Interface, Version 2.5*, May 2005.
117. J. M. Ortega. *Introduction to Parallel and Vector Solutions of Linear Systems*. Plenum Publishing Corp., 1988.
118. J.M. Ortega and R.G. Voigt. *Solution of Partial Differential Equations on Vector and Parallel Computers*. SIAM, 1985.
119. P.S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, 1997.
120. C.H. Papadimitriou and M. Yannakakis. Towards an Architecture-Independent Analysis of Parallel Algorithms. In *Proc. 20th ACM Symposium on Theory of Computing*, pages 510–513, 1988.
121. D.A. Patterson and J.L. Hennessy. *Computer Organization & Design — The Hardware/Software Interface*. Morgan Kaufmann, 2006.
122. S. Pelegatti. *Structured Development of Parallel Programs*. Taylor and Francis, 1998.
123. L. Peterson and B. Davie. *Computer Networks - A Systems Approach*, 3. Auflage. Morgan Kaufmann, 2003.
124. G.F. Pfister. *In Search of Clusters*. Prentice Hall, 2nd edition, 1998.
125. A. Podehl, T. Rauber, and G. Rünger. A Shared-Memory Implementation of the Hierarchical Radiosity Method. *Theoretical Computer Science*, 196(1-2):215–240, 1998.
126. C.D. Polychronopoulos. *Parallel Programming and Compilers*. Kluwer Academic Publishers, 1988.
127. S. Prasad. *Multithreading Programming Techniques*. McGraw-Hill, 1997.
128. R.M. Ramanathan. Intel Multi-core Processors: Leading the Next Digital Revaluation. Intel White Paper, TechnologyIntel Magazine, 2005.
129. S. Ramaswamy, S. Sapatnekar, and P. Banerjee. A Framework for Exploiting Task and Data Parallelism on Distributed-Memory Multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1098–1116, 1997.
130. T. Rauber and G. Rünger. Optimal Data Distribution for LU Decomposition. In *Proc. of the EuroPar'95*, Lecture Notes in Computer Science 966, pages 391–402. Springer, 1995.
131. T. Rauber and G. Rünger. A Transformation Approach to Derive Efficient Parallel Implementations. *IEEE Transactions on Software Engineering*, 26(4):315–339, 2000.
132. T. Rauber and G. Rünger. Deriving Array Distributions by Optimization Techniques. *Journal of Supercomputing*, 15:271–293, 2000.
133. T. Rauber and G. Rünger. Tlib - A Library to Support Programming with Hierarchical Multi-Processor Tasks. *Journal of Parallel and Distributed Computing*, 65(3):347–360, 2005.

134. T. Rauber, G. Rünger, and C. Scholtes. Execution Behavior Analysis and Performance Prediction for a Shared-Memory Implementation of an Irregular Particle Simulation Method. *Simulation: Practice and Theory*, 6:665–687, 1998.
135. T. Rauber, G. Rünger, and C. Scholtes. Scalability of Sparse Cholesky Factorization. *Int. Journal of High Speed Computing*, 10(1):19–52, 1999.
136. J.K. Reid. On the Method of Conjugate Gradients for the Solution of Large Sparse Systems of Linear Equations. In *Large Sparse Sets of Linear Equations*, pages 231–254. Academic Press, 1971.
137. M. Rosing, R.B. Schnabel, and R.P. Waever. The DINO Parallel Programming language. Technical Report CU-CS-501-90, Computer Science Dept., University of Colorado at Boulder, Boulder, CO, 1990.
138. E. Rothberg and A. Gupta. An Evaluation of Left-Looking, Right-Looking and Multifrontal Approaches to Sparse Cholesky Factorization on Hierarchical-Memory Machines. *Int. J. High Speed Computing*, 5(4):537–593, 1993.
139. J. Rothnie. Overview of the KSR1 Computer System. Technical Report, Kendall Square Research, 1992.
140. G. Rünger. Parallel Programming Models for Irregular Algorithms. In *Parallel Algorithms and Cluster Computing*, pages 3–23. Springer Lecture Notes in Computational Science and Engineering, 2006.
141. Y. Saad. *Iterative Methods for Sparse Linear Systems*. International Thomson Publ., 1996.
142. Y. Saad. Krylov Subspace Methods on Supercomputers. *SIAM Journal on Scientific and Statistical Computing*, 10:1200–1332, 1998.
143. J.E. Savage. *Models of Computation*. Addison-Wesley, 1998.
144. R. Schaback and H. Werner. *Numerische Mathematik*. 5. Auflage, Springer, 2004.
145. C. Scheurich and M. Dubois. Correct Memory Operation of Cache-Based Multiprocessors. In *Proc. 14th Int. Symp. on Computer Architecture (ISCA'87)*, pages 234–243. ACM, 1987.
146. D. Sima, T. Fountain, and P. Kacsuk. *Advanced Computer Architectures*. Addison-Wesley, 1997.
147. J.P. Singh. *Parallel Hierarchical N-Body Methods and Their Implication for Multiprocessors*. PhD Thesis, Stanford University, 1993.
148. D. Skillicorn and D. Talia. Models and Languages for Parallel Computation. *ACM Computing Surveys*, 30(2):123–169, 1998.
149. B. Smith. Architecture and Applications on the HEP Multiprocessor Computer Systems. *SPIE (Real Time Signal Processing IV)*, 298:241–248, 1981.
150. M. Snir, S. Otto, S. Huss-Ledermann, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, 1996. Zugreifbar über: www.netlib.org/utk/papers/mpi_book/mpi_book.html.
151. M. Snir, S. Otto, S. Huss-Ledermann, D. Walker, and J. Dongarra. *MPI: The Complete Reference, Vol.1: The MPI Core*. MIT Press, Cambridge, MA, 1998. Zugehörige Webseite: mitpress.mit.edu/book-home.tcl?isbn=0262629155.
152. SPEC Newsllett. 2(3):3-4. *SPEC Benchmark Suite Release 1.0*, 1990.
153. W. Stallings. *Computer Organization and Architecture*, 6. Auflage. Prentice Hall, 2002.
154. J. Stoer. *Numerische Mathematik I*. 9. Auflage, Springer, 2004.
155. J. Stoer and R. Bulirsch. *Numerische Mathematik II*. 5. Auflage, Springer, 2005.
156. H. S. Stone. Parallel Processing with the Perfect Shuffle. *IEEE Transactions on Computers*, 20(2):153–161, 1971.
157. H. S. Stone. An Efficient Parallel Algorithm for the Solution of a Tridiagonal Linear System of Equations. *Journal of the ACM*, 20:27–38, 1973.

- 158. H. Sullivan and T. R. Bashkow. A Large Scale, Homogeneous, Fully Distributed Parallel Machine. *Proc. 4th Int. Symp. on Computer Architecture (ISCA '77)*, pages 105–124, 1977.
- 159. A. Tanenbaum. *Structured Computer Organization, 5th Edition*. Pearson-Prentice Hall, 2006.
- 160. S. Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley, 1999.
- 161. L.G. Valiant. A Bridging Model for parallel Computation. *Commun. ACM*, 33(8):103–111, 1990.
- 162. E.F. van de Velde. *Concurrent Scientific Computing*. Springer, 1994.
- 163. R.P. Weicker. Dhrystone: A Synthetic System Programming Benchmark. *Commun. ACM*, 29(10):1013–1030, 1984.
- 164. M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.
- 165. M. Yannakakis. Computing the Minimum Fill-in is NP-complete. *SIAM J. Algebraic Discrete Methods*, 2:77–79, 1991.
- 166. S.N. Zheltov and S.V. Bratanov. Measuring HT-Enabled Multi-Core: Advantages of a Thread-Oriented Approach. *Technology & Intel Magazine*, December 2005.
- 167. A.Y.H. Zomaya, editor. *Parallel & Distributed Computing Handbook*. Computer Engineering Series. McGraw-Hill, 1996.

Index

- Abarbeitungsplan, 119
- adaptiver Routingalgorithmus, 54
- Akkumulationsoperation
 - Einzel-Akkumulation, 144
 - in MPI, 226
 - Multi-Akkumulation, 145
- Alpha 21164, 13
- Alpha 21264, 13
- AMD
 - Athlon, 13
 - Opteron, 13
- Amdahlsches Gesetz, 170
- Antwortzeiten, 158
- Architekturmodell, 114
- assoziativer Cache, 79
- Assoziativität beim Cache, 77
- asymptotische Laufzeit, 175
- asynchrone Kommunikation, 210
- aufspannender Baum, 147

- Backoff-Strategie, 293
- Bandbreite, 65
- Bandbreite einer Bandmatrix, 409
- Bandmatrix, 409
- Banyan-Netzwerk, 50
- Barrier-Synchronisation
 - in Java, 349
 - in MPI, 239
 - in UPC, 380
- Baseline-Netzwerk, 50
- Baum-Netzwerk, 42
- Bedingungsvariable
 - in Pthreads, 296
 - mit Java-Threads, 350
- Benchmarkprogramme, 164
- Benes-Netzwerk, 50
- Benutzer-CPU-Zeit, 158
- Berechnungsmodell, 114
 - BSP, 200
 - LogP, 203
 - PRAM, 198

- Bisektionsbandbreite, 35
- BLAS, 464
- Block-Tridiagonalsystem, 421
- blockierende MPI-Operation, 209
- blockierendes Netzwerk, 63
- blockweise Datenverteilung, 135
- blockzyklische Datenverteilung, 135
- Broadcastoperation
 - auf einem Hyperwürfel, 180
 - auf binärem Baum, 189
 - auf einem Gitter, 179
 - auf einem Ring, 178
 - auf linearem Feld, 177
 - Einzel-Broadcast, 143
 - MPI, 224
 - Multi-Broadcast, 145
- BSP-Modell, 200
 - Barrier-Synchronisation, 202
 - h-Relation, 202
 - Superschritt, 201
- Bus-Snooping, 86
- Butterfly-Netzwerk, 50
- Bytetransferzeit, 66

- Cache, 30, 73–92
 - assoziativer Cache, 79
 - Assoziativität, 77
 - Datencache, 83
 - Instruktionscache, 83
 - LFU-Ersetzungsmethode, 81
 - LRU-Ersetzungsmethode, 81
 - mehrstufiger Cache, 83
 - Mengen-assoziativer Cache, 79
 - Rückschreibestrategien, 82
 - write-back-Cache, 82
 - write-through-Cache, 82
- cache hit, 75
- cache miss, 75
- Cache-Fehlzugriff, 75
- Cache-Kohärenz, 84–92
 - Bus-Snooping, 86

- Directory-Protokoll, 90
- Invalidierungs-Protokoll, 87
- MESI-Protokoll, 89
- MSI-Protokoll, 87
- Update-Protokoll, 89
- Cache-Kohärenz-Problem, 84
- Cacheblock, 74
- Cachetreffer, 75, 78
- CC-NUMA-Rechner, 31
- CCC-Netzwerk, 41
- Cell-Prozessor, 107
 - schematischer Aufbau, 108
- CG-Verfahren, 460–468
 - A-orthogonal, 462
 - konjugierte Vektoren, 462
- Chapel, 372
- Chip Multiprocessing, 98
- Cholesky-Zerlegung, 200, 443–460
 - Abspeicherungsschema, 450
 - Eliminationsbaum, 453
 - Left-Looking, 447
 - Right-Looking, 448
 - sequentieller Algorithmus, 444
 - Superknoten, 448
- Circuit-Switching, 67
- Client-Server-Modell, 133, 312
- Cluster, 24
- Congestion, 53
- Contention, 53
- COW, 24
- CPI-WERT, 159
- CRCW-PRAM, 199
- CREW-PRAM, 199
- Crossbar-Netzwerk, 46
- Cube-connected-cycles, 41
- Cut-Through-Routing, 69

- Datenabhängigkeit
 - Anti-, 120
 - Ausgabe-, 120
 - Fluss-, 120
- Datenabhängigkeiten, 120
- Datenabhängigkeitsgraph, 121
- Datencache, 83
- datenparallele Programmiersprache, 122
- Datenparallelität, 122
- Datenverteilung, 134–139
 - blockweise, 135, 136
 - blockzyklische, 135, 136
 - Einzelverteilung, 137
 - für zweidimensionales Feld, 135
 - gesamtzyklische, 392
 - parametrisierte, 137
 - replizierte, 137
 - schachbrettartige, 136
 - streifenweise, 136
 - Superblock, 393
 - Verteilungsfunktion, 137
 - Verteilungsvektor, 137
 - zyklische, 135, 136
- Deadlock, 292
- Deadlock bei MPI, 237
- Deadlock bei Routingalgorithmen, 55
- Deadlockfreiheit für XY-Routing, 56
- Deadlockgefahr in MPI, 215
- DEC Alpha 21164, 13
- DEC Alpha 21264, 13
- deterministischer Routingalgorithmus, 54
- Differentialgleichungen
 - Poisson-Gleichung, 403
- Differenz von Gruppen in MPI, 242
- dimensionsgeordnetes Routing, 54
- Directory-Protokoll, 90
- direkt-abgebildeter Cache, 78
- Dirichlet-Randbedingung, 403
- distributed shared memory, 371
- DMA-Controller, 23
- DMM, 21
- Dreieckszerlegung, 385
- duale Kommunikationsoperation, 148
- Durchmesser eines Netzwerkes, 34
- Durchsatz, 66, 158
- dynamischer Baum, 52

- E-Cube Routing, 55
- Effizienz, 170
- Einbettung, 36
 - mehrdimensionales Gitter in Würfel, 45
 - Ring in Würfel, 43
 - zweidimensionales Feld in Würfel, 44
- Einzel-Akkumulationsoperation, 143
- Einzel-Broadcastoperation, 143
 - auf einem Hyperwürfel, 180
 - auf binärem Baum, 189
 - auf einem Gitter, 179
 - auf einem Ring, 178
 - auf linearem Feld, 177
 - in MPI, 224
- Einzelschritt-Verfahren, 427
- Einzeltransfer, 143
- Empfängeroverhead, 66
- ERCW-PRAM, 199
- EREW-PRAM, 199

- Erzeugung von Prozessen, 131
- Erzeugung von Threads, 131
- explizite Parallelität, 129
- Fünfpunkt-Gleichung, 405
- Fünfpunkt-Stern, 405
- Fat-Tree, 52
- Feld-Netzwerk, 39
- finite Differenzen, 404
- flit, 71, 73
- Flusskontrollmechanismen, 72
- Flynnsche Klassifizierung, 18
- forall-Schleife, 124
- Fork-Join, 131
- fork-join-Prinzip, 355
- FORTRAN 90/95, 122
- Fortress, 372
- funktionale Programmiersprache, 129
- Funktionsparallelität, 127
- Gatheroperation, 144
 - in MPI, 229
- Gauß-Elimination, 384–402
 - gesamtzyklische Implementierung, 392
 - Pivotsuche, 387
 - Rückwärtssubstitution, 385
 - Vorwärtselimination, 385
 - zeilenzyklische Implementierung, 388
- Gauß-Seidel-Verfahren, 427
 - parallele Realisierung, 432
- gemeinsame Variablen, 139
- gemeinsamer Speicher, 25
- Gesamtaustausch, 146
 - auf linearem Feld, 178
 - auf binärem Baum, 192
 - auf einem Gitter, 179
 - auf einem Hyperwürfel, 188
 - in MPI, 235
- Gesamtschritt-Verfahren, 427
- Gesetz von Moore, 10, 100
- gespiegelter Gray-Code, 43
- Gitter-Netzwerk, 39
- globale Kommunikationsoperation, 142, 224
- globaler Speicher, 25
- Grad der Parallelität, 6
- Grad eines Netzwerkes, 35
- Granularität, 5, 120
- Graph
 - Taskgraph, 127
- Gustafson-Gesetz, 171
- Hamming-Distanz, 40
- Hardware Router, 23
- Header einer Nachricht, 64
- High Performance FORTRAN, 128
- Hyperthreading, 99
- Hyperwürfel, 23, 39
- IBM
 - Power5, 13
- IBM Blue Gene/L, 108
 - schematischer Aufbau, 110
- ILP-Prozessoren, 11
- implizite Parallelität, 129
- Instruktionscache, 83
- Instruktionsfenster, 15
- Intel
 - Itanium, 13
 - Pentium 4, 13, 104
 - Pentium II, 13
 - Pentium III, 13
 - Pentium Pro, 13
- inverse perfect shuffle, 42
- Iterationsverfahren, 425
- Jacobi-Verfahren, 427
 - parallele Realisierung, 430
- Java-Threads, 334–354
 - Bedingungsvariable, 350
 - Klasse Thread, 335
 - Matrix-Multiplikation, 339
 - Mutexvariable, 341
 - Scheduling, 351
 - Signalmechanismus, 344
 - Synchronisation, 339
- JOR-Verfahren, 428
- Kanalabhängigkeitsgraph, 56
- kohärent, 85
- Kohärenz eines Speichersystems, 85
- kollektive Kommunikationsanweisung
 - in MPI, 224
- Kommunikationsgebiet in MPI, 240
- Kommunikationsoperation, 142
- Kommunikator in MPI, 210, 240
- Konflikte im dynamischen Netzwerk, 62
- konjugierter Vektor, 462
- Kosten eines parallelen Programmes, 168
- kritischer Bereich, 140
 - in OpenMP, 365
- LAN, 24
- Laplace-Operator, 403
- Lastgleichgewicht, 5
- Lastverteilung, 118

- Latenz, 65
- Leistung, 157
- LFU-Ersetzungsmethode, 81
- lineare Gleichungssysteme, 383–443
 - Bandmatrix, 409
 - CG-Verfahren, 460
 - Cholesky-Zerlegung, 443
 - direkte Verfahren, 383
 - Gauß-Elimination, 384
 - iterative Verfahren, 383
 - LR-Zerlegung, 385
 - Pivotsuche, 387
 - Poisson-Gleichung, 403, 423
 - rekursives Verdoppeln, 411
 - Tridiagonalsysteme, 409
 - zyklische Reduktion, 415
- lineares Feld, 37
- Load balancing, 5
- logischer Prozessor, 99
- LogP-Modell, 203
- Lokalität der Speicherzugriffe, 75
- LR-Zerlegung, 385
- LRU-Ersetzungsmethode, 81

- Mapping, 4, 118
- Maschinenmodell, 114
- Master-Slave-Modell, 133, 283
- Master-Worker-Modell, 133
- Matrix-Multiplikation
 - in Java, 339
 - in OpenMP, 360
 - in Pthreads, 287
 - in UPC, 380
- Matrix-Vektor-Multiplikation, 150
 - in UPC, 379
 - Laufzeit, 196
- mehrstufiger Cache, 83
- mehrstufiges Schalter-Netzwerk, 47
- Mengen-assoziativer Cache, 79
- message passing, 142
- Methode der konjugierten Gradienten, 460
- Methode des steilsten Anstieges, 461
- MFLOPS-Rate, 161
- MIMD, 19
- minimale Routingalgorithmen, 54
- MIPS-Rate, 160
- MISD, 19
- Mittlere Speicherzugriffszeit, 163
- Modellproblem, 403
- Monitor, 142
- MPI, 208–251
 - Übertragungsmodus, 221
 - Akkumulationsoperation, 226
 - Broadcastoperation, 224
 - Datentypen, 212
 - Deadlock, 215, 237
 - Gatheroperation, 229
 - Gesamtaustausch, 235
 - kollektive Kommunikationsanweisung, 224
 - Kommunikationsgebiet, 240
 - Kommunikator, 240
 - MPI_Allgather, 233
 - MPI_Allgatherv, 234
 - MPI_Allreduce, 234
 - MPI_Alltoall, 235
 - MPI_Alltoallv, 236
 - MPI_Bcast, 224
 - MPI_Bsend, 223
 - MPI_Buffer_attach, 223
 - MPI_Buffer_detach, 223
 - MPI_Cart_coords, 247
 - MPI_Cart_create, 246
 - MPI_Cart_get, 250
 - MPI_Cart_rank, 247
 - MPI_Cart_shift, 247
 - MPI_Cart_sub, 249
 - MPI_Cartdim_get, 249
 - MPI_Comm_compare, 243
 - MPI_Comm_create, 243
 - MPI_Comm_dup, 244
 - MPI_Comm_free, 244
 - MPI_Comm_group, 241
 - MPI_Comm_rank, 243
 - MPI_Comm_size, 243
 - MPI_Comm_split, 244
 - MPI_Dims_create, 246
 - MPI_Gather, 229
 - MPI_Gatherv, 230
 - MPI_Get_count, 211
 - MPI_Group_compare, 243
 - MPI_Group_difference, 242
 - MPI_Group_excl, 242
 - MPI_Group_free, 243
 - MPI_Group_incl, 242
 - MPI_Group_intersection, 241
 - MPI_Group_rank, 242
 - MPI_Group_size, 242
 - MPI_Group_union, 241
 - MPI_Ibsend, 223
 - MPI_Irecv, 219
 - MPI_Isend, 218
 - MPI_Issend, 223
 - MPI_Op_create, 227
 - MPI_Recv, 211

- MPI.Reduce, 226
- MPI.Scatter, 231
- MPI.Send, 210
- MPI.Sendrecv, 217
- MPI.Sendrecv_replace, 217
- MPI.Ssend, 223
- MPI.Test, 219
- MPI.Wait, 219
- MPI.Wtick, 251
- MPI.Wtime, 250
- Multi-Akkumulationsoperation, 234
- Multi-Broadcastoperation, 233
- nichtblockierende Sendeoperation, 218
- Prozessgruppe, 240
- Puffermodus, 223
- Reduktionsoperation, 226
- Scatteroperation, 231
- Standardmodus, 221
- synchroner Modus, 222
- Synchronisation, 238
- virtuelle Topologie, 245
- MPI-2, 260–273
 - einseitige Kommunikation, 265
 - Fenster, 264
 - MPI.Accumulate, 266
 - MPI.Comm_get_parent, 262
 - MPI.Comm_spawn, 261
 - MPI.Comm_spawn_multiple, 262
 - MPI.Get, 266
 - MPI.Info_create, 261
 - MPI.Info_delete, 261
 - MPI.Info_get, 261
 - MPI.Info_set, 261
 - MPI.Put, 265
 - MPI.Win_complete, 269
 - MPI.Win_create, 264
 - MPI.Win_fence, 268
 - MPI.Win_free, 264
 - MPI.Win_lock, 272
 - MPI.Win_post, 270
 - MPI.Win_start, 269
 - MPI.Win_test, 270
 - MPI.Win_unlock, 272
 - MPI.Win_wait, 270
 - Prozeßerzeugung, 261
 - RMA-Operation, 265
 - Sperr-Synchronisation, 271
 - Synchronisation, 267
 - Teilsynchronisation, 269
- MSI-Protokoll, 87
- Multi-Akkumulationsoperation, 145
 - in MPI, 234
- Multi-Broadcastoperation, 145
 - auf binärem Baum, 190
 - auf einem Hyperwürfel, 181
 - auf einem Ring, 178
 - auf linearem Feld, 177
 - in MPI, 233
- Multicore-Prozessor, 13, 98, 100
- Multicore-Prozessoren
 - Cell-Prozessor, 107
 - Intel Tera-scale Computing, 103
- Multipräfixoperation, 199
- Multiprozessor-Task-Scheduling, 119
- Multithreading, 30
 - ereignisbasiertes, 99
 - Hyperthreading, 99
 - simultanes, 99
 - Zeitscheiben, 99
- Mutexvariable
 - in Java, 341
 - in Pthreads, 287
- mutual exclusion, 140
- Nachrichtenübertragung, 21
- Nachrichtenaustausch, 142
- Netzwerk
 - Banyan, 50
 - Baseline, 50
 - Baum, 42
 - Benes, 50
 - Bisektionsbandbreite, 35
 - blockierendes, 63
 - Butterfly, 50
 - Crossbar, 46
 - Cube-connected-cycles, 41
 - Durchmesser, 34
 - dynamische Netzwerke, 62
 - dynamischer Baum, 52
 - Einbettung, 36
 - Fat-Tree, 52
 - Gitter, 39
 - Grad, 35
 - lineares Feld, 37
 - Omega, 49
 - Ring, 37
 - Shuffle-Exchange, 42
 - Torus, 39
 - vollständiger Graph, 37
 - Würfel, 39
- nichtblockierende MPI-Operation, 209
- nichtblockierendes Senden in MPI, 218
- nichtdeterministisches Verhalten, 140
- nichtminimale Routingalgorithmen, 54
- NOW, 24

NUMA, 31

Omega-Netzwerk, 49

OpenMP, 354–371

- atomare Operation, 366
- default Parameter, 357
- kritischer Bereich, 365
- `omp_destroy_lock`, 369
- `omp_destroy_nest_lock`, 369
- `omp_get_dynamic`, 364
- `omp_get_nested`, 358, 365
- `omp_init_lock`, 368
- `omp_init_nest_lock`, 368
- `omp_set_dynamic`, 364
- `omp_set_lock`, 369
- `omp_set_nest_lock`, 369
- `omp_set_nested`, 358, 365
- `omp_set_num_threads`, 364
- `omp_test_lock`, 369
- `omp_test_nest_lock`, 369
- `omp_unset_lock`, 369
- `omp_unset_nest_lock`, 369
- parallele Schleife, 358
- paralleler Bereich, 356, 362
- `pragma omp atomic`, 366
- `pragma omp barrier`, 365
- `pragma omp critical`, 365
- `pragma omp flush`, 366
- `pragma omp for`, 358
- `pragma omp master`, 367
- `pragma omp parallel`, 356
- `pragma omp sections`, 362
- `pragma omp single`, 363
- private Parameter, 356
- reduction Parameter, 367
- schedule Parameter, 359
- shared Parameter, 356
- Sperrmechanismus, 368

Owner-Computes-Regel, 123

P-cube-Routing, 59

Paket-Switching, 68

parallele Laufzeit, 5, 167

parallele Schleife, 123, 130

- doall-Schleife, 126
- dopar-Schleife, 125
- forall-Schleife, 124
- in OpenMP, 358
- in UPC, 379

paralleler Bereich

- in OpenMP, 356

paralleles Rechnersystem, 113

parallelisierender Compiler, 129

Parallelrechner

- SB-PRAM, 199

parametrisierte Datenverteilung, 137

Parbegin-Parend-Konstrukt, 132

Partial-Store-Ordering-Modell, 97

partieller Cut-Through-Routing, 71

PC-Modell, 97

perfect shuffle, 42

PGAS, 371

phits (physical units), 67

Pipelining, 11, 69

- in Pthreads, 306

- von Prozessen, 133

Pivotsuche, 387

Poisson-Gleichung, 403

potentieller Parallelitätsgrad, 6

PRAM-Modell, 115, 198

- Kosten, 199

Produzenten-Konsumenten

- mit Java-Threads, 351

- Pthreads-Implementierung, 323

Programmiermodell, 115

- BSP, 200

- Datenparallelität, 122

- Kriterien, 115

- LogP, 203

- Master-Slave, 133

- parallele Schleifen, 123

- PRAM, 198

Prozess, 118

Prozessgruppe in MPI, 240

Prozessor-Konsistenzmodelle, 97

Prozessorkern, 98, 101

Pthreads, 281–334

- Abbruch von Threads, 320

- Attribute, 292

- Bedingungsvariable, 296

- Client-Server, 312

- Datentypen, 284

- Deadlock, 292

- Erzeugung von Threads, 285

- Mutexvariable, 287

- Pipelining, 306

- Prioritätsinversion, 329

- `pthread_attr_getdetachstate`, 317

- `pthread_attr_getinheritsched`, 328

- `pthread_attr_getschedparam`, 326, 329

- `pthread_attr_getschedpolicy`, 327

- `pthread_attr_getscope`, 328

- `pthread_attr_getstackaddr`, 319

- `pthread_attr_getstacksize`, 319

- `pthread_attr_init`, 317

- pthread_attr_setdetachstate, 317
- pthread_attr_setinheritsched, 328
- pthread_attr_setschedparam, 326, 329
- pthread_attr_setschedpolicy, 327
- pthread_attr_setscope, 328
- pthread_attr_setstackaddr, 319
- pthread_attr_setstacksize, 319
- pthread_cancel, 320
- pthread_cleanup_pop, 322
- pthread_cleanup_push, 321
- pthread_cond_broadcast, 298
- pthread_cond_destroy, 296
- pthread_cond_init, 296
- pthread_cond_signal, 298
- pthread_cond_timedwait, 298
- pthread_cond_wait, 297
- pthread_create, 285
- pthread_detach, 286
- pthread_equal, 285
- pthread_exit, 286
- pthread_getspecific, 334
- pthread_join, 286
- pthread_key_create, 333
- pthread_key_delete, 333
- pthread_mutex_destroy, 289
- pthread_mutex_init, 289
- pthread_mutex_lock, 289
- pthread_mutex_trylock, 290
- pthread_mutex_unlock, 289
- pthread_mutexattr_init, 292
- pthread_mutexattr_setpshared, 292
- pthread_once, 302
- pthread_self, 285
- pthread_setcancelstate, 321
- pthread_setcanceltype, 321
- pthread_setspecific, 334
- pthread_testcancel, 320
- sched_get_priority_min, 326
- sched_rr_get_interval, 327
- Scheduling, 324
- Sichtbarkeit von Daten, 283
- Sperrmechanismus, 289
- thread-spezifische Daten, 333
- Puffermodus in MPI, 223
- Punkt-zu-Punkt-Verbindungen, 21
- PVM, 251–260
 - Akkumulationsoperation, 259
 - Austausch von Nachrichten, 255
 - Broadcastoperation, 258
 - Erzeugung von Tasks, 252
 - Prozeßgruppen, 257
 - pvm_addhosts, 253
 - pvm_barrier, 258
 - pvm_bcast, 258
 - pvm_config, 254
 - pvm_delhosts, 254
 - pvm_gsize, 258
 - pvm_initsend, 255
 - pvm_joining, 257
 - pvm_lvgroup, 258
 - pvm_mcast, 257
 - pvm_pkint, 256
 - pvm_probe, 257
 - pvm_recv, 256
 - pvm_reduce, 259
 - pvm_send, 256
 - pvm_spawn, 252
 - pvm_tasks, 255
 - pvm_upkint, 256
 - virtuelle Maschine, 253
- quellenbasiertes Routing, 57
- Rückschreibestrategien, 82
- Rückwärtssubstitution, 385
- RAM-Modell, 115
- Rechner mit gemeinsamem Speicher, 25
- Rechner mit verteiltem Speicher, 21
- red-black-ordering, 437
- Reduktionsoperation
 - für eine Akkumulation, 144
 - in MPI, 226
 - in OpenMP, 367
- reguläre Datenverteilung, 134
- Rekurrenzgleichung, 411
- rekursives Verdoppeln, 410–423
- Relaxationsparameter, 428
- Ring-Netzwerk, 37
- Rot-Schwarz-Anordnung, 437, 439
- Routing, 53–63
 - Cut-Through-Routing, 69
 - dimensionsgeordnetes, 54
 - E-Cube-Routing, 55
 - flit, 71
 - Kanalabhängigkeitsgraph, 56
 - P-cube-Routing, 59
 - partielles Cut-Through-Routing, 71
 - quellenbasiertes, 57
 - Store-and-Forward, 68
 - tabellenorientiertes, 57
 - Turn-Modell, 58
 - virtuelle Kanäle, 54, 59
 - virtuelles Cut-Through-Routing, 71
 - West-First-Routing, 58
 - Wormhole-Routing, 71
 - XY-Routing, 54, 56

- Routingalgorithmus, 53
 - adaptiver, 54
 - Deadlock, 55
 - deterministischer, 54
 - minimaler, 54
 - umgekehrt-dimensionsgeordneter, 61
- Routingtechnik, 33
- SB-PRAM, 199
- Scatteroperation, 144
 - auf binärem Baum, 190
 - in MPI, 231
- Schachbrett-Nummerierung, 437
- schachbrettartige Datenverteilung, 136
- Scheduling, 6, 14, 118
 - Java-Threads, 351
 - Prioritätsinversion, 329, 354
 - Pthreads, 324
- Scheduling-Verfahren, 119
- Schnittmenge von Gruppen in MPI, 241
- Semaphor, 141
 - Thread-Implementierung, 322
- Senderoverhead, 66
- Sequentialisierung, 139, 141
- sequentielle Konsistenz, 93
- Shuffle-Exchange-Netzwerk, 42
- sichere Implementierung in MPI, 216
- Signalmechanismus
 - in Java, 344
- Signalverzögerungszeit, 66
- SIMD, 19, 122, 132
- SIMD-Modell, 122
- SISD, 19
- Skalierbarkeit, 171
- SMM, 25
- SMP, 26
- Solaris 2, 279
- SOR-Verfahren, 428
 - parallele Realisierung, 432
- Spaltenpivotsuche, 387
- SPEC-Benchmarks, 165
- SPEC-Benchmarks, 10
- Speedup, 168
- Speicherkonsistenzmodelle, 92–98
 - Partial-Store-Ordering, 97
 - Prozessor-Konsistenz, 97
 - sequentielle Konsistenz, 93
 - Total-Store-Ordering, 97
 - Weak-Ordering, 98
- Sperr-Reihenfolge, 292
- Sperrmechanismus, 141
 - in MPI-2, 271
 - in OpenMP, 368
- Sperrvariable, 141
 - in UPC, 381
- SPMD, 123, 132
- Standardmodus in MPI, 221
- stark diagonaldominant, 427
- starkes Zeilensummenkriterium, 427
- Startupzeit, 66
- statisches Verbindungsnetzwerk, 34
- Store-and-Forward-Routing, 68
- streifenweise Datenverteilung, 136
- superpipelined, 11
- superskalar, 11, 14, 121
- Switching, 64–72
 - Circuit-Switching, 67
 - Contention, 71
 - Header einer Nachricht, 64
 - Paket-Switching, 68
 - phits, 67
 - Pipelining, 69
- Switching-Strategie, 64
- symmetrische Multiprozessoren, 26
- synchrone Kommunikation, 210
- synchroner Modus in MPI, 222
- Synchronisation
 - in MPI, 238
 - bei gemeinsamem Adressraum, 139
 - in MPI-2, 267
 - mit Java-Threads, 339
- tabellenorientiertes Routing, 57
- Taktrate, 158
- Task, 117
- Taskgraph, 127
- Taskparallelismus, 127
 - in MPI, 240
- Taskpool, 127
 - Pthread-Implementierung, 303
- Thread, 118
 - Erzeugung, 131
- Threadmodell, 277
- Threads, 277
 - 1:1-Abbildung, 279
 - in OpenMP, 355
 - N:1-Abbildung, 279
 - N:M-Abbildung, 279
- Topologie, 33
- Topologie eines Netzwerkes, 53
- Topologie in MPI, 245
- Torus-Netzwerk, 39
- total exchange, 146
- totaler Austausch, 146
- Transportlatenz, 66

- Tridiagonalmatrix, 409
- TSO-Modell, 97
- Turn-Modell, 58
- umgekehrt-dimensionsgeordneter
 - Routingalgorithmus, 61
- Untergruppen in MPI, 242
- UPC, 371
 - affinity, 371
 - gemeinsame Felder, 374
 - parallele Schleife, 379
 - Synchronisation, 380
 - Thread, 372
- Vektoranweisung, 122
- Verbindungsnetzwerk, 32
- Vereinigung von Gruppen in MPI, 241
- verteilter gemeinsamer Speicher, 117
- verteilter Speicher, 21
- virtuell-gemeinsamer Speicher, 28
- virtuelle Kanäle, 54, 59
- virtuelles Cut-Through-Routing, 71
- VLIW-Prozessor, 11, 14, 121
- voll-assoziativer Cache, 79
- vollständiger Graph, 37
- Vorwärtselimination, 385
- Würfel-Netzwerk, 39
 - k -facher d -Würfel, 42
- Weak-Ordering-Modell, 98
- wechselseitiger Ausschluss, 140
- West-First-Routing, 58
- Wormhole-Routing, 71
- write-back-Cache, 82
- Write-Back-Invalidierungs-Protokoll, 87
- Write-Back-Update-Protokoll, 89
- write-through-Cache, 82
- X10, 371
- XY-Routing, 54
- Zeilenpivotsuche, 387
- zeitkritischer Ablauf, 140
- Zerlegung, 117
- zyklische Datenverteilung, 135
- zyklische Reduktion, 410–423
 - parallele Realisierung, 417
 - Poisson-Gleichung, 423
- Zykluszeit, 158