Parallelisierung von
Anwendungen der Finite-Element-Methode
im Bauingenieurwesen

Lutz Lämmer
Technische Hochschule Darmstadt
Institut für Numerische Methoden und Informatik im Bauwesen

20. März 1997

# Vorwort

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Institut für Numerische Methoden und Informatik im Bauwesen des Fachbereichs Bauingenieurwesen der TH Darmstadt. Unter der wissenschaftlichen Leitung des Institutsdirektors Prof. Dr.-Ing. Udo Meißner hatte ich von 1991 an Gelegenheit, die Parallelverarbeitung kennenzulernen. Von den ersten Schritten bei der Installation eines Parallelrechners am Institut hin bis zur Projektleitung für die Entwicklung und Erforschung von parallelen Algorithmen konnte ich dabei Höhen, aber auch Tiefen dieses Wissenschaftsgebiets kennenlernen und studieren. Für diese Möglichkeiten bin ich Prof. Meißner sehr dankbar.
Gemeinsam mit Prof. Wriggers, Institut für Mechanik IV der TH Darmstadt, hatte er schon 1989, damals noch an der Universität Hannover, das zukünftige Potential dieser Entwicklung erkannt und einen Foschungsschwerpunkt zur „Entwicklung und Parallelisierung von Berechnungs- und Entwurfsprozessen im Bau- und Maschinenwesen auf massiv-parallelen Rechnersystemen“ an der TH Darmstadt initiiert. Als Leihgabe der DFG wurde von beiden Instituten ein leistungsfähiges Workstation-Netzwerk installiert, in das nunmehr zwei Parallelrechner mit 64 bzw. 32 Prozessoren eingebunden sind.
Die ersten Schritte in der Forschungsarbeit konnten wir, dank der kooperativen Zusammenarbeit mit den Mathematikern und Mechanikern der TU Chemnitz, namentlich Prof. Meyer und Prof. Mach, jetzt FH Schmalkalden, sehr zügig gehen. Erste Ergebnisse konnten dann auf dem Workshop Parallelisierung an der TH Darmstadt im Februar 1993 vorgestellt werden, der die Entwicklungsarbeiten von parallelen Algorithmen auf dem Gebiet des Bauingenieurwesens, vor allem im Deutschland repräsentativ darstellte.
In interdisziplinärer Zusammenarbeit entstand in der Folge das Darmstädter Zentrum für Wissenschaftliches Rechnen (DZWR), das eine fruchtbare Basis für die weitere erfolgreiche Arbeit ergab. Aufgaben der Visualisierung von Ergebnissen der numerischen Simulation auf Parallelrechnern konnten zum Beispiel nur mit der technischen Ausstattung des DZWR und des Hochschulrechenzentrums der TH Darmstadt erfüllt werden.
Eine Bereicherung der interdisziplinären Zusammenarbeit mit anderen Fachkollegen ergab sich dank der langjährigen freundschaftlichen Beziehung von Prof. Meißner zu Prof. M. Kawahara an der Chuo-Universität, Tokyo. Die Parallelisierung von Algorithmen der numerischen Strömungsmechanik mit den interessanten Anwendungen für dynamische Lastverteilungsverfahren entstanden in Zusammenarbeit mit meinem Freund T. Umetsu, ehemaliger Mitarbeiter von Prof. Kawahara, jetzt Professor am Maebashi City Institute of Technology, Gunma.
Während meiner Arbeit am Institut von Prof. Meißner hatte ich vielfältige Möglichkeiten, unsere Ergebnisse im In- und Ausland vorzustellen. Eine besonders fruchtbringende Zusammenarbeit konnte sich auf dieser Basis zur Gruppe für Structural Engineering Computational Technology von Prof. Topping an der Heriot-Watt-Universität, Edinburgh entwickeln. Ich hatte darüberhinaus die Gelegenheit, im Rahmen des Programms Training and Research on Advanced Computer Systems

insgesamt ein halbes Jahr am Parallelcomputerzentrum der Universität Edinburgh, lange Zeit das am leistungsstärksten ausgerüstete Rechenzentrum Europas, zu arbeiten.
Das vorliegende Ergebnis wäre ohne aktive Hilfe und Zusammenarbeit mit meinen Kollegen und Studenten unmöglich. Ein Teil der Arbeit floß unmittelbar in die Lehrveranstaltung „Paralleles Rechnen“ ein. Stellvertretend möchte ich hier Michael Burghardt für die ausdauernde und zuverlässige Umsetzung neuer Ideen, Ingo Schönenborn für die kritische Durchsicht und Diskussion des Manuskripts und Hartmut Katz für die manchmal etwas mühselige programmtechnische Umsetzung der konzipierten Graphikanbindung danken. Janos Sziveri von der SECT-Group der Heriot-Watt-University hat viele Aspekte der MPI-Programmierung und der Partitionierungsalgorithmen konstruktiv diskutiert. Ein entscheidender Beitrag zum Gelingen der Arbeit stammt nicht zuletzt von Jan Olden, mit dem ich gemeinsam das parallelisierte FEM-Programm zur Strukturanalyse von Plattentragwerken und eine Vielzahl von nützlichen Werkzeugen auf dem Weg dorthin entwickelt habe.
Zuletzt sei meiner Familie gedankt, die mir das notwendige Hinterland bereitete und manchmal auch die Motivation für dieses umfangreiche Vorhaben neu belebte.

Lutz Lämmer, Oktober 1996

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Kapitel 1

# Einleitung

## 1.1 Zielstellung und Abgrenzung

Die numerische Simulation ist heute die dritte wichtige Säule des Erkennsnisgewinns, neben dem physikalischen Experiment, d.h. der praktischen Erprobung und dem theoretischen Ansatz, d.h. der mathematischen Formulierung der Gesetzmäßigkeiten in der Natur mit den Mitteln der Differential- und Integralrechnung.
Leistungsfähige Simulationssoftware gehört zu den Werkzeugen des praktisch tätigen Ingenieurs. Schwerpunkte der Anwendung numerischer Simulationsmethoden im Bauingenieurbereich bilden dabei

- die Untersuchung des Tragverhaltens komplexer Baustrukturen (Strukturmechanik)
- die Untersuchung des Verhaltens von Grund- und Oberflächenwasser einschließlich des Stofftransports (Strömungsmechanik) und
- die Erforschung neuartiger Werkstoffe (Materialwissenschaften).

Die eingesetzten Standardmethoden zur Bearbeitung dieser Probleme basieren bevorzugt auf der Finite-Element-Methode, die sich als universell einsetzbares, flexibles und anpassungsfähiges Simulationswerkzeug etabliert hat. Die Aufgabe des Ingenieurs bei der Anwendung der Finite-Element-Methode besteht in der Modellierung einer konstruktiven Problemstellung mit numerischen Methoden in den Arbeitsschritten:

- Modellbildung (Abstraktion der Geometrie und des physikalischen Verhaltens, Auswahl geeigneter Materialgesetze),
- Diskretisierung (Aufstellen eines diskreten, numerischen Modells),
- Berechnung des numerischen Modells und
- Interpretation der Ergebnisse der Berechnung, Abschätzung der Diskretisierungsfehler und Kontrolle der Modellbildung.

Die ersten beiden Aufgaben sind im herkömmlichen Sinn dem Finite–Element–Präprozeß zuzuordnen, die beiden anderen Aufgaben entsprechen der Analyse bzw. dem Postprozeß. Während im Präprozeß entscheidende Arbeitsanteile durch schöpferische Ingenieurarbeit erbracht werden müssen, kann die Berechnung zu großen Teilen automatisch ablaufen.
Gegenstand dieser Arbeit sind Verfahren zur Leistungssteigerung der Finite-Element-Methode mit den Mitteln der Parallelverarbeitung. Die betrachteten Aspekte der Leistungssteigerung sind dabei

- die Erhöhung des beherrschbaren Problemumfangs und
- die Verringerung der Antwortzeit.

Beide Anforderungen können eigenständig betrachtet werden. In der Praxis des Bauingenieurs werden sie konkurrierend bewertet - um in einer bestimmten Zeit eine Antwort zu erhalten, wird gegebenenfalls die Problemgröße reduziert. Dabei wird oft eine ingenieurgerechte Idealisierung unter Berücksichtigung praktisch relevanter Randbedingungen erforderlich. Umgekehrt wird auch oft in Kauf genommen, daß für die Analyse eines Problems eine sehr lange Antwortzeit erforderlich ist.
Die Substrukturtechnik ist eine bereits seit langem bekannte Methode, ein Problem in beherrschbare Teilprobleme zu zerlegen, die nacheinander gelöst werden. Die Begrenzung des verfügbaren Hauptspeichers früherer Rechnergenerationen motivierte die Entwicklung dieser Methode besonders.
Die Lösung des Gesamtproblems kann mit der Substrukturtechnik entweder direkt aus der Überlagerung der Lösungen für die Teilprobleme ermittelt werden (entkoppelte Teilprobleme, explizite Lösungsverfahren) oder das Koppelproblem ist als implizites Problem gegeben, daß die Lösung eines Gleichungssystems erfordert. Dazu wird die Lösung der Teilprobleme unter Dirichletschen Randbedingungen erzeugt und daraus ein reduziertes Koppelproblem konstruiert. Die Lösung des Koppelproblems wird danach in die Lösung der Teilprobleme eingesetzt und die Systemantwort kann für jedes Teilproblem getrennt ermittelt werden. Voraussetzung für dieses Vorgehen ist die Gültigkeit des Superpositionsprinzips. Das bedeutet für nichtlineare Fragestellungen, daß das Problem linearisiert werden muß und die Schritte

- Aufbau und Lösen der Teilprobleme,
- Aufbau und Lösen des Koppelproblems und
- Einsetzen der Koppellösung und Ermitteln der Systemantwort

für jeden Iterationsschritt zu wiederholen sind. Aufbau und Lösen des Gleichungssystems in jedem Teilgebiet und die Rückrechnung unter Einsetzen der Koppellösung müssen dabei für jedes Teilgebeit getrennt, d.h. auf einem seriellen Rechner auch nacheinander abgearbeitet werden. Dieser Nachteil kann mit einem Parallelrechner, der in der Lage ist, alle Teilprobleme simultan zu lösen, ausgeglichen werden. Damit kann ein entscheidender Geschwindigkeitsvorteil gewonnen werden. Kritisch ist die Parallelisierung der Lösung des Koppelproblems. Dafür werden in der Arbeit Lösungen aufgezeigt. Die Parallelisierung erlaubt die Beherrschung größerer Systeme und die drastische Reduzierung der Antwortzeiten der Berechnungsprozesse.
Darüberhinaus ist es naheliegend, die parallel in den Teilstrukturen erzeugten Ergebnisdaten durch direkte Kopplung an ein verteiltes Postprozessorsystem auszuwerten. Die zeit- und vor allem speicherplatzintensive Übertragung der Ergebnisdaten auf einen Massenspeicher zur Auswertung im Postprozeß entfällt, wenn die verteilt auf dem Parallelrechner vorliegenden Ergebnisdaten zunächst selektiert, verdichtet und bewertet werden, bevor sie zu Dokumentationszwecken zusammengestellt werden. Die im Postprozeß beherrschbare Problemgröße kann damit erweitert werden und das Anwortzeitverhalten und die interaktiven Bearbeitungsmöglichkeiten sind unter bestimmten Voraussetzungen günstiger.
Die Arbeit stellt die Grundlagen der Hardware von Parallelrechnern und des verwendeten Programmiermodells dar. Sie beschreibt die Implementation und die daraus resultierenden Eigenschaften eines Softwaresystems zum Nachrichtenaustausch in einem Parallelrechner. Aufbauend auf dieser konzeptionellen Grundlage wird die Anwendung der Substrukturtechnik für die Lösung von Aufgaben aus der Strukturmechanik und der Strömungsmechanik gezeigt. Schwerpunkte werden dabei auf



die effiziente Lösung des linearen Gleichungssystems und die Implementation von numerischen Kernfunktionen auf einem Parallelrechner gelegt. Die Probleme der Visualisierung von verteilt vorliegenden Finite-Element-Daten werden abschließend behandelt.
Die algorithmischen Grundlagen der dem parallelen Herangehen entsprechenden, herkömmlichen, seriellen Verfahren werden nur insoweit dargestellt, wie es zum Verständnis des Parallelisierungsansatzes erforderlich ist. Der Parallelisierungsansatz benutzt das Modell der kommunizierenden Prozesse, die auf parallel arbeitenden Rechnern ohne gemeinsamen Hauptspeicher abgearbeitet werden. Andere parallele Programmierparadigmen werden lediglich so dargestellt, wie es zum Verständnis der Darlegungen oder zu Vergleichszwecken notwendig ist.
Das Ergebnis ist ein praktikables Herangehen an die Parallelisierung von rechenaufwendigen Problemen aus der Bauingenieurtätigkeit. Mit dieser modernen Methode können die Anwortzeit von Analyseprogrammen um Zehnerpotenzen reduziert und die Größe der beherrschbaren Probleme entscheidend vergrößert werden.

## 1.2 Entwicklung des Hochleistungsrechnen

### 1.2.1 Grand Challenges

Die großen Herausforderungen (engl. *grand challenges*) an die Leistungsfähigkeit der numerischen Simulation formulieren sich aus konkreten wirtschaftlichen und gesellschaftlichen Interessen. Angefangen bei dem Programm der USA, haben nachfolgend alle Industrienationen und auch die Europäische Union diese Herausforderungen übereinstimmend formuliert. Sie sollen deshalb hier noch einmal dargestellt werden.

**Wettervorhersage**

Die Wettervorhersage basiert, ausgehend von beobachteten Daten, auf der Simulation zukünftiger atmosphärischer Verhältnisse. Gegenwärtig sind rund um die Welt Vorhersagezentren mit Supercomputern wie CRAY X-MP und CDC 205 u.a. damit beschäftigt, praktisch rund um die Uhr Daten für die Berechnung geeignet aufzubereiten, Vorhersagewerte zu berechnen und Vorhersagemodelle zu verifizieren.
Der Vorhersagezeitraum, der mit der heute zur Verfügung stehenden Rechenleistung genügend genau simuliert werden kann, beträgt in Abhängigkeit von der geographischen Region bis zu 5 Tage. Die Genauigkeit hängt davon ab, wieviele Eingangsdaten in welcher Zeit zur Verfügung stehen und wie genau die geographischen und metereologischen Verhältnisse einer Region modelliert werden können. Mit einer höheren Auflösung und verbesserten Beobachtungsmethoden, unter Einbeziehung von geostationären Satelliten, hofft der *National Weather Service* der USA den Vorhersagezeitraum auf 8 bis 10 Tage auszudehnen.
Ein weiterer Schwerpunkte in der Klimaforschung bestehen in der Einbeziehung der Interaktionen zwischen den Ozeanen und der Atmosphäre in die Modellierung und in der Entwicklung von geeigneten, dreidimensionalen Modellen der Ozeane. Mit einer entsprechenden Rechenleistung scheint es möglich, ein Modell zu entwickeln, das die atmosphärischen Verhältnisse der gesamten Erde über einen längeren Zeitraum simulieren kann. Das Problem besteht neben der Datenerfassung und -aufbereitung in der Modellierung der komplexen, hochgradig nichtlinearen Abhängigkeiten zwischen den Modellkomponenten. Mit einer entspechenden räumlichen und zeitlichen Auflösung, die nur durch ein paralleles Rechnersystem beherrscht werden kann, ist eine genügend akkurate Berechnung möglich.

**Ingenieurwissenschaften**

Das Hochleistungsrechnen ist aus dem modernen Ingenieurwesen für Forschungs- und Entwicklungsaufgaben nicht mehr wegzudenken. Mehr und mehr werden ana-

lytische Lösungen oder aufwendige Laborexperimente durch Simulationsergebnisse ersetzt. Neben der Produktivitätssteigerung ermöglicht die Parallelverarbeitung auf diesem Gebiet als Schlüsseltechnologie Ergebnisse, die nicht sinnvoll mit anderen, z.B. labortechnischen Vorgehensweisen erreichbar sind.
Strömungssimulationen und Turbulenzmodelle der numerischen Strömungsmechanik (engl. *Computational Fluid Dynamics* - CFD) sind zum Beispiel sehr aufwendige Verfahren in den Ingenieurwissenschaften. Die Berechnung von diskreten Daten in einer Strömung erfordern viele Stunden Rechenzeit auf den heutigen Superrechnern. Neuere theoretische Untersuchungen auf diesem Gebiet zeigen, daß zur realitätsnahen Modellierung von turbulenten Strömungen die komplette Simulation des dreidimensionalen Phänomens erforderlich ist. Um einen vollständigen Datensatz zur Beschreibung turbulenter Strömungen durch experimentielle Versuchsanordnungen zu messen, müßten hochentwickelte Meßverfahren eingesetzt werden, deren Entwicklung und Einsatz um ein Vielfaches kostenintensiver wären, als die numerische Simulation.

**Materialwissenschaften**

Anwendungen von Höchstleistungsrechnungen in den Materialwissenschaften sind noch in einem frühen Stadium der Entwicklung. Auf zahlreichen Gebieten sind aber schon große Veränderungen sichtbar. In Zukunft wird der Erfolg auf diesem Gebiet davon abhängen, inwieweit es gelingt, verläßliche Aussagen auf der Basis von Computersimulationen zu erhalten. Viele Phänomene gerade im Zusammenhang mit der Entwicklung von Materialien für mikroelektronische Bauelemente können ohne numerisch aufwendige Berechnungsverfahren nicht korrekt erfaßt und erforscht werden. Parallelverarbeitung auf diesem Gebiet wird in der Zukunft die Voraussetzung für die Entwicklung neuer Materialien sein, deren Eigenschaften zuerst im Computer simuliert wurden.

**Plasmaphysik**

In diesem Wissenschaftsgebiet hat sich die Quanten-Chromo-Dynamik (QCD), die Theorie der Atombindungskräfte, als ein Problem entwickelt, das heute unter Einsatz modernster Höchstleistungsrechner modelliert wird. Da die Algorithmen zur Modellierung der Atombindungskräfte relativ einfach strukturiert sind, wird dieses Problem von den Herstellern moderner Superrechner oft als erste Höchstleistungs-Applikation auf ihre neuen Plattformen portiert. Andererseits ist QCD eine Anwendung, die sich in ihrer Detaillierung, in ihrem Datenaufwand und dem sinnvoll einzusetzenden Rechenaufwand bisher immer noch steigern läßt.

### 1.2.2 HPCN in den USA

1991 wurde vom USA Kongreß ein Programm zur Förderung der Entwicklung von Informationstechnologien zur Lösung einer Reihe von technologischen und rechentechnischen Herausforderungen - den *Grand Challenges* im sogenanten *High Performance Computing and Communicating (HPCC) Act* (Gesetz Nr. 102-194) verabschiedet. Die vier wesentlichen Einrichtungen ARPA (*Advanced Research Project Agency*), DOE (*Department of Energy*), NASA (*National Aeronautics and Space Administration*) und NSF (*National Science Foundation*) wurden mit der Koordination der Umsetzung des Gesetzes betraut. Das Programm definierte die *Grand Challenges* und legte Verantwortlichkeiten fest.
1991 umfaßte das Programm vier Hauptsäulen

- Hochleistungrechnersysteme (Hardware - *High Performance Computing Systems* HPCS)
- hochentwickelte Programmentwicklungstechnologie und Algorithmen (Software - *Advanced Software Technology and Algorithms* ASTA)



- nationale Infrastruktur zum Datenaustausch in Forschung und Ausbildung (Netzwerk - *National Research and Education Network* NREN) und
- Grundausstattung für Forschung und Personal (Menschen - *Basic Research and Human Resources* BRHR).

1993 ist das Programm um eine fünfte Komponente erweitert worden

- Technologie und Anwendung der Informationsinfrastruktur (Datenautobahn - *Information Infrastructure Technology and Applications* IITA).

In diesem Programm werden jährlich ca. 1 Milliarde US$ investiert, deren Aufteilung in Tabelle 1.1 dargestellt ist.

Tabelle 1.1: *Investitionen im HPCC-Programm für das Jahr 1995 (in Millionen US$), Organisationen und Programmkomponenten, Quelle: National Coordination Office (NCO) for HPCC, Washington, DC, Stand 1.2.1995*

| Einrichtung | HPCS | NREN | ASTA | IITA | BRHR | gesamt |
|---|---|---|---|---|---|---|
| ARPA | 110.7 | 61.1 | 29.6 | 140.8 | 15.2 | 357.4 |
| NSF | 21.7 | 52.9 | 141.2 | 50.6 | 62.2 | 328.6 |
| DOE | 10.9 | 16.8 | 75.5 | 1.2 | 21.0 | 125.4 |
| NASA | 9.7 | 12.7 | 81.2 | 17.5 | 3.8 | 124.9 |
| NIH | 4.9 | 8.4 | 23.8 | 29.1 | 15.6 | 81.8 |
| NIST | 6.8 | 3.7 | 4.6 | 41.4 | | 56.6 |
| NSA | 16.1 | 11.8 | 11.8 | 0.2 | 0.2 | 40.1 |
| NOAA | | 8.7 | 16.1 | 0.5 | | 25.3 |
| EPA | | 0.7 | 11.7 | 0.3 | 2.0 | 14.7 |
| gesamt | 180.8 | 176.8 | 395.5 | 281.6 | 120.0 | 1,154.7 |

ARPA wurde ausgewählt, die Entwicklung von massiv-parallelen Rechnern zu koordinieren (HPSC). Das Programm soll die Führungsrolle der USA auf dem Gebiet des Höchstleistungsrechnens mit der Entwicklung von skalierbaren Computersystemen sichern, die mit der entsprechenden Software eine realisierbare Leistungsfähigkeit von mindestens 1 Trillion arithmetischen Operationen pro Sekunde (Teraflops) garantieren. ARPA förderte dabei unter anderem die Anstrengungen von Intel Supercomputer Systems Division und Thinking Machines Corporation. In einem *Concurrent Supercomputer Consortium* (CSCC) wurden die Mittel zur Anschaffung und zum Betrieb eines Intel Touchstone DELTA Systems eingeworben. Intel Touchstone DELTA war der erste MIMD-Rechner mit einer bedeutend höheren Rechenleistung als ein herkömmlicher Vektorrechner. Die Arbeit des CSCC wurde zum Modell für eine Reihe weiterer Großrechnerbeschaffungen.
Die Entwicklung von Parallelrechnern brachte eine Reihe von unterschiedlichen Konzepten der amerikanischen Hersteller auf den Markt: Intel Paragon und nCube nCube-2 als Rechner mit ausschließlich lokalem Speicher und einem Verbindungsnetzwerk zum Datenaustausch zwischen den Prozessoren, TMC CM-5 und Cray Research T3D als datenparallele Maschinen mit virtuell gemeinsamen Speicher, Kendall Square KSR-2 mit hierarchisch organisiertem gemeinsamen Speicher und IBM SP2 als Netzwerk gekoppelter Workstations. Die neuesten Entwicklungen von SGI PowerChallenge und Convex Exemplar stellen einen Kompromiß zwischen hochentwickelter Vektorprozessortechnik und moderat parallelem ($p \leq 256$) Ansatz dar. Die Entwicklungsgeschwindigkeit für einen neuen Rechner erreicht, sicher auch aufgrund der durch das Programm wesentlich gesteigerten Nachfrage von mit öffentlichen

Mittel geförderten Einrichtungen, jetzt 12 bis 18 Monaten zwischen aufeinanderfolgenden Produktgenerationen.
Für 1996 wurde die Anschaffung des bisher größten Parallelrechners, eines massivparallelen ($p > 1024$) Rechnersystems angekündigt. Am Sandia National Laboratory, Albuquerque, New Mexico wird ein 8192-Prozessor Intel System installiert. Dieser Rechner wird die Leistungsmarke von 1 Teraflops erreichen. Ein Prototyp des neuen Intel-Systems mit 2256 Prozessoren gekoppelt mit einem 6768 Prozessor Intel i860$^{TM}$ XP-Systems konnte Mitte 1995 bereits 281 GFLOPS mit dem massiv-parallelen Linpack Benchmark erreichen.
Die im HPCC-Programm geförderten Einrichtungen schafften eine Anzahl der unterschiedlichen Rechnerarchitekturen an (z. Zt. mehr als 100 Installationen unterschiedlicher Parallelrechnersystemen in den USA). Die Installationen in den heute vier Nationalen Supercomputerzentren, die von NSF evaluiert und gefördert werden, zeigt Tabelle 1.2. Schwerpunkt der zukünftigen Entwicklung ist die Organisation des Metacomputing, d.h. der Nutzung der über das Land verteilten Rechnerressourcen über die Grenzen der Nationalen Supercomputerzentren hinaus. Ziel ist es, die zur Verfügung stehende Rechnerleistung optimal zur Verfügung zu stellen.
NSF wurde bestimmt, die Arbeiten am Netzwerk zu leiten. Das NREN vereint das landes- und weltweit organisierte Internet mit lokalen oder regionalen Netzwerken. Neben dem Ausbau dedizierter Verbindungen zu Hochgeschwindigkeitsbahnen steht im Vordergrund die einheitliche und kompatible Bereitstellung von Netzen für die weltumspannende Kommunikation auf Basis des TCP/IP Protokolls und die Entwicklung der Kommunikationssoftware (Gopher, World Wide Web oder Videokonferenzsysteme) und zugehöriger Datenaustauschstandards (Hypertext, Video- und Klanginformationen).
Das DOE installierte eine Auswahl von Prototypen von massivparallelen Rechner (engl. *massive parallel processing* - MPP) -systemen und machte sie für eine Anzahl von Softwareprojekten (ASTA) verfügbar. Die bereits unternommenen Anstrengungen von verschiedenen Softwareentwicklungsfirmen und der Informatikwissenschaften wurden unter den neuen Elementen *Grand Challenges* und Hochleistungsrechen- und -forschungszentren gebündelt. *Grand Challenges* vereinen dabei angewandte Mathematik und Informatik mit konkreten Anwendungsfeldern, die unter massiver Förderung innerhalb weniger Jahre signifikante Forschungsergebnisse mit Hilfe der bereitgestellten Infrastruktur und des konzentrierten Wissenschaftlerpotentials erreichen. Neben dem DOE traten als Sponsoren für diese Zentren NSF und NASA auf. Primäre Aufgabe der Forschungszentren ist die Lösung der *Grand Challenge*, daneben bewährten sie sich aber auch als Testfeld der installierten Prototypen neuer Rechner. Die Rückkopplung zu den Herstellern und Entwicklern der Hardware konnte entscheidend verbessert werden. Außerdem konnte innerhalb der Zentren ein überdurchschnittlicher Betreuungsgrad der Nutzer und Anwender erreicht werden, der die Verfügbarkeit der neuen Rechner und damit die Akzeptanz der Rechner wesentlich steigerte. Nicht zuletzt die relativ freizügige Zugriffspolitik innerhalb der vorhandenen Netzinfrastruktur garantierte den Erfolg des Konzepts der Hochleistungsrechenzentren.
Die NASA koordinierte eine Reihe von Anwendungssoftwareprojekten für die Raumfahrt und Raumfahrtwissenschaft. Nicht zu unterschätzen ist die immense militärtechnische Bedeutung bzw. die militärtechnische Verwertbarkeit eines Großteils der Entwicklungen.
Der Erfolg von HPCC auf der Seite der Softwaretechnologie wurde durch eine geschickte Kombination unterschiedlicher Interessenlagen und Voraussetzungen der Sponsorenschaft gesichert. Während die durch die NSF vorwiegend im universitären Bereich geförderten Supercomputerzentren exzellenten Service für einen weiten Nutzerkreis bereitstellen konnten, verfügen die Softwareentwicklungszentren von DOE und NASA über wissenschaftliche und anwendungsbezogene Grundlagen und einen



Fundus hervorragend ausgebildeter Spezialisten. Frei von akademischen Geplänkeln kann damit dem interdiziplinären Charakter der Forschung wesentlich besser Rechnung getragen werden.

Tabelle 1.2: *Installationen an den Nationalen Supercomputerzentren der USA*

| Rechnertyp | Knoten | Speicher Gbyte | Leistung GFLOPS |
|---|---|---|---|
| NCSA National Center for Supercomputer Applications (University of Illinois, Urbagna-Champaign, IL, Stand 18.12.95) | | | |
| CM-5 | 512 | 16.0 | 64.0 |
| PowerChallenge | 64 (2x16, 4x8) | 16.0 | 19.2 |
| Convex Exemplar | 64 | 8.0 | 8x1.9 |
| San Diego Supercomputer Center (University of California, San Diego, CA, Stand 9/95) | | | |
| Intel Paragon | 400 | 8.4 | 30.0 |
| Cray C98 | 8 | 2.0 | 8.0 |
| Cray T3D | 128 | 8.2 | 19.2 |
| Cornell Theory Center (Cornell University Ithaca, NY, Stand 2/96) | | | |
| IBM SP2 | 512 | 80 (128 MByte ... 2 GByte) | >130 |
| Pittsburgh Supercomputer Center (Pittsburgh, PA, Stand 2/96) | | | |
| Cray T3D | 512 | 32.8 | 76.8 |
| Cray C90 | 16 | 4.0 | 16.0 |

### 1.2.3 HPCN in Japan

Japan investiert bereits seit 1960 öffentliche Mittel in dem Bereich der Informationstechnologien. Damals startete das erste nationale Projekt zur Innovation in Wissenschaft und Technologie unter Leitung des MITI. Mit Beginn des Projekts „Computer der fünften Generation“ (1982-1993, Anschlußprojekte bis 1995) wurde massiv der Einstieg in die Parallelverarbeitung vollzogen. Zunehmend wird aber ein Großteil der Investitionen im Bereich Informationstechnologie durch den privaten Sektor abgedeckt, der bis 1990 schon 85% Anteil erreicht hatte.
Die Entwicklung auf dem Gebiet der Parallelverarbeitung wird durch folgende Besonderheiten charakterisiert [115]:

- Das Datenflußmodell wird im überwiegenden Maße als Grundlage der Hardware- und der Softwareentwicklung benutzt. Die Parallelität in einer Anwendung wird aus der zeitlichen Aufeinanderfolge von Arbeitsschritten abgeleitet. Probleme der Synchronisation können mit Hilfe des Datenflusses beherrscht werden. Das Paradigma wird durchgängig sowohl für moderate Prozessorzahlen ($p \leq 256$) als auch im Bereich der massiv-parallelen ($p > 1024$) Anwendungen benutzt.

- Die ursprünglich öffentlich geförderten Projekte im nationalen Schwerpunktprogramm haben extreme Langzeitwirkung und beeinflussen mit ihren Synergieeffekten nachwievor die Entwicklungen im privaten Sektor. Eine Viel-

zahl aktueller Entwicklungsprojekte japanischer Konzerne hat signifikant vom Know-How der vorangegangenen Forschungsaktivitäten partizipiert.

- Mit der Entwicklung der Halbleitertechnologie und den Möglichkeiten, extrem leistungsstarke Prozessoren zu entwickeln, hat Japan den eigenen Weg der Vektor-Parallel-Verarbeitung (engl. *vector parallel processing* - VPP) eingeschlagen. Diese Technologie stellt einen Kompromiß zwischen dem herkömmlichen Vektorprozessor und dem vor allem in den USA entwickelten, modernen MPP-Systemen dar. Das erste kommerziell erfolgreiche VPP-System, die VPP500 kam 1993 auf den Markt und ist mit 222 Prozessoren und 355.2 GFLOPS eines der leistungsstärksten Computer auf der Welt (vgl 2.3.2).
- Der Entwicklungsansatz japanischer Parallelrechner ist immer zunächst auf Spezialaufgaben (z.B. Signalverarbeitung oder CFD) konzentriert und weniger davon geprägt, zunächst einen *general purpose* Computer zu konstruieren, für den dann Anwendungsfelder entwickelt werden, bzw. der ad-hoc für die Abarbeitung von Standardsoftware entworfen wurde.
- Japanische Firmen begreifen Parallelverarbeitung zunehmend als Schlüsseltechnologien. Als sogenannte *embedded systems* (in zukünftigen Kommunikationseinrichtungen, in intelligenten Gebäuden, zur Steuerung von Motoren oder des Autoverkehrs usw.) kommen Parallelrechner mit spezifischen Anwendungsfeldern als Bestandteil der Firmenprodukte auf den Markt.

Eine Auswahl der Entwicklungsprojekte im öffentlich geförderten und im privaten Sektor der japanischen Forschung auf dem Gebiet der Parallelverarbeitung zeigt die Tabelle 1.3.

### 1.2.4 HPCN in Europa

Im Jahre 1984 beschloß die Europäische Gemeinschaft, die Maßnahmen im Bereich der Forschung und technologischen Entwicklung (FTE) zur besseren Koordinierung in mehrjährigen Rahmenprogrammen zusammenzufassen. Das erste Rahmenprogramm (1984 - 1987) wurde daraufhin verabschiedet, gefolgt vom zweiten (1987 - 1991).
Professor Carlo Rubbia ist Physik-Nobelpreisträger und Generaldiretor des CERN. Er wurde von der Europäischen Kommision 1990 mit dem Vorsitz eines Beratergremiums zur Informationstechnologie betraut. Aufgabe des Gremiums war die Analyse der Situation auf dem Gebiet des Hochleistungsrechnen in Europa und die Erstellung von Empfehlungen für das weitere Vorgehen und den Ressourcenbedarf auf diesem Sektor der Schlüsseltechnologien, letztlich die Formulierung eines Programms zur Förderung des Hochleistungsrechnen. Mit wesentlich mehr Konsequenz als die entsprechenden nationalen Initiativen, z.B. in Großbritannien oder in Deutschland vorangebracht, stellt dieses Programm die einzige ernstzunehmende Konkurrenz zu dem HPC-Programm der USA dar.
Der (erste) Rubbia Report [178] stellt dazu fest: *Scientific and societal progress, industrial competiveness, the understanding and control of environmental factors necessary to human well-being will be governed by the availablity of adequate computing power.*
Zwei Aussagen bilden den Kern der Charakteristik der Ausgangssituation im Europa der neunziger Jahre:

- HPCN ist essentiell für die wissenschaftliche und wirtschaftliche Wettbewerbsfähigkeit Europas.
- Europa nutzt 30% der auf der Welt installierten Hochleistungsrechner und stellt nur einen verschwindend geringen Anteil davon selbst her.



Tabelle 1.3: *Parallelrechnersysteme und -projekte in Japan, Quelle: [115]*

| Einrichtung | Projekt | PE | Leistung | Anwendungsbereich |
|---|---|---|---|---|
| ETL | Coda | 64 | 3.2 GIPS | Echzeitanwendungen |
| | EM-X | 80 | 1 GIPS | data-flow testbed |
| Fujitsu | AP1000+ | 1024 | 55.2 GFLOPS | general-purpose |
| | VPP500 | 222 | 355.2 GFLOPS | Strömungssimulation |
| Hitachi | Neurocomputer | 512 | 1.26 GCUPS | Mustererkennung |
| | PIM/c | 256 | 25 MLIPS | symbolisch |
| | SR2001 | 128 | 23 GFLOPS | general-purpose |
| JUMPP | JUMP-1 | 1024 | k.A. | general-purpose |
| Matsushita | ADENART | 256 | 2.56 GFLOPS | Strömungssimulation, Graphik |
| Mitsubishi | DPP/CAP | 4096 | k.A. | Bildverarbeitung |
| | Neurochip | 12 | 1.2 GFLOPS | Dig. Signalverarbeitung |
| | PIM/m | 256 | 615 KLIPS | symbolisch |
| NEC | Cenju-3 | 256 | 12.8 GFLOPS | Strömungssimulation, general-purpose |
| | NEDIPS | 8 | 2 GFLOPS | Bildverarbeitung |
| NTT | LISCAR | 64 | 512 MOPS | Mustererkennung |
| | NOVI-II | 128 | 15.3 GFLOPS | Bildverarbeitung |
| | R256 | 256 | 500 MFLOPS | Monte Carlo Simulation |
| | SIGHT-2 | 16 | 67 MFLOPS | Raytracing |
| Oki | PIM/i | 16 | k.A. | symbolisch |
| RWC | RWC-1 | 1024 | k.A. | VR-Modellierung |
| Sanyo | Cyberflow | 64 | 640 MFLOPS | Bildverarbeitung, Strömungssimulation |
| Toshiba | PRODIGY | 128 | k.A. | Künstliche Intelligence |

Der Rubbia Report schlägt deshalb die folgenden Maßnahmen vor

- Förderung von HPC durch ein europäisches Netzwerk
- Entwicklung einer europäischen HPC Industrie
- Entwicklung europäischer Software
- Forschung und Entwicklung im HPCN-Sektor
- Aus- und Weiterbildung in HPCN

Diese Empfehlungen fanden ihren Niederschlag im dritten Rahmenprogramm der Europäische Gemeinschaft (1990 - 1994). Bestandteil dieses Programms war die Installation einer Reihe von europäischen Supercomputerzentren. Dabei wurde der Schwerpunkt auf eine Ausrüstung mit europäischer Rechentechnik (vorwiegend Systeme des deutschen Herstellers Parsytec) gelegt. Die Tabelle 1.4 gibt eine Übersicht der zur Zeit an den ZEUS-Zentren installierten Parallelrechner.
Ein Blick auf die derzeit in Deutschland installierte Rechnerleistung (Tabelle 1.5) zeigt aber, daß die leistungsstärksten Rechner dieses Ausrüstungsprogramms erst auf Platz 128 der Liste der weltweiten Installationen einkommen. Die Vorreiterrolle übernehmen auch hier System von Cray und Vektor-Parallelrechner von NEC und Fujitsu. Installationen der SP-2 von IBM halten ebenfalls Schritt.

Tabelle 1.4: *Installationen im ZEUS-Programm der Europäischen Union*

| Rechnertyp | Knoten | Speicher Gbyte | Leistung MFLOPS |
|---|---|---|---|
| IC3A Amsterdam, Niederlande (Interdisciplinary Center for Complex facilities) | | | |
| PowerXPlorer | 32 | 1 | 2000 |
| Parsytec GCel-3 | 512 | | |
| Meiko CS-1 | 64 | 1 | |
| IBM SP1 | 8 | | |
| HPC-Labor Athen, Griechenland (Athens High Performance Computing Laboratory) | | | |
| Parsytec GCel-3 | 512 | | |
| Parallab Bergen, Norwegen | | | |
| Intel Paragon XP/S | 98 | 3 | |
| Parsytec GCPowerPlus | 128 | 2 | 1024 |
| MasPar MP2 | 16k | 1 | 2500 |
| Zentrum für Paralleles Rechnen, Universität Köln | | | |
| Parsytec GCel-3 | 1024 | | 971 |
| $PC^2$ Paderborn, Deutschland Paderborn Centre of Parallel Computing | | | |
| Parsytec GCel-3 | 1024 | 4 | 4400 |
| Parsytec SuperCluster | 320 | 1.25 | 1100 |
| Parsytec GCPowerplus | 128 | 4 | 1024 |
| Parsytec PowerXPlorer | 4 | | 264 |
| IWR Heidelberg, Deutschland Institut für Wissenschafltiches Rechnen | | | |
| Parsytec PowerXPlorer | 16 | 0.5 | 1280 |
| Parsytec GCPowerplus | 196 | 6.2 | 15360 |
| TU Chemnitz, Deutschland | | | |
| Parsytec GCel-3 | 192 | | |
| Parsytec GCPowerplus | 128 | | 10000 |
| TU Dresden, Deutschland | | | |
| Parsytec PowerXPlorer | 16 | 0.5 | 1280 |
| Linköping, Schweden | | | |
| MasPar MP1 | | | |
| Parsytec GCPowerplus | 128 | | 10000 |



Derzeit läuft das vierten Rahmenprogramm (1994-1998), dessen Grundlagen und Richtlinien anläßlich der Tagung des Europäischen Rates in Edinburgh im Dezember 1992 festgelegt wurden. Die wesentliche Zielsetzung des vierten Rahmenprogramms ist es, der japanischen und amerikanischen Herausforderung im Bereich der technologischen Entwicklung zu begegnen. Europa besitzt besonders in der Intensität der Forschungsarbeiten und bei den eingesetzten Personalressourcen erheblich Defizite. In Europa werden nur 2% des Brutto-Inlands-Produktes (BIP) gegenüber 3% in Japan und den USA für Forschungsarbeiten eingesetzt. 4,3 Forscher pro 100 Erwerbstätige in Europa stehen 7,5 Forscher pro 100 Erwerbstätige in Japan und den USA gegenüber. Dieser Rückstand muß kompensiert werden. Zu diesem Zweck werden bisher außerhalb des Rahmenprogramms durchgeführte Einzelmaßnahmen im vierten Rahmenprogramm gebündelt.
Die folgenden Einzelanforderungen sind in das vierte Rahmenprogramm aufgenommen worden:

- Größere Selektivität der gemeinschaftlichen Maßnahmen zur Erhöhung des wirtschaftlichen Nutzens, insbesondere zur Ermöglichung einer neuen Offensive im weltweiten Wettbewerb für die europäische Industrie und ihre Zulieferer (Konzentration auf grundlegende Technologien)
- Verbesserung der Integration einzelstaatlicher und gemeinschaftlicher FTF-Maßnahmen und Förderung der Synergie zwischen diesen und den Strukturfonds
- Entwicklung von Synergien zwischen Forschung und Ausbildung
- Ausreichende Flexibilität der Gemeinschaftsmaßnahmen bezüglich neuer wissenschaftlicher und technologischer Herausforderungen
- Ausreichende Finanzmittel zur Erreichung der festgelegten Zielsetzungen

Die im Maastrichter Vertrag (Artikel 130g) präzisierten Maßnahmen des vierten Rahmenprogramms finden sich in sogenannten Aktionsbereichen wieder. Die Aktionsbereiche stellen die gemeinschaftlichen Maßnahmen der Europäischen Union in den Bereichen Forschung, technologische Entwicklung und Demonstration (FTE) dar. Sie sollen die für eine dauerhafte und umweltgerechte Entwicklung erforderlichen wissenschaftlichen und technologischen Grundlagen liefern. Damit werden die Wettbewerbsfähigkeit der europäischen Industrie gestärkt und die Lebensqualität erhöht.
Insgesamt vier Aktionsbereiche sind Bestandteile des Rahmenprogramms:

1. Aktionsbereich
   Durchführung von Forschungstechnologischen Entwicklungs- und Demonstrationsprogrammen durch Förderung der Zusammenarbeit mit und zwischen Unternehmen, Forschungszentren und Hochschulen.
   - Informations- und Kommunikationstechnologien
   - Industrielle Technologien
   - Umwelt
   - Biowissenschaften und -technologien
   - Nichtnukleare Energien
   - Verkehrssysterne
   - Gesellschaftliche Schwerpunktforschung

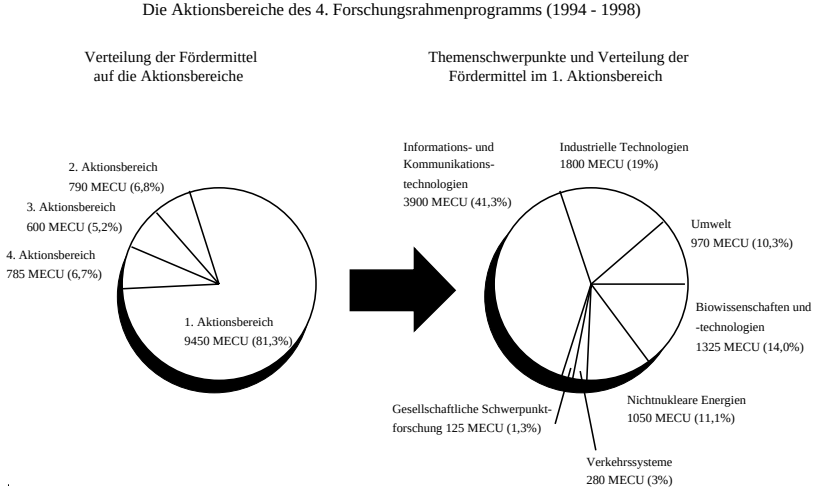


Abbildung 1.1: *Verteilung der Fördermittel im 4. Rahmenprogramm der EU*

2. Aktionsbereich
   Förderung der Zusammenarbeit mit Drittländern und internationalen Organisationen im Bereich der gemeinschaftlichen Forschung, technologischen Entwicklung und Demonstration.
   - Wissenschaftlich-technische Zusammenarbeit in Europa
   - Zusammenarbeit mit außereuropäischen Industriestaaten
   - Wissenschaftlich-technologische Zusammenarbeit mit den Entwicklungsländern
3. Aktionsbereich
   Verbreitung und Verwertung der Ergebnisse der Gemeinschaftsmaßnahmen auf dem Gebiet der Forschung, technologischen Entwicklung und Demonstration.
   - Verbreitung und Nutzung der Ergebnisse
   - Verbreitung und Einführung der Technologien bei den Unternehmen
   - Verbesserung der finanziellen Rahmenbedingungen für die Verbreitung der Technologien
   - Wissenschaftliche Dienstleistungen für die Gemeinschaft durch die GES (Gemeinschaftliche Forschungsstelle)
4. Aktionsbereich
   Förderung der Ausbildung und Mobilität von Forschern in der Gemeinschaft.
   - Ausbildung durch Forschung und Förderung der Mobilität
   - Partnerschaften zwischen Laboratorien aus unterschiedlichen Ländern
   - Zugang zu Forschungs- und Großanlagen



Tabelle 1.5: *Auszug aus den Top500 der schnellsten Rechner der Welt - deutsche Installationen, Stand November 1995 (Quelle: http://parallel.rz.uni-mannheim.de)*

| Platz | Einrichtung | Jahr | Typ | Hersteller | PE |
|---|---|---|---|---|---|
| 38 | ZIB | 95 | T3D SC256 | Cray | 256 |
| 62 | DKRZ | 95 | Y-MP C916 | Cray | 16 |
| 95 | Max-Planck-Gesellschaft | 95 | T3D MCA128-8 | Cray | 128 |
| 100 | Leibniz Rechenzentrum | 95 | SP2/70 | IBM | 70 |
| 105 | DLR | 94 | SX-3/24R | NEC | 2 |
| 108 | VW (Volkswagen AG) | 95 | SX-3/24R | NEC | 2 |
| 112 | DLR | 95 | SP2/60 | IBM | 60 |
| 128 | IWR, U Heidelberg | 95 | GC PP/192 | Parsytec | 192 |
| 129 | $PC^2$, U Paderborn | 95 | GC PP/192 | Parsytec | 192 |
| 134 | PIK | 94 | SP2/43 | IBM | 43 |
| 166 | GMD | 95 | SP2/37 | IBM | 37 |
| 172 | U Marburg | 95 | SP2/35 | IBM | 35 |
| 202 | RWTH Aachen | 93 | VPP500/4 | Fujitsu | 4 |
| 203 | TH Darmstadt | 94 | VPP500/4 | Fujitsu | 4 |
| 214 | TU Chemnitz | 94 | GC PP/128 | Parsytec | 128 |
| 215 | U Hamburg-Harburg | 94 | GC PP/128 | Parsytec | 128 |
| 243 | KFA Jülich | 94 | XP/S10 | Intel | 144 |
| 248 | DKFZ | 95 | SP2/24 | IBM | 24 |

Die eingeplanten Fördermittel zur Durchführung des vierten Rahmenprogramms belaufen sich auf insgesamt 11.625 Millionen ECU. Die Aufschlüsselung dieses Gesamthöchstbetrages auf die vier vorgenannten Aktionsbereiche sind in Abbildung 1.1 dargestellt (nach [176]).

## 1.3 Zukunft des parallelen Hochleistungsrechnen

Viele große Computerzentren verfügen heute über einen oder mehrere MPP-Rechner. Die MPP-Systeme bedeuten einen wesentlichen technologischen Schritt über die bisher dominanten Vektorcomputer hinaus. Der Trend hin zu MPP-Systemen ist deutlich, läßt allerdings bisher keine eindeutigen Prognosen zu. Die Hersteller bieten Hardware mit Standard-RISC oder -Vektorprozessoren als Rechenknoten an, die über unterschiedliche Konzepte eines Verbindungsnetzwerkes (MIMD) oder gemeinsamen Speicher (SIMD) kommunizieren können. Skalierbare Ein- und Ausgabefunktionalität, parallele Betriebssysteme und die dazugehörigen Entwicklungswerkzeuge sind vorhanden oder werden entwickelt. Es ist klar, daß nur die Parallelität auch in der Zukunft den notwendigen Zuwachs an Rechenleistung garantiert. Allerdings ist nicht absehbar, ob die MPP-Systeme tatsächlich für alle Klassen von Anwendungsproblemen geeignet sind und ob für alle Klassen von Anwendungsproblemen der immense Aufwand der Neuentwicklung von speziell für MPP-Systeme geeigneter Software adäquat ist.
Trotz aller Anstrengungen, Standards für das parallele Rechnen zu entwickeln (PVM [51], MPI [68]) sind heute erst wenige Produkte des kommerziellen Softwaremarktes parallelrechnerbasiert. Das ist zum Teil in der initial übergroßen Anzahl von verschiedenartigen Softwareumgebungen und den Parallelisierungstools begründet. Die EU unternimmt hier mit der Schwerpunktlegung im EUROPORT-Projekt des ESPRIT III Programms einen Versuch, die in der Hardwareentwicklung verlorene Position mit der Bereitstellung von kommerzieller Software für Parallelrechner wie-

derzuerringen. Neben der Parallelisierung existierender und etablierter Codes liegt ein Schwerpunkt auf der Entwicklung von parallelisierenden Compilern und von Entwicklungswerkzeugen.
Alternativ zur Parallelisierung für Systeme mit verteiltem Speicher bietet sich der weiche Übergang auf Parallelrechnerhardware vom Typ SIMD, mit einigen wenigen, dafür leistungsfähigen Prozessoren an. Als Software setzt man dabei *High Performance Fortran* (HPF) ein. Mit dem in HPF realisierten datenparallelen Ansatz, der von der tatsächlichen physikalischen Struktur eines Rechners und Speicherung der Daten abstrahiert und der eine zum Großteil automatisierbare Parallelisierung erlaubt, werden aber sowohl die moderat parallelen Systeme vom Typ SIMD als auch die modernen MPP-Systeme mit Hochleistungskommunikationshardware effizient programmierbar. Es bleibt abzuwarten, ob dieser wesentlich leichter zugängliche Weg über HPF erstens zur Renaissance des Klassikers FORTRAN führt und zweitens die Softwareinvestitionen sichern und auf modernster Hardwarebasis etablierte kommerzielle Codes verfügbar machen wird. Nicht zu unterschätzen ist der Aufwand für die Parallelisierung von Algorithmen, die auf unstrukturierten Daten basieren. Diese Algorithmen mit ihren dynamischen Datenstrukturen lassen sich nicht durch Compilerdirektiven parallelisieren, sondern erfordern explizit Verwaltungsdatenstrukturen für die Parallelverarbeitung.
Nach einer Phase der Euphorie und des Neubeginns der Softwareentwicklung für MIMD-Parallelrechner stehen wir zur Zeit in einer Phase der relativen Ernüchterung und Rückbesinnung.

# Kapitel 2

# Grundlagen der Parallelverarbeitung

Parallelrechner stehen in der Linie der konsequenten Weiterentwicklung der modernen Rechnerarchitekturen in der vordersten Reihe. Beginnend mit den ersten Rechenmaschinen, die bereits im 18. Jahrhundert von Leibniz entwickelt wurden, erlaubten im neunzehnten Jahrhundert die mechanischen Rechenmaschinen, z.B. von Babbage (1791-1871) und Mitte des zwanzigsten Jahrhunderts zuerst die elektrischen und dann die elektronischen Rechenmaschinen als serielle Rechenanlagen einen einzelnen, gesteuerten Operationsablauf mit einem Datenfluß und einer Anweisungsfolge. Dabei gehört die Entwicklung der ersten vollautomatischen, programmgesteuerten und frei programmierbaren, auf dem Dualsystem basierenden Rechenanlage 1941 durch den deutschen Ingenieur Konrad Zuse (1910-1995) zu den herausragenden technischen Leistungen unseres Jahrhunderts.
Der Weg führt dann über die Entwicklung von Halbleiterbauelementen und ihrer Integration in Schaltkreisen mit immer weiter zunehmender Dichte. Man spricht heute von der vierten Generation der Computer. In Japan läuft das zukunftsweisende Programm zur Entwicklung der Computer der fünften Generation. Das Potential an Zuwachs der Rechenleistung wird im zunehmenden Maße von den technologischen und physikalischen Grenzen der Miniaturisierung der beherrschbaren Strukturen, den werkstofftechnologischen Grenzen, den technischen Möglichkeiten der Ausnutzung der Supraleitfähigkeit und der Lichtgeschwindigkeit der Signalausbreitung begrenzt. Die Ausnutzung von Parallelität in der Applikation durch entsprechende Parallelität in den Verarbeitungseinheiten, den Prozessoren selbst, ist deshalb ein erfolgreicher Weg, diesen Engpaß in der Entwicklung leistungsfähiger Rechenanlagen zu umgehen. Dazu kommt, daß bestimmte Anwendungen mit einem natürlichen Potential an Parallelität auf paralleler Hardware am besten abgearbeitet werden können. Das wurde bereits bei der Entwicklung der ersten Rechenmaschinen berücksichtigt. Die von Babbage entwickelte *Analytical Engine* wurde zum Beispiel so beschrieben:
*When a long series of identical comutations is to be performed, such as those required for the formation of numerical tables, the machine can be brought into play so as to give several results at the same time, which will greatly abridge the whole amount of the processes.*[1]
Inzwischen gibt es eine unübersehbare Vielzahl von Realisierungen des Parallelrechnerparadigmas.
Aufbauend auf einer Klassifikation der parallelen Rechnerarchitekturen werden deshalb für die Parallelrechner mit verteiltem Speicher die Grundelemente der Archi-

[1] General L.F. Menabrea in einem Aufsatz über die von Charles Babbage erfundene *Analytical Engine*, 1842

tektur und des Verbindungsnetzwerkes dargestellt. Ergänzt wird die theoretische und klassifizierende Beschreibung mit einer exemplarischen Gegenüberstellung von typischen Vertretern der genannten Architekturen. An die Klassifikation der Hardware schließt sich eine Beschreibung der Programmiermodelle an. Die Darstellung der Programmiermodelle konzentriert sich auf die Beschreibung des Modells der kommunizierenden Prozesse, dessen Umsetzung in der Laufzeitumgebung PARIX und der Realisierung in der portablen Programmierschnittstelle MPI.
Abschließend werden die mathematischen und systemtheoretischen Grundlagen für die Leistungsbewertung, die Effizienzbetrachtung und die Beurteilung der Skalierbarkeit und damit die Evaluierung des Nutzens, der aus der Parallelisierung einer bestimmten Applikation für einen konkreten Parallelrechner gewonnen werden kann, dargestellt.

## 2.1 Architektur von Parallelrechnern

Aufbauend auf einer eingehenden Darstellung verschiedener Klassifikationsschemata für Rechnerarchitekturen werden dann die beiden im Mittelpunkt der Darstellung von Anwendungen des parallelen Rechnens stehenden Systemarchitekturen WorksationCluster und Multiprozessorsysteme mit verteiltem Speicher näher vorgestellt. Die Klassifikationen werden nach verschiedenen, zumeist pragmatischen Gesichtspunkten entwickelt.

### 2.1.1 Klassifikation

Die Grundlage der modernen Rechnerarchitekturen ist das Modell des von-Neumann-Rechners[2]. Der von-Neumann-Rechner besteht aus Verarbeitungseinheit mit Steuer- und Arithmetikeinheit, Ein- und Ausgabemodul und Hauptspeicher. Der Rechner arbeitet nach den folgenden Prinzipien:

- Die Verarbeitungseinheit ist über einen Kommunikationsbus mit dem Hauptspeicher verbunden.
- Die Zellen des Hauptspeichers haben eine feste Länge (Wortlänge) und sind linear organisiert.
- Die Arithmetikeinheit realisiert elementare Operationen auf elementaren Operanden.
- Die Kontrolleinheit arbeitet sequentiell einen Anweisungsstrom ab.

Dieser Rechner verhält sich streng deterministisch. Aus einem Eingabedatenstrom wird ein genau definierter Ergebnisdatenstrom erzeugt. Die dazu notwendigen Operationen werden mit dem Anweisungsstrom definiert. Aus den Funktionseinheiten des von-Neumann-Rechners kann man Parallelrechner zusammensetzen. Dazu sind einzelne oder auch alle Funktionseinheiten zu vervielfachen. Eine Klassifikation von verschiedenen Parallelrechnern auf dieser Basis gibt Flynn [67].

**SISD** *single instruction single data*
: Der klassische von-Neumann-Rechner mit einer Verarbeitungseinheit, einem Daten- und einem Anweisungsstrom. SISD-Rechner können nur eine Anweisung pro Zeiteinheit auf einer Dateneinheit ausführen. Die Übertragungsgeschwindigkeit zwischen Speicher und Verarbeitungseinheit begrenzt die Leistungsfähigkeit dieses Rechners.

[2] benannt nach John von Neumann (1903-1957)



**SIMD** *single instruction multiple data*
: Ein Rechner mit parallel arbeitenden Verarbeitungseinheiten, die synchron mit einem Anweisungsstrom gesteuert werden und getrennte Datenströme verarbeiten. Dieser Rechner besitzt einen Programmspeicher. Der Datenspeicher ist logisch partitioniert und den Verarbeitungseinheiten zugeordnet. Die Verarbeitungseinheiten können aber direkt auf den gesamten Datenspeicher zugreifen. Vektorrechner und Feldrechner sind in dieses Konzept einzuordnen.

**MISD** *multiple instruction single data*
: Ein Rechnermodell, daß mit unterschiedlichen Verarbeitungsanweisungen aus einem Eingabedatenstrom einen Ausgabedatenstrom erzeugt, das Potential der Parallelität also mit dem Pipelineeffekt ausnutzt.

**MIMD** *multiple instruction multiple data*
: Mehrfache Datenströme und mehrfache Verarbeitungseinheiten arbeiten parallel eigene Anweisungsströme ab. Die Verbindung zwischen den für sich selbständigen von-Neumann-Rechnern wird mit einem Verbindungsnetzwerk oder mit einem gemeinsamen Speicher realisiert.

Diese sehr einfache Klassifikation geht nicht auf technische Details der Speicherorganisation von SIMD-Rechnern oder auf die Verbindung der Prozessorelemente von MIMD-Rechnern ein. Es gibt daher eine Reihe anderer Klassifikationen, um die Vielzahl der technischen Lösungsmöglichkeiten in eine geeignete Taxonomie einzuordnen, z.B.

**Klassifikation nach Shore** [189]
: Shore klassifiziert die Rechner nach der logischen Anordnung der vier Basiskomponenten Steuereinheit, Arithmetikeinheit, Datenspeicher und Befehlsspeicher. Die Speichereinheiten werden als Primärspeicher oder Hauptspeicher zusammengefaßt, im Gegensatz zu Sekundärspeicher (als Massenspeicher, wie z.B. Festplattensysteme). Shore bildet aus diesen vier Basiskomponenten durch unterschiedliche Anordnungen sechs verschiedene Rechnertypen.

**Erlanger Klassifikationssystem** [91] (engl. *Erlangen classification system* ECS)
: ECS klassifiziert Rechner nach den Eigenschaften von Steuereinheiten, arithmetischen oder logischen Funktionseinheiten und der Wortlänge. Charakterisiert werden diese Eigenschaften nach absoluter Anzahl und der Anzahl der gleichzeitig unabhängig, d.h. parallel arbeitenden Einheiten.

Im folgenden soll ein Rechner im Mittelpunkt der Betrachtung stehen, der folgende Eigenschaften hat

- Der Rechner besteht aus einer Anzahl von eigenständigen Prozessoren.
- Die Prozessoren haben ausschließlich Zugriff auf lokalen Speicher, d.h. auf einen eigenen Daten- und Anweisungsstrom.
- Alle Prozessoren sind in ihrer Funktion gleichberechtigt, es gibt keinen bevorzugten Kontrollprozessor.
- Die Prozessoren verfügen über ein Verbindungsnetzwerk, daß es erlaubt, Nachrichten zwischen zwei beliebigen Prozessoren auszutauschen.
- Das natürliche Programmiermodell des Rechners ist Nachrichtenaustausch (engl. *message passing*).

Dieser Rechner ist vom Typ MIMD. Falls die Prozessoren ein identisches Programm abarbeiten, kann die Funktionsweise des beschriebenen MIMD-Rechners auch auf einem SIMD-Rechner nachempfunden werden. Allerdings ist Message Passing nicht unbedingt das effizienteste Programmiermodell für SIMD-Rechner (z.B. PowerChallenge von Silicon Graphics Inc.). Umgekehrt ist mit einem MIMD-Rechner die Simulation eines SIMD-Rechners möglich, wenn der Zugriff auf externen Speicher in geeigneter Weise mit den Mitteln des Nachrichtenaustauschs realisiert wird. Auf dem MIMD-Rechner Cray T3D stehen z.B. zur Realisierung des Konzepts des virtuellen gemeinsamen Speichers hochgradig optimierte Hardware- und Softwarelösungen zur Verfügung, um den gegenseitigen Zugriff der Prozessorelemente auf den jeweils lokalen Speicher transparent, sicher und schnell zu erlauben. Auf dieser Hardwareplattform sind die expliziten Speicherzugriffsroutinen effizienter implementiert, als die Funktionen zum Nachrichtenaustausch im Verbindungsnetzwerk.
Zwei wesentliche Vertreter der MIMD-Rechner sind Cluster und Multiprozessornetzwerke. Die Eigenschaften dieser beiden Architekturen sollen deshalb im Hinblick auf ihre Eignung für die Parallelisierung von numerischen Berechnungsverfahren diskutiert werden.

### 2.1.2 Cluster

Ein Cluster ist eine logische Einheit von eigenständigen Rechnern. Die Rechner sind über ein LAN (engl. *local area network*), WAN (engl. *wide area network*) oder ein speziell konfigurierbares Netzwerk verbunden. Es ist möglich, über Dienste des Betriebssystem (i.d.R. UNIX-Derivate) Prozesse auf einem beliebigen Rechner im Cluster zu starten und die Kommunikation zwischen den Prozessen zu organisieren. Die Prozessoren der Rechner können unterschiedliche Leistungsfähigkeit haben, bedingt durch Art und Typ des Rechners, Speicherausbau usw. Man spricht dann von einem heterogenen Cluster. Auf jedem Rechner läuft ein vollständiges Betriebssystem und in der Regel befindet sich der Rechner im Mehrbenutzerbetrieb (engl. *multi user*). In einem Cluster ist deshalb die zur Verfügung stehende Rechenleistung sowohl von der Anzahl und der Ausstattung der tatsächlich einbezogenen Rechnern (statische Komponente) als auch von der aktuellen Belastung (dynamische Komponente) dieser Rechner abhängig.
Die Kommunikation zwischen Prozessen auf unterschiedlichen Rechnern bedient sich eines Netzwerkprotokolls (ISO-Schichtenmodell). Damit ist der Datenaustausch zwischen den Rechnern, die nicht die gleiche Binärrepräsentation von Daten benutzen, sicher und transparent. Die Umwandlung der Datenrepräsentation in ein netzwerkunabhängiger Format wird mit dem XDR- (engl. *external data representation*) Protokoll realisiert.
Die Standard-Netzwerktechnologie ist Ethernet. Ethernet ist ein Bussystem. Die kommunizierenden Rechner teilen sich die verfügbare, physikalisch begrenzte Bandbreite. Bei gleichzeitiger Anforderung von Kommunikationsleistung von mehreren Rechnern reduziert sich die effektive Datenübertragungsrate zwischen zwei Prozessen und die Transferzeit erhöht sich. Das Zeitverhalten der Kommunikation ist somit nicht eindeutig bestimmbar. Es läßt sich lediglich das ideale Zeitverhalten als oberer Grenzwert der Leistungsfähigkeit angeben. Mit hierarchisch organisierten Bussystemen, in denen jeder Rechner Zugriff auf mindestens zwei Netzwerkebenen hat, kann dieser Engpaß teilweise umgangen, aber nicht eliminiert werden.
Hardwaretechnisch ist ein Bussystem als Netzwerk in einem Parallelrechner ideal. Jede Komponente, die auf den Bus zugreift, muß dafür lediglich über genau einen Anschluß verfügen. Das System ist deshalb einfach um weitere Komponenten erweiterbar.
Das Problem von Clustern ist neben der Bandbreite der Datenübertragung die Startzeit einer Kommunikation. Werden für die Kommunikation UNIX-Dienste benutzt, ist der Aufwand zur Initialisierung einer Kommunikation wesentlich höher, als beim Aufsetzen der Kommunikation in einem speziellen Multiprozessornetzwerk.



### 2.1.3 Multiprozessorsysteme

Multiprozessorsysteme bestehen aus gleichartigen Prozessoren. Im Hinblick auf die zur Verfügung stehende Rechenleistung sind sie deshalb mit homogenen Clustern vergleichbar. Der Datenaustausch zwischen Prozessen auf unterschiedlichen Prozessoren kann sehr effizient implementiert werden, weil keine Umwandlung der Datenrepräsentation in ein netzwerkunabhängiger Format erforderlich ist. Die Transparenz der Datenrepräsentation auf allen Prozessoren ist a-priori gegeben.
Das Multiprozessorsysteme besteht neben den Prozessoren aus dedizierten Linkverbindungen. Moderne Multiprozessornetzwerke besitzen eine feste Topologie. Typische Topologien für Verbindungsnetzwerke sind

- Hypercube (nCube, nCube Inc.),
- Gitter (Paragon XP, Intel Corp., PowerGC, Parsytec Computer GmbH),
- Torus (Cray T3D, Cray Research Inc.) oder
- Baum (CM-5, Thinking Machines, als sogenannter *Fat Tree*).

Der Vorteil von Netzwerken mit fester Topologie ist die hohe Effizienz des Routingalgorithmus, der zur Kommunikation zwischen zwei Prozessen auf beliebigen Prozessoren benötigt wird. Im Vergleich zu frei konfigurierbaren Netzwerken (SuperCluster, Parsytec Computer GmbH) besitzen fest konfigurierte Netzwerke deshalb die Vorteile der

- hoch optimierten Routingalgorithmen,
- einfachen Verbindungshardware und
- mit der Prozessorzahl skalierbaren Leistungsfähigkeit.

Die mit der Prozessorzahl skalierbaren Leistungsfähigkeit ist der entscheidende Vorteil von Multiprozessornetzwerken gegenüber den Netzwerken von Clustern. Die Kommunikation zwischen zwei Prozessen kann die ideale Datenübertragungsrate erreichen und gleichzeitig können andere Kommunikationen auf anderen physikalischen Verbindungen aktiv sein. Die global nutzbare Übertragungsrate vervielfacht sich damit. Solche Kommunikationsmuster (Belegung physikalischer Links durch unterschiedliche logische Verbindungen in kollektiven Kommunikationen, vgl. Abs. 2.6) können am besten auf Multiprozessornetzwerke mit fester Topologie implementiert werden.
Das Zeitverhalten eines Programms auf diesem Rechner ist deterministisch, Einbenutzerbetrieb vorausgesetzt. In der Regel ist das mit dem Laufzeitsystem des Parallelrechners gesichert. Dieses Laufzeitsystem teilt einem Nutzer eine bestimmte Partition, d.h. einen Pool von Prozessoren zur exklusiven Nutzung zu.
Alternativ ist ein UNIX-ähnliches Betriebssystem (z.B. das SunOS-Derivat für die *Connection Machine* CMOST) in der Lage, neben der Partitionierung auch ein Zeitscheibenverfahren zu realisieren, daß einen Mehrbenutzerbetrieb ermöglicht. CMOST sieht aber die Vergabe von Prioritäten vor, so daß die Auslastung des Parallelrechners administriert werden kann. Damit zeigt ein Multiprozessorsystem auch im Mehrbenutzerbetrieb ein deterministisches Verhalten.

## 2.2 Verbindungsnetzwerke

Die Topologie von Netzwerken läßt sich mit Hilfe von Graphen beschreiben. Prozessoren werden mit Knoten und Verbindungen zwsichen Prozessoren mit Kanten dargestellt. Ein formales Werkzeug zur Definition und zur Beschreibung von Eigenschaften stellt die Graphentheorie dar, die im folgenden mit den benötigten Begriffen dargestellt wird. Weitere Details können der umfangreichen Standardliteratur entnommen werden.
Danach werden freikonfigurierbare und statische Netzwerke vorgestellt. Freikonfigurierbare Netzwerke werden in Clustern und in Multiprozessorsystemen eingesetzt. Die Mehrzahl der aktuellen Multiprozessorsystemen benutzt aber statische Verbindungsnetzwerke. Auf diesen Verbindungsnetzwerken muß ein Routingssystem die Punkt-zu-Punkt-Kommunikation unterstützen. Die wesentlichen Routingverfahren werden am Ende dieses Abschnitts vorgestellt.

### 2.2.1 Grundlagen der Graphentheorie

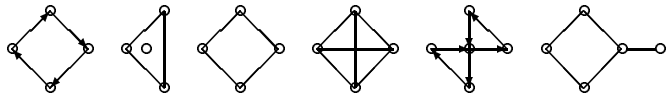


Abbildung 2.1: *Graphen*

Ein einfacher Graph $G$ besteht aus einer endlichen Zahl von Knoten (engl. *vertex*) und einer endlichen Zahl von Kanten (engl. *edge*), die jeweils genau ein Paar von Knoten verbinden. Wenn eine Kante durch ein geordnetes Paar von Knoten beschrieben wird, dann kann man den Graph $G$ auffassen als das Paar $< V(G), E(G) >$, wobei $V(G)$ eine Menge von Knoten und $E(G)$ eine Menge von zweielementigen, geordneten Untermengen von $V(G)$ ist. Ist die Menge $E(G)$ leer, spricht man von einem Nullgraphen. Wir bezeichnen die Kante, die Knoten $v_1$ mit Knoten $v_2$ verbindet, mit $v_1v_2$. Die Größe $|G|$ bezeichnet die Anzahl der Knoten des Graphen $G$. Abbildung 2.1 zeigt vier Graphen mit Größe 4 und zwei Graphen der Größe fünf.
In Abbildung 2.1 sind elementare Graphen angegeben. Diese Graphen zeichnen sich dadurch aus, daß kein Knoten mit sich selbst verbunden ist und keine zwei Knoten durch mehr als eine Kante direkt verbunden sind. Der Graph $G$ ist der *vollständige Graph* $K_n$, wenn $n = |G|$ und alle Verbindungen zwischen allen Knoten bestehen. In Abbildung 2.1 ist z.B. $G_4$ der vollständige Graph $K_4$. Die Anzahl der Kanten, die an einen Knoten anschließen, bezeichnet man als Knotengrad. So haben z. B. in $G_5$ 4 Knoten den Grad 2 und ein Knoten hat Grad 4. In einem vollständigen Graphen $K_n$ haben alle $n$ Knoten den Grad $n-1$. Haben alle Knoten den gleichen Grad, spricht man von einem *regulären* Graphen.
In einem Graph ist eine Folge von $n$ Knoten $v_1, \ldots v_n$, für die $v_1v_2, \ldots, v_{n-1}, v_n$ eine Folge von $n-1$ Kanten ohne Wiederholung ist ein *Pfad* der Länge $n-1$. Der Pfad ist geschlossen, wenn $v_1 = v_n$. Ist auch die Knotenfolge ohne Wiederholung, spricht man von einer Kette. Die Kette ist geschlossen, wenn der Spezialfall $v_1 = v_n$ gilt. Eine geschlossene Kette mit mindestens 3 Knoten ist ein *Kreis*.
Ein *planarer Graph* kann in einer Ebene mit allen Knoten und Kanten dargestellt werden, ohne daß zwei Kanten sich schneiden. Ein Graph ist *zusammenhängend*, wenn für jedes Paar von Knoten ein Pfad existiert. Alle Graphen in Abbildung 2.1, bis auf Graph $G_2$, sind zusammenhängend. Der *Durchmesser* eines zusammen/-hängenden Graphen gibt die größte Pfadlänge für die jeweils kürzeste Verbindung



zwischen zwei Knoten in einem Graphen an. In Abbildung 2.1 hat Graph $G_4$ Durchmesser 1, die Graphen $G_1, G_3$ und $G_5$ haben $G_6$ hat Durchmesser 3. Der Durchmesser von Graph $G_2$ ist nicht definiert - $G_2$ ist kein zusammenhängender Graph.
Kann man für einen Graphen $G_n$ mit $n$ Knoten einen geschlossenen Pfad der Länge $n-1$ angeben, dann spricht man von einem Euler-Graph. In solchen Graphen lassen sich alle Kanten ohne Wiederholung traversieren und man kehrt wieder zum Ausgangspunkt zurück. Kann man eine Kette der Länge $n-1$ angeben, spricht man von einem Hamilton-Graph. Jeder Graph, der aus einem Kreis ohne weitere Knoten oder Kanten besteht, ist sowohl Euler- als auch Hamilton-Graph. In Abbildung 2.1 sind der Hamilton-Pfad mit dickeren Kanten verdeutlicht und der Euler-Pfad durch Pfeile angegeben (nicht zu verwechseln mit den später eingeführten gerichteten Kanten). Ein Beispiel dafür aus Abbildung 2.1 ist Graph $G_1$. Die Graphen $G_3$ und $G_4$ sind Hamilton-Graphen, aber keine Euler-Graphen, während $G_5$ Euler-Graph, aber nicht Hamilton-Graph ist. $G_6$ ist weder Euler- noch Hamilton-Graph. Ein zusammenhängender Graph ist ein Euler-Graph dann und nur dann, wenn alle Knoten einen geradzahligen Grad haben. Ist für jeden Knoten in einem zusammenhängenden Graphen $G_n$ der Grad größer als $n/2$, dann und nur dann ist $G_n$ ein Hamilton Graph.
Zwei Graphen sind *isomorph*, wenn man zwischen ihnen eine eineindeutige Abbildung (Isomorphismus) in der Weise finden kann, daß für jeweils zwei Knoten, die in dem einen Graphen durch eine Kante verbunden sind, ein entsprechendes Knotenpaar im anderen Graphen gegeben ist. Anschaulich bedeutet das, daß man für beide Graphen identische Darstellungsformen wie z.B. in Abbildung 2.1 angeben kann.
Ein Graph ist *automorph*, wenn man einen Isomorphismus mit dem Graphen selbst angeben kann, der vom identischen Automorphismus (Abbildung des Graphen auf sich selbst) verschieden ist. So können zum Beispiel im Graphen $G_1$ die Knoten $pqrs$ sowohl auf die Knoten $psrq, qrsp, qpsr, rspq, rqps, spqr$ oder $srqp$ als auch auf $pqrs$ abgebildet werden. Das sind insgesamt acht Automorphismen. Darstellungen der automorphen Repräsentation können durch Permutationen des Ausgangsschemas gewonnen werden. Ein Graph mit einer hohen Zahl von Automorphismen weist einen hohen Grad an Symmetrie auf. Ein vollständiger Graph besitzt $n!$ Automorphismen.

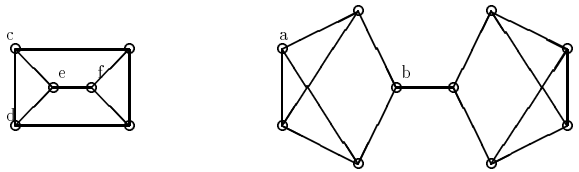


Abbildung 2.2: *Homogene und nichthomogene Graphen*

Kann für einen Graphen ein Automorphismus angegeben werden, der keinen Knoten auf sich selbst abbildet, spricht man von einem homogenen Graphen. Anders ausgedrückt: hat ein Graph die Eigenschaft, daß sich für jedes Knotenpaar ein Automorphismus angeben läßt, der Knoten $v$ auf Knoten $w$ abbildet und umgekehrt Knoten $w$ auf Knoten $v$, dann ist der Graph homogen. Anschaulich sieht ein solcher Graph von jedem Knoten aus gleich aus. Homogene Graphen sind $G_1$ und $G_4$ in Abbildung 2.1 als auch $G_7$ in Abbildung 2.2. Ein homogener Graph muß gleichzeitig auch ein regulärer Graph sein. Alle Knoten haben den gleichen Grad. Diese Bedingung ist notwendig, aber nicht hinreichend. Als Beispiel dient $G_8$ in Abbildung 2.2. Dieser Graph ist zusammenhängend und regulär, aber nicht homogen. - Es läßt sich

kein Automorphismus für die Knoten $a$ und $b$ angeben. Ein geeigneter Schnitt durch eine Kante, die von $b$ ausgeht, zerlegt den Graphen in zwei nichtzusammenhängende Teile, das ist aber nicht für den Knoten $a$ gültig.

Die Homogenität eines Graphen, die bisher auf der Basis von Knotenabbildungen erläutert wurde, läßt sich auch für die Abbildung von Knotengruppen auf andere Knotengruppen angeben. In Abbildung 2.1 ist das für den Graphen $G_1$ möglich. Faßt man die Knoten $r$ und $s$ einerseits und die Knoten $p$ und $q$ andererseits zu zwei Gruppen zusammen, läßt sich auf einer höheren Stufe Homogenität nachweisen. Man kann die erste Gruppe auf die zweite abbilden. Anders ist die Situation bei Graph $G_7$ in Abbildung 2.2. Das Paar $(c, d)$ kann nicht geeignet auf das Paar $(e, f)$ abgebildet werden, weil ersteres der Teil eines Dreieckes ist, was zweiteres nicht ist.

Wenn in einem Graph die Verbindung zwischen den Knoten $v_1$ und $v_2$ - $v_1 v_2$ von der Verbindung $v_2 v_1$ unterschieden wird, wenn also nur eine Verbindung in einer Richtung erfolgt, dann spricht man von einem gerichteten Graphen. Die bisher erläuterten Eigenschaften ungerichteter Graphen lassen sich sinngemäß erweitern und auf gerichtete Graphen anwenden.

### 2.2.2 Freikonfigurierbare Netzwerke

Freikonfigurierbare Verbindungsnetzwerke erlauben es, eine Nachricht von einem beliebigen Prozessor zu einem beliebigen anderen Prozessor effizient mit Hardwareunterstützung zu versenden. Mit steigender Leistungsfähigkeit aber auch mit steigenden Kosten (vergleiche [5], S. 394) unterscheidet man

- Busverbindung,
- hierarchische Verbindungsnetzwerke und
- Kreuzverbindung

die im folgenden dargestellt werden.

#### 2.2.2.1 Busverbindung

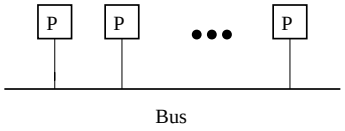


Abbildung 2.3: *Bussystem*

Die Busverbindung ist eine billige und einfach um weitere Prozessoren zu erweiternde Topologie (siehe Abbildung 2.3). Ein Bussystem mit fester Übertragungsbandbreite erlaubt es einer Anzahl von Prozessoren nacheinander ihre Informationen auf den Bus zu geben. Je mehr Prozessoren den Bus benutzen, umso länger wartet jeder Prozessor, bis seine Nachricht tatsächlich übertragen werden kann. Die Übertragungsleistung ist nicht skalierbar.

Mit lokalen Cachespeichern können Nachrichten zwischengepuffert werden. Damit kann in Abhängigkeit von der Berechnungsaufgabe ein Teil der Wartezeit überbrückt werden.



#### 2.2.2.2 Hierarchische Verbindungsnetzwerke

Es gibt eine Vielzahl zum Teil topologisch äquivalenter hierarchischer Verbindungsnetzwerke, die aus einer Anzahl von meist $2 \times 2$ Kreuzverbindungsbausteinen aufgebaut sind (vergleiche zum Beispiel [221]). Diese Kreuzverbindungsbausteine zur Verbindung von $n$ Eingabe- und Ausgabekanälen sind in einem Gitter der Komplexität $O(n) \times O(log_2 n)$ angeordnet. Alle Netzwerke können jeden Anfangsknoten mit jedem Endknoten verbinden. Dabei wird eine feste Anzahl von Verbindungsbausteinen benutzt. Unterschiede bestehen in den Möglichkeiten, dabei Nachrichten zu permutieren und zusammenzufassen.

Als ein Vetreter der Klasse der hierarchischen Verbindungsnetzwerke soll das sogenannte *Omega*-Netzwerk [133] vorgestellt werden. Es ist ein ökonomisches Verbindungsnetz, daß genau einen Verbindungsweg zwischen einem Paar von Ein- und Ausgangsknoten besitzt.

Abbildung 2.4: *Vier Schaltfunktionen des Omega-Kreuzverbinders: Durchgang, Kreuzung, unterer und oberer Broadcast*

Das Original-Omega-Netzwerk besteht aus $2 \times 2$ Kreuzverbindern. Diese Kreuzverbinder erlauben vier verschiedene Schaltfunktionen, die in Abbildung 2.4 verdeutlicht sind: Durchgang, Kreuzung, unterer und oberer Broadcast.

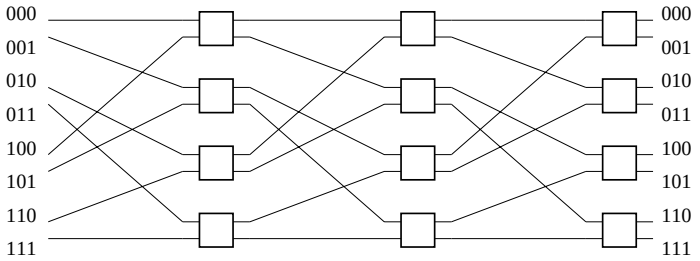


Abbildung 2.5: *Omega-Netzwerktopologie*

Das Verbindungsnetzwerk der $n/2 \times log_2 n$ Schaltelemente ist in Abbildung 2.5 dargestellt. Zwischen zwei Spalten von Elementen ist jeweils eine *perfect shuffle* Verbindung geschaltet. Die Verbindung hat diesen Namen, weil mit ihr alle Elemente der oberen Hälfte des Stapels jeweils genau zwischen die Elemente des unteren Stapels gemischt werden: Ein Eingangsknoten $i$ ist mit dem Ausgangsknoten $M(i)$ verbunden, so daß gilt

$$M(i) = \begin{cases} 2i & \text{für} \quad i < \frac{n}{2} \\ 2i - n + 1 & \text{für} \quad i \geq \frac{n}{2} \end{cases} \tag{2.1}$$

Mit $log_2 N$ Spalten besitzt das Omega-Netzwerk $log_2 N$ Schaltebenen.

Nachrichten werden auf diesem Netzwerk vorzugsweise als Paket mit Kennung und Datum verschickt. Initial ist die Kennung gleich der Binärrepräsentation des Ausgangsknotens. In jeder Spalte $i$ des Verbindungsnetzwerkes wird die Bitposition $i$ der Kennung ausgewertet. Liefert diese Auswertung eine 1 bzweine 0 wird die Nachricht entsprechend auf dem unteren bzw. oberen Kanal des Kreuzverbinder weitergeleitet.

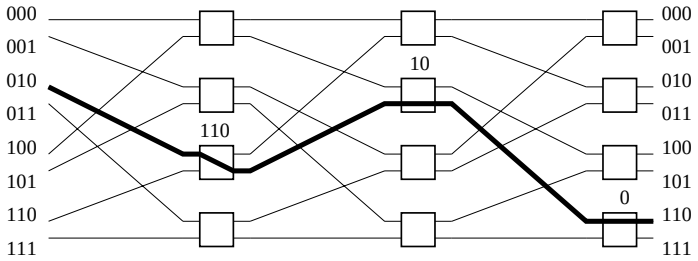


Abbildung 2.6: *Routing einer Nachricht von Knoten* $2 = (010)_2$ *zu Knoten* $6 = (110)_2$ *im Omega-Netzwerk*

In Abbildung 2.6 ist der Transportweg für eine Nachricht vom Eingangsknoten $2 = (010)_2$ zum Ausgangsknoten $6 = (110)_2$ dargestellt. An jedem Kreuzverbinder ist der für das Routing relevante Teil der Kennung zu sehen. In Spalte 1 und Spalte 2 wird die Nachricht entsprechend Bit= 1 auf dem unteren Pfad weitergeleitet. In Spalte 3 wird entsprechend Bit= 0 auf dem oberen Pfad weitergeleitet.
Die nicht mehr benötigten Elemente der Kennung können dazu benutzt werden, die Binärrepräsentation des Ausgangsknotens zu generieren. Dazu ist in jeder Stufe des Netzwerkes das jeweils benutzte Bit der Adreskennung zu invertieren, wenn der Nachrichtenpfad zwischen oberen und unteren Kanal wechselt und das benutzte Bit bleibt unverändert, wenn der Ein- und Ausgang des Nachrichtenpfads auf der gleichen Ebene liegen. Der Empfänger der Nachricht erhält dann sofort ein Nachrichtenpaket mit der Absenderkennung, das für eine Quittung benutzt werden kann.
Der benutzte Routingalgorithmus benötigt keinen zentralisierten Kontrollmechanismus. Er ist deshalb billig und effektiv. Das Netzwerk kann einfach aufgebaut werden. Allerdings besteht eine große Wahrscheinlichkeit, daß das Netzwerk blockiert, weil zwei Nachrichten denselben Verbindungsweg benutzen müssen. Deshalb gibt es eine Vielzahl weiterer, zum Teil erheblich komplexer aufgebauter, hierarchischer Verbindungsnetzwerke.

#### 2.2.2.3 Kreuzverbindung

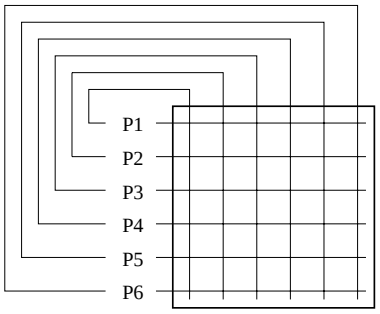


Abbildung 2.7: *Kreuzverbindung*



Die flexible und direkte Verbindung von $p$ Ein- und $q$ Ausgabekanälen wird mit sogenannten Kreuzverbindungen realisiert (siehe Abbildung 2.7). Die tatsächlich geschalteten Verbindungen können mit Software manipuliert werden. Die Anzahl der Schaltelemente beträgt $pq$. Für die Anzahl der Ausgabekanäle sollte $q \geq p$ gelten, um jeden Eingabekanal mit genau einem Ausgabekanal verbinden zu können. Die Komplexität der Kreuzverbindung wächst deshalb mit der Ordnung $O(p^2)$.
In Verbindung mit Transputern wird der Kreuzverbindungsbaustein C004 eingesetzt, der bis zu 32 Eingabekanäle mit der gleichen Anzahl Ausgabekanäle verbinden kann. Die Vektor-Parallelrechner von Fujitsu benutzt einen 224×224 Kreuzverbinder, um bis zu 222 Prozessoren und 2 Kontrollprozessoren miteinander zu verbinden (vgl. 2.3.2).
Kreuzverbindungsnetzwerke können sehr groß und komplex werden. Sie sind deshalb mit steigender Prozessorzahl verhältnismäßig teurer als die anderen freikonfigurierbare Verbindungssysteme.

### 2.2.3 Statische Netzwerke

Im Gegensatz zu freikonfigurierbaren Netzwerken, besitzen statische Netzwerke eine feststehende Topologie, die mit dedizierten Punkt-zu-Punkt Verbindungen gebildet wird. Die Eigenschaften der Topologie

- maximale Distanz (Durchmesser des Graphen),
- erforderliche Verbindungsanzahl (maximaler Knotengrad) und
- Abbildungsmöglichkeit für andere Topologie (Grapheinpassung)

bestimmen die Eignung der Hardwaretopologie für einen speziellen Algorithmus. Durchmesser und maximaler Knotengrad können beim Entwurf eines Netzwerkes bereits eindeutig bewertet werden. Die von einem Algorithmus benutzte Topologie wird aber oft erst zur Laufzeit bestimmt bzw. ist auch zur Laufzeit veränderlich. Insofern sind die Abbildungsmöglichkeiten einer Verbindungstopologie für beliebige Nutzertopologien nur als Annäherung an ein Optimum zu garantieren. Die Auswirkungen der unvollständigen Abbildung einer Anwendertopologie auf das statische Punkt-zu-Punkt-Netzwerk sind mit effizienten Routingalgorithmen zu minimieren.

#### 2.2.3.1 Vollständige Graphen

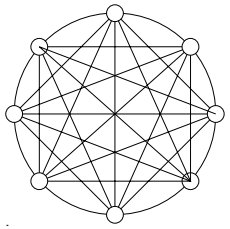

Abbildung 2.8: *Vollständiges Netzwerk*

Sind alle Prozessorelemente über dedizierte Verbindungen direkt mit allen anderen Prozessorelementen verbunden (siehe Abbildung 2.8), dann besitzt dieses Netzwerk ideale Eigenschaften für jede Art von Punkt-zu-Punkt Kommunikation. Der Durchmesser des Netzwerkes ist konstant $d = 1$ und unabhängig von der Anzahl der Knoten. Der Graph dieses Verbindungsnetzwerkes ist homogen.

Allerdings ist der Aufwand für die Konstruktion eines solchen Netzwerkes erheblich. Das Hinzufügen eines neuen Prozessorelements erfordert die Erweiterung der Verbindungsanzahl von jedem anderen Prozessor. Für eine Netzwerk mit $n$ Knoten sind $n-1$ Verbindungen je Knoten erforderlich. Die Gesamtzahl der Verbindungen beträgt $(n^2-n)/2$. Kreuzverbindungsbausteine (vergleiche 2.2.2.3) sind das dynamisch konfigurierbare Pendant zu diesem statischen, vollständigen Netzwerk. Die theoretisch mögliche, vollständig parallele Nutzung aller Verbindungen zur gleichen Zeit wird von numerischen Algorithmen kaum ausgenutzt.

#### 2.2.3.2 Pipeline und Ring

In einem linearen Netzwerk (Abbildung 2.9) sind die $n$ Prozessoren fortlaufend numeriert, z.B. von $0,\ldots,n-1$. In einer Pipeline besitzt jeder Prozessor, mit Ausnahme der Prozessoren 0 und $n-1$ einen Vorgänger und einen Nachfolger. Der Graph einer Pipeline besitzt genau eine automorphe Abbildung, die die Knoten $0,\ldots,n-1$ auf die Knoten $n-1,\ldots,0$ abbildet. In einem Ring sind die Prozessoren 0 und $n-1$ zusätzlich verbunden. Der Graph eines Ringes ist deshalb homogen.

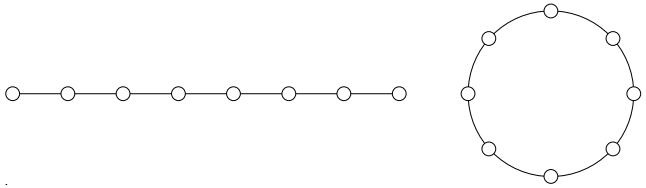

Abbildung 2.9: *Pipeline und Ring* $n=8$

Die Hardwareanforderungen sind mit dem Knotengrad 2 sehr gering. Allerdings ist für den Nachrichtenaustausch mit dem am weitesten entfernten Prozessor im ungünstigsten Fall ein Routing über $n/2$ Verbindungen im Ring, bzw. $n-1$ Verbindungen in der Pipeline erforderlich.

#### 2.2.3.3 Gitter und Torus

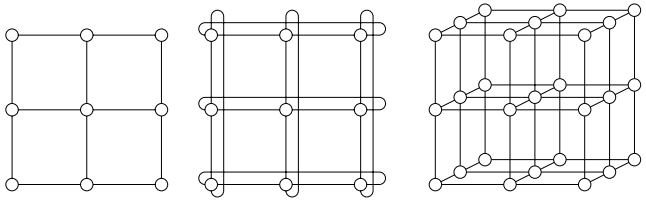


Abbildung 2.10: *Gitter und Torus*

Gitter und Torus (an den Gitterrändern verbundene Gitter, engl. *wraparound mesh*) sind in Abbildung 2.10 dargestellt. Sie besitzen unabhängig von der Anzahl der einbezogenen Prozessorelemente eine konstante Anzahl von Verbindungen pro Knoten. Unter der Dimensionalität versteht man die Anordnung der Knoten in Reihen,



Spalten usw. Ein Gitter der Ordnung 1 ist eine einfache Pipeline, ein Torus der gleichen Ordnung ist ein Ring (vgl. 2.2.3.2). Ein zweidimensionales Gitter mit gleicher Spalten- und Zeilenzahl bezeichnet man als quadratisches Gitter. Der Graph eines quadratischen Gitters besitzt fünf automorphe Abbildungen. Sind Spalten- und Zeilenzahl unterschiedlich, spricht man von einem Rechteckgitter. Der Graph eines Rechteckgitters besitzt drei automorphe Abbildungen. Typische Anwendungen benutzen zwei- und dreidimensionale Gitter bzw. Tori.
Nachrichten zwischen zwei Prozessoren im Gitter kann man zunächst entlang der Zeile des Absenders bis zur Spalte des Empfängers und dann entlang dieser Spalte bis in die Zeile des Empfängers transportieren. Die Entfernung zwischen zwei Knoten im Gitter mit $n$ Knoten der Ordnung $d$ beträgt $d\sqrt[d]{n}$. Im entsprechenden idealen Torus ist diese Entfernung auf $\sqrt[d]{n}$ reduziert. Die Anzahl der Verbindungen je Knoten beträgt im Gitter und im Torus konstant $2d$, unabhängig von der Gesamtzahl der Knoten.
Varianten dieser Topologie sind mit mehr Verbindungen pro Knoten möglich, z.B. ein zweidimensionales Hexaedernetz u.a.

#### 2.2.3.4 De Bruijn-Netzwerke

Ein (d,D) de Bruijn-Netzwerk $\mathbf{B_{d,D}}$ ist ein gerichteter Graph, der aus Worten der Länge $D$ aus einem Alphabet mit $d$ Buchstaben konstruiert wird. Die Verbindung vom Knoten $w_1$ zum Knoten $w_2$ existiert genau dann, wenn die letzten $(D-1)$ Buchstaben von $w_1$ mit den ersten $(D-1)$ Buchstaben von $w_2$ übereinstimmen. Häufig wird ein binäres Alphabet benutzt und die Namen der Knoten ergeben sich aus der Binärrepräsentation der Numerierung $0,\ldots,n-1$. Dieses De Bruijn-Netzwerk hat Knotengrad vier. Eine Verbindung von Knoten $i$ besteht dann zu den Knoten $2i \mod p$ und $2i+1 \mod p$.
Formal gilt: Für eine gegebene ganze Zahl $h \geq 3$ besitzt das de Bruijn-Netzwerk $\mathbf{B_{2,h}}$ $2^h$ Knoten, die fortlaufend mit $0,\ldots,2^h-1$ bezeichnet sind. Jeder Knoten $k=[k_{h-1},k_{h-2},\ldots,k_0]_2$ ist mit den Knoten

$$\begin{aligned} &[k_{h-2},k_{h-3},\ldots,k_0,0]_2\\ &[k_{h-2},k_{h-3},\ldots,k_0,1]_2\\ &[0,k_{h-1},k_{h-2},\ldots,k_1]_2\\ \text{und}\\ &[1,k_{h-1},k_{h-2},\ldots,k_1]_2 \end{aligned} \tag{2.2}$$

verbunden.
Eine interessante Eigenschaft dieser Netzwerke ist, daß nicht alle Knoten den gleichen Knotengrad haben. So sind z.B. die Knoten 0 und $n-1$ mit sich selbst verbunden. Diese Knoten können dann etwa als Ausgabeknoten benutzt werden. Sie werden häufig in Graphikapplikationen angewandt. Die Ein- und Ausgabeknoten sind dann direkt mit dem Graphikterminal oder dem Bildspeicher verbunden.
In [26] werden de Bruijn-Netzwerke als Alternative zu den üblichen Gitter- und Hypercubetopologien vorgeschlagen.

#### 2.2.3.5 Baum

In einem Baum gibt es nur einen Pfad zwischen zwei beliebigen Knoten. Die Pipeline ist ein spezieller Baum. Abbildung 2.11 zeigt vollständige Binärbäume. In einem statischen Binärbaum, ist jeder Knoten des Baumes ein Prozessor. In einem dynamischen Binärbaum sind nur die Blätter des Baumes Prozessoren. An den

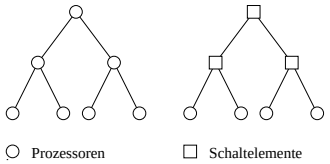


Abbildung 2.11: *Vollständiger Binärbaum mit 4 Blättern*

anderen Knoten arbeiten Schaltelemente, die gegebenenfalls eine Nachricht in die nächtshöhere oder in die darunterliegende Ebene weitergeben.
Der Transportweg von einem Prozessor zu einem anderen benutzt in einem Baum immer den kleinsten Teilbaum, in dem Absender und Empfänger enthalten sind. Zunächst wird die Nachricht zur Wurzel dieses Teilbaumes aufwärts gesendet und danach wieder abwärts zum Empfänger geleitet.

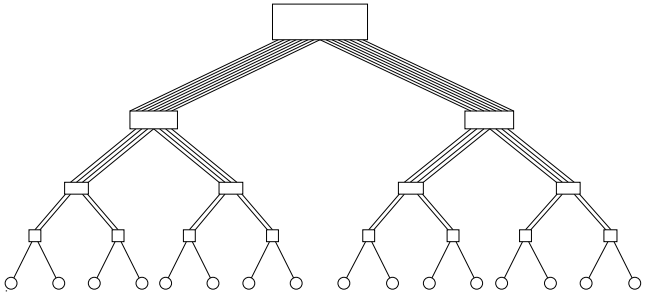

Abbildung 2.12: *Fat Tree mit 16 Prozessoren*

Eine interessante Abwandlung der Baumtopologie ist der sogenannte *Fat Tree* [135] (siehe Abbildung 2.12). Die Äste des Baumes werden von Verbindungsbausteinen gebildet, die entsprechend der Hierarchiestufe immer leistungsfähiger werden. Mit dem Zuwachs an Bandbreite auf hohem Hierarchieebenen wird dem potentiell höheren Kommunikationsbedarf auf dieser Ebene Rechnung getragen.
Zumindestens theoretisch kann damit ein optimales Netzwerk konstruiert werden. Die tatsächlich erreichbaren Übertragungsraten hängen von der Belastung des Netzwerkes und der Verbindungsbausteine ab.
Die Nachricht zwischen zwei Knoten benötigt im *Fat Tree* mit $n$ Blättern maximal $2log_2 n$ Schritte. Die theoretische Übertragungszeit mit den unterschiedlich stark ausgelegten Verbindungspfaden kann aber mindestens den Faktor 2 wettmachen, so daß der *Fat Tree* hinsichtlich der Übertragungszeit die gleichen Eigenschaften wie der im folgenden vorgestellte Hypercube besitzt.

#### 2.2.3.6 Hypercube

Die $2^d$ Knoten und $2^{d-1}$ Kanten eines $d$-dimensionalen Cubes bilden einen Hypercube der Ordnung $d$ bzw. einen $d$-Cube. Numeriert man die Knoten von 0 bis $2^d - 1$ durch, dann kann man die Knoten eines Hypercubes entsprechend eines sogenannten *Graycodes* bezeichnen. Ein *Graycodes* wird aus den mit ihrem Bitmuster repräsentierten Ordnungsnummern der Prozessorknoten abgeleitet. Das Bitmuster



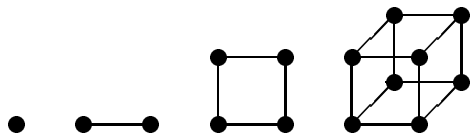

Abbildung 2.13: *Hypercube* $d = 0, \ldots, 3$

der Numerierung zweier, in der Dimension $n$ benachbarter Knoten unterscheidet sich gerade in der $n$-ten Position.
Das folgt direkt aus der rekursiven Bildungsvorschrift für Hypercubes [30]. Der Hypercube der Ordnung 0 ist ein einzelner Knoten - ein 0-dimensionales Objekt (mit der Ordnungsnummer 0). Er wird erweitert durch identische Replikation und eine Verbindung in der Dimension 1 zu einem Hypercube der Ordnung 1 (mit den Ordnungsnummern $[0]_2$ und $[1]_2$). Es entsteht ein eindimensionale Objekt - eine Linie. Entsprechend bildet man durch identische Replikation des Hypercubes der Ordnung 1 und die Verbindung zugeordneter Knoten in der Richtung 2 den Hypercube der Ordnung 2 mit insgesamt vier Knoten (vgl. Abbildung 2.13), der ein zweidimensionales Objekt - eine Fläche - beschreibt. Die alten Knoten behalten ihre Ordnungsnummern ($[0]_2$ und $[1]_2$ bzw. $[00]_2$ und $[01]_2$). Die neuen Knoten werden mit 2 und 3 bzw. $[10]_2$ und $[11]_2$ bezeichnet. Die Hammingdistanz zwischen zwei Knoten $i$ und $j$ bestimmt sich zu $H(i,j) = \sum_{m=0}^{n-1}(i_m \oplus j_m)$. Allgemein bildet man den d-dimensionalen Hypercube aus zwei $(d-1)$-dimensionalen Hypercube mit Verbindung einander entsprechender Knoten beider Cubes.
Eine Nachricht zwischen zwei Knoten des Hypercubes wird mit Hilfe des *Graycodes* von Absender und Empfänger verschickt. Dazu wird eine bitweise, exklusive ODER-Verknüpfung benutzt. Die Anzahl der unterschiedlichen Bitpositionen bestimmt die Anzahl der Verbindungen[3]und der Pfad muß entlang der Dimensionen entsprechend der Bitpositionen gewählt werden. Die maximale Entfernung zweier Knoten im $d$-Cube beträgt deshalb $d = log_2 p$.
Jeder Knoten im Hypercube der Ordnung $d$ besitzt $d$ Verbindungen, bzw. ein Hypercube mit $p$ Knoten erfordert $log_2 p$ Verbindungen pro Knoten. Die Hardwarelösung muß deshalb höheren Anforderungen genügen, als bei einem Gitter oder einem Torus, die eine feste Anzahl von Nachbarn, unabhängig von der Gesamtzahl der Prozessoren aufweisen.
Eine Verallgemeinerung der Hypercubestruktur sind die sogenannten „im Cube verbundene Ringe“ - CCC (engl. *Cube Connected Cycles*). Dabei ist jeder Knoten des $d$-Cubes ein Ring aus $n$ Einzelknoten [172]. Die interessante Eigenschaft dieser Topologie ist der flexible Knotengrad der Ringe. Aus Hardwareknoten mit beschränktem Grad kann durch Kombination in einem Ring ein logischer Knoten gewünschten Grades konstruiert werden. Benutzt man beispielsweise Transputer mit Knotengrad 4, entsteht durch Zusammenschalten von drei Transputern ein Element mit $3 * 4 - 6 = 6$ externen Verbindungen. Daraus kann dann einen Hypercube mit $2^6 = 64$ Knoten und damit $3 * 2^6 = 192$ Transputern, aufgebaut werden.
Eine andere Beschreibung stellt die Verwandschaft zwischen Gitter, Torus und Hypercube her. Der $(n,d)$-Cube ist ein $d$-dimensionaler Cube mit $n$ Knoten in jeder Dimension (vgl. [127], S. 36). $n$ wird dabei als *Radix* bezeichnet. Ein $(2,d)$-Cube, d.h. ein $d$-dimensionaler Hypercube, ist ein $d$-dimensionaler Torus mit zwei Knoten in jeder Richtung. Ein Ring ist im Gegensatz dazu eine eindimensionale Struktur mit $p$ Knoten in dieser einen Dimension und damit ein $(p,1)$-Cube. Ein zweidimensionaler Torus mit $p$ Knoten ist ein $(\sqrt{p},2)$-Cube. Ein $(n,d)$-Cube kann aus $n$

[3] Diese Entfernung ist identisch mit der *Hamming-Distanz*.

Tabelle 2.1: *Charakteristik der Verbindungsnetzwerke (nach [42])*

| | d-Cube | Ring | 2D-Torus | Binärbaum | De Bruijn |
|---|---|---|---|---|---|
| Knotenzahl | $2^d$ | $2^d$ | $2^d$ | $2^d-1$ | $2^d$ |
| Anzahl Links | $d2^d$ | $2^d$ | $2^{d+1}$ | $2^d-2$ | $2^{d+1}-2$ |
| Durchmesser | $d$ | $2^{d11}$ | $2^{d/2}$ | $2(d-1)$ | d |
| Knotengrad | $d$ | 2 | 4 | 3 | 4 |

$(n, d-1)$-Cubes durch Zusammenfassen der einander entsprechenden Knoten zu Ringen gebildet werden.

#### 2.2.3.7 Zusammenfassung

Verbindungsnetzwerke haben sowohl als Topologie der Prozessorelemente (das physikalische Netzwerk) als auch als Kommunikationstopologie einer Applikation (das logische Netzwerk) eine Bedeutung. Die unter dem Hardwareaspekt dargestellten Netzwerke haben Vor- und Nachteile hinsichtlich

**Knotengrad** Der Knotengrad bestimmt die Komplexität des einzelnen Knotens. Ein hoher Knotengrad bedeutet hohen Hardware- und Verwaltungsaufwand. Vorteilhaft sind Netzwerke mit konstantem Knotengrad, unabhängig von der Prozessorzahl, z.B. Baum, Gitter, Torus.

**Durchmesser** Der Durchmesser bestimmt die maximale Wegstrecke zwischen zwei Prozessoren im Netzwerk. Diese Kennzahl ist eine Schranke, die besonders für die kollektiven Kommunikationen maßgebend wird, bei denen Informationen von allen Prozessoren an alle Prozessoren weitergegeben werden müssen.

Diese Kennzahlen sind für die oben beschriebenen Netzwerke in Tabelle 2.1 zusammengestellt.
Moderne Multiprozessorsysteme basieren auf Gitter-, Torus- oder Baumtopologien. Für diese Topologien sind sehr gute Kommunikationsalgorithmen bekannt. Extrem niedrige Startup-Zeit und hohe Bandbreiten der Datenübertragung der modernen Multiprozessorsysteme verringern außerdem den Einfluß auf die Kommunikationszeit, den eine geeignete oder ungeeignete Abbildung der logischen auf die physikalische Topologie haben kann.

### 2.2.4 Routingverfahren

Multiprozessorsysteme mit einem Kommunikationsnetzwerk mit beschränkter Anzahl von Verbindungen je Knoten sind, mit Ausnahme von sehr kleinen Netzwerken, nicht vollständig verbunden. Nachrichten zwischen zwei Prozessoren, die nicht unmittelbar über Linkverbindungen benachbart sind, müssen deshalb über einen oder mehrere Zwischenknoten gesandt werden. Diese Aufgabe wird von einem Router übernommen, der eintreffende Nachrichten zum Empfänger weiterleitet, d.h. die nicht für den Prozessor bestimmten Nachrichten an einen geeigneten Nachbarn übergibt. Routingverfahren in Netzen mit Knotengrad vier (Transputernetzwerke) werden zum Beispiel in [38] diskutiert.
Man unterscheidet prinzipiell die folgenden drei Routingtechniken in Multiprozessornetzwerken:

- Store-and-forward Routing [86], [146], [126],
- Wormhole Routing [46] und



- Cut-through Routing [164].

Die Routingmethode kann dabei deterministisch sein, d.h. eine Nachricht zwischen zwei Prozessoren folgt immer einem vorher festgelegten Pfad (i.d.R. der kürzeste Pfad) oder das Routing erfolgt adaptiv und paßt sich der Momentanbelastung des Verbindungsnetzwerkes an. Adaptive Routingverfahren vermeiden dann den unter Umständen stark belasteten, kürzesten Pfad. Hybride Routingalgorithmen benutzen sowohl deterministische als auch adaptive Techniken.
Beim *store-and-forward* Routing wird die Nachricht von einem Prozessor auf einen Nachbarprozessor geschickt und dort vollständig gespeichert. Der Vorgang wird wiederholt, bis die Nachricht beim Empfänger angelangt ist. Das Modell ist sehr einfach und benötigt lediglich elementare Kommunikationsprimitive. Für eine Nachricht für ein bestimmtes Ziel wird auf jedem Prozessor ein Pufferbereich eingerichtet. Problematisch ist die Festlegung der Größe dieses Pufferspeicher. Der insgesamt erforderliche Pufferspeicher wächst linear mit dem Durchmesser des Verbindungsnetzwerkes [146]. Die lastangepaßte Zuordnung von Pufferspeicher kann nur mit adaptiven Techniken gelöst werden [195].
Das *wormhole* (dt. Wurmloch)-Verfahren benutzt Spezialhardware und eine paketierte Datenübertragung mit Nachrichtenpaketen fester, kurzer Länge. Die Nachricht wird in einen Nachrichtenkopf und die eigentlichen Datenpakete geteilt. Von der Kommunikationshardware wird lediglich der Kopf einer Nachricht mit dem Adressaten interpretiert. Er wird bereits dem nächstfolgenden Links zugeordnet, während die eigentlichen Daten noch übertragen werden. Der Nachrichtenkopf ist somit schneller im Netzwerk unterwegs als die Daten und kann für sie, wie ein Wurm, den benötigten Pfad reservieren. Allerdings blockiert bei diesem Routingverfahren der gesamte Kommunikationspfad zwischen Absender und Empfänger, wenn dort die Nachricht nicht abgenommen werden kann. Die Aufteilung der Nachricht in viele kleine Pakete kann bei latenzdominierter Übertragung zum Engpaß werden.
Analog zum *wormhole* Routing arbeitet die *cut-through* Methode. Im Falle einer Blockierung der Übertragung wird aber der Kommunikationspfad freigegeben. Die Nachricht wird vom Routingsystem gepuffert und andere Kommunikationsanforderungen können inzwischen auf den benötigten physikalischen Linkverbindungen realisiert werden. Der Datendurchsatz steigt damit. Allerdings erfordert das Zwischenspeichern der Nachrichten Speicherplatz. Ist die Anzahl der blockierten Nachrichten groß, wird der Speicherplatz zu einem Engpaß und die Zeit zum Allokieren und Deallokieren von Speicher signifikant für die Leistungsfähigkeit des Kommunikationssystems.

## 2.3 Aktuelle Parallelrechner

### 2.3.1 Parallelrechner von IBM

Das Konzept des WorkstationCluster wird z.B. von IBM mit dem SP-2 (engl. *ScalablePOWERparallel*) System verfolgt. Sie sind üblicherweise aus 2 bis 128 Prozessoren bzw. Workstationbausteinen aufgebaut. Aber auch größere Installationen sind möglich (siehe auch Tabelle 1.2). Diese Workstation können mit unterschiedlich schnellen Prozessoren der RS/6000 RISC Familie und mit unterschiedlichem Speicherausbau ausgerüstet sein. Damit sind sie sowohl als Server- als auch als reiner Rechenknoten prädestiniert. Auf jedem Prozessor wird das Betriebssystem AIX benutzt und der Zugriff auf den Rechner erfolgt direkt über die übliche Netzwerkdienste, wie *rlogin* oder *telnet*, auf einen der Knoten, der als Server konfiguriert ist.
Im Unterschied zur herkömmlichen Netzwerkkopplung der Workstation besitzt das IBM-Cluster eine Hochgeschwindigkeitsverbindung HPS (engl. *high performance*

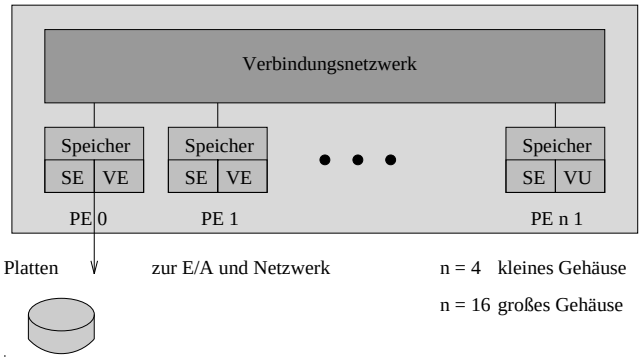


Abbildung 2.14: *VPP500 - Systemarchitektur*

*switch*). HPS benutzt ein IBM proprietäres paketbasiertes Protokoll mit *wormhole* Routing. Mit einer Bandbreite von bis zu 30 MByte/s Allerdings ist eine Startup-Zeit von 40 $\mu$s für den Nachrichtenaustausch im Vergleich zu anderen Multiprozessorsystemen sehr groß (vgl. Tabelle 2.3).
IBM bietet für die SP-2 zusätzliche Betriebssoftware. Das *Loadleveler* Programm basiert auf einem Scheduling System, das die Zuordnung von Ressourcen zu Nutzern und die Auslastung der Rechenknoten überwacht. Serielle und parallele Applikationen können sowohl im Batchbetrieb als auch im interaktiven Betrieb benutzt werden. Die Lastverteilung im heterogenen Clustern ist nicht trivial und wird vom *Loadleveler* unterstützt.

### 2.3.2 Parallelrechner von Fujitsu

Die Fujitsu VPP500 ist ein paralleler Vektorrechner. Er besteht aus bis zu 222 Prozessorelementen (PE), die mit einem proprietären Kreuzverbinder verbunden sind. Jedes Prozessorelement besteht im wesentlichen aus der Skalareinheit, der Vektoreinheit, dem Hauptspeicher und der Datentransfereinheit. Die Skalareinheit (SU) eines Prozessorelements arbeitet nach dem RISC-Prinzip mit langem Instruktionswort, das die parallele Abarbeitung von mehreren Instruktionen in jedem Maschinenzyklus ermöglicht. Gleitpunkt-, Vektor-, Speicher- und Kommunikationsoperationen werden parallel gestartet und asynchron ausgeführt. Die Vektoreinheit (VU) jedes PEs ist ein leistungsstarker Vektorprozessor mit großem Vektorregistersatz und extrem schnellen Vektorpipelines. Jede Vektoreinheit enthält dynamisch rekonfigurierbare Vektorregister der Kapazität 128 kByte mit dazu passenden Maskenregistern.
Die baugleiche NWT(*Numerical Wind Tunnel*-Maschine konnte 1993 das höchste Linpack-Benchmark Ergebnis mit 124.5 GFLOPS für 140 Prozessoren liefern. Die besten Ergebnisse werden für hochoptimierte Vektorroutinen und mit Vektorlängen größer als 500 erreicht. Damit ergibt sich das Anwendungsfeld dieses Parallelrechners für grobgranulare, parallele Probleme mit einer geringen Anzahl von Prozessoren [33].
Die VPP500 verfügt über einen virtuellen globalen Adreßraum. Er wird unter FORTRAN mit dem Betriebssystem realisiert. Allerdings ist der Speicherzugriff auf lokale Speicherbereiche wesentlich effizienter als der Zugriff auf Speicher anderer



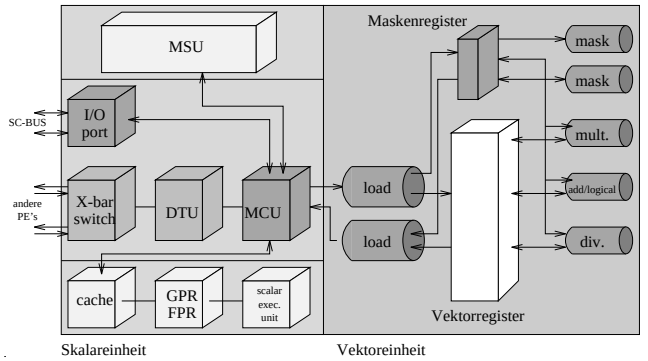


Abbildung 2.15: *VPP300 - Aufbau eines Prozessorelements*

Prozessoren. Abgelöst wird dieses Rechnersystem mit der VPP300, ein vollständig in CMOS-Technologie realisierter Vektorparallelrechner mit ähnlichem Aufbau. Die Einzelprozessorleistung kann in Abhängigkeit von der Taktrate bis zu 2.2 GFLOPS betragen. Jedes Prozessorelement besitzt einen lokalen Speicher von 512 MByte oder 2048 MByte. Die Gesamtkapazität des physikalisch verteilten Speichers eines VPP300-Systems beträgt damit maximal 32 GByte. Das Prozessorelement 0 koordiniert einerseits die Aktivitäten der anderen Prozessoren, andererseits realisiert es die Verbindung zu den angeschlossenen I/O-Geräten bzw. Netzwerken. Die VPP300 ist damit ein selbständiger, im Netzwerk plazierbarer Rechner, der ohne Hostsystem auskommt.
Das Kreuzverbinder-Netzwerk des VPP300 realisiert eine konfliktfreie, schnelle und flexible Verbindung zwischen den einzelnen Prozessoren. Jedes Prozessorelement kann bidirektional mit einer maximalen Transferrate von 400 bzw. 570 MByte/s mit einem anderen Prozessorelement kommunizieren. Alle Prozessorelemente sind dabei gleichberechtigt. Bei der Parallelisierung und bei der Lastverteilung muß die Netzwerktopologie deshalb nicht berücksichtigt werden.

### 2.3.3 Parallelrechner auf Transputerbasis

Das Wort Transputer ist eine Neuschöpfung. Eine Erklärung gibt die Ursprünge aus den Worten TRANSistor und comPUTER an. Ein Transputer ist also ein Baustein für komplexe, umfangreiche Rechnersysteme, wie Transistoren Bausteine für Mikroprozessoren sind. Auf der Basis dieser Technologie werden seit 1985 zunächst von der Firma Inmos, später von SGS Thomson Transputer hergestellt, die als Bausteine von Parallelrechnern geeignet sind.
Im folgenden wird zunächst der Transputer vorgestellt. Danach werden Parallelrechnersysteme verschiedener Hersteller und mit unterschiedlichen Architekturen des Rechenknotens beschrieben. Dabei wird ein proprietäres Laufzeitsystem eines transputerbasierten Parallelrechners vorgestellt.

#### 2.3.3.1 Transputer

Der häufig eingesetzte Transputer T805 ist ein RISC-ähnlicher Prozessor mit bis zu 30 MIPS und 2 MFLOPS. Er besteht aus einer Verarbeitungseinheit mit integriertem Arithmetikprozessor, 4 kByte statischen RAM (OnChip-Memory), einem

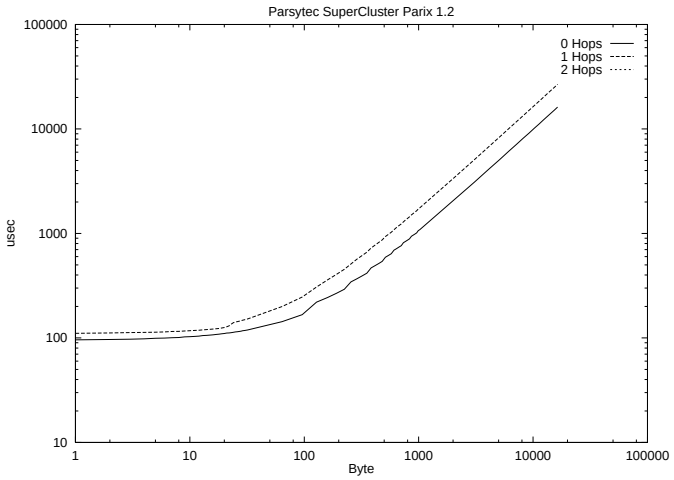


Abbildung 2.16: *Kommunikationsleistung T800 unter PARIX*

Interface zu externem Hauptspeicher und vier Links mit jeweils bis zu 20 Mbit/s Datendurchsatz. Das OnChip-Memory ist für eine Vielzahl von Transputeranwendungen ausreichend, um Daten und Programm zu speichern. Die Zugriffszeiten auf diesen Speicherbereich sind um ein Vielfaches höher, als die Zugriffszeiten auf das externe RAM.
Die auf dem Chip integrierte Linklogik macht den Mikroprozessor als Baustein von Multiprozessornetzwerken geeignet. Ohne zusätzliche Hardware kann durch direktes Verbinden der Transputerlinks ein Netzwerk mit fester Topologie aufgebaut werden. Das Netzwerk hat den maximalen Knotengrad vier (entsprechend den vier Links pro Transputer). Damit lassen sich zweidimensionale Gitter verschalten.
Der Befehlssatz des Transputers umfaßt Instruktionen zum Senden und Empfangen von Nachrichten über die Links. Verzögerungen für die Interprozessorkommunikation werden damit minimiert. Ist die Kommunikation auf einem Kanal durchzuführen, der als physikalischer Link zwischen zwei Prozessen auf unterschiedlichen Transputern geschaltet ist, dann wird die zugeordnete Linklogik mit der Behandlung des Übertragungsmechanismus beauftragt. Alle vier Logikeinheiten für die Links können nach der Initialisierung durch die CPU selbständig die Nachricht übertragen. Der Datendurchsatz erreicht damit die theoretische Obergrenze von $4 \times 20$ Mbit/s=80 Mbit/s.
Der Transputer verfügt über einen mikrocodierten Scheduler. Damit wird das Konzept der parallelen Prozesse in der Hardware selbst realisiert. Zu jeder Zeit ist genau ein Prozeß auf dem Transputer aktiv. Der Scheduler hat die Aufgabe, Prozesse in eine Reihe von ausführungsbereiten Prozessen einzuordnen und nach einem Zeitscheibenverfahren (pro Prozeß mit niedriger Priorität 2 msec) nacheinander zur Ausführung zu bringen (Scheduling) und wieder zu unterbrechen (Rescheduling). Beim Descheduling wird ein Prozeß aus dieser Warteschlange ausgegliedert. Prozesse können nur an ausgewählten Punkten der Programmabarbeitung scheduled, rescheduled oder descheduled werden. Diese Programmpunkte sind Befehle, bei denen nur wenige Statusinformationen (Registerinhalte, Instruktionszähler) abgespeichert werden müssen. Die Liste der Prozesse sieht dafür Datenstrukturen vor. Der Wechsel zwischen zwei Prozessen kann deshalb sehr schnell erfolgen (typischerweise 19 Prozessortakte).



Der Nachteil der beschriebenen Transputergeneration besteht in der hardwareseitigen Begrenzung auf vier gleichzeitig ansteuerbare physikalische Links. Programme, die mehr logische Verbindungen benötigen, als sich auf einer Hardwaretopologie mit Knotengrad vier abbilden lassen, benötigen eine spezielle Routingsoftware, um Nachrichtenaustausch zwischen Prozessen auf nicht unmittelbar benachbarten Transputern zu ermöglichen.
Da der Transputerhersteller Inmos (später SGS Thompson) diese Software nicht mitentwickelte, oblag es den Anwendungsprogrammierern, geeignete Routingalgorithmen zu implementieren. Der Router hat die Aufgabe, Nachrichten für nicht direkt miteinander verbundene Transputer über Zwischenknoten weiterzureichen. Diese Routingsoftware nutzt Rechenleistung, die nicht für die Erfüllung der lokalen Aufgaben zur Verfügung steht.
Alternativ zu applikationsspezifischen Routingsystemen [62], [195], [124] wurden Entwicklungsumgebungen (z.B. TINY vom EPCC, CSTools von Meiko) und ein Transputerbetriebsystem (HELIOS von Perihelion [194], [167]) entwickelt. Parsytec liefert mit seinen Rechnern (s.a. 2.3.3.3) seit 1992 ein Routingsystem mit transparenter Schnittstelle zum UNIX-Hostsystem, das aber nur auf den von Parsytec gefertigten Transputersystemen lauffähig ist.
Die neue Transputergeneration des T9000 besitzt ein in Hardware realisiertes Routingsystem. In Zusammenarbeit mit leistungsstarken und intelligenten Routingchips können die vier physikalischen Links transparent beliebig vielen logischen Verbindungskanälen zugeordnet werden. Die vier Links werden mit einer Kommunikationslogik gesteuert, die eine effiziente Zerlegung der Nachrichten in Pakete und die Nutzung der logischen Kanäle entsprechend der Belastung der physikalischen Linkverbindungen realisiert. Der Nachrichtenaustausch ist paketorientiert und wird vollständig synchronisiert und bestätigt (engl. *acknowledged*). Der entscheidende Vorteil besteht in der Trennung der Routingaufgaben vom Prozessor. Der T800 muß die Routingaufgaben noch mit der CPU realisieren. Entsprechend sinkt der Datendurchsatz auf T800-Systemen bei belasteten Routingknoten erheblich.
An der University of Southampton wurde für die T800 Transputer Generation zur Simulation des virtuellen Linkinterface des Inmos T9000 und des Kreuzverbindungsbausteines C104 die Bibliothek VCR (engl. *Virtual Channel Router*) entwickelt [49]). Diese Bibliothek erlaubt auch auf den T800-Systemen die Entwicklung von T9000-Programmen unter Ausnutzung der vollständig transparenten Kommunikation zwischen beliebigen Prozessen auf beliebigen Prozessoren.
Die neue Tranputergeneration wurde bereits 1991 angekündigt. Die Entwicklung bis zu verfügbaren Stückzahlen dauerte dann aber wesentlich länger als erwartet. Verschiedene Hersteller von Parallelrechnern sind deshalb zwischenzeitlich auf andere Prozessoren ausgewichen oder haben die Hardwareeigenschaften des neuen Transputers in einem proprietären Laufzeitsystem implementiert.

#### 2.3.3.2 Transtech

Transtech ist eine amerikanische Firma, die ursprünglich Parallelrechner auf der Basis der Transputer und der Intel Prozessoren i860 herstellte. Im Vordergrund standen dabei Konfigurationen mit einer moderaten Anzahl von Prozessoren ($p \leq 128$). Die Firma spezialisierte sich auch für Einsteckkarten mit parallel arbeitenden Prozessoren. Das stellt eine kostengünstige Alternative zur Ausrüstung mit einem speziellen Parallelrechner mit separater Stromversorgung und Schnittstelle dar.
Neben den Transputersystemen baut Transtech auch Systeme aus Prozessoren von Texas Instruments, die über sechs Links verfügen und damit wesentlich komplexere

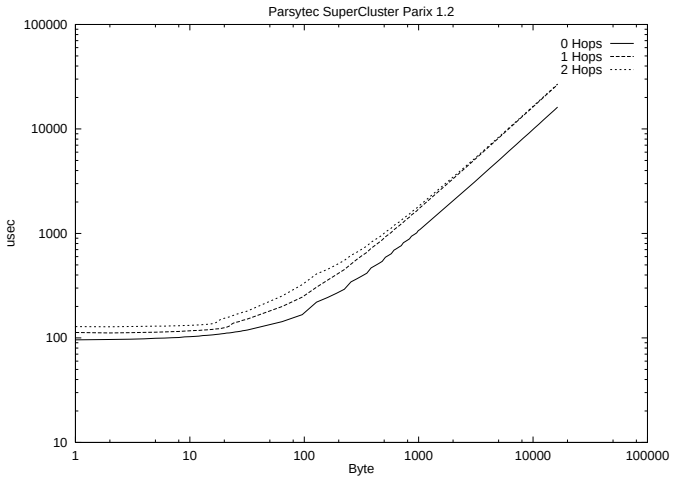


Abbildung 2.17: *Kommunikationsleistung T800 unter PARIX, belastete Routingprozessoren*

Netzwerke (Knotengrad 6) bilden können. Diese Lösung ist als Anwort auf die relativ lange Entwicklungszeit von SGS Thomson für die neue Transputergeneration T9000 zu verstehen.
Neuere Entwicklungen nutzen den Transputer nur noch als Kommunikationsprozessor und setzen den PowerPC von IBM, Motorola und Apple als Applikationsprozessor ein. Die Systeme von Transtech werden mit einem rudimentären Laufzeitsystem, daß im wesentlichen eine Schnittstelle zur Funktionalität der Transputer ist programmiert. Auf den PowerPC-basierten Systemen wird eine identische Funktion zur Verfügung gestellt. Damit können Anwendungen nach Recompilation weiterverwendet werden. Auf ein aufwendiges Betriebssystem wurde verzichtet, um die maximale Leistungsfähigkeit der Hardware an den Anwender weitergeben zu können.

#### 2.3.3.3 Parsytec

Parsytec Computer GmbH produziert und vertreibt seit 1985 Parallelrechnersysteme für Industrie und Forschung. Die Firma entwickelte Parallelrechner auf der Basis des Transputers von Anfang an auch für den Bereich der massiv-parallelen Systeme ($p > 1024$). Rechnersysteme der ersten Generation waren mit einem softwarekonfigurierbaren Netzwerk konstruiert. Mit Hilfe von sogenannten Kreuzverbindungsbausteinen konnten 64 Transputerlinks mit 64 anderen Trasnputerlinks beliebig verschaltet werden. Parsytec entwickelte ein hierarchisch organisiertes System von Rechenclustern mit Transputern und Netzwerkkonfigurationseinheiten, um eine faktisch unbegrenzte Anzahl von Knoten zu verschalten. Dabei war zwar bei mehr als 64 Prozessoren keine direkte Verschaltung von jedem Prozessor zu jedem anderen Prozessor mehr möglich, aber für praktische Belange konnte hinreichend flexibel eine Anzahl von nutzerdefinierten Hardwareverbindungen geschaltet werden. Für eine Anwendung mußte sowohl ein Skript zu Beschreibung der physikalischen Verschaltung des Prozessornetzwerkes als auch ein Skript zur Beschreibung der Abbildung von Anwendungsprogrammen auf die Prozessoren bereitgestellt werden. Spezielle Partitionierungssoftware organisierte die Zuweisung der freien Ressourcen (Prozessoren und Verbindungen) zu Nutzern.



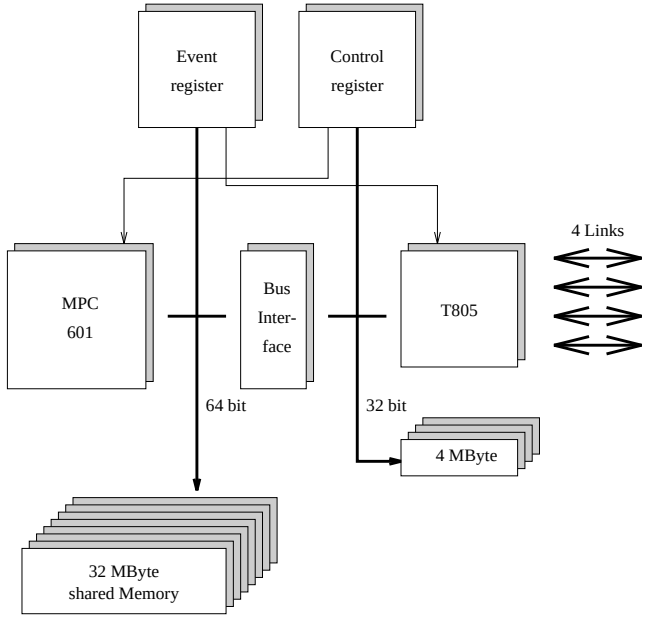


Abbildung 2.18: *Hybride Knotenarchitektur PowerPC und Transputer T805*

Die Parallelrechner der zweiten Generation wurden mit fester Topologie konstruiert und waren aus Transputern als Rechen- und Kommunikationsknoten aufgebaut. 16 Transputer waren zu einem Cluster zusammengefaßt. Jedes Cluster verfügte über einen redundanten, im Fehlerfall benutzbaren Prozessor. Das Cluster bildete ein $4x4$ Gitter und in jede der vier freien Richtungen mit vier freien Transputerlinks konnte direkt ein weiteres Cluster angekoppelt werden. Aus den Clustern wurden durch Zusammenschalten große Systeme aufgebaut. Parsytec konnte mehrere dieser sogenannten GigaCluster auf der Basis von Transputern T805 mit Prozessorzahlen bis zu $p = 1024$ in Europa installieren. Diese Systeme arbeiten mit dem Laufzeitsystem PARIX (vgl. 2.3.3.4). Sie sind einfach um weitere Prozessorelemente erweiterbar. Mit ihrer relativ geringen Einzelprozessorleistung sind sie modernen MPP-Systemen bei weitem nicht mehr gewachsen.
Parsytec setzt deshalb seit 1993, so wie Transtech, auf eine hybride Knotenarchitektur (siehe Abbildung 2.18). In einem Arbeitsplatzgerät werden jeweils ein Kommunikationsprozessor (ein Transputer T805) und ein Applikationsprozessor PowerPC zu einem Rechenknoten zusammengefaßt. Beide Prozessoren greifen auf gemeinsamen Speicher zu. Der Transputer besitzt zusätzlich Hauptspeicher, um die Routingaufgaben unabhängig vom Applikationsprozessor realisieren zu können.
Die Rechenknoten werden in einem 2D-Gitter zusammengeschaltet. 4 Rechenknoten

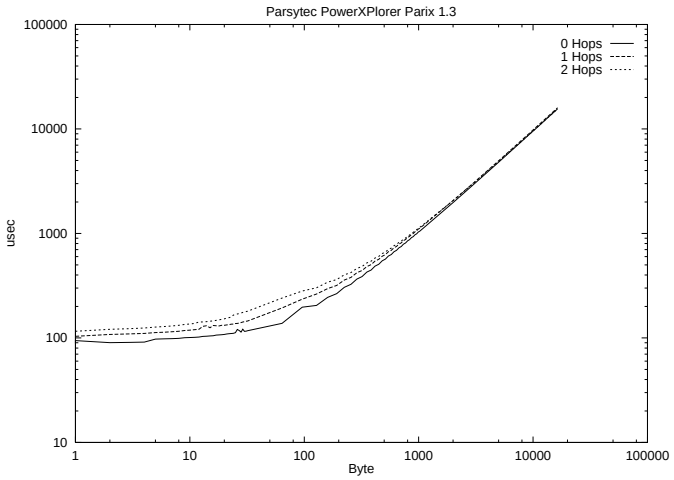


Abbildung 2.19: *Kommunikationsleistung PowerXPlorer*

sind in einem Desktop-Gehäuse untergebracht. Üblicherweise werden bis zu 32 Rechenknoten zu einem PowerXPlorer-Gesamtsystem kombiniert. Im Vergleich zu den reinen Transputersystemen haben diese PowerXPlorer diesselbe Kommunikationsleistung (≈4 MByte/s, siehe Abbildung 2.19), aber eine um ein Vielfaches gesteigerte Rechenleistung. Dieses Verhältnis zwischen Rechen- und Kommunikationsleistung führt zu einschneidenden Veränderungen der Effizienz und der Skalierbarkeit der parallelen Anwendungen.
Um das Mißverhältnis zwischen Rechen- und Kommunikationsleistung auszugleichen, das besonders bei größeren Prozessorzahlen ($p > 32$) zum Tragen kommt, wurde von Parsytec deshalb für die massiv-parallelen Rechner eine andere Knotenarchitektur entwickelt. Pro Knoten arbeiten zwei PowerPC mit vier Kommunikationsprozessoren T805 zusammen (vgl. Abbildung 2.20). Auch diese Rechenknoten werden im 2D-Gitter verschaltet, somit steht in jede der vier Richtungen die Vierfache Kommunikationsleistung zur Verfügung (insgesamt bis zu 16 MByte/s). Das Laufzeitsystem wurde so erweitert, daß die Plazierung von Prozessen auf den Prozessoren transparent ist und die Kommunikation gegebenenfalls alle vier freien Links in eine Richtung simultan benutzen kann.
Die neuesten Entwicklungen von Parsytec auf der Basis des PowerPC benutzen anstelle der Transputerkommunikation einen sogenannten HS-Link, der einen paketorientierten und vor allem wesentlich schnelleren Datenaustausch mit bis zu 60 MByte/s erlaubt. Die Knotenarchitektur wurde weiterentwickelt, um in der Zukunft einen vollwertigen Computer mit PCI und SCSI Schnittstelle auf jedem Rechenknoten zur verfügung zu haben.

#### 2.3.3.4 Laufzeitsystem PARIX

PARIX wird hier als Stellvertreter für eine Reihe von proprietären Entwicklungen von Parallelrechnerherstellern vorgestellt. Initial wurde die Entwicklung von PARIX mit dem Ausbleiben der Lieferung von T9000-Prozessoren motiviert. Die



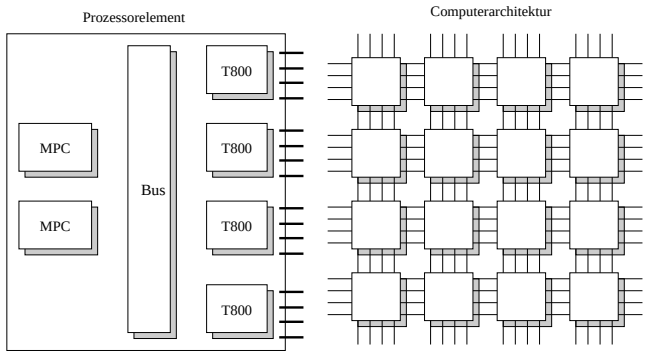


Abbildung 2.20: *Knoten- und Systemarchitektur PowerGC (Parsytec Computer GmbH)*

Haupteigenschaften, die die Eignung des T9000 als Baustein für ein flexibles und leistungsfähiges Parallelrechnerkonzept mit großen Prozessorzahlen begründen, sind

- dynamischer und effizienter Verbindungsaufbau zwischen beliebigen Prozessoren,
- effektive Behandlung von Parallelität auf einem Prozessor (engl. *multitasking*).

Diese Eigenschaften sind zunächst für Transputer T800, später auch auf PowerPC, als parallele Erweiterung von UNIX implementiert worden. Dieses Laufzeitsystem PARIX (*Paralleles UNIX*) ist kein Ersatz für ein Betriebssystem. PARIX stellt ein Programmiermodell basierend auf dem Prinzip des Message-Passing und eine flexible Schnittstelle zu den Betriebssystemfunktionen der angeschlossenen Hostsysteme zur Verfügung. Grundprinzip von PARIX ist es, auf einem homogenen Multiprozessorsystem, das an ein UNIX-Hostsystem angeschlossen ist

- die UNIX-Dienste(vor allem Ein- und Ausgabe) des Hosts zur Verfügung zu stellen,
- Kommunikation mit Message-Passing zu realisieren und
- auf jedem Prozessor multitasking zu unterstützen.

Im Mittelpunkt steht dabei die Skalierbarkeit des Konzepts auf massivparallele Anwendungen (mehr als 1024 Prozessoren).
Das Programmiermodell von PARIX ist das Konzept SPMD (engl. *single program multiple data*), d.h. identische Programme werden auf eine bestimmte Anzahl von Prozessoren geladen und bilden eine Einheit. PARIX ist ursprünglich für die T800-Plattform auf der Basis eines Routingsystems entwickelt worden. Dieses Routingsystem setzt physikalisch zweidimensionale oder dreidimensionale Gitternetzwerke voraus. Auf der physikalischen Gittertopologie erlaubt PARIX wahlfreie Kommunikation zwischen beliebigen Prozessoren.
Entsprechend kann der Nutzer vor Programmstart spezifizieren, auf wieviel Prozessoren sein Programm ablaufen soll. Die Prozessoren müssen als Teilgitter der vorhandenen physikalischen Konfiguration beschrieben werden. Ein Teilgitter, das

einem Nutzer zugeordnet ist, bezeichnet man als Partition. Eine Partition kann nicht gleichzeitig von mehreren Anwendern benutzt werden.

Die einzelnen, gestarteten Programme können mit Hilfe von PARIX-Diensten ihre Position im Netzwerk und die Größe der Nutzerpartition bestimmen. Jedes Programm kann weiteren Programmcode, zur Ausführung bringen. Damit ist das Programmiermodell nicht auf SPMD beschränkt.

Das beim Programmstart geladene ausführbare Programm erzeugt auf dem zugeordneten Prozessor einen Kontext, der einen eigenen, abgeschlossenen Adreßraum besitzt. Symbolische Referenzen auf diesem Adreßraum zwischen verschiedenen Kontexten sind nicht zulässig. Aktive Objekte innerhalb eines Kontext sind *threads* (dt. Faden, im Sinne von Anweisungsstrom).

Initial wird das Hauptprogramm als erster *thread* ausgeführt. Innerhalb eines Kontext kann eine Anzahl von weiteren *threads* abgespaltet werden, die im gleichen Adreßraum arbeiten, über einen Satz von lokalen Variablen (auf dem *thread stack*) verfügen und im Zeitschleifenverfahren, quasi parallel abgearbeitet werden. Das ist vor allem dann sinnvoll, wenn parallel arbeitende Funktionsgruppen, beispielsweise die Kommunikationshardware angesprochen werden soll. Kommunikation und Berechnung können dann in unabhängigen *threads* abgearbeitet werden.

Eine andere Anwendung besteht in der logischen Aufspaltung der Ressourcen eines Prozessors in sogenannte virtuelle Prozessoren für die Nutzung durch mehrere Instanzen eines Programms. Damit kann auf einem Rechner mit beispielsweise 32 Prozessoren ein Algorithmus für 64 Prozessoren getestet werden.

Zwischen beliebigen Prozessoren können logische, bidirektionale Kommunikationskanäle initialisiert werden, auf denen blockierend oder nichtblockierend kommuniziert werden kann. PARIX organisiert transparent für den Anwender das notwendige Routing. Die Reihenfolge der Nachrichten auf einem Kanal bleibt erhalten. Der Empfänger einer Nachricht muß einen Pufferbereich bereitstellen, in den die Nachricht hineinpaßt. Ist die eintreffende Nachricht größer, wird der verbleibende Rest vom Routingsystem verworfen und der Empfänger muß eine Fehlerbehandlung initiieren.

Die Kommunikationskanäle können logischen Kanälen zugeordnet werden, und zu einem Kommunikationspartner sind mehrere logische Kanäle möglich. Diese logischen Kanäle werden unabhängig voneinander verwaltet und benutzt, so daß eine blockierende Kommunikation auf einem Kanal nicht zur Blockierung der anderen Kanäle führt. Die logischen Kanäle können zu Topologien zusammengefaßt werden. Innerhalb einer Topologie werden die Kanäle dann mit Identifikatoren angesprochen. Damit ist die Entwicklung von Bibliotheken für effiziente Kommunikationsalgorithmen [174] und für parallele, numerische Basisalgorithmen möglich.

Parallelität wird unter PARIX nicht implizit mit der Beschreibung des verteilten Datenlayouts (vgl. 2.4.1) erzeugt, sondern muß explizit mit der Instantiierung von parallelen Prozessen, die Nachrichten austauschen, organisiert werden. Damit werden keine neuen Sprachkonstrukte benötigt, sondern es ist ausreichend, eine Anzahl von Funktionen in einer Programmierbibliothek zur Verfügung zu stellen. Zur Anwendung kommen deshalb die Programmiersprachen Fortran, C und C++. Sequentielle und parallele Programmversionen können damit aus einem Programmtext entwickelt werden. Für die Parallelverarbeitung sind keine speziellen Compiler erfoderlich. Der Einarbeitungsaufwand für den Programmierer wird damit minimiert.

PARIX ist aufgrund des Systemkonzepts geeignet, als Basis für die Implementation von weiteren Programmierschnittstellen für Parallelrechner zu dienen. Die Anwendung für eine Implementierung des Message-Passing-Interface Standards wird im Abschnitt 2.5.2 beschrieben.



## 2.4 Programmiermodelle

Für die Programmierung von Parallelrechnern unterscheidet man prinzipiell zwei Herangehensweisen:

- Datenparallele Programmierung und
- Nachrichtenaustausch.

Das datenparallele Programmiermodell ist auf der CM-5 (Thinking Machines Inc., MIMD) und auf einer Reihe von SIMD-Rechnern (Power Challenge, Silicon Graphics Inc., Exemplar, CONVEX) realisiert. Standardisierungsbestrebungen für die Implementation dieses erfolgreichen Programmiermodells führten zur Entwicklung von *High Performance Fortran* - HPF. Analog konnten die datenparallelen Konzepte in C, C++, LISP, ADA und anderen Programmiersprachen realisiert werden. Die Vorteile dieses Ansatzes bestehen nach Fox [70], S. 550 in den folgenden Eigenschaften

- große Portabilität und Skalierbarkeit sowohl auf SIMD- als auch auf MIMD-Rechnern
- geeignet für die Behandlung von synchronen und lose synchronisierten Problemen
- Industriestandard, der wesentliche Eigenschaften aller relevanter, datenparalleler Fortran-Dialekte abdeckt.

Die Parallelisierung einer vorhandenen Applikation mit dem datenparallelen Konzept ist gut automatisierbar. Übernimmt der Compiler vollständig diese Aufgabe, spricht man auch von indirekter Parallelisierung. Besonders matrix- und vektororientierte Applikationen können mit Blockstrukturierung, d.h. im wesentlichen mit Indexmanipulationen, parallelisiert werden. Das Programmiermodell erlaubt jedem Prozessor Zugriff auf die Daten jedes anderen Prozessor und arbeitet damit auf einem logisch gemeinsamen Speicher. Damit wird die Programmierung nicht durch zusätzliche Anweisungen zum Nachrichtenaustausch kompliziert.
Im Gegensatz dazu, benutzt das Programmiermodell Nachrichtenaustausch explizite Anweisungen zum Austausch von nutzerdefinierten Daten zwischen Prozessen. Diese Anweisungen sind als Funktionsaufrufe einer Kommunikationsbibliothek (z.B. MPI [68]) oder als Spracherweiterung für die Behandlung von verteilten Daten (z.B. CC++ [37]) realisiert. Der Aufwand zur Erstellung fehlerfreier Programm mit diesem Programmiermodell ist deshalb signifikant größer. Allerdings wird die Unterscheidung zwischen lokalen und entfernt liegenden Daten explizit im Programm gehandhabt. Damit sind entsprechend effizienter arbeitende Algorithmen möglich. Das Programmiermodell unterstützt die Programmentwicklung sowohl für heterogene Workstation Netzwerke als auch für massiv parallele Multiprozessorsysteme.

### 2.4.1 Datenparallele Programmierung

Das Prinzip der datenparallelen Programmierung basiert auf dem Ansatz des *single point of control* - ein Steuerfluß, d.h. ein und dasselbe Programm, läuft synchronisiert auf einer Anzahl von Prozessoren (für einen Rechner vom Typ MIMD). Ein datenparalleles Programm entspricht damit im wesentlichen dem sequentiellen Programm. Der Unterschied besteht im Entwurf der Datenverteilung, vor allem der Datenfelder. Mit Hilfe von Compilerdirektiven wird das konkrete physikalische Layout eines Datenfelds festgelegt. Die Festlegung kann flexibel an die Anzahl der eingesetzten Prozessoren angepaßt werden. Eine Anweisung zur Addition von zwei

Vektoren $A$ und $B$ zu einem dritten Vektor $C$ wird bei einer geeigneten Verteilung der Elemente von $A$,$B$ und $C$ zu einer parallel ausführbaren Anweisung.
Dazu muß die Größe und die Verteilung der zu verknüpfenden Daten zueinander kompatibel sein, das heißt, einander zuzuordnende Vektorelemente mit dem gleichen Index müssen im Zugriff ein und derselben Verarbeitungseinheit liegen. Gegebenenfalls ist das Laufzeitsystem des Parallelrechners in der Lage, die Datenverteilung vor Ausführung der mathematischen Operation abzustimmen. Dazu sind dann Umverteilen und Kopieren von Datenbereichen erforderlich. Das geht natürlich auf Kosten der Abarbeitungsgeschwindigkeit.
Vier elementare Interprozeßkommunikationen werden im datenparallelen Ansatz durch entsprechende Spracherweiterungen effizient unterstützt - Replikation, Reduktion, Permutation und Parallele Prefix-Operation.

**Replikation** Ein Datum wird durch Kopieren in die gewünscht Anzahl von Instanzen vervielfacht. Implizit wird die Replikation z.B. bei der Skalierung eines Vektors $A$ mit dem Faktor $\alpha$ benutzt, indem die Anweisung $A = \alpha * A$ so ausgeführt wird, daß eine ensprechende Anzahl von Instanzen von $\alpha$ bereitgestellt wird, um die Multiplikation mit den Vektorelementen parallel durchführen zu können.

**Reduktion** Die Reduktion ist die Umkehrung der Replikation. Aus einer Anzahl von verteilt vorliegenden Daten wird mit einer Verknüpfungsfunktion, z.B. Addition, genau ein Datum erzeugt.

**Permutation** Das Layout der verteilt vorliegenden Daten wird mit einer Permutation geändert. Transponieren einer Matrix oder Umkehren eines Vektors sind typische Anwendungen.

**Parallele Prefix-Operation** Resultat dieser Operation sind genausoviele Ergebnisdaten wie Eingangsdaten, die ebenfalls verteilt vorliegen. Ein Beispiel für eine eindimensionale Prefix-Summe ist die Umwandlung einer Häufigkeitsverteilung in eine kumulative Summenreihe.

Neben der auf der Datenverteilung beruhenden Parallelisierung sind im datenparallelen Programmierkonzept die Kontrollelemente der strukturierten Programmierung (Alternative, Zählschleife, Iteration) mit ihren parallelen Entsprechungen realisiert. Eine parallele Zählschleife wird von allen Prozessen gleichzeitig ausgeführt. Jeder Prozeß bearbeitet aber nur die in seinem direktem Zugriffsbereich liegenden Indizes. Vor und nach der Zählschleife werden die Prozessoren synchronisiert. Alternativen werden von jedem Prozeß lokal entschieden, so daß bestimmte Programmpassagen nicht von allen Prozessen abgearbeitet werden müssen.
Die Synchronisation muß durch einen Kontrollprozeß realisiert werden. Sie erfordert eine direkte Kontrolle der Prozessoren über ein gesondertes Steuernetzwerk oder einen gemeinsamen Speicherbereich (z.B. mit Semaphoren).

### 2.4.2 Message-Passing

#### 2.4.2.1 Kommunizierende sequentielle Prozesse

Das Modell der kommunizierenden sequentiellen Prozesse (nach Hoare [100] *communicating sequential processes*) beschreibt die Komponenten eines parallelen Programms (engl. *task*) mit den Elementen Prozeß und Kanal. Ein Kanal ist dabei eine unidirektionale Verbindung zwischen zwei Prozessen. Jeder Prozeß ist ein Programm im herkömmlichen Sinn, mit einem Anweisungs-, einem Eingabe- und einem Ausgabedatenstrom. Die Prozesse sind die aktiven Objekte.



Die Kommunikation zwischen Prozessen basiert auf einem einfachen Prinzip. Die Kopie eines fortlaufenden Speicherbereichs beim Senders wird an den Empfnger bertragen. Zwei Voraussetzungen müssen erfüllt sein:

- Zwischen Sender und Empfänger muß eine Nachrichtenverbindung existieren. Der entsprechende Kommunikationskanal und das zugehörige Routingsystem muß wie eine Telefonverbindung initialisiert sein.
- Sender und Empfänger müssen den Inhalt der Nachricht interpretieren können. Dazu ist ein Protokoll erforderlich. Das Transportsystem sichert lediglich die voll synchronisierte und sichere Datenübertragung. Mit Hilfe des Protokolls ist der Empfänger in der Lage, die Nachricht auszuwerten. Der Empfänger muß in der Lage sein, die Menge der eintreffenden Nachrichten abzunehmen.

#### 2.4.2.2 Nachrichtenaustausch und Programmablauf

Im folgenden werden die Kommunikationstypen

- Blockierende Kommunikation,
- Nichtblockierende Kommunikation,
- Mailbox und
- Interrupt

mit ihrem Einfluß auf die Fortsetzung des Programmablaufes und die notwendige Realisierung in einem Betriebs- oder Laufzeitsystem beschrieben. Die Charakterisierung der vier Kommunikationstypen erfolgt unabhängig von einer Implementation in einem konkreten Message-Passing-System, daß auch mit einer Teilmenge dieser Funktionen realisiert werden kann.
**Blockierende Kommunikation**
Bei der blockierenden Kommunikation sind Sender und Empfänger in der Programmabarbeitung blockiert, bis die Nachricht ausgetauscht ist. Diese Form der Kommunikation führt zur zeitlichen Abstimmung der Programmabarbeitung auf Sender- und Empfängerseite. Sie bietet darüberhinaus ein Höchstmaß an Sicherheit über den Erfolg des Nachrichtenaustausches.
Arbeitet ein Prozeß eine Programmanweisung zum Senden einer Nachricht ab, wird der Prozeß blockiert und der Übertragungsmechanismus aktiviert. Der Prozeß bleibt blockiert, bis die Nachricht den Prozeß über den Transportmechanismus verlassen hat und eine Erfolgsmeldung des Empfängers eingegangen ist.
Analog wird der Empfängerprozeß blockiert, bis der Übertragungsmechanismus den Nachrichtenaustausch realisieren konnte, d.h. den vom Sender bereitgestellten Datenbereich auf den vom Empfänger spezifizierten Datenbereich kopieren konnte. Danach wird eine Erfolgsmeldung erzeugt, die zunächst Empfänger und dann Sender von der erfolgreichen Nachrichtenübertragung informiert. Nach diesem Datenaustausch setzen beide Prozesse mit der Abarbeitung ihrer Programme fort.
Den zeitlichen Abgleich beider Prozesse bezeichnet man als Synchronisation. Algorithmen, die auf der Übertragung der richtigen Daten zum richtigen Zeitpunkt in der Programmabarbeitung angewiesen sind, müssen synchronen Nachrichtenaustausch nutzen. Andererseits ist dieser Nachrichtenaustausch ein einfach realisierbares Mittel zur erwünschten Synchronisation parallel laufender Prozesse in lose gekoppelten Systemen.
Die Zeit, die ein Prozeß auf den Erfolg einer synchronen Kommunikation warten muß, hat negativen Einfluß auf die Effizienz des parallelen Programms (vgl. 2.7).

**Nichtblockierende Kommunikation**
Als Alternative bietet sich die nichtblockierende Kommunikation an. In diesem Fall wird der den Austausch anfordernde Prozeß nicht blockiert, sondern setzt seine Programmabarbeitung fort, bis der Partner zum Nachrichtenaustausch bereit ist. Dabei können in der Zwischenzeit Daten verarbeitet werden, die die für die Nachrichtenübertragung bereitgestellten Daten nicht mehr verändern (nichtblockierendes Senden) oder die die angeforderten Daten nicht mehr benötigen (nichtblockierendes Empfangen).
Für die Implementation und die Nutzung der nichtblockierenden Übertragung ist es notwendig, nach dem Absetzen der Kommunikationsforderung den Zustand oder den Erfolg der Datenübertragung überprüfen zu können, um die benutzten ressourcen (Pufferbereiche) wiederverwenden zu können. Ist die Kommunikation erfolgreich gewesen, kann auf die übertragenen Daten in gewohnter Weise zugegriffen werden und die Programmabarbeitung kann uneingeschränkt fortgesetzt werden.
In der Regel ist es nur für den Sender oder den Empfänger der Nachricht sinnvoll, im nichtblockierenden Modus zu arbeiten. Der Verwaltungsaufwand für das Zwischenspeichern von zu beliebigen Zeiten eintreffenden Nachrichten kann erheblich sein und unter Umständen die Ressourcen des Message-Passing-Systems erschöpfen. Hängt die Fortsetzung der Programmabarbeitung nicht vom erfolgreichen Datenaustausch ab, kann als Übertragungsmechanismus zum Beispiel das Mailboxprinzip genutzt werden.
Eine sinnvolle Anwendung nichtblockierender Kommunikation ist gegeben, wenn der Parallelrechner in der Lage ist, Kommunikation und Befehlsabarbeitung parallel und unabhängig mit verschiedenen Funktionseinheiten auszuführen (z.B. Transputer, vgl. 2.3.3.1). Nutzen die Kommunikationsalgorithmen keine Rechenleistung, kann durch geschickte Reihung von Berechnung auszutauschender Daten, deren Austausch und der Berechnung lokal benötigter Daten, ein zeitlicher Vorteil entstehen.
**Mailbox**
Ungeachtet der Empfangsbereitschaft des Adressaten wird beim Mailboxprinzip die Nachricht abgeschickt. Der Sender hat keine Möglichkeit, direkt den Empfang der Nachricht zu kontrollieren. Der Empfänger ist nicht verpflichtet, eintreffende Nachrichten tatsächlich zu verarbeiten. Das Mailboxprinzip nutzt einen Datenpuffer beim Empfänger, in dem eintreffende Nachrichten bereitgestellt werden, bis sie gelesen oder gelöscht wurden. Aus Speichermangel kann die Nachricht verloren gehen oder an den Absender zurückgeschickt werden.
**Interrupt**
Eine weitere Alternative für die Kommunikation ist der interruptgesteuerte Nachrichtenaustausch. Bei diesem Modell führt die Anforderung der Kommunikation durch die aktive Seite auf der passiven Seite zu einer Unterbrechung der Programmabarbeitung. Nach dieser Unterbrechung erfolgt der Nachrichtenaustausch analog der blockierenden Datenübertragung. Auf einem MIMD-System kann diese Art der Nachrichtenübertragung nur mit einem zusätzlichen Kontrollsystem realisiert werden. Dieses Kontrollsystem muß in höherer Priorität als das Anwendungsprogramm laufen. Benötigt wird die interruptgesteuerte Kommunikation für Anwendungen, in denen das parallel arbeitende Programm als Server konfiguriert wird.

## 2.5 Message-Passing-Interface

Das Message-Passing-Interface (MPI) ist ein Standard für die Programmierschnittstelle von Multiprozessornetzwerken und Workstationclustern [68]. Ein erstes Standarddokument vom 5. Mai 1994 (MPI 1.0) wurde von einem Standardisierungsforum diskutiert und korrigiert. Es liegt nunmehr die geringfügig geänderte Fassung



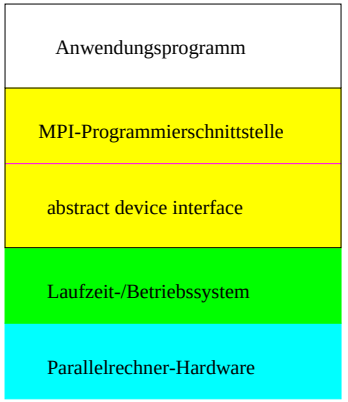


Abbildung 2.21: *MPI-Implementation Schichtenmodell*

vom Juni 1995 (MPI 1.1) vor. Das Ziel von MPI ist es, die Portabilität über eine Vielzahl von verschiedenen Hardwareplattformen durch die Abstraktion von der spezifischen Architektur der Parallelrechner zu erreichen. Für Hardwarehersteller ist die Verfügbarkeit einer standardisierten Schnittstelle zu ihren Systemen Schlüssel zu einer Vielzahl von Anwendungsprogrammen, die unmittelbar auch auf ihren Systemen lauffähig sind.
Die Formulierung des Standards hat deshalb zwar Rücksicht auf die aktuellen Entwicklungen der Parallelrechnertechnologie genommen, indem mehrere, auf den verschiedenen Rechnern unterschiedlich effizient implementierte Kommunikationsformen Bestandteil der Standardisierung wurden. Andererseits haben bereits während der Erarbeitung des Standards mehrere Hersteller erklärt, direkte Implementationen von MPI zu entwickeln bzw. entwickeln zu lassen. Insofern ist MPI eine Schnittmenge der besten Eigenschaften der gegenwärtig auf Parallelrechnern verfügbaren Programmierschnittstellen.
Wesentlichen Einfluß auf das Zustandekommen und die endgültige Formulierung des Standards hatten die Hardwarehersteller von IBM (T.J. Watson Research Center) [14], Intel (NX/2) [166] und nCube (Vertex) [155] sowie die Softwareentwickler von Express (Parasoft Inc.) [163], P4 (Argonne National Laboratory) [35] und PARMACS (GMD) [29],[36]. Weitere wichtige Impulse kamen von Zipcode (Lawrence Livermoore National Laboratory) [192], CHIMP (Edinburgh University) [150],[4], PVM [51] und PICL [76] (beide Oak Ridge National Laboratory).
Für den Anwendungsprogrammierer ist MPI eine Bibliothek. Festgelegt wurden die Sprachbindungen zu FORTRAN und C. Weitere Sprachbindungen werden mit nachfolgenden Standardversionen folgen. Die Entscheidung für eine Sprachbindung von MPI und nicht für eine Erweiterung bestehender Sprachen erleichert die Standardisierung, denn damit finden die Standardisierungsbestrebungen im Bereich der Programmiersprachen unmittelbar Anwendung.
MPI wurde nach der Standarddefinition von verschiedenen Gruppen, aufbauend auf Erfahrungen mit der eigenen Message-Passing-Software, entweder neu implementiert oder auf die vorhandenen Bibliotheken aufgesetzt. Eine portable Lösung bieten die Argonne National Laboratories mit einer MPI Version, die ursprünglich auf der Basis des Tools Chameleon entwicklet wurde, an - MPIch. Diese im

INTERNET verfügbare Implementation besitzt den Vorteil, daß die Schnittstelle zum darunterliegenden Programmiersystem oder zum Laufzeitsystem des Parallelrechners mit elementaren Operationen in einer abstrakten Geräteschnittstelle (engl. *abstract device interface*, ADI) definiert ist (siehe Abbildung 2.21). Dieses hardwareunabhängige Schnittstelle wurde benutzt, um auf dem Laufzeitsystem PARIX eine MPI-Version zur Verfügung zu stellen (vgl. 2.5.2).

### 2.5.1 Funktionalität von MPI

Elementare Konzepte von MPI sind der Prozeß und die Nachricht. Somit ergeben sich Parallelen zum Konzept der kommunizierenden sequentiellen Prozesse von Hoare [100]. MPI verwaltet Prozeßgruppen, den Kontext einer Nachricht, Datentypen und virtuelle Toplogien.
Im folgenden werden die Behandlung von Prozessen und die Identifizierung von Nachrichten zwischen diesen Prozessen dargestellt. Daran anschließend werden die Möglichkeiten der Benutzung von Standard- und der Definition von neuen Datentypen beschrieben. Die Klassifikation der verschiedenen Kommunikationsfuntionen schließt sich an. Dabei wird zunächst auf die Funktionalität der Punkt-zu-Punkt-Verbindungen und dann auf spezifische, kollektive Kommunikationsformen eingegangen.

#### 2.5.1.1 Prozeß und Nachricht

Ein MPI-Prozeß zeichnet sich aus durch

- einen eineindeutigen Prozeßidentifikator und
- einen privaten Datenbereich

aus. Im weiteren Sinne ist ein Prozeß ein ausführbares Programm. MPI unterstützt die gleichzeitige, parallele Abarbeitung von mehreren Instanzen eines Programms. Das entspricht dem Programmiermodell SPMD (engl. it single program multiple data). Prozesse werden zu Prozeßgruppen zusammengefaßt. Die Mitglieder einer Prozeßgruppe werden, beginnend bei 0, lückenlos aufsteigend numeriert. Standardmäßig initialisiert MPI eine Gruppe, in der alle $n$ gestarteten Prozeße enthalten sind und die von $0, \ldots, n-1$ durchnumeriert sind. Diese Nummer ist der Prozeßidentifikator. Zur Laufzeit kann die Anzahl der gestarteten Prozesse nicht verändert werden. Insofern besitzt MPI ein statisches Prozeßmodell[4]. Neue Gruppen können aus vorhandenen Gruppen durch Auswahl und durch Mengenverknüpfungen erzeugt werden. Die Prozesse einer Gruppe führen in der Regel eine gemeinsame Berechnung durch. Der Nachteil, das initial für die Prozeßgruppe nur ein identisches Programm benutzt werden kann, ist leicht zu umgehen, indem in Abhängigkeit vom Prozeßidentifikator unterschiedliche, alternative Programmzweige abgearbeitet werden.
Die Nachricht besteht aus einer Nachrichtenhülle und der eigentlichen Nachricht, die eine Kopie eines fortlaufenden Speicherbereiches aus dem privaten Adreßbereich eines Prozesses darstellt. Die Hülle enthält Informationen über den Empfänger und Absender der Nachricht, über den Inhalt und die Art und Weise des Transports und der Interpretation. Eine Nachricht erhält ein Kennzeichen und wird einer Prozeßgruppe und einem Kommunikationskontext zugeordnet.
Der Nachrichtenaustausch ist für jeden Prozeß die einzige Möglichkeit, auf private Daten eines anderen Prozesses zuzugreifen. Mit einem expliziten Aufruf einer entsprechenden Funktion (mit C- oder FORTRAN-Schnittstelle) wird der Nachrichtenaustausch ausgeführt.

[4] Es gibt Vorschläge, dieses statische Modell in MPI-2 durch ein dynamisches Prozeßmodell zu ersetzen. Allerdings ist das Erzeugen neuer Prozesse nicht sehr portabel zu implementieren und deshalb schlecht standardisierbar.



Tabelle 2.2: *Wirkung der Kommunikationsmodi für das blockierende Senden in MPI*

| Modus | Name | Resultat | Wirkung |
|---|---|---|---|
| Standard | MPI_SEND | synchron oder gepuffert | global |
| Gepuffert | MPI_BSEND | unabhängig vom Empfänger | lokal |
| Synchron | MPI_SSEND | nach abgeschlossenem Empfang | global |
| Bereit | MPI_RSEND | nach begonnenem Empfang | global |

Die Zuordnung einer Kommunikation allein nach Prozeßnummern ist nicht immer ausreichend. Von einer Prozeßgruppe können verschiedene Bibliotheken benutzt werden, die intern wiederum innerhalb der Gruppe Nachrichten austauschen müssen. Die Kommunikation innerhalb der Bibliotheksfunktion ist transparent für den Anwendungsprogrammierer. Ein möglicher Konflikt mit den Kommunikationsanforderungen des Anwendungsprogramms muß durch einen weiteren Mechanismus - den Kommunikationskontext - ausgeschlossen werden. Standardmäßig sind alle Prozesse Mitglied im Kommunikationskontext MPI_COMM_WORLD.
Da die Prozeßnumerierung in einer Gruppe keine Informationen über den physikalischen oder auch den logischen Zusammenhang der Prozesse gibt, kann der Nutzer virtuelle Topologien mit den zugehörigen Prozeßidentifikatoren bilden. Damit wird die Referenzierung auf Nachbarn im Gitter oder im Hypercube einfach und die Abbildung auf die wahren Prozeßidentifikatoren wird von MPI geleistet.
Prozeßgruppe, Kommunikationskontext und virtuelle Topologie werden zum Kommunikator zusammengfaßt. Der Kommunikator kennzeichnet eine Nachricht und wird als Parameter an alle Austauschoperationen übergeben.

#### 2.5.1.2 Datentypen

Die Definition der Datentypen in MPI profitiert von den bereits in den Programmiersprachen C und FORTRAN festgelegten Datentypen. Es werden zunächst die atomaren Datentypen verwendet. Diese Datentypen bilden die Elemente fortlaufender Vektoren mit beliebiger Länge.
Darüberhinaus kann aber auch ein spezielles Zugriffsmuster auf einen verteilt vorliegenden Vektor, beispielsweise nur auf jedes zweite Datenelement festgelegt werden. MPI kopiert vor der Versendung einen verteilt gespeicherten Vektor in einen fortlaufenden Datenbereich. Eine eintreffende Nachricht wird auf das vorgegebene Muster verteilt. Manche Parallelrechner (z.B. CM-5) bieten für diese Zugriffe (Zusammenfassen, engl. *Gather* und Verteilen, engl *Scatter*) Hardwarefunktionen an, die dann von MPI direkt angesprochen werden können.
MPI unterstützt neben konstanten Verschiebungen (engl. *stride*) auch unregelmäßige, sogenannte indizierte Zugriffe. Darüberhinaus wird die Definition neuer Datentypen (analog zu Strukturen in C oder FORTRAN) erlaubt. Die neuen Datentypen können ihrerseits wiederum fortlaufend, mit konstanter Verschiebung oder indiziert vorliegen.

#### 2.5.1.3 Punkt-zu-Punkt Kommunikation

Die elementare Punkt-zu-Punkt Kommunikation erfordert lediglich zwei Prozesse - einen Empfänger und einen Absender. Für jeden Kommunikationsmodus in MPI gibt es eine blockierende und eine nicht-blockierende Version. Die blockierende Version terminiert, wenn die vom Transportsystem benötigten Ressourcen (die Parameter der Kommunikationsfunktion) wiederverwendet werden können. Die nichtblockierende Version erlaubt die Programmfortsetzung unmittelbar nach Initialisierung. In diesem Fall wird die Kommunikation mit weiteren Funktionen zum

Testen oder zum Warten auf die Wiederverwendbarkeit der benötigten Ressourcen ergänzt. Die Wirkung der Kommunikationsfunktion ist lokal, wenn lediglich lokale Daten benötigt werden, um die Kommunikation abzuschließen, sie ist global, wenn Informationen des Kommunikationspartners, d.h. globale Daten, benötigt werden.
Die Standardsendeoperation hat folgende Syntax:

```
MPI_Send(buf, count, datatype, dest, tag, comm)
[IN]    void *buf               Anfangsadresse des Sendepuffers
[IN]    int count               Anzahl der Elemente im Sendepuffer
[IN]    MPI_Datatype datatype   Datentyp
[IN]    int dest                Empfanger
[IN]    int tag                 Nachrichtenkennung
[IN]    MPI_Comm comm           Kommunikator
```

Der Nachrichteninhalt wird mit den Parametern Puffer, Anzahl und Datentyp beschrieben. Die Parameter Empfänger, Nachrichtenkennung und Kommunikator bestimmen die Nachrichtenhülle und den Kontext der Nachricht. Danach kann ein Prozeß eine Nachricht selektieren und gegebenenfalls der richtigen Softwareschicht zuordnen.
Die Punkt-zu-Punkt Kommunikationen sind in MPI in den vier Kommunikationsmodi Standard, Gepuffert (engl. *buffered*), Synchron (*engl. synchronous*) und Bereit (*engl. ready*) implementiert (siehe Tabelle 2.2), die im folgenden ausführlich beschrieben werden:
Im **Standard**modus wird eine Sendeoperation initialisiert und an die Transportschicht übergeben. Danach kann die Programmabarbeitung fortgesetzt werden, ohne das das Programm blockiert. Das erfordert eine entsprechende Zwischenpufferung der Nachricht durch die Transportschicht, die gegebenfalls nicht erfolgreich ist. Dann führt der Standardmodus zur Blockierung des Programms. Hängt die korrekte Programmabarbeitung davon ab, daß die Nachricht nicht zur Blockierung führt, ist anstatt des Standardmodus einer der folgenden Modi für die Kommunikation auszuwählen.
Der Datenbereich für die Nachrichtenübertragung wird in der Regel in Speicherbereichen der Transportschicht bereitgestellt. Dabei kann es unter Umständen zu Problemen und Ressourcenmangel (vor allem Speicherplatzmangel) kommen. Sollen diese Probleme umgangen werden oder wird ein bestimmter Speicherbereich fortgesetzt zum Übertragen von Nachrichten benutzt, ist es sinnvoll und effektiver, den Übertragungspuffers im Anwendungsprogramm bereitzustellen. Dafür ist der Übertragungsmodus **Gepuffert** geeignet, der ausschließlich Speicherbereich, der im Anwendungsprogramm allokiert wurde benutzt. Diese Operation führt deshalb niemals zur Blockierung des Senders. Der Speicherbereich muß mit einer speziellen MPI-Funktion der Transportschicht bekannt gemacht werden. Die Kommunikationsoperation ist lokal. Der Erfolg hängt lediglich von den lokal verfügbaren Ressourcen und nicht vom Adressaten ab. Die Funktion kann ausgeführt werden, unabhängig davon, ob eine dazu passende Empfangsroutine aufgerufen wurde.
Im Gegensatz dazu wird im Modus **Synchron** vor Versenden der Nachricht ein Abgleich mit dem Empfänger vorgenommen, der sichert, daß dort genügend Speicherplatz zum Abnehmen der Nachricht bereitgestellt wurde. Die Datenübertragung beginnt erst, wenn der Empfänger Abnahmebereitschaft erklärt hat. Insofern ist die Operation global. Die Funktion ist identisch mit einem Paar blockierend ausgeführter Sende- und Empfangsoperationen.
Im Modus **Bereit** muß der Empfänger vor dem Absender eine entsprechende Anforderung absetzen. Andernfalls terminiert die Sendeoperation sofort mit einer Fehlermeldung. Die Operation ist deshalb global. Im folgenden Ablauf ist die Funktion identisch zum Standardmodus und sollte die selben Ergebnisse liefern. Der Status



des Empfängers kann aus dem Erfolg der Operation nicht abgeleitet werden. Die Funktion kann aber auf einigen Parallelrechnern den Aufwand vor der Initialisierung des Sendens reduzieren, indem ein Handshake vermieden wird.
Für die Empfangsoperation gibt es in MPI nur den Standardmodus. Diese Funktion hat die folgende Syntax:

```
MPI_Recv(buf, count, datatype, source, tag, comm, status)
[OUT]   void *buf               Anfangsadresse des
                                Empfangspuffer
[IN]    int count               Anzahl der Elemente
                                im Empfangspuffer
[IN]    MPI_Datatype datatype   Datentyp
[IN]    int soure               Absender
[IN]    int tag                 Nachrichtenkennung
[IN]    MPI_Comm comm           Kommunikator
[OUT]   MPI_Status status       Statusobjekt
```

Für Absender und Nachrichtenkennung können wahlfreie Angaben (mit den Symbolen MPI_ANY_SOURCE bzw. MPI_ANY_TAG) gemacht werden. Mit diesen *wildcards* können beliebige Nachrichten von beliebigen Quellen empfangen werden. Der Kommunikator muß allerdings spezifiziert werden, um die Zuordnung der eintreffenden Nachrichten korrekt vornehmen zu können. Die Länge des bereitgestellten Datenpuffers muß groß genug sein, die ankommende Nachricht komplett aufzunehmen, andernfalls bricht die Operation mit einem Fehler ab und das Ergebnis ist undefiniert. Ist die ankommende Nachricht kürzer als der Empfangspuffer, wird der Pufferbereiches vom Beginn an benutzt, der Rest bleibt undefiniert. Es wird keine Endemarkierung vorgenommen und der verbleibende Rest wird nicht initialisiert.
Die Angaben zur tatsächlichen Länge der eingetroffenen Nachricht, zum Absender und zur benutzten Nachrichtenkennung können über das Statusobjekt abgefragt werden. Dazu sind die entsprechend implementierten Zugriffsfunktionen zu benutzen.
Alle genannten Kommunikationfunktionen sind sowohl als blockierende und als nicht blockierende Funktionen realisiert. Für die sichere Ausführung der nichtblockierenden Funktionen ist es erforderlich, zusätzliche Informationen über den Status dieser Kommunikationen zu liefern. Dazu existieren wiederum Funktionen zum blockierenden und nichtblockierenden Testen, ob eine spezielle Kommunikationsfunktion ausgeführt wurde, d.h. ob es möglich ist, die Ressourcen der Kommunikation, vor allem Speicherbereiche, wiederzubenutzen.

#### 2.5.1.4 Kollektive Kommunikationen

Kollektive Kommunikationen werden von allen Prozessen innerhalb des Kommunikators gemeinsam durchgeführt. MPI definiert für die praktisch relevanten, kollektiven Kommunikationen Funktionen, die von verschiedenen Multiprozessorsystemen auch in optimierten Versionen unterstützt werden. Sie arbeiten im Standardmodus, d.h. sie blockieren in Abhängigkeit von der Implementierung und von der Länge des auszutauschenden Datums die beteiligten Prozeße. Die wichtigsten kollektiven Operationen sind:

**Broadcast** Ein Prozeß (die Quelle oder die Wurzel des Broadcast) liefert ein und dieselbe Information an alle anderen beteiligten Prozesse.

**Reduce** Alle Prozesse liefern ein Datum. Die einander in ihrer Position im Eingabedatenstrom entsprechenden Daten der Prozessoren werden verknüpft. Die Information wird mit dieser Operation verdichtet. Der Datenumfang bleibt

gleich. Alle Empfänger erhalten die gleiche Information. Werden als Empfänger der Ergebnisdaten alle Datenquellen spezifiziert, spricht man auch von einem globalen Austausch. Der globale Austausch kann oft sehr effizient als eine Reduktion zu einer Wurzel mit anschließendem Broadcast implementiert werden.

**Scan** Alle Prozesse liefern ein Datum. Dieses Datum wird derart verknüpft, daß ein Prozeß mit der Ordnungsnummer $i$ das Resultat der Reduktionsoperation mit den Prozessen $0, \ldots, i-1$ erhält (Paralleler Prefixscan). In MPI ist eine inklusive Scan-Operation definiert. Aus dem Ergebnis kann sehr leicht mit einer lokalen Operation das Ergebis der exklusiven Scanoperation erhalten werden.

Die Ähnlichkeiten zu den Grundprimitiven der datenparallelen Programmierung (Replikation, Reduktion, Permutation und Parallele Prefix-Operation - vgl. Abschnitt 2.4.1) sind offensichtlich.
Eine kollektive Operation, die keine Nachricht austauscht, aber die beteiligten Prozesse synchronisiert, ist die Funktion **Barrier**. Die Operation blockiert die beteiligten Prozesse, bis alle Prozesse die Funktion aufgerufen haben. Die Operation wird im synchronen Modus ausgeführt.

### 2.5.2 Implementation von MPI auf einem Multiprozessornetzwerk

Im folgenden wird die Implementation von MPI auf der Basis des in Abschnitt 2.3.3.4 beschriebenen Laufzeitsystems PARIX dargestellt. Die Ausführungen beziehen sich dabei auf die Version 1.0.9 der MPIch-Implementation von ANL/MSU. In diesem Paket wird eine Schnittstelle zur Zielplattform, das sogenannte *Channel Device Interface*, realisiert [85], [141]. Die Operationen dieses *abstract device interface* (ADI) basieren auf den Grundfunktionen zum Nachrichtenaustausch zwischen zwei Prozessoren. Das kann mit den UNIX-Systemfunktionen *select*, *read* und *write*, aber auch mit den PARIX-Funktionen *Select*, *Read* und *Write* realisiert werden.
Konzeptionell liegt der Schnittstelle das CSP-Prinzip [100] zugrunde, d.h. ein Prozeß kommuniziert über unidirektionale Kanäle mit anderen Prozessen. Die Zuordnung der Kanäle zu den Prozessen ist statisch. MPI wird deshalb mit einem vollständigen (logischen) Verbindungsnetzwerk initialisiert. Damit kann dann jede Kommunikationsanforderung im Netzwerk auf einem initialisierten Kanal abgearbeitet werden. Unter UNIX ist diese Annahme keine Einschränkung. Für andere Systeme (z.B. PARIX oder ein store-and-forward Routingsystem) ist diese Annahme Grundvoraussetzung für eine erfolgreiche Implementierung.
Die vom ADI insgesamt bereitzustellenden Funktionen lassen sich in Verwaltungsfunktionen, Funktionen zum Austausch von Kontrollnachrichten und Funktionen zum Austausch der Anwenderdaten unterteilen. Sie werden jeweils in blockierenden und nichtblockierenden Varianten bereitgestellt, wobei die nichtblockierenden Operationen um Probe- und Abschlußoperationen ergänzt werden müssen.
Die wichtigsten Verwaltungsfunktionen sind:

**MPID_PARIX_init** Mit dieser Funktion wird MPI initialisiert. Dabei werden die Programmargumente nach spezifischen PARIX-Informationen durchsucht. Die erforderlichen Kommunikationskanäle und Datenstrukturen werden initialisiert. Die Funktion darf von einem Anwendungsprogramm nur einmal vor einem Aufruf von MPID_PARIX_finalize aufgerufen werden.

**MPID_PARIX_finalize** Diese Funktion beendet MPI und erlaubt es, die Programmabarbeitung fortzusetzen. Vorher müssen alle *threads* von MPI beendet



werden. Die Funktion hat keine Wirkung, wenn vorher nicht die Initialisierung mit der Funktion MPID_PARIX_init durchgeführt wurde.

**MPID_PARIX_abort** Diese Funktion beendet alle Prozesse auf dem Parallelrechner. Sie wird auch am Ende einer Fehlerbehandlung aufgerufen.

Das ADI versteht unter einer Nachricht eine zusammengesetzte Struktur aus einem Kontollteil und dem Anwenderdatum. Der Kontrollteil enthält die wesentlichen MPI-Informationen über die Nachrichtenkennung, die Länge (in Byte) und den Kommunikator. Separate Routinen werden für die Übertragung des Kontrollteils und des Anwenderdatum benutzt. Im einzelnen sind folgende Routinen definiert:

**MPID_ControlMsgAvail** Diese Funktion gibt an, ob und wenn ja von welchem Kanal ein Kontrollteil eingetroffen ist.

**MPID_RecvAnyControl** Diese Funktion empfängt den nächsten Kontrollteil, ggf. blockiert die Funktion, bis ein Kontrollteil eingetroffen ist.

**MPID_SendControl** Diese Funktion sendet einen Kontrollteil auf einem Kanal. Die Funktion arbeitet nicht blockierend. Somit darf das Terminieren dieser Funktion nicht davon abhängen, ob der Empfängerprozeß die Nachricht abnimmt. In der PARIX-Umgebung erfordert diese Anforderung die Initialisierung eines unabhängig weiterarbeitenden Programmteils (*thread*). Ist das aus Effizienzgründen unerwünscht, sollte stattdessen die auf jeden Fall blockierende Funktion **MPID_SendControlBlock** benutzt werden.

**MPID_RecvFromChannel** Diese Funktion empfängt eine Nachricht von einem Kanal.

**MPID_SendChannel** Diese Funktion sendet eine Nachricht auf einem Kanal. Die Funktion blockiert, bis die Nachricht vom Empfänger abgenommen wird.

Die Reihenfolge des Eintreffens der Nachrichten auf einem Kanal muß identisch zur Reihenfolge des Absendens der Nachrichten sein. Damit wird auch gesichert, daß zum Beispiel die Kontrollnachricht immer vor der dazugehörigen eigentlichen Information gesendet wird.
Das korrekte Einordnen der eintreffenden und ausgehenden Nachrichten in Warteschlangen, die Verwaltung der Statusinformationen und die Bereitstellung von Speicherplatz für die Nachrichtenhülle werden von der über dem Channel device interface liegendem Teil des ADI geleistet. Diese Verwaltung kann bei Bedarf (z.B. wenn das Zielsystem ein adäquates Pufferungssystem bereitstellt) reduziert werden. Für die PARIX-Implementation wurde sie vollständig übernommen, da die PARIX-Programmierschnittstelle selbst keine gepufferte Kommunikation zur Verfügung stellt. Hier sind die wesentlichen Ursachen für einen erheblichen Effizienzverlust beim Einsatz von MPI zu sehen. Eine Implementation direkt auf dem Routingssytem von PARIX und die Abstimmung auf die interne Datenhaltung wäre von Vorteil. Allerdings sind die mit der vorliegenden Implementierung erreichten Leistungsdaten nur bei kurzen Nachrichten schlechter als die PARIX-Grundfunktionen. Für die meisten Multiprozessor-Verbindungsnetzwerke ist es effizient, zwischen langen und kurzen Nachrichten zu unterscheiden. Kurze Nachrichten werden sofort mit dem Kontrollteil der Nachricht verschickt, also ohne die sonst erforderliche, nachfolgende Übertragung im Datenteil zu benutzen. Dazu muß die Paketlänge der Kontrollnachricht so erweitert werden, daß die kurze Nachricht hineinpaßt. Die Länge einer kurzen Nachricht ist nun so zu definieren, daß der Mehraufwand zum Bereitstellen und Verschicken des größeren Kontrollteils sinnvoll begrenzt ist. Dazu wird der Aufwand zum Senden eines Kontrollteils mit hineinkopierter Information dem

Aufwand zum separaten Senden des Kontrollteils und der Information gegenübergestellt:

$$(s + r * (n + h)) + c * n \approx (s + r * n) + s + r * h \tag{2.3}$$

$s$ ist die Start-up Zeit, $r$ ist die Zeit um eine Nachricht der Länge 1 Byte zu versenden, $n$ ist die Gesamtlänge der Nachricht, $h$ ist die Länge des Kontrollteils und $c$ ist die Zeit um ein Byte im Speicher zu kopieren. Die Abschätzung reduziert sich auf

$$c * n \approx s \tag{2.4}$$

womit sich die geeignete Paketlänge z.B. für ein PowerXPlorer-System mit $s \approx 2.2 * 10^{-4}$ und $c \approx 4.2 * 10^{-8}$ zu

$$n = \frac{s}{c} = \frac{220 * 10^{-6}}{42 * 10^{-9}} = 5240 \tag{2.5}$$

ergibt.
Betrachtet man das Versenden einer Nachricht im nichtblockierenden Modus, entsteht das Problem der Pufferung dieses Datenbereiches. Diese Pufferung kann zumindest für die Kontrollnachrichten (d.h. für alle kurzen Nachrichtenpakete) vom ADI geleistet werden. Zweckmäßigerweise werden dafür vorher allokierte Datenbereiche benutzt, die über eine Freiliste verwaltet werden. Lange Nachrichten können effizient nur vom Anwendungsprogramm bereitgestellt werden. Die Schnittstelle verwaltet nur einen Verweis auf diese Nutzerdatenbereiche.
Die nichtblockierenden Sendeoperationen werden mit Hilfe von PARIX- *threads* implementiert. Jede dieser Kommunikationsanforderungen startet eine neue *thread*. Dabei bilden die einem logischen Kanal zugeordneten *threads* eine Schlange. Am Beginn der Schlange steht die nächste auszuführende Kommunikation. Jede folgende Kommunikation reiht sich am Ende der Schlange ein und wartet auf die Beendigung der vorherigen Kommunikation. Die zuletzt eingereihte Sendeoperation wird mit ihrem *thread*pointer gespeichert und kann von einer blockierenden Sendeoperation oder von der nächsten nichtblockierenden Sendeoperation als Referenz benutzt werden. Der Ablauf der blockierenden Sendeoperation sieht dann schematisch wie folgt aus:

1. Test der Parameter
2. Warten auf Beendigung der vorherigen Kommunikation
3. Aufruf der Sendefunktion von PARIX
4. Fehlerbehandlung

Zur Behandlung der nichtblockierenden Übertragung langer Nachrichten werden zusätzlich die folgenden Funktionen benötigt:

**MPID_ISendChannel** Diese Funktion sendet eine Nachricht auf einem Kanal. Die Funktion blockiert nicht, d.h. es wird lediglich die Datenübertragung initialisiert und der Prozeß kann weitere Berechnungen durchführen oder Kommunikationen initialisieren. Die Funktion liefert eine Referenz, über die der Status der Kommunikation abgefragt werden kann.

**MPID_WSendChannel** Diese Funktion gibt eine Information über den erfolgreichen Abschluß einer MPID_ISendChannel Anweisung. Die Funktion blockiert, bis die Nachricht tatsächlich vom Empfänger abgenommen wird. Der Prozessor führt in der Zwischenzeit keine Berechnungen durch.



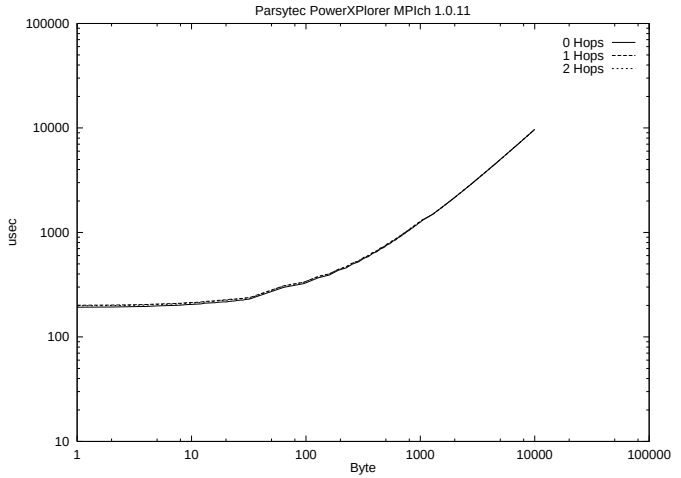


Abbildung 2.22: *Kommunikationsleistung MPI*

Eine nichtblockierene Sendeoperation führt die gleichen Parameterüberprüfungen wie die blockierende Operation durch, erzeugt einen Parametersatz für die zu startende *thread* und startet diese dann. Im Parametersatz für die neue *thread* werden die folgenden Informationen abgelegt

- Kanal, Puffer und Datenlänge
- Referenz auf vorherige Kommunikationsthread mit dem gleichen Kanal und
- eine Referenz auf die Statusinformation

Die neugestartete *thread* führt dann folgende Operationen aus:

1. Warten auf Beendigung der vorherigen Kommunikation
2. Aufruf der Sendefunktion von PARIX
3. Fehlerbehandlung
4. ggf. setzen der Statusinformation
5. ggf. Freisetzen des Datenpuffers (nur für kurze Nachrichten)
6. Freisetzen der *thread*parameter

Die spezielle Implementation der PARIX-Schnittstelle benutzt zur Information über den Status einer nichtblockierenden Sendeoperation eine Datenstruktur der MPI-Schnittstelle.
Die Implementation der Funktion MPID_TSendChannel zum nichtblockierenden Testen einer Nachrichtenübertragung kann deshalb durch eine Abfrage des Statusvariablen realisiert werden. Die Statusinformation wird nur von Sendeoperationen mit langen Nachrichten benötigt, weil nur in diesem Fall der Anwender dafür Sorge tragen muß, daß die Austauschpuffer nicht überschrieben werden.

Eine Nutzung der MPI-Implementation durch verschiedene Schichten eines Anwendungsprogramms und unter Umständen durch ein selbst paralleles Anwendungsprogramm (engl. *Multithreading*) muß sicher möglich sein. Kritisch ist in diesem Zusammenhang die Notwendigkeit, für lange Nachrichten die Folge von Nachrichtenhülle und der eigentlichen Nachricht, die in zwei aufeinanderfolgenden Paketen übertragen werden, zu garantieren. Das kann nur erreicht werden, wenn eine Anforderung für den Austausch einer langen Nachricht durchgängig exklusiv von einer *thread* ausgeführt werden kann. Insbesondere darf zwischen der Initialisierung MPID_SendControl und der zugehörigen MPID_xSendChannel-Funktion keine andere *thread* eine Nachricht initialisieren. Sollte das wider Erwarten passieren, kann der Empfänger die Länge des eintreffenden Nachrichtenpaketes nicht zutreffend bestimmen und ein Fehler in der Nachrichtenübertragung ist die Folge. Deshalb muß die Initialisierung der Übertragung der langen Nachricht und der schreibende Zugriff auf die Liste der auf die Kommunikation wartenden *threads* mit Semaphoren geschützt werden.
Die MPI-Schnittstelle ist in der Lage, die eintreffenden Nachrichten den Kommunikationskontexten und den Nachrichtenkennungen zuzuordnen. Das wird wiederum durch eine geeignete Pufferung auf Empfängerseite erreicht.

## 2.6 Kommunikation in MIMD-Rechnern

### 2.6.1 Modell eines MIMD-Rechners

Die Untersuchung der Leistungsfähigkeit der Kommunikation in Netzwerken kann nur unter Beachtung von charakteristischen Eigenschaften eines Multiprozessorsystems erfolgen. Diese Eigenschaften werden von den im folgenden definierten technischen Parametern Startup-Zeit, Kommunikationslatenz und Bandbreite sowie Knotenlatenz bestimmt.
Die *Startup-Zeit* $t_{startup}$ bestimmt, wie lange der Absender benötigt, die Nachricht zum Verschicken vorzubereiten (Fehlerbehandlung, Bereitstellung der Nachrichtenhülle), den Routingprozeß zu initialisieren und die Verbindung zum Empfänger herzustellen. Dieser Aufwand ist konstant und unabhängig von der Größe der Nachricht. Er tritt jeweils nur einmal auf.
Als *Kommunikationslatenz* bezeichnet man die Zeit, die eine Nachricht der Länge Null (d.h. lediglich die Nachrichtenhülle) für den Weg zwischen zwei Prozessoren benötigt. Mit der *Bandbreite* $b$ in $Byte/sec$ des Übertragungskanals kann die Zeit zur Übertragung eines Bytes zu $t_{transfer} = 1/b$ bestimmt werden. Einen entscheidenden Einfluß auf die Übertragungszeit besitzt die Auslastung des Kommunikationsnetzwerkes. Messungen der tatsächlichen Transferzeiten in belasteten Netzwerken ergeben starke Abweichungen von den Idealdaten. Weitere Einflußfaktoren für die erzielbare Bandbreite sind Datenlänge und Paketlänge in der Anwendungs- und Transportschicht, zusätzliches Kopieren oder Verarbeiten der Nachricht in der Transportschicht (z.B. Berechnen von Checksummen), Ausrichtung der Nachrichtenpuffer im Speicher (Adresse, Cacheline und/oder Speicherseite).
Die Startup-Zeit und die Bandbreite kann mit einem relativ einfachen Testprogramm auf jedem Multiprozessorsystem ermittelt werden (vgl. [52], Benchmarkprogramm aus *netlib*). Dazu sendet ein Client beständig Nachrichten bestimmter Länge an einen Empfänger, der sofort das Echo zrücksendet. Der Absender mïst die Rundlaufzeit (engl. *round-trip time*). Die Zeiten werden für verschiedene Nachrichtenlängen (von 0 bis 1 MByte) gemessen. Aus der minimal erreichbaren Übertragungszeit wird die Startup-Zeit ermittelt. Dabei kann unter Umständen die Auflösung der Zeitmessung auf dem Rechner maßgeblich die Genauigkeit der Messung beeinflussen.



Tabelle 2.3: *Vergleich von Taktzeit und Startup-Zeit verschiedener Multiprozessorsysteme, Quelle: [52]*

| System | Topologie | Taktzeit [ns] | Startup [$\mu$s] | Startup [Takte] | Bandbreite [MByte/s] |
|---|---|---|---|---|---|
| CM-5 | Fat-Tree | 25 | 95 | 3800 | 9 |
| nCube/2 | Hypercube | 25 | 154 | 6200 | 1.7 |
| Cray T3D | 3D-Torus | 7 | 3 | 429 | 128 |
| Paragon | 2D-Gitter | 12 | 25 | 2083 | 171 |
| Supercluster | 2D-Gitter | 50 | 120 | 2400 | 1.0 |
| PowerXplorer | 2D-Gitter | 12 | 98 | 8200 | 1.2 |

In Tabelle 2.3 sind ausgewählte Daten aus [52] angegeben. Zusätzlich wurden mit dem ebenfalls in [52] beschriebenen Benchmarkprogramm die entsprechenden Daten für Parsytec SuperCluster und PowerXPlorer ermittelt. Die Angabe der Startup-Zeit in Taktzyklen des Applikationsprozessors gibt Aufschluß über das Verhältnis von Kommunikations- zu Rechenleistung des Multiprozessorsystems und ist damit neben der maximal erreichbaren Bandbreite ein wesentlicher Parameter für die Skalierbarkeit von parallelen Programmen auf dem Rechnersystem.
Pro Routingknoten wird beim Transport der Nachricht eine bestimmte Zeit benötigt. Die Übertragungszeit $t_{routing}$ für eine Nachrichtenhülle zwischen zwei direkt verbundenen Prozessoren bezeichnet man als *Knotenlatenz* (engl. *per-hop time*). Sie wird durch die Schaltzeit des Routingalgorithmus bzw. der Routinghardware bestimmt, die festlegen müssen, in welchen Übertragungspuffer oder auf welchen Kanal die Nachricht weitergeleitet werden muß. Für die Mehrzahl der aktuellen Multiprozessorsysteme ist die Knotenlatenz vernachlässigbar, weil ein effizientes Routingsystem zugrundeliegt und ein kleiner Netzwerkdurchmesser (vgl. 2.2.3.7) erreicht wird [52], [57].

Tabelle 2.4: *Vergleich von Startup-Zeit und Latenz verschiedener Multiprozessorsysteme, Quelle: [152]*

| System | Topologie | Taktzeit [ns] | Startup [Takte] | Latenz [Takte] |
|---|---|---|---|---|
| CM-5 | Fat-Tree | 25 | 3600 | 114 |
| nCube/2 | Hypercube | 25 | 6400 | 360 |
| Paragon | 2D-Gitter | 12 | 2500 | 83 |

In [152] wird neben der Startup-Zeit auch die Latenzzeit in quasi unbelasteten Verbindungsnetzwerken angegeben (vgl. Tabelle 2.4). Man erkennt, daß die Startup-Zeit die Übertragung von kurzen Datenvektoren (hier: 160 Bit) eindeutig dominiert.
Untersucht wurde von [152] die Kommunikation zwischen zufällig ausgewählten Prozessoren.
Das zugrundegelegte, ideale Multiprozessorsystem besteht aus $p$ gleichartigen Prozessoren und das Kommunikationsnetzwerk besitzt die folgenden Eigenschaften:

- Ein Prozessor kann zu einer Zeit nur mit einem Prozessor kommunizieren.
- Zwischen zwei Prozessoren $i$ und $j$ kann zu einer Zeit eine Nachricht entweder nur von $i$ nach $j$ oder von $j$ nach $i$ übertragen werden.

- Die Datenübertragung auf den Pfaden $(i,j)$ und $(j,i)$ dauert gleich lang und benutzen die physikalischen Pfade $[i,...,j]$ bzw. $[j,...,i]$.
- Die Übertragung einer Nachricht der Länge $n$ auf dem Pfad $(i,j)$ erfordert, falls keine Netzwerkkonflikte auftreten, die Zeit

$$t = t_{startup} + hops * t_{routing} + n * t_{transfer}. \quad (2.6)$$

  *hops* ist die Anzahl der Routingknoten im Pfad $[i,...,j]$.

In Details können die tatsächlichen Eigenschaften von Multiprozessorsystemen vom beschriebenen Modell abweichen. Die Eigenschaften sind so definiert, daß sie auch mit einem transputerbasierten System realisiert werden können. Es gibt Netzwerke die darüberhinaus

- den gleichzeitigen Nachrichtenaustausch auf den Pfaden $(i,j)$ und $(j,i)$ beschleunigen (z.B. nur einmal $t_{startup}$),
- den gleichzeitigen Nachrichtenaustausch zu mehr als einem Prozessor unterstützen (z.B. Transputer mit 4 gleichzeitig ausführbaren Kommunikationen) oder
- unterschiedliche $t_{transfer}$ in Abhängigkeit vom Pfad erlauben (z.B. *Fat Tree*).

Natürlich ist der Kommunikationsaufwand nur für eine konkrete physikalische Netzwerktopologie unter Berücksichtigung des benutzten Routingsystems bestimmbar.

### 2.6.2 Kollektive Allreduce-Operation

Das Konzept der MPI-Implementation auf der ADI-Schnittstelle mit den Basisfunktionen zum Nachrichtenaustausch zwischen jeweils zwei Prozessoren bildet auch die kollektiven Kommunikationsoperationen mit den Punkt-zu-Punkt-Kommunikationsfunktionen ab. Es wird aber alternativ unterstützt, daß Hard- und Softwareentwickler optimierte Versionen dieser Kommunikationsroutinen bereitstellen.
Basis für die MPI-Implementation der häufig benutzten Allreduce-Funktion ist ein Binärbaum. In dem Binärbaum werden zunächst die Daten aufwärts transportiert und dabei gleichzeitig zusammengefaßt. Damit bleibt das transportierte Datenvolumen konstant. Danach wird das an der Wurzel bereitstehende Ergebnis wieder im Baum abwärts transportiert. Der Binärbaum wird einfach aus der fortlaufenden Numerierung der Prozessoren ermittelt. Die benötigte Zeit für diesen Algorithmus ergibt sich bei $p = 2^d$ Prozessoren und einer Nachrichtenlänge $N$ zu

$$T_{Baum-Allreduce} = d[N(t_{transfer} + t_{op}) + t_{startup}] + d[Nt_{transfer} + t_{startup}] \quad (2.7)$$

Der Algorithmus ist effizient für kurze Nachrichten und kleine Prozessorzahlen. Für große Prozessorzahlen und lange Vektoren hat der Algorithmus den Nachteil, daß die meisten Prozessoren den größten Teil der Zeit untätig (engl. *idle*) sind. Für lange Vektorlängen wird die Laufzeit zudem durch die Anteile $dN(2t_{transfer} + t_{op})$ dominiert.
Alternativ hat die Implementation in einem Hypercube mit dimensionsweisem Austausch des zu reduzierenden Vektors den gleichen Aufwand mit

$$T_{Cube-Allreduce} = d[N(2t_{transfer} + t_{op}) + 2t_{startup}] \quad (2.8)$$

Eine entscheidende Reduktion dieses Aufwandes ist mit einer Technik, die als rekursives Halbieren (vgl. [69]) bekannt ist, möglich: Angenommen $N$ sei ein Vielfaches



von $p = 2^d$. Dann teilt zunächst jeder Prozessor seinen Vektorbeitrag in zwei Hälften. Der Prozessor habe die Ordnungsnummer $p_i$. Die Kommunikationsdimension sei $d-1$. In dieser Dimension wird die erste Hälfte verschickt, wenn das $d-1$te Bit des *Graycodes* von $p_i$ gesetzt ist, sonst wird die zweite Hälfte des Vektors verschickt. Danach kombiniert jeder Prozessor die eingetroffene Vektorhälfte mit der eigenen, noch nicht verschickten Vektorhälfte. Das Verfahren wird für die weiteren Dimensionen $d-2,\ldots,0$ jeweils mit den aktualisierten und weiter halbierten Vektorhälften fortgesetzt. Am Ende besitzt jeder Prozessor einen korrekten Anteil des Ergebnisvektors mit der Länge $N/p$. Für acht Prozessoren im Hypercube sind diese Schritte in Abbildung 2.23 dargestellt.

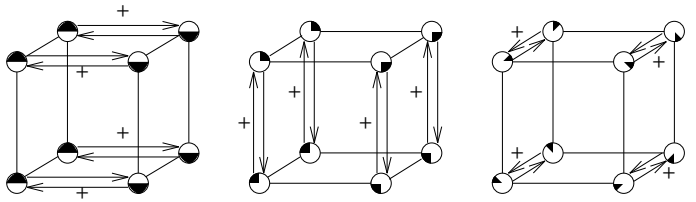

Abbildung 2.23: *Allreduce im Hypercube, rekursives Halbieren*

Danach wird, beginnend mit Dimension 0, der jeweils lokal bekannte Anteil am Ergebnisvektor an den Partner in dieser Dimension verschickt. Dort wird der eintreffende Anteil mit dem lokalen Anteil zusammengesetzt. Mit steigender Dimension $1,\ldots,d-2$ verdoppelt sich die Länge des auszutauschenden Vektors, bis in der Dimension $d-1$ ein Ergebnisteilvektor der Länge $N/2$ ausgetauscht wird. Nach dem letzten Austauschschritt liegt in jedem Prozessor der vollständige Ergebnisvektor vor. Die entsprechenden Schritte für acht Prozessoren sind in Abbildung 2.24 dargestellt.

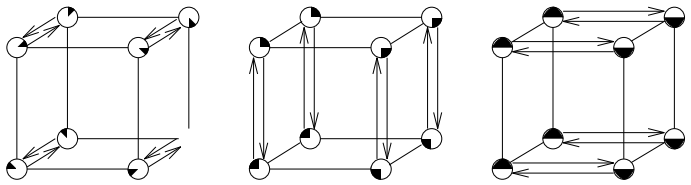

Abbildung 2.24: *Allreduce im Hypercube, rekursives Zusammensetzen*

Der Zeitaufwand für diese Methode ergibt sich zu

$$\begin{aligned} T_{Cube-Rekursiv} &= \sum_{1}^{d}[4t_{startup} + \frac{N}{2^i}(4t_{transfer} + t_{op})] \\ &= 4dt_{startup} + \frac{p-1}{p}N(4t_{transfer} + t_{op}) \end{aligned} \quad (2.9)$$

Im Vergleich zum Hypercube-Algorithmus fällt die doppelte Anzahl an $t_{startup}$ besonders für kurze Vektorlängen ins Gewicht. Offensichtlich ergibt sich in Abhängigkeit von den Hardwareeigenschaften $t_{startup}$, $t_{transfer}$ und $t_{op}$ und den Parametern

Vektorlänge $N$ und Dimension $d$ eine kürzere Laufzeit entweder für den einfachen Algorithmus mit vollem Vektortausch oder den Algorithmus mit rekursiver Halbierung.

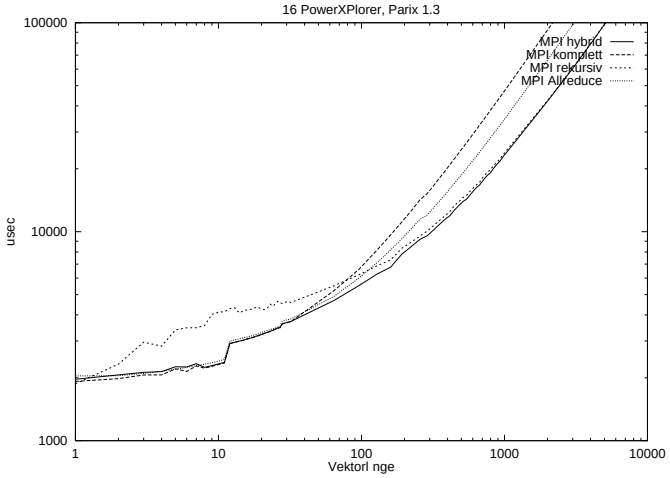


Abbildung 2.25: *Allreduce auf PowerXPlorer, 16 Prozessoren*

Eine hybride Strategie, die in jedem Kommunikationsschritt die vorteilhaftere von beiden Methoden benutzt, wird in [210] für Hypercube-Architekturen vorgeschlagen und in [17] auf Gitter und Tori erweitert. Van de Geijn beweist in [210] die Existenz eines optimalen Verfahrens auf der Basis von initial $k$ Schritten rekursiven Halbierens in den Subcubes der Dimension $d-1,\ldots,d-k$ und anschließend $d-1-k$ Schritten vollständigen Vektortauschs in den nächstkleineren Subcubes. Die Laufzeit des hybriden Algorithmus ergibt sich zu

$$\begin{aligned} T_{Cube-Hybrid} &= \sum_{1}^{d-k-1} (N(2t_{transfer}+t_{op})+2t_{startup}) \\ &\quad + \sum_{d-k}^{d} (4t_{startup} + \frac{N}{2^i}(4t_{transfer}+t_{op})) \end{aligned} \tag{2.10}$$

Die Wahl von $k$ bestimmt sich nun so, daß ausgehend von $1,\ldots,d-k-1$ Schritten des vollständigen Vektortauschs der Schritt $d-k$ mit dem Aufwand

$$T_{d-k,Halb} = 4t_{startup} + \frac{N}{2^{d-k}}(4t_{transfer}+t_{op}) \tag{2.11}$$

günstiger ist, als mit dem Aufwand für den vollen Vektortausch mit

$$T_{d-k,Voll} = 2t_{startup} + N(2t_{transfer}+t_{op}) \tag{2.12}$$

Daraus ergibt sich das Kriterium zur Bestimmung von $k$ zu

$$2t_{startup} \geq \frac{N}{2^{d-k}}[k(2t_{transfer}+t_{op})+2t_{transfer}] \tag{2.13}$$



Dieses Kriterium ist in jedem Schritt des hybriden Austauschs zu kontrollieren.
In den Abbildungen 2.25 und 2.26 sind für 16 bzw. 32 Prozessoren PowerXPlorer die Laufzeiten der vier Algorithmen gegenübergestellt. Deutlich wird die Gleichwertigkeit des Baum- und Hypercube-Algorithmus für die kleinen Vektorlängen. Die Strategie des rekursiven Halbierens ist für große Vektorlängen fast doppelt so schnell wie die Hypercubestrategie. Das hybride Verfahren hat bei extrem kurzen Vektorlängen zusätzliche Vergleiche durchzuführen und ist deshalb geringfügig schlechter als das einfache Verfahren. Im weiteren Verlauf mit mittleren Vektorlängen ist sehr gut der weiche Übergang zur asymptotisch besseren Strategie des rekursiven Halbierens dokumentiert.

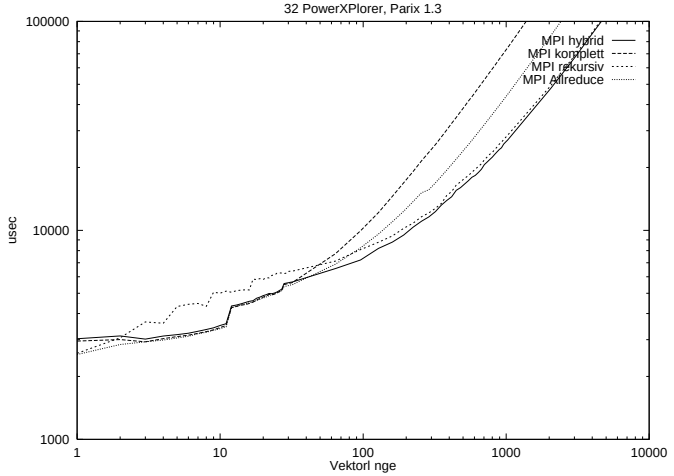


Abbildung 2.26: *Allreduce auf PowerXPlorer, 32 Prozessoren*

Legt man anstatt des in Abschnitt 2.6.1 beschriebenen Modells ein bidirektional kommunizierendes Netzwerk zugrunde (z.B. Intel Paragon), dann ergeben sich folgende Laufzeiten für die oben angeführten Varianten des Allreduce-Algorithmus: Der Baum-Algorithmus gewinnt nichts von dem bidirektionalen Datenaustausch, da die Kommunikation nur in einer Richtung (zunächst im Baum aufwärts, dann im Baum abwärts) erfolgt. Für den Hypercube reduziert sich der Aufwand auf

$$\bar{T}_{Cube-Allreduce} = d[N(t_{transfer}+t_{op})+t_{startup}] \tag{2.14}$$

und ist damit wesentlich günstiger als der entsprechende baumorientierte Algorithmus. Offensichtlich ist aber auch die Laufzeit dieses Algorithmus für lange Vektoren von der Austauschzeit und den mathematischen Operationen dominiert. Der Ablauf der Kommunikation wird in Abbildung 2.27 für 8 Prozessoren im dreidimensionalen Hypercube gezeigt.
Die Strategie des rekursiven Halbierens liefert für eine bidirektionale Kommunikation die Laufzeit

$$\bar{T}_{Cube-Rekursiv} = \sum_{1}^{d} [2t_{startup} + \frac{N}{2^i}(2t_{transfer}+t_{op})]$$

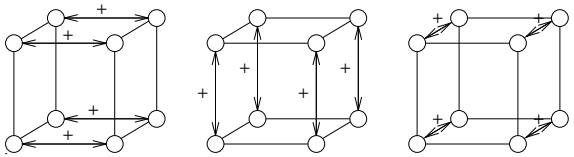

Abbildung 2.27: *Allreduce im Hypercube, bidirektionaler Datenaustausch*

$$= \quad 2dt_{startup} + \frac{p-1}{p} N(2t_{transfer} + t_{op}) \tag{2.15}$$

Der Aufwand für die hybride Strategie berechnet sich nach [210] mit

$$\begin{aligned} \bar{T}_{Cube-Hybrid} \quad = \quad & \sum_{1}^{d-k-1} (N(t_{transfer} + t_{op}) + t_{startup}) \\ & + \sum_{d-k}^{d} (2t_{startup} + \frac{N}{2^i}(2t_{transfer} + t_{op})) \end{aligned} \tag{2.16}$$

und das Kriterium zur Bestimmung von $k$ reduziert sich auf

$$t_{startup} \geq \frac{N}{2^{d-k}}[k(t_{transfer} + t_{op}) + t_{transfer}] \tag{2.17}$$

Analoge Betrachtungen sind für den Vektor-Broadcast sinnvoll (siehe [18], [211], [19]). Die hier dargestellten Erkenntnisse sind sofort anwendbar, wenn man den Broadcast als Teiloperation der Allreduce-Kommunikation auffaßt[5].

## 2.7 Effektivität paralleler Programme

Die Leistungsparameter paralleler Programme werden durch den charakteristischen arithmetischen Aufwand zur Lösung eines Problems bestimmt. Zusätzlich ist der in der Parallelverarbeitung begründeten Aufwand einzubeziehen. Der zusätzliche Aufwand (engl. *overhead*) setzt sich zusammen aus

- der zeitlichen Abstimmung der parallelen Prozesse,
- der Aufgabenverteilung und Lastbalancierung und der
- der notwendigen Kommunikation.

Erfordert die parallele Bearbeitung eines Problems eine enge zeitliche Abstimmung der Prozesse, dann spricht man von synchronisierten Problemen. Die Lösung von linearen Gleichungssystemen gehört zu dieser Problemklasse.
Die **Granularität** bezeichnet das Verhältnis zwischen Kommunikations- und Arithmetikbedarf eines parallelen Verfahrens. Grobgranulare Algorithmen besitzen wenig Kommunikationsbedarf, demzufolge auch wenig Synchronisationspunkte. Die Granularität ist darüberhinaus eine Funktion der Problemgröße. Die Verteilung eines Problems fester Größe auf wenige Prozessoren erfordert weniger Kommunikation, als die Verteilung desselben Problems auf mehr Prozessoren. Mehr Prozessoren erfordern mehr Kommunikation und das führt damit auf feingranulare Parallelität.

[5] Das wird besonders im baumorientierten Allreduce-Algorithmus deutlich. Dort ist das Verteilen des Ergebnis von der Wurzel identisch mit dem Broadcast.



### 2.7.1 Speed-up und Effizienz, Amdahls Gesetz

Vergleicht man die Leistungsfähigkeit von parallelen und seriellen Algorithmen für ein gegebenes Problem, erhält man eine anschauliche Maßzahl für den Vorteil, der aus der Parallelisierung gewonnen werden kann. Diese Maßzahl ist der Speed-up (dt. *Beschleunigung*), der wie folgt definiert ist:

$$S(p) = \frac{Rechenzeit\ auf\ einem\ Prozessor\ T_1}{Rechenzeit\ auf\ p\ Prozessoren\ T_p} \tag{2.18}$$

Der Speed-up ist eine Funktion von $p$. Er beschreibt, wievielmal schneller ein Problem fester Größe auf einem Parallelrechner mit $p$ Prozessoren im Vergleich zu einem Rechner mit einem Prozessor abgearbeitet werden kann.
Der parallelisierte Lösungsansatz kann sich methodisch von einem seriellen Algorithmus unterscheiden. Für manche Applikationen ist es daher sinnvoll, die parallele Implementation mit der besten seriellen Implementation zu vergleichen. Ein anderer Ansatz wäre es, eine serielle Implementation aus der parallelen Implementation abzuleiten und vorauszusetzen, daß die serielle Leistungsfähigkeit von der parallelen Programmversion, abgearbeitet auf einem Prozessor, bestimmt wird.
Der ideal erreichbare Speed-up ist identisch mit der Anzahl der eingesetzten Prozessoren $p$, er wird deshalb auch als linearer Speed-up bezeichnet. Realistische Probleme lassen sich nicht mit idealen Speed-up parallelisieren. Die Zeitverluste aus dem zusätzlichen Kommunikationsaufwand führen zu einem unterlinearen Speed-up. Die Dokumentation von überlinearen Speed-up, d.h. das überproportionale Wachstum der Abarbeitungsgeschwindigkeit beim Übergang auf eine größere Prozessorzahl, weist systematische Fehler auf. Dabei werden zum Beispiel Cache-Effekte ungenügend berücksichtigt, die zu einer erhöhten Rechenleistung führen, die nicht unmittelbar auf die Parallelisierung zurückzuführen sind. Bei der Verteilung eines gegebenen Problems auf eine größere Anzahl von Prozessoren wird die von einem Prozessor zu berechnende Problemgröße reduziert und kann plötzlich wesentlich schneller bearbeitet werden, weil die benötigten Daten besser in den Prozessor-Cache passen. Das ist ein allgemeines Problem moderner RISC-Architekturen, deren Arithmetikleistung primär von der Effizienz des Datenzugriffs bestimmt wird.
Setz man den Speed-up ins Verhältnis zur eingesetzten Prozessorzahl $p$, erhält man die Effizienz

$$E(p) = \frac{S(p)}{p} \tag{2.19}$$

Die Effizienz gibt an, zu welchem Grad das Leistungspotential des Parallelrechners tatsächlich ausgenutzt wird. Üblich ist auch eine Angabe in Prozent. Dabei erlaubt die Bestimmungsgleichung 2.19 nur eine Aussage über den langsamsten Prozessor. Insofern sind die Effizienzuntersuchungen auf die einzelnen Prozessoren auszudehnen, um zu identifizieren, ob lediglich einzelne Prozessoren die Effizienz bestimmen oder ob alle Prozessoren gleichmäßig belastet sind.
Der Computerentwickler Gene Amdahl formulierte 1967 das nach ihm benannte Gesetz [6]. Er teilt die Bearbeitungszeit $T_1$ für eine bestimmte Berechnung in einen nicht weiter parallelisierbaren Anteil $t_{seq}$ und einen parallelen Anteil $t_{par}$

$$T_1 = t_{seq} + t_{par} \tag{2.20}$$

Die beste Laufzeit $T_p$ für $p$ Prozessoren ergibt sich daraus zu

$$T_p = t_{seq} + \frac{t_{par}}{p} \tag{2.21}$$

Setzt man z.B. die Gesamtzeit auf einem Prozessor $T_1$ zu 1 und drückt $t_{par}$ durch $t_{seq}$ aus mit

$$t_{par} = 1 - t_{seq} \tag{2.22}$$

dann gilt für den maximal erreichbaren Speedup

$$S(p) = \lim_{p\to\infty} \frac{1}{t_{seq} + \frac{1-t_{seq}}{p}} = \frac{1}{t_{seq}} \tag{2.23}$$

Amdahls Gesetz bedeutet also, daß unabhängig davon, wieviele Prozessoren eingesetzt werden, zum Beispiel bei 5% nicht parallelisierbarem Berechnungsaufwand $t_{seq}$ der maximale Speedup 20 ist. Praktische Anwendungen besitzen sehr viel weniger nicht parallelisierbare Anteile. Aber lange Zeit wurde in diesem Gesetz ein Hindernis für die Entwicklung von massivparallelen Algorithmen gesehen.
Ein maximaler Speedup von 200 schien zum Beispiel realistisch, da mindestens 0.5% serieller Anteil an der Problembearbeitung allein durch Ein- und Ausgabe gegeben sind. Wäre mit dem Einsatz von mehr als 200 Prozessoren kein praktischer Gewinn erreichbar, würden Parallelrechner zwar für gewisse Spezialaufgaben, nicht aber als Hochleistungsrechner für eine Vielzahl von numerischen Applikationen geeignet sein.
Der wesentliche Fehler in der Schlußfolgerung aus Amdahls Gesetz ist die Annahme, daß der Anteil $t_{seq}$ konstant ist und nicht von der Prozessorzahl und der Problemgröße abhängt. Dieses Phänomen wird mit dem oben definierten Speed-up nicht berücksichtigt, der eine feste Problemgröße voraussetzt (engl. *fixed size speed-up*).
Die Aufteilung in konstante Anteile beschleunigbarer und nicht beschleunigbarer Programmteile ist für die Vektorisierung einer Applikation sinnvoll. Der Vektorisierungsgrad bestimmt, wieviel Nutzen aus der Programmabarbeitung auf einem Vektorrechner gezogen werden kann. Dieser Anteil ist für eine bestimmte Implementation konstant und wird nur dadurch relativiert, wie die Größe des Problems zur Größe der Vektorregister paßt.

### 2.7.2 Skalierbarkeit

Für ein Problem fester Größe wird auf einer steigenden Anzahl von parallel arbeitenden Prozessoren nur ein bestimmter Grenzwert oder Maximalwert des Speed-up erreicht, weil

- der zusätzliche Aufwand durch die Parallelisierung mit zunehmender Prozessorzahl steigt oder
- der in einer Applikation inhärente Parallelisierungsgrad erreicht ist.

Der letztere Fall soll aus der weiteren Betrachtung ausgeschlossen werden.
Es ist bekannt, daß man bei einer Vielzahl von praktischen Anwendungen und Parallelrechnern den relativen Zusatzaufwand für die parallele Abarbeitung durch Skalierung der Problemgröße senken kann. Erhöht man für einen parallelen Algorithmus den Gesamtaufwand der lokalen, d.h der parallelen Berechnungen, minimiert man den Anteil der nicht parallelen Programmteile.
Die Regel von Gustafson [89] besagt, daß bei geeigneter Skalierung der Problemgröße $n$ auf einer bestimmten Prozessoranzahl $p$ jeder beliebige Speed-up, beliebig nahe dem idealen Speed-up $S_{ideal} = p$, eingestellt werden kann. Bleibt man bei der Terminologie von Amdahls Gesetz und definiert einen konstanten seriellen Programmteil $t_{seq}$ und den parallelisierbaren Programmteil als Funktion der Problemgröße $n$ und der Prozessorzahl $p$ mit $t_{par}(n,p)$ dann erhält man für den Speed-up als Funktion von $n$ und $p$

$$S(n,p) = \frac{t_{seq} + t_{par}(n,1)}{t_{seq} + t_{par}(n,p)} \tag{2.24}$$

Ohne Einschränkung der Allgemeinheit soll für $t_{par}(n,p) = n^2/p$ gelten, d.h. der parallele Anteil steigt quadratisch mit der Problemgröße und kann ideal auf die



Prozessoren aufgeteilt werden. Für diesen Fall ergibt sich

$$\begin{aligned} \lim_{n\to\infty} S(n,p) &= \lim_{n\to\infty} \frac{t_{seq} + n^2}{t_{seq} + \frac{n^2}{p}} \\ &= \lim_{n\to\infty} \frac{pt_{seq} + pn^2}{pt_{seq} + n^2} \\ &= p, \end{aligned}$$

d.h. der ideale Speed-up. Relevant ist diese Gesetzmäßigkeit, wenn ein Speed-up in der gewünschten Nähe zum idealen Speed-up für praktisch realisierbare Problemgrößen erreicht wird.
Die maximale Problemgröße wird im wesentlichen vom zur Verfügung stehenden Speicherplatz bestimmt, der in homogenen Multiprozessorsystemen proportional zur Prozessorzahl ist. Da moderne Multiprozessoren mit Standardkomponenten ausgerüstet sind (z.B. auch Speicherbausteine), ist die Ausstattung mit ausreichend lokalem Hauptspeicher lediglich ein Problem des Hardwarelayouts, das Platz für die notwendigen Speicherbausteine vorsehen muß.
Die Skalierbarkeit eines Algorithmus ist eine Maßzahl dafür, wie sich die Laufzeit bei einer konstanten Problemgröße pro Prozessor $n_p$ entwickelt. Die Skalierbarkeit (engl. *scale-up*) ist definiert

$$S_n(p) = \frac{T_p}{T_1} \quad \text{für} \quad n_p = \text{const.}, \quad n = pn_p \tag{2.25}$$

Sie gibt Aufschluß über die mit einem parallelen Algorithmus beherrschbare Problemgrößen unter begrenzten Ressourcen (Rechenzeit und Prozessorzahl, damit auch Speicherplatz).

### 2.7.3 Isoeffizienz

Als direkt am Algorithmus orientierte Größe wird die Isoeffizienz definiert, die angibt, wie sich die Problemgröße entwickeln muß, um die gleiche Effizienz auf verschiedenen Prozessorzahlen zu erreichen. Kumar et.al. [128], [127] entwickelten auf Basis der Isoeffizienz eine Metrik zum Vergleich von parallelen Algorithmen.
Definiert man den spezifischen Zeitaufwand zur Lösung eines Problems mit dem besten bekannten, seriellen Algorithmus mit $W(n)$ und die Abarbeitungszeit eines parallelen Algorithmus zur Lösung desselben Problems auf $p$ Prozessoren mit $T_p(n)$, dann ergibt sich der Speed-up zu $S(p) = W/T_p$ (vgl. 2.18). $W(n)$ wird als Funktion der Problemgröße definiert. Es ist oft ausreichend, die Ordnung dieser Abhängigkeit zu kennen, wie z.B. den charakteristischen Aufwand der Cholesky-Faktorisierung einer Bandmatrix mit $O(nb^2)$.
In der gleichen Weise kann der für die parallele Lösung erforderliche, zusätzliche Aufwand mit $T_O(W,p) = pT_p - W$ bestimmt werden. Er schließt sowohl den oben definierten *Overhaed* als auch den nicht parallelisierbaren Aufwand ein. Die Effizienz ergibt sich zu (vgl. 2.19)

$$E = \frac{W}{T_p} = \frac{W}{W + T_O(W,p)}. \tag{2.26}$$

Die Effizienz der Lösung eines Problems fester Größe wird kleiner werden, wenn die Prozessorzahl erhöht wird. Die Ursache liegt im steigenden *Overhead*. Für eine feste Prozessorzahl $p$ kann mit steigender Problemgröße $W$ eine höhere Effizienz erreicht werden, wenn der Overhead $T_O(W,p)$ nicht gleichmäßig mit $W$ steigt. Ein Algorithmus, der sich auf einem bestimmten Parallelrechner so verhält, heißt **skalierbar**.
Für einen Algorithmus und einen bestimmten Parallelrechner (ein **paralleles System**) kann man angeben, in welchem Maße die Problemgröße gesteigert werden

muß, um bei steigender Prozessorzahl gleiche Effizienz zu erreichen. Diese Abhängigkeit kann zum Beispiel exponentiell sein. Damit ist das parallele System schlecht skalierbar, weil das Problem für eine größere Prozessorzahl ungleich viel größer sein muß, so daß es unter Umständen gar nicht mehr mit den anderen Ressourcen (vor allem Speicherplatz) beherrschbar ist.
Andererseits charakterisiert eine lineare Abhängigkeit von Problemgröße und Prozessorzahl für konstante Effizienz ein sehr gut skalierbares paralleles System.
Die Skalierbarkeit eines parallelen Systems drückt sich also im Verhältnis von Overhead $T_O(W,p)$ zum algorithmischen Aufwand aus. Für eine feste Effizienz $0 < E \leq 1$ folgt aus (2.26)

$$E = \frac{1}{1 + \frac{T_O(W,p)}{W}} \tag{2.27}$$

Daraus folgt Proportionalität von $W$ und $P_O(W,p)$ bzw.

$$W = eT_O(W,p), \tag{2.28}$$

und

$$e = \frac{E}{1-E}, \tag{2.29}$$

Diese Isoeffizienzmetrik bestimmt die Effizienz als einen funktionalen Zusammenhang zwischen der Problemgröße $W$ und der Prozessoranzahl $p$. Damit kann ein paralleles System in einer kompakten Form charakterisiert werden. Der Vorteil besteht in der Berücksichtigung von Hard- und Softwareeigenschaften in dieser Metrik. Mit der Isoeffizienzanalyse kann die Leistungsfähigkeit eines Algorithmus mit einigen wenigen Prozessoren getestet werden. Aus der theoretischen Abschätzung des Overhead läßt sich dann die Skalierbarkeit des parallelen Systems einschätzen oder das Laufzeitverhalten auf einem anderen Parallelrechner bestimmen.
Die Isoeffizienzanalyse ist ein geeignetes Mittel für Parameterstudien zur Untersuchung des Einflusses von Hardwareänderungen. Die Auswirkungen von veränderter Prozessor- und Kommunikationsleistung auf die Effizienz eines Algorithmus lassen sich damit im voraus bestimmen.

## 2.8 Zusammenfassung

Die Grundlagen der Parallelverarbeitung wurden im Hinblick auf die Architektur der existierenden Parallelrechner mit verteiltem Arbeitsspeicher dargestellt. Ein zentrales Problem stellte dabei die Topologie der Verbindungsnetzwerke dar. Die existierenden parallelen Programmierparadigmen unterscheiden sich hinsichtlich ihres Abstraktionsgrades von dem zugrundeliegenden physikalischen Verbindungsnetzwerk. Mit Schwerpunktlegung auf das explizite Message-Passing-Paradigma wurden einzelne Aspekte einer portablen Programmierschnittstelle und der Kommunikation auf Parallelrechnern dargestellt. Abschließend wurden Maßzahlen und Verfahren zur Bestimmung der Effizienz paralleler Programme dargestellt.

# Kapitel 3

# Gebietszerlegungs- und Abbildungsverfahren

Die Methode der finiten Elemente hat sich aufgrund ihrer Universalität und ihrer Eignung für die Bearbeitung auf elektronischen Rechenanlagen in der Praxis durchgesetzt. Voraussetzung für die Anwendung der Methode auf Parallelrechnern ist die geeignete Problemzerlegung. Dabei unterscheidet man die Gebietszerlegung des Finite-Element-Netzes oder die Datenzerlegung der Matrizen und Vektoren. Die Partitionen des Problems sind anschließend auf die Prozessoren eines Parallelrechners abzubilden. Dabei ist zu berücksichtigen, daß eine geeignete Abbildung häufig benutzte Kommunikationspfade möglichst kurzen physikalischen Kommunikationsverbindungen zuordnet. Kommunikationsnetzwerke moderner Parallelrechner mit ihrem gleichbleibend hohem Datendurchsatz unabhängig von der physikalischen Zuordnung von Prozessen und ihren Kommunikationsverbindungen auf die Hardware lassen dieses Problem aber zunehmend in den Hintergrund treten.
Der Begriff Gebietszerlegung hat im Zusammenhang mit Lösungstechniken für partielle Differentialgleichungen mehrere Bedeutungen (vgl. [193]), das sind u.a.:

- Für die Parallelverarbeitung bedeutet Gebietszerlegung die Aufteilung der Daten des Berechnungsmodells auf die einzelnen Prozessoren. Dabei sind die Datenstrukturen zu verteilen und zusätzliche Verwaltungsstrukturen erforderlich. Die Gebietszerlegung hat in diesem Zusammenhang keine unmittelbare Beziehung zur Lösungstechnik. Der Begriff der Datenzerlegung oder Partitionierung beschreibt diesen Sachverhalt deshalb besser.

- Für die Konstruktion von Vorkonditionierungen für iterative Lösungsverfahren bedeutet Gebietszerlegung die Unterteilung der Lösung des großen linearen Gesamtproblems in eine Anzahl kleinerer Probleme. Die Lösungen der Teilprobleme werden zur Konstruktion von Vorkonditionierungen benutzt, um das Gesamtproblem zu lösen. Gebietszerlegung bezeichnet in diesem Zusammenhang lediglich die Lösungstechnik für das aus der Diskretisierung der partiellene Diffentialgleichung gewonnene algebraische Gleichungssystem.

Beide Aspekte haben nebeneinander innerhalb eines parallelen Programms ihre Bedeutung. Das vorliegende Kapitel beschäftigt sich mit Aspekten der Datenzerlegung bzw. Partitionierung. Im Zentrum stehen daher die geometrisch und topologisch motivierten Verfahren zur Zerlegung eines Finite-Element-Berechnungsmodells und zur effektiven Abbildung auf ein Multiprozessornetzwerk. In Kapitel 5 wird der zweite Aspekt zur Konstruktion effizienter Gleichungslöser in den Mittelpunkt gestellt.

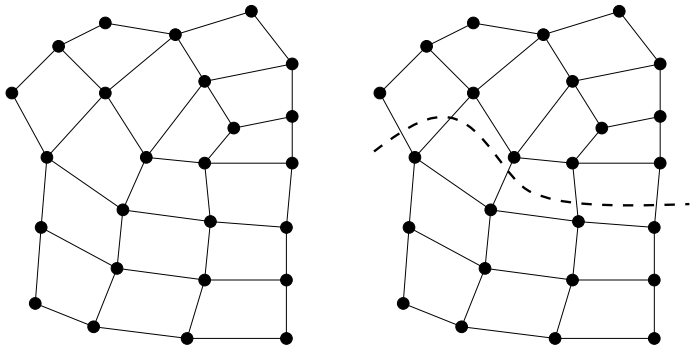

Abbildung 3.1: *Graphpartitionierung*

## 3.1 Gebietszerlegung

### 3.1.1 Grundlagen

#### 3.1.1.1 Formale Problembeschreibung

Formal kann das Partitionierungsproblem mit den Mitteln der Graphentheorie beschrieben werden. Ein unstrukturiertes finite Elementnetz wird als ungerichteter Graph beschrieben. Dieser Graph wird in Subgraphen zerlegt. Entsprechend läßt sich das Problem wie folgt formulieren:
Für einen gegebenen, ungerichteten Graphen $< V(G), E(G) >$, besteht das Partitionierungsproblem in der Bestimmung von disjunkten Teilmengen von $V(G)$, $P = V_i$ unter der Einhaltung von bestimmten Randbedingungen, z.B. von:

Lastbalance:

$$min \sum_i (V_i^2 - \bar{V}^2)$$

Interprozessorkommunikation:

$$min \sum_{i,j} |C_{i,j}| \quad \forall C_{i,j} = \{vw | v \in V_i, w \in V_j\}$$

Netzwerkkomplexität:

$$min \sum_{i,j} N_{i,j} \quad \forall N_{i,j} = \begin{cases} 1: \exists vw | v \in V_i, w \in V_j \\ 0: \quad \text{sonst.} \end{cases}$$

Für $i = 2$ gilt $min \sum_i (V_i^2 - \bar{V}^2) \leq 1$ und das Problem heißt Bisektionsproblem. Durch fortgesetzte, rekursive $n$-fache Anwendung des Bisektionsalgorithmus kann eine Partitionierung in $2^n$ Teilgebiete erzeugt werden.
Zur Verdeutlichung diene das Beispiel in Abbildung 3.1.
Das Partitionierungsproblem ist NP-vollständig (vgl. [137]). Praktisch bedeutet diese Aussage, daß das Partitionierungsproblem nicht mit polynomialem Aufwand von einem deterministischen Algorithmus gelöst werden kann.
Es ist trotzdem eines der aktivsten Teilgebiete aktueller Forschung im Bereich Parallelverarbeitung. Es sind eine Reihe von heuristischen Verfahren entwickelt worden, die das Problem in guter Qualität und für viele praktische Aufgabenstellungen



zufriedenstellend lösen können. Man unterscheidet global und lokal wirkende Heuristiken. Während globale Methoden eine Partitionierung (z.B. einen Bisektion) direkt erzeugen, verbessern lokale Methoden (Optimierungsverfahren, Kernighan-Lin-Heuristik, Algorithmus der hilfreichen Mengen s.u.) eine gegebene Partitionierung.

#### 3.1.1.2 Bewertungskriterien

Kennzeichnend für die mit der Methode der finiten Elemente behandelten Probleme sind komplexe Geometrien, die oft mehrfachen topologischen Zusammenhang aufweisen. Lokale Verfeinerungen führen zu unterschiedlich detaillierter Diskretisierung der Geometrie. Damit ist der Finite-Element-Graph stark strukturiert und unregelmäßig.
Für eine effiziente parallelisierte Bearbeitung ist es erforderlich, Kriterien zur Bewertung der gefundenen Partitionierung und der resultierenden Interprozessorkommunikation zu definieren. Diese Kriterien lassen sich in rechnerarchitekturbezogene, problembezogene und methodenbezogene Anforderungen differenzieren. Im folgenden werden die einzelnen Anforderungsklassen dargestellt.
**Problembezogene Anforderungen:**

1. Die Teilprobleme sollen so partitioniert werden, daß die resultierende Bandbreite eines jeden Teilproblems nur ein Teil der Bandbreite des Gesamtsystems ist.

2. Unregelmäßige Geometrie und beliebige Diskretisierungen sollen im Sinne eines universellen Werkzeuges beherrscht werden.

**Rechnerarchitekturbezogene Anforderungen:**

3. Die Rechenlast pro Teilgebiet soll ausgewogen verteilt sein, um die Gesamtrechenlast gleichmäßig auf alle Prozessoren zu verteilen.

4. Die Anzahl der Koppelknoten soll so klein wie möglich sein, um den Umfang der Interprozessorkommunikation zu minimieren.

5. Die Anzahl benachbarter Teilgebiete soll minimiert werden, um die Anzahl der benötigten Verbindungen so gering wie möglich zu halten und die Abbildung des Kommunikationsgraphen auf die Prozessortopologie zu optimieren.

**Methodenbezogene Anforderungen:**

6. Die Teilprobleme sollen möglichst viele Koppelknoten haben, um den Zusammenhang zwischen ihnen gut zu vermitteln. Damit wird in jedem Iterationsschritt die Verbesserung in möglichst vielen Teilgebieten gleichzeitig erreicht.

7. Die Teilprobleme sollen geometrisch möglichst ausgewogen ausgebildet sein (Vermeidung von flachen oder langgestreckten Teilgebieten), um die Kondition des Problems im Teilgebiet so klein wie möglich zu halten.

8. Die Berandungen der Teilgebiete sollen möglichst glatt sein, um die Kondition des Interfaceproblems so klein wie möglich zu halten.

Offensichtlich sind diese Kriterien nicht ohne Einschränkungen miteinander vereinbar. Deshalb kann eine gefundene Partitionierung nur einen Kompromiß darstellen. Zur Beurteilung der Zerlegung ist die Kenntnis der folgenden Parameter des benutzten Parallelrechners erforderlich:

- Speicherbandbreite und Speichertransferraten für verschiedene Vektorlängen

- Verarbeitungsgeschwindigkeit der Prozessoren
- Startup-Zeit und Transferleistung der Punkt-zu-Punkt-Kommunikation (unter Umständen in Abhängigkeit von der Entfernung der Kommunikationspartner)
- Einfluß von Linkkonflikten auf den Datendurchsatz im Kommunikationsnetz

Diese Parameter beeinflussen die Wertigkeit der Kriterien. Für eine optimalen Partitionierung ist eine Auswahl von Partitionierungsstrategien erforderlich, um unterschiedliche Probleme auf einer Reihe von Parallelrechnern zu behandeln.

#### 3.1.1.3 Taxonomie der Gebietszerlegungsverfahren

Gebietszerlegungsverfahren lassen sich entsprechend ihrem Anwendungsgebiet als statische oder dynamische Verfahren charakterisieren. Während die statische Gebietszerlegung einmalig das Paritionierungsproblem löst, hat die dynamische Gebietszerlegung dieses Problem während der Simulation zu lösen, wenn sich durch Systemveränderungen Ungleichgewichte in der Lastverteilung ergeben. Ursachen dafür sind:

- adaptive Finite-Element-Verfahren (h-Adaptivität mit wechselnder Elementanzahl bzw. p-Adaptivität mit wechselndem Berechnungsaufwand pro Element),
- nichtlineare Simulationen mit zustandsabhängigem Systemverhalten, das auf Knoten- oder Elementebene zu unterschiedlichem Berechnungsaufwand in Abhängigkeit vom physikalischen Verhalten führt oder
- zeitabhängige Simulationen mit veränderlichem Systemverhalten (wechselnde Randbedingungen, Berechnungsaufwand in Abhängigkeit von der zeitlichen Entwicklung des simulierten Phänomens).

Ein weiterer Unterschied ist in der Verteilung der Ausgangsdaten gegeben. Für die statische Gebietszerlegung liegt die Problembeschreibung in der Regel noch nicht paritioniert vor. Dagegen ist bei der dynamischen Gebietszerlegung das Problem bereits partitioniert und ist auf einem Parallelrechner geladen. Die Aufteilung des Problems auf die Prozessoren erfüllt aber nicht die o.g. Anforderungen an eine ausgewogene Verteilung.
Darüberhinaus unterscheidet man globale und lokale Verfahren. Globale Gebietszerlegungsverfahren arbeiten mit Informationen über den gesamten Graphen während lokale Verfahren die Partitionierung auf einem beschränkten Teilgraphen durchführen. Lokale Verfahren sind deshalb in der Regel besser parallelisierbar.

### 3.1.2 Statische Verfahren

Die Ansätze zur Lösung des statischen Partitionierungsproblems lassen sich entsprechend ihrer algorithmischen Grundlagen in vier Gruppen gliedern, die im folgenden exemplarisch mit wichtigen Vertretern diskutiert werden:

- geometrisch basierte Heuristiken,
- graphbasierte Heuristiken,
- algebraisch basierte Heuristiken und
- Partitionierung durch Optimierung.



Eine spezielle Klasse von Partitionierungsalgorithmen sind die Bisektionsverfahren. Mit $n$-facher rekursiver Anwendung sind Zerlegungen in $2^n$ Teilgebiete möglich. Die folgenden, rekursiven Bisektionsverfahren sind in die oben genannten Gruppen eingeordnet:

- rekursive Koordinaten-Bisektion (RKB, Algorithmus 1)
- rekursive Graph-Bisektion (RGB, Algorithmus 5)
- rekursive Spektral-Bisektion (RSB, Algorithmus 9).

Für jede Partitionierungsmethode werden Aspekte der Parallelisierung des Algorithmus selbst diskutiert.

#### 3.1.2.1 Geometrisch basierte Heuristiken

**3.1.2.1.1 Algorithmus 1: Geometrisches Sortieren und rekursive Koordinaten-Bisektion** Ein finites Elementnetz kann durch die Position der Knoten im Raum und die Zuordnung der Elemente zu diesen Knoten beschrieben werden. Sortiert man die Elemente entlang von gewählten Koordinatenachsen und ordnet sie Teilgebieten zu, läßt sich ein einfaches und effektives Bipartitionsverfahren konstruieren.
Erstmals beschrieben für die Zerlegung von zweidimensionalen Gebieten von Bokhari [28], wird diese Methode auch $P \times Q$ Partitionierung genannt [41]. Dabei wird zunächst das Gebiet in $P$ Teilgebiete mit gleicher Knotenzahl entlang der x-Achse partitioniert und danach werden die $P$ Teilgebiete in jeweils $Q$ Partitionen in Richtung der y-Achse zerlegt. Berger und Bokhari [25] bezeichnen die Methode als orthogonale, rekursive Bisektion, die zum Beispiel auch zur Datenpartitionierung für Matrixoperationen benutzt wird [222].
Die Technik ist sehr schnell ($O(logn)$) und garantiert eine gute Lastbalance, erfüllt aber im allgemeinen nicht die Forderung an die kompakten Teilgebiete. Es können sehr langgestreckte, schmale Gebiete entstehen. Diese Gebiete haben lange Koppelränder und erfordern deshalb einen großen Kommunikationsaufwand (siehe Abbildung 3.2). Die Ursache liegt in der Vernachlässigung des Elementzusammenhangs bei der Zerlegung. Bokhari gibt in [25] eine obere Grenze für die Anzahl der benachbarten Gebiete mit $\sqrt{p}$ an.
Die Methode kann mit zwei unterschiedlichen Verallgemeinerungen verbessert werden (vgl. Abschnitt 3.1.2.1.2 und Abschnitt 3.1.2.1.3).

**3.1.2.1.2 Algorithmus 2: Unbalancierte rekursive Bisektion** Jones und Plassman [113] schlagen vor, die rekursive Partitionierung nicht in gleichgroße Abschnitte vorzunehmen, sondern stattdessen die Partitionierung zu benutzen, die die besten geometrischen Eigenschaften aufweist. Als Maß für die Güte der Teilgebietsgeometrie wird z.B für den zweidimensionalen Fall das maximale Verhältnis der Ausdehnung des Teilgebietes (engl. *aspect ratio a.r.*) definiert [112]

$$a.r = max(\frac{h}{v}, \frac{v}{h}) \tag{3.1}$$

mit $h$ und $v$ als maximale horizontale bzw. vertikale Ausdehnung.
Aus den $p$ möglichen Kombinationen der Partitionierung in Teilgebiete mit jeweils $\frac{nk}{p}$ und $\frac{n(p-k)}{p}$ Elementen, wobei $n = |V|$, $p$ die Anzahl von Prozessoren und $k \in \{1, 2, ..., p-1\}$, wird die Variante mit dem besten *aspect ratio* ausgewählt. In den so gewonnenen Teilgebieten wird die Partitionierung fortgesetzt. Dabei sind die Teilaufgaben entsprechend dem gewählten Teilungsverhältnis gegebenenfalls nicht

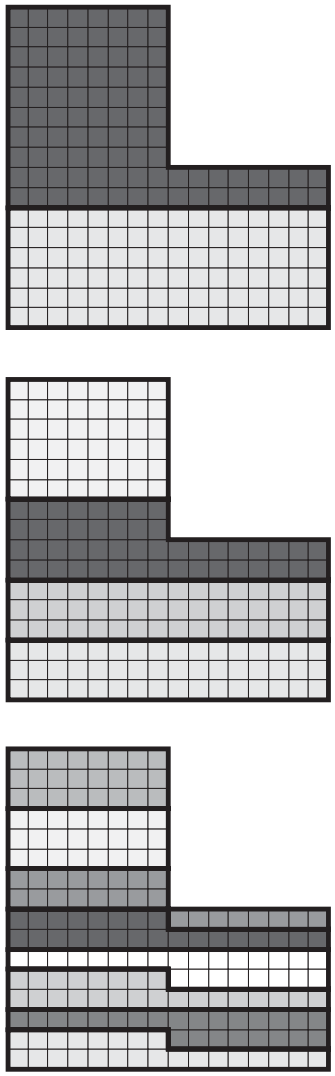

Abbildung 3.2: *L-Gebiet, Partitionierung in 2,4 und 8 Teilgebiete mit Koordinaten-Bisektion*



mehr balanciert. Das Verfahren ist deshalb kein echtes Bisektionsverfahren (Lastverteilungsdifferenz $\geq 1$).
Das Endresultat der Zerlegung mit diesem Algorithmus sind, wie im Originalalgorithmus, Teilgebiete mit perfekt verteilter Last. Die Teilgebiete haben aber wesentlich ausgewogenere Seitenverhältnisse.
Das Verfahren ist ohne Probleme parallelisierbar. Die knotenbezogenen Algorithmen können perfekt parallelisiert werden. Das Festlegen der Teilungskriterien erfordert globale Kommunikation.

**3.1.2.1.3 Algorithmus 3: Schwerachsenmethode** Dieser verallgemeinerte Algorithmus 1 wird zum Beispiel in [59] beschrieben und benutzt nicht die gegebenen Koordinatenachsen, sondern die Schwerachsen des Netzes, die als Eigenvektoren $I_1$, $I_2$ und $I_3$ der $3 \times 3$ Matrix

$$I = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{bmatrix} \tag{3.2}$$

bestimmt werden können. Die Matrixelemente lassen sich effizient nach Anwendung des Gaußschen Integralsatz auf die diskreten Knotenkoordinaten $x_i$, $y_i$ und $z_i$ und den Schwerpunkt des Netzes $x_s$, $y_s$ und $z_s$ wie folgt berechnen:

$$\begin{aligned} I_{xx} &= \sum_i (y_i - y_s)^2 + (z_i - z_s)^2 \\ I_{yy} &= \sum_i (x_i - x_s)^2 + (z_i - z_s)^2 \\ I_{zz} &= \sum_i (x_i - x_s)^2 + (y_i - y_s)^2 \\ I_{xy} = I_{yx} &= \sum_i (x_i - x_s)(y_i - y_s)^2 \\ I_{xz} = I_{zx} &= \sum_i (x_i - x_s)(z_i - z_s)^2 \\ I_{yz} = I_{zy} &= \sum_i (y_i - y_s)(z_i - z_s)^2 \end{aligned} \tag{3.3}$$

Auch dieser Algorithmus [159], [23], [191] kann rekursiv angewendet werden, um mehr als zwei Teilgebiete zu erzeugen (siehe Abbildung 3.3). Die Qualität der Zerlegung ist problemabhängig und kann von der Auswahl und der Reihenfolge der Anwendung der Achsen abhängen (vgl. 2D- und 3D-Beispiele nach Farhat in [59]). Sie kann entscheidend durch geeignete Nachbehandlung der Teilergebnisse in jeder Rekursionsstufe mit Algorithmus 7 oder 8 verbessert werden. Diekmann gibt für verschiedene Verbesserungsstrategien Vergleichswerte an [50].
In [40] wird die Verwendung von problembezogenen, berandungskonformen krummlinigen Koordinatensystemen vorgeschlagen. Damit kann für die Klasse von mit konformen Abbildungsverfahren generierten Finite-Element-Netzen eine elegante Partitionierung angegeben werden. Die Schwerachsenmethode wird auf das gleiche krummlinige Koordinatensystem, das für die Generierung des Netzes benutzt wurde, angewendet.

### 3.1.2.2 Graphbasierte Heuristiken

**3.1.2.2.1 Algorithmus 4: Greedy-Algorithmus** Dieser Algorithmus benutzt die Information über den Netzzusammenhang und wurde von Farhat in [58] vollständig beschrieben und mit einer Programmversion veröffentlicht.

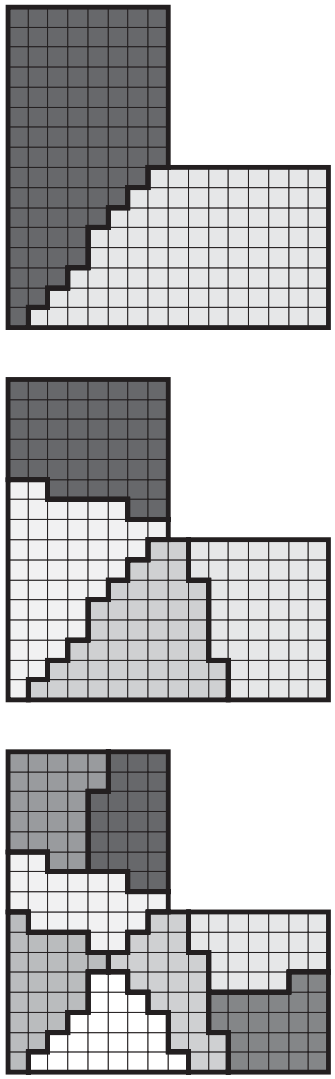

Abbildung 3.3: *L-Gebiet, Partitionierung in 2,4 und 8 Teilgebiete mit Schwerachsenmethode*



Für jeden finiten Elementknoten wird ein Gewicht entsprechend der Anzahl anschließender Elemente bestimmt. Ein Knoten mit minimalen Gewicht wird als Startpunkt ausgewählt. Danach werden finite Elemente entsprechender Anzahl den Partitionen zugeordnet, indem alle anschließenden Elemente des aktuellen Knotens zugeordnet werden und die bisher nicht benutzten Elementknoten als Kandidaten für die Verteilung im nächsten Durchlauf bereitgestellt werden. Das Gewicht der Knotenkandidaten wird entsprechend der Anzahl der noch nicht verteilten, angrenzenden Elemente korrigiert und der Knoten mit dem geringsten Gewicht wird als nächster behandelt. Das Verfahren wird fortgesetzt, bis alle Elemente verteilt worden sind.
Der Algorithmus arbeitet sehr schnell ($O(n)$), weil jeder Knoten nur genau einmal mit seinen noch nicht verteilten Knotennachbarn traversiert werden muß.
Der Algorithmus liefert topologisch „runde“ Partitionen. Es ist aber oft zu beobachten, daß das letzte Teilgebiet nicht zusammenhängend ist. Der Algorithmus wertet keine globalen Informationen über das Problem aus, sondern benutzt lediglich die im Graphen abgelegten lokalen Informationen über den Zusammenhang des unstrukturierten Netzes. So entsteht bei der sukzessiven Aufteilung des Gesamtgebiets auf die Teilgebiete ein Rest, dessen Zusammenhang sich mehr oder weniger zufällig ergibt.
Die Erweiterung des Standardalgorithmus um die Ausnutzung geometrische Informationen beschreiben Al-Nasra und Nguyen in [3]. Mit diesem Ansatz läßt sich das Problem des letzten Gebietes lösen.

**3.1.2.2.2 Algorithmus 5: Rekursive Graph-Bisektion** Das Verfahren entspricht im Prinzip dem Greedy-Algorithmus, mit dem Unterschied, daß die in jedem Schritt des Algorithmus zu bildenden Teilgebiete nicht sofort die Größe $n/p$ haben, sondern daß durch rekursive Bisektion in $log_2 p$ Schritten die gewünschte Partitionierung erzeugt wird. Ausgehend von einem gewählten Startpunkt, werden solange benachbarte Elemente einer Partition zugeordnet, bis die Hälfte der zu verteilenden Elemente vergeben ist. Das Verfahren wird in beiden Teilgebieten rekursiv fortgesetzt.
Dieser graphentheoretisch begründete Algorithmus [191] bietet Vorteile gegenüber Algorithmus 1, weil er die Elementzusammenhänge unabhängig von den geometrischen Verhältnissen betrachtet. Mindestens eines der in jedem Partitionierungsschritt erzeugten Gebiete ist zusammenhängend. Gegenüber Algorithmus 5 ist die globale Wirkung der rekursiven Graph-Bisektion vorteilhaft.
Die Wahl einer geeigneten Zerlegung in jedem Bisektionsschritt erfordert die Bestimmung des Durchmessers eines beliebigen Graphen, ein ebenfalls NP-vollständiges Problem, so daß auch hier nur suboptimale Ergebnisse erwartet werden können.
Die Komplexität wird entscheidend von der Implementation auf jedem Rekursionslevel bestimmt, die ideal als lineare *breadth-first*-Traversierung [204] gestaltet werden kann, so daß sich die Komplexität zu $(O(nlogn)$ ergibt.

**3.1.2.2.3 Algorithmus 6: Bandbreiten-Algorithmus** Ein Algorithmus zur Minimierung der Bandbreite[1] kann ebenfalls benutzt werden, um eine geeignete Partitionierung zu erzeugen. Diese Algorithmen bestimmen eine Permutation der Knotenanordnung mit minimaler Indexdifferenz. Entsprechend entstehen bei einer fortlaufenden Aufteilung der Knoten auf Prozessoren Teilgebiete mit wenigen Nachbarn und kurzen Koppelrändern.
Da der Algorithmus dazu neigt, schlecht konditionierte Teilgebiete (lang und schmal, aber mit wenig Nachbarn) zu liefern, ist seine Anwendung auf Lösungsmethoden beschränkt, die entscheidend von der Größe des Koppelproblems abhängen und unempfindlich gegenüber der Form der Teilgebiete sind.

[1] z.B. Reverse Cuthill-McKee, Implementation siehe [80] oder [185], vergleichende Übersicht verschiedener anderer Algorithmen siehe z.B. Bremer [32]

**3.1.2.2.4 Algorithmus 7: Kernighan-Lin-Heuristik** Kernighan und Lin haben bereits 1970 einen Partitionierungsalgorithmus veröffentlicht [122], der wie Algorithmus 4 das Gewicht von Knoten auswertet und als lokale Optimierungsstrategie arbeitet. Eine gegebene Partitionierung wird durch punktuellen Austausch von Knoten zwischen den Subgraphen $G_1$ und $G_2$ verbessert. Das Gewicht eines Knotens wird wie folgt definiert:

$$diff(v) = ext(v) - int(v) \tag{3.4}$$

Die Anzahl der Kopplungen verändert sich um $diff(v)$, wenn $v$ zwischen den beiden Subgraphen ausgetauscht wird. Es sei für ein Knotenpaar $(u,v)|u \in G_1, v \in G_2$ die Änderung der Kopplungszahl gegeben mit

$$b(u,v) = diff(u) + diff(v) - e(u,v), \quad |e(u,v) = \begin{cases} 2 & : \quad (u,v) \in E \\ 0 & : \quad sonst \end{cases} \tag{3.5}$$

Der Algorithmus sucht nicht markierte Paare von Knoten $(u,v)$, bestimmt $b(u,v)$ und markiert die Knoten. Für das Maximum $b(u,v)$ wird der Austausch durchgeführt und die Auswahl wird wiederholt, bis kein Knotenpaar mehr gefunden werden kann, das die Bisektion verbessert. Diese Heuristik erfordert in der Originalversion $O(n^2)$ Schritte zur Auswahl von geeigneten Knotenpaaren.
Der Algorithmus wurde von Fiduccia und Mattheyses effizient mit linearer Komplexität implementiert [63]. Die Qualität der Partitionierung ist entscheidend von der Anzahl der Versuche abhängig. Die Laufzeit für eine qualitiativ hochwertige Zerlegung ist deshalb nur für ausreichend gute Ausgangszerlegungen akzeptabel.

**3.1.2.2.5 Algorithmus 8: Hilfreiche Mengen** Diekmann et. al. [50] schlagen diesen Algorithmus als Alternative zu Algorithmus 7 vor. Der entscheidende Unterschied besteht in der Auswahl von Mengen von Knoten anstatt von einzelnen Knotenpaaren. Eine hilfreiche Menge ist eine Teilmenge von Knoten, die genau einem Subgraphen zugeordnet sind. Die Kopplungen eines Knotens bezüglich der hilfreichen Menge $S$ berücksichtigt man mit einer Unterscheidung der internen Kopplungen in 3.4 und definiert den Nutzen $H(S)$ einer hilfreichen Menge $S$ mit

$$H(S) = \sum_{v \in S} (ext(v) - int(v) + int_s(v)) \tag{3.6}$$

der damit für die Menge $S$ die gleich Rolle spielt, wie das Knotengewicht $diff(v)$ für den Knoten $v$ in Algorithmus 7.
Ist $k = H(S)$, bezeichnet man die Menge $S$ als $k$-hilfreich. Es sei $S$ eine $k$-hilfreiche Menge im Subgraph $G_i$, dann bezeichnet eine ausgleichende Menge $A$ eine dem anderen Subgraphen $G_j$ zugeordnete Teilmenge, die zumindest $-k+1$-hilfreich ist. Ein Austausch der Mengen $S$ und $A$ zwischen den Subgraphen verringert die Länge des Koppelrandes um 1. Der Algorithmus wird mit der Suche von maximal hilfreichen Mengen und entsprechend balancierten Mengen wiederholt, bis keine Verbesserung mehr möglich ist.
Die Heuristik liefert bei kürzerer Laufzeit oft bessere Ergebnisse als der Kernigham-Lin-Algorithmus. Das bessere Ergebnis ist aber nicht garantiert. Es lassen sich auch Beispiele für schlechtere Zerlegungen, auch bei längerer Laufzeit als mit dem Kernigham-Lin-Algorithmus, angeben.
Einen ähnlichen Algorithmus beschreiben Savage und Wloka [183], die zusätzlich den Austausch zwischen Subgraphen, die nicht benachbart sind, zulassen. Damit wird zwar die Wahrscheinlichkeit für nichtzusammenhängende Gebiete verringert, aber der Algorithmus wird entscheidend langsamer [204], [48].
Die Implementierung des Algorithmus in die Partitionierungssoftware PUL-SM des EPCC beschreiben [205] und [206]. Walshaw et.al. beschreiben eine alternative Implementation [214].



### 3.1.2.3 Algebraisch basierte Heuristiken

**3.1.2.3.1 Algorithmus 9: : Rekursive Spektral-Bisektion bzw. rekursive Eigenwert-Bisektion** Bei diesem Verfahren[2] werden Eigenschaften einer aus der Inzidenzmatrix eines Finite−Element-Graphen abgeleiteten Laplace-Matrix $L(G) = (l_{i,j})$

$$l_{i,j} = \begin{cases} -1 & : \quad (v_i, v_j) \in E \\ deg(v_i) & : \quad i = j \\ 0 & : \quad sonst \end{cases} \tag{3.7}$$

ausgenutzt. Die verwendete Laplace-Matrix ist eine symmetrische Bandmatrix von der Ordnung der Knotenzahl des Systems, deren Hauptdiagonale mit den Knotengraden der zugehörigen Knoten und deren Nebendiagonalelemente für gekoppelte Knoten mit -1 belegt ist. Die Matrix selbst muß bei geeigneter Implementierung nicht aufgebaut werden. Verwendet man z.B. iterative Verfahren vom Lanczos-Typ, die nur ein Matrix-Vektor-Produkt benötigen, kann dieses Matrix-Vektor-Produkt aus dem Graphzusammenhang direkt abgeleitet werden.
Die Bestimmung des zweiten Eigenvektors dieser Laplace-Matrix des Knotengraphes und die Bisektion entsprechend den Eigenvektorkomponenten (alle Knoten mit positiven Eigenvektorkomponenten werden dem einen Teilgebiet zugeordnet, alle weiteren Knoten bilden das andere Teilgebiet und Eigenvektorkomponenten Null bezeichnen den Knoten-Separator) liefern einen zusammenhängenden Separator und entsprechend zwei Subgraphen, deren Zusammenhang gleich oder kleiner dem Ausgangsgraphen ist.
Die mechanische Interpretation dieses Ansatzes liefert dafür eine anschauliche Motivation. Berechnet man für ein kinematisch bestimmtes mechanisches System die zweite Eigenform, wird die Eigenvektormagnitude das System in eine positiv und eine negativ durchschwingende Hälfte an einem möglichst kurzen Durchmesser teilen.
Fiedler zeigt in [64], [65] aufbauend auf den Arbeiten von Cvetkovic (vgl. [44]) diese Eigenschaften. Der zweite Eigenvektor der Laplace-Matrix wird auch Fiedler-Vektor genannt (erstmals von Simon in [191] eingeführt). Mohar vermutet im Spektrum der Laplace-Matrix eine natürliche Norm für Grapheigenschaften [151].
Simon [191] beweist an Hand von praktischen Beispielen, daß der Algorithmus die effizienteste Methode aus der Klasse der rekursiven Bisektionsverfahren ist, eine gute Zerlegung in kurzer Zeit zu erhalten.
In Abbildung 3.4 ist die Spektralbisektion des Modellgebietes in 2, 4 und 8 Teilgebiete zu sehen. Dabei sind die Unterteilung durch fortgesetzte Anwendung des Verfahrens entstanden. Es ist deutlich zu erkennen, daß die Spektralbisektion einen kurzen Separator des zu partitionierenden Gebietes mit großer Sicherheit findet. Die unregelmäßige Koppelrandausbildung ergibt sich bei der Approximation des kontinuierlichen Eigenvektorverlaufs auf die diskreten Knotenkoordinaten.
Eigenwert-Quadrisektion bzw. Eigenwert-Octasektion sind multidimensionale Erweiterungen des Grundalgorithmus unter Einbeziehung des dritten und vierten Eigenvektors. Die Qualität der Zerlegung ist mit dem Grundalgorithmus vergleichbar, zusätzlich haben die so gewonnenen Partitionierungen aber bessere Abbildungseigenschaften auf 2D- bzw. 3D-Netzwerke von Prozessoren, weil die Interprozessorkommunikation optimiert wird (vgl. [95]).
Die Eigenschaften des Fiedler-Vektors allein rechtfertigen aber noch nicht den Einsatz der Eigenwert-Bisektion als Partitionierungsverfahren. Das Verfahren erfordert die Lösung eines speziellen Eigenwertproblems in der Größenordnung des eigentlich

[2] erstmals 1990 vorgeschlagen von Pothen, Simon und Liou [168], basierend auf Arbeiten an der Graph-Sortierung von Barnes und Hoffman, 1982 [16], Asprall und Gilbert, 1984 [8] und Powers, 1988 [170], der die Anwendung zur Partitionierung für die parallele Matrixfaktorisierung anregte

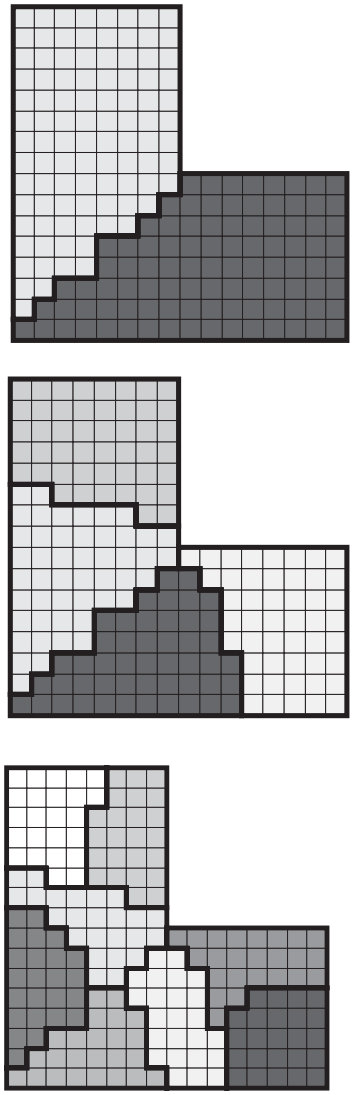

Abbildung 3.4: *L-Gebiet, Partitionierung in 2,4 und 8 Teilgebiete mit Spektralbisektion*



zu lösende lineare Gleichungssystem. Der Algorihmus muß deshalb für einen effizienten Einsatz parallelisiert und wesentlich optimiert werden.
Die Laufzeit des Algorithmus kann drastisch reduziert werden, wenn der Fiedler-Vektor mit einem Multilevel-Algorithmus (vgl. [15], [96], [116]) bestimmt wird. Der Multilevel-Spektralbisektion-(MSB) Algorithmus beschleunigt das Verfahren, ohne die Güte der Partitionierung zu verschlechtern.
Der MSB-Algorithmus besteht aus den folgenden Schritten:

**Graphkontraktion** Mit einem Greedy-Algorithmus (vgl. Algorithmus 4) wird ein Graph erzeugt, dessen Knoten gleichmäßig über dem initialen Graphen verteilt sind. Der Graph soll die Eigenschaften des unstrukturierten Netzes bestmöglich repräsentieren und die Größe des Problems so reduzieren, daß die Bestimmung des Fiedler-Vektors effizient erfolgen kann (z.B. mit Lanczos-Algorithmus).

**Interpolation** Der auf dem groben Netz bestimmte Fiedler-Vektor wird auf das feinere Netz extrapoliert und geglättet.

**Verfeinerung** Der Fiedler-Vektor auf dem feinen Gitter wird iterativ verbessert (mit Rayleigh-Iteration).

Die rekursive Eigenwert-Bisektion in der Multilevel-Variante ist das derzeit beste Verfahren zur Partitionierung von beliebig strukturierten Netzen [218], [136]. Der Algorithmus ist parallelisierbar [110].
Der Charakter als Bisektionsverfahren beschränkt die Auswahl der Partitionsanzahl auf Potenzen von Zwei. Das ist vor allem für Parallelrechner mit Hypercubearchitektur akzeptabel. Diese Einschränkung wird von Hsieh [104] mit zwei verallgemeinerten Varianten des Verfahrens aufgehoben:

**RSS** Spektralbisektion mit sequentielle Abspaltung (engl. *recursive spectral serial-cut*) - dabei wird mit dem algorithmischen Apparat des rekursiven Verfahrens (rekursive Bipartition entlang des Fiedler-Vektors) in sequentieller Abspaltung jeweils genau ein Teilgebiet direkt erzeugt. Die Knoten des in $p_0 = p$ Teilgebiete mit jeweils $n$ Knoten zu partitionierenden Graphen werden entsprechend der Größe der Fiedler-Vektorkomponente sortiert. Es werden dann zwei Teilgraphen mit $1/np$ bzw. $n(p-1)/np$ Knoten gebildet. Das Verfahren wird mit der größeren Teilmenge für $p_i = p_{i-1} - 1$ und $p > 1$ fortgesetzt.

**RST** Spektralbisektion mit angepaßter Zerlegung (engl. *recursive spectral two-way*) - bei dieser Variante wird in jeder Rekursionsebene des Originalverfahrens nicht das Bipartitionsproblem gelöst, sondern der Graph wird in Teilgraphen mit $m_1 = \lceil n/2 \rceil$ bzw. $m_2 = n - m_1$ Knoten zerlegt. Der modifizierte Separator wird analog zum RSS-Verfahren mit den nach der Größe sortierten Komponenten des Fiedler-Vektors bestimmt. Ist $p$ eine Potenz von Zwei, ist das Verfahren identisch zum Originalalgorithmus.

Die angegebenen Verallgemeinerungen von Hsieh et.al. [104] stellen eine Analogie zu den von Jones et.al. [113] vorgestellten Modifikationen der Koordinatenbisektion (siehe Algorithmus 2, Abschnitt3.1.2.1.2) dar.
Eine Verbesserung der Partitionierung kann mit lokal wirkenden Heuristiken (Kernigham-Lin-Algorithmus [122] - Algorithmus 7, Algorithmus der hilfreichen Mengen [50] - Algorithmus 8) erreicht werden.
In Abbildung 3.5 ist die Wirkung des Kernigham-Lin-Algorithmus auf das mit Multilevel-Spektralbisektion partitionierte Probleme (dargestellt in Abbildung 3.4) dokumentiert. Nach jedem Partitionierungsschritt mit der Multilevel-Spektralbisektion wird die Kernigham-Lin-Verbesserung durchgeführt. Die nachfolgende Partitionierung in der nächsten Rekursionsebene erfolgt mit dem bereits verbesserten Teilgebiet. Die Qualität von Teilgebietsform und Koppelrand ist im Vergleich

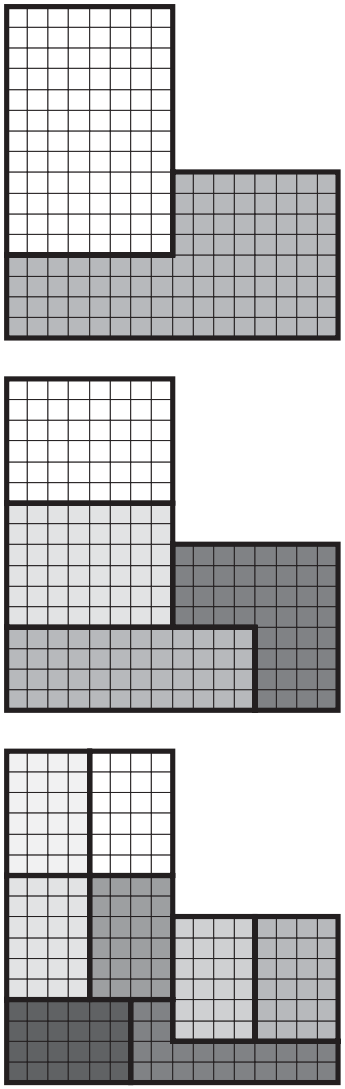

Abbildung 3.5: *L-Gebiet, Partitionierung in 2,4 und 8 Teilgebiete mit Multilevel-Spektralbisektion und lokaler Kernigham-Lin-Verbesserung*



zur alleinigen Anwendung der Multilevel-Spektralbisektion erheblich gestiegen. Die Lastbalance ist erhalten geblieben. Die Laufzeit der Kernigham-Lin-Verbesserung beträgt nur ein Bruchteil der Laufzeit der Multilevel-Spektralbisektion und fällt daher kaum ins Gewicht.

#### 3.1.2.4 Partitionierung durch Optimierung

**3.1.2.4.1 Algorithmus 10: Simulated Annealing** Simulated Annealing („Simuliertes Erstarren“) ist ein heuristisches Optimierungsverfahren und wurde 1983 erstmals von Kirkpatrick et.al. [125] veröffentlicht. Die Optimierungsaufgabe wird dabei nach einem stochastischen Iterationsverfahren gelöst. Streng nach Kosten (Temperatur) optimierende Verfahren werden dabei mehr oder weniger schnell auf Konfigurationen führen, die ein lokales Minimum darstellen. Deren Kosten können durch kleine Veränderungen (Abkühlungen) nur noch vergrößert werden.
Die Metropolis-Methode (bereits 1953 als Variante des Monte-Carlo-Algorithmus von Metropolis u.a. vorgeschlagen, vgl. [147]) akzeptiert deshalb mit einer gewissen Wahrscheinlichkeit auch Konfigurationen, die eine geringe Kostensteigerung nach sich ziehen. Dafür erreicht man bei geschickter Wahl der Schrittweite (Abkühlungsschema) und der Parameter für die Metropolis-Methode das globale Minimum.
Beispiele für die Anwendung der Methode zur Graphpartitionierung sind [66], [108], [159].
Eine Implementation auf einem Parallelrechner und Details der Anwendung des Verfahrens beschreibt Williams in [70]. Zunächst wird eine beliebige Graphpartitionierung erzeugt und auf die Prozessoren verteilt. Entsprechend der Knotenverteilung wird für jeden Prozessor eine Populationstabelle aufgebaut, die angibt, welcher Subgraph welchen Knoten und wieviele Knoten insgesamt enthält. Ziel des Verfahrens ist es nun, die Färbung der Knoten so zu ändern, daß die Knoten gleichmäßig auf die verschiedenen Farben aufgeteilt sind und sich möglichst zusammenhängende Gebiete ergeben.
Auf der Basis der Knotenverteilung wird die Temperatur (die Kostenfunktion, z.B. mit Abschätzung des Kommunikationsaufwandes) bestimmt. In einer bestimmten Anzahl von Permutationen versucht nun jeder Prozessor die Temperatur auf Null zu reduzieren. Am Ende hat jeder Prozessor eine andere Konfiguration in seiner Populationstabelle aufgebaut. Die Unterschiede werden ausgeglichen, indem durch lokale Kommunikation benachbarter Prozessoren eine Anzahl von Knotentauschs akzeptiert bzw. verworfen wird.
Das Verfahren ist auch gut geeignet, eine bestehende Netzpartitionierung, die sich durch Netzadaption aus einer optimalen Verteilung entwickelt hat, zu korrigieren. Eine ähnliche Anwendung ist die Partitionierung von veränderlichen Netzen zur Simulation von zeitabhängigen Strömungsvorgängen [218].
Algorithmus 10 liefert bei ausreichend großer Laufzeit bessere Ergebnisse als Algorithmus 9. Farhat bezweifelt allerdings die Effektivität des Verfahrens, wenn man nur eine bestimmte Laufzeit des Algorthmus erwartet und dann die Qualität der Partitionierung beurteilt [59].

**3.1.2.4.2 Algorithmus 11: Genetische Algorithmen** Wie Algorithmus 10 ist auch dieses Verfahren ein iteratives Vorgehen. Grundlagen für genetische Algorithmen zur Anwendung in Optimierungsproblemen wurden von Holland in [102] beschrieben. Goldberg gibt in [83] einen Überblick über aktuelle Entwicklungen auf diesem Gebiet. Klassischerweise wird ein Parametersatz für das zu lösende Problem in einem Bitstream kodiert. Der Bitstream kann wiederum relativ einfach mit integralen Datentypen implementiert werden. Ein Datensatz beschreibt den Chromosomensatz eines Individuum. Die Gesamtheit der Individuen bildet die Population. Ein große Zahl von Individuen bzw. Parametersätzen wird parallel und

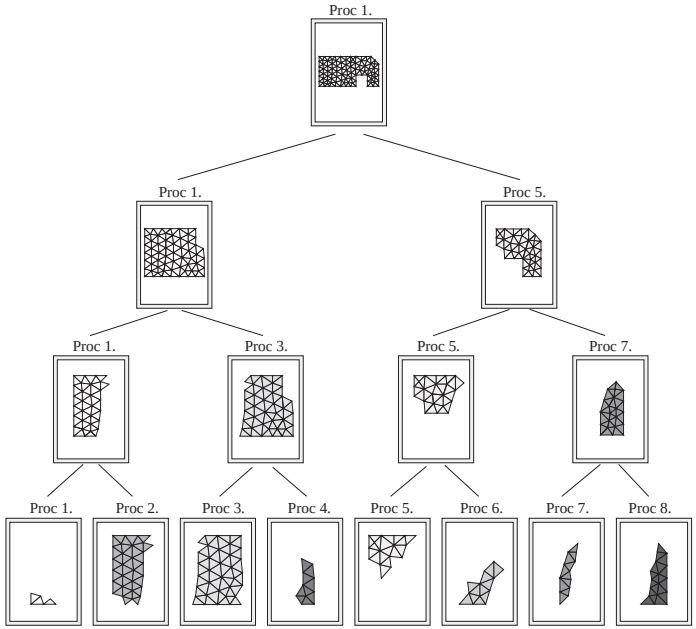


Abbildung 3.6: *Parallele Substrukturgenerierung mit Genetischen Algorithmen*

generationsweise durch Kreuzung bzw. Mutation erzeugt und bewertet. Die Parameterveränderung erfolgt nach genetischen Regeln, was dem Verfahren den Namen gibt. Die Populationsgröße entscheidet maßgeblich über die Kombinationsvielfalt und damit über die Wahrscheinlichkeit, eine optimale Lösung zu finden. Eine große Anzahl an Individuen sichert die Stabilität des Verfahrens auf Kosten eines erhöhten Rechenaufwandes. Das Vorgehen gliedert sich in folgende Schritte:

**Definition der Kostenfunktion** Typischerweise als Maximum- oder Minimumfunktion definiert, basiert die Kostenfunktion auf dem diskreten Parametern der Chromosomensätze der Individuen.

**Kodierung der Parameter** Hier kommt der wesentliche Vorteil genetischer Algorithmen zum Tragen, daß nicht auf den realen Daten selbst, sondern auf einem dazugehörigen Parametersatz gearbeitet wird. Es bietet sich hier z.B. eine indirekte Beschreibung der Netzverteilung über die Netzdichtefunktion an.

**Initialisierung** Ein Satz von initialen Parameterdaten wird zufällig oder zielgerichtet erzeugt.

**Selektion, Mutation und Kombination** bis zum Erreichen der Kostenrestriktionen.

Implementierungen und Vergleiche der Güte der mit genetischen Algorithmen erzeugten Partitionierungen geben z.B. [83], [153]. Das Potential für die Parallelisierung genetischer Algorithmen wurde bereits 1962 von Holland hervorgehoben [101]. Eine erste parallele Implementation exisitiert von Grefenstette 1981 [84]. Eine parallele Implementierung des Algorithmus für die Graphpartitionierung stellen Laszewski und Mühlenbein [130] vor.



Tabelle 3.1: *Abweichung der tatsächlich generierter Elementanzahl je Teilgebiet von der idealen Gleichverteilung ($n_s = 146,5$) (nach [123])*

| Teilgebiet Nr | Elementzahl n | $x - \bar{x}$ n | $x - \bar{x}$ % |
|---|---|---|---|
| 1 | 128 | -18.5 | -12.62 |
| 2 | 150 | 3.5 | 2.39 |
| 3 | 150 | 3.5 | 2.39 |
| 4 | 144 | -2.5 | -1.71 |
| 5 | 152 | 5.5 | 3.75 |
| 6 | 149 | 2.5 | 1.71 |
| 7 | 150 | 3.5 | 2.39 |
| 8 | 147 | 0.5 | 0.34 |

Khan und Topping [123] implementieren auf einem Transputernetzwerk einen Netzgenerator, der zunächst rekursiv ein Hintergrundnetz partitioniert. Initial ist ein Prozessor mit der Partitionierung des Gesamtnetzes beauftragt. Die entstehenden Netzhälften werden nachfolgend rekursiv auf der doppelten Prozessorzahl partitioniert. Das Schema wird für $p$ Prozessoren solange fortgesetzt, bis auf $p/2$ Prozessoren die Bipartitionsaufgabe gelöst wurde. Dieser Ablauf ist für 8 Prozessoren in Abbildung 3.6 dargestellt. Problematisch ist die Effizienz dieser Lösung. Mindestens die Hälfte der Prozessoren ist während der Partitionierung untätig. Dieser Nachteil wird nur dadurch relativiert, daß das Hintergrundnetz nur einen Bruchteil der Element- und Knotenzahl des eigentlichen Berechnungsnetzes besitzt und damit eine relativ kleine Partitionierungsaufgabe gelöst werden muß.

Die Partitionierung wird für einen gewichteten Graphen durchgeführt. Die Knotengewichte werden entsprechend der zu erwartenden Elementierung im Berechnungsnetz definiert. Die Abschätzung der Anzahl der zu generierenden Elemente in jedem Element des Hintergrundnetzes wird mit Hilfe eines neuronalen Netzwerkes realisiert. Dieses neuronale Netzwerk kann bei geeignetem Training eine gute Näherung an die Lastbalance erreichen (vgl. 3.1). Im Beispiel beträgt die entscheidende maximal positive Abweichung vom Idealwert 5.5%. Problematisch sind stark graduierte Netze. Dann werden die verteilbaren Lasteinheiten des Hintergrundnetzes sehr uneinheitlich und die Lastbalance wird nur annähernd erreicht.

Sind die den Prozessoren zugeordneten Partition des Hintergrundnetzes ermittelt, werden sie nachfolgend vollständig parallel diskretisiert. Die Netzkonformität über die Koppelränder wird algorithmisch, ohne zusätzliche Kommunikation gesichert. Für die Koppelränder wird eine der Reihenfolge der Problembeschreibung entsprechende globale Numerierung gewählt. Damit ist ein globaler Namensraum für die Koppeldaten gesichert.

In Abbildung 3.7 wird das Problem aus Abbildung 3.6 noch einmal im Zusammenhang dargestellt. Die Elementanzahl in den Teilgebieten ist unterschiedlich groß, entsprechend der unterschiedlich großen Anzahl der zu generierenden Elemente des Berechnungsnetzes. In Abbildung 3.8 ist schließlich das auf den acht Prozessoren parallel generierte Berechnungsnetz dargestellt.

Im Vergleich zur RSB-Methode (Originalverfahren ohne Multilevelbeschleunigung) wird bei einem Viertel der Laufzeit ein adäquates Ergebnis erreicht. Qualität und

Laufzeitverhalten der Methode sind mit Algorithmus 10 vergleichbar.

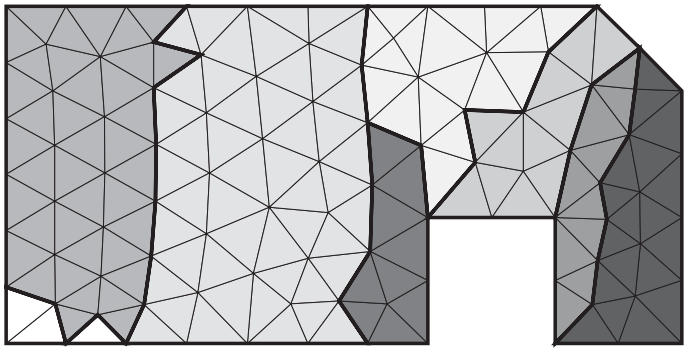

Abbildung 3.7: *Partioniertes Hintergrundnetz für das Problem aus Abbildung 3.6*

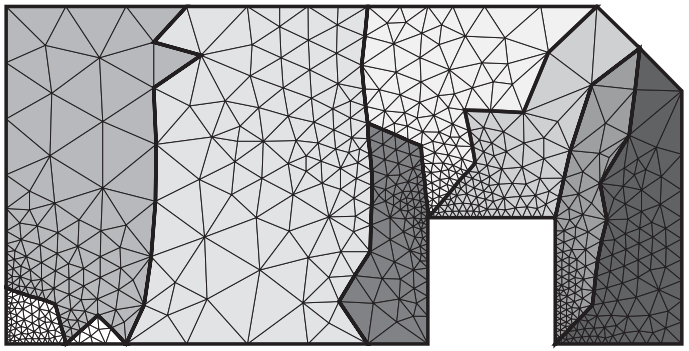

Abbildung 3.8: *Partioniertes Berechnungsnetz für das Problem aus Abbildung 3.6*

**3.1.2.4.3 Algorithmus 12: Neuronale Netze** Eine Implementierung eines neuronalen Netzwerkes zur Partitionierung wird zum Beispiel von Khan und Topping [123] vorgestellt. Im Vergleich zur rekursiven Spektralbisektion (Algorithmus nach [190]) kann ein neuronales Netz in der halben Zeit Partitionierungen gleicher Güte erzeugen. Dieser hoffnungsvolle Ansatz hält aber dem Laufzeitvergleich mit der effizienteren Mutlilevelimplementierung [15] nicht stand.

Das neuronale Netz kann in der Regel nur bis zu einer gelernten Größe des Netzes gute Partitionierungen liefern. Bei stark schwankenden Anforderungen ist deshalb das Training des neuronalen Netzwerkes ein nicht zu unterschätzender Zeitfaktor.



#### 3.1.2.5 Zusammenfassung

Eine Auswahl der bisher beschriebenen Algorithmen sind in der Chaco-Bibliothek [97] (z.B. Kernighan-Lin-Algorithmus, Multilevel-Spektralbisektion, verschiedene Verfahren mit experimentiellem Charakter) bzw. in der METIS-Bibliothek [117] implementiert. Bei der Nutzung dieser Algorithmensammlung zeigt sich, daß eine Kombination der lokal und global wirkenden Zerlegungsalgorithmen in Abhängigkeit von der betrachteten Problemklasse die besten Ergebnisse bringt. Kein Verfahren erzielt immer ideale Ergebnisse. Ein universelles Tool zur Netzpartitionierung muß deshalb

- eine Anzahl globaler Partitionierungsalgorithmen,
- eine Auswahl lokaler Verbesserungen und
- eine maschinenbezogene Bewertung einer gefundenen Zerlegung

anbieten. Solche Tools beschreibt z.B. Farhat in [61].
Leland und Hendrickson [136] geben die folgenden Empfehlungen: Sind die Koordinateninformationen verfügbar und die Laufzeit des Algorithmus ist unkritisch, dann ist die Kombination von Schwerachsen-Bisektion und Kernighan-Lin-Algorithmus zu bevorzugen. Ohne die heuristische Verbesserung kann die Laufzeit drastisch reduziert werden. Sind die Koordinateninformationen nicht verfügbar oder ist die Qualität der Partitionierung entscheidend, so sollte die Multilevel-Spektralbisektion eingesetzt werden. Für finite Elementapplikationen mit verfügbaren Koordinateninformationen stellt daher die Kombination aus Schwerachsen-Bisektion und Kernighan-Lin-Algorithmus die Methode der Wahl dar, zumal sie, bei wesentlich geringeren Berechnungsaufwand, Partitionierungen vergleichbarer Güte liefert [136], siehe auch Tabelle 6.1. [41] favorisieren ebenfalls die geometrisch basierten Methoden.
Lokal wirkende Heuristiken haben im Kontext der adaptiven Finite-Element-Methoden eine besondere Bedeutung. Ihre Parallelisierung erlaubt die effiziente dynamische Lastverteilung während der Berechnung ohne die Informationen über das Finite-Element-Netz einsammeln und mit einem seriellen Partitionierungsverfahren neu verteilen zu müssen. Die resultierenden starken Umverteilungen werden bei lokal wirkenden Verbesserungen einer bestehenden Partitionierung vermieden. Diese dynamischen Lastverteilungsverfahren werden im folgenden beschrieben.

### 3.1.3 Dynamische Gebietszerlegungsverfahren

#### 3.1.3.1 Typische Problemstellung und Randbedingungen

Folgende Problemstellungen, bei denen sich der während der numerischen Simulation mit der Finite-Element-Methode der absolute Berechnungsaufwand oder die Verteilung des Berechnungsaufwandes im Simulationsgebiet lokal ändert, führen auf Lastverteilungsprobleme, die mit dynamischen Gebietszerlegungsverfahren (vergleiche auch [94]) zu lösen sind:

- adaptive Netzverfeinerung,
- physiklaisch nichtlineare Probleme,
- zeitabhängige Probleme und
- Kontaktprobleme.

Im folgenden werden einige dynamische Lastverteilungsverfahren gegenübergestellt. Die Maßeinheit für die Belastung eines Prozesses ist dabei mit finiten Elementen

bzw. finiten Knoten festgelegt. Diese Größen können z.B. für nichtlineare Berechnungen noch gewichtet werden.
Während die statischen Gebietszerlegungsverfahren nur im Ausnahmefall auf bereits verteilt vorliegenden Datenmengen arbeiten, ist für die dynamischen Zerlegungsverfahren die bereits verteilte Datenorganisation der Regelfall. Außerdem muß die Lastverteilung sehr effizient und schnell arbeiten, da sie auf dem Parallelrechner ausgeführt wird und damit kostenintensiv und unter Speicherplatzrestriktionen realisiert werden muß.
Die dynamische Lastverteilung muß unter den folgenden Kriterien beurteilt werden

- Bedeutung eines vorhandenen (geringen) Lastungleichgewichts,
- Kommunikationskosten,
- Häufigkeit der Lastumverteilung,
- Kosten des Lastumverteilungsalgorithmus,
- Kosten der Datenbewegung (Abbau und Aufbau der notwendigen Datenstrukturen, Konsistenzsicherung) und
- Komplexität der Implementierung.

Dabei kann eine allgemeingültige Aussage über die Notwendigkeit der dynamischen Lastumverteilung nur unter konkreten Bedingungen einer Applikation getroffen werden.
Die dynamische Partitionierung ist stark an die Applikation gekoppelt. Eine sinnvolle Anzahl der Umverteilungen wird entscheidend von den Auswirkungen eines etwaigen Ungleichgewichts bestimmt. Vorhandene Datenstrukturen müssen ausgenutzt werden und die tatsächliche Datenverteilung hat einen Einfluß auf die Effektivität des Verfahrens. Zur Minimierung der Datenbewegung und der Abhängigkeiten muß in die Gebietszerlegung die Information über die Lokalisation der Daten einfließen. Dafür gibt es im statischen Verfahren keine Entsprechung. Allerdings sind diese Kriterien entscheidend für die Effizienz. Es bieten sich im Prinzip lediglich inkrementielle Herangehensweisen an.

### 3.1.3.2 Klassifikation der dynamischen Lastverteilungsverfahren

Es gibt in der Literatur keine durchgängig benutzte Klassifikation der dynamischen Lastverteilungsverfahren. Die Begriffe adaptive und dynamische Lastverteilung werden oft synonym benutzt und kennzeichnen den Laufzeitcharakter des Verteilungsalgorithmus, der in eine laufende Applikation eingebettet ist. Die Parameter der Partitionierungsaufgabe - die tatsächliche Verteilung der Rechenlast entsteht während der Applikationslaufs.
Die Applikation selbst kann während des Berechnungsablaufes den Zeitpunkt, zu dem die Lastbalancierung erfolgt, festlegen. Typischerweise wird dazu ein Schwellwert für das zu tolerierende Lastungleichgewicht definiert. Die Größenordnung dieses Schwellwertes bestimmt sich aus den zu erwartenden Kosten für eine Lastumverteilung. Diese Kosten werden durch die Laufzeit des Ausgleichalgorithmus und die erforderliche Datenbewegung und Datenorganisation (Datenerzeugung und -löschung) bestimmt. Damit sind diese Kosten selbst problemabhängig.
Kennzeichnend für die verschiedenen Lastverteilungsverfahren ist es, ob eine zentrale Instanz zur Verwaltung der Information über die tatsächliche Lastverteilung und zur eigentlichen Lastumverteilung benutzt wird oder nicht. Zentrale Lastverteilung scheidet für Anwendungen auf dem Parallelrechner in der Regel aus, weil die Applikation die verteilte Speicherkapazität des Rechnersystems ausnutzt und die erforderliche Problembeschreibung zu umfangreich für einen Prozessor ist. Nur



wenn die Lasteinheiten sehr groß gewählt werden, kann das Partitionierungsproblem zentral gelöst werden. Die zentrale Instanz wird dann in der Regel die Zuordnung der Lasteinheiten zu Prozessoren steuern können.
Effizienter ist die sogenannte dezentrale Lastverteilung. Dabei wird das Partitionierungsaufgabe von allen Prozessoren gemeinsam gelöst, ohne die Problembeschreibung in einer zentralen Instanz zusammenzufassen. Die Information über die Zuordnung der Lasteinheiten zu den Prozessoren und die genaue Lage der Lasteinheiten im Bezug auf die benachbarten Prozessoren kann im Prozeß der Lastverteilung ohne zusätzlichen Datenaustausch ausgewertet werden.
Eine weitere Differenzierung ergibt sich aus der Wirkungsweise des Lastverteilungsverfahrens. Implizite Verfahren propagieren die lokale Überlast an weniger belastete Prozessoren. Die balancierte Lastverteilung stellt sich erst nach einigen Iterationsschritten ein. Demgegenüber bestimmen explizite Verfahren die endgültige Lastzuordnung unmittelbar. Im folgenden sollen beide Strategien und eine Mischform gegenübergestellt werden.

**3.1.3.2.1 Implizite Verfahren** Dieses Verfahren wirkt lediglich lokal. Mit einem lokalen Datenaustausch zwischen unmittelbar benachbarten Gebieten wird entsprechend einem Diffusionsvorgang das Gleichgewicht der Lastverteilung hergestellt. Dabei wird in der Regel die Geometrieinformation und nicht die topologische Information verwendet. Beispiele dafür sind die rekursive Koordinatenbisektion oder ihre Varianten nach [113] (vgl. Abschnitt 3.1.2.1.2). Das Verfahren erfordert Kommunikation, um die durchschnittliche Lastverteilung zu ermitteln. Die Konvergenz des Verfahrens ist asymptotisch gesichert. Cybenko [45] gibt für den Lastausgleich im Hypercube bis zur Konvergenz $d+1$ Iterationsschritte an. Boillat [27] beweist, daß die Diffusionsmethode für einen zusammenhängenden Graphen in polynomialer Zeit konvergiert. Grundlage der Konvergenzuntersuchung sind dabei statische Kommunikationstopologien. Ändert sich auch der Zusammenhang der Teilgebiete während des LastausgleichsAllerdings kann dabei der Umfang der transportierten Nachrichten wesentlich höher sein, als bei expliziten Verfahren. Bei moderatem Lastungleichgewicht sind die Auswirkungen der Lastumverteilung gering und die Zerlegung ändert sich nicht sehr stark.

**3.1.3.2.2 Explizite Verfahren** Bei einem expliziten Verfahren werden die Lasteinheiten entsprechend ihrem topologischen Zusammenhang für eine Umverteilung ausgewählt. Es wird daher der problembeschreibende Graph und nicht die Geometrie benutzt. Zwei Ansätze, die das Problem vereinfachen und globale Kommunikation vermeiden, erlauben eine effiziente Implementierung:

- Datenaustausch lediglich zwischen bereits im Ausgangsgraphen benachbarten Prozessen. Für den eigentlichen Datenaustausch wird damit, ideale Abbildung des Prozeßgraphen auf das Verbindungsnetzwerk vorausgesetzt, eine Überlastung der Kommunikationskanäle vermieden. Allerdings ist nicht gesichert, daß die Teilgebiete zusammenhängend bleiben.
- Datenaustausch nur zwischen Prozessoren, die keine gemeinsamen Kanten haben. Nach [94] werden damit zwar bessere Zerlegungen gefunden, aber die Implementation ist aufwendiger und die Kommunikation in der Balancierungsphase kann keinen Nutzen aus einer vorteilhaften Kommunikationsstruktur für das Ausgangsproblem ziehen.

Diese Verfahren, die einen Austausch zwischen gewählten oder tatsächlich im Problem vorhandenen Teilproblemen nutzen, werden als Austauschverfahren bezeichnet.

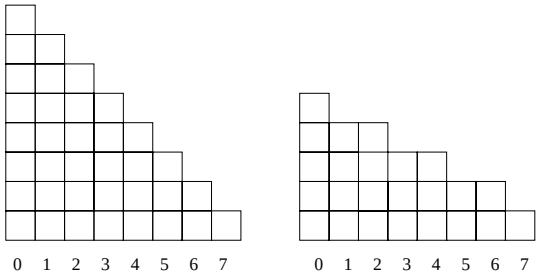


Abbildung 3.9: *Lastungleichgewicht in einer Pipeline, das (a) bei direktem Nachbarausgleich bzw. (b) bei baumbasierten Ausgleich nicht beseitigt werden kann*

**3.1.3.2.3 Hybride Verfahren** Ein hybrides Herangehen, das abwechselnd, beide Verfahren nutzt scheint in Abhängigkeit von der konkreten Applikation das Optimum darzustellen. Dabei sollte seltener der implizite Algorithmus benutzt werden, um die globalen Datenstrukturen, insbesondere die Kommunikationsstrukturen, weiterverwenden zu können. Der explizite Algorithmus wird öfter benutzt, um eine bessere Lastverteilung zu erreichen.

### 3.1.3.3 Spezielle dynamischen Lastverteilungsverfahren

Von den existierenden statische Gebietszerlegungsverfahren sind die Algorithmen mit lokaler Wirkung als Kandidaten für die implizite, dynamische Gebietszerlegung geeignet (z.B. Greedy-Algorithmus nach [58], lokaler, paralleler Algorithmus von Walshaw [214]). Der Vorteil dieser Algorithmen liegt in der Ausnutzung lokaler Informationen, so daß sie auch ohne globale Kommunikation implementiert werden können.
Durch die beschränkte lokale Wirkung der Algorithmen kann aber auch eine Behinderung eintreten, weil nicht genügend Daten bewegt werden können bzw. die Lastdifferenz zwar zwischen benachbarten Gebieten ausgeglichen ist, aber die Differenz zwischen Maximal- und Minimallast im System extrem wird [93].
In Abbildung 3.9 ist für zwei Konfigurationen diese Situation dargestellt. Lokal ist der Lastunterschied minimiert, global herrscht Ungleichgewicht. Fall a) zeigt diese stabile Konfiguration für Migrationsverfahren mit ausschließlichem Datenaustausch zwischen Nachbarn. In Fall b) ist diesselbe Situation für einen zugrundeliegenden Binärbaum dargestellt. Im Binärbaum würde Fall a) in einem Schritt zum idealen Gleichgewicht konvergieren. Im allgemeinen kann der Lastausgleich in dieser Situation lediglich durch zusätzliche globale Kommunikation und damit Synchronisation erreicht werden.
Alternativ entwickelt Horton eine Multistage-Diffusionsmethode [103], die den Nachteil des lokalen Datenaustausches und den Nachteil des globalen Lastungleichgewichts vermeidet. Allerdings erfordert seine Methode globale Kommunikation. Das Verfahren basiert auf einer Idee von Cybenko [45]. Cybenko stellt ein dynamisches Lastverteilungsverfahren für Hypercubenetzwerke vor. Dabei wird nacheinander in jeder Dimension des Hypercubes ein lokaler Lastausgleich vorgenommen. Der Algorithmus ignoriert die Topologie des Problems und nutzt lediglich die Kommunikationsstruktur aus. Damit ist das Verfahren nur für parallele Algorithmen anwendbar, die unabhängig von der tatsächlichen Datenverteilung sind. Horton nutzt in [103] nun statt der Hardwaretopologie das logische Verbindungsnetzwerk des Problems, das aus der statischen Lastverteilung gewonnen wurde.
Werden die Kosten der Datenbewegung in die Beurteilung einer gefundenen Neuverteilung einbezogen, gewinnt man ein realistischeres Bild von einer Umverteilung. Dabei kann die Umverteilung als Wichtung des Graphen eingebaut werden. Dieses Herangehen ist sowohl für graphbasierte (Greedy-Algorithmus, Kernighan-Lin-Algorithmus) als auch für Spektralbisektionsverfahren [53] geeignet.
Das Spektralbisektionsverfahren wird erst mit dieser Modifikation für eine dynamische Lastumverteilung brauchbar. Die Ursache dafür ist nicht eindeutig geklärt, aber Erfahrungen mit der Umverteilung von h-adaptiv verfeinerten Netzen zeigen, daß „eine kleine Änderung in der Netzverfeinerung zu einer großen Änderung des zweiten Eigenvektors führen kann“ [218], der die Partitionsgrenzen bestimmt (Anm. des Autors).
Zwei Strategien zur Anpassung des Spektralbisektionsverfahrens an die Anforderungen der dynamische Lastumverteilung sollen im folgenden vorgestellt werden:

- Graphkontraktion [213] und
- gewichtete Spektralbisektion [53].

**Graphkontraktion** Die Idee besteht darin, das partitionierte aber unbalancierte Elementnetz auf weniger Elemente zu reduzieren und mit einem kontrahierten Elementgraph das Partitionierungsproblem zu lösen. Zur Kontraktion werden nur bestimmte Elemente ausgewählt. Zugrundeliegt die anschauliche Überlegung, daß eine kleine Änderung der Elementkonzentration im Gebiet auch nur zu einer kleinen Änderung der Gebietsgrenzen führen wird. Elemente, die weit genug von der bestehenden Teilgebietsgrenze liegen, werden deshalb einfach zu einem einzigen Element zusammengefaßt.
Das Verfahren erzeugt zunächst eine Stufenstruktur des Elementgraphen in jedem Teilgebiet. Die erste Stufe wird entlang der Gebietsgrenze bestimmt. Die weiteren Stufen schließen sich nach Innen an, bis im Zentrum die letzte Stufe gebildet wird. Danach erfolgt von innen nach außen die Zusammenfassung der Stufen zu einem Makroelement. Die am weitesten außenliegenden Elemente sind Kandidaten für den Austausch.
Sind die Teilgebiete kompakt, verbleibt ein Band von Originalelementen entlang der bestehenden Teilgebietsgrenzen. Das Partitionierungsproblem wird mit diesen wenigen Elementen gelöst. Die Makroelemente erhalten dabei natürlich eine Wichtung entsprechend der Anzahl der Elemente, die durch sie repräsentiert werden.
Das Verfahren der Graphkontraktion ist sowohl für die Anwendung mit anderen Partitionierungsalgorithmen, z.B. Simulated Annealing [218] (vgl. 3.1.2.4.1), als auch zur Beschleunigung der seriellen Implementation des Originalalgorithmus geeignet.

**Gewichtete Spektralbisektion** Einen anderen Weg beschreiten v. Driesche und Roose [53]. Sie bilden die Restriktion durch zwei zusätzliche virtuelle Knoten und Kanten zwischen zwei Knoten $v_1$ und $v_2$, für die $v_1 \in V_1$ und $v_2 \in V_2$ gilt. Die virtuellen Knoten werden a priori den Teilgebieten $V_1$ bzw. $V_2$ zugewiesen. Diese Knoten werden gewichtet, um die Bedeutung der Zuordnung der virtuellen Knoten zu den Teilgebieten zu kennzeichnen.
Das im Originalalgorithmus zu lösende Eigenwertproblem (vgl. 3.1.2.3.1) geht dann in die spezielle Formulierung $Ax = \lambda x + g$ über. Dabei sind die Komponenten des Vektors $g$ durch die Gewichte der virtuellen Knoten bestimmt. Ein spezieller Lanczos-Löser für dieses Eigenwertproblem konvergiert in Abhängigkeit von der

Wichtung schneller als der Standard-Lanczos, in [54] wird für realistische Wichtungen eine Beschleunigung um den Faktor 3 bis 5 angegeben.

#### 3.1.3.4 Zusammenfassung

[94] stellt folgenden Kriterienkatalog zur Entscheidungsfindung, ob eine dynamische Lastverteilung erforderlich ist, auf:

1. Ist das Problem auch ohne einen Parallelrechner lösbar?
2. Kann man einen einfacher zu parallelisierenden Algorithmus für die Lösung des Problems benutzen?
3. Kann man einen alternativen Weg der Parallelisierung benutzen, der keine dynamische Lastverteilung erfordert?
4. Ist das Lastungleichgewicht, das aus der statischen Gebietszerlegung resultiert, akzeptabel?

Nur wenn alle Fragen eindeutig negativ zu beantowrten sind, sollte der große Aufwand einer Problemlösung auf einem Parallelrechner und einer dynamischen Lastumverteilung betrieben werden.

## 3.2 Abbildungsverfahren

Eine hohe Systemleistung auf einem Parallelrechner wird nebend der Einzelprozessorleistung vor allem vom Datendurchsatz im Kommunikationsnetzwerk des Parallelrechners bestimmt. Die Architektur und Topologie dieses Netzwerkes bestimmt wesentlich die zur Verfügung stehende Bandbreite für die Übertragung einer Nachricht zwischen zwei beliebigen Prozessoren.. Bei limitierter Übertragungsrate kann die optimierte Zuordnung der parallelen Teilaufgaben mit ihren spezifischen Kommunikationsanforerungen zu den Prozessoren entscheidend die Laufzeit eines Programms beeinflussen. Stimmt die Topologie des Applikationsprogramms (die sogenannte *virtuelle* Topologie) mit der Topologie des Verbindungsnetzwerkes (der sogenannten *physikalischen* Topologie) gut überein, kann ein Optimum bei der Ausnutzung der Hardwareleistung erreicht werden. Die Abbildung der Prozesse und Teilprobleme auf die Prozessoren ist mit den Hilfsmitteln der Graphentheorie als Grapheinbettungsproblem zu behandeln:
Es seien $G =< V(G), E(G) >$ und $H =< V(H), E(H) >$ zwei ungerichtete Graphen, dann ist die Einbettung $f$ von $G$ auf $H$ die injektive Abbildung $f : V(G) \mapsto V(H)$. $G$ ist der Gast- und $H$ ist der Wirtgraph der Einbettung. Die Dilation einer Kante $e \in E(G)$ ist der topologische Abstand (Distanz) zweier zu $e$ inzidenter Knoten in $H$. Die Dilation der Einbettung $f$ ist die maximale Distanz zweier in $G$ benachbarter Knoten in $H$. Die Aufgabe besteht in der Bestimmung einer minimalen Dilation der Einbettung.
Der Gastgraph ist als die virtuelle und der Wirtgraph als die physikalische Topologie anzusehen. Ist die Kommunikationsstruktur einer Applikation vor der Laufzeit des entsprechenden Programms bekannt, kann das Abbildungsproblem für einen bestimmten Parallelrechner bereits vor Programmstart gelöst werden.

### 3.2.1 Spezielle Einbettungsprobleme für 2D-Gitter-Wirt

Gitter sind in einer Reihe von aktuellen Multiprozessorsystemen im Kommunikationsnetzwerk physikalisch realisiert: T3D (3D-Torus), Intel Paragon und Parsytec GC (2D-Gitter). Im folgenden sollen deshalb exemplarisch Lösungen für spezielle Einbettungsprobleme mit 2D-Gittern als Wirtgraphen dargestellt werden.



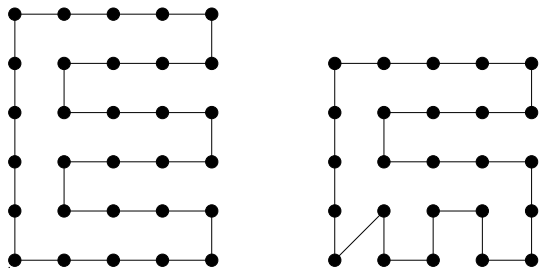

Abbildung 3.10: Einbettung von Ring in 2D-Gitter

Ohne Beschränkung der Allgemeinheit sollen im folgenden die optimalen Einbettungen von Graphen mit $n$ Knoten in 2D-Gitter der Dimension $n = x \times y$ betrachtet werden. Eine Einbettung heißt optimal, wenn kein $\hat{x} \times \hat{y}$ Subgitter mit $n \leq \hat{x} * \hat{y}$ existiert.
Die hier dargestellten Einbettungsverfahren sind in der PARIX- (vgl. Abschnitt 2.3.3.4) Bibliothek für virtuelle Bibliotheken implementiert und in [174] ausführlich dargestellt.

#### 3.2.1.1 Einbettung von Ring und Pipeline

Die Einbettung einer Pipeline ist elementar und liefert immer Dilation Eins: Die Knoten der Pipeline werden zeilenweise auf das 2D-Gitter abgebildet. Dabei startet man in der ersten Spalte und Zeile. In der letzten Spalte geht man auf die nächste Zeile über und wechselt die Richtung.
Die Einbettung von Ringen in 2D-Gitter ergibt für ungeradzahlige Knotenzahl $n$ Dilation Zwei für eine Kante und Eins für alle anderen Kanten und für geradzahlige Knotenzahl $n$ immer Dilation Eins.

**$x$ oder $y$ sind geradzahlig:** Angenommen, $y$ ist geradzahlig, dann werden die Knoten zeilenweise auf das 2D-Gitter abgebildet. Dabei startet man in der zweiten Spalte der ersten Zeile. In der letzten Spalte geht man auf die nächste Zeile über und wechselt die Richtung. Der nächste Zeilensprung mit Richtungswechsel erfolgt wieder in der zweiten Spalte. In der letzten Zeile wird bis in die erste Spalte aufgefüllt und danach werden die letzten $y - 2$ Knoten in die erste Spalte abgebildet. Der Knoten $n - 1$ ist dann in der zweiten Zeile unmittelbarer Nachbar von Knoten 0.

Ist nur $x$ geradzahlig, dann ist das Verfahren sinngemäß unter Vertauschung von Spalten bzw. Zeilen anzuwenden.

**$x$ und $y$ sind ungeradzahlig:** In diesem Fall wird die oben beschriebene Technik benutzt, um alle Knoten bis auf die letzten zwei Zeilen abzubilden. In den letzten beiden Zeilen wird eine spezielle Mäandertechnik (vgl. 3.10 nach [174]) benutzt, die den Knoten 0 über zwei Kanten mit dem Knoten $n-1$ verbindet.

#### 3.2.1.2 Einbettung von Gittern

Es sei das Gastgitter mit den Dimensionen $a, b$ und das Wirtgitter mit den Dimensionen $x, y$ bezeichnet. Ohne Beschränkung der Allgemeinheit gelte $a \leq b$ und

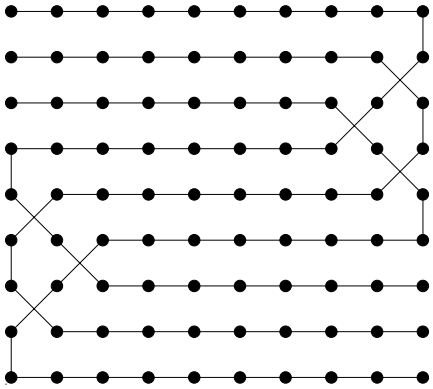

Abbildung 3.11: Einbettung eines $3 \times 30$ Gitters in ein $10 \times 9$ Gitter durch Faltung

$x \leq y$.

**Einbettung durch Faltung** Im Falle $a\lceil b/y \rceil \leq x$ und $y \geq a$ kann die Faltungstechnik angewendet werden. Eine Zeile des Gastgraphen wird in die entsprechende Zeile des Wirtgraphen abgebildet. Dabei werden zunachst direkt die Spalten der Originalzeile $i$ in Zeile $i$ und die Spalten $0, \ldots, i$ abgebildet. Dann erfolgt eine Faltung mit einer Folge von diagonaler (Vorwärts-), vertikaler und wieder diagonaler (Rückwärts-) Verbindung. Anschließend wird die Abbildung in Zeile $i + b$ in entgegengesetzter Richtung bis in Spalte $i + 2b$ fortgesetzt. Die Einbettung wird mit einer Folge von Faltung und Abbildung in die Zeilen wiederholt, bis alle Elemente eingebettet sind. Die Einbettung eines $3 \times 30$ Gitters in ein $10 \times 9$ Gitter durch Faltung ist in Abbildung 3.11 dargestellt.

Die Einbettung durch Faltung hat eine konstante Dilation von Zwei. Die maximalen Dilationen einer Kante treten in den „Faltungen“ auf.

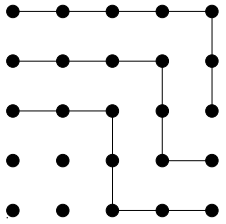

Abbildung 3.12: Einbettung eines $3 \times 7$ Gitters in ein $5 \times 5$ Gitter durch Abstufung

**Einbettung durch Abstufung** Im Falle $x - a \geq b - y$ kann die Abstufungstechnik angewendet werden. Jede Zeile des Gastgraphen wird auf die entsprechende Zeile des Wirtgraphen abgebildet. Dabei werden die Spalten $0, \ldots, y - i$ benutzt. Danach wird bis in Zeile $(i + b - y - 1$ vertikal fortgesetzt, um dann in die ursprüngliche Richtung weiter horizontal fortzusetzen.

Die Einbettung durch Abstufung hat eine konstante Dilation von Drei. Die maximale Dilation einer Kante tritt im „Knickpunkt“ auf.



**Standardeinbettung** Im Trivialfall $a \leq x$ und $b \leq y$ ist $a \times b$ ein Teilgraph von $x \times y$ ist und die Abbildung ist die Identität. Andernfalls gilt $a \leq y$ und $b > x$. [174] implementieren in diesem Fall eine Variante der Kettentechnik nach [182]. Jede Spalte des Gastgraphen wird auf eine Folge von horizontalen (Vorwärts-) und diagonalen (Rückwärts-) bzw. vertikalen Verbindungen abgebildet. Die Länge der horizontalen Pfade wird mit $\lceil jb/x \rceil - \lceil (j-1)b/x \rceil - 1$ für ausgewählte $j \in 1, \ldots, x$ bestimmt. Aufeinanderfolgende $j$ werden für aufeinanderfolgende Spalten benutzt.

Die resultierende Dilation ist nicht beschränkt, d.h. es existieren Einbettungungen von 2D-Gittern in 2D-Gitter, die nicht mit konstanter Dilation realisierbar sind.

#### 3.2.1.3 Einbettung von Tori

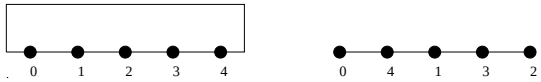


Abbildung 3.13: Einbettung von Ring in Pipeline

Die Einbettung von Tori bedient sich eines zweistufigen Verfahrens. Zunächst wird ein Torus auf ein Gitter gleicher Dimension, d.h. auf die gleiche Knoten- und Kantenkonfiguration aber ohne die schließende Kantenverbindung zwischen Anfangs- und Endknoten in jeder Dimension, abgebildet.
Dabei wird in jeder Dimension die Abbildung eines Rings auf eine Pipeline wie folgt benutzt:

$$v_i(G) \mapsto v_i(H) \begin{cases} i \leq \frac{n-1}{2} & : \quad 2i \\ sonst & : \quad 2(n-1)-1 \end{cases} \tag{3.8}$$

Das Vorgehen ist in Abbildung 3.13 dargestellt und ist mit Dilation Zwei realisierbar. Im zweiten Schritt wird das so erhaltene Gitter auf die Knoten des endgültigen Gitters abgebildet. Dabei werden die Verfahren entsprechend Abschnitt 3.2.1.2 zur Abbildung von 2D-Gittern auf andere 2D-Gitter angewendet.

### 3.2.2 Zusammenfassung

Das Einbettungsproblem ist, genau so wie das Partitionierungsproblem, NP-vollständig. Für eine Anzahl von regulären Gasttopologien lassen sich Einbettungen mit konstanter Dilation angeben. Es gibt aber Kombinationen von Gast- und Wirtgraphen, für die sich keine Einbettungen mit beschränkter Dilation realisieren lassen. Für unregelmäßige Kommunikationsstrukturen (wie sie in partitionierten, unstrukturierten FE-Netzen vorherrschen) existieren deshalb lediglich suboptimale, heuristische Lösungen für das Einbettungsproblem.

Moderne Parallelrechner verfügen in der Regel über in Hardware realisierte *cut-through* Routingverfahren. Damit ist die Übertragungszeit zwischen zwei Prozessoren für eine Nachricht unabhängig von der Anzahl der zu traversierenden Verbindungen. Die Dilation der Einbettung des Kommunikationsnetzwerkes in das physikalische Verbindungsnetzwerk hat nur einen geringen Einfluß auf die Kommunikationszeit, die im wesentlichen proportional zur Nachrichtenlänge ist. Den entscheidenden Einfluß auf die realisierbare Übertragungsrate zwischen zwei Prozessoren hat die Anzahl der logischen Verbindungen, die diesselbe physikalische Verbindung zwischen den beiden Prozessoren benutzen. Die Anzahl logischer Verbindungen pro physikalische Verbindung wird als Kongestion bezeichnet. Für eine Reihe von Applikationen treten die Kommunikationsanforderungen der Prozessoren gleichzeitig auf. Dann stimmt die Bestimmung der Kongestion unter Berücksichtigung der zu erwartenden Nachrichtenlänge gut mit der tatsächlichen Belastung des Verbindungsnetzwerkes überein. Die Kongestion ist wiederum eine Funktion der Dilation. Das wird sofort anschaulich, wenn man die oben dargestellten Abbildungsverfahren und die angegebenen Beispiele untersucht.

# Kapitel 4

# Parallele Strömungsmechanik

## 4.1 Motivation und Abgrenzung

Anhand eines konkreten Problems der Strömungsmechanik, speziell für inkompressible und turbulente Strömungen sollen Grundlagen der Parallelisierung erläutert werden.

Auf der Basis der Grundgleichungen der Strömungsmechanik, der Navier-Stokes-Gleichungen, können zum Beispiel Flutwellen nach einem brechenden Damm, das Schwappen einer Flüssigkeit in einem bewegten Behälter oder die Konstruktion von Trinkwasserspeichern, in denen nach Möglichkeit keine Totwassergebiete entstehen sollen, in denen kein Wasseraustausch mehr erfolgt, numerisch simuliert werden.

Die numerische Strömungsmechanik (engl. *Computational Fluid Dynamics* - CFD) ist eng mit der Entwicklung der Rechentechnik verbunden. Erst mit entsprechend leistungsstarken Computern wurde sie eine ernsthafte Alternative zum physikalischen Experiment.

Im folgenden wird ein explizites Verfahren der Zeitdiskretisierung erster Ordnung der Navier-Stokes-Gleichungen beschrieben, das auf der Basis einer speziellen Finite-Element-Formulierung ein freies Randwertproblem für turbulente Strömungen in einem zweidimensionalen Gebiet löst. Explizites Verfahren erster Ordnung bedeutet in diesem Fall, daß alle in der Differentialgleichung vorkommenden Größen, insbesondere die Ortsableitungen, zum Zeitpunkt $t^n$ ausgewertet werden, um die Ergebnisse zum Zeitpunkt $t^{n+1}$ zu erhalten. Die numerische Stabilität kann bei expliziten Verfahren nur durch sehr kleine Zeitschritte gesichert werden. Implizite Verfahren erlauben demgegenüber wesentlich größere Zeitschritte, erfordern aber in jedem Zeitschritt die Lösung eines linearen oder auch nichtlinearen Gleichungssystems. Bei implizite Verfahren werden die Abhängigkeiten der Ortsableitungen eines Zeitschrittes mit berücksichtigt.

Für die in Abschnitt 4.2.1 beschriebene Lösung des Problems wird dann in Abschnitt 4.2.2 eine Parallelisierungsstrategie vorgestellt. Die Leistungsfähigkeit des Parallelisierungsansatzes mit statischer Lastverteilung wird danach in Abschnitt 4.3 analysiert und es werden dann verschiedene Techniken der Leistungssteigerung des parallelen Algorithmus motiviert und in ihrem Effekt bewertet.

## 4.2 Ein freies Randwertproblem

### 4.2.1 Serielles Verfahren

Ein Testbeispiel für die Simulation freier Randwertprobleme ist der „brechende Damm“: Eine senkrechte Wand, die ein Flüssigkeitsbecken begrenzt und hinter der eine Flüssigkeit angestaut wurde, wird plötzlich entfernt, so daß sich die Flüssigkeit im Becken ausbreiten kann. Dieses Problem kann zum Beispiel mit einem expliziten Zeitschrittverfahren modelliert werden. Das Ausbreitungsgebiet der turbulenten, inkompressiblen Strömung ist zeitabhängig veränderlich. Eine Ausbreitungsfront bewegt sich durch das Simulationsgebiet.

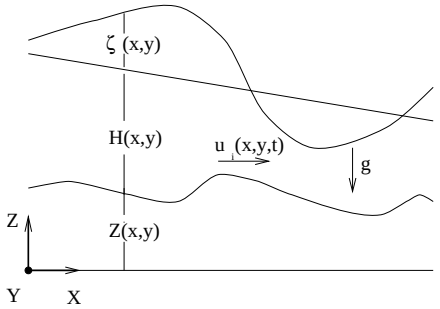


Abbildung 4.1: *Flachwasserströmung - Mechanisches System*

Die zugrundeliegende Flachwassergleichung mit Impulserhaltungs- und Kontinuitätsbedingungen kann unter Annahme der Inkompressibilität von Wasser und hydrostatischer Druckverteilung aus der dreidimensionalen Navier-Stokes-Gleichung abgeleitet werden [119]. Die Gleichungen lauten mit der mittleren Horizontalgeschwindigkeit $u$, der Wasserhöhe $\zeta$, der Wassertiefe $H$ und dem Bodenniveau $Z$ (vgl. Abbildung 4.1)

$$\begin{aligned} \dot{u}_i + (u_j u_i)_{,j} + g(H+\zeta+Z)_{,i} - A(u_{i,j}+u_{j,i})_{,j} + \frac{gn^2\sqrt{u_k u_k}}{(H+\zeta)^{4/3}} u_i &= 0 \\ \dot{\zeta} + \{(H+\zeta)u_i\}_{,i} &= 0. \end{aligned} \quad (4.1)$$

$A$ ist die künstliche Reibung, $g$ die Gravitationskonstante und $n$ der Manning-Koeffizient (oder ein Reibungsbeiwert zur Charakterisierung der Bodenbeschaffenheit).
Die Gleichungen werden linearisiert und können mit Hilfe des Standard-Galerkin-Verfahrens mit linearen Dreieckselementen diskretisiert werden. Die Kapazitätsmatrix $\bar{\mathbf{K}}$ und die Massenmatrix $\bar{\mathbf{M}}$ werden auf Elementebene formuliert. Mit einem numerischen Integrationsschema können dann die Gleichungen gelöst werden. Mit einer speziellen Technik der selektiven Diagonalisierung [119] können die Elementgleichungen entkoppelt werden, so daß die Elementknotenbeiträge unabhängig voneinander berechnet werden können. Die Zeitdiskretisierung erfolgt zur Sicherung der numerischen Stabilität in zwei bzw. drei Schritten. Im folgenden ist die Drei-Schritt-Formulierung [208] angegeben mit:
Schritt 1

$$\bar{\mathbf{M}} u_i^{n+1/3} = \bar{\mathbf{M}} u_i^n + \frac{1}{3}\Delta t \bar{\mathbf{K}}_1(u_i^n, \zeta^n)$$



$$\bar{\mathbf{M}} \zeta^{n+1/3} = \bar{\mathbf{M}} \zeta^n + \frac{1}{3}\Delta t \bar{\mathbf{K}}_2(u_i^n, \zeta^n) \quad (4.2)$$

Schritt 2

$$\begin{aligned} \bar{\mathbf{M}} u_i^{n+1/2} &= \bar{\mathbf{M}} u_i^n + \frac{1}{2}\Delta t \bar{\mathbf{K}}_1(u_i^{n+1/3}, \zeta^{n+1/3}) \\ \bar{\mathbf{M}} \zeta^{n+1/2} &= \bar{\mathbf{M}} \zeta^n + \frac{1}{2}\Delta t \bar{\mathbf{K}}_2(u_i^{n+1/3}, \zeta^{n+1/3}) \end{aligned} \quad (4.3)$$

und Schritt 3

$$\begin{aligned} \bar{\mathbf{M}} u_i^{n+1} &= \bar{\mathbf{M}} u_i^n + \Delta t \bar{\mathbf{K}}_1(u_i^{n+1/2}, \zeta^{n+1/2}) \\ \bar{\mathbf{M}} \zeta^{n+1} &= \bar{\mathbf{M}} \zeta^n + \Delta t \bar{\mathbf{K}}_2(u_i^{n+1/2}, \zeta^{n+1/2}) \end{aligned} \quad (4.4)$$

Die Elementmatrizen müssen dabei für jeden Zeitschritt aufgebaut werden. Die Ergebniswerte in den Knoten werden durch Akkumulation der Elementbeiträge ermittelt. Dabei sind bei diesem expliziten Verfahren jeweils nur die Knotenwerte des vorangegangenen Zeitschritts erforderlich. Mehr Einzelheiten zur verwendeten Technik, auch zur Unterdrückung der in den Beispielrechnungen sichtbaren numerischen Oszillation in der Ausbreitungsfront, sind in [109], [207] und [118] angegeben. Im Mittelpunkt der Betrachtung soll im folgenden die Parallelisierung dieses einfachen Ansatzes unter Vernachlässigung der numerischen Probleme stehen.
Die Besonderheit der Lösung des freien Randwertproblems ist die Berücksichtigung der Wasserausbreitung.

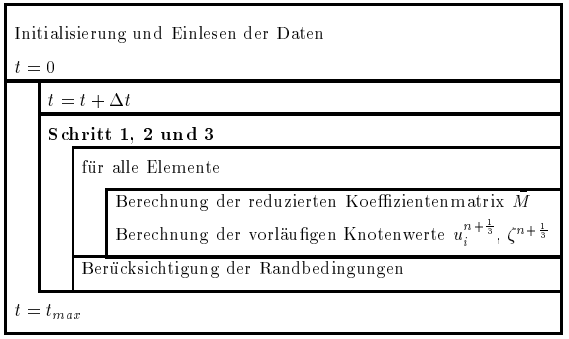


Abbildung 4.2: *Struktogramm serielle Berechnung*

Die zeitlich veränderliche Ausbreitungsfront im mechanischen und numerischen Modell wird mit der *moving boundary technique* abgebildet [120]. Der Berechnungsablauf ist im Struktogramm Abbildung 4.2 dargestellt. Bei dem benutzten Verfahren sind Elementmatrizen nur für die mit Wasser benetzten Elemente definiert. Entsprechend kann die Berechnung vereinfachend auf die Elemente beschränkt werden, die an allen drei Elementknoten eine von Null verschiedene Wasserhöhe besitzen. Der Rechenaufwand für einen Zeitschritt ist von der Anzahl der in dem Zeitschritt benetzten Elemente abhängig. Er ist damit veränderlich im Ablauf der Berechnung. Diese Situation stellt eine besondere Herausforderung an eine effiziente Parallelisierung dar.

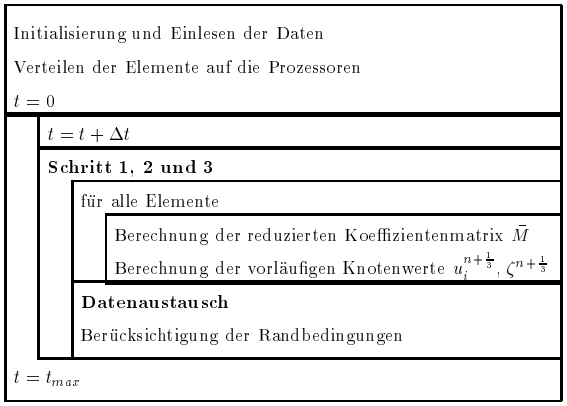


Abbildung 4.3: *Struktogramm parallele Berechnung*

### 4.2.2 Paralleles Verfahren

Die naive Verfahrensweise bei der Parallelisierung des expliziten Berechnungsalgorithmus nutzt das offensichtlich vorhandene Parallelisierungspotential in der Elementschleife aus. Ein erster Ansatz führt damit zu einer Parallelisierung des Rechenkerns durch Verteilung der Elemente auf die eingesetzten Prozessoren. Da sich die Knotenwerte aus der Summe der Einzelbeiträge der an einem Knoten anschließenden Elemente ergeben, stehen bei der teilgebietsweisen, parallelen Berechnung die Anteile aus den Elementen in jedem Teilgebiet sofort zur Verfügung. Die Parallelisierung erfordert aber zusätzlich einen Akkumulationsschritt für die Koppelknotenwerte. Dabei müssen alle Teilgebietsbeiträge für jeden Koppelknoten in jedem beteiligten Teilgebiet bereitgestellt werden.
Die Akkumulation muß für jedes Zeitintervall, in jedem Berechnungsschritt der Drei-Schritt-Technik durchgeführt werden. Damit ist der Berechnungsgang äquivalent zum seriellen Ablauf. Dieser zusätzliche Aufwand, der in der seriellen Bearbeitung nicht anfällt bestimmt wesentlich die Effizienz der Parallelisierung. Das Struktogramm des Algorithmus ist in Abbildung 4.3 dargestellt.
Folgende Komponenten eines parallelen Programms sind erforderlich:

**Gebietszerlegung** Aufteilen der Elemente (und der zugehöriger Knoten) auf die Prozessoren und Organisation einer Kommunikationsstruktur, d.h. Vergabe globaler Namen für Knoten, die auf mehr als einem Prozessor benutzt werden, Aufbau eines entsprechenden Verbindungsnetzwerkes

**Akkumulation** Sammeln der lokal berechneten Daten für die Koppelknoten im Austauschpuffer, Austauschen der globalen Daten, Verteilen der eintreffenden Daten auf die lokalen Daten einschließlich Akkumulation.

Die Gebietszerlegung wird mit einem Standardverfahren realisiert (vgl. Abschnitt 3.1.2). Die Anforderungen an die Gebietszerlegung werden im vorliegenden Problem lediglich aus dem Aufwand für den Datenaustausch auf dem Parallelrechner



formuliert. Die Gebietszerlegung muß daher die Kriterien

- gleichmäßige Rechenlast (gleiche Elementanzahl),
- geringes Datenaufkommen in der Kommunikation (kurze Gebietsränder) und
- geringe Anzahl Kommunikationspartner (wenige Gebietsnachbarn)

erfüllen. Anforderungen aus der numerischen Behandlung des Problems erwachsen nicht. Die Form der Gebiete ist beliebig. Auch die Reihenfolge der Abarbeitung innerhalb der Elementschleife ist beliebig und kann deshalb optimal an die Bedürfnisse der Parallelisierung angepaßt werden.

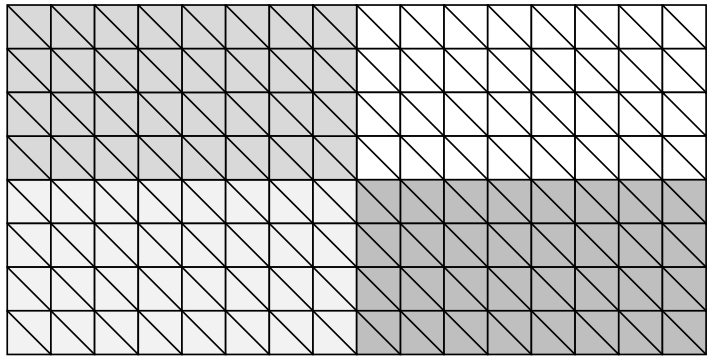

Abbildung 4.4: *Modellgebiet - unterteilt für vier Prozessoren*

Für ein rechteckiges Modellgebiet ist in Abbildung 4.4 eine mögliche Gebietszerlegung angegeben. Unter der Voraussetzung, daß das gesamte Modellgebiet von Wasser benetzt, und damit in die Berechnung einbezogen ist, garantiert die angegebene Zerlegung eine gleichmäßige Rechenlast auf den vier Prozessoren.

#### 4.2.2.1 Standardakkumulation

Die den Knoten zugeordneten Daten sind in Abbildung 4.5 symbolisch dargestellt. Die dunkel gekennzeichneten Datenfelder bezeichnen die den Koppelknoten zugeordneten Daten. Man erkennt, daß es bei einer am Finite-Element-Netz orientierten Datenstruktur fortlaufende und verteilt vorliegende Koppeldaten gibt. Entsprechend sind die Datenfelder für jeden Akkumulationsschritt noch aufzubauen und für jeden Austausch erneut bereitzustellen. Für den allgemeinen Fall eines unstrukturierten Gitters liegen alle Knotendaten verteilt vor. Der Datenaustausch inklusive Einsammeln, Austauschen, Verknüpfen und Verteilen der Knotenwerte ist für jeden Iterationszyklus in jedem Schritt des Algorithmus durchzuführen.
Als Varianten der Akkumulation sind der globale Austausch oder der Punkt-zu-Punkt Austausch möglich. Beim globalen Austausch erhalten allerdings alle Prozessoren Informationen über alle, auch die nicht in ihrem Teilgebiet liegenden Koppelknoten. Die Anzahl der Kommunikationsschritte steigt z.B. für den idealen Hypercube logarithmisch mit der Prozessorzahl und das Datenaufkommen linear mit der Anzahl der Koppelknoten im Gesamtsystem. Das Datenaufkommen steigt deshalb bei gleicher Modellgröße mit der Anzahl der Teilgebiete.

Damit das Datenaufkommen in diesem kostspieligen Schritt reduziert werden kann, wird die Akkumulation als Punkt-zu-Punkt-Austausch zwischen interessierten Prozessoren implementiert. Eine Rechteck-Partitionierung im Modellgebiet unterstellt, ergibt eine konstante Anzahl der Kommunikationsschritte je Teilgebiet (mit vier Teilgebieten über Gebietsränder und mit vier Teilgebieten über Kreuzungsknoten). Der Kommunikationsaufwand sinkt für steigende Prozessorzahlen bei kleiner werdenden Teilgebieten und Teilgebietsrändern. Allerdings ist zu beachten, daß der Austausch von zu kleinen Datenmengen dazu führt, daß der Kommunikationsaufwand im Wesentlichen von der Start-up Zeit bestimmt wird und nicht mehr linear mit der Anzahl der Koppelknoten pro Teilgebiet sinkt.

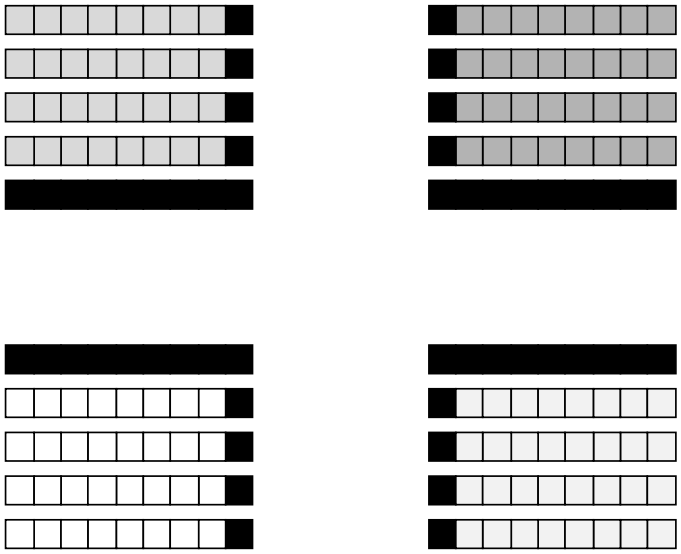

Abbildung 4.5: *Datenstruktur für knotenbezogene Daten*

Zur Realisierung des Superpositionsschrittes ist die Bereitstellung von fortlaufenden Speicherfeldern mit den erforderlichen Knotendaten für die Gebietsränder notwendig. Dabei sind auf jedem Prozessor jeweils Datenfelder für das Versenden der eigenen Knotenwerte und Datenfelder (gleicher Struktur) für das Empfangen der Knotenwerte des benachbarten Prozessors erforderlich. Eine Anzahl von Knotenwerte muß an mehrere Prozessoren verschickt und deshalb in mehrere Austauschfelder kopiert werden (vgl. Abbildung 4.6).
Bei streng sequentieller Abarbeitung der algorithmischen Teilschritte

- Berechnung der Knotenwerte
- Akkumulation der Koppelwerte

erfolgt in jedem Austauschschritt eine Synchronisation aller Prozesse. Der nächste Bearbeitungsschritt kann erst erfolgen, wenn alle Prozessoren ihre Kommunikationsanforderungen - Versenden der lokal berechneten Knotenwerte, Empfangen der auf



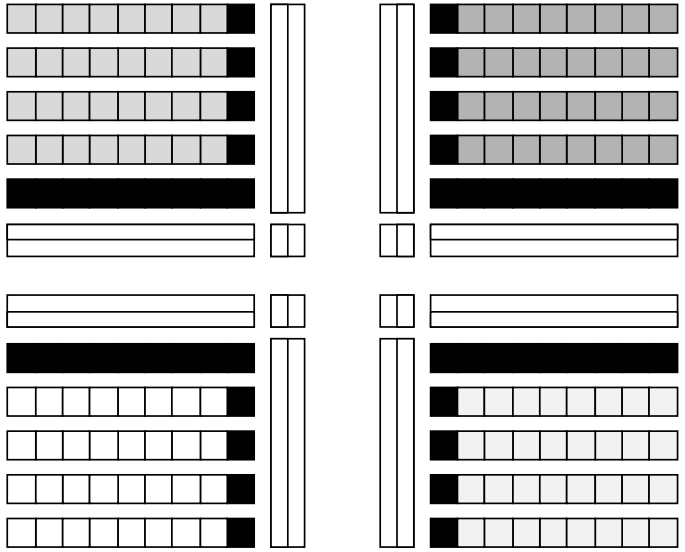

Abbildung 4.6: *Kommunikationsdatenstruktur für Koppeldaten*

den anderen Prozessoren berechneten Knotenwerte und Superposition zum Gesamtergebnis - realisiert haben. Ein lokales Lastungleichgewicht führt zu erheblich höheren Laufzeiten auf einzelnen Prozessoren und damit zu Stillstandszeiten anderer Prozessoren. Daraus resultiert ein erheblicher Effizienzverlust. Die Kommunikations- und Stillstandszeiten werden für die Laufzeit der Simulation relevant.

#### 4.2.2.2 Nichtblockierende Akkumulation

Mit einer Überlappung von Kommunikation und Berechung in einer nichtblockierenden Akkumulation kann dieser Engpaß umgangen werden. Der Datenaustausch erfolgt dann in den folgenden beiden Schritten:

**Schritt 1** Sobald alle Koppelinformationen auf einem Prozessor vorhanden sind werden die Datenfelder zum Versenden vorbereitet und nicht blockierend an die Nachbarn verschickt.

**Schritt 2** Von allen Nachbarn wird genau eine Nachricht erwartet, die sobald sie eingetroffen ist, mit den entsprechenden lokalen Koppelinformationen verknüpft (hier: superponiert) wird.

Zwischen Schritt 1 und Schritt 2 werden vom Prozessor sinnvolle Berechnungen durchgeführt.
Das Vorgehen ist optimal, wenn mit den zwischenzeitlich durchgeführten Berechnungen die Kommunikations- und Stillstandszeiten, die im Zusammenhang mit dem Nachrichtenaustausch entstehen vollständig überbrückt werden können.
Im hier vorgestellten Berechnungsverfahren ist die Reihenfolge der Elementbearbeitung beliebig. Die Elementschleife kann so gestaltet werden, daß zuerst alle Ele-

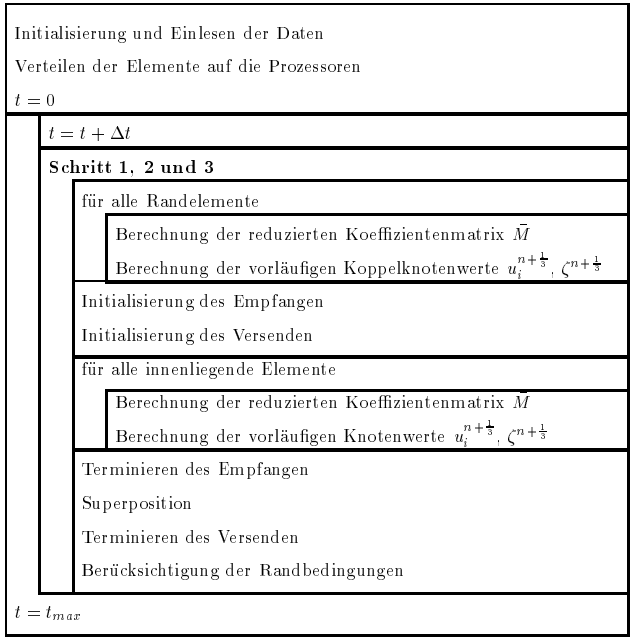


Abbildung 4.7: *Struktogramm parallele Berechnung mit nichtblockierender Akkumulation*



mente mit Koppelknoten bearbeitet werden und daran anschließend der Kommunikationsschritt 1 durchgeführt werden kann. Die Elementreihenfolge wird in einem initialen Sortierschritt entsprechend verändert. Die Berechnung wird mit den übrigen Elementen fortgesetzt. Schritt 2 wird durchgeführt, wenn die Elementschleife abgearbeitet ist.
Der Schritt 2 kann noch einmal in den eigentlichen Nachrichtentransfer (Empfangen einer Nachricht) und die Berechnung (Superposition) aufgespaltet werden. Während der Nachrichtentransfer in der Regel keine Rechenleistung erfordert, wird für die Superposition Rechenleistung abgefordert. So ist es günstiger, das Nachrichtenempfangen als nichtblockierende Operation unmittelbar nach Schritt 1 zu initialisieren. Während der anschließenden Berechnung mit den Elementen im Innengebiet können diese Nachrichtenoperationen terminieren. Die lokale Superposition wird im Anschluß an die Elementschleife durchgeführt, wenn die vollständige Rechenleistung wieder zur Verfügung steht. Im Idealfall kann das Datenempfangen zeitlich vollständig von der Berechnung der inneren Knotenwerte überlappt werden. Abbildung 4.7 zeigt das Struktogramm dieses optimierten parallelen Verfahrens.

#### 4.2.2.3 Datenstrukturen

Neben den übliche Datenstrukturen für die finite Elementberechnung, die Knotenkoordinaten und Anfangsrandbedingungen in den Knoten, Elementzuordungen zu Knoten und Elementbeschreibungen enthalten, werden für die Organisation der parallelen Berechnung folgende knoten- und elementbezogene Datenstrukturen benötigt:

- Zuordung der Knoten zu inneren Knoten und Koppelknoten,
- Zuordung der Koppelknoten zu bestimmten Rändern,
- Austauschpuffer für die Gebietsränder
- Zuordnung der Elemente zu Innengebieten (kein Elementknoten ist Koppelknoten) und
- Elementnachbarschaftsbeziehungen für den Partitionierungsalgorithmus

Zur Organisation des Datenaustausches werden weitere Informationen über die Kopplung der Teilgebiete benötigt:

- Nachbarschaftsbeziehungen der Teilgebiete
- globale Namen für die Koppelknoten und Zuordnungstabellen zwischen den internen Knotennamen und den globalen Koppelknotennamen

Die Datenstrukturen sind an eine Partitionierung gebunden und müssen bei einer Umverteilung von Elementen und Knoten aktualisiert werden. Sie können vor Beginn der Berechnung initialisiert werden. Auch die Austauschdatenstrukturen können vor Beginn der Berechnung bereits initialisiert werden.

## 4.3 Lastverteilung bei in Ort und Zeit veränderlichen Randbedingungen

### 4.3.1 Modellproblem

Für ein Modellproblem werden parallelisierte Berechnungen auf einem Transputersystem, einem parallelen PowerPC-System und einem Workstation-Cluster durchgeführt. Diese Parallelrechnersysteme unterscheiden sich in der Kommunikationshardware (Multiprozessornetzwerk bzw. Bussystem) und im Verhältnis von Kommunikations- und Arithmetikleistung. Ein Transputersystem mit verhältnismäßig

geringer Arithmetikleistung und entsprechend moderater Kommuikationsleistung ist besser balanciert, als eine Workstation mit relativ hoher Arithmetikleistung aber geringer Kommunikationsbandbreite. Das paralleles Berechnungsprogramm zeigt auf einem Workstation-Cluster auch wegen der hohen Start-up Zeiten trotz höherer Arithmetikleistung eine weniger gute Skalierbarkeit als die gleiche Implementation auf einem Transputersystem. Das PowerPC-System bietet bei einer mit dem Transputersystem vergleichbaren Kommunikationsleistung bessere Arithmetikleistung. Es ist zu erwarten, daß die Skalierbarkeit besser als für das Workstation-Cluster ist. Bei starkem Kommunikationsbedarf ist die Skalierbarkeit allerdings schlechter als beim Transputersystem.

Tabelle 4.1: *Daten für das Modellproblem*

| Bezeichnung | Elementzahl | Knotenzahl |
|---|---|---|
| kleines Problem | 6560 | 3411 |
| mittleres Problem | 13120 | 6691 |
| großes Problem | 26240 | 13381 |

Simuliert wird das Fluten eines ebenen, leeren Bassins. Berücksichtigt werden die ersten 6 Sekunden der Flutung, in denen die Wasserfront das Bassin vollständig durchläuft. Für die Modellierung werden drei verschiedene Diskretisierungen benutzt, um das Laufzeitverhalten für ein kleines, ein mittleres und ein großes Problem vergleichen zu können. Die Modelle haben die in Tabelle 4.1 aufgeführten Eigenschaften.

### 4.3.2 Statische Lastverteilung

Zunächst wurde die Berechnung mit einer statische Lastverteilung durchgeführt. Ungeachtet der tatsächlichen Rechenlastverteilung wurde das finite Elementnetz vor Beginn der Berechnung auf die gewünschte Anzahl von Prozessoren verteilt. Zur Anwendung kam dabei der Greedy-Algorithmus nach Farhat [58].

Tabelle 4.2: *Laufzeit (in s) auf Transputer T805 bei statischer Lastverteilung*

| Prozessorzahl | 1 | 2 | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|---|---|
| kleines Problem | 4750 | 2986 | 1621 | 882 | 476 | 267 | 157 |
| mittleres Problem | 23271 | 14978 | 7924 | 4171 | 2270 | 1246 | 756 |
| großes Problem | 74670 | 61482 | 31586 | 16721 | 8893 | 4903 | 2595 |

Eine Übersicht der Laufzeiten für das Transputersystem mit bis zu 64 Transputern vom Typ T805 gibt Tabelle 4.2. Die Laufzeiten für einen Transputer wurden mit einer speicheroptimierten, seriellen Programmvariante ermittelt. Die besten Laufzeiten wurden mit 64 Transputern erreicht (157s, 756s und 2595s).
Die Ergebnisse für das Workstation-Cluster sind in Tabelle 4.3 zusammengestellt. Das Cluster bestand aus bis zu 10 Sun Workstation Sparc 10/20. Als Verbindungsnetzwerk wurde Standard Thin-Ethernet eingesetzt. Die Implementierung basiert auf dem Paket *p4* [35]. Für die Berechnung des kleinen Problems mit 16 Prozessoren wurden auf 6 Workstation je zwei Berechnungsprozesse plaziert. Das Workstation-Cluster erreichte für das kleine Problem mit vier Prozessoren die beste Rechenzeit (185s) und mit 8 Prozessoren die absolut beste Rechenzeit für das mittlere und das



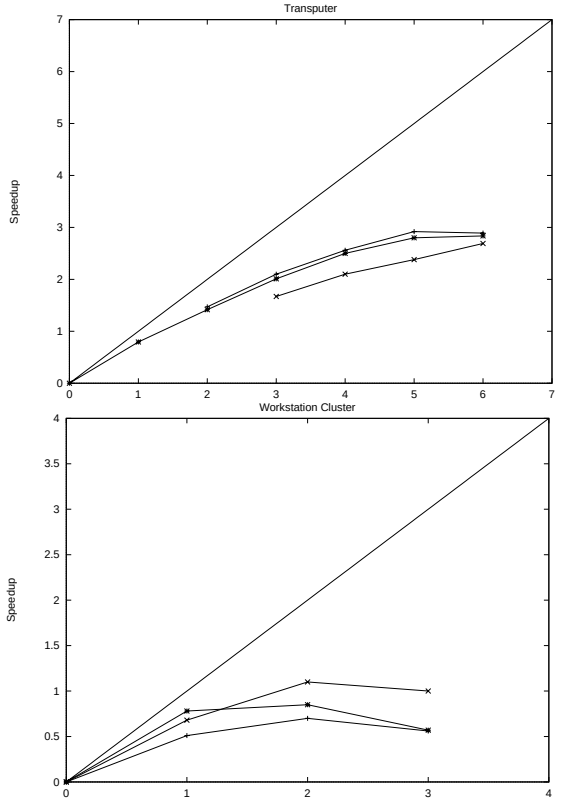


Abbildung 4.8: *Speed-up der Lösungsvarianten für Transputer und Workstation-Cluster*

Tabelle 4.3: *Laufzeit (in s) auf Workstation-Cluster bei statischer Lastverteilung*

| Prozessorzahl | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| kleines Problem | 316 | 202 | 185 | 209 | 473 |
| mittleres Problem | 900 | 887 | 641 | 608 | - |
| großes Problem | 3637 | 2675 | 1652 | 1373 | - |

Tabelle 4.4: *Laufzeit (in s) auf PowerPC bei statischer Lastverteilung und blockierender Kommunikation*

| Prozessorzahl | 1 | 2 | 4 | 8 | 16 | 32 |
|---|---|---|---|---|---|---|
| kleines Problem | 315 | 223 | 145 | 95 | 83 | 85 |
| mittleres Problem | 1552 | 1075 | 687 | 476 | 359 | 354 |
| großes Problem | 1955 | 4294 | 2611 | 1681 | 1282 | 1126 |

große Problem (608s bzw. 1373s). Die resultierenden Daten für den Speed-up auf Transputersystem und Workstation-Cluster sind in Abbildung 4.8 dargestellt.
Der Rechenzeitgewinn beim Einsatz von mehr Prozessoren ist nur für das Transputersystem akzeptabel. Mit zusätzlichen Transputern kann zugleich ein Zuwachs an Abarbeitungsgeschwindigkeit erreicht werden. Für das Workstation-Cluster ist der Zuwachs an Geschwindigkeit nicht adäquat dem Ressourceneinsatz. Im Gegenteil, aus den Ergebissen ist abzulesen, daß der Einsatz von mehr als vier Workstation keine weitere Steigerung der Abarbeitungsgeschwindigkeit ergibt. Der mit zunehmender Prozessorzahl wachsende Kommunikationsaufwand übersteigt den Laufzeitgewinn aus der Aufteilung der Berechnungsaufgabe auf mehr Teilgebiete. Die Anforderungen des Algorithmus an die Kommunikationsleistung sind so erheblich, daß der Transputer mit besser balancierter Rechen- und Kommunikationsleistung relative Vorteile gegenüber dem Workstation-Cluster hat. Das drückt sich in der besseren Skalierbarkeit des Algorithmus auf dem Transputersystem aus.

### 4.3.3 Laufzeitprofil

Zur genaueren Untersuchung der gewählten Programmlösung wurde das parallele Programm um detaillierte Laufzeitmessungen erweitert. Die Zeitmessung wurde so gestaltet, daß Aussagen über die Laufzeit von allen wesentlichen Programmabschnitten für jeden Prozessor getrennt möglich sind. Mit diesen Informationen kann dann ein Laufzeitprofil erstellt werden. Dieses Laufzeitprofil gibt Auskunft über die Verteilung der Rechen- und Stillstandszeiten während der Berechnung. Stillstandszeiten entstehen bei der parallelen Abarbeitung, wenn ein Prozessor auf Kommunikation warten muß und in dieser Wartezeit keine Berechnungen durchführen kann.

Tabelle 4.5: *Laufzeit (in s) auf PowerPC bei dynamischer Lastverteilung und blockierender Kommunikation*

| Prozessorzahl | 1 | 2 | 4 | 8 | 16 | 32 |
|---|---|---|---|---|---|---|
| kleines Problem | 315 | 187 | 127 | 101 | 95 | 104 |
| mittleres Problem | 1552 | 880 | 559 | 411 | 345 | 345 |
| großes Problem | 1955 | 3424 | 2117 | 1463 | 1145 | 1049 |



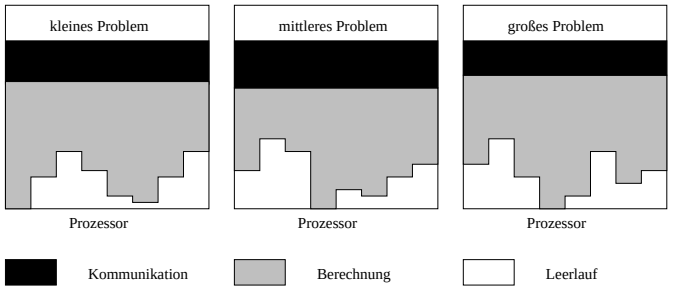


Abbildung 4.9: *Laufzeitprofil für 8 PowerPC (Berechnung des Modellproblems, statische Lastverteilung)*

Eine effiziente Parallelisierung muß solche Stillstandszeiten vermeiden.
Die Zeitmessungen wurden auf dem PowerPC-System durchgeführt. In Abbildung 4.9 ist die tatsächliche Auslastung von 8 PowerPC bei der Berechnung der unterschiedlich großen Modellprobleme angegeben. Die Zeitscheiben wurden dabei auf 100% normiert, so daß eine Aussage über die prozentuale Auslastung der Prozessoren möglich ist.
Man erkennt, daß die gegenüber der sequentiellen Variante zusätzlich erforderliche Kommunikation nur einen verschwindenden Anteil an der Gesamtlaufzeit des Algorithmus hat. Da in der parallelen Superposition eine Kommunikation nur zwischen benachbarten Prozessoren erforderlich ist, ist der relative Anteil der Kommunikation bei gleicher Modellgröße und steigender Prozessorzahl rückläufig.
Allerdings ist als Resultat der statischen Lastverteilung, bei der die finiten Elemente im Modellgebiet fest den Teilgebieten zugeordnet sind, der Aufwand zur Berechnung der Elementbeiträge (der nur für die benetzten Elemente erforderlich ist) über den Simulationszeitraum sehr ungleichmäßig verteilt. Der Prozessor, der das Einströmgebiet berechnet, hat in mit dem ersten Zeitschritt beginnend, garantiert die meisten Elementbeiträge zu berechnen, während andere Teilgebiete noch gar nicht vom Wasser benetzt sind und deshalb überhaupt keinen Beitrag zu berechnen haben. Die überdurchschnittlich beschäftigten Prozessoren blockieren damit auch alle anderen Prozessoren, die in jedem Zeitschritt ihre Koppeldaten aktualisieren müssen. Damit erklärt sich der erhebliche Verlust an Speedup bei steigender Prozessorzahl.
Die unnötig großen Leerlaufzeiten bei der parallelisierten Berechnung des Problems resultieren aus der starren Zuordnung der Elemente zu Teilgebieten. Diese Zuordnung soll deshalb in einem Algorithmus mit dynamischer Lastverteilung an die tatsächliche Rechenlast angepaßt werden.

### 4.3.4 Dynamische Lastverteilung

Die Idee ist nun, die Menge der im Problem vorhandenen Elemente in die Teilmenge der vom Wasser benetzten Elemente (in denen der Rechenaufwand entsteht) und die Teilmenge der trockenen Elemente (in denen keine Berechnung erforderlich ist) zu unterteilen. Nur die vom Wasser benetzten Elemente müssen auf die Prozessoren verteilt werden.
Allerdings kann sich bei diesem Herangehen in jedem Berechnungsschritt die Menge der benetzten Elemente verändern, indem neue Elemente hinzukommen. Eine

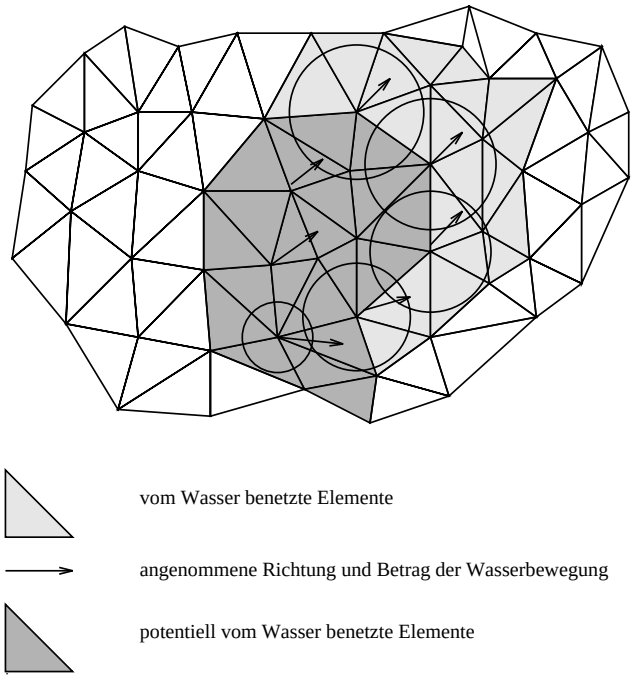


Abbildung 4.10: *Dynamische Bestimmung des zu verteilenden Gebiets*



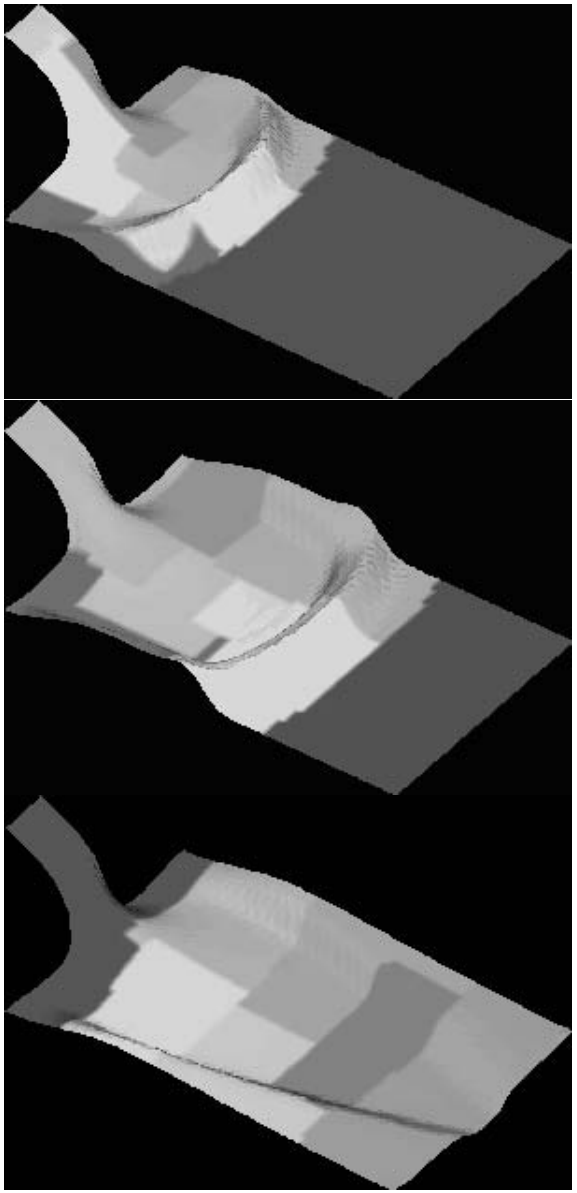

Abbildung 4.11: *Elementverteilung auf 8 Prozessoren nach 2s, 3s und 6s*

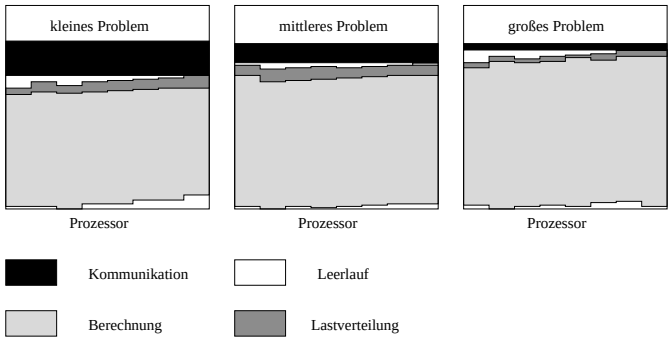


Abbildung 4.12: *Laufzeitprofil für 8 PowerPC (Berechnung des Modellproblems, dynamische Lastverteilung)*

Neuverteilung wird sofort wieder notwendig und der Gewinn aus dem reduzierten Aufwand im Berechnungsschritt muß gegen den Aufwand für die neue Partitionierung aufgerechnet werden. Es ist daher sinnvoll, zusätzlich eine Teilmenge der potentiell benetzten Elemente einzuführen, und diese in die Partitionierung einzubeziehen. Man erreicht damit um den Preis eines geringfügigen Lastungleichgewichts eine Reduktion der Häufigkeit für die Neuverteilung der Elemente.
Die potentiell benetzten Elemente bestimmt man aus dem zu einem Zeitschritt vorhandenen Geschwindigkeitsfeld und einer Anzahl von Zeitschritten, die bis zur nächsten Partitionierung vergehen soll. Unter der Voraussetzung, daß die Geschwindigkeit nicht wesentlich größer wird, kann man mit einer gewissen Sicherheit und unter Auswertung der Elementkantenlänge einen Ring von Elementen um die benetzten Elemente bestimmen, der wahrscheinlich naß werden wird.
Für jeden Knoten, an dem eine Wassergeschwindigkeit $v$ berechnet wird, ist der Ausbreitungsradius $r$ in einer vorgegebenen Anzahl von Simulationsschritten $n$ mit einem Zeitschrittweite $\delta_t$ und einem Sicherheitsbeiwert $\nu$ zu berechnen durch

$$r = v * n * \delta_t * \nu \tag{4.5}$$

Diese Untersuchung muß natürlich nur für die Knoten durchgeführt werden, die auf der Ausbreitungsfront liegen, d.h. für die Knoten, die Elementknoten in einem Element sind, das nicht vollständig von Wasser benetzt ist. Beispielhaft ist die Bestimmung der potentiell vom Wasser benetzten Elemente in Abbildung 4.10 angegeben.
Zu beachten ist bei der Bestimmung der Teilmengen, daß eine Mindestanzahl von Elementen zur Verteilung kommen sollte, um alle Prozessoren in die Berechnung einzubeziehen. Die Teilgebiete sollten groß genug sein, um eine sinnvolle Verteilung von Berechnungs- und Kommunikationsaufwand zu ermöglichen.
Das Laufzeitprofil bei Nutzung von asynchroner, überlappender Kommunikation und der dynamischen Lastverteilung sind in Abbildung 4.12 dargestellt. Die Lastverteilung hat nur bei den kleinen Problemen signifikanten Anteil an der Laufzeit des Algorithmus. Die Leerlaufzeiten konnten erheblich reduziert werden (vgl. Abbildung 4.9). Entscheidend für die Effizienz der vorgeschlagenen Lösung ist die Reduktion des Kommunikationsanteils. Mit der vollständigen berlagerung von Kommunikation und Berechnung kann man ein sehr gut skalierbares Verfahren implementieren.



Tabelle 4.6: *Laufzeit (in s) auf PowerPC bei statischer Lastverteilung und nichtblockierender Kommunikation*

| Prozessorzahl | 1 | 2 | 4 | 8 | 16 | 32 |
|---|---|---|---|---|---|---|
| kleines Problem | 315 | 205 | 118 | 71 | 49 | 37 |
| mittleres Problem | 1552 | 993 | 586 | 342 | 224 | 161 |
| großes Problem | 1955 | 4009 | 2276 | 1276 | 793 | 544 |

Tabelle 4.7: *Laufzeit (in s) auf PowerPC bei dynamischer Lastverteilung und nichtblockierender Kommunikation*

| Prozessorzahl | 1 | 2 | 4 | 8 | 16 | 32 |
|---|---|---|---|---|---|---|
| kleines Problem | 315 | 169 | 103 | 72 | 58 | 54 |
| mittleres Problem | 1552 | 799 | 463 | 291 | 206 | 174 |
| großes Problem | 1955 | 3152 | 1782 | 1048 | 684 | 522 |

### 4.3.5 Vergleich der Lösungsvarianten

Die Skalierbarkeit von vier Lösungsvarianten wird nun anhand der drei bereits beschriebenen Modellvarianten (kleines, mittleres und großes Problem) untersucht. Die Modellprobleme werden dafür so auf einer ausgewählten Anzahl von Pozessoren bearbeitet, daß diese mit maximal 820, 1640 und 3280 Elementen belastet sind. Ensprechend kann für die kleine Belastung mit bis zu 820 Elementen das kleine Problem auf 2, 4 und 8 Prozessoren, das mittlere Problem auf 4, 8 und 16 Prozessoren und das große Problem auf 8, 16 und 32 Prozessoren abgearbeitet werden. Die Angabe der Maximalzahl der Elemente bezieht sich auf den Fall der dynamischen Lastverteilung, bei der erst bei der Verteilung des vollständig mit Wasser benetzten Modellgebiets diese Anzahl von Elementen ergibt. Die resultierenden Effizienzen der folgenden Varianten sind in Abbildung 4.13 dargestellt:

- statische Lastverteilung (Originalalgorithmus, vgl. [209]),
- statische Lastverteilung und asynchrone Kommunikation,
- dynamische Lastverteilung und
- dynamische Lastverteilung und asynchrone Kommunikation.

Die besten Laufzeiteigenschaften und die beste Skalierbarkeit zeigen die Programmlösungen mit asynchroner Kommunikation. Sie liegen mindestens 20 Prozentpunkte über den vergleichbaren Laufzeiten für die synchrone Kommunikation. Die Skalierbarkeit kann durch die dynamische Lastverteilung noch verbessert werden. Das Feintuning der parallelen Lösungsalgorithmen ergibt eine Verbesserung der Effizienz um 41 Prozentpunkte von 40% für den Originalalgorithmus auf 81% für die beste Lösung mit asynchroner Kommunikation (vergleiche auch [129]).

## 4.4 Zusammenfassung

Es wurde eine effiziente Parallelisierung expliziter Zeitschrittverfahren zur Simulation zeitlich veränderlicher, turbulenter Strömungen inkompressibler Flüssigkeiten vorgestellt. Dabei lag ein Schwerpunkt auf der statischen Lastverteilung und der

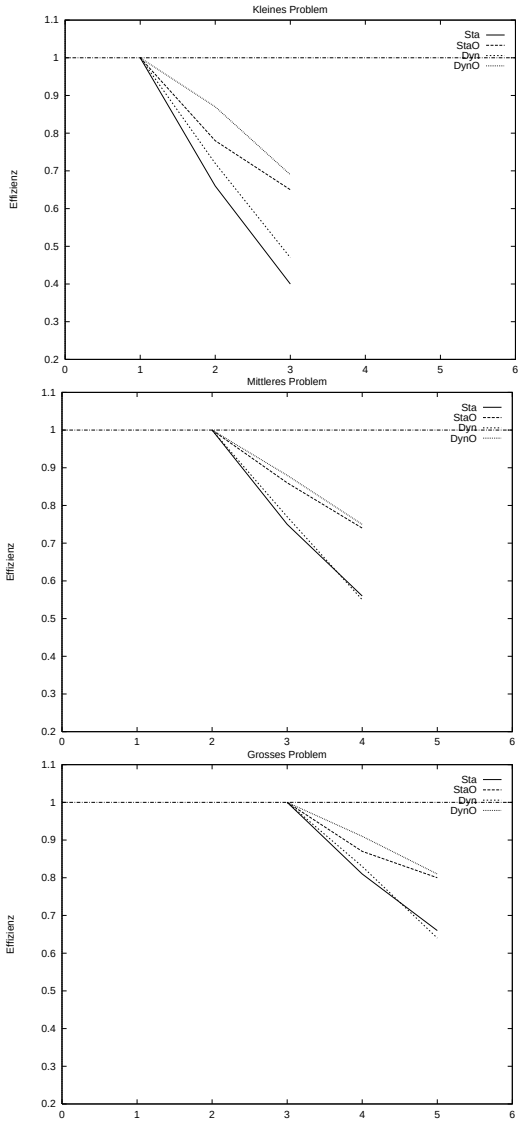


Abbildung 4.13: *Effizienz der Lösungsvarianten auf PowerPC*



dafür erforderlichen neuen Datenstrukturen für die Lösung des Problems mit finite Elementen.
Unter Berücksichtigung der zeitlich veränderlichen freien Randbedingungen bei der Ausbreitung der Flüssigkeit im Berechnungsgebiet entstehen weitergehende Anforderungen an eine effiziente Parallelisierung. Die problemabhängige Lastverteilung mit einem Verfahren der dynamischen Lastzuordnung brachte einen wesentlichen Effektivitätszuwachs. In Abhängigkeit von den Eigenschaften des benutzten Parallelrechners (hier besonders das verhältnis von Kommunikations- zu Rechenleistung) konnte eine zufriedenstellende algorithmische Lösung mit einer Kombination der dynamischen Lastzuordnung und der Überlappung von Kommunikation und (sinnvoller) Berechnung erreicht werden.

# Kapitel 5

# Lösungsverfahren für dünnbesetzte Gleichungssysteme

Die Formulierung des Elastizitätsproblems als Variationsproblem erfolgt mit gegebenen Gebietsbelastungen $f$ und Gebietsrandlasten $h$ sowie dem unbekannten Verschiebungsfeld $u$ mit

$$\Pi(u) = \frac{1}{2}a(u,u) - (u,f) - (u,h)_\Gamma \tag{5.1}$$

Dabei gelten

$$\begin{aligned} a(u,w) &= \int_\Omega u_{i\,j} C^{i\,jkl} u_{kl}\, d\Omega \\ (u,f) &= \int_\Omega u_i\, f_i\, d\Omega \\ (u,h)_\Gamma &= \int_{\Gamma_k} u_i\, h_i\, d\Gamma \end{aligned} \tag{5.2}$$

Ein Verschiebungsfeld $u$, das in Bezug auf kleine Änderungen $\delta u$ das Funktional $\Pi(u)$ stationär macht, d.h. die Bedingung

$$\delta\Pi = 0 \tag{5.3}$$

erfüllt, ist eine Lösung des Minimalproblems (5.1). Näherungslösungen lassen sich zum Beispiel mit dem Reihenansatz

$$\hat{v} = \sum_i N_i\, v_i \tag{5.4}$$

konstruieren. Daraus folgt

$$\delta\Pi = \frac{\partial\Pi}{\partial v_1}\delta v_1 + \frac{\partial\Pi}{\partial v_2}\delta v_2 + \ldots + \frac{\partial\Pi}{\partial v_n}\delta v_n = 0 \tag{5.5}$$

Das ist für beliebige Variationen $\delta v_i$ nur erfüllt, wenn

$$\frac{\partial \Pi}{\partial v_i} = \left\{ \begin{array}{c} \frac{\partial \Pi}{\partial v_1} \\ \vdots \\ \frac{\partial \Pi}{\partial v_n} \end{array} \right\} = 0 \tag{5.6}$$

gilt. Damit werden die Parameter $v_i$ so bestimmt, daß wiederum der Stationaritätspunkt des Funktionals $\Pi(u)$ erreicht wird. Da (5.1) ein quadratisches Funktional ist, indem die Verschiebungsfunktion $u$ und ihre Ableitungen höchstens als quadratische Ausdrücke auftreten, reduziert sich (5.3) auf eine lineare Form mit

$$\frac{\partial \Pi}{\partial v_i} \delta v_i = \delta v_i (\mathbf{K}v + f) = 0 \tag{5.7}$$

Die Matrix $\mathbf{K}$ ist stets symmetrisch und positiv definit

$$\mathbf{K} = \mathbf{K}^T \quad \text{bzw.} \quad \delta v^T \mathbf{K} \delta v > 0. \tag{5.8}$$

Die Matrix ist dünnbesetzt und es lohnt sich, bei der Berechnung und Speicherung der Matrix leere Matrixelemente nicht zu berücksichtigen. Die Einsparung von Speicherplatz und numerischen Operationen ist signifikant. Ein Kriterium zur Charakterisierung als dünnbesetzt ist, das nur $O(n)$ Matrixelemente ungleich Null sind, d.h. die Anzahl der wesentlichen Matrixelemente pro Zeile ist unabhängig von der Dimension der Matrix, sie ist begrenzt und sie ist bedeutend kleiner als die Matrixdimension.
Dünnbesetzte Gleichungssysteme haben eine überragende Bedeutung bei der Lösung von Aufgaben der Strukturmechanik mit Hilfe der Methode der finiten Elemente. Die numerischen Algorithmen zur Lösung von dünnbesetzten Gleichungssystemen sind wesentlich komplexer strukturiert als vergleichbare Algorithmen für regelmäßig oder vollbesetzte Gleichungssysteme. Aufgrund der unregelmäßigen Speicherzugriffsmuster und des größeren Verwaltungsaufwandes wird man deshalb für diese Probleme in der Regel nicht die arithmetische Spitzenleistung eines Rechners dokumentieren. Dafür erhält man aber eine umso realistischere, weil an der Praxis orientierten Aussage, über die Leistungsfähigkeit der Implementierung und des benutzten Rechners.
Anhand der Cholesky-Faktorisierung als ein Vertreter der direkten Lösungsverfahren für symmetrische Gleichungssysteme werden wesentliche Eigenschaften der Parallelisierung von Matrixalgorithmen diskutiert. Unvollständige Faktorisierungen (ohne oder nur mit teilweiser Berücksichtigung des Auffüllens der Matrix) mit dem analogen algorithmischen Apparat haben außerdem als Vorkonditionierungsverfahren für iterative Löser Bedeutung. Sie werden zunächst in ihrer seriellen Variante diskutiert.
Das Gebietszerlegungsverfahren erlaubt eine effektive Parallelisierung der iterativen Lösungsmethoden. Zwei Ansätze, die zum einen auf der Anwendung der Substrukturtechnik, d.h. der statischen Kondensation des Gleichungssystems und zum anderen auf der Einführung von Lagrange-Multiplikatoren und damit der Erfüllung der Übergangsbedingungen durch zusätzliche Gleichgewichtsbedingungen beruhen, werden mit ihrer algorithmischen Umsetzung gegenübergestellt.

## 5.1 Direkte, serielle Lösungsverfahren

Man betrachte das lineare Gleichungssystem

$$\mathbf{K}v = f \tag{5.9}$$



mit $\mathbf{K}$ als symmetrische, positiv definite Matrix $\mathbf{K} \in R^n$, $f$ als Vektor der rechten Seite und $v$ als Unbekanntenvektor, der berechnet werden soll. Dann ist eine Möglichkeit, dieses lineare Gleichungssystem zu lösen, die Cholesky-Faktorisierung

$$\mathbf{K} = \mathbf{L}\mathbf{L}^T \tag{5.10}$$

zu berechnen. Dabei ist $\mathbf{L}$ eine untere Dreiecksmatrix mit positiven Diagonalelementen. Der Lösungsvektor kann dann durch sukzessives Vorwärtseinsetzen und Rückwärtssubstituieren berechnet werden

$$\mathbf{L}\bar{v} = f, \mathbf{L}^T v = \bar{v} \tag{5.11}$$

Während der Faktorisierung werden einige Matrixelemente in der unteren Dreiecksmatrix $LL$ ungleich Null, an deren Position in der Matrix $\mathbf{K}$ keine Einträge vorhanden waren. Diesen Vorgang bezeichnet man als Auffüllen (engl. *fill-in*). Für eine schnellere Lösung des Gleichungssystems ist es entscheidend, daß der *fill-in* minimal ist, weil die mathematischen Operationen nur mit den wesentlichen Matrixelementen ausgeführt werden müssen. Minimaler *fill-in* sichert außerdem eine speichereffiziente Bearbeitung.
$\mathbf{K}$ ist für viele Anwendungen in der Finite-Element-Methode positiv definit. Sämtliche Hauptdiagonalelemente sind ungleich Null. Der Austausch von Spalten und Zeilen hat keinen Einfluß auf die numerische Stabilität der Faktorisierung. Vernachlässigt man Rundungsfehler, dann ist das Ergebnis sogar unabhängig von der Reihenfolge der Gleichungen oder Unbekannten. Damit hat man die notwendige Freiheit, die Struktur der dünn besetzten Matrix durch Umsortieren von Zeilen und Spalten so zu ändern, daß der *fill-in* minimiert wird.
Es sei $\mathbf{P}$ eine beliebige Permutationsvorschrift. Für die positiv definite Matrix $\mathbf{K}$ ist dann auch $\mathbf{P}\mathbf{K}\mathbf{P}^T$ positiv definit. Wir können also $\mathbf{P}$ so wählen, daß sich ein geeignetes Belegungsmuster der Matrix $\mathbf{P}\mathbf{K}\mathbf{P}^T$ ergibt, d.h. der *fill-in* im Choleskyfaktor $\bar{\mathbf{L}}$ von $\mathbf{P}\mathbf{K}\mathbf{P}^T$ geringer ist als in $\mathbf{L}$. Das permutierte Gleichungssystem ist in der gleichen Weise geeignet, das Ausgangssystem zu lösen. Die Vorwärts/Rückwärtsphase wird zu

$$\bar{\mathbf{L}}\bar{v} = \mathbf{P}f, \bar{\mathbf{L}}^T \bar{v} = \bar{v}, v = \mathbf{P}^T \bar{v} \tag{5.12}$$

Da eine Pivotisierung im Faktorisationsschritt nicht notwendig ist, kann bei Kenntnis einer geeigneten Permutation $\mathbf{P}$ die genaue Lokalisation des *fill-in* in $\mathbf{L}$ bestimmt werden. Die für die Speicherung von $\mathbf{L}$ notwendige Datenstruktur kann damit aufgebaut werden, bevor die numerische Faktorisierung beginnt. Sie muß dann während der Faktorisierung nicht erweitert werden. Das garantiert eine effiziente Bearbeitung. Den Prozeß des Aufbaus dieser Datenstruktur bezeichnet man als symbolische Faktorisierung. Die Lösung des Gleichungssystems besteht damit aus folgenden vier Schritten

**Sortierung** Finde eine gute Permutation $\mathbf{P}$ so, daß der Choleskyfaktor $\mathbf{L}$ von $\mathbf{P}\mathbf{K}\mathbf{P}^T$ möglichst wenig *fill-in* aufweist.

**Symbolische Faktorisierung** Bestimme die Struktur von $\bar{\mathbf{L}}$ und initialisiere die Datenstruktur mit den Elementen von $\mathbf{K}$.

**Numerische Faktorisierung** Berechne den Choleskyfaktor $\bar{\mathbf{L}}$ von $\mathbf{P}\mathbf{K}\mathbf{P}^T$.

**Dreieckslösung** Berechne $v$ entsprechend 5.12.

Die ersten beiden Schritte erfolgen nur symbolisch, ohne aufwendige Gleitkommaarithmetik. Im folgenden soll jeder dieser Schritte im einzelnen dargestellt werden.

### 5.1.1 Sortierung

Die Graphentheorie ist ein sinnvolles Hilfsmittel zur Beschreibung der Faktorisierung einer dünnbesetzten Matrix. Der ungeerichtete Graph einer Matrix $\mathbf{K}$ der Ordnung $n$ $G(\mathbf{K}) =< V(G), E(G) >$ besteht aus der Knotenmenge $V(G)$ mit $V = \{v_1, \ldots, v_n\}$ (entsprechend den Unbekannten) und einer Kantenmenge $E(G)$, wobei zwischen zwei Knoten $i$ und $j$ genau dann eine Kante $v_i v_j$ existiert, wenn gilt $k_{ij} \neq 0$ und $i \neq j$.
Der Effekt der Faktorisierung nach Cholesky auf die Matrixstruktur kann dann einfach anhand des zugehörigen Graphen beschrieben werden. Der ungerichtete Graph $G(\mathbf{L}) = G_{fak}(\mathbf{K})$ des Cholekyfaktors von $\mathbf{K}$ Ein neues Nichtnullelement als Resultat der Elimination einer Variablen fügt Kanten in den Graph so ein, daß die Nachbarn des eliminierten Knotens auf einem Pfad liegen:
Daraus kann man schlußfolgern, daß das *fill-in* minimiert oder gar vermieden werden kann, wenn man zunächst die Knoten eliminiert, die die wenigsten Nachbarn haben (engl. *minimum degree*).
Der Eliminations- oder Faktorisierungsprozeß kann somit durch eine Folge von Graphen beschrieben werden, wobei jeder Graph einen Knoten weniger aber eventuell mehr Kanten besitzt als sein Vorgänger. Außerdem kann man den Graphen $F(\mathbf{K})$ erzeugen, der die Struktur der unteren Dreiecksmatrix mit allen *fill-in*-Elementen besitzt. In diesem Graphen haben zwei Knoten $i$ und $j$ eine gemeinsame Kante, wenn der zugehörige Matrixeintrag $l_{ij}$ im Choleskyfaktor $\mathbf{L}$ ein Nichtnullelement ist.
Diese Überlegungen sind die Grundlage für den *minimum degree* Algorithmus [175][1]. Er ist die erfolgreichste und am weitesten verbreitete Heuristik zur Minimierung des *fill-in* bei der Choleskyfaktorisierung. In jedem Schritt der Eliminationsverfahrens wählt diese Heuristik den Knoten mit dem kleinsten Grad (mit den wenigsten Nachbarn, d.h. mit dem potentiell geringsten *fill-in*) aus den Nachbarn des Eliminationsknotens im vorausgegangen Schritt aus.
Das Problem der Wahl des Startknotens ist dabei nicht gelöst. Die Auswahlkriterien für Knoten bei Chancengleichheit von Kandidaten in einem Eliminationsschritt sind nicht ausreichend motiviert. Die Wahl des Startknotens und die Auswahl eines Knotens aus einer Liste gleichwertiger Knoten kann für spezielle Topologien heuristisch verbessert werden. Es gibt aber kein robustes Verfahren, um die große Streubreite der Ergebnisse des *minimum degree* Algorithmus für allgemeine Probleme einzuschränken. Trotzdem ist dieser Algorithmus, gerade wegen seiner Einfachheit und Effizienz, das Mittel der Wahl zur Bestimmung optimaler Sortierungen für die Choleskyfaktorisierung.
Eine Alternative zur Erzeugung geeigneter Permutationen ist das Verfahren der rekursiven Zweiteilung (engl. *nested dissection*). Dieses Verfahren basiert auf dem Prinzip Teile-und-Herrsche: Es sei $S$ eine Menge von Knoten (Separatoren), deren Streichung aus dem Graphen $G(\mathbf{K})$ der Matrix $\mathbf{K}$ zwei nicht zusammenhängende Graphen erzeugt. Werden die Knoten der beiden Teilgraphen fortlaufend numeriert und die Knoten des Separators zuletzt, dann hat das Belegungsmuster der resultierenden Matrix Blockstruktur. Damit kann die Faktorisierung der Teilmatrizen entkoppelt werden.
Die Elimination eines Knoten in einem Teilgraphen hat Auswirkungen nur in diesem Teilgraphen selbst und auf die Matrixelemente, die dem Separator zugeordnet sind. Sie hat keine Auswirkungen im anderen Teilgraphen. Das Verfahren kann rekursiv auf die Teilgraphen angewendet werden. Die Güte der Sortierung hängt von der Größe der Separatoren ab. Dieses Verfahren ist identisch mit der Suche nach einer möglichst effizienten Zerlegung eines Graphen in Teilgraphen mit wenigen Koppelknoten und vielen, vollständig innerhalb eines Teilgraphen liegenden Knoten. Für reguläre, zweidimensionale Diskretisierungen können effektive Separatoren einfach bestimmt werden. Das Verfahren wird wesentlich aufwendiger und die Zerlegung entsprechend weniger wirkungsvoll, wenn Gleichungssysteme für dreidimensionale und hochgradig irreguläre Diskretisierungen mit lokalen Verfeinerungen und unregelmäßigen Vernetzungen betrachtet werden.

[1] Der Algorithmus wurde 1969 von Tinney [203] für symmetrische Matrizen implementiert. Er entspricht dem sog. *Markowitz* Verfahren für unsymmetrische Matrizen [143].



### 5.1.2 Symbolische Faktorisierung

Nach der Bestimmung einer geeigneten Sortierung ist die Struktur des Choleskyfaktors zu bestimmen. Der naive Ansatz geht von einer symbolischen Umsetzung des eigentlichen Choleskyalgorithmus aus. Dieser Algorithmus hat dieselbe Komplexität wie das Choleskyverfahren, d.h. er würde die gleiche Anzahl von mathematischen Operationen erfordern wie die numerische Faktorisierung. Mit Hilfe der folgenden Überlegung kann der Aufwand für die symbolische Faktorisierung auf $O(h(\mathbf{L}))$ reduziert werden. $h$ bezeichnet dabei die Anzahl der Nichtnulleinträge im Cholekyfaktor $\mathbf{L}$. Für eine gegebene, dünnbesetzte Matrix $\mathbf{K}$ definieren wir

$$Struct(\mathbf{K}_{i*} := \{j < i | k_{ij} \neq 0\} \tag{5.13}$$

und

$$Struct(\mathbf{L}_{i*} := \{j > i | l_{ij} \neq 0\} \tag{5.14}$$

(5.13) bezeichnet die Belegungsstruktur der Zeile $i$ der strikten unteren Dreiecksmatrix von $\mathbf{K}$ und (5.14) bezeichnet die Belegungsstruktur der Spalte $j$ der strikten unteren Dreiecksmatrix von $L$. Für einen gegebenen Choleskyfaktor (untere Dreiecksmatrix) $L$ sei die Funktion $p$ wie folgt definiert

$$p(j) := \begin{cases} min\{i \in Struct(\mathbf{L}_{*j})\} & : \text{ für } Struct\mathbf{L}_{*j} \neq 0 \\ j & : \text{ sonst} \end{cases} \tag{5.15}$$

Existiert also mindestens ein Nebendiagonalelement in Spalte $j$ der Matrix $\mathbf{L}$, dann ist $p(j)$ der Index des ersten Nebendiagonalelements. Es ist einfach zu zeigen, daß

$$Struct(\mathbf{L}_{*j}) \subseteq Struct(\mathbf{L}_{*,p(j)}) \cup \{p(j)\}. \tag{5.16}$$

Weiterhin läßt sich zeigen, daß die Struktur der Spalte $j$ von $L$ mit

$$Struct(\mathbf{L}_{*j}) := Struct(\mathbf{K}_{*j}) \cup \left[ \bigcup_{i<j} \{Struct(\mathbf{L}_{*i}) | p(i) = j\} \right] - \{j\} \tag{5.17}$$

charakterisiert werden kann.
Die Struktur der Spalte $j$ der Matrix $\mathbf{L}$ unter der Diagonalen ist damit bestimmt durch die Struktur dieser Spalte in der unteren Dreiecksmatrix von $\mathbf{K}$ und der Struktur einer jeden Spalte von $\mathbf{L}$, deren erstes Nichtnullelement in Zeile $j$ liegt. Ein Beispiel dafür zeigt Abbildung 5.1. In Spalte 4 ergibt sich das Belegungsmuster aus der Vereinigung mit der Struktur von Spalte 3, Spalte 5 hat bereits die Struktur analog Spalte 2, wie auch Spalte 6 bereits analog Spalte 1 aufgebaut ist. In Spalte 7 ergibt sich das Belegungsmuster aus der Vereinigung mit Spalte 4.
Diese Eigenschaft der Matrixstruktur führt direkt auf einen Algorithmus zur symbolischen Faktorisierung entsprechend Abbildung 5.2. Die Mengen $R_j$ werden benutzt, um die Spalten von $\mathbf{L}$ zu speichern, deren Struktur die Spalte $j$ in $\mathbf{L}$ beeinflussen.
Der Algorithmus zur symbolischen Faktorisierung läßt sich weiter vereinfachen. Zum Beispiel gilt für Mengen $R_j$ mit einem Element $i$ und $Struct(\mathbf{K}_{*j}) \subseteq Struct(\mathbf{L}_{*i})$ sofort $Struct(\mathbf{L}_{*j}) = Struct(\mathbf{L}_{*i}) - \{j\}$. Mit dieser Technik (engl. *subscript supression*) lassen sich die Abarbeitungsgeschwindigkeit des Algorithmus erhöhen und

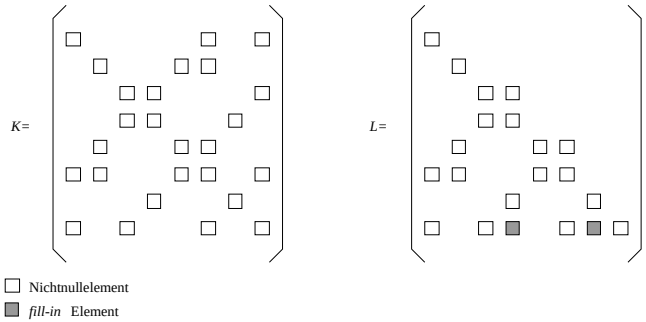


Abbildung 5.1: *Struktur einer Matrix und ihres Cholesky-Faktors*

für $j = 1$ bis $n$
  $R_j = 0$
für $j = 1$ bis $n$
  $S = Struct(\mathbf{K}_{*j})$
  für $i \in R_j$
    $S = S \cup Struct(\mathbf{L}_{*j}) - \{j\}$
  $Struct(\mathbf{L}_{*j}) = S$
  falls $Struct(\mathbf{L}_{*j}) \neq 0$
    $p(j) = min\{i \in Struct(\mathbf{L}_{*j})\}$
    $R_{p(j)} = R_{p(j)} \cup j$

Abbildung 5.2: *Symbolische Faktorisierung*

der Speicherbedarf verringern. Je größer $j$ wird, umso größer ist die Wahrscheinlichkeit, daß die Spalten im Laufe des Eliminationsprozesses bereits dicht gefüllt sind. Damit wächst die Wahrscheinlichkeit, daß oben genannte Bedingung erfüllt ist. Mit seiner geringen Komplexität und bei effizienter Implementierung ist die symbolische Faktorisierung gewöhnlich die am schnellsten zu lösende Teilaufgabe der Gleichungslösung.
Ist die Struktur der unteren Dreiecksmatrix bekannt, dann können die Datenstrukturen initialisiert werden. Die Ausnutzung der dünnen Besetzung erfordert das Speichern zusätzlicher Informationen über das Belegungsmuster. Auch hier kann man die *subscript supression*-Technik benutzen, um gleiche Spaltenstrukturen gemeinsam zu speichern. Der zusätzliche Aufwand für die Speicherung der Zeilen- bzw. Spalten-Informationen wird damit minimiert.

### 5.1.3 Numerische Faktorisierung

In ihrer einfachsten Form kann die Elimination einer dichtbesetzten Matrix $\mathbf{K}$ durch drei ineinander geschachtelte Schleifen über eine einzige Anweisung formuliert werden. Die Reihenfolge des Durchlaufes der Indizes $i, j$ und $k$ ist dabei beliebig. Es wird damit lediglich eine unterschiedliche Reihenfolge beim Zugriff auf den Hauptspeicher



bewirkt. Die freie Wahl der Schleifenreihenfolge erlaubt es, spezifische Eigenschaften von verschiedenen Rechnerarchitekturen (Cache, virtueller Speicher, Vektorregister u.a.) optimal auszunutzen. Je nachdem, welchen Index man als äußeren Schleifenindex benutzt, ergeben sich für den symmetrischen Fall der Choleskyfaktorisierung drei Grundtypen von Algorithmen für die Erzeugung der unteren Dreiecksmatrix $\mathbf{L}$:

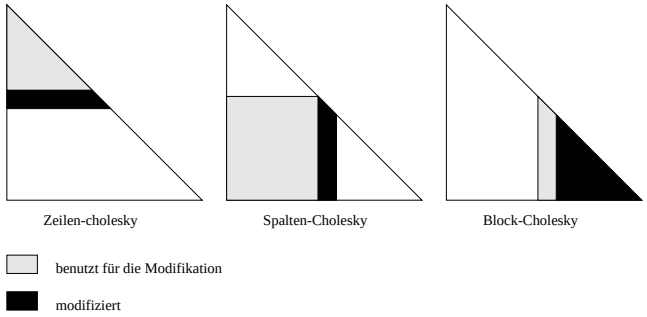


Abbildung 5.3: *Drei Formen der seriellen Cholesky-Faktorisierung (vgl. [80])*

**Zeilen-Cholesky** Wählt man $i$ als äußeren Schleifenindex, wird die Matrix $L$ zeilenweise berechnet. Die inneren Schleifen lösen ein Dreieckssystem für jede neue Zeile mit den bis dahin aufgelösten Zeilen.

**Spalten-Cholesky** Wählt man $j$ als äußeren Schleifenindex, wird die Matrix $\mathbf{L}$ spaltenweise aufgebaut. Die inneren Schleifen berechnen ein Matrix-Vektor-Produkt, das den Einfluß der bis dahin berechneten Spalten auf die gerade aufzulösende Spalte darstellt.

**Block-Cholesky** Ist $k$ der äußere Schleifenindex, wird die Matrix $\mathbf{L}$ spaltenweise berechnet. Die inneren Schleifen benutzten dabei die aktuelle Spalte für einen *rank-1 Update* der verbleibenden, erst teilweise reduzierten Submatrix.

Jeder Typ des Choleskyalgorithmus hat ein anderes Speicherzugriffsschema und ein anderes Vorgehen beim Eliminieren der Matrix $\mathbf{K}$ (siehe Abbildung 5.3). Jedes der Verfahren hat seine spezifischen Vor- und Nachteile. Für die Cholesky-Elimination von dünnbesetzten Gleichungssystemen werden die zeilenorientierten Algorithmen kaum genutzt, weil es sehr schwer ist, eine kompakte zeilenweise Speicherung zu erzeugen, die sich effizient in der Faktorisierungsphase nutzen läßt. Für Vektorrechner oder Parallelrechner ist dieser Typ ebenfalls nicht interessant, weil es sehr schwierig ist, diese dünnbesetzten Matrixzeilen mit ihrem komplizierten Zugriffsmuster effizient zu verarbeiten (Zweifachindizierung für jedes Matrixelement im Gegensatz zur Einfachindizierung beim Zugriff innerhalb einer Spalte). Im folgenden sollen deshalb nur der spalten- und der blockorientierte Ansatz diskutiert werden.
Beim Spalten-Cholesky unterscheidet man zwei Grundaufgaben:

$cmod(j,k)$ Modifikation der Spalte $j$ durch die Spalte $k, k < j$

$cdiv(j,\alpha)$ Division der Spalte $j$ durch einen Skalar $\alpha$.

Die Spalten $j$ der Matrix $\mathbf{K}$ bleiben unverändert, bis der Index der äußeren Schleife den entsprechenden Wert annimmt. Ab diesem Moment wird in dem Algorithmus

die Spalte $j$ für jeden Nichtnulleintrag entsprechend für alle Spalten $k < j$ aktualisiert, für die es einen Eintrag in $L$ gibt. Sind alle Spalten mit der Spalte $j$ modifiziert, wird der Diagonaleintrag in der Spalte $j$ berechnet, um die komplett modifizierte Spalte $j$ zu skalieren. Der spaltenorientierte Cholesky-Algorithmus orientiert sich damit immer nach links. Diese Operation wird von verschiedenen Autoren als *fan-in* bezeichnet, weil der Einfluß aller links liegenden, schon eliminierten Spalten auf die aktuelle Spalte durch Zusammenfassen berechnet wird. Der spaltenorientierte Cholesky-Algorithmus ist der am weitesten verbreitete Algorithmus in kommerziellen Produkten zur Lösung dünnbesetzter linearer Gleichungssysteme.

Beim blockorientierten Cholesky wird sofort nach der Auflösung einer Spalte $k$ deren Einfluß auf alle weiteren, verbleibenden Matrixspalten berechnet. Dieser Algorithmus orientiert sich somit immer nach rechts, weil die beeinflußten Spalten rechts von der aktuellen liegen. Diese Operation wird im Gegensatz zum spaltenorientierten Choleky als *fan-out* bezeichnet.

Neben den bisher dargestellten grundlegenden Techniken zur Choleskyfaktorisierung existieren eine Reihe von Varianten und gemischten Methoden, die sich vor allem in der Berechnung von Zwischenergebnissen unterscheiden. Frontlösungsmethoden oder deren Verallgemeinerung, die Multifrontmethoden, sind die wichtigsten Vertreter.

Multifrontmethoden organisieren die Berechnung in einer sequentiellen Folge von erweiterten Additions- und Faktor-Updateoperationen. Die Abfolge wird vom Eliminationsbaum gesteuert. Die aktiven Matrixblöcke sind dicht gespeichert. Es kommen deshalb im wesentlichen Level-3-BLAS Routinen zum Einsatz. Bis zu 90% der Verarbeitungszeit werden somit in den effektivsten Matrixoperationen benötigt. Damit ergeben sich für die Verarbeitung auf modernen RISC-Prozessoren Vorteile.

Der Hauptteil der Matrix kann bei diesen Verfahren auf einem Massenspeicher zwischengespeichert werden. Um den Zugriff auf die externen Daten nicht zum Engpaß werden zu lassen, muß der Zugriff auf die inaktiven Matrixblöcke minimiert werden. Das war die ursprüngliche Motivation zur Entwicklung von Multifrontmethoden für die Abarbeitung auf Rechnern mit begrenztem Hauptspeicher.

Zwei Begriffe sind für die weiteren Erläuterungen interessant. Ein Superknoten (engl. *supernode, inode, identity node*) ist eine Menge von aufeinanderfolgenden Matrixspalten, die das gleiche Belegungsmuster besitzen. Spalten eines Superknoten können für die Berechung und die Speicherung als eine Einheit aufgefaßt werden. Damit kann die Behandlung eines Superknoten mit Matrix-Matrix- oder Matrix-Vektor-Operationen formuliert werden. Diese speziellen Operationen reduzieren bei Ausnutzung von Cache oder Vektorregistern die Speicherzugriffszeit. Sie tragen deshalb auf RISC-Architekturen zusätzlich zur Steigerung der Rechenleistung bei.

Der Eliminationsbaum $T(\mathbf{K})$, der zum Choleskyfaktor $\mathbf{L}$ einer gegebenen Matrix $\mathbf{K}$ der Ordnung $n$ gehört, besitzt $n$ Knoten. Eine Kante existiert zwischen den Knoten $i$ und $j$ genau dann, wenn $i > j$ und $i = p(j)$. Der Knoten $i$ ist in diesem Fall Vater des Knoten $j$ und der Knoten $j$ ist Kind von Knoten $i$. Der Eliminationsbaum beschreibt die Datenabhängigkeiten innerhalb des Faktorisierungsprozesses und hat deshalb sowohl Bedeutung für die Multifrontmethoden als auch für die parallele Implementierung.

### 5.1.4 Dreieckslösung

Die Struktur des Vorwärtseinsetzen und der Rückwärtselimination wird bestimmt durch das Belegungsmuster des Choleskyfaktors und die Abhängigkeiten, die im Eliminationsbaum beschrieben sind. Die Dreieckslösung benötigt wesentlich weniger Fließkommaarithmetik als die Faktorisierung. Auf seriellen Rechnern wird sie daher in einem Bruchteil der für die Faktorisierung benötigten Zeit durchgeführt.



## 5.2 Direkte, parallele Lösungsverfahren

Die Aufgabe bei der Parallelisierung der Gleichungslösung besteht zunächst in der Verteilung des Berechnungsaufwandes auf Teilprobleme, die den benutzten Prozessoren zugeordnet werden Dabei hängt es vom verwendeten Rechnertyp ab, welche Konfiguration und welche Verteilungsstrategie eine effiziente Berechnung erlaubt. Insbesondere das Verhältnis von Rechenleistung zur Kommunikationsleistung bestimmt, ob die Unterteilung in wenige, dafür sehr große Teilaufgaben oder ob die Unterteilung in sehr viele, aber sehr kleine Teilaufgaben den größten Gewinn an Rechenzeit erbringt. Der Begriff der Granularität wird in diesem Zusammenhang gebraucht, um die Größe der Teilaufgaben in einer parallelen Applikation zu charakterisieren. Folgende Stufen der Granularität lassen sich unterscheiden ([92]):

**Feine Granularität** Jede einzelne arithmetische Fließkomma-Operation wird als Aufgabe verstanden.

**Mittlere Granularität** Jede *cmod*- oder *cdiv*-Operation wird als Teilaufgabe formuliert.

**Grobe Granularität** Die unabhängigen Eliminationsteilbäume werden jeweils genau einer Teilaufgabe zugeordnet.

Die parallele Gleichungslösung wird in der gleichen Art und Weise strukturiert, wie die serielle Lösungstechnik. Die vier Teilaufgaben sind analog zu lösen - die Matrix ist geeignet zu sortieren, um den *fill-in* zu minimieren, die Datenstruktur der aufgelösten unteren Dreiecksmatrix ist mit der symbolischen Faktorisierung zu initialisieren, die numerische Faktorisierung und die abschließende Dreieckslösung ist durchzuführen. Zusätzlich entsteht aber eine neue Teilaufgabe - die Teilprobleme sind auf die Prozessoren abzubilden. Dabei soll zum einen die resultierende Rechenlast der einzelnen Prozessoren ausgewogen sein, zum anderen ist die Abbildung so vorzunehmen, daß der zusätzliche Aufwand für Kommunikation und Synchronisation der Prozesse minimal wird. Auf MIMD-Rechnern ist dafür ein statische Zuordnung der Teilprobleme auf das Prozessornetzwerk zu finden.

### 5.2.1 Sortieren

In parallelen Lösern sind zwei unterschiedliche Teilprobleme mit dem Sortierproblem verknüpft:

- Bestimme eine geeignete Reihenfolge der Gleichungen, um eine effiziente Abarbeitung der aufeinanderfolgenden Faktorisierungsschritte zu sichern.
- Berechne die Sortierung selbst parallel.

Auf seriellen Maschinen besteht die wichtigste Aufgabe, durch Sortieren der Gleichungen sowohl den benötigten Speicherplatz als auch den Rechenaufwand zu minimieren. Aus Erfahrung und Intuition kann man beide Ziele auf ein Ziel reduzieren - Minimierung des *fill-in*. Allerdings führt die Minimierung des *fill-in* allein nicht auf eine für die Parallelisierung geeignete Sortierung.

Am Beispiel von tridiagonalen Gleichungssystemen soll dieser Sachverhalt erläutert werden. Die Sortierung, die die Tridiagonalstruktur bewahrt, ist die lexikographische oder auch natürliche Reihenfolge der Gleichungen. Mit der natürlichen Reihenfolge gibt es bei der Faktorisierung keinen *fill-in*. Damit wird sowohl der benutzte Speicherplatz als auch der benötigte Rechenaufwand minimal. Trotzdem ist die natürliche Reihenfolge die schlechteste Wahl für die Parallelisierung. Diese Sortierung führt auf einen sehr schlecht balancierten Eliminationsbaum. Dieser Eliminationsbaum

besteht aus einer einfachen Kette. Somit ist die Ausnutzung von grober Granularität durch Abspalten von Teilbäumen des Eliminationsgraphen nicht möglich. Die Elimination jeder Gleichungsspalte $i > 0$ hängt genau von der Gleichungsspalte $i-1$ ab. Damit ist auch eine mittlere Granularität auf der Ebene der Spaltenelimination nicht nutzbar. Andererseits ist bekannt, daß die sogenannte Schachbrettsortierung, obwohl sie den numerischen Aufwand erhöht, einen höheren Grad an Parallelität erzielt. Die Schachbrettsortierung entsteht u.a. durch die Anwendung des *nested dissection algorithm*. Die resultierende Höhe des Eliminationsbaums beträgt $\frac{log_2 n}{n-1}$. Der absolute numerische Aufwand steigt damit etwa um das Zwei- bis Dreifache. Der parallele Aufwand ist mit Ordnung $O(log\ n)$ ideal. Im Vergleich dazu beträgt der ideale Aufwand für die Elimination bei natürlicher Reihenfolge $O(n)$.

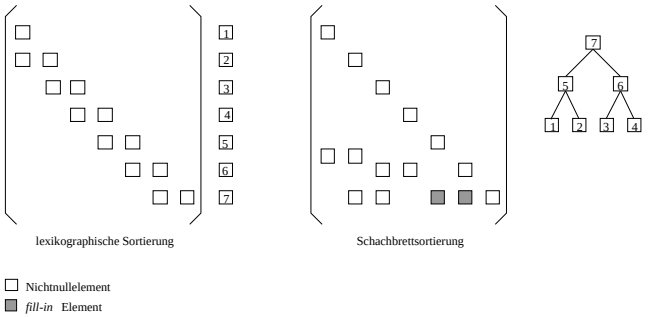


Abbildung 5.4: *Struktur des Choleskyfaktors und des Eliminationsbaumes von Tridiagonalmatrizen in lexikographischer bzw. Schachbrettsortierung (o.a. even-odd, red-black, nested dissection)*

Das Beispiel zeigt, wie ungeeignet die Minimierung des *fill-in* als Kriterium für eine für die parallele Berechnung geeignete Sortierung sein kann. Andererseits gibt es bisher keine Systematik zur vergleichenden Beurteilung der Güte einer Sortierung für die parallele Berechnung. Zur Zeit benutzte Heuristiken verwenden die Eigenschaften des Eliminationsbaumes. Dabei werden kurze Bäume schmaleren Bäumen vorgezogen. Diesen Entscheidung liegen aber eher intuitive als exakt beweisbare Kriterien zugrunde. Nicht zuletzt muß darauf hingewiesen werden, daß das Sortieren einer Matrix, so daß ein möglichst flacher Eliminationsbaum entsteht, genau wie die Aufgabe zur Minimierung des *fill-in*, ein hartes kombinatorisches Problem ist.

#### 5.2.1.1 Serielles Sortieren

Im folgenden sollen Heuristiken für die Bestimmung einer geeigneten Zerlegung mit Hilfe von Baumstrukturierung (engl. *Tree restructuring for parallel elimination*) und Graphteilung (engl. *Nested dissection and graph partitioning*) diskutiert werden.
Eine Herangehensweise, um geringen *fill-in* als auch eine geeignete Zerlegung für die parallele Berechnung zu erhalten, ist es, beide Aufgabe separat zu behandeln. Zunächst wird mit einem Standardverfahren, wie *minimum degree algorithm* eine Sortierung mit minimalem *fill-in* bestimmt und danach wird aufbauend auf dieser Sortierung eine äquivalente Sortierung erzeugt, die für die parallele Faktorisierung geeignet ist. Dabei soll der *fill-in* möglichst gleich sein, aber der Eliminationsbaum verändert werden. Die Effizienz dieser Algorithmen zum *tree restructuring for parallel elimination* hängt davon ab, ob für die Parallelverarbeitung geeignete Sortierungen in der Menge der möglichen Sortierungen enthalten sind. Das Beispiel oben mit der Tridiagonalmatrix zeigt, daß dieser Fall nicht notwendig eintreten muß. Es gibt bis jetzt keine exakten Kriterien, um zu bestimmen, ob für reale Problemstellungen eine geeignete Sortierung existiert.



Implementierungen dieses Ansatzes benutzen eine initiale Sortierung zur Minimierung des *fill-in*, einen Mechanismus zur Umstrukturierung des Eliminationsbaumes und eine Bewertungsfunktion zur Einschätzung, ob eine gegebene Permutation tatsächlich die resultierende parallele Eliminationszeit verringert. Als Mechanismen zur Umstrukturierung werden verschiedene Verfahren genutzt, die alle relativ kleine nderungen der initial berechneten Sortierung ergeben. Die Komplexität der besten Verfahren ist linear in der Anzahl der Supernodes des Choleskyfaktors.
Die Effektivität des *nested dissection algorithm* für die Minimierung von *fill-in* bzw. Maximierung der Parallelität hängt davon ab, ob es gelingt, genügend viele und genügend kleine Separatoren zu finden. Das grundlegende Schema besteht aus drei Schritten:

- Nutze ein heuristisches Verfahren, um einen kleinen Kantenseparator zu bestimmen, oder genauer - bestimme eine möglichst kleine Menge von Kanten im Graphen der Matrix, deren Streichen aus dem Graphen zwei unabhängige Teilgraphen mit annähernd gleicher Knotenzahl erzeugt.
- Leite aus dem Kantenseparator einen Knotenseparator ab.
- Numeriere die Teilgraphen nacheinander und den Separator zuletzt.

Das Verfahren wird rekursiv auf die entstehenden Teilgraphen angewandt, bis die gewünschte Anzahl der Teilprobleme erreicht ist.
George und Liu benutzen in [78] und [79] die Stufenstruktur eines Graphen zur Konstruktion geeigneter Separatoren. Ausgehend von einem geeigneten Startknoten der Stufe 0 werden seine direkten Nachbarknoten der nächsten Stufe zugeordnet. Alle diese Knoten bestimmen mit ihren noch nicht bezeichneten Nachbarknoten die Knoten der nachfolgenden Stufe. Das Verfahren wird fortgesetzt, bis alle Knoten genau einer Stufe zugeordnet sind. Eine Stufe in der Mitte wird als Separator ausgewählt. Der Vorteil der Methode ist, daß direkt eine Knotenseparator ermittelt wird und damit der Schritt 1 des allgemeinen Algorithmus entfällt. Der Nachteil der Methode ist der Mangel an geeigneten Kriterien für die Auswahl des Startknotens und die fehlende Möglichkeit, diesen Algorithmus selbst zu parallelisieren. Die Stufenstrukturen müssen streng sequentiell aufgebaut werden und insofern ist die Bestimmung eines Separators ein sequentieller Prozeß. Parallelität kann erst in einem gewissen Grad ausgenutzt werden, wenn die Separatoren der entstehenden Teilgebiete bestimmt werden. Desweiteren liefert dieser Algorithmus für die Minimierung des *fill-in* schlechtere Ergebnisse als die Verfahren zur Optimierung des Eliminationsbaumes.
Kernighan und Lin [122] entwickelten ein heuristisches Verfahren, um eine bestehende Sortierung zu verbessern und kleine Kantenseparatoren zu generieren. Sie definieren im Zusammenhang mit einem Kantenseparator zwei Knotenmengen wie folgt. Es seien $P_1$ und $P_2$ die beiden Knotenmengen, in die der Kantenseparator den Graphen teilt. Es sei $V_1$ die Menge der Knoten aus $P_1$, die mit mindestens einem Knoten im Kantenseparator verbunden ist. $V_2$ sei analog zu $P_2$ definiert. Entsprechend ist $V = V_1 \cup V_2$ der gemeinsame Kantenseparator. $V_1$ und $V_2$ bezeichnen den sogenannten erweiterten Separator. Der Algorithmus verbessert jetzt durch eine heuristisch begründete Auswahl von Knoten aus dem zuvor bestimmten erweiterten Separator schrittweise günstigere gemeinsame Kantenseparatoren. Dazu werden aus dem erweiterten Separator Knotenpaare ermittlet, die bei Austausch zwischen

den beiden Knotenmengen den Kantenseparator minimieren. Das Knotenpaar mit der größten Verbesserung wird ausgewählt. Dann sind natürlich die entsprechenden Knotenmengen zu aktualisieren. Das Verfahren wird fortgesetzt, bis kein Austausch mehr zu einer Verkürzung des Kantenseparators führt.

Auch dieser Algorithmus wirkt im wesentlichen lokal, kann also durchaus gute Ergebnisse bei der Verbesserung einer geeigneten Startlösung liefern. Interessant wurde dieses Verfahren mit den Verbesserungen des Laufzeitverhaltens durch Fiduccia und Mattheyses [63], die allerdings kein parallelisiertes Verfahren entwickelten. Suaris und Kedem [197] zeigten, wie der Algorithmus zur Konstruktion von Kantenseparatoren für vier Teilgebiete erweitert werden kann. Von Hendrickson und Leland [95] stammt ein Vorschlag, das Verfahren zur Partitionierung in eine beliebige Anzahl von Teilgebieten zu nutzen.

Leiserson und Lewis [134] verbesserten das Originalverfahren von Kernighan und Lin zur Konstruktion von Kantenseparatoren und zeigten, daß mit ihrem Verfahren der beste Kantenseparator aus dem erweiterten Separator identifiziert werden kann. Als Maßzahl für die Güte der Lösung verwendeten sie die Höhe des Eliminationsbaums für die resultierende Zerlegung.

Bei dem von Pothen, Simon und Liou [168] vorgeschlagenen Verfahren werden Eigenschaften einer aus der Inzidenzmatrix eines Graphen abgeleiteten Laplace-Matrix ausgenutzt. Bei sehr großen Graphen ist der algorithmische Aufwand für die Eigenwertbestimmung erheblich. Man kann deshalb versuchen, den Ausgangsgraphen in einen äquivalenten Graphen mit weniger Knoten zu überführen und die Partitionierung mit diesem Supergraphen durchzuführen.

Das Verfahren wird wegen seiner algorithmischen Umsetzung auch Spektral-(Bi-) Sektion genannt. Es ist in einer parallelisierten und hochoptimierten Implementierung z.B. in der wissenschaftlichen Bibliothek der CM-5 verfügbar [200] (vgl. auch Abschnitt 3.1.2.3.1).

Liu [138] schlägt eine Kombination aus dem *minimum degree* und dem *nested dissection* Algorithmus vor. Damit wird innerhalb eines Partitionierungsverfahrens die Strategie der Erzeugung einer neuen Lösung systematisch gewechselt. Mit *minimum degree* wird zunächst eine gute Startlösung ermittelt, die anschließend mit *nested dissection* und heuristisch begründeten Strategien verbessert wird. Dabei benutzt Liu Techniken zur Baumrotation, um aus einer guten Startlösung für einen Kantenseparator einen äquivalenten Kantenseparator abzuleiten, der im Graph eine andere Orientierung hat.

#### 5.2.1.2 Paralleles Sortieren

Die Parallelisierung der Berechnung eines geeigneten Permutationsvektors ist eine Aufgabe, die aus der Motivation erwächst, parallele Rechner einzusetzen, wenn das zu lösende Gleichungssystem sehr groß ist und die Anforderungen, z.B. an den zur Verfügung stehenden Speicherplatz mit einem seriellen Rechner schwer realisierbar sind. Für diese Aufgabenklasse kann bereits die Bestimmung einer geeigneten Permutation ein Speicherplatzproblem sein. Somit besteht Bedarf, die Sortierung selbst bereits auf dem Parallelrechner durchzuführen.

### 5.2.2 Abbilden

Die Bewertung der Güte einer gewählten Aufgabenverteilung für einen MIMD-Rechner erfordert die Abschätzung des Kommunikationsaufwandes. Dieser Kommunikationsaufwand, der Overhead in der parallelen Berechung bedeutet muß im Verhältnis zum Rechenaufwand für die Teilaufgaben stehen. Es ist abzuwägen, inwieweit die parallele Datenhaltung und -erzeugung vorteilhaft gegenüber dem



Datenaustausch ist. Der dynamische Austausch von größeren Datenmengen (Gleichungsspalten oder -blöcke) ist auf den heute zur Verfügung stehenden Rechnern nicht sinnvoll. Die Abbildung erfolgt deshalb immer statisch, das Abbildungsproblem ist identisch mit dem Datenverteilungsproblem.

Der Eliminationsbaum enthält Informationen über die Abhängigkeiten innerhalb des Eliminationsschrittes und signalisieren damit notwendigen Kommunikationsbedarf. Darüberhinaus liefert er Informationen über die Reihenfolge der Eliminationsschritte. Leider kann ein Eliminationsbaum erst nach erfolgter symbolischer Faktorisierung aufgestellt werden. Es ist deshalb erforderlich, die symbolische Faktorisierung selbst effizient, evtl. parallelisiert durchzuführen und dabei ggf. auf eine gute Lastverteilung zu verzichten. Bei entsprechenden Problemgrößen wird es aus Speicherplatzgründen nicht möglich sein, den gesamten Eliminationsbaum in einem sequentiellen Prozeß aufzustellen.

Gilbert und Zmijewski [82] schlagen z.B. einen sehr schnellen Algorithmus vor, der den Eliminationsbaum parallelisiert bestimmt. Jeder Prozeß benutzt einen lokalen Anteil des Adjazenzgraphen der Matrix $\mathbf{K}$ und berechnet daraus den lokalen Eliminationsbaum. Anschließend werden die lokalen Informationen in komprimierter Form (engl. *subscript supression*) im Netzwerk zusammengefaßt und zum globalen Eliminationsbaum aufgebaut.

Nachdem der Eliminationsbaum berechnet wurde, werden die Matrixspalten auf die Prozesse verteilt. Ziele sind dabei Lastgleichgewicht und wenig Kommunikationsbedarf. Diese Ziele sind bei irregulären Netzen nicht gemeinsam zu verwirklichen. Eine der ersten Ideen, die verfolgt wurden, dieses Problem zu lösen, verteilte die Elemente der Stufenstruktur des Eliminationsbaumes in einem wrap-around Schema auf die Prozessoren. Das führte zwar zu einer gleichmäßigen Lastverteilung, aber der Kommunikationsaufwand ist oft unnötig hoch.

Dagegen liefert die Abbildung von Teilbäumen auf Prozessoren (engl. *subtree-to-subcube*) sowohl gute Lastbalance als auch gute Kommunikationseigenschaften [92]. Am besten sind aber reguläre Strukturen für diese Abbildungsverfahren geeignet. George et. al. [78] zeigten, daß mit diesem Herangehen für *fan-out* Algorithmen auf $k \times k$ Gittern die Kommunikationsrate $O(Pk^2)$ ist. Darüberhinaus haben Gao und Parlett [74] bewiesen, daß für die einzelnen Prozessoren der Kommunikationsaufwand $O(k^2)$ und somit auch balanciert ist.

### 5.2.3 Symbolische Faktorisierung

Für jede Teilaufgabe ist die Berechnung der zugehörigen Strukturen der eliminierten Dreiecksmatrixzeilen erforderlich. Dabei muß jeder Prozeß nur über die Gleichungszeilen oder -spalten informiert sein, für die er verantwortlich ist. Es ist daher ein verteilter Algorithmus erforderlich. Der serielle Ablauf selbst ist bereits sehr effizient, hat aber sehr wenige parallelisierbare Anteile.

Wir haben bei der Erläuterung der symbolischen Faktorisierung gesehen, daß die Berechnung der Struktur der Spalte $k$ von der Struktur der Spalte $j$ genau dann abhängig ist, wenn $k > j$ und wenn $k$ ein Kind von $j$ im Eliminationsbaum der Matrix $\mathbf{K}$ ist. Somit ist auch bei der symbolischen Faktorisierung die Parallelisierbarkeit nur in dem Maße gegeben, wie es gelingt, große Eliminationsteilbäume zu identifizieren.

Ng [156] beschreibt die Parallelisierung eines seriellen Algorithmus für die symbolische Faktorisierung und seine effiziente Implementierung unter Ausnutzung der *supernode*- Struktur.

### 5.2.4 Numerische Faktorisierung

Die numerische Faktorisierung ist, wie im seriellen Fall, die rechenaufwendigste Teilaufgabe. Es ist nicht verwunderlich, daß deshalb die numerische Faktorisierung intensiver erforscht wurde, als die anderen Teilschritte der Cholesky-Elimination. Zudem ist die Realisierung von Faktorisierungsalgorithmen auf MIMD-Rechnern wesentlich komplizierter ist, als die Parallelisierung von Algorithmen für dichtbesetzte Matrizen. Die komplexen Datenstrukturen und deren Handhabung im Faktorisierungsschritt, stellen mehr Probleme und Schwierigkeiten auf. Zudem muß der Vorteil der dünnbesetzten Algorithmen in Hinblick auf die Speicherausnutzung und die numerische Komplexität auch für die parallele Variante erhalten bleiben.
Parallele Algorithmen orientieren sich an einer initialen Datenverteilung. Damit die Interprozessorkommunikation minimal wird, muß solange wie möglich von der Lokalität der Daten Gebrauch gemacht werden. Die in einem Speicher vorhandenen Daten sollten also auch durch lokale Operationen verarbeitet werden und möglichst wenig an anderer Stelle benutzt werden. Die verteilten *fan-out-*, *fan-in-* und Multifront-Algorithmen sind typische Vertreter dieser Art paralleler Algorithmen. Alle drei Formen des Algorithmus nutzen die spaltenweise Verteilung der Matrix auf die Prozesse. Die Aufgaben werden entsprechend der äußeren Schleife des zugehörigen seriellen Algorithmus verteilt. Die Unterschiede der drei Ansätze resultieren aus den verschiedenen Formulierungen der zugehörigen seriellen Implementation.

#### Parallele fan-out Cholesky-Faktorisierung

Als Variante ist der blockorientierte *fan-out*-Algorithmus die erste realisierte parallele Cholesky-Faktorisierung für MIMD-Rechner [77]. Die Spalten sind auf jeweils genau einen Prozessor verteilt. Zur Identifikation der lokalen und globalen Spalten dient eine Abbildungsvorschrift. Wenn man die k-te Aufgabe in der äußeren Schleife mit $Tsub(k)$ bezeichnet, dann realisiert der Algorithmus auf den $p$ Prozessoren

$$Tsub(k) := \{cdiv(k)\} \bigcup \{cmod(j,k) | j \in Struct(\mathbf{L}_{*k})\} \tag{5.18}$$

Zunächst werden also alle $\mathbf{L}_{*k}$ mit der *cdiv*-Operation aktualisiert und danach werden alle Spalten mit der *cmod*-Operation modifiziert, die im Eliminationsbaum identifiziert sind. Diesen Algorithmus bezeichnet man als datengesteuerten Algorithmus, da nachdem die Elimination einer Spalte erfolgt ist, diese Spalte im Prozessornetz verschickt wird. Die äußere Schleife des *fan-out*-Algorithmus muß also ständig überprüfen, ob die eintreffenden Nachrichten relevant für die Modifikation der lokalen Spalten sind

$$\{cmod(j,k) | j \in Struct(\mathbf{L}_{*k}) \subset mycols(p)\} \tag{5.19}$$

Dabei bezeichnet $mycols(p)$ die Menge der lokalen Spalten. Sind für eine Spalte $j$ alle erforderlichen Modifikationen durchgeführt, dann kann diese Spalte selbst eliminiert (*cdiv(j)*-Operation) und an die anderen Prozesse verschickt werden. Die symbolische Beschreibung des Algorithmus ist in 5.5 angegeben.
Der *fan-out*-Algorithmus benötigt mehr Interprozessor-Kommunikation als die beiden anderen parallelen Verfahren. Eine Reihe von Verbesserungen des Algorithmus reduzieren diese Kommunikationsabforderungen ([78], [225], [154]). Schlanke Eliminationsbäume mit wenigen Abhängigkeiten unter den Teilbäumen führen nicht zu einer Reduktion der Anzahl der Datenaustausche und auch nicht zu einer Beschränkung des Datenumfanges. Der Algorithmus kann keinen Nutzen aus diesen optimalen Eigenschaften ziehen. Ein weiteres Problem ist die Struktur der *cmod*-Operation, in der Informationen über das Belegungsmuster der lokalen Spalte $k$ als auch der Spalte $j$ benötigt werden. Sowohl $Struct(\mathbf{L}_{*j})$ als auch $Struct(\mathbf{L}_{*k})$



```
für   j ∈ mycols(p)
  falls   j   Vater in   T(K)
    cdiv(j)
    sende   L_*j   zu den Prozessoren in   procs(L_*j)
    mycols(p) := mycols(p) − {p}
solange   mycols(p) ≠ 0
  empfange beliebige Spalte   L_*k
  für   j ∈ Struct(L_*k) ⊂ mycols(p)
    cmod(j,k)
    falls Spalte   j   fertig mit allen   cmod
      cdiv(j)
      sende   L_*j   zu den Prozessoren in   procs(L_*j)
      mycols(p) := mycols(p) − {p}
```

Abbildung 5.5: *Parallele fan-out Cholesky-Faktorisierung*

müssen traversiert werden, um gemeinsame Einträge zu identifizieren. Diese Operation hat deshalb eine sehr schlechte serielle Effizienz. Zudem wird es erforderlich, nicht nur die aufgelöste Spalte $j$ an alle Prozessoren zu verschicken, sondern auch die Struktur der Spalte zu beschreiben. Die Datenmenge im Austausch verdoppelt sich dadurch. Selbst wenn man die Technik der *subscript supression* anwendet, und damit eine deutlich geringere Laufzeit für diese eine Operation erreicht, wird die parallele Effizienz des Gesamtalgorithmus nicht wesentlich verbessert. In [226] wird der Einfluß verschiedener Partitionierungs- und Abbildungsstrategien auf die Effizienz des Verfahrens untersucht. Es wird vorgeschlagen, statt des *fan-out*-Algorithmus eine parallele Multifront-Cholesky-Faktorisierung zu benutzen, weil nur dieses Verfahren lokale, auf dichtbesetzte Submatrizen arbeitende Rechenkerne besitzt und die Abbildung *subtree-to-subcube* effizient ausnutzen kann [92].

#### Parallele fan-in Cholesky-Faktorisierung

Wie beim blockorientierten fan-out-Algorithmus ist auch bei dieser Variante jeder Prozessor für genau eine Spalte des Gleichungssystems verantwortlich. Auch hier kann die *cdiv*-Operation erst durchgeführt werden, wenn die Modifikationen $cmod(j,k)$ für alle $k \in Struct(\mathbf{L}_{j*})$ erfolgt sind. Der *fan-in*-Algorithmus ([139], [77], [7], [131]) ist aber anforderungsgesteuert. Die angeforderten Daten treffen beim Empfängerprozeß zusammengefaßt ein. Jeder Absender bestimmt diese modifizierten Spalten lokal, bevor sie versandt werden, um eine bestimmte Spalte zu eliminieren (siehe Abbildung 5.6).
Wird im *fan-in*-Algorithmus ein Ast des Eliminationsbaumes bearbeitet, dann erfolgt in diesem Abschnitt keine Kommunikation. Der Algorithmus hat in Abhängigkeit von der Abbildung des Eliminationsbaumes ideal parallele Abschnitte. Das führt zu Varianten des *fan-in*-Algorithmus, die berücksichtigen, ob eine Spalte mit hoher Priorität zu bearbeiten ist, wenn sie für den späteren Austausch benötigt wird oder mit niedriger Priorität zu bearbeiten ist, wenn das Ergebnis tatsächlich nur lokal benötigt wird. Dabei können die lokalen Aufgaben mit Standardalgebraroutinen (*saxpy*) bearbeitet werden, da lokale Spalten in einem Ast des Eliminationsbaumes mit dem gleichen Belegungsmuster arbeiten. Gleichwohl hat die Reduktion der Rechenzeit durch Ausnutzung der Superknoten keinen allzu große Bedeutung. Die

```
für   j = 1   bis   n
  falls   j ∈   mycols(p)   oder   Struct(L_{j*}) ⊂ mycols(p) ≠ 0
    u := 0
    für   k ∈ Struct(L_{j*}) ⊂ mycols(p)
      u := u + u(j,k)
    falls   map[j] ≠ 0
      sende  u an Prozessor   q = map[j]
    sonst
      aktualisiere Update-Faktor von j mit u
      solange ein Update-Faktor von j fehlt
        empfange einen weiteren Update-Faktor von j in u
        aktualisiere den Update-Faktor von j mit u
      cdiv(j)
```

Abbildung 5.6: *Parallele fan-in Cholesky-Faktorisierung*

parallele Organisation und die initiale Datenverteilung führt in der Regel nicht zu ideal verteilten Superknoten.

#### Parallele Multifront-Cholesky-Faktorisierung

Auch bei Multifrontalgorithmen [56], [140], [169], [199], [81], [171] sind die Matrixspalten eindeutig den Prozessoren durch die Vorschrift $map[k]$ zugeordnet. Betrachten wir einen Superknoten K. Es sei $map[k]$ die Menge der Prozessoren, die mindestens eine Spalte des Superknotens $K$ oder einen Nachfahren der Spalten von $K$ im Eliminationsbaum speichern. Die wesentliche Eigenschaft des Algorithmus ist die Verteilung aller Spalten der $K$-Frontmatrix auf die Prozessoren von $map[K]$. Sowohl die faktorisierten Spalten als auch die zusammengefaßten Spalten für die Modifikation werden unter den Prozessoren $map[K]$ verteilt.
Alle Prozessoren von $map[K]$ arbeiten zusammen, um die Faktorisierung der Untermatrix $Tsub(K)$ durchzuführen. Dabei kann der Faktorisierungsalgorithmus für dichbesetzte Matrizen benutzt werden, da die Spalten der Fronmatrix das gleiche Belegungsmuster besitzen. Haben die Prozessoren $p = map[k] \in map[K]$ die Faktorisierung der Spalte $k \in K$ durchgeführt, dann wird $\mathbf{L}_{*k}$ an die anderen Prozessoren aus $map[K]$ verschickt. Diese benutzen dann $\mathbf{L}_{*k}$ zur Modifikation jeder Spalte der lokalen Frontmatrizen. Dieser Schritt ist vergleichbar mit dem *fan-out*-Algorithmus in Abbildung 5.6.
Bevor die Aufgabe $Tsub(K)$ bearbeitet werden kann, muß die zugehörige Frontmatrix $K$, die verteilt vorliegt, aufgebaut werden. Beiträge von verteilt vorliegenden Update-Matrizen müssen für jedes Kind von $K$ auf dem richtigen Prozessor eingearbeitet werden. Ist also eine Update-Spalte einer Update-Matrix eines Kindes auf einem anderen Prozessor zu finden als die zugehörige Spalte ihrer Vater-Frontmatrix, dann muß die zusammengefaßte Update-Spalte zu dem neuen Eigentümer geschickt werden. Hier wird sie in die entsprechende Spalte der Frontmatrix eingebaut.
Der Kommunikationsbedarf der Multifrontmethode besteht zum einen aus dem beschränkten Broadcast innerhalb eines Superknotens und den erforderlichen Punkt-zu-Punkt-Verbindungen zum Austausch der Update-Spalten. Dabei ist der reine Kommunikationsbedarf wesentlich geringer als beim *fan-out*, aber höher als beim *fan-in*. Insgesamt ist aber der zusätzliche Aufwand in der parallelen Abarbeitung geringer als beim *fan-out*. Die parallele Effizienz von Multifront- und *fan-in*-Algorithmen liegt in der gleichen Größenordnung.



Tabelle 5.1: *Komplexität und Skalierbarkeit der Cholesky-Faktorisierung dünnbesetzter Matrizen für zweidimensionale Finite-Element-Probleme in Abhängigkeit von Partitionierungs- und Abbildungsstrategie (vgl. [87])*

| | globale Abbildung | subtree-to-subcube |
|---|---|---|
| spaltenweise Partitionierung | Overhead $O(Np\,log(p))$<br>Skalierbarkeit $O((p\,log(p))^3)$ | Overhead $O(Np)$<br>Skalierbarkeit $O(p^3)$ |
| boxweise Partitionierung | Overhead $O(Np^{0.5}log(p))$<br>Skalierbarkeit $O(p^{1.5}(log(p))^3)$ | Overhead $O(Np^{0.5})$<br>Skalierbarkeit $O(p^{1.5})$ |

### 5.2.5 Dreieckslösung

Die Datenabhängigkeiten und die ungleich verteilte Arbeit zwischen den Prozessoren machen eine effiziente, parallelisierte Dreieckslösung außerordentlich schwierig. Zum einen ist der Rechenbedarf wesentlich geringer als bei dichtbesetzten Gleichungssystemen. Im Verhältnis dazu ist der Aufwand für die Kommunikation - Austausch der bis dahin berechneten Unbekannten - gleich groß. Effekte lassen sich nur bei der Ausnutzung von Superknoten-Strukturen nachweisen, obwohl die parallele Aufgabenverteilung keine gleichmäßige Superknoten Verteilung begünstigt. Zum anderen kann durch Ausnutzung des Pipeline-Effektes in den Eliminationsteilbäumen eine gewisse Beschleunigung erreicht werden. [88]

### 5.2.6 Skalierbarkeit der parallelen Lösung

Von Schreiber wird in [184] die Skalierbarkeit von verschiedenen Implementationen der Cholesky-Faktorisierung untersucht. Er stellt fest, daß die spaltenweise arbeitenden Algorithmen nicht skalierbar sind. Die Untersuchungen wurden sowohl auf Parallelrechnern mit 2D (Intel Paragon)- und 3D (Cray T3D)-Netzstruktur als auch auf *Fat-Tree* (CM-5)-Struktur durchgeführt. Ein Verbesserung kann, wenn überhaupt, nur mit blockorientierter Verteilung der Matrix erzielt werden. Schreiber stellt fest, daß es noch nicht klar ist, ob direkte Löser überhaupt für hochparallele ($p > 256$) oder massivparallele ($p > 1024$) Computer skalierbar sein können.
Kumar et. al. [87] geben für zweidimensionale Probleme die Komplexität und die Skalierbarkeit der spalten- und blockorientierten Methoden in Abhängigkeit von der Zuordnung der Teilaufgaben zu Parallelrechnerpartitionen an.
Bei der Implementation der *subtree-to-subcube* Abbildung wird eine Reduktion des Kommunikationsoverheads um $O(log\ p)$ erreicht. Die Skalierbarkeit verbessert sich dadurch um $O(log(p)^3)$. Das kann die Voraussetzung für die Praktikabilität dieser Methode für hochparallele Computer ($p > 256$) sein. Für mittelgroße Applikationen (ca. 50000 Unbekannte), u.a. aus dem Gebiet der Strukturmechanik, konnte damit erstmals auch praktisch die Skalierbarkeit auf bis zu 1024 Prozessoren nachgewiesen werden.
Übereinstimmend stellen alle Autoren fest, daß skalierbare Implementationen von iterativen Gleichungslösern wesentlich einfacher zu erhalten sind, als skalierbare Implementationen von direkten Eliminationsverfahren.

## 5.3 CG-Verfahren

### 5.3.1 Grundlagen

Für die symmetrische, positiv definite Matrix $\mathbf{K} \in R^{n \times n}$ bestimmt die nichttriviale Lösung des Gleichungssystems $\mathbf{K}v = f$ das globale Minimum des Funktionals

$$F(v) = \frac{1}{2}(r, \mathbf{K}^{-1}r). \tag{5.20}$$

$r$ ist das Residuum $r = \mathbf{K}v - f$ und $(x, y)$ ist ein inneres Produkt (Skalarprodukt), definiert mit $(x, y) = x^T\mathbf{W}y$ für jede symmetrische, positiv definite Matrix $\mathbf{W}$. $\mathbf{K}$ ist selbstadjungiert im Bezug auf das innere Produkt, d.h. $(x, \mathbf{K}y) = (\mathbf{K}x, y)$.
(5.20) ist äquivalent zu

$$F(v) = \frac{1}{2}v^T\mathbf{K}v - f^T v + \frac{1}{2}f^T\mathbf{K}^{-1}f. \tag{5.21}$$

Da $\mathbf{K}$ selbstadjungiert im Bezug auf das innere Produkt ist, gilt die Äquivalenz

$$\begin{aligned} F(v + \alpha d) - F(v) &= \frac{1}{2}(r + \alpha\mathbf{K}d, \mathbf{K}^{-1}(r + \alpha\mathbf{K}d)) - \frac{1}{2}(r, \mathbf{K}^{-1}r) \\ &= \alpha(r, d) + \frac{1}{2}\alpha^2(d, \mathbf{K}d). \end{aligned} \tag{5.22}$$

Beim Verfahren der konjugierten Gradienten (CG-Verfahren) wird das Minimum schrittweise erreicht. Jeder Schritt $k$ berechnet das Minimum von $F(v)$ bezüglich einer gewählten Suchrichtung $d_k$. Die erzeugten Suchrichtungen sind dabei konjugiert in Bezug auf $\mathbf{K}$, d.h. es gilt für jedes Paar $(i, j)$ von Suchrichtungen

$$d_i^T\mathbf{K}d_j = 0 \tag{5.23}$$

Im einzelnen geht man bei diesem Verfahren wie folgt vor. Zunächst wird eine Startlösung gewählt. Mangels besserer Alternativen ist auch die Wahl $v_0 = 0$ möglich. Mit der Näherungslösung $v_k$ aus Schritt $k$ berechnet man das zugehörige Residuum $r_k$ (oder auch den Gradienten) aus $r_k = \mathbf{K}v_k - f$. Die erste Suchrichtung wählt man sinnvoll in die Richtung des steilsten Abstieges, d.h. in die Richtung des negativen Gradienten $d_0 = -r_0$. Die Bestimmung der Suchrichtungen in den folgenden Schritten $k = 1, 2, ..$ wird weiter unten motiviert und beschrieben.
Danach wird die Schrittweite $\alpha$ berechnet. Sie bestimmt, wie weit der Unbekanntenvektor $v_k$ in Richtung $d_k$ korrigiert werden muß, um die Näherung im Schritt $k + 1$ zu bestimmen

$$v_{k+1} = v_k + \alpha d_k \tag{5.24}$$

Dazu ist das Minimum der quadratischen Form (5.20) in Bezug auf $\alpha$ zu bestimmen

$$minF(v_{k+1}) = minF(v_k + \alpha d_k). \tag{5.25}$$

Gleichzeitig wird auch die Funktion $h(\alpha) = F(v_k + \alpha d_k) - F(v_k)$ minimiert. Mit (5.22) folgt aus der Differentiation nach $\alpha$ die Bestimmungsgleichung

$$\begin{aligned} (r_k, d_k) + \alpha(d_k, \mathbf{K}d_k) &= 0 \\ \alpha &= -\frac{d_k^T r_k}{d_k^T\mathbf{K}d_k} \end{aligned} \tag{5.26}$$

Mit $\alpha$ läßt sich auch sofort der neue Residuenvektor bestimmen

$$\begin{aligned} r_{k+1} &= \mathbf{K}v_{k+1} - f \\ &= \mathbf{K}(v_k + \alpha d_k) - f \\ &= \mathbf{K}v_k - f + \alpha\mathbf{K}d_k \\ &= r_k + \alpha\mathbf{K}d_k \end{aligned} \tag{5.27}$$



d.h. $r_{k+1}$ kann rekursiv berechnet werden.
Bis hierher ist das Verfahren auch als Verfahren des steilsten Abstiegs bekannt. Berechnet man die neue Suchrichtung aus dem neuen Gradienten entsprechend dem Residuum, erfüllen die Suchrichtungen im allgemeinen nicht die Orthogonalitätsbedingung (5.23), d.h. eine bereits früher gewählte Richtung bei der Minimumsuche wird wiederholt gewählt.
Eine entsprechend zu $d_k$ bezüglich $\mathbf{K}$ konjugierte Richtung

$$d_{k+1} = -r_{k+1} + \beta d_k \tag{5.28}$$

bestimmt sich aus (5.23) durch Einsetzen mit

$$\begin{aligned} 0 &= d_{k+1}^T\mathbf{K}d_k \\ &= (-r_{k+1} + \beta d_k)^T\mathbf{K}d_k \\ &= -r_{k+1}^T\mathbf{K}d_k + \beta d_k\mathbf{K}d_k \\ \beta &= \frac{r_{k+1}^T\mathbf{K}d_k}{d_k^T\mathbf{K}d_k} \end{aligned} \tag{5.29}$$

Das Iterationsverfahren verbessert damit eine gewählte Startlösung $v_0$ mit dem zugehörigen Residuum $r_0 = \mathbf{K}v_0 - f$ und der gewählten Suchrichtung $d_0 = -r_0$ in den Schritten

$$\begin{aligned} \alpha &= -\frac{d_k^T r_k}{d_k^T\mathbf{K}d_k} \\ v_{k+1} &= v_k + \alpha d_k \\ r_{k+1} &= r_k + \alpha\mathbf{K}d_k \\ \beta &= \frac{r_{k+1}^T\mathbf{K}d_k}{d_k^T\mathbf{K}d_k} \\ &= -r_{k+1} + \beta d_k \end{aligned} \tag{5.30}$$

Das Verfahren wird wiederholt, bis ein $\alpha = 0$ gefunden wird.

### 5.3.2 Lösungsverhalten

Der Fall $d_k = 0$ muß noch näher untersucht werden. (5.26) liefert

$$\begin{aligned} r_{k+1} &= r_k + \alpha\mathbf{K}d_k \\ r_{k+1}^T d_k &= r_k^T d_k + \alpha d_k^T\mathbf{K}d_k \\ &= 0 \end{aligned} \tag{5.31}$$

d.h. Residuum und Suchrichtung von aufeinanderfolgenden Iterationsschritten stehen senkrecht aufeinander.
(5.28) auf $d_k$ angewandt und (5.31) eingesetzt liefert

$$\begin{aligned} d_k &= -r_k + \beta_{k-1}d_{k-1} \\ d_k^T r_k &= -r_k^T r_k + \beta_{k-1}d_{k-1}^T r_k \\ d_k^T r_k &= -r_k^T r_k \end{aligned} \tag{5.32}$$

und daraus sofort für $d_k = 0$ sowohl $r^T r = 0$ als auch $r_k = \mathbf{K}v_k - f = 0$. Damit gilt für $d_k = 0$, daß die gefundene Näherung $v_k = v$, d.h. die Lösung des Gleichungssystems ist.
Für den Fall, daß die Näherung $v_k$ nicht die Lösung des Gleichungssystems ist, gilt $d_k^T r_k \neq 0$, und damit sowohl $d_k^T r_k \neq 0$ als auch $\alpha \neq 0$. Unabhängig von der speziellen Wahl der $d_k$ gilt damit: Entweder führt der Iterationsprozeß auf keine Lösung oder im Schritt $m$ wird die Norm des Residuums zu Null und die Lösung $v = v_m$ ist gefunden.

### 5.3.3 Kommentar

Die spezielle Voraussetzung, daß alle $d_k$ so gewählt werden, daß sie $\mathbf{K}$-konjugiert orthogonal sind (5.23), sichert, das höchstens $n$ verschiedene, paarweise orthogonale Vektoren in $R^n$ konstruiert werden können.
Exakte Numerik vorausgesetzt, ist die Konvergenz deshalb für eine positiv definite Matrix $\mathbf{K} \in R^{n \times n}$ in $m \leq n$ Schritten garantiert. Formal ist das CG-Verfahren deshalb ein direktes Lösungsverfahren.
Praktisch kann allerdings keine exakte Numerik garantiert werden. Die später berechneten, kleiner werdenden Residuenvektoren entstehen aus Linearkombinationen von größer werdenden Vektoren. Die Vektoren $d_0, d_1, \ldots, d_k - 1$ bilden deshalb kein konjugiertes Orthogonalsystem.
Das CG-Verfahren qualifiziert sich trotzdem als iteratives Lösungsverfahren, weil eine approximierte Lösung $v_m$ mit hinreichender Genauigkeit bereits in $m \ll n$ Schritten bereitgestellt wird. Die Konvergenzgeschwindigkeit ist vom Verhältnis des größten zum kleinsten Eigenwert der Matrix $\mathbf{K}$

$$\kappa(\mathbf{K}) = \frac{\lambda_n}{\lambda_1} \tag{5.33}$$

(der Konditionszahl $\kappa$ der Matrix $K$ [185], Beweis siehe u.a. [31]) und dem Eigenwertspektrum abhängig (vgl. Bem. 9.4.12 in [90]). Die Konvergenzeigenschaften können mit vorkonditionierten Varianten des Verfahrens (s.a. Abschnitt 5.3.5) verbessert werden.
Aufgrund der Orthogonalitätseigenschaft (5.23) erhielt das von Hestenes und Stiefel 1952 zunächst unabhängig voneinander [98] und dann gemeinsam in [99] beschriebene Verfahren den Namen Methode der konjugierten Gradienten (engl. *conjugate gradient - CG*). Korrekterweise müßte es aber Methode der konjugierten Richtungen heißen. Allerdings folgt aus (5.23), daß auch die resultierenden Residuen $\mathbf{K}$-konjugiert, paarweise orthogonal sind.
Die Methode gehört zur Klasse der Krylov-Unterraum-Methoden. Nach ihrer ursprünglichen Entdeckung in den frühen fünfziger Jahren gewann sie erst mit Beginn der siebziger Jahre praktische Bedeutung. Eine ganze Familie verwandter Verfahren ist heute bekannt (vgl. Axelsson [9]). Hackbusch beschreibt in [90] wesentliche Aspekte der Implementation und Anwendung der Methode. Die Interpretation als iteratives Verfahren geht auf Reid [173] zurück.

### 5.3.4 Implementation

Mit (5.32) kann die Bestimmungsgleichung für $\alpha$ vereinfacht werden zu

$$\alpha = \frac{r_k^T r_k}{d_k^T \mathbf{K} d_k} \tag{5.34}$$

Aus (5.27) folgt für das Skalarprodukt $\alpha_k d_k^T \mathbf{K} d_k$ wegen (5.31) und (5.32)

$$\begin{aligned} \alpha_k d_k^T \mathbf{K} d_k &= d_k^T (r_{k+1} - r_k) \\ &= -d_k^T r_k \\ &= r_k^T r_k \end{aligned} \tag{5.35}$$

Aus (5.27) folgt

$$\begin{aligned} \mathbf{K} d_k &= \frac{1}{\alpha_k}(r_{k+1} - r_k) \\ r_{k+1}^T \mathbf{K} d_k &= \frac{1}{\alpha_k} r_{k+1}^T (r_{k+1} - r_k) \end{aligned}$$



und wegen $r_{k+1}^T r_k = 0$ gilt dann

$$r_{k+1}^T \mathbf{K} d_k = \frac{1}{\alpha_k} r_{k+1}^T r_{k+1} \tag{5.36}$$

Die Bestimmungsgleichung für $\beta$ läßt sich dann zu

$$\beta = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k} \tag{5.37}$$

vereinfachen. Damit und unter Verwendung eines Hilfsvektors $u$ für das Matrix-Vektor-Produkt $u = \mathbf{K} d_k$ ergibt sich folgende Implementation des CG-Verfahrens

$$\begin{aligned} \text{wähle} \quad & v_0 \\ & r_0 = \mathbf{K} v_0 - f \\ & d_0 = -r_0 \\ & \gamma_0 = r_0^T r_0 \\ \text{Iteration für} \quad & k = 1, 2, .. \\ & u = \mathbf{K} d_{k-1} \\ & \alpha = \frac{\gamma_0}{d_{k-1}^T u} \\ & v_k = v_{k-1} + \alpha d_{k-1} \\ & r_k = r_{k-1} + \alpha u \\ & \gamma_k = r_k^T r_k \quad \mapsto \quad \text{Abbruchkriterium} \quad \gamma_k < \epsilon \\ & \beta = \frac{\gamma_k}{\gamma_{k-1}} \\ & d_k = -r_k + \beta d_k \end{aligned} \tag{5.38}$$

Indizierte Größen werden in jedem Iterationsschritt aktualisiert, die nicht indizierten Größen können überschrieben werden.
Üblicherweise wird die effiziente rekursive Bestimmungsgleichung für die Aktualisierung des Residuums $r_k$ benutzt. Dabei können sich Arithmetikfehler (Rundungsfehler) zu unerwünschten Fehlern aufsummieren, die die Fehlerabschätzung zu optimistisch angeben oder die Orthogonalität der Suchrichtungen stören. Es ist daher besser, in Abhängigkeit von der Größenordnung der gewählten Abbruchschranke $\epsilon$ oder der Anzahl der Iterationsschritte eine Korrektur des Residuums vorzunehmen.
Praktische Abbruchschranken können absolut vorgegeben werden, oder aber die Reduktion des Startresiduums, z.B. mit $\epsilon = \delta^2 \sqrt{r_0^T r_0}$, vorsehen. Ist dann der Reduktionsparameter $\delta$ wesentlich größer als die Rechengenauigkeit, treten in der Regel keine Probleme auf.
Ansonsten ergeben sich zwei Möglichkeiten. Entweder wird für jeden Iterationsschritt $k = i * \sqrt{n}$ die Korrektur des aktuellen Residuums vorgenommen (zyklischer Restart des CG-Verfahrens), oder bei Erreichen der gewünschten Genauigkeit von $\gamma_k$ wird in jedem Schritt das tatsächliche Residuum berechnet.
Das CG-Verfahren erfüllt hinsichtlich Speicherplatzbedarf und arithmetischer Komplexität eines Iterationsschrittes die Anforderungen an ein Lösungsverfahren für große lineare Gleichungssysteme. Neben dem Lösungsvektor und dem Vektor der rechten Seite werden drei Vektoren benötigt. Die Matrix muß lediglich in einem Matrix-Vektor-Produkt bereitgestellt werden. Pro Iteration werden zwei Skalarprodukte und drei Rekursionsformeln, das sind insgesamt 5 Additionen und 5 Multiplikationen je Vektorelement, erforderlich. Schwerpunkt bei der Anwendung des Verfahrens ist deshalb die Minimierung der Iterationszahlen und die effiziente Implementierung des Matrix-Vektor-Produktes.

### 5.3.5 Vorkonditionierte CG-Verfahren

Formal ist das Verfahren der vorkonditionierten konjugierten Gradienten (engl. *preconditioned conjugate gradient* - PCG) identisch mit dem CG-Verfahren, angewendet auf vorkonditionierte Gradienten.
Die Vorkonditionierung ist eine Technik, die Kondition des Gleichungssystems zu verbessern (vgl. 5.4.1). Mit einer geeignet gewählten, nichtsingulären (symmetrischen) Vorkonditionierung $\mathbf{C}$ überführt man das Problem $\mathbf{K}v = f$ in das vorkonditionierte Gleichungssystem

$$\mathbf{C}^{-1}\mathbf{K}v = \mathbf{C}^{-1}f \tag{5.39}$$

Zerlegt man $\mathbf{C} = \mathbf{L}\mathbf{L}^T$, z.B. durch Choleskyfaktorisierung, kann (5.39) in ein Ersatzsystem umgeformt werden, das einen symmetrischen Operator enthält und auf das der CG-Algorithmus direkt angewendet werden kann. Das Ersatzsystem ist

$$\mathbf{L}^{-1}\mathbf{K}\mathbf{L}^{-T}\hat{v} = \mathbf{L}^{-1}f, \quad \hat{v} = \mathbf{L}^T v. \tag{5.40}$$

Das CG-Verfahren hat dann die folgende Form:

$$\begin{aligned}
\text{wähle} \quad & \hat{v}_0 \\
& \hat{r}_0 &&= \mathbf{L}^{-1}\mathbf{K}\mathbf{L}^{-T}\hat{v}_0 - \mathbf{L}^{-1}f \\
& \hat{d}_0 &&= -\hat{r}_0 \\
& \gamma_0 &&= \hat{r}_0^T\hat{r}_0 \\
\text{Iteration für} \quad & k = 1, 2, \ldots \\
& \hat{u} &&= \mathbf{L}^{-1}\mathbf{K}\mathbf{L}^{-T}\hat{d}_{k-1} \\
& \alpha &&= \frac{\gamma_0}{\hat{d}_{k-1}^T\hat{u}} \\
& \hat{v}_k &&= \hat{v}_{k-1} + \alpha\hat{d}_{k-1} \\
& \hat{r}_k &&= \hat{r}_{k-1} + \alpha\hat{u} \\
& \gamma_k &&= \hat{r}_k^T\hat{r}_k \quad \mapsto \quad \text{Abbruchkriterium} \quad \gamma_k < \epsilon \\
& \beta &&= \frac{\gamma_k}{\gamma_{k-1}} \\
& \hat{d}_k &&= -\hat{r}_k + \beta\hat{d}_k
\end{aligned} \tag{5.41}$$

Problematisch ist die nunmehr notwendige Faktorisierung des Vorkonditionierers und der erhöhte algorithmische Aufwand bei der Anwendung der Matrixprodukte. Dieses Verfahren bezeichnet Shewchuk [188] als transformiertes PCG-Verfahren. Durch geeignete Umformung läßt sich aber diese Transformation eliminieren.
Ersetzt man $\hat{v} = \mathbf{L}^T v$, $\hat{d} = \mathbf{L}^T d$, $\hat{r} = \mathbf{L}^{-1}$, $\hat{u} = \mathbf{L}^{-1}$, $\mathbf{L}^{-T}\mathbf{L}^{-1} = \mathbf{C}^{-1}$ erhält man das transformierten CG-Verfahren:

$$\begin{aligned}
\text{wähle} \quad & v_0 \\
& \mathbf{L}^{-1}r_0 &&= \mathbf{L}^{-1}\mathbf{K}\mathbf{L}^{-T}\mathbf{L}^{-1}v_0 - \mathbf{L}^{-1}f \\
& \mathbf{L}^T d_0 &&= -\mathbf{L}^{-1}r_0 \quad | * \mathbf{L}^{-T} \\
& \gamma_0 &&= r_0^T\mathbf{L}^{-T}\mathbf{L}^{-1}r_0 \\
\text{Iteration für} \quad & k = 1, 2, \ldots \\
& \mathbf{L}^{-1}u &&= \mathbf{L}^{-1}\mathbf{K}\mathbf{L}^{-T}\mathbf{L}^T d_{k-1} \\
& \alpha &&= \frac{\gamma_0}{d_{k-1}^T\mathbf{L}^T\mathbf{L}^{-T}u} \\
& \mathbf{L}^T v_k &&= \mathbf{L}^T v_{k-1} + \alpha\mathbf{L}^T d_{k-1} \\
& \mathbf{L}^{-1}r_k &&= \mathbf{L}^{-1}r_{k-1} + \alpha\mathbf{L}^{-1}u
\end{aligned}$$



$$\begin{aligned}
& \gamma_k &&= r_k^T\mathbf{L}^{-T}\mathbf{L}^{-1}r_k \\
& \beta &&= \frac{\gamma_k}{\gamma_{k-1}} \\
& \mathbf{L}^T d_k &&= -\mathbf{L}^{-1}r_k + \beta\mathbf{L}^{-T}d_k \quad | * \mathbf{L}^{-T}
\end{aligned} \tag{5.42}$$

Durch Vergleich erhält man daraus:

$$\begin{aligned}
\text{wähle} \quad & v_0 \\
& r_0 &&= \mathbf{K}v_0 - f \\
& d_0 &&= -\mathbf{C}^{-1}r_0 \\
& \gamma_0 &&= r_0^T r_0 \\
\text{Iteration für} \quad & k = 1, 2, \ldots \\
& u &&= \mathbf{K}d_{k-1} \\
& \alpha &&= \frac{\gamma_0}{d_{k-1}^T u} \\
& v_k &&= v_{k-1} + \alpha d_{k-1} \\
& r_k &&= r_{k-1} + \alpha u \\
& \gamma_k &&= r_k^T\mathbf{C}^{-1}r_k \quad \mapsto \quad \text{Abbruchkriterium} \quad \gamma_k < \epsilon \\
& \beta &&= \frac{\gamma_k}{\gamma_{k-1}} \\
& d_k &&= -\mathbf{C}^{-1}r_k + \beta d_k
\end{aligned} \tag{5.43}$$

Daraus kann eine effiziente Implementierung abgeleitet werden, indem das vorkonditionierte Residuum in einem zusätzlichen Vektor $p = \mathbf{C}^{-1}r$ bereitgestellt wird. Das PCG-Verfahren erfordert dann gegenüber dem CG-Verfahren Speicherplatz für diesen Vektor und den Vorkonditionierer sowie den zusätzlichen Rechenaufwand für die Anwendung der Vorkonditionierung:

$$\begin{aligned}
\text{wähle} \quad & v_0 \\
& r_0 &&= \mathbf{K}v_0 - f \\
& d_0 &&= -\mathbf{C}^{-1}r_0 \\
& \gamma_0 &&= r_0^T r_0 \\
\text{Iteration für} \quad & k = 1, 2, \ldots \\
& u &&= \mathbf{K}d_{k-1} \\
& \alpha &&= \frac{\gamma_0}{d_{k-1}^T u} \\
& v_k &&= v_{k-1} + \alpha d_{k-1} \\
& r_k &&= r_{k-1} + \alpha u \\
& p &&= \mathbf{C}^{-1}r_k \quad \text{(Vorkonditionierung)} \\
& \gamma_k &&= r_k^T p \quad \mapsto \quad \text{Abbruchkriterium} \quad \gamma_k < \epsilon \\
& \beta &&= \frac{\gamma_k}{\gamma_{k-1}} \\
& d_k &&= -p + \beta d_k
\end{aligned} \tag{5.44}$$

## 5.4 Vorkonditionierung für iterative Löser

### 5.4.1 Anforderungen an die Vorkonditionierung

Eine Vorkonditionierung $\mathbf{C}$ wird so gewählt, daß die Konditionszahl $\kappa(\mathbf{C}^{-1}\mathbf{K})$ kleiner ist, als die des Ausgangssystems. Bei der Anwendung des PCG-Verfahrens muß

die Matrix $\mathbf{C}^{-1}$ selbst nicht bereitgestellt werden. Es muß lediglich die Wirkung der Inversen von $\mathbf{C}$ auf einen Vektor in einem Matrix-Vektor-Produkt bereitgestellt werden. Die algorithmischen Anforderungen an eine geeignete Vorkonditionierung sind deshalb

- gute Approximation der exakten Zerlegung,
- effiziente und speicherplatzsparende Berechnung und Anwendung und
- stabil und flexibel in der Anwendung für breite Problemklassen.

### 5.4.2 Diagonal- und Blockdiagonalskalierung

Der beste Vorkonditionierer von $\mathbf{K}$ ist die Inverse $\mathbf{K}^{-1}$ selbst. Bei diagonaldominanten Systemen ist eine gute Schätzung von $K^{-1}$ z.B. mit den Diagonaleinträgen $\mathbf{D} = diag(\mathbf{K})$ möglich. Die Inverse von $\mathbf{D}$ läßt sich elementweise berechnen. Dieses Herangehen läßt sich auf Diagonalblöcke erweitern. Dei Blockgröße wird bei der Finite-Element-Methode problemabhängig gewählt.
Folgende Eigenschaften gelten für die (Block-) Jacobi-Methode

- Die Methode ist definiert, wenn alle Diagonalblöcke $\mathbf{K}_{ii}$ regulär sind.
- Ist $\mathbf{K}$ positiv definit, dann sind $\mathbf{D}$ und alle Diagonalblöcke $\mathbf{K}_{ii}$ positiv definit und damit regulär.

### 5.4.3 Unvollständige Zerlegungen

Die unvollständige Zerlegung ist eine attraktive Vorkonditionierungstechnik auf der Basis einer unvollständigen Faktorisierung (nach Gauß oder Cholesky) der Matrix $\mathbf{K}$. Die Idee besteht darin, eine gute Näherung $\bar{\mathbf{L}}\bar{\mathbf{U}}$ der faktorisierten Matrix $\mathbf{LU}$ zu erzeugen, die um eine Restmatrix $\mathbf{R}$ von der exakten Faktorisierung abweicht

$$\mathbf{K} = \bar{\mathbf{L}}\bar{\mathbf{U}} - \mathbf{R} \approx \mathbf{LU} \tag{5.45}$$

Generell entstehen bei der exakten Faktorisierung von $K$ wesentlich mehr Nichtnull-Elemente (*fill-in*) als in der Ausgangsmatrix. Damit werden die vorteilhaften Eigenschaften von dünnbesetzten Matrizen (geringer Speicherplatzbedarf und weniger numerischer Aufwand als bei dichtbesetzten Matrizen) zum großen Teil zunichte gemacht. Unvollständige Zerlegungen werden nun von der exakten Zerlegung abgeleitet, indem nur ein Teil der bei der Faktorisierung entstehenden Nichtnull-Elemente berücksichtigt werden. Man unterscheidet die folgende Varianten der ILU- (engl. *incomplete LU*-) Faktorisierung:

1. ILU-Faktorisierung auf vorgegebener Struktur,
2. modifizierte ILU-Faktorisierung und
3. approximierte ILU-Faktorisierung.

Die Verfahren unterscheiden sich in den Kriterien für die Auswahl der mitzuführenden oder nicht zu berücksichtigenden Nichtnull-Elementen. Diese Kriterien müssen so gewählt werden, daß die Anforderungen an einen Vorkonditionierer für die iterative Lösung des Gleichungssystems $\mathbf{K}v = f$ erfüllt werden.
Die Wahl der Bezeichnung ist in der Literatur verwirrend und uneinheitlich. Eine Diskussion der Namensgebung gibt zum Beispiel Hackbusch [90], S. 246. Die hier angegebenen Bezeichnungen entsprechen den von Hackbusch gewählten. Auf die Einführung von Abkürzungen wurde bewußt verzichtet.
In den folgenden Abschnitten wird eine Darstellung und Diskussion der Stabilität der Varianten der unvollständigen Zerlegung gegeben.



**Definitionen** Für das Verständnis der folgenden Darstellungen sind zwei Begriffe notwendig, die hier noch einmal nach Hackbusch [90] definiert werden.

- **M-Matrix**
  Eine Matrix $\mathbf{K} \in R^{n\times n}$ ist eine *M-Matrix*, wenn
  - $k_{ii} > 0$
  - $k_{ij} \leq 0$ für $i \neq j$
  - $\mathbf{K}$ ist regulär und $\mathbf{K}^{-1} \geq O$

  M steht als Abkürzung für Minkowski (Minkowskische Determinante) nach [161].
- **H-Matrix**
  Eine Matrix $\mathbf{K} \in R^{n\times n}$ ist eine *H-Matrix*, wenn $\mathbf{B} = |\mathbf{D}| - |\mathbf{K} - \mathbf{D}|$ mit $\mathbf{D} = diag(\mathbf{K})$ eine M-Matrix ist. Die Konstruktion $\mathbf{B} = |\mathbf{D}| - |\mathbf{K} - \mathbf{D}|$ ändert gerade die Vorzeichen aller Matrixelemente $k_{i,j}$ so, daß $b_{ii} \geq 0$ und $b_{ij} \leq 0$ für $i \neq j$ gilt. H steht als Abkürzung für Hadamard nach [161].

#### 5.4.3.1 ILU-Faktorisierung auf vorgegebener Struktur

Die unvollständige Zerlegung unter Beibehaltung der Originalbelegung der Matrix $\mathbf{K}$ ist die Grundform der unvollständigen Zerlegungen. Die Belegung kann dabei in Abhängigkeit von der Speichertechnik von $\mathbf{K}$ sowohl für die einzelnen Matrixeinträege als auch für ihre Submatrizen beschrieben werden. Entsprechend ist die unvollständige Zerlegung für die Matrixelemente $k_{ij} \neq 0$ oder die Submatrizen $\mathbf{K}_{ij} \neq O$ erklärt. Die folgenden Ausführungen beschränken sich auf die Elementformulierungen.
Man definiert

$$\mathbf{D} = diag(\mathbf{U}) \tag{5.46}$$

und $\hat{\mathbf{U}}$ und $\hat{\mathbf{L}}$ als strikte obere bzw. untere Dreiecksmatrix mit

$$\hat{\mathbf{U}} = \mathbf{U} - \mathbf{D}, \quad \hat{\mathbf{L}} = (\mathbf{L} - \mathbf{E})\mathbf{D} \tag{5.47}$$

Die benötigte Zerlegung ergibt sich zu

$$\mathbf{K} = (\mathbf{D} + \hat{\mathbf{L}})\mathbf{D}^{-1}(\mathbf{D} + \hat{\mathbf{U}}) - \mathbf{R} \tag{5.48}$$

In jeder Zeile $i$ des Gleichungssytems wird auf die in der Matrix $\mathbf{K}$ besetzten Elemente in den Spalten $j < i$ bzw. Zeilen $k < j$ der folgende Algorithmus angewandt und man erhält man direkt $\mathbf{D}$, $\hat{\mathbf{L}}$ und $\hat{\mathbf{U}}$

$$\begin{aligned}
&\text{für alle} \quad (i,j) \quad | \quad k_{ij} \neq 0 \\
&\qquad d_{ii} \;=\; k_{ii} - \sum_{k=1}^{k<i} \hat{l}_{ik} d_{kk}^{-1} \hat{u}_{ki} \\
&\qquad \hat{l}_{ij} \;=\; k_{ij} - \sum_{k=1}^{k<j} \hat{l}_{ik} d_{kk}^{-1} \hat{u}_{kj} \\
&\qquad \hat{u}_{ji} \;=\; k_{ji} - \sum_{k=1}^{k<j} \hat{l}_{jk} d_{kk}^{-1} \hat{u}_{ki}
\end{aligned} \tag{5.49}$$

Die Defektmatrix $\mathbf{R}$ berechnet sich mit

$$\begin{aligned}
r_{ij} &= 0 \quad \text{für} \quad k_{ij} \neq 0 \\
r_{ij} &= \sum_{k=1}^{k<n} \hat{l}_{ik}\hat{u}_{kj} - k_{ij} \quad \text{für} \quad k_{ij} = 0
\end{aligned} \tag{5.50}$$

Die so gewonnene Matrix benutzt man als Vorkonditionerer. Die Speicherplatzanforderung des Vorkonditionierers ist identisch mit der der Matrix $\mathbf{K}$ und der arithmetische Aufwand zur Anwendung des Vorkonditionierers ist nur doppelt so groß, wie ein Matrix-Vektor-Produkt mit der Matrix $\mathbf{K}$.
Die unvollständige Zerlegung auf einem vor der Faktorisierung festgelegten Belegungsmuster, das von der Originalbelegung abweicht, wurde z.B. von Meijerink [144] untersucht. Er schlägt vor, eine Menge geordneter Indexpaare (i,j) vor Beginn der Faktorisierung mit

$$\mathbf{G} = \{(i,j) \quad | \quad 1 \leq i,j \leq n\}. \tag{5.51}$$

festzulegen. Während der Zerlegung wird ein Matrixeintrag $l_{ij} \neq 0$ nur dann berücksichtigt, wenn $(i,j) \in \mathbf{G}$ gilt.

für alle $(i,j) \in \mathbf{G}$

$$\begin{aligned}
d_{ii} &= k_{ii} - \sum_{k=1}^{k<i} \hat{l}_{ik} d_{kk}^{-1} \hat{u}_{ki} \\
\hat{l}_{ij} &= \begin{cases} k_{ij} - \sum_{k=1}^{k<j} \hat{l}_{ik} d_{kk}^{-1} \hat{u}_{kj} & : \quad k_{ij} \neq 0 \\ -\sum_{k=1}^{k<j} \hat{l}_{ik} d_{kk}^{-1} \hat{u}_{kj} & : \quad k_{ij} = 0 \end{cases} \\
\hat{u}_{ji} &= \begin{cases} k_{ji} - \sum_{k=1}^{k<j} \hat{l}_{jk} d_{kk}^{-1} \hat{u}_{ki} & : \quad k_{ij} \neq 0 \\ \sum_{k=1}^{k<j} \hat{l}_{jk} d_{kk}^{-1} \hat{u}_{ki} & : \quad k_{ij} = 0 \end{cases}
\end{aligned}$$

und die Defektmatrix $\mathbf{R}$ berechnet sich mit

$$\begin{aligned}
r_{ij} &= 0 \quad \text{für} \quad (i,j) \in \mathbf{G} \\
r_{ij} &= \sum_{k=1}^{k<n} \hat{l}_{ik} \hat{u}_{kj} - k_{ij} \quad \text{für} \quad (i,j) \notin \mathbf{G}
\end{aligned}$$

Ajiz und Jennings [2] benutzen als Kriterium für die Zuordnung eines Indexelements

$$g_{ij} \in \mathbf{G} \quad | \quad \forall (i,j) \quad \text{mit} \quad |k_{ij}| \geq \epsilon. \tag{5.52}$$

Der Spezialfall mit $\epsilon = 0$

$$g_{ij} \in \mathbf{G} \quad | \quad \forall (i,j) \quad \text{mit} \quad |k_{ij}| \neq 0. \tag{5.53}$$

entspricht dem Standardalgorithmus (Gleichung 5.49).
Alternativ untersucht Timenetsky [202] die Effektivität von Zerlegungsmustern, die aufgrund der Größe des Matrixeintrags in der faktorisierten Matrix bestimmt werden, d.h. $\mathbf{G}$ wird während der Faktorisierung mit Hilfe des Kriteriums

$$g_{ij} \in \mathbf{G} \quad | \quad \forall (i,j) \quad \text{mit} \quad |l_{ij}| \geq \epsilon \tag{5.54}$$

aktualisiert. Diese Variante erfordert eine geschickte Implementierung. Die Struktur der dünnbesetzten, faktorisierten Matrix verändert sich während der Zerlegung. Damit ändert sich auch der Speicherplatzbedarf. Wird der Parameter $\epsilon$ so gewählt, daß keine Matrixelemente aus der Faktorisierung herausgenommen werden, ist der Aufwand für die unvollständige Zerlegung identisch zur exakten Faktorisierung.
Suarjana gibt in [198] unter Berücksichtigung von Speicherplatzrestriktionen Implementationsvorschläge, auch für die Kombination der Kriterien 5.52 und 5.54 an.
Die Matrix $\hat{\mathbf{K}} = \hat{\mathbf{L}}\hat{\mathbf{U}} = \mathbf{K} + \mathbf{R}$ ist nicht notwendig positiv definit. Die Existenz einer konsistenten ILU-Zerlegung ist deshalb nicht gesichert. Das führt auf die Formulierung der modifizierten ILU-Faktorisierung.



#### 5.4.3.2 Weitere ILU-Faktorisierungen

Die Matrix kann mit einer geeigneten Skalierung [142]

$$\bar{\mathbf{K}} = \mathbf{K} + (1+\gamma) D \tag{5.55}$$

mehr diagonal dominant gemacht werden. Die Wahl des Parameters $\gamma$ ist problemabhängig. Es gibt keine robuste Methode zur Bestimmung dieses Faktors [198].
Eine andere Möglichkeit der Behandlung des Rangabfalls ist, den Hauptdiagonaleintrag durch die Einheitsmatrix zu ersetzen und das Verfahren fortzusetzen. Diese modifizierten ILU-Faktorisierungen lassen sich allgemein in Analogie zu Gauß-Seidel-Überrelaxationsverfahren formulieren.
Wittum und Liebau [220] entwickeln eine approximierte ILU-Faktorisierung, Das Verfahren weist für ausgewählte Probleme sehr gute Konvergenzeigenschaften auf.

### 5.4.4 Elementweise-Vorkonditionierung

Die bisher genannten Vorkonditionierungstechniken basieren auf komplett aufgebauten Systemmatrizen. EBE- (engl. *element-by-element-*) Techniken benutzen im Gegensatz dazu lediglich Finite-Element-Matrizen. Die Idee bei der Anwendung der EBE-Technik für die Vorkonditionierung besteht darin, auf Elementebene einen Operator zu formulieren, der einerseits einfach (lokal) zu berechnen ist und andererseits eine globale Wirkung erreicht.
EBE-Vorkonditionierer wurden z.B. von Hughes[105], Nour-Omid [158] und Papadrakakis [162] vorgeschlagen. Wir folgen in der Darstellung der verschiedenen Varianten Bartelt [20]].

#### 5.4.4.1 Sortierte Cholesky-EBE Vorkonditionierung (Hughes)

Bei der sortierten Cholesky- (oder als Variante des Eliminationsverfahrens Gauß-) EBE -Vorkonditionierung wird die Systemsteifigkeitsmatrix $\mathbf{K}$ durch ein Produkt approximiert ($nel :=$ Anzahl der Elemente):

$$\mathbf{K} \approx \mathbf{M} = \mathbf{D}^{0.5} \prod_{e+1}^{nel} \mathbf{L}^e \prod_{e=nel}^{1} \mathbf{U}^e \mathbf{D}^{0.5} \tag{5.56}$$

Dabei sind die Produkte über die oberen bzw. unteren Elementmatrizen wie folgt definiert

$$\begin{aligned}
\prod_{e+1}^{nel} \mathbf{L}^e &= \mathbf{L}^1 * \mathbf{L}^2 * \ldots * \mathbf{L}^{nel} \\
\prod_{e=nel}^{1} \mathbf{U}^e &= \mathbf{U}^{nel} * \mathbf{U}^{nel-1} * \ldots * \mathbf{U}^1
\end{aligned}$$

Die Matrizen $\mathbf{L}^e$ und $\mathbf{U}^e$ sind das Resultat der Choleskyzerlegung der Matrix

$$\mathbf{K}^e = \mathbf{L}^e \mathbf{U}^e = \mathbf{E} + D^{-0.5} B^{eT} (K^e - diag(K^e)) B^e D^{-0.5} \tag{5.57}$$

Die Matrix $\mathbf{K}^e$ (Hughes [105] bezeichnet sie nach Winget [106] als *Winget regularized element matrix*) erhält man aus einer Elementsteifigkeitsmatrix $\mathbf{K}^e$ mit folgenden Umformungen

1. Nullsetzen der Diagonalelemente
2. symmetrische Skalierung mit einer Diagonalmatrix (Die Elemente der Diagonalmatrix sind die den globalen Freiheitsgraden des Elements zugeordneten Einträge $diag\mathbf{K}^{-0.5}$ ),

3. Einheitsmatrix addieren

Mit dieser Regularisierung wird gesichert, daß die Choleskyzerlegung stabil ist. Als Vorkonditionierer interessant ist die Inverse der approximierten Steifigkeitsmatrix **M**

$$\mathbf{M}^{-1} = \mathbf{D}^{-0.5} \prod_{e=nel}^{1} (\mathbf{U}^e)^{-1} \prod_{e+1}^{nel} (\mathbf{L}^e)^{-1} \mathbf{D}^{-0.5} \tag{5.58}$$

Sie kann vollständig auf Elementebene formuliert werden. Da im PCG-Verfahren nur die Wirkung des Vorkonditionierers auf einen Vektor benötigt wird, muß sie also nicht explizit aufgebaut werden, es ist ausreichend ein Produkt der Form

$$y = \mathbf{M}^{-1} x \tag{5.59}$$

bereitzustellen. Das geschieht mit den im folgenden beschriebenen Schritten.

1. Post-Diagonalskalierung
$$y' = \mathbf{D}^{-0.5} x \tag{5.60}$$

2. Vorwärtseinsetzen auf Elementebene
$$y'' = \prod_{e=nel}^{1} \mathbf{L}^{e^{-1}} y' = \mathbf{L}^{nel^{-1}} * \mathbf{L}^{nel-1^{-1}} * \ldots * \mathbf{L}^{1^{-1}} y' \tag{5.61}$$

3. Rückwärtseinsetzen auf Elementebene
$$y''' = \prod_{1}^{e=nel} \mathbf{U}^{e^{-1}} y'' = \mathbf{U}^{1^{-1}} * \mathbf{U}^{2^{-1}} * \ldots * \mathbf{U}^{nel^{-1}} y'' \tag{5.62}$$

4. Prä-Diagonalskalierung
$$y'''' = \mathbf{D}^{-0.5} y''' \tag{5.63}$$

Dieses Verfahren wird von Papadrakakis [162] wegen der Anwendung der eliminierten Matrizen im Produkt als multiplikative EBE-Methode bezeichnet.

**Bemerkung:** Dieses Verfahren erfordert die sequentielle Bearbeitung des Vektors $x$ in zwei Elementschleifen. Dabei sind alle faktorisierten Elementsteifigkeitsmatrizen bereitzustellen. Der dafür erforderliche Speicherbedarf ist für unstrukturierte Vernetzungen erheblich. Die Parallelisierung kann einfach durch eine Verteilung der Elemente geschehen, dabei dürfen die Elementmengen keine Elemente mit gemeinsamen Knoten enthalten.

#### 5.4.4.2 *Global extraction*-EBE Vorkonditionierung

Die Diagonal-Vorkonditionierung vernachlässigt die Interaktion von (in der Systemmatrix über Nebendiagonalelemente) gekoppelten Freiheitsgraden. Dabei ist die Diagonal-Vorkonditionierung nicht auf einen Diagonalvektor beschränkt - man kann ein solches Verfahren auch mit einer beliebigen Anzahl von Sub- oder Superdiagonalen formulieren.
Bei der elementbezogenen Vorkonditionierung, die im folgenden beschrieben wird, benutzt man jeweils die den Elementfreiheitsgraden zugeordneten Anteile der globalen Steifigkeitsmatrix, um eine leicht invertierbare Approximation der Systemsteifigkeitsmatrix zu konstruieren. Physikalisch interpretiert löst man ein Problem,



in dem das betrachtete Element in einem allseitig gelagerten, umschließenden Elementpatch eingeschlossen ist.
Der Vorkonditionierer wird berechnet mit

$$\mathbf{M}^{-1} = \sum_{e=1}^{nel} \mathbf{B}^{eT} (\mathbf{K}^e_{ext})^{-1} \mathbf{B}^e \tag{5.64}$$

Die extrahierten Elementmatrizen sind invertierbar. Die Anwendung des Vorkonditionierers läßt sich durch eine elementweise Vorwärtselimination/Rückwärtssubstitution mit den elementbezogenen Vektoranteilen und anschließender Superposition formulieren:

$$\begin{aligned} x^e_{ext} &= \mathbf{B}^e x \\ y^e_{ext} &= (\mathbf{K}^e_{ext})^{-1} x^e_{ext} = \mathbf{U}^e_{ext} \mathbf{L}^e_{ext} x^e_{ext} \\ y &= \sum_{e=1}^{nel} \mathbf{B}^{eT} y^e_{ext} \end{aligned} \tag{5.65}$$

Wegen der summarischen Anwendung des Vorkonditionierers wird dieser Typ auch als additive EBE-Methode [162] bezeichnet.
Mit der Rechenvorschrift 5.65 wird allerdings jeder Anteil des Vektors $x$ so oft verarbeitet, wie er als Elementfreiheitsgrad im Gesamtsystem auftaucht. Dieser Fehler kann näherungsweise durch eine Skalierung entsprechend der Anzahl von Elementen, die an einen Knoten anschließen, vermieden werden. Die Skalierung muß dabei symmetrisch auf die extrahierte Elementmatrix angewendet werden.

$$(\bar{\mathbf{K}}^e_{ext})^{-1} = \mathbf{S}^{e-0.5} (\mathbf{K}^e_{ext})^{-1} \mathbf{S}^{e-0.5} \tag{5.66}$$

Als Alternative zur anzahlmäßigen Skalierung mit

$$\mathbf{S}^e_{ii} = (\textit{Anzahl der mit Knoten i benachbarten Elemente})^{-1} \tag{5.67}$$

bietet sich eine Skalierung entsprechend den Steifigkeitsverhältnissen an. Dabei wird der Hauptdiagonalwert der Systemsteifigkeitsmatrix $d_{ii}$ entsprechend dem Elementanteil $d^e_{ii}$ gewichtet

$$\mathbf{S}^e_{ii} = \frac{d^e_{ii}}{d_{ii}} \tag{5.68}$$

Steife Elemente erhalten damit größeren Einfluß als weichere Elemente.

**Bemerkung:** Dieser Vorkonditionierer stellt keine Anforderungen an die Reihenfolge der elementbezogenen Schritte. Er ist daher parallelisierbar. Für den Fall der nichtüberlappenden Gebietszerlegung entsteht das Problem, Teile der Steifigkeitsmatrix, die Steifigkeitsbeziehungen zwischen Koppelknoten in einem Element vermitteln, auszutauschen.

#### 5.4.4.3 Reduzierte *Global extraction*-EBE Vorkonditionierung

Nachteile der bisher dargestellten EBE-Vorkonditionierer sind zusätzlicher Speicherbedarf für die zerlegten Elementmatrizen (für jedes Element eine Matrix) und zusätzlicher Rechenaufwand in der Größenordnung der Elementanzahl. Im parallelen Fall ist der Austausch und die Speicherung zusätzlicher Elemente der Systemmatrix erforderlich. In Gleichung 5.66 wurde zur Berücksichtigung von mehrfacher Beteiligung eines Knotenfreiheitsgrades im Elementprozeß eine Skalierung eingeführt.

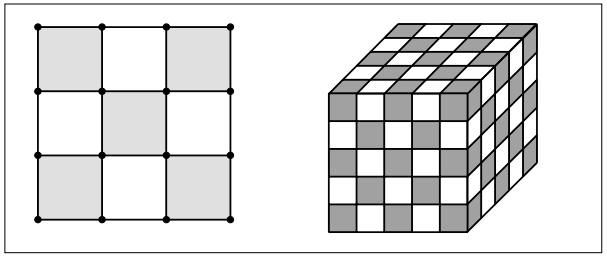

Abbildung 5.7: *Extrahierten Elemente bei ungerader Elementanordnung*

Zur Vermeidung dieser Skalierung (die fehlerbehaftet ist) läßt sich das folgende reduzierte EBE-Schema entwickeln. Die Elementschleife umfaßt nur eine Teilmenge aller Elemente, so daß jeder Knotenwert nur von genau einer Elementmatrix modifiziert wird (Abbildung 5.7).
Die Einsparungen an Aufwand (Speicherplatz, Operationen) können nach Bartelt [20] für ein dreidimensionales Gebiet, das mit $n \times n \times n$ Würfeln diskretisiert wurde, wie folgt abgeschätzt werden: Angenommen, $n$ ist ungerade, dann werden bei der globalen Extraktion $n^3$ Elemente benutzt . Für die reduzierte Extraktion benötigt man nur noch $(\frac{n-1}{2})^3$ Elemente. Konkret bedeutet das für $n = 15$ eine Verringerung des Aufwandes von 3.375 Elementen auf 512 Elemente.
Das Problem besteht in der Auswahl der Elementmenge. Für beliebige Netzkonfigurationen werden eine Anzahl von Knoten nicht durch vollständige Elemente in die Modifikation einbezogen. Ist zum Beispiel für den einfachen Fall des regulären Rechteckgitters die Anzahl der Element auf einer Seite gerade, dann bleibt eine komplette Reihe von Knoten übrig (Abbildung 5.8). Als Lösung bietet sich die Verwendung von „Teil"elementen an. Dabei werden nur Teile der elementbezogenen Steifigkeitsmatrix aufgebaut, um einen Operator für jeden Freiheitsgrad der verbleibenden Knoten zu konstruieren. Dabei sollte aber vermieden werden, diese Teilmatrizen lediglich für einen Knoten aufzubauen. Dann wäre der Vorkonditionierer in diesem Knoten identisch mit der Diagonalskalierung und sehr wenig effizient. Es sollten in einem „Teil"element also mindestens zwei Knoten erfaßt werden.
Der Algorithmus zum Aufstellen des Vorkonditionierers besteht damit aus folgenden Teilen

1. Identifiziere die Menge der Elemente, bestimme ggf. „Teil"elemente, d.h. modifizierte boolsche Operatoren, um z.B. Randknoten zu erfassen.
2. Bestimme die Choleskyfaktoren der unskalierten Element- und „Teil"elementmatrizen.

Dieser Vorkonditionierer wird dann wie in Abschnitt 5.4.4.2 beschrieben, angewendet.

#### 5.4.4.4 Polynomiale Vorkonditionierung

Trotz der Reduzierung des Speicheraufwandes für den zuletztgenannten Vorkonditionierer im Vergleich zu EBE-Cholesky oder *Global extraction*-EBE ist der zusätzliche Speicherbedarf erheblich. Wünschenswert ist ein Vorkonditionierer, der keinen zusätzlichen Speicherbedarf hat.



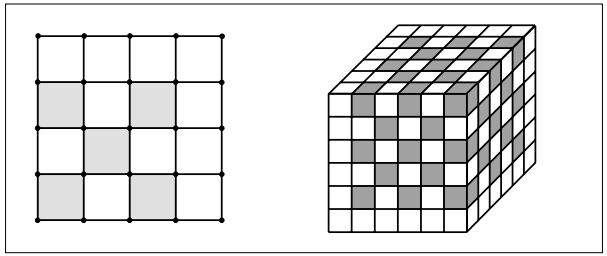

Abbildung 5.8: *Extrahierten Elemente bei gerader Elementanordnung*

Als Alternative zu den *faktorisierten* Vorkonditionierern (die die Matrix $\mathbf{K}$ approximieren) bieten sich *inverse* Vorkonditionierer an, die $\mathbf{K}^{-1}$ approximieren. Einem Vorschlag von Dubois, Greenbaum und Rodrigue [55] folgend, wird ein inverser Vorkonditionierer formuliert mit

$$\mathbf{M}^{-1} = (\mathbf{N}^0 + \mathbf{N}^1 + \mathbf{N}^2 + \cdots + \mathbf{N}^{p-1})D^{-1} \tag{5.69}$$

und

$$\mathbf{N}^i = (\mathbf{E} - \mathbf{D}^{-1}\mathbf{K})^i \tag{5.70}$$

Diese Abschätzung ist gültig, wenn für den Spektralradius (betragsmäßig größter Eigenwert) gilt:

$$\varrho(\mathbf{E} - \mathbf{D}^{-1}\mathbf{K}) = \max\left\{|\lambda| \quad \text{mit} \quad \lambda \in \sigma(\mathbf{E} - \mathbf{D}^{-1}\mathbf{K})\right\} < 1. \tag{5.71}$$

Dafür ist es erforderlich, daß $\mathbf{K}$ eine M-Matrix ist. Um die Methode auch für allgemeine FEM-Systeme zu erschließen, führt Adams [1] deshalb folgende Zerlegung ein:

$$\mathbf{K} = \mathbf{D} - \mathbf{Q} \tag{5.72}$$

Dabei ist $\mathbf{D} = diag(\mathbf{K})$. Die Hauptdiagonale von $\mathbf{Q}$ besteht aus Nullelementen, die Nebendiagonalelemente sind die Nebendiagonalelemente von $\mathbf{K}$ mit umgekehrtem Vorzeichen. $\mathbf{D}$ und $\mathbf{Q}$ können direkt aus $\mathbf{K}$ abgeleitet werden. Es entsteht damit kein zusätzlicher Speicherbedarf. Mit der Beziehung

$$\mathbf{G} = \mathbf{D}^{-1}\mathbf{Q} \tag{5.73}$$

erhält man den in $\mathbf{G}$ polynomialen Vorkonditionierer

$$\mathbf{M}^{-1} = (\mathbf{G}^0 + \mathbf{G}^1 + \mathbf{G}^2 + \cdots + \mathbf{G}^{p-1})\mathbf{D}^{-1} \tag{5.74}$$

Mit den von Johnson, Micchelli und Paul [111] benutzten Koeffizienten $\alpha_i$ erhält man damit den polynomialen Vorkonditionierer

$$\mathbf{M}^{-1} = (\alpha_0\mathbf{E}^0 + \alpha_1\mathbf{G}^1 + \alpha_2\mathbf{G}^2 + \cdots + \alpha_{p-1}\mathbf{G}^{p-1})\mathbf{D}^{-1} \tag{5.75}$$

Die Konvergenz des PCG-Algorithmus ist garantiert, wenn der Vorkonditionierer $M$ symmetrisch und positiv-definit ist. Für ungerade $p$ ist $\mathbf{M}$ positiv-definit dann und nur dann, wenn $\mathbf{D}$ positiv-definit ist. Für geradzahlige $p$ ist $\mathbf{M}$ positiv-definit dann und nur dann, wenn $\mathbf{D}+\mathbf{Q}$ positiv-definit ist. Diese Bedingung kann im allgemeinen

nicht eingehalten werden, für die FEM ist deshalb die Wahl von ungeraden $p$ erforderlich, da die Diagonalmatrix immer positiv-definit ist. Im Fall $p = 1$, $\alpha_0 = 1.0$ ist das Verfahren identisch zur Diagonalskalierung.
Für den Fall $p = 3$ ergibt sich mit

$$\mathbf{M}^{-1} = (\alpha_0 \mathbf{E}^0 + \alpha_1 \mathbf{G}^1 + \alpha_2 \mathbf{G}^2)\mathbf{D}^{-1} \tag{5.76}$$

und

$$\alpha_0 = \frac{35}{32}, \quad \alpha_1 = \frac{50}{32}, \quad \alpha_2 = \frac{35}{32} \tag{5.77}$$

(nach [111]) für

$$y = \mathbf{M}^{-1} x \tag{5.78}$$

folgendes Verfahren (mit den Hilfsvektoren u und v)

$$\begin{aligned} y' = u &= \mathbf{D}^{-1} x \\ v &= \mathbf{D}^{-1}\mathbf{Q} y' \\ y'' &= \alpha_1 u + \alpha_2 v \\ v &= \mathbf{D}^{-1}\mathbf{Q} y'' \\ y''' &= \alpha_0 u + v \end{aligned} \tag{5.79}$$

Dabei bezeichnen $y, y', y''$ und $y'''$ fortgesetztes Überschreiben des Ergebnisvektors $y$. Die angegebene Formulierung ist ideal für eine Implementierung auf einem Vektorrechner.
Das Verfahren kann bei modifizierter Benutzung der Routinen aus Abschnitt 5.4.4.2 auch als polynomiale EBE-Vorkonditionierung formuliert werden. Dabei werden die Matrix-Vektor-Produkte auf Elementebene realisiert. Die Anzahl der wesentlichen Operationen wird gegenüber dem vollständigen Matrix-Vektor-Produkt entscheidend verringert. Der Nachteil der (im Fall $p = 3$) zweimaligen Ausführung des Matrix-Vektor-Produktes kann damit ausgeglichen werden.

**Bemerkung:** Das Verfahren ist parallelisierbar. Die Koeffizienten des Polynoms sind problemabhängig und können nicht allgemeingültig festgelegt werden. Weitere Angaben z.B. in [162],[43].

### 5.4.5 Zusammenfassung

Die elementweise formulierten Vorkonditionierer kann man auch auf Blöcke von Elementen anwenden. Sinngemäß ist die Konstruktion von Vorkonditionierern für Superelemente möglich (s.a. Nour-Omid [157]). Die Wirksamkeit der angegebenen verfahren hängt von problemspezifischen Parametern ab, die gegebenefalls zuvor zu bestimmen sind.

## 5.5 Das primäre Gebietszerlegungsverfahren

Gebietszerlegungsverfahren (DD-Verfahren, Domain Decomposition Method) bezeichnen eine Klasse von Lösungstechniken für Systeme von partiellen Differentialgleichungen in einem gegebenen Gebiet durch Aufteilung des Gesamtgebiets in kleinere Teilgebiete und die Konstruktion der Gesamtlösung aus Lösungen der kleineren Teilprobleme.
Für den einfachsten Fall der Gebietszerlegung in zwei Teilgebiete mit einer gemeinsamen Koppelkante und den damit entstehenden disjunkten Knotenmengen $\Omega^1$ und



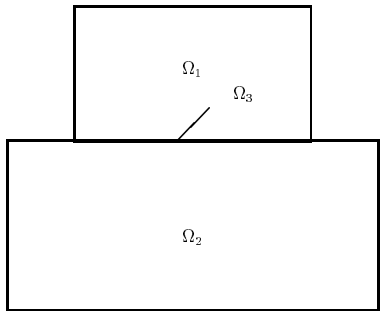


Abbildung 5.9: *Nichtüberlappende Gebietszerlegung*

$\Omega^2$ und der Koppelknotenmenge $\Omega^3$ (Abbildung 5.9) kann das Problem $Kv = f$ in Blockformulierung angegeben werden:

$$\begin{pmatrix} \mathbf{K}^{11} & & \mathbf{K}^{13} \\ & \mathbf{K}^{22} & \mathbf{K}^{23} \\ \mathbf{K}^{31} & \mathbf{K}^{32} & \mathbf{K}^{33} \end{pmatrix} \begin{pmatrix} v^1 \\ v^2 \\ v^3 \end{pmatrix} = \begin{pmatrix} f^1 \\ f^2 \\ f^3 \end{pmatrix} \tag{5.80}$$

Dieses Blockgleichungssystem kann in folgenden Schritten gelöst werden. Zunächst werden die lokalen Gleichungen in $v^1$ und $v^2$ durch Multiplikation von links mit $\mathbf{K}^{31}\mathbf{K}^{11^{-1}}$ bzw. $\mathbf{K}^{32}\mathbf{K}^{22^{-1}}$ an die globale Gleichung $v^3$ angepaßt:

$$\mathbf{K}^{31}v^1 + \mathbf{K}^{31}\mathbf{K}^{11^{-1}}\mathbf{K}^{13}v^3 = \mathbf{K}^{31}\mathbf{K}^{11^{-1}}f^1 \tag{5.81}$$

$$\mathbf{K}^{32}v^2 + \mathbf{K}^{32}\mathbf{K}^{22^{-1}}\mathbf{K}^{23}v^3 = \mathbf{K}^{32}\mathbf{K}^{22^{-1}}f^2 \tag{5.82}$$

Damit kann die Gleichung in $v^3$ zu

$$(\mathbf{K}^{33} - \mathbf{K}^{31}\mathbf{K}^{11^{-1}}\mathbf{K}^{13} - \mathbf{K}^{32}\mathbf{K}^{22^{-1}}\mathbf{K}^{23})v^3 = f^3 - \mathbf{K}^{31}\mathbf{K}^{11^{-1}}f^1 - \mathbf{K}^{32}\mathbf{K}^{22^{-1}}f^2 \tag{5.83}$$

reduziert und gelöst werden. Die Lösung für die Koppelvariablen $v^3$ wird dann in die Teilgleichungssysteme eingesetzt und es wird die korrigierte, endgültige Lösung in den Teilgebieten ermittelt:

$$v^1 = f^1 - \mathbf{K}^{11^{-1}}\mathbf{K}^{13}v^3 \tag{5.84}$$

$$v^2 = f^2 - \mathbf{K}^{22^{-1}}\mathbf{K}^{23}v^3 \tag{5.85}$$

Dieses Verfahren ist identisch mit

1. der Lösung der Teilprobleme unter Dirichletschen Randbedingungen auf den Koppelrändern,
2. dem Einsetzen der Lösung in das reduzierte Koppelknotensystem
3. dem Lösen des reduzierten Koppelsystems und
4. dem Rückrechnen in den Teilgebieten mit den ermittelten Unbekannten auf den Koppelrändern.

Diese Lösungsidee ist als *Substrukturtechnik* bei begrenztem Hauptspeicher geeignet, ein Gesamtproblem in Teilprobleme so zu zerlegen, daß jedes Teilproblem für sich genommen vollständig im Hauptspeicher bearbeitet werden kann. Diesen Vorteil erkauft man sich durch den Mehraufwand beim Aufbau und Lösen des reduzierten Koppelsystems und dem Rückrechnen in den Teilgebieten.
Allgemein für eine nichtüberlappende Zerlegung in die $i$ Teilgebiete $\Omega = \bigcup_i \Omega^i$ hat das System $\mathbf{K}v = f$ die folgende Blockstruktur

$$\begin{pmatrix} \mathbf{K}^{11} & & & \mathbf{K}^{1k} \\ & \mathbf{K}^{22} & & \mathbf{K}^{2k} \\ & & \ddots & \vdots \\ \mathbf{K}^{k1} & \mathbf{K}^{k2} & \cdots & \mathbf{K}^{kk} \end{pmatrix} \begin{pmatrix} v^1 \\ v^2 \\ \vdots \\ v^k \end{pmatrix} = \begin{pmatrix} f^1 \\ f^2 \\ \vdots \\ f^k \end{pmatrix} \tag{5.86}$$

oder allgemein

$$\begin{pmatrix} \mathbf{K}^{ll} & \mathbf{K}^{lk} \\ \mathbf{K}^{kl} & \mathbf{K}^{kk} \end{pmatrix} \begin{pmatrix} v^l \\ v^k \end{pmatrix} = \begin{pmatrix} f^l \\ f^k \end{pmatrix} \tag{5.87}$$

Dabei bezeichnet der Index $k$ die Koppelknoten und der Index $l$ die lokalen, d.h. vollständig innerhalb der Teilgebiete liegenden, Knoten. Die Matrix $\mathbf{K}^{ll}$ besitzt Blockdiagonalgestalt während die Matrix $\mathbf{K}^{lk}$ aus rechteckigen Koppelmatrizen besteht. Das Belegungsmuster von $\mathbf{K}^{lk}$ hängt vom Zusammenhang der Teilgebiete ab. Die Struktur der Matrix $\mathbf{K}^{kk}$ ist abhängig von der Ausbildung der Gebietsberandungen. Für den Spezialfall der streifenweisen Zerlegung eines Gesamtgebietes ist auch $\mathbf{K}^{kk}$ blockdiagonal.
Blockweise Gauß-Elimination liefert

$$\begin{pmatrix} \mathbf{K}^{11} & & & \mathbf{K}^{1k} \\ & \mathbf{K}^{22} & & \mathbf{K}^{2k} \\ & & \ddots & \vdots \\ \mathbf{O} & \mathbf{O} & \cdots & \mathbf{S} \end{pmatrix} \begin{pmatrix} v^1 \\ v^2 \\ \vdots \\ v^k \end{pmatrix} = \begin{pmatrix} f^1 \\ f^2 \\ \vdots \\ f^k \end{pmatrix} \tag{5.88}$$

mit

$$\mathbf{S} = \mathbf{K}^{kk} - \sum_i \mathbf{K}^{ki}\mathbf{K}^{ii^{-1}}\mathbf{K}^{ik} \tag{5.89}$$

bzw. allgemein

$$(\mathbf{K}^{kk} - \mathbf{K}^{kl}\mathbf{K}^{ll^{-1}}\mathbf{K}^{lk})v^k = f^k - \mathbf{K}^{kl}\mathbf{K}^{ll^{-1}}f^l \tag{5.90}$$

mit

$$\mathbf{S} = \mathbf{K}^{kk} - \mathbf{K}^{kl}\mathbf{K}^{ll^{-1}}\mathbf{K}^{lk} \tag{5.91}$$

als kondensierte Steifigkeitsmatrix (Anwendung in der Substrukturtechnik), als Kapaziätsmatrix (Anwendung in der Elektrotechnik) oder als sogenanntes Schurkomplement (Mathematik) bezeichnet.
Die Bedeutung dieses Herangehens liegt nicht in der direkten Anwendung in einem parallelen Gleichungslöser, sondern in der Eignung für die Konstruktion sogenannter Dirichlet-Vorkonditionierer.
Das Schurkomplement läßt sich als Summe über lokale Beiträge ermitteln. Damit ist diese Formulierung hervorragend für die Anwendung in parallelen Lösungstechniken geeignet.

### 5.5.1 Datenverteilung auf Grundlage der nichtüberlappenden Gebietszerlegung

Mit der eindeutigen Zuordnung von finiten Elementen zu Teilgebieten und damit Prozessoren wird eine Datenverteilung für die Gesamtsteifigkeitsmatrix $\mathbf{K}$ bestimmt, die als *additive* Datenverteilung bezeichnet werden soll. Es sei $B^s$ die Zuordnungsmatrix für die Abbildung der zu Teilgebiet $s$ gehörenden Freiheitsgrade



zu den Freiheitsgraden des Gesamtsystems. Dann besteht der additive Beitrag des Teilgebiets $s$ zur Gesamtsteifigkeitsmatrix in ${\mathbf{B}^s}^T\mathbf{K}^s\mathbf{B}^s$. Analog ist die additive Verteilung der Rechten Seite zu verstehen und wir erhalten die entsprechenden globalen Daten mit

$$\mathbf{K} = \sum_s {\mathbf{B}^s}^T\mathbf{K}^s\mathbf{B}^s, \quad f = \sum_s {\mathbf{B}^s}^T f^s \tag{5.92}$$

Eine Datenverteilung, die identische Werte für die entsprechenden Freiheitswerte in allen betroffenen Teilgebieten vorhält, ist die *globale* Datenverteilung. Die einem Teilgebiet zugeordneten Anteile am globalen Verschiebungsvektor bestimmen sich zu

$$v^s = \mathbf{B}^s v. \tag{5.93}$$

Die Anzahl der redundant vorgehaltenen Entsprechungen eines Eintrages entspricht der Anzahl der Teilgebiete, in denen ein bestimmter Freiheitsgrad vertreten ist.
Die *konsistente* Datenverteilung sichert die redundanzfreie Speicherung, indem ein Datum für einen Koppelfreiheitsgrad nur in einem Prozessor gespeichert wird (dem *master*- Prozessor):

$$v^s = \begin{cases} v & : \quad s = \text{master} \\ 0 & : \quad \text{sonst} \end{cases} \tag{5.94}$$

Sie ist in ihrer Handhabung äquivalent zur additiven Datenspeicherung mit dem Vorteil, daß bei Bedarf in ausgezeichneten Prozessoren bereits der globale Wert vorliegt. In den anderen Prozessoren kann der globale Wert nur durch Kommunikation erhalten werden. Diese Operation läßt sich auch als globale Superposition auffassen, wobei die lokalen Beiträge der nicht ausgezeichneten Prozessoren zu Null gesetzt sind.
Die konsistente Datenverteilung ist mit einer einfachen logischen Operation aus der globalen Datenverteilung zu erzeugen. Die Bereitstellung einer globalen Verteilung aus einer additiven erfordert einen Datenaustausch. In diesem Datenaustausch sind zwischen benachbarten Teilgebieten die additiven Daten auszutauschen und zu den globalen Werten zu superponieren. Dieser Austausch kann auch als kollektive Kommunikation aufgefaßt werden, allerdings ist der komplette Datenaustausch nicht zwischen allen Teilgebieten nicht erforderlich. Für große Prozessorzahlen ist dieses Herangehen deshalb nicht effektiv.
Das Skalarprodukt aus einem additiv und einem global verteilten Vektor ergibt sich direkt aus

$$(\sum_s {\mathbf{B}^s}^T f^s)^T x = \sum (f^{s^T}\mathbf{B}^s x) = \sum ({f^s}^T x^s) \tag{5.95}$$

und damit zu einem vollständig lokal zu berechnenden Skalarprodukt mit anschließender Summenbildung über die Teilergebnisse. Im Vergleich zur seriellen Implementation besteht der zusätzliche Aufwand in der Kommunikation und Summenbildung über die Teilergebnisse und in den für die Koppelknoten mehrfach ausgeführten Produkt über die zugeordneten Vektorkomponenten.
Skalarprodukte aus konsistent verteilten Vektoren berechnen sich ähnlich.

$$v^T\tilde{v} = \sum_s ({v^s}^T\tilde{v}^s) \tag{5.96}$$

Die konsistente Datenverteilung für die Vektoren muß dazu identisch sein. Der zusätzliche Aufwand reduziert sich auf die Kommunikation und Summenbildung über die Teilergebnisse.
Skalarprodukte von global verteilten Vektoren berechnen sich entweder nach Umwandlung in konsistente Vektoren oder durch eine geeignete Skalierung eines Vektorbeitrages und anschließender Summenbildung. Der Skalierungsfaktor ist das Re-

ziproke der Anzahl der redundanten Entsprechungen eines Vektorwertes $r^s$ im Gesamtsystem.

$$x^T x = \sum_s (r^s x^{s^T} x^s) \tag{5.97}$$

Die Entscheidung für die Skalierung oder für die Umwandlung in eine konsistente Datenspeicherung (vorzugsweise durch eine Maskierung der Vektoreinträge) kann auf der Basis der lokalen Leistungsfähigkeit der Rechner getroffen werden (z.B. Eignung der Skalierung für Vektorisierung bzw. Maskierung vorzugsweise für RISC-Architekturen). Der Kommunikationsaufwand ist bei beiden Verfahren identisch.
Das Produkt aus einer additiv verteilten Matrix und einem globale verteiltem Vektor ergibt sich zu

$$\sum_s (\mathbf{B}^{s^T} \mathbf{K}^s \mathbf{B}^s) x = \sum_s \mathbf{B}^{s^T} (\mathbf{K}^s \mathbf{B}^s x) = \sum_s \mathbf{B}^{s^T} (\mathbf{K}^s x^s) \tag{5.98}$$

und damit zu einem additiv verteiltem Vektor, dessen Anteile vollständig parallel mit ausschließlich lokalen Daten berechnet werden können.

### 5.5.2 Iterative Lösung des Schurkomplementproblems

Ein PCG-Algorithmus, der nun das Schurkomplementproblem iterativ löst, hat folgende Gestalt

$$\begin{array}{rrcl}
\text{wähle} & v_0^k & & \\
 & r_0^k & = & \mathbf{K}^{kk} v_0^k - f^k - \mathbf{K}^{kl} \mathbf{K}^{ll^{-1}} f^l \\
 & d_0^k & = & -\mathbf{C}^{kk^{-1}} r_0^k \\
 & \gamma_0 & = & d_0^{k^T} r_0^k \\
\text{Iteration für} & i = 1, 2, \ldots & & \\
 & u^k & = & (\mathbf{K}^{kk} - \mathbf{K}^{kl} \mathbf{K}^{ll^{-1}} \mathbf{K}^{lk}) d_{i-1}^k \\
\dagger & \alpha & = & \dfrac{\gamma_0}{d_{i-1}^{k^T} u^k} \\
 & v_i^k & = & v_{i-1}^k + \alpha d_{i-1}^k \\
 & r_i^k & = & r_{i-1}^k + \alpha u^k \\
\ddagger & p^k & = & \mathbf{C}^{kk^{-1}} r_i^k \quad \text{(Vorkonditionierung)} \\
\dagger & \gamma_i & = & r_i^{k^T} p^k \quad \mapsto \quad \text{Abbruchkriterium} \quad \gamma_i < \epsilon \\
 & \beta & = & \dfrac{\gamma_i}{\gamma_{i-1}} \\
 & d_i^k & = & -p^k + \beta d_i^k
\end{array} \tag{5.99}$$

Anschließend muß in den Teilgebieten noch zurückgerechnet werden

$$v^l = f^l - \mathbf{K}^{ll^{-1}} \mathbf{K}^{lk} v^k. \tag{5.100}$$

Der Aufwand pro Iteration wird wesentlich vom Schurkomplement-Produkt bestimmt. Die Berechnung erfolgt vollständig parallel und ausschließlich mit lokalen Daten. Es entsteht ein additiv verteilter Vektor $u^k$. Eine anschließende globale Vektoraddition ist nicht erforderlich, weil der Vektor $u^k$ nur in einem Skalarprodukt mit einem absolut verteilten Vektor und zur Aktualisierung des additiv verteilten Residuenvektors $r^k$ benötigt wird.
Vor der Anwendung der Vorkonditionierung ist der additiv verteilte Residuenvektor



$r^k$ mit einer globalen Vektoraddition in einen absolut verteilten Vektor zu transformieren. Der Vorkonditionierer muß ebenfalls als absolut verteilter Operator wirken, um ohne zusätzlichen Kommunikationsaufwand einen absolut verteilten Vektor der vorkonditionierten Residuen $p^k$ zu erzeugen. Der absolut verteilte Vektor $p^k$ wird dann zur Berechnung des Skalarprodukts mit einem additiv verteiltem Vektor und zur Aktualisierung des absolut verteilten Vektors der Suchrichtungen benutzt.
Die Kommunikationsanforderungen bestehen daher lediglich im Austausch von zwei Skalardaten (gekennzeichnet mit †) und in einer globalen Vektorsuperposition (gekennzeichnet mit ‡. Der Skalaraustausch führt zur Synchronisation der parallelen Prozesse. Die Vektorsuperposition wird am besten mit einem Punkt-zu-Punkt-Austausch zwischen benachbarten Teilgebieten implementiert.

### 5.5.3 Vollständig iterative Lösung

Alternativ zur Lösung des Gleichungssystems mit Hilfe des Schurkomplements kann auch direkt unter Ausnutzung der oben beschriebenen Datenverteilung ein paralleler CG-Algorithmus formuliert werden.

$$\begin{array}{rrcl}
\text{wähle} & v_0 & & \\
 & r_0 & = & \mathbf{K} v_0 - f \\
 & d_0 & = & -\mathbf{C}^{-1} r_0 \\
 & \gamma_0 & = & d_0^T r_0 \\
\text{Iteration für} & i = 1, 2, \ldots & & \\
 & u & = & \mathbf{K} d_{i-1} \\
 & \alpha & = & \dfrac{\gamma_0}{d_{i-1}^T u} \\
 & v_i & = & v_{i-1}^k + \alpha d_{i-1} \\
 & r_i & = & r_{i-1}^k + \alpha u \\
 & p & = & \mathbf{C}^{-1} r_i \quad \text{(Vorkonditionierung)} \\
 & \gamma_i & = & r_i^T p \quad \mapsto \quad \text{Abbruchkriterium} \quad \gamma_i < \epsilon \\
 & \beta & = & \dfrac{\gamma_i}{\gamma_{i-1}} \\
 & d_i & = & -p + \beta d_i
\end{array} \tag{5.101}$$

Angegeben ist hier die vorkonditionierte Variante (PCG-Algorithmus), die für $\mathbf{C} = \mathbf{E}$ identisch zum CG-Verfahren ist. Das Verfahren besitzt die gleichen Kommunikationsanforderungen wie die in Abschnitt 5.5.2 beschriebene iterative Lösung des Schurkomplementproblems. Law [132] veröffentlichte 1986 diese Variante eines auf Elementebene formulierten parallelen CG-Algorithmus. Er referiert dabei frühere Arbeiten von Fried [72] und Fox und Stanton [71].
Meyer [148] benutzt den gleichen Ansatz, um weiterführend den Vorkonditionierungsschritt mit Hilfe der Schurkomplement-Formulierung einzuführen. Damit ist ein vollständig iteratives PCG-Verfahren, das für das lokale und das Koppelproblem unterschiedliche Vorkonditionierungsoperatoren einsetzt, verfügbar:

$$\begin{aligned}
p^k &= \mathbf{C}^{kk^{-1}} (r^k - \mathbf{K}^{kl} \mathbf{C}^{ll^{-1}} r^l) \\
p^l &= \mathbf{C}^{ll^{-1}} (r^l - \mathbf{K}^{lk} p^k)
\end{aligned} \tag{5.102}$$

Um den Preis der zweifachen, aber vollständig parallelisierbaren, Anwendung des inneren Vorkonditionierers ist diese Technik äquivalent zur sequentiellen, vollständig iterativen Lösungstechnik. Analog dem Vorgehen bei der iterativen Lösung des

Schurkomplementproblems bestehen Kommunikationsanforderungen bei der Vorkonditionierung des Koppelproblems (globalen Vektorsuperposition) und bei der Bereitstellung der beiden Skalarprodukte (globale Skalaraddition).

## 5.6 Das duale Gebietszerlegungsverfahren

### 5.6.1 Variationsformulierung

Ausgehend von der Standardformulierung des Elasitzitatsproblems (5.1) ist alternativ auch folgender Ansatz möglich:
Ist $\Omega$ in $n_s$ Teilgebiete zerlegt, dann ist das Elastizitätsproblem äquivalent zu dem Problem, Stationaritätspunkte der Energiefunktionale in den Teilgebieten zu finden

$$\Pi^s(u^s) = \frac{1}{2}a(u^s,u^s)_{\Omega^s} - (u^s,f)_{\Omega^s} - (u^s,h)_{\Gamma^s}, \quad s = 1,2,..N_s \tag{5.103}$$

Die Gebietsintegrale (5.2) beziehen sich sinngemäß auf die Teilgebiete $s$. Auf dem Rand ist zusätzlich die Kontinuitätsbedingung

$$u^p = u^q \quad \forall p,q, \emptyset \neq \bar{\Omega}^p \cap \bar{\Omega}^q \tag{5.104}$$

zu erfüllen.
Die Lösung der $n_s$ Variationsprobleme (5.103) unter der zusätzlichen Kontinuitätsbedingung (5.104) ist äquivalent zur Lösung eines Sattelpunktproblems

$$\Pi^*(u^1,u^2,..,u^{n_s},\mu^1,\mu^2,..,\mu^t = \sum_{s=1}^{n_s}\Pi^s u^s + \sum_{s,q=1}^{n_s}(u^s-u^q,\mu^{sq})_{\Gamma^s} \tag{5.105}$$

mit

$$(u^s-u^q,\mu^{sq}) = \int_{\Gamma^s}\mu^{sq}(u^s-u^q)d\Gamma \tag{5.106}$$

d.h. der Bestimmung der Verschiebungsfelder $v^s$, $s = 1,2,...,n^s$ und der Lagrangefaktoren $\lambda^{sq}$ für die Übergangsbedingungen miteinander benachbarter Gebiete, die die Ungleichungen

$$\begin{aligned} & \Pi^*(v^1,v^2,\ldots,v^{n_s},\mu^1,\mu^2,\ldots,\mu^t) \\ \leq\ & \Pi^*(v^1,v^2,\ldots,v^{n_s},\lambda^1,\lambda^2,\ldots,\lambda^t) \\ \leq\ & \Pi^*(u^1,u^2,\ldots,u^{n_s},\lambda^1,\lambda^2,\ldots,\lambda^t) \end{aligned} \tag{5.107}$$

für freiwählbare $u^s$ und $\mu^s$ erfüllen.
Auch dieses Funktional kann durch eine Standard-Galerkinformulierung approximiert und in das folgende algebraische Gleichungssystem

$$\mathbf{K}^s v^s = f^s - \mathbf{B}^{s^T}\lambda, \quad s = 1,2,\ldots,n_s \tag{5.108}$$

unter der Nebenbedingung

$$\sum_{s=1}^{n_s}\mathbf{B}^s v^s = \mathbf{O} \tag{5.109}$$

überführt werden. Dabei beschreiben $\mathbf{K}$, $u$ und $f$ entsprechend die Steifigkeitsmatrix, den Verschiebungsvektor und die vorgeschriebenen äußeren Knotenlasten der Diskretisierung der Gebiete $\Omega^s$.
$\mathbf{B}$ ist eine boolsche Matrix, deren Komponenten den Zusammenhang zwischen benachbarten Teilgebieten herstellen. Bei Anwendung dieser Matrix auf Matrizen oder



Vektoren werden Koppelkomponenten der miteinander verbundenen Teilgebiete extrahiert und mit positivem oder negativen Vorzeichen gewichtet. Für eine Implementierung dieses Operators ist diese Eigenschaft auszunutzen, d.h. die Matrixoperation wird so implementiert, daß sie keine Fließkomma-Arithmetik erfordert.
Der Vektor der Lagrangefaktoren $\lambda$ repräsentiert die Gleichgewichtskräfte zwischen den Teilgebieten entlang der gemeinsamen Gebietsberandungen.
Die Methode läßt sich anschaulich auch so erklären, daß für ein zerlegtes Gebiet, für welches die Lösung der Teilprobleme bestimmt wurde, die Fehlerrandlasten zur Sicherung der Kontinuitätsbedingung minimiert werden. Die Unbekannten der Methode sind somit keine Verschiebungen, sondern Kräfte. Die mathematisch hergeleiteten Größen haben einen direkten mechanischen Inhalt als Steifigkeit bzw. Verschiebung und Belastung.
In praktischen Fällen führt die Gebietszerlegung zu einer oder mehreren Teilgebieten, die als freie Teilgebiete nicht genügend vorgeschriebene Randbedingungen besitzen, um die Starrkörperverschiebungen des Teilgebietes zu behindern. Die Steifigkeitsmatrix dieser Teilgebiete ist singulär und die Lösung der Gleichgewichtsbedingung ist nur eingeschränkt möglich, indem eine generalisierte Inverse $\mathbf{K}^{s+}$ der Steifigkeitsmatrix $\mathbf{K}^s$ und eine Linearkombination $\alpha$ der Basis für den Kern $\mathbf{R}^s$ der Steifigkeitsmatrix (den linear abhängigen Teil von $\mathbf{K}^s$) eingeführt werden.
Für $\mathbf{K}^{s+}$ sei die Bedingung

$$u^s = \mathbf{K}^{s+}(f^s - \mathbf{B}^{s^T}\lambda) + \mathbf{R}^s\alpha^s \tag{5.110}$$

$$\mathbf{K}^s\mathbf{K}^{s+}\mathbf{K}^s = \mathbf{K}^s \tag{5.111}$$

erfüllt, d.h. $\mathbf{K}^{s+}$ ist eine Pseudoinverse von $\mathbf{K}^s$. Wir haben mit (5.110) daher nur Bestimmungsgleichungen für den regulären Anteil der Matrix $\mathbf{K}^s$.
Aus der Tatsache, daß Starrkörperverschiebungen keine innere Energie induzieren, können die noch fehlenden Gleichungen zur Lösung von (5.110) bereitgestellt werden

$$\mathbf{R}^{s^T} = \mathbf{K}^s u^s = \mathbf{R}^{s^T}(f^s - \mathbf{B}^{s^T}\lambda) = \mathbf{O}. \tag{5.112}$$

Die Pseudoinverse $\mathbf{K}^{s^+}$ und der Kern $\mathbf{R}^s$ müssen nicht explizit berechnet werden. Für einen gewählten Vektor $v$ können das Matrix-Vektor-Produkt $\mathbf{K}^{s^+}v$ und die Starrkörperverschiebungen $\mathbf{R}^s$ mit den Herleitungen in Abschnitt 5.6.5 bereitgestellt werden.
Das Gleichungssystem zur Bestimmung der Lagrangefaktoren $\lambda$ und der Linearkombinationen der Kerne der Teilgebietsmatrizen $\alpha$ erhält unter Anwendung von (5.108), (5.109) und (5.110) sowie nach einigen Umformungen die folgende Gestalt

$$\begin{pmatrix} \mathbf{F}_I & -\mathbf{G}_I \\ -\mathbf{G}_I^T & \mathbf{O} \end{pmatrix}\begin{pmatrix} \lambda \\ \alpha \end{pmatrix} = \begin{pmatrix} d \\ -e \end{pmatrix} \tag{5.113}$$

Dabei ist $F_I$ die gewichtete Summe der Koppelanteile über die Inversen bzw. Pseudoinversen der Teilgebiete

$$\mathbf{F}_I = \sum_{s=1}^{n_s}\mathbf{B}^s\mathbf{K}^{s^+}\mathbf{B}^{s^T} \tag{5.114}$$

$\mathbf{G}_I$ beschreibt für die $N_f$ freien Teilgebiete die Basen der Kerne der Teilgebietsmatrizen

$$\mathbf{G}_I = (\mathbf{B}^1\mathbf{R}^1, \mathbf{B}^2\mathbf{R}^2, \ldots, \mathbf{B}^{N_f}\mathbf{R}^{N_f}) \tag{5.115}$$

$\alpha$ ist der Vektor der Linearkombinationen der Kerne dieser Teilgebietsmatrizen und $\lambda$ ist der Vektor der Lagrangefaktoren. Der Vektor $d$ beschreibt die modifizierte Rechte Seite für alle Teilgebiete mit

$$d = \sum_{s=1}^{n_s}\mathbf{B}^s\mathbf{K}^{s^+}f^s \tag{5.116}$$

und der Vektor $e$ beschreibt für alle freien Teilgebiete das Matrix-Vektor-Produkt

$$e^s = \mathbf{R}^{s^T} f^s \tag{5.117}$$

Im allgemeinen ist das Gleichungssystem (5.113) indefinit. Es ist mit einer parallelisierten Lösungstechnik darüberhinaus nicht möglich, die Matrix $\mathbf{F}_I$ explizit aufzubauen. Ein direkter Löser scheidet damit für die Lösung des Koppelproblems aus. Als Alternative bietet sich ein CG-Verfahren an, daß die Choleskyfaktoren der Teilgebietsmatrizen lediglich in einem Matrix-Vektor-Produkt benötigt und deshalb parallelisiert bereitgestellt werden kann. Dieser modifizierte CG-Algorithmus, der wegen der möglichen Singularitäten für einzelne Teilgebiete erweitert werden muß, wird im Abschnitt 5.6.2 beschrieben.

### 5.6.2 CPG-Verfahren

Aus der Symmetrie des Operators $\mathbf{F}_I$ folgt, daß das Koppelproblem (5.113) auch wie folgt formuliert werden kann:
Gesucht wird das Minimum

$$min\Phi(\lambda) = \frac{1}{2}\lambda^T \mathbf{F}_I \lambda - \lambda^T(\mathbf{G}_I\alpha + d) \tag{5.118}$$

unter der Restriktion

$$\mathbf{G}_I^T \lambda = e \tag{5.119}$$

Darüberhinaus ist $\mathbf{F}_I$ zumindestens positiv semi-definit (Beweis siehe z.B. [61]). Deshalb kann dieses Minimierungsproblem mit einem speziellen CG-Verfahren gelöst werden. Unter Anwendung von (5.118) und (5.119) wurde in [60] dieses Verfahren der konjugierten projizierten Gradienten (engl. *conjugate projected gradients*- CPG-Verfahren) beschrieben, das in jedem Iterationsschritt mit einer geeigneten Projektion $\mathbf{P}$ alle Komponenten von $\lambda$ extrahiert, die nicht die Restriktion (5.119)erfüllen. Angenommen, ein Vektor $\lambda_0$ (s.a. 5.6.6) erfüllt die Restriktion (5.119). Dann muß für jedes $\lambda_k | k > 0$ die Bedingung $\mathbf{G}_I^T \lambda_k = 0$ erfüllt sein. Gilt die Zerlegung

$$\lambda_k = \lambda_k^1 + \lambda_k^2 \tag{5.120}$$

mit

$$\mathbf{G}_I^T \lambda_k^1 = \mathbf{O}, \quad \mathbf{G}_I^T \lambda_k^2 \neq \mathbf{O} \tag{5.121}$$

dann ist der Operator $\mathbf{P}$ ein geeigneter Projektor, der $\lambda_k$ auf den Kern von $\mathbf{G}_I^K$ abbildet:

$$\lambda_k \mapsto \mathbf{P}\lambda_k = \lambda_k^1 = \lambda_k - \lambda_k^2 \tag{5.122}$$

Der Operator $\mathbf{P}$ ist damit noch nicht eindeutig bestimmt, weil die Projektionsrichtung nicht bestimmt ist. Die Dimension des Raumes des Restriktionsoperators ist aber bestimmt mit

$$\mathbf{R}^{n_k} = Ker(\mathbf{G}_I^T) \oplus Ker(\mathbf{G}_I^T)^\perp = Ker(\mathbf{G}_I^T) \oplus Range(\mathbf{G}_I) \tag{5.123}$$

Damit gilt $\lambda_k^s \in Range(\mathbf{G}_I)$ und $P$ wird ein Orthogonalprojektor auf $Ker(\mathbf{G}_I^T)$.
Für einen symmetrisch posititiv definiten Operator $\mathbf{Q}$ gilt, $\mathbf{G}_I^T\mathbf{Q}\mathbf{G}_I$ ist nicht singulär und damit auch $\mathbf{G}_I^T(\mathbf{Q}\mathbf{G}_I\eta) \neq \mathbf{O}$, für beliebige Vektoren $\eta \neq 0$. Dann gilt für den Raum des Restriktionsoperators

$$\mathbf{R}^{n_k} = Ker(\mathbf{G}_I^T) \oplus Range(\mathbf{Q}\mathbf{G}_I) \tag{5.124}$$

für beliebige, symmetrisch positiv definite Matrizen $\mathbf{Q}$. Die entsprechende Zerlegung von $\lambda_k$ ist

$$\lambda_k = \lambda_k^1 + \mathbf{Q}\mathbf{G}_I\eta_k \tag{5.125}$$



Daraus läßt sich nun $\mathbf{P}$ in Abhängigkeit von $\mathbf{Q}$ bestimmen. Zunächst wird diese Gleichung von links mit $\mathbf{G}_I^T$ multipliziert und unter Ausnutzung der Restriktion (5.121) bestimmt mit

$$\mathbf{G}_I^T\lambda_k = \mathbf{G}_I^T\lambda_k^1 + \mathbf{G}_I^T\mathbf{Q}\mathbf{G}_I\eta_k \mapsto \eta_k = (\mathbf{G}_I^T\mathbf{Q}\mathbf{G}_I)^{-1}\mathbf{G}_I^T\lambda_k \tag{5.126}$$

Damit kann dann eine Bestimmungsgleichung für $\lambda_k^1$ gefunden werden

$$\lambda_k = \lambda_k^1 + \mathbf{Q}\mathbf{G}_I(\mathbf{G}_I^T\mathbf{Q}\mathbf{G}_I)^{-1}\mathbf{G}_I^T\lambda_k \mapsto \lambda_k^1 = (\mathbf{E} - \mathbf{Q}\mathbf{G}_I(\mathbf{G}_I^T\mathbf{Q}\mathbf{G}_I)^{-1}\mathbf{G}_I^T)\lambda_k \tag{5.127}$$

die in (5.122) eingesetzt, einen Projektor $\mathbf{P}$ als Funktion der Matrix $\mathbf{Q}$ bestimmt

$$\mathbf{P}(\mathbf{Q}) = \mathbf{E} - \mathbf{Q}\mathbf{G}_I(\mathbf{G}_I^T\mathbf{Q}\mathbf{G}_I)^{-1}\mathbf{G}_I^T \tag{5.128}$$

Wählt man z.B. $\mathbf{Q} = \mathbf{E}$, so erhält man einen numerisch effizienten, orthogonalen Projektor

$$\mathbf{P} = \mathbf{E} - \mathbf{G}_I(\mathbf{G}_I^T\mathbf{G}_I)^{-1}\mathbf{G}_I^T \tag{5.129}$$

Das zum CG-Verfahren äquivalente CPG-Verfahren erhält man jetzt mit Anwendung von (5.38) auf den projizierten Vektor $\mathbf{P}r$. Es liefert die Lösung des Koppelproblems in den unbekannten Linearkombinationen der Basen der singulären Teilsystemmatrizen und den Lagrangefaktoren und hat dann folgende Gestalt:

$$\begin{aligned}
\lambda_0 &= \mathbf{G}_I(\mathbf{G}_I^T\mathbf{G}_I)^{-1}[f_1^T\mathbf{R}_1, f_2^T\mathbf{R}_2, \ldots, f_{n_I}^T\mathbf{R}_{n_I}] \\
r_0 &= \mathbf{F}_I\lambda_0 - \bar{f} \\
d_0 &= -\mathbf{P}r_0 \\
\gamma_0 &= d_0^T d_0 \\
\text{Iteration für} \quad k &= 1, 2, \ldots \\
u &= \mathbf{F}_I d_{k-1} \\
\alpha &= \frac{\gamma_0}{d_{k-1}^T u} \\
\lambda_k &= \lambda_{k-1} + \alpha d_{k-1} \\
r_k &= r_{k-1} + \alpha u \\
u &= \mathbf{P}r_k \\
\gamma_k &= u^T u \quad \mapsto \quad \text{Abbruchkriterium} \quad \gamma_k < \epsilon \\
\beta &= \frac{\gamma_k}{\gamma_{k-1}} \\
d_k &= -u + \beta d_k
\end{aligned} \tag{5.130}$$

Indizierte Größen werden in mehr als einem Iterationsdurchlauf benötigt, nicht indizierte Größen sind temporäre Hilfsdaten innerhalb einer Iteration. Sind die Lagrangefaktoren $\lambda$ bestimmt, kann der Vektor der Linearkombinationen $\alpha$ für die freien Teilgebiete mit

$$\alpha = \mathbf{G}_I^{-1}(\mathbf{F}_I\lambda - d) \tag{5.131}$$

berechnet werden.
Das Matrix-Vektor-Produkts der Form $u = \mathbf{F}_I d$ kann vollständig unabhängig in jedem Teilgebiet berechnet werden

$$u = \mathbf{F}_I d = \sum_{s=1}^{N_s} \mathbf{B}^s \mathbf{K}^{s^+} \mathbf{B}^{s^T} d \tag{5.132}$$

Der resultierende Vektor $u$ ist ein additiv verteilter Vektor, der in einem Skalarprodukt mit einem absolut verteilten Vektor $d_{k-1}^T u$ und zur Aktualisierung des additiv verteilten Vektors $r_k$ benötigt wird.

Die Matrix-Vektor-Produkte der Form $u = \mathbf{P}r$ können in den drei Schritten

$$\bar{r}^s = \mathbf{R}^{s^T}\mathbf{B}^{s^T} r, \quad s = 1, \ldots N_f$$
$$\text{Lösen von} \quad \mathbf{G}_I \mathbf{G}_I^T [\bar{r}^1 \ldots \bar{r}^{N_f}]^T = [\bar{r}^1 \ldots \bar{r}^{N_f}]^T$$
$$u = r - \sum_{s=1}^{n_f} \mathbf{B}^s \mathbf{R}^s \bar{r}^s$$

berechnet werden. Bis auf Schritt zwei können diese Berechnungen parallel in allen Teilgebieten durchgeführt werden. Für kleine Prozessorzahlen ($p \leq 16$) kann $\mathbf{G}_I\mathbf{G}_I^T$ vor Beginn der Iteration faktorisiert bereitgestellt werden. Für größere Prozessorzahlen ist die iterative, parallele Lösung dieses Gleichungssystems angezeigt. Der resultierende Vektor $u$ ist ein absolut verteilter Vektor, der nachfolgend im Skalarprodukt mit einem additiv verteilten Vektor und zur Aktualisierung des absolut verteilten Vektors $d_k$ benötigt wird.

Im Vergleich zur primären Gebietszerlegung und dem dort dargestellten, parallelen CG-Algorithmus wird deutlich, daß bis auf die Projektionsschritte für die duale Gebietszerlegung die gleichen elementaren Operationen benötigt werden. Die Operationen vereinfachen sich noch, weil die Unterscheidung in die Koppel- und Innenfreiheitsgrade im Matrix-Vektor-Produkt entfällt. Das Koppelproblem wird vollständig mit dem Vektor der Lagrange-Multiplikatoren abgebildet, für den zusätzlicher Speicherplatz erforderlich ist.

### 5.6.3 PCPG-Verfahren

Für einen geeigneten Vorkonditionierer $\mathbf{C}^{-1}$ (s. Abs. 5.6.4) leitet sich das vorkonditionierte CPG-Verfahren (PCPG-Verfahren) aus dem CPG-Verfahren wie folgt ab

$$\begin{aligned}
\lambda_0 &= \mathbf{G}_I(\mathbf{G}_I^T\mathbf{G}_I)^{-1}[f_1^T\mathbf{R}_1, f_2^T\mathbf{R}_2, \ldots, f_{n_f}^T\mathbf{R}_{n_f}] \\
r_0 &= \mathbf{F}_I\lambda_0 - \bar{f} \\
u &= -\mathbf{P}r_0 \\
d_0 &= \mathbf{P}\mathbf{C}^{-1}u \\
\gamma_0 &= d_0^T d_0 \\
\text{Iteration für} \quad k &= 1, 2, \ldots \\
u &= \mathbf{F}_I d_{k-1} \\
\alpha &= \frac{\gamma_0}{d_{k-1}^T u} \\
\lambda_k &= \lambda_{k-1} + \alpha d_{k-1} \\
r_k &= r_{k-1} + \alpha u \\
u &= \mathbf{P}r_k \\
p &= \mathbf{P}\mathbf{C}^{-1}u \\
\gamma_k &= p^T u \quad \mapsto \quad \text{Abbruchkriterium} \quad \gamma_k < \epsilon \\
\beta &= \frac{\gamma_k}{\gamma_{k-1}} \\
d_k &= -p + \beta d_k
\end{aligned} \tag{5.133}$$

Dabei werden der Projektions- und Rückprojektionsschritt benötigt, um die Symmetrie des Operators zu erhalten.



### 5.6.4 Vorkonditionierung

Die duale Gebietszerlegungsmethode kann wie folgt interpretiert werden: In jedem Iterationsschritt wird für einen vorgegebenen Belastungsvektor $\bar{f}^k$ auf den Koppelrändern jedes Teilgebietes die Änderung in den Verschiebungen des Teilgebietsinneren berechnet. Das inverse Problem ist die Berechnung der erforderlichen Belastung eines Teilgebietsinneren für ein vorgegebenes Verschiebungsfeld des Teilgebietskoppelrandes. Dieses primäre Dirichletproblem kann wie folgt geschrieben werden:

$$\begin{pmatrix} \mathbf{K}^{ll^s} & \mathbf{K}^{lk^s} \\ \mathbf{O} & \mathbf{E} \end{pmatrix} \begin{pmatrix} \bar{v}^l \\ \bar{v}^k \end{pmatrix} = \begin{pmatrix} \mathbf{O} \\ \bar{f}^k \end{pmatrix} \tag{5.134}$$

Die Lösung dieses Problems ist gegeben mit

$$\bar{v}^l = -\mathbf{K}^{ll^{-1}}\mathbf{K}^{lk}\bar{v}^k \tag{5.135}$$

Zugehörige Kräfte erhält man mit dem Matrix-Vektor-Produkt wie folgt:

$$\begin{pmatrix} \mathbf{K}^{ll^s} & \mathbf{K}^{lk^s} \\ \mathbf{K}^{kl^s} & \mathbf{K}^{kk^s} \end{pmatrix} \begin{pmatrix} \bar{v}^l \\ \bar{v}^k \end{pmatrix} = \begin{pmatrix} \mathbf{O} \\ (\mathbf{K}^{kk^s} - \mathbf{K}^{kl^s}\mathbf{K}^{ll^{s^{-1}}}\mathbf{K}^{lk^s})\bar{v}^k \end{pmatrix} \tag{5.136}$$

Ein Vorkonditionierer auf Basis dieser Schurkomplementformulierung entsprechend der primären Gebietszerlegung ergibt sich zu

$$\begin{aligned}
\bar{\mathbf{F}}_I^{\mathbf{S}^{-1}} &= \sum_s \mathbf{B}^s \begin{pmatrix} \mathbf{O} & O \\ \mathbf{O} & (\mathbf{K}^{kk^s} - \mathbf{K}^{kl^s}\mathbf{K}^{ll^{s^{-1}}}\mathbf{K}^{lk^s}) \end{pmatrix} B^{s^T} \\
&= \sum_s \mathbf{B}^s \begin{pmatrix} \mathbf{O} & \mathbf{O} \\ \mathbf{O} & \mathbf{S}^s \end{pmatrix} \mathbf{B}^{s^T}
\end{aligned} \tag{5.137}$$

Farhat schlägt alternativ in [60] vor, den in (5.137) benutzten Vorkonditionierer zu vereinfachen. Ein reduzierter Vorkonditionierer in der Form

$$\bar{\mathbf{F}}_I^{\mathbf{L}^{-1}} = \sum_s \mathbf{B}^s \begin{pmatrix} \mathbf{O} & \mathbf{O} \\ \mathbf{O} & \mathbf{K}^{kk^s} \end{pmatrix} \mathbf{B}^{s^T} \tag{5.138}$$

berücksichtigt lediglich die den Koppelknoten zugeordneten Steifigkeiten. Der Vorkonditionierer erfordert lediglich den Aufwand eines Matrix-Vektor-Produkts für das Koppelproblem, das bereits vorhanden ist und keinen zusätzlichen Speicher. Bei mechanischer Interpretation dieser Vorkonditionierung werden damit Kräfte bestimmt, die eine vorgegebene Verschiebung des Koppelrandes erzeugen. Die Randbedingungen sind so gewählt, daß dabei nur der Koppelrand selbst beweglich ist.

### 5.6.5 Behandlung der singulären Teilsysteme

Die Ausführungen des folgenden Abschnitts beziehen sich auf die Matrizen und Vektoren eines Teilgebietes $s$. Zur Vereinfachung der Schreibweise wird deshalb der hochgestellte Index $s$ weggelassen. Der Algorithmus wurde im Zusammenhang mit der dualen Gebietszerlegungsmethode von Farhat und Roux in [60], Anhang 1 dargestellt.

Die Matrix $\mathbf{K}$ läßt sich blockweise in einen linear unabhängigen und in einen redundanten Anteil zerlegen, die im folgenden mit $u$ und $r$ indiziert werden

$$\mathbf{K} = \begin{pmatrix} \mathbf{K}_{uu}\mathbf{K}_{ur} \\ \mathbf{K}_{ru}\mathbf{K}_{rr} \end{pmatrix} \tag{5.139}$$

Dabei ist $\mathbf{K}_{uu}$ regulär mit Rang $n^l + n^k - n^r$ (Anzahl der lokalen ($n^l$) und Koppelfreiheitsgrade ($n^k$) weniger Anzahl der redundanten ($n^r$) - nicht behinderten - Starrkörperverschiebungen). Im dreidimensionalen Fall gilt $n^r \leq 6$, im zweidimensionalen Anwendungsfall gilt $n^r \leq 3$. Es sei eine Matrix $\mathbf{R}$ definiert mit

$$\mathbf{R} = \begin{pmatrix} \mathbf{K}_{uu}^{-1}\mathbf{K}_{ur} \\ \mathbf{E}_{n^r} \end{pmatrix} \tag{5.140}$$

wobei $\mathbf{E}_{n^r} \in R^{n^r \times n^r}$ eine Einheitsmatrix ist. Dann erfüllt $\mathbf{R}$ die Gleichung

$$\mathbf{K}\mathbf{R} = O \tag{5.141}$$

und ist eine Basis für den Kern von $\mathbf{K}$. Die Blockzerlegung der singulären Matrix $\mathbf{K}$ garantiert, daß

$$\mathbf{K}_{rr} = \mathbf{K}_{ru}\mathbf{K}_{uu}^{-1}\mathbf{K}_{ur} \tag{5.142}$$

gilt. Daraus folgt, daß $\mathbf{K}^+$, definiert durch

$$\mathbf{K}^+ = \begin{pmatrix} \mathbf{K}_{uu}^{-1} & \mathbf{O} \\ \mathbf{O} & \mathbf{O} \end{pmatrix} \tag{5.143}$$

eine Pseudoinverse von $\mathbf{K}$ darstellt. Eine Lösung der Form $\mathbf{K}^+ f$ kann deshalb wie folgt formuliert werden

$$u = \mathbf{K}^+ f = \begin{pmatrix} \mathbf{K}_{uu}^{-1} f_u \\ \mathbf{O} \end{pmatrix} \tag{5.144}$$

Natürlich kann in der Regel das Gleichungssystem $\mathbf{K}$ nicht in der angegebenen Form umsortiert werden.
Für eine praktische Implementation ist das Vorgehen deshalb wie folgt. Wird eine Nullpivot bei der Faktorisierung von $\mathbf{K}_{uu}$ gefunden, dann korrespondiert die zugehörige Zeile bzw. Spalte zu einer Zeile bzw. Spalte des Kerns von $\mathbf{K}$. Das Pivotelement wird zu Eins gesetzt und die reduzierte Zeile bzw. Spalte wird als zusätzliche Rechte Seite behandelt. Die Nebendiagonaleinträge dieser Zeile werden zu Null gesetzt. Das entspricht einer Vorwärtsreduktion mit $\mathbf{K}^+ f$ als Rechter Seite. Die nach vollständiger Faktorisierung verbleibenden, nichtmarkierten Zeilen und Spalten von $\mathbf{K}$ beschreiben die Inverse der reguläre Matrix $\mathbf{K}_{uu}$ bzw. die gewünschte Pseudoinverse von $\mathbf{K}$. Die Rückwärtssubstitution wird so modifiziert, daß sie die zusätzlichen $n^r$ Rechten Seiten berücksichtigt und die Korrektur

$$u_u = -\mathbf{K}_{uu}^{-1}\mathbf{K}_{ur}\,u_r \tag{5.145}$$

durchgeführt wird.

### 5.6.6 Bestimmung der Startvektoren der singulären Teilsysteme

Nur für die singulären Teilsysteme ist der Startvektor der Lagrangefaktoren ungleich dem Nullvektor. Aus (5.113) läßt sich ablesen, daß für diese Teilsysteme die Gleichung

$$\mathbf{G}_I^{s^T}\lambda_0 = \mathbf{R}^{s^T} f^s \tag{5.146}$$

gilt. Die Matrix $\mathbf{R}^s$ beschreibt den Zusammenhang zwischen allen $n^l + n^k$ Freiheitsgraden des Teilsystems $s$ und seinen $n^r$ redundanten Freiheitsgraden und hat die Dimension $(n^l + n^k) \times n^r$. Die Matrix $\mathbf{G}_I^s$ ist die Projektion von $\mathbf{R}^s$ auf den Koppelrand. Zur Vereinfachung der Schreibweise wird im folgenden der hochgestellte Index $s$ zur Kennzeichnung des Teilgebietes weggelassen.



Es sei

$$\lambda_0 = \mathbf{G}_I \mu_0 \tag{5.147}$$

dann wird aus (5.146)

$$\mathbf{G}_I^T \mathbf{G}_I \mu_0 = \mathbf{R}^T f \tag{5.148}$$

mit der Lösung

$$\mu_0 = (\mathbf{G}_I^T \mathbf{G}_I)^{-1} \mathbf{R}^T f \tag{5.149}$$

Ein Startvektor, der die Restriktion (5.146) erfüllt, kann deshalb wie folgt berechnet werden

$$\lambda_0 = \mathbf{G}_I (\mathbf{G}_I^T \mathbf{G}_I)^{-1} \mathbf{R}^T f \tag{5.150}$$

Diese Berechnung kann parallel in jedem singulären Teilgebiet durchgeführt werden. Die Größe des zu faktorisierende Gleichungssystem ist abhängig von der Anzahl der möglichen Starrkörperverschiebungen (Dimension des Kerns von $K^s$), d.h. maximal 6 Unbekannte.

## 5.7 Zusammenfassung

Tabelle 5.2: *Gegenüberstellung von primärer und dualer Gebietszerlegung*

| $\mathbf{K}v = f \mapsto \sum_{p=1}^{n} \mathbf{B}_p^T \mathbf{K}_p \mathbf{B}_p v = \sum_{p=1}^{n} \mathbf{B}_p f_p$ | |
|---|---|
| primäre Methode | duale Methode |
| Matrixsplitting<br>$\begin{pmatrix} \mathbf{K}^{ii} & \mathbf{K}^{ic} \\ \mathbf{K}^{ci} & \mathbf{K}^{cc} \end{pmatrix} \begin{pmatrix} v^i \\ v^c \end{pmatrix} = \begin{pmatrix} f^i \\ f^c \end{pmatrix}$ | lokale Problemformulierung<br>$\mathbf{K}_s v_s = f_s$ |
| Koppelproblem<br>für *gekoppelte* Teilgebiete | Lagrange-Multiplikatoren<br>für *getrennte* Teilgebiete |
| $\mathbf{S} = \sum_{p=1}^{n} (\mathbf{K}_p^{cc} - \mathbf{K}_p^{ci}\mathbf{K}_p^{ii^{-1}}\mathbf{K}_p^{ic})$<br>$\bar{f} = \sum_{p=1}^{n} (f_p^c - \mathbf{K}_p^{ci}\mathbf{K}_p^{ii^{-1}} f_p^i$<br>$\mathbf{S}v^c = \bar{f}$ | $\mathbf{K}_p v_p = f_p - \mathbf{B}_p \lambda$<br>$\sum_{p=1}^{n} \mathbf{B}_p v_p = 0$ |
| zusätzliches Grobproblem | natürliches Grobproblem |

Für die Lösung von dünnbesetzten, symmetrischen, positiv definiten Gleichungssystemen sind grundsätzlich direkte Verfahren und iterative Verfahren geeignet. Direkte Verfahren erfordern einen hohen Speicherplatzaufwand für die faktorisierte Matrix. Dieser Speicherplatz ist problemabhängig und nur mit heuristischen Sortierverfahren reduzierbar. Der Rechenaufwand hängt von der Problemgröße $N$ mit $O(N^2)$ und vom Speicherplatzbedarf ab. Die Kriterien zur Sortierung zur Minimierung des Speicherplatzbedarfs konkurrieren mit Kriterien zur Sortierung für die

Parallelisierung der direkten Löser. Direkte Löser sind nur für ausgewählte Problemstellungen skalierbar. Besseres Parallelisierungspotential haben iterative Löser. Der Speicherplatzaufwand ist für die Vorkonditionierung und Hilfsvektoren signifikant. Der Rechenaufwand ist linear von der Größe des Gleichungssystems und der Anzahl der Iterationen abhängig.
Für die Parallelisierung interessant sind die Verfahren der Gebietszerlegung (Tabelle 5.2). Das primäre Verfahren erlaubt die Formulierung von Vorkonditionierern getrennt für die lokalen und das Koppelproblem. Skalierbare Lösungen erfordern die Einführung eines Grobproblems. Alternativ ist das duale Gebietszerlegungsverfahren mit dem globalen Problem der Lagrange-Multiplikatoren unmittelbar skalierbar. Beide Verfahren sind mit identischem algorithmischen Kernen zu implementieren.

# Kapitel 6

# Parallele Strukturmechanik

Die Anwendung des parallelen Programmierparadigmas auf Rechnersysteme mit verteiltem Speicher auf der Basis von Nachrichtenaustausch auf ein Beispiel der Strukturmechanik soll im folgenden für die lineare statische Analyse von Plattentragwerken beschrieben werden. Grundlegende Bausteine zur Anwendung sind zunächst die klassische Plattentheorie und ihre Umsetzung in ein rechnergestütztes Approximationsverfahren mit der Methode der finiten Elemente, die Partitionierung des finite Elementnetzes und die parallele Lösung des linearen Gleichungssystems. Probleme der Lastverteilung, der parallelen Netzadaption und der Lastanpassung werden in diesem Zusammenhang dargestellt. In einem Anwendungsbeispiel werden die Auswirkungen der Parallelverarbeitung auf die Softwarewerkzeuge demonstriert.

## 6.1 Diskrete Kirchhoff-Plattentheorie

Betrachtet man eine Platte im zweidimensionalen Gebiet $\Omega$, die hinreichend kinematisch bestimmt gelagert ist und deren statische Randbedingungen auf den Rändern $\Gamma_\sigma$ durch Linienmomente bzw. auf den Rändern $\Gamma_{\bar{\sigma}}$ durch Linienlasten vorgegeben sind, dann wird mit der Kirchhoff-Theorie vom Ebenbleiben der Plattenquerschnitte die Plattenverschiebung durch die Differentialgleichung vierter Ordnung

$$K \Delta\Delta w = q \tag{6.1}$$

beschrieben. Die Finite-Element Diskretisierung wird aus dem Funktional für die potentiellen Energie der Platte:

$$\hat{\Pi} = \hat{\Pi}_i + \hat{\Pi}_a \tag{6.2}$$

mit

$$\hat{\Pi}_i = \frac{1}{2}\int_\Omega \hat{\beta}_{\alpha|\beta} K^{\alpha\beta\gamma\delta} \hat{\beta}_{\gamma|\delta}\, d\Omega + \int_\Omega (\hat{w}_{,\alpha} + \hat{\beta}_\alpha)(K^{\alpha\beta\gamma\delta}\hat{\beta}_{\gamma|\delta})_{|\beta}\, d\Omega \tag{6.3}$$

$$\hat{\Pi}_a = -\int_\Omega \hat{w}\hat{q}\, d\Omega - \int_{\Gamma_\sigma} \hat{w}\bar{p}\, d\Gamma - \int_{\Gamma_\sigma} \hat{\beta}_\alpha \bar{m}^\alpha\, d\Gamma \tag{6.4}$$

durch ein Minimalprinzip mit

$$\delta\hat{\Pi}(\hat{w}, \hat{\beta}_\alpha) = 0 \tag{6.5}$$

gewonnen. Dabei ist $\hat{\Pi}_i$ die innere Energie und $\hat{\Pi}_a$ die Energie aus der Summe der äußeren Belastung, die als Funktionen der Weggrößen $\hat{w}$ und $\hat{\beta}_\alpha$ formuliert

wurden. Als äußere Belastungen werden die vorgeschriebenen Gebietslasten $\bar{q}$, die vorgeschriebenen Randlasten $\bar{p}$ und Randmomente $\bar{m}^{\alpha}$ berücksichtigt.
Eine konforme Finite-Element-Approximation mit einem DKQ-(*discrete Kirchhoff quadrilateral*) Element [22], entwickelt aus einem DKT-(*discrete Kirchhoff triangle*) Element [21], führt dann auf ein lineares Gleichungssystem der Form

$$\mathbf{K}v = f \tag{6.6}$$

mit der Steifigkeitsmatrix $\mathbf{K}$, den unbekannten Verschiebungen $v$ und der äußeren Belastung $f$.
Aus dem primären Ergebnis von 6.6, den Verschiebungen $v$, können durch Ableitung Schnittgrößen und andere sekundäre Ergebnisgrößen in ausgezeichneten Punkten, den Finite-Element-Knoten berechnet werden. Die Güte der Approximation wird mit geeigneten Fehlerfunktionen abgeschätzt. In einem adaptiven Prozeß kann mit geeigneter Anpassung des Finite-Element-Modells eine Lösung gewünschter Güte ermittelt werden. Einen typischen Ablauf in der linearen Strukturmechanik zeigt das Struktogramm in Abbildung 6.1.

### 6.1.1 Fehlerabschätzung und adaptive Netzanpassung

Die Charakterisierung der FEM als Näherungsverfahren mit Konvergenzeigenschaften impliziert, daß eine bessere Annäherung der Lösung an das wahre Verhalten des modellierten Problems nur durch Berechnung mit einer Unterteilung in mehr und damit kleinere finite Elemente oder durch Erhöhung des Polynomgrades der Ansatzfunktion erreicht werden kann.

#### 6.1.1.1 Adaptive Strategien

Im einzelnen werden folgende adaptive Strategien zur Steigerung der Approximationsgüte des Finite-Elemente-Modells unterschieden:

**h-Methode** selektives Verfeinern des Netzes durch Verringerung der Elementkantenlänge (Babuska [10], Zienkiewicz [121], [73]) ,

**p-Methode** Erhöhen des Grades der Ansatzfunktionen (Babuska, Szabo [12], [13], Rank [11], [24]),

**r-Methode** Verschieben der vorhandenen Knotenpunkte in die Bereiche, in denen Schwierigkeiten bei der Approximation des Lösungsverlaufes auftreten - Optimierung des vorhandenen Netzes oder

**d-Methode** Verändern der Dimensionalität des Modells, z.B. Nutzung von hybriden oder gemischten Elementen, bei denen Ansatzfunktionen für diejenigen Größen eingeführt werden, für deren Ermittlung bei den Ausgangsmodellen Differentiationen erforderlich sind, oder Übergang zur Modellierung mit 3D-Elementen in Bereichen mit dreidimensionaler Tragwirkung (Jensen [107], Stein [196])

Als Alternative zur h-Methode ist eine homogene Verfeinerung möglich. Bei Problemen der Strukturmechanik ist die notwendige Auflösung der Diskretisierung oft von der filigranen Geometrie der Teilstrukturen bestimmt. Im folgenden sollen deshalb nur die in einem Prototypprogramm [160] implementierten Verfeinerungsstrategien nach der h- und r-Methode betrachtet werden.
Während eine adaptive Netzanpassung mit der r-Methode einen geeigneten, dichtefunktionsgesteuerten Netzgenerator erfordert (z.B. [160]), sind für die adaptive h-Methode zwei Strategien möglich:



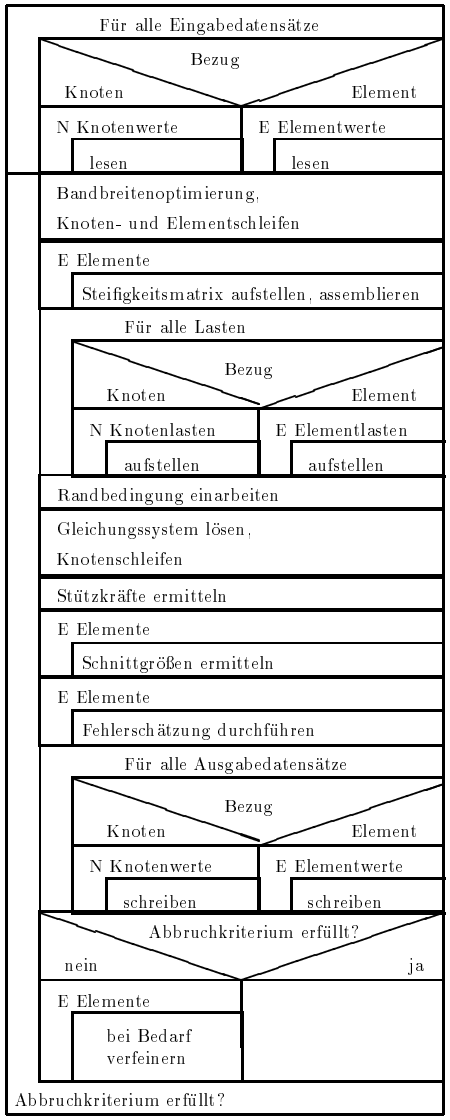


Abbildung 6.1: *Struktogramm adaptive FEM*

- konforme Verfeinerung mit Übergangselemente oder
- nicht-konforme Verfeinerung mit Elementen mit zusätzlichen Seitenmittenknoten.

Zusätzliche Knoten können durch Erweiterung des Funktionals, durch Strafterme (Penalty-Verfahren) oder auch topologisch berücksichtigt werden. [160] benutzt eine spezielle Erweiterung des Funktionals. Dadurch vermeidet man künstliche Versteifungen des verfeinerten Gebietes durch Strafterme, stark verzerrte Elemente oder den Einsatz von Dreieckelementen. Andererseits wirft die nichtkonforme Verfeinerung neue Fragestellungen bei der Implementierung der topologischen Funktionen zur nichtkonformen Netzverfeinerung mit irregulären Elementen und bei der Parallelisierung, hier besonders bei der Verfeinerung über die Prozessorgrenzen auf.
Die Anzahl der finiten Knoten hat unmittelbar Einfluß auf die Größe des zu bearbeitenden Gleichungssytems. Für große Probleme ist deshalb die Parallelisierung der Berechnung zum einen wegen des benötigten Speicherplatzes und zum anderen wegen der benötigten Rechenleistung erforderlich.

#### 6.1.1.2 Fehlerschätzung

Mit der h-Methode erfolgt die Verfeinerung am besten unter Auswertung von apriori Wissen über Problemzonen (Singularitäten in der Geometrie und in der Belastung, entsprechend Singularitäten in der Lösung) oder mit einer geeigneten aposteriori Fehlerabschätzung.
Die Differenz zwischen exakter Lösung und approximierter Finite-Element-Lösung

$$\bar{w} = w - \hat{w}, \quad \bar{\beta}_\alpha = \beta_\alpha - \hat{\beta}_\alpha \tag{6.7}$$

gibt den Approximationsfehler an. Die Energie des Fehlers berechnet sich aus

$$\bar{E}(\bar{\beta}_\alpha) = \frac{1}{2} \int_\Omega \bar{\beta}_{\alpha|\beta} K^{\alpha\beta\gamma\delta} \bar{\beta}_{\gamma|\delta} \, d\Omega \tag{6.8}$$

Eine Zahl $\eta$, die den Diskretisierungsfehler im Gesamtgebiet $\Omega$ abschätzt, heißt *Fehlerestimator*. Eine Größe $\lambda_i$, die den Beitrag eines Elements $i$ zum Fehler im Gesamtgebiet abschätzt, heißt Fehlerindikator. Es gilt:

$$\bar{E}(\bar{\beta}_{\alpha|\beta}) \leq \eta^2 = \sum_i \lambda_i^2 \tag{6.9}$$

#### 6.1.1.3 Fehlerindikatoren

Der lokale Fehlerindikator kann mit verschiedenen Strategien ermittelt werden, die ursprünglich für elliptische Randwertprobleme zweiter Ordnung entwickelt wurden. In der demonstrierten Anwendung werden die folgenden Fehlerindikatoren benutzt:
**Residual-Typ** nach Babuska et.al. [13], [73] und [121]:

$$\lambda_i^2 = C_1 h_i^2 \int_\Omega \hat{r}^2 d\Omega + C_2 h_i^2 \int_{\Gamma_\sigma} \hat{J}_m^2 d\Gamma + C_3 h_i^2 \int_{\Gamma_\sigma} \hat{J}_q^2 d\Gamma \tag{6.10}$$

$\hat{r}$ bezeichnet dabei den Residuenvektor, welcher beim Einsetzen der Finite-Elemente-Lösung in die Differentialgleichung des Randwertproblems entsteht und als Maß der Nichterfüllung der Gleichgewichtsbedingungen interpretiert werden kann. $\hat{J}_m$ und $\hat{J}_q$ stellen den Vektor der Momentensprünge an der Oberfläche bzw. den Vektor der Querkraftsprünge an den Rändern des Elements dar.



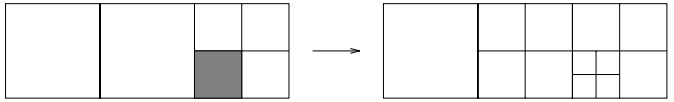

Abbildung 6.2: *Rekursive Verfeinerung im Fünfknotenelement*

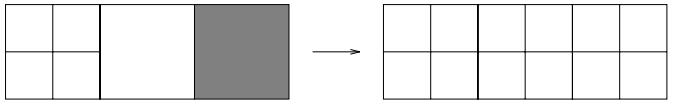

Abbildung 6.3: *Rekursiwer Verfeinerung im Sechsknotenelement*

Mit $h_i$ wird eine charakteristische Elementabmessung gekennzeichnet und $C_*$ sind unbekannte Konstanten. Sie sind von den globalen Eigenschaften des Randwertproblems, dessen Diskretisierung und Materialeigenschaften abhängig.
**Recovery-Typ** nach Zhienkiewicz und Zhu [223], [224]:

$$\lambda_i^2 = \frac{1}{2} \int_{\Omega_i} \bar{\beta}_{\alpha|\beta} K^{\alpha\beta\gamma\delta} \bar{\beta}_{\gamma|\delta} \, d\Omega \tag{6.11}$$

Dabei werden aus den Ableitungen der Verdrehungen mit einer Polynominterpolation höherer Ordnung die verbesserten Approximationen ermittelt. Die Freiwerte dieser Interpolation können z.B. aus der Forderung bestimmt werden, daß das Quadrat der Differenz zwischen der Finite-Element- und der geglätteten Approximation im Gesamtgebiet minimiert wird ($L_2$-Projektion). Superkonvergenzeffekte werden ausgenutzt oder verbesserte Knotenfreiwerte werden einfach aus der Mittelung der Ableitungen der am Knoten anliegenden Elemente erreicht. Die Differenz zwischen der geglätteten Lösung und der Finite-Element-Approximation wird für die Berechnung elementbezogener Fehlerindikatoren verwendet:

$$\bar{\beta}_{\alpha|\beta} = \beta_{\alpha|\beta} - \hat{\beta}_{\alpha|\beta} \approx \hat{\hat{\beta}}_{\alpha|\beta} - \hat{\beta}_{\alpha|\beta} \tag{6.12}$$

**Patch-Typ** nach Meißner, Wibbeler und Olden [145], [160]:
Bei diesem Ansatz werden verbesserte Approximationslösungen durch die patchweise Auswertung lokaler Randwertprobleme erzeugt. Aus dem Gesamtsystem wird dabei ein Elementpatch herausgelöst und die Finite-Element-Schnittgrößen werden als äußere Belastungen an den Patch angetragen. Durch Netzverfeinerung oder durch eine Erhöhung der Ansatzordnung wird eine geglättete Approximation der Verdrehungsableitungen erreicht.

#### 6.1.1.4 Netzverfeinerung

Die Verfeinerung erfolgt nach der h-Methode. Sie wird so vorgenommen, daß alle Elemente $i$, deren Fehlerindikator $\lambda_i$ einen Grenzwert $\lambda_\epsilon$ überschreitet, verfeinert werden. Jedes Viereckelement wird dazu in vier Elemente zerlegt. Auf allen vier Elementkanten werden Seitenmittenknoten eingefügt. Die Nachbarelemente werden daraufhin überprüft, ob die weiter zu verfeinernde Kante bereits einen Seitenmittenknoten besitzt oder ob es sich um ein Fünfknoten-Übergangselement handelt.

Trifft eines dieser beiden Kriterien zu, ist das Nachbarelement ebenfalls zu verfeinern (rekursiver Aufruf der Verfeinerungsroutine). Beide Situationen sind in den Abbildungen 6.2 und 6.3 dargestellt.
Übergangselemente mit mehr als einem Seitenmittenknoten sind nicht vorgesehen. Ein Element mit sieben Knoten kann ohne Einfügen weiterer Knoten mit einem Schnitt über die einander gegenüberliegenden Seitenmittenknoten in zwei Elemente zerlegt werden. Ein Element besitzt dann vier, das andere fünf Knoten (Abbildung 6.4).

### 6.1.2 Parallelisierung

Bei der Parallelisierung des Algorithmus erweisen sich die mit Gebietszerlegung relativ einfach parallelisierbaren Knoten- und Elementschleifen als Vorteil. Allerdings verschließt sich der rechenintensivste Teil des Algorithmus - die Lösung des dünnbesetzten, linearen Gleichungssystems - einer offensichtlichen Parallelisierung. Hier ist es erforderlich, parallele direkte oder besser parallele iterative Verfahren einzusetzen. Diese Algorithmen müssen im Vergleich zur seriellen Lösung wesentlich erweiterte Funktionalität aufweisen. Die methodischen Grundlagen und eine Abschätzung der Effektivität und der Skalierbarkeit der eingesetzten Gleichungslöser sind in Abschnitt 5 angegeben.
Folgende, für die Parallelisierung wesentlichen Programmkomponenten werden im folgenden vorgestellt:

- Gebietszerlegung (Partitionierung) und Abbildung einer gefundenen Partitionierung auf ein Multiprozessornetzwerk
- Implementation eines parallelen Gleichungslösers
- parallele, adaptive Netzverfeinerung

Schwerpunkt der Darstellung sind dabei Anwendungen aus der linearen Strukturmechanik, d.h. adaptive Berechnung von Plattentragwerken. Die parallelisierten Programmbausteine können aber auch für weitere Problemklassen eingesetzt werden (vgl. Abschnitt 4). Die Kopplung an die graphische Auswertung wird in Abschnitt 7 behandelt.

## 6.2 Partitionierung und Abbildung

In Tabelle 6.1 sind Ergebnisse der Partitionierung mit dem Programm Chaco Version 1.0 auf SPARC 20/51 unter Solaris 2.4 zusammengefaßt. Das in Abschnitt 6.5 weiter behandelte Plattenproblem mit 4480 Knoten und 4151 Elementen wurde in 4, 8 bzw. 16 Teilgebiete entsprechend der Hypercube-Dimension 2, 3 und 4 zerlegt. Zum Einsatz kamen dabei die Verfahren Multilevel-(ML)-, Spektral-(Spec)-,

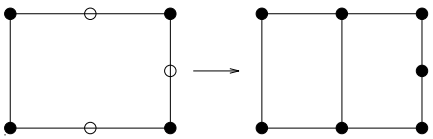

Abbildung 6.4: *Verfeinerung - Ersatz eines Siebenknotenelements*



Tabelle 6.1: *Partitionierung des Plattenproblems (Erläuterungen im Text)*

| Methode | dim | Zeit | | Partitionierung | | | |
|---|---|---|---|---|---|---|---|
| | | global [s] | lokal [s] | cuts | hops | bndy | adj |
| ML | 2 | 1.8 | 0.7 | 67 | 87 | 128 | 8 |
| Spec | 2 | 7.1 | 0.3 | 82 | 82 | 149 | 6 |
| Iner | 2 | 1.3 | 0.5 | 93 | 93 | 172 | 6 |
| Linear | 2 | 0.8 | 0.4 | 123 | 166 | 229 | 6 |
| ML | 3 | 2.6 | 1.1 | 149 | 231 | 276 | 24 |
| Spec | 3 | 10.8 | 0.7 | 159 | 197 | 286 | 18 |
| Iner | 3 | 2.1 | 1.2 | 162 | 204 | 302 | 20 |
| Linear | 3 | 1.7 | 1.2 | 213 | 324 | 400 | 22 |
| ML | 4 | 4.0 | 2.0 | 281 | 402 | 514 | 44 |
| Spec | 4 | 14.9 | 1.4 | 296 | 403 | 526 | 44 |
| Iner | 4 | 2.8 | 1.7 | 295 | 411 | 539 | 48 |
| Linear | 4 | 2.7 | 2.1 | 348 | 518 | 639 | 52 |

Schwerachsen-(Iner)- und Koordinaten-(Linear)-Bisektion (in Klammern die Bezeichnung entsprechend Tabelle 6.1). Die erzeugten Partitionierungen wurden mit der Kernighan-Lin-Heuristik verbessert. Die benötigten Rechenzeiten für das globale Verfahren und die lokale Verbesserung sind in den Spalten global bzw. lokal angegeben.
Die Qualität der Partitionierung wird anhand der vier Parameter beschrieben, die in den Spalten mit den Überschriften *cuts*, *hops*, *bndy* und *adj* angegeben sind. Diese Parameter haben die folgende Bedeutung:

**cuts** Die Anzahl der Kanten im zugeordneten Graphen, die von der Partitionierung geschnitten werden. Diese Größe bestimmt wesentlich den Gesamtumfang der Nachrichten, die ausgetauscht werden müssen.

**hops** Die Anzahl der von der Partitionierung geschnittenen Kanten, gewichtet mit der Anzahl der Hops, die zwischen den beiden benachbarten Gebieten liegen. Diese Anzahl berechnet sich für Hypercubes aus der Anzahl der unterschiedlichen Bitpositionen in den Ordnungszahlen der beiden Gebiete. Für 2D- und 3D-Gebiete berechnet sich diese Anzahl aus der Anzahl der differierenden Zeilen- bzw. Spaltenpositionen. Für Hypercube-, 2D- und 3D-Gitter-Topologien kann damit eine zutreffendere Charakteristik der zu erwartenden Netzbelastung angegeben werden.

**bndy** Die Anzahl der Knoten, die an eine geschnittene Kante anschließen. Für eine Bipartition gilt $n_{bndy} = 2n_{cuts}$.

**adj** Die Anzahl von Nachbarschaftsgruppen, d.h. die Anzahl der Kommunikationspfade beim Austausch über die Koppelkanten. Diese Maßzahl charakterisiert die Anzahl der verschiedenen Kommunikationsaufrufe (mit sich unterschiedenden Abesnder bzw. Empfänger).

Aus den Ergebnissen in jeder Gruppe von Partitionierungsproblemen wird deutlich sichtbar, daß die Kombinationen von Multilevel-, Spektral- bzw. Schwerachsenbisektion und heuristischer Verbesserung qualitativ vergleichbare Ergebnisse liefern. Das schnellste Verfahren ist generell die Schwerachsenbisektion, gefolgt von der Multilevel- und der Spektralbisektion, die deutlich langsamer arbeitet. Die Besonderheit, daß Finite-Element-Netze mit ihren Koordinateninformationen vorliegen,


Abbildung 6.5: *Benutzeroberfläche für die Partitionierung*

kann also vorteilhaft genutzt werden. Mit Koordinatenbisektion werden demgegenüber deutlich schlechtere Ergebnisse erreicht. Allerdings benötigt dieses Verfahren auch die mit Abstand geringste Rechenzeit.
Relativiert werden diese Aussagen, wenn man die Zeit für die Problemlösung der Partitionierungszeit gegenüberstellt. Das schnellste Lösungsverfahren berechnet auf 16 Prozessoren das Ergebnis für einen Lastfall in 4.1 Sekunden. Der gesamte Berechungsdurchlauf erfordert 6.1 Sekunden. Dem stehen etwa 4-5 Sekunden Partitionierungszeit gegenüber. Das Beispiel zeigt, daß eine qualitativ hochwertige Partitionierung sich nur lohnt

- für wiederkehrende Berechnungen eines Problems mit mehreren Lastfällen,
- für eine moderate Prozessorzahl oder
- für zeitaufwendige, nichtlineare Berechnungen.

Die Abbildung einer gefundenen Partitionierung kann mit Hilfe der Hops-Metrik optimiert werden. Auf der Grundlage einer initial gefundenen Zuordnung wird eine heuristische Umsortierung der Partitionen erzeugt, die eine Minimierung der Hops-Anzahl zur Folge hat. Dabei wird mit einer Monte-Carlo-Methode vorgegangen.

## 6.3 Parallele Gleichungslösung

Mit den in 5 beschriebenen mathematischen Grundlagen wurde in ein bestehendes FEM-Programm [160] ein parallelisierter Gleichungslöser eingebaut. Der Gleichungslöser basiert auf der nichtüberlappenden Gebietszerlegung, d.h. finite Elemente werden genau einem Teilgebiet zugeordnet. Die Knoten werden unterschieden



nach den vollständig innerhalb eines Teilgebiets liegenden Knoten, den Innenknoten, und auf dem Rand liegenden Knoten, den Koppelknoten. Ein und derselbe Koppelknoten ist mehr als einem Teilgebiet zugeordnet. Die numerischen Bausteine, die dafür benötigt wurden sind

- globale Skalarprodukte für unterschiedlich verteilte Vektoren (vgl. 5.5.1)
- globale Vektorsuperposition (vgl. 6.3.2)
- lokale Matrix-Vektor-Operationen (vgl. 6.3.3)

In zweidimensionalen Problemen lassen sich Koppelknoten auf Gebietsrändern, die genau zwei Teilgebieten zugeordnet sind und Kruezungspunkte, die mehr als zwei Teilgebieten zugeordnet sind, unterscheiden. In der hier vorgestellten Implementation wird diese Unterschiedung in der globalen Vektorsuperposition nicht ausgenutzt.

### 6.3.1 Datenstruktur

Die Datenstrukturen für die benötigten Matrizen und Vektoren sind in [160] beschrieben. Die Steifigkeitsmatrix $\mathbf{K}$ ist als einfach verkettete Liste von Submatrizen implementiert. Die Dimension der Submatrizen entspricht der Anzahl der Freiheitsgrade der zugeordneten Knoten. Für das Plattenproblem ergibt das maximal 3×3-Submatrizen.
Jeder Knoten verwaltet eine Liste der ihm zugeordneten Matrixzeile, über die die Submatrizen dieser Zeile referenziert werden. Die Knoten selbst sind ebenfalls in einer einfach verketteten Liste abgespeichert. Knoten werden mit einem Namen identifiziert. Der Namen ist für einen Prozessor bzw. ein Teilgebiet eineindeutig.
Der Name wird intern so gewählt, daß aus ihm eine Vorgänger-Nachfolger-Beziehung abgeleitet werden kann. Insbesondere kann daraus ermittelt werden, ob ein Knoten $i$ vor oder nach dem Knoten $j$ in der Knotenliste steht. Für die symmetrische Steifigkeitsmatrix $\mathbf{K}$ ist es damit ausreichend, z.B. nur die Submatrizen der oberen Dreiecksmatrix zu speichern und die Zeiger auf die Submatrizen der unteren Dreiecksmatrix auf die entsprechend transponierten Elemente der oberen Dreieckmatrix verweisen zu lassen. Ob eine so referenzierte Submatrix in einer Matrixoperation transponiert oder nicht transponiert verarbeitet werden muß, wird dann anhand der Vorgänger-Nachfolger-Beziehung entschieden.
Die Systemmatrix ist unregelmäßig auf den Hauptspeicher verteilt. Nur die Elemente der knotenbezogenen Submatrizen bzw. Subvektoren (die direkt mit der Knotenliste verwaltet werden) sind als fortlaufende Speicherbereiche verfügbar. Numerische Kernalgorithmen auf dieser Datenstruktur umfassen Matrix-Vektor-Multiplikationen, Cholesky-Faktorisierung, Vorwärts- und Rückwärtseinsetzen sowie Matrix-Matrix-Multiplikationen mit zwei bzw. vier Varianten für die eventuell transponierten Operanden. Diese Operationen können in Anlehnung an die BLAS Level 2 Funktionsbibliothek formuliert werden.

### 6.3.2 Implementation der globalen Vektorsuperposition

Die für die parallele Berechnung partitionierten Netzinformationen werden um die Zuordnung von globalen Namen zu den prozessorlokalen Knoten erweitert. Mit dieser Information kann in einem Initialisierungsschritt auf jedem Prozessor festgestellt werden, mit welchem anderen Prozessor Nachbarschaftsbeziehungen bestehen. Dazu wird für jeden im Gesamtsystem vorhandenen Koppelknoten eine Inzidenzliste aufgestellt. Diese Inzidenzliste wird in einem globalen Austausch ermittelt. Danach kann jeder Prozessor selbständig für jeden Rand mit einem anderen Prozessor Austauschpuffer und die notwendigen Methoden zum Füllen der Austauschpuffer

```
für alle Ränder
  initialisiere Sendepuffer mit lokalen Daten - Gather
  initialisiere Empfangsoption
  i++
für alle Ränder
  Starte Sendeoperation
solange  i > 0
  warte auf Daten eines Randes
  falls Empfangsoption  j  initialisiert
    verknüpfe externe Daten mit lokalen Daten - Scatter
    lösche Empfangsoption  j
    i--
```

Abbildung 6.6: *Austausch über die Koppelränder*

(*Gather*) und zum Verteilen der eintreffenden Daten der Nachbarn (*Scatter*) initialisieren. Der Austausch über die Koppelränder erfolgt dann auf Basis dieser Datenstruktur wie in Abbildung 6.6 dargestellt.
Die benutzte Sendeoperation ist nicht blockierend. Damit wird sichergestellt, daß jeder Prozessor zunächst seine Nachrichten absetzen kann. Die Reihenfolge des Eintreffens der Nachrichten wird im Algorithmus nicht fixiert. Somit kann flexibel auf Einflüsse des gegebenenfalls erforderlichen Routings durch das Message-Passing System reagiert werden. Der Algorithmus ist unabhängig von der speziellen Abbildung des logischen Kommunikationsnetzwerkes zwischen den benachbarten Prozessoren auf eine bestimmte Hardwaretopologie. Problematisch ist der Speicherplatzbedarf. Zumindest der Sendepuffer muß für alle Ränder allokiert werden. Die eintreffenden Nachrichten sind am besten auch in zugehörige, bereits vorher allokierte Speicherbereiche einzulesen, um das Message-Passing-System von kostspieliger Speicherverwaltung zu entlasten. In MPI arbeitet man am besten mit nutzerdefinierten Pufferbereichen und mit den MPI_Ibsend bzw. MPI_Brecv-Funktionen.

### 6.3.3 Implementation lokaler Matrixoperationen auf RISC-Mikroprozessoren

Der wesentliche Anteil an der Laufzeit des PCG-Algorithmus wird vom Matrix-Vektor-Produkt und von der Vorkonditionierung konsumiert. Am Beispiel der Optimierung dieser Operationen für den DEC $\alpha$21064 Mikroprozessor (eingesetzt in der Cray-T3D) soll die Effizienz der gewählten Datenstruktur diskutiert werden.
Die Laufzeitanalyse zeigt deutliche Diskrepanzen zwischen der nominal erreichbaren und der tatsächlich erreichten Einzelprozessorleistung. Die Rechenleistung eines Einzelprozessors in den berechnungsintensiven Teilen lagen bei den ersten Laufzeitmessungen bei etwa 1.0 MFLOPS im Vergleich zu 150.0 MFLOPS Spitzenleistung bzw. bis zu 40.0 MFLOPS für optimierte BLAS Level 2-Routinen, die auch die Rechenkerne der lokalen Matrixoperationen bilden. Allerdings erreichen BLAS Level 2-Routinen diese Werte nur für entsprechend große Matrixdimensionen. Die Speicherstruktur der Steifigkeitsmatrix $K$ für das Plattenproblem basiert aber auf Submatrizen mit der maximalen Dimension drei. Entsprechend kann für die lokalen Matrixoperationen nur ein Spitzenwert bis zu etwa 5 MFLOPS erwartet werden.



Tabelle 6.2: *Laufzeit verschiedener PCG-Varianten auf VPP500, Quelle: [33]*

| Abschnitt | CG | PCG-Jacobi | PCG-IC |
|---|---|---|---|
| send p | 1.3 | 0.6 | 0.4 |
| Ap | 8.9 | 0.9 | 3.0 |
| p'Ap | 1.3 | 0.6 | 0.4 |
| 2 saxpy | 1.9 | 0.9 | 0.6 |
| r'z | 1.2 | 0.6 | 0.5 |
| 1 saxpy | 0.9 | 0.5 | 0.3 |
| solve | 0 | 4.5 | 14.6 |
| gesamt | 15.5 | 8.6 | 19.8 |

Der Mikroprozessor DEC $\alpha$21064 besitzt je 64 Register für 64-bit-Gleitkommazahlen und 64-bit-Integerzahlen. Daten in den Registern können sofort in arithmetischen Operationen genutzt werden. Auf der nächsten Hierarchieebene verfügt der Prozessor über einen 8 kByte großen *second-level* Datencache. Die Cacheline ist 256 bit und vier 64-bit-Gleitkommazahlen können gleichzeitig vom Hauptspeicher in den Datencache geladen werden. Daten aus dem Cache können in einem Taktzyklus in ein Register geladen werden. Sind Daten nicht im Cache, müssen sie direkt aus dem Hauptspeicher geladen werden. Dafür sind mehrere Taktzyklen erforderlich.
Ein unregelmäßiges Speicherzugriffsmuster kann die Rechenleistung erheblich reduzieren. Die Hauptspeicheradressen werden direkt im Cache abgebildet (*direct mapped cache*). Damit kann der Inhalt einer bestimmten Hauptspeicheradresse nur an genau eine Stelle im Cache geladen werden. Besitzen zwei Operanden Adressen, die auf diesselbe Stelle im Cache abgebildet werden, können nicht beide Operanden gleichzeitig im Cache zur Verfügung gestellt werden. Der Prozessor kann somit keinen Nutzen aus dem schnelleren Zugriff auf die Daten im Cache ziehen (*cache miss*). Mindestens ein Operand muß dann direkt aus dem Hauptspeicher geladen werden.
Problematisch ist weiterhin eine Situation, in der ein Cacheplatz mit dem Ergebniswert einer bereits abgearbeiteten Operation belegt ist. Bevor ein neuer Operand für eine folgende Operation in den Cache geladen werden kann, muß das alte Ergebnis in den Hauptspeicher zurückgeschrieben werden. In diesem Fall kann der Prozessor nur mit der halben Geschwindigkeit des Hauptspeicherzugriffs arbeiten. Ein Großteil der Leistungsverluste in der Verarbeitung von dünnbesetzten Matrix ist auf solche Cache-Effekte zurückzuführen.
Brent et.al. berichten in [33] über die Implementation des PCG-Verfahrens auf dem Vektor-Parallelrechner VPP500. Dabei wurden verschiedene Varianten der Vorkonditionierung ausgetestet. Die Vorkonditionierung mit unvollständiger Cholesky-Zerlegung erforderte mit Abstand die wenigsten Iterationen. Sie kann aber nur etwa die Hälfte der Arithmetikleistung z.B. der Block-Jacobi-Vorkonditionierung leisten, die bei fast doppelter Iterationszahl vergleichbare Gesamtlaufzeiten bis zum Erreichen der gewünschten Abbruchgrenze zeigt.
In Tabelle 6.2 sind diese Ergebnisse gegenübergestellt. Für ein festes Problem sind die Zeitscheiben für die einzelnen Bearbeitungsschritte bei der Lösung mit nichtvorkonditioniertem CG-Vergfahren (CG), PCG-Verfahren mit Blockdiagonalskalierung (PCG-Jacobi) und PCG-Verfahren mit unvollständiger Cholesky-Faktorisierung als Vorkonditionierung gegenübergestellt. Die Anwendung der Vorkonditionierung kostet bei PCG-IC mit Abstand die meiste Zeit.
Das Speicherformat für die dünnbesetzten Matrizen ist eine Variante des komprimierten Spaltenformats. Die Matrixspalten wurden blockweise auf die Prozessoren

Tabelle 6.3: *Laufzeitverteilung PCG-Zyklus (16 Knoten Cray-T3D)*

| Abschnitt | avg [s] | max [s] | min [s] | MFLOP PE | MFLOP total |
|---|---|---|---|---|---|
| Ap | 3.73 | 3.79 | 3.66 | 2.08 | 33.23 |
| p'Ap | 0.10 | 0.18 | 0.05 | 1.67 | 26.70 |
| 2 saxpy | 0.05 | 0.05 | 0.04 | 25.62 | 409.93 |
| vorkond | 10.54 | 10.71 | 10.42 | 3.26 | 52.11 |
| send r | 1.52 | 1.51 | 1.58 | | |
| forslv | 4.90 | 5.00 | 4.81 | 3.59 | 57.50 |
| backslv | 4.12 | 4.20 | 4.03 | 4.02 | 64.37 |
| r'd | 0.22 | 0.34 | 0.04 | 0.87 | 13.93 |
| 1 saxpy | 0.02 | 0.02 | 0.02 | 23.80 | 380.76 |

verteilt, so daß jeder Prozessor $n/p$ Spalten erhält. Der Flaschenhals bei der Implementierung des Vorkonditionierers ist der Lösungsschritt, in dem mehrere Ebenen der indirekten Adressierung benutzt werden.
Der Vorteil der gewählten Speichertechnik besteht in der fortlaufenden Speicherung der Matrixelemente einer Spalte. Diese Optimierung kann mit der Listenstruktur nach [160] nur dadurch erreicht werden, daß die den Listenelementen zugeordneten Submatrizen in geeigneter Weise im Speicher angeordnet werden. Dazu ist vor der Anforderung des Speicherplatzes der Submatrizen zunächst ein Sortierschritt in den Gleichungszeilen erforderlich, so daß die Submatrixzeiger in aufsteigender Reihenfolge traversiert werden. Im Gegensatz zu [160] wird bei dieser Umsortierung auch die Speicherung von eventuell bereits vorhandenen Submatrizen verändert. Es wird gesichert, daß die obere Dreiecksmatrix entsprechend der aktuellen Reihenfolge der Knoten in der Knotenliste gespeichert wird. Die Reihenfolge ist damit hinreichendes Kriterium für die Charakterisierung der Speicherung der Submatrix. Gegebenenfalls müssen dafür die Submatrizen transponiert werden. Im Matrix-Vektor-Produkt entfällt dann die Fallunterscheidung im Produkt über die Submatrizen der Gleichungszeile. Das Zugriffsmuster auf die Elemente der Submatrix entspricht der linearen Traversierung der Matrixelemente.
Nach der Änderung der Speicheranforderung für die Systemmatrix, so daß die nacheinander zu verarbeitenden Submatrizen einer Zeile im Gleichungssystem auch unmittelbar nacheinander im Hauptspeicher angeordnet sind, konnte deshalb ein entscheidender Effektivitätszuwachs nachgewiesen werden. Der Effekt wurde noch verstärkt, indem die Submatrizen nicht fortlaufend im Hauptspeicher angeordnet wurden, sondern jede Submatrix auf den Beginn einer Cacheline ausgerichtet wurde. Im Lösungsschritt des Vorkonditionierers werden auf jedem DEC $\alpha$ Prozessor ca. 4 MFLOPS erreicht. Die Abarbeitungsreihenfolge innerhalb der Matrixoperationen wurde so verändert, daß fortlaufend abgespeicherte Elemente fortlaufend benutzt werden. Damit konnte maximaler Gewinn aus jedem Hauptspeicherzugriff gezogen werden. Eine Übersicht über die Verteilung der Gesamtrechenzeit auf die einzelnen PCG-Schritte bei Abarbeitung auf einer 16 Knoten Cray-T3D zeigt Tabelle 6.3.
Über analoge Probleme bei der Implementierung eines parallelen Finite-Element-Codes auf einem Intel Touchstone Delta-System berichten Das et. al. in [47]. Der dort eingesetzte Intel i860-Mikroprozessor verfügt ebenfalls nur über 8 kByte *second-level Cache*. Mit geeigneten Umordnung, so daß die Knotenreihenfolge eine geringe Bandbreite garantiert und die einem Knoten zugeordneten Daten entsprechend dieser Reihenfolge aufsteigend sortiert werden, konnte auch dort ein besser strukturiertes Speicherzugriffsmuster erreicht werden. Problemabhängig konnte damit eine



Tabelle 6.4: *Laufzeitverteilung PCG-Zyklus (16 Knoten PowerXPlorer)*

| Abschnitt | avg [s] | max [s] | min [s] | MFLOP PE | MFLOP total |
|---|---|---|---|---|---|
| Ap | 3.09 | 3.12 | 3.05 | 2.52 | 40.31 |
| p'Ap | 1.11 | 1.23 | 0.27 | 0.24 | 3.82 |
| 2 saxpy | 0.19 | 0.19 | 0.18 | 6.28 | 100.46 |
| vorkond | 12.65 | 12.85 | 12.56 | 2.72 | 43.45 |
| send r | 2.00 | 1.97 | 2.15 | | |
| forslv | 5.81 | 5.94 | 5.68 | 3.03 | 48.43 |
| backslv | 4.83 | 4.94 | 4.73 | 3.42 | 54.75 |
| r'd | 0.51 | 0.60 | 0.30 | 0.49 | 7.87 |
| 1 saxpy | 0.09 | 0.09 | 0.09 | 6.56 | 104.89 |

Steigerung der Rechenleistung bis zu einem Faktor zwei erzielt werden.
Der Parsytec PowerXPlorer arbeitet mit dem Prozessor PowerPC MPC601, der nur mit 80 MHz getaktet ist, dafür aber über einen 32 kByte großen *Second-level Cache* verfügt. Der Prozessor kann deshalb bei den hier beschriebenen Anwendungen den Nachteil der geringen Taktfrequenz und damit der geringeren Einzelprozessorleistung zum großen Teil wettmachen. Die oben beschriebene, optimierte Programmversion wurde auch auf dem PowerXPlorer getestet. Die Laufzeitstatistik ist in Tabelle 6.4 angegeben. Der Vergleich zu den Laufzeitdaten auf der Cray-T3D zeigt die Überlegenheit der hochoptimierten BLAS-Level 1 Routinen (*saxpy*) der Cray-Bibliothek. Während die anderen lokalen Operationen in etwa gleiche Leistungsdaten zeigen, erreicht der DEC $\alpha$ fast die vierfache Rechengeschwindigkeit in diesen Routinen. Außerdem macht sich das schnellere Kommunikationsnetzwerk der Cray-T3D bei den *Start-up* kritischen Skalarprodukten signifikant bemerkbar, während die von der tatsächlichen Transferzeit und der Scatter/Gather-Operation beherrschten parallelen Vektorsuperpositionen fast in gleicher Geschwindigkeit ausgeführt werden.
Eine weitere Verbesserung der Laufzeit ist denkbar durch

- Reduktion der Kommunikationszeit, die ca. 80% der Zeiten für das Skalarprodukt bzw. 15% der Zeit für die Vorkonditionierung, inklusive der lokalen Speicheroperationen beim Einsammeln bzw. Verteilen der Daten, ausmacht
- Erhöhung der Blockgröße in den Matrixroutinen (Matrix-Vektor-Produkt, Vorkonditionierung)

Die Erhöhung der Blockgröße der Submatrizen kann nur mit einer entsprechenden Reorganisation der Matrixdatenstruktur realisiert werden. Dazu sind die Nachbarschaftsverhältnisse im Finite-Element-Netz auszunutzen, um die *Clique*-Struktur zu ermitteln. Die den Knoten einer *Clique* zugeordneten Submatrizen können zu größeren, dichtgespeicherten Blöcken zusammengefaßt werden.

## 6.4 Adaptivität und Lastanpassung

### 6.4.1 Parallele Netzverfeinerung

Die Verfeinerungsstrategie auf der Basis eines der oben beschriebenen, elementweise berechneten Fehlerindikatoren arbeitet rekursiv in einer Elementschleife und kann wie folgt angegeben werden:

1. Verfeinerung des Elements $i$, falls für Fehlerindikator $\lambda_i > \lambda_\epsilon$ gilt
2. Kontrolle über alle Elementkanten
3. ggf. Verfeinerung des Nachbarelements und Fortsetzung bei 2.

Schritt 2 ist im parallelen Fall für die an den Teilgebietsrand grenzenden Elemente mit Kommunikation verbunden. Das Element am Teilgebietsrand muß die im anderen Teilgebiet benachbarten Elemente über die neuentstandenen Elementknoten auf dem Koppelrand informieren. Diese Information wird auf die gleiche Art ausgetauscht, wie die oben beschriebene Vektorsuperposition. Anstelle der Vektorkomponenten werden Informationen über die Koppelkanten verschickt.
Das Verfeinerungskonzept mit irregulär verfeinerten Fünf- und Sechsknotenelementen setzt voraus, das benachbarte Elemente nur zu unmittelbar aufeinanderfolgenden Verfeinerungsstufen gehören. Aufeinanderfolgende Verfeinerungen an einem Punkt des FE-Netzes mit lokaler Singularität führen bei Anwendung dieser Regel zu in konzentrischen Kreisen zunehmend verfeinerten Strukturen. Die Auswirkung der Verfeinerung eines Elements ersteckt sich im Grenzfall auf das gesamte Netz. Für die serielle Implementation ist das kein Nachteil. Der Algorithmus kann effizient als Rekursion formuliert werden. Wird das gleiche Vorgehen auch für die parallele Netzverfeinerung benutzt, wird insbesondere an Kreuzungspunkten eine nicht vorhersehbare Anzahl von Kommunikationen zwischen rekursiv weiterzuverfeinernden Elementen erforderlich. Die Verfeinerung eines einzelnen Elements zieht eine Kettenreaktion nach sich, deren Ausbreitung nur durch benachbarte Elemente der gleichen Verfeinerungsstufe gestoppt werden kann. Die Anzahl der Kommunikationen ist zwar beschränkt, ideal ist aber eine konstante Anzahl von Kommunikationen. Dazu wird der Verfeinerungsalgorithmus für die Elemente an den Teilgebietsrändern modifiziert. In jedem Verfeinerungsschritt wird gesichert, daß über Teilgebietsgrenzen benachbarte Elemente zur gleichen Verfeinerungsebene gehören. Elemente, deren irreguläre Knoten auf Teilgebietsgrenzen liegen, werden in die Verfeinerung mit einbezogen. Damit stoppt die rekursive Verfeinerung sofort an den Teilgebietsgrenzen.

### 6.4.2 Dynamische Lastanpassung

Für die dynamische Lastanpassung bei der parallelen, adaptiven Plattenberechnung wurde eine lokale Strategie implementiert (vgl. Abschnitt 3.1.3). Der Vorteil der lokalen Lastanpassung liegt in

- der Reduktion des Kommunikationsaufwandes,
- dem wesentlich geringeren Speicheraufwand und
- der Skalierbarkeit.

Die zuletzt genannte Eigenschaft wird durch die Ausnutzung der lokalen Parallelität des Verfahrens garantiert. Die implementierte Lösung benötigt lediglich einen globalen Initialisierungsschritt.
In diesem Initialisierungsschritt wird die Verteilung der Lasteinheiten auf die Teilgebiete bestimmt. Die Lasteinheiten sind die Knoten des Finite-Element-Netzes. Jedes Teilgebiet bestimmt seine lokale Last und gibt diese Information an alle anderen Teilgebiete weiter. Außerdem wird der Zusammenhang der Teilgebiete ausgetauscht, so daß jeder Prozeß vollständige Informationen über die vorhandene Kommunikationsstruktur besitzt. Diese globale Kommunikation (MPI_Allgather) führt zu einer Synchronisation der Prozesse. Das Datenaufkommen ist moderat und läßt sich mit der Anzahl der benachbarten Teilgebiete $n$ (für den 2D-Fall gilt üblicherweise $2 \leq n \leq 10$, für den 3D-Fall etwa $2 \leq n \leq 30$) mit $O(np)$ abschätzen.



Auf der Basis des zuvor bestimmten Kommunikationsgraphen wird dann der in [103] beschriebene *Multilevel Diffusionsalgorithmus* angewendet. Die Grundidee besteht darin, die Lastverteilung rekursiv zu formulieren und in jeder Rekursionsebene das Lastverteilungsproblem als Bisektionproblem zu lösen:

**Schritt 0** Aktueller Teilgraph ist der Kommunikationsgraphen

**Schritt 1** Halbierung des aktuellen Teilgraphen

**Schritt 2** Bestimmung der Prozessorposition im aktuellen Teilgraphen

**Schritt 3** Bestimmung der Lastdifferenz zum benachbarten Teilgraphen

**Schritt 4** Bestimmung der Kopplungen an den benachbarten Teilgraphen

**Schritt 5** Lastausgleich über diese Kopplungen

**Schritt 6** Falls der aktuelle Teilgraph mehr als ein Element hat, Fortsetzung mit Schritt 1.

Das Verfahren wurde modifiziert. Der Lastausgleich über die bestehende Kopplungen zum benachbarten Teilgraphen (Schritt 5) wurde so gestaltet, das Prozessoren mit großem Lastanteil proportional mehr Lasteinheiten austauschen als Prozessoren mit weniger Lasteinheiten. Es werden dabei die in diesem Austauschschritt tatsächlich noch vorhandenen kumulativen Lasteinheiten der aktiven Prozessorgruppe berücksichtigt.

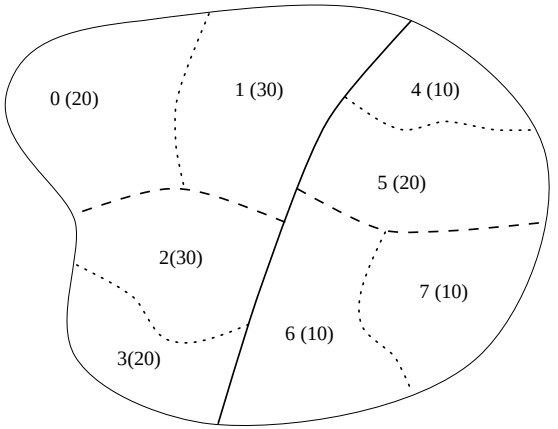


Abbildung 6.7: *Problem mit 8 Teilgebiete und ungleichmäßig verteilter Last (Lasteinheiten in Klammern)*

Der Ablauf soll anhand eines Beispiels verdeutlicht werden. In Abbildung 6.7 ist ein Problem mit acht Teilgebieten angegeben. Die Teilgebiete sind acht Prozessoren entsprechend der angegebenen Numerierung zugeordnet. Die Rechenlasten $N_i$ der Teilgebiete $i$ nach einer adaptiven Verfeinerung sind in Klammern angegeben. Die Gesamtlast des Problems ergibt sich zu $N_{ges} = \sum_i N_i = 160$. Damit ergibt sich eine durchschnittliche Teilgebietslast von $\frac{N_{ges}}{p} = \frac{160}{8} = 20$ als Ziel des Balancierungsprozesses.

Die Lastumverteilung erfolgt rekursiv. Zunächst wird der Lastausgleich zwischen den zwei Teilgebietsgruppen $g_0 = [0, 1, 2, 3]$ und $g_1 = [4, 5, 6, 7]$ vorgenommen. Zwischen beiden Gruppen mit den kumulativen Teillasten von $l_0 = \sum_{i=0}^{3} = 110$ und $l_1 = \sum_{i=4}^{7} = 50$ besteht eine Differenz von 60 Lasteinheiten, die mit dem Transport von $M_{ges} = 30$ Lasteinheiten von der stärker belasteten zur schwächer belasteten Teilgebietsgruppe ausgeglichen werden kann. Die Anzahl der tatsächlich zu transportierenden Lasteinheiten $M_i$ bestimmt sich zu

$$M_i = \frac{N_i}{l_j} M_{ges} \tag{6.13}$$

Der Lastausgleich erfolgt ausschließlich über die bestehenden Teilgebietsgrenzen. Somit geben nur die Teilgebiete 1, 2 und 3 mit insgesamt $l_0 = 90$ Lasteinheiten entsprechend $\frac{30}{90} * 30 = 10$, $\frac{40}{90} * 30 = 13$ bzw. $\frac{20}{90} * 30 = 7$ Lasteinheiten an die Teilgebietsgruppe $g_1$ ab. Die Teilgebiete 0 und 7 sind nicht am Lastausgleich beteiligt. Sie haben keinen direkten Nachbarn in der jeweils anderen Gruppe. Danach ist zwischen den beiden Teilgebietsgruppen das Lastgleichgewicht hergestellt.

Tabelle 6.5: *Lastbewegung für das Beispiel in Abbildung 6.7*

| Teilgebiet | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | = | 0 | 0 | - | - | - | - | - |
| 1 | 0 | = | 0 | - | -10 | 0 | 0 | - |
| 2 | 0 | 0 | = | -7 | - | - | -13 | - |
| 3 | - | - | 7 | = | - | - | -7 | - |
| 4 | - | 10 | 0 | 0 | = | - | - | - |
| 5 | - | 0 | 0 | 0 | 0 | = | - | 0 |
| 6 | - | 0 | 13 | 7 | - | | = | -10 |
| 7 | - | - | - | - | - | 0 | 10 | = |

Im nächsten Schritt erfolgt rekursiv die Bildung von weiteren Teilgruppen. Im einzelnen bedeutet das: aus $g_0$ werden $g_{00} = [0, 1]$ und $g_{01} = [2, 3]$ gebildet und aus $g_1$ werden $g_{10} = [4, 5]$ und $g_{11} = [6, 7]$ gebildet. Zwischen diesen neuen Teilgruppen wird die Last ausgeglichen und im dritten und letzten Schritt wird zwischen den einzelnen Teilgebieten dieser Gruppen die Last balanciert. In der Summe ergeben sich die in Tabelle 6.5 angegebenen Lastbewegungen.
Das Verfahren würde fehlschlagen, wenn auf einer Lastverteilungsstufe $M_i > N_i$ gilt. Dieser Konflikt entsteht immer dann, wenn ein oder mehrere Teilgebiete eine hohe lokale Last aufweisen, aufgrund ihrer Lage aber in diesem Schritt nicht am Lastausgleich beteiligt sind. Dann gilt auch $M_{ges} > l_j$ (aus Gleichung 6.13) und damit für die am Lastausgleich beteiligten Teilgebiete $M_i > N_i$. Diese Konstellation muß innerhalb der Teilgebietsgruppe durch vorgezogene Lastumverteilung vermieden, so daß die weniger belasteten Teilgebiete von den nicht im Austausch beteiligten, aber mehr belasteten Teilgebieten Lasteinheiten zum Weitertransport in die andere Teilgebietsgruppe erhalten.
Der Initialisierungschritt kann bei großen Prozessorzahlen zum Engpaß werden, weil die Kommunikations zum Aufbau und die Speicherung des Kommunikationsgraphen zu aufwendig ist. Alternativ kann dieser Schritt auch mit lokaler Kommunikation wie folgt implementiert werden:

**Schritt 1** Halbierung des aktuellen Teilgraphen

**Schritt 2** Bestimmung der Prozessorposition im aktuellen Teilgraphen



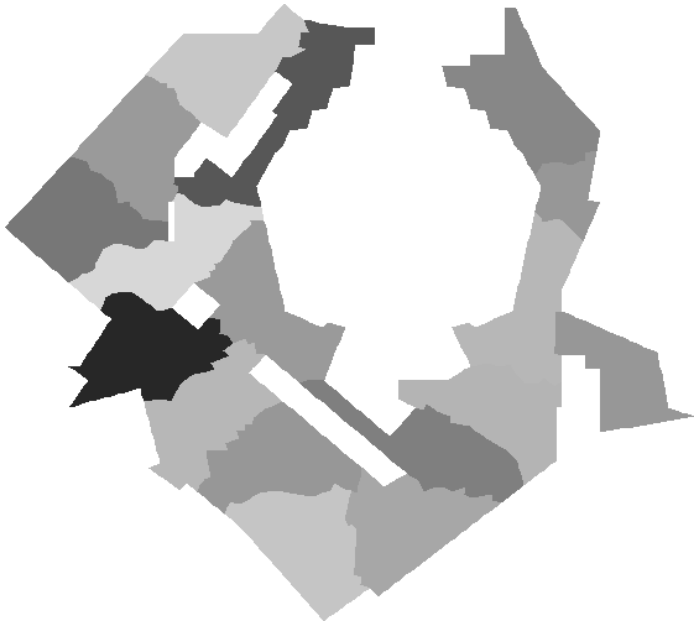

Abbildung 6.8: *Partitioniertes Plattenproblem*

**Schritt 3** Bestimmung der Lastdifferenz zum benachbarten Teilgraphen

**Schritt 4** *Kommunikation* zur Bestimmung der Kopplungen an den benachbarten Teilgraphen

**Schritt 5** Lokaler Lastausgleich

**Schritt 6** Falls der aktuelle Teilgraph mehr als ein Element hat, Fortsetzung mit Schritt 1.

Die Kommunikationen beziehen sich dabei immer nur auf einen Teilgraphen, können deshalb mit zunehmender Rekursionstiefe und entsprechend weniger Teilnehmern, immer effizienter ablaufen.
Im Resultat des Initialisierungsschrittes ist auf jedem Prozessor die Anzahl der für die Lastbalancierung zu transportierenden Lasteinheiten pro benachbarten Prozeß bekannt. Es schließt sich der eigentliche Lastausgleich mit der Auswahl und dem Austausch der Lasteinheiten an.

## 6.5 Anwendungsbeispiel

Als Anwendungsbeispiel soll exemplarisch ein anspruchsvolles Plattenproblem gelöst werden. Eine punktweise gestützte und partiell an den Rändern eingespannte Platte wurde mit Viereck-DKT-Elementen vernetzt und für den Lastfall Eigengewicht

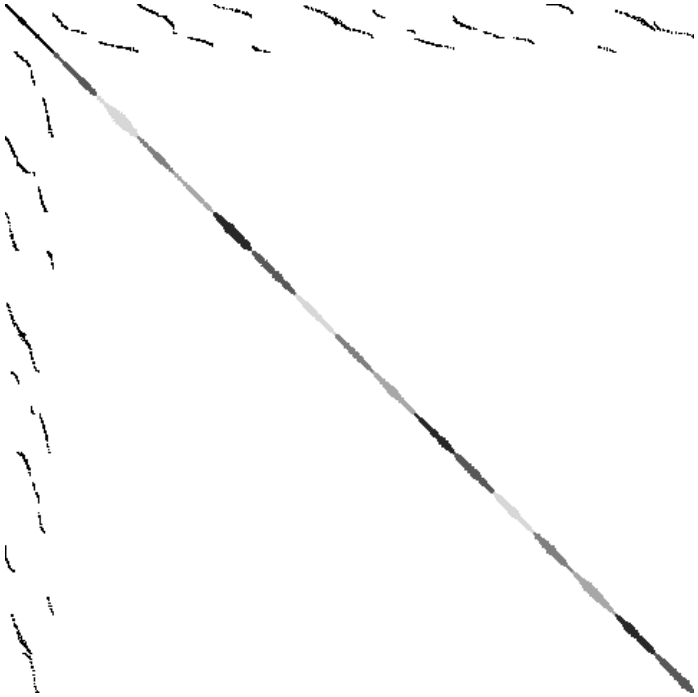

Abbildung 6.9: *Matrixstruktur*

berechnet. Die Berechnung erfolgte auf einem Parallelrechner mit 16 Prozessoren. Es wurden 3 Adaptionsschritten berücksichtigt.

Das unstrukturierte Finite-Element-Netz besitzt initial 4480 Knoten und 4151 Elemente. Insgesamt hat das Problem 12637 Freiheitsgrade. Zur Partitionierung wurde das Multilevel-Spektralbisektionsverfahren eingesetzt. Nach der Kernighan-Lin-Glättung weist das Problem insgesamt 316 Koppelknoten mit insgesamt 932 Koppelfreiheitsgraden auf. Die Geometrie des Problems und eine Partitionierung für 16 Teilgebiete ist in Abbildung 6.8 dargestellt. Die Zerlegung mit der Multilevel-Spektralbisektion erfordert auf einer Sun-Workstation (SPARC 10/20) etwa 5 Sekunden und die anschließende heuristische Verbesserung der Zerlegung mit dem Kernighan-Lin-Algorithmus weitere 2 Sekunden. Ein Teilgebiet ist nicht zusammenhängend. Das ist Resultat der unregelmäßigen Berandungsgeometrie.

Die Struktur der verteilten Matrix ist in Abbildung 6.9 zu sehen. Das Koppelproblem ist in der ersten Zeile bzw. Spalte dargestellt. Deutlich ist die unregelmäßige Struktur der Koppelmatrizen zu erkennen. Die Teilgebietsprobleme selbst haben eine geringe Bandbreite.

Drei verschiedene Löservarianten wurden für die Berechnung des initialen Problems benutzt:



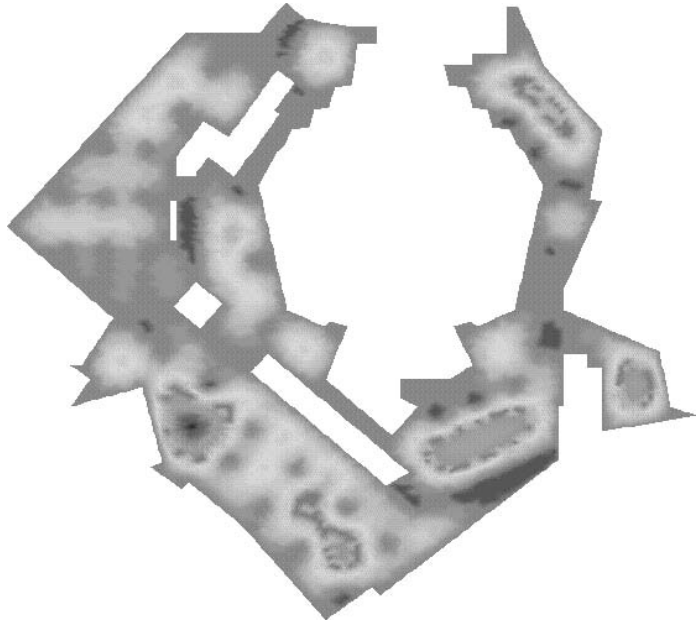

Abbildung 6.10: *Berechnungsergebnis z-Verschiebung*

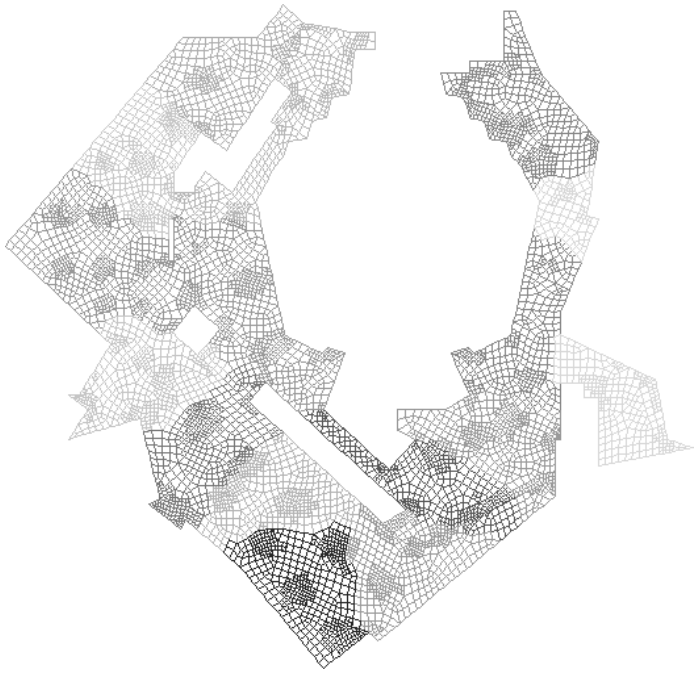

Abbildung 6.11: *Verfeinerung nach dem ersten Berechnungsschritt*

**Primäres Gebietszerlegungsverfahren - iterative Lösung des Gesamtproblems** Bei diesem Löser wurden sowohl die lokalen Probleme als auch das Koppelproblem iterativ gelöst. Als Vorkonditionierung wurde für die lokalen Probleme eine unvollständige LU-Zerlegung benutzt. Die LU-Zerlegung wurde auf das vorhandene Belegungsmuster der blockweise gespeicherten Steifigkeitsmatrizen $K_{ii}$ angewendet. Die Koppelfreiheitsgrade wurden mit der Blockdiagonalskalierung vorkonditioniert. Die Blockgröße entspricht den Knotenfreiheitsgraden. Dazu mußten in einem Initialisierungsschritt zunächst die Einträge der zugehörigen Diagonalblöcke superponiert werden. Das wurde in einem erweiterten Vektoraustausch realisiert.
Die Berechnung erforderte insgesamt 431 Iterationen und wurde in 37 Sekunden abgeschlossen.

**Primäres Gebietszerlegungsverfahren - iterative Lösung des Schurkomplementproblems** Als zweite Variante der Lösung wurde eine Kombination aus lokalem, direktem Löser und iterativem Löser für das Koppelproblem gewählt. Die lokalen Probleme wurden mit einer Cholesky-Faktorisierung für dünnbesetzte Matrizen gelöst. Die faktorisierte Matrix erforderte 267 kByte Speicherplatz und wurde in 0.2 Sekunden zerlegt. Dabei wurde mit 16 Knoten PowerXPlorer (PowerPC MPC601) eine summarische Spitzenleistung von 105 MFLOPS erreicht.
Das Koppelproblem ist wiederum durch Blockdiagonalskalierung vorkonditioniert worden. Insgesamt werden 155 Iterationen benötigt. Das erforderte eine Rechenzeit von 10.8 Sekunden.



**Duales Gebietszerlegungsverfahren** Dieses Verfahren wurde als dritte Lösungstechnik eingesetzt. Auch hierbei wird ein direkter Löser für die lokalen Probleme benutzt. Zum Einsatz kam der Band-Cholesky aus der LAPACK-Implementation. Dieser Löser benutzt geblockte BLAS Level 3-Routinen und erreicht insgesamt 242 MFLOPS auf 16 Knoten PowerXPlorer (PowerPC MPC601). Allerdings müssen bei Berücksichtigung der vollständigen Bandstruktur bedeutend mehr arithmetische Operationen durchgeführt werden, so daß insgesamt 0.5 Sekunden für die Faktorisierung der lokalen Probleme benötigt werden.
Die Iteration über das Koppelproblem der Lagrange-Multiplikatoren erfordert 36 Iterationen. Das Problem ist in 4.1 Sekunden gelöst. Nach 3 Verfeinerungsschritten ist das Problem auf insgesamt 65381 Freiheitsgrade gewachsen. Das Berechnungsergebnis steht dafür nach insgesamt ca. 7 Minuten bereit. In Tabelle 6.6 sind einige statistische Angaben für diesen Berechnungslauf angegeben.

Tabelle 6.6: *Statistik und Laufzeit*

| | initial | 1. Schritt | 2. Schritt | 3. Schritt |
|---|---|---|---|---|
| Knoten | 4480 | 7704 | 13096 | 22339 |
| Elemente | 4151 | 6746 | 11294 | 19415 |
| Koppelknoten | 316 | 451 | 643 | 874 |
| Freiheitsgrade | 12637 | 22103 | 38042 | 65381 |
| Iterationen | 36 | 36 | 42 | 46 |
| Lösung | 4.1 | 11.9 | 33.1 | 83.7 |
| Stützkräfte | 0.4 | 0.6 | 0.8 | 1.0 |
| Schnittkräfte | 0.7 | 1.2 | 2.1 | 3.8 |
| Netzanpassung | 15.4 | 33.6 | 62.6 | 113.5 |

In den Abbildungen 6.11 und 6.12 ist das Netz nach dem ersten und dritten Verfeinerungsschritt gezeigt. Die Netzverfeinerung erfolgte im Beispiel entsprechend dem Fehlerindikator nach Zienkiewicz und Zhu. Im ersten Schritt wurden 920 Elemente verfeinert. Damit ergeben sich insgesamt 7905 Knoten und 6911 Elemente. Das Koppelproblem wächst auf 445 Koppelknoten. Das Gesamtproblem hat im zweiten Berechnungsschritt 22694 Freiheitsgrade. Nach drei Lösungsschritten ergibt sich ein Problem mit 784 Koppelknoten und 66515 Freiheitsgraden.

## 6.6 Zusammenfassung

Für die lineare statische Analyse von Plattentragwerken mit Hilfe der Methode der finiten Elemente wurden Aspekte der Elementtheorie, der Fehlerabschätzung und der Netzanpassung dargestellt. Die für eine Bearbeitung der Problemstellungen Partitionierung des Berechnungsnetzes, effiziente, parallele Lösung des linearen Gleichungssystems und Netzadaption und Lastneuverteilung gefundenen Softwarelösungen wurden beschrieben.
Für das exemplarisch gewählte Beispiel konnte nachgewiesen werden, daß Probleme dieser Größenordnung sinnvoll und effizient auf dem Parallelrechner unter Einsatz der verschiedenen Gebietszerlegungsverfahren zu lösen sind. Besonders die adaptive und damit mehrfache Berechnung eines Problems mit wachsender Anzahl von finiten Knoten und Elementen läßt sich nur mit Hilfe eines Parallelrechners beherrschen.

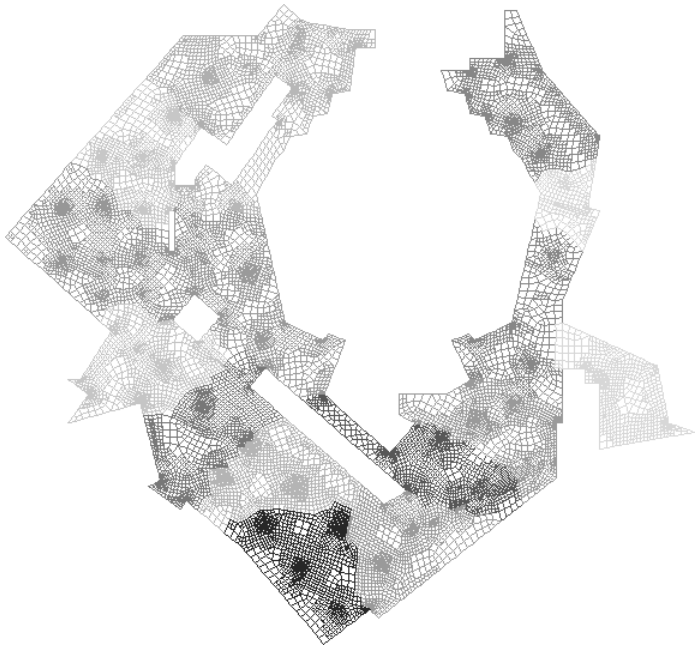

Abbildung 6.12: *Verfeinerung nach dem dritten Berechnungsschritt*

# Kapitel 7

# On-Line Visualisierung

Komplexe Simulationen mit großen Modellen können sehr effizient mit Hilfe von Parallelrechnern durchgeführt werden. Sind diese Simulationen auch zeitabhängig, wird in jedem Zeitschritt ein entsprechend großer Datensatz erzeugt, der dann für die Auswertung zur Verfügung stehen muß. Praktisch verwertbare Simulationen benötigen eine sehr große Zahl von Zeitschritten, um sinnvolle Ergebnisse zu produzieren. Üblicherweise werden diese Daten auf einem Massenspeicher abgelegt, um sie nach dem Simulationslauf mit geeigneter Graphikhard- und -software zu visualisieren. Berechnung und Analyse der Berechnungsergebnisse erfolgen nacheinander und voneinander unabhängig. Man bezeichnet dieses Vorgehen auch als Batch-Modus.
Wissenschaftliche Visualisierungssoftware unterstützt in der Regel diese Herangehensweise. Sie ist oft so strukturiert, daß die Datenauswertung erst erfolgen kann, wenn alle Daten vorliegen. Die Ressourcen an verfügbaren Hauptspeicher und die Verarbeitungsgeschwindigkeit von Parallelrechnern erlauben es aber, Daten in einem Umfang zu produzieren, der die komplette Speicherung aller entstehender Daten sehr teuer, unter Umständen ganz und gar unmöglich macht.
On-line Visualisierung umgeht diesen Speicherengpaß, indem die Daten, sobald sie verfügbar sind, unmittelbar verarbeitet und visualisiert werden. Während ein Datensatz noch dargestellt wird, wird der folgende Datensatz bereits auf dem Parallelrechner berechnet und bereitgestellt. Damit wird zum einen der Einsatz von Massenspeicher zur Datenzwischenspeicherung reduziert und zum anderen ist es möglich, die Auswertung von Ergebnissen und die Beurteilung der Simulationsparameter bereits während der Berechnung selbst durchzuführen. Die Simulationsrechnung selbst kann damit zum interaktiven Prozeß werden. Die unmittelbare Einbeziehung des Ingenieurs in den Ablauf ermöglicht die Einflußnahme auf die Ergebnisqualität noch während der Berechnung. Mit einer interaktiven Simulation können somit kürzere Produktionszeiten erreicht werden. Fehler in der Modellbildung können schneller erkannt werden. Die Simulation kann dann entweder abgebrochen oder mit korrigierten Parametern in Richtung eines sinnvollen Ergebnisses gelenkt werden.
Die on-line Visualisierung stellt außerdem ein unentbehrliches Hilfsmittel für die Programmentwicklung und die Validierung neuer Lösungsstrategien auf Parallelrechnern dar.

## 7.1 Modulare Visualisierungsumgebung

Der gesamte on-line Visualisierungsprozeß kann, wie jeder andere Visualisierungsprozeß auch, in die folgenden fünf Verarbeitungsschritte gegliedert werden:

1. Berechnung

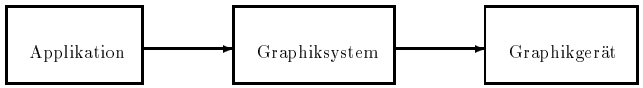


Abbildung 7.1: *Abbildung von graphischen Datenstrukturen (nach [114])*

2. Datenselektion und -filterung
3. Datentransfer (optional)
4. Rendering
5. Darstellung

Berechnung, Datenselektion und -filterung erfolgen mit den Applikations- oder Modelldaten. Aus den Modelldaten lassen sich die geometrischen Daten und die Attribute (Linienart, Farbe) von graphischen Grundelementen des benutzten Graphiksystems ableiten. Der Charakter der Graphikelemente wird durch die Funktionalität des Graphiksystems bestimmt. Eine Unterscheidung ist zum Beispiel in Vektor-, Strich- und Flächen- bzw. in zwei- und dreidimensionale Graphik möglich. Zusätzliche Hierarchiestufen, die aus den graphischen Grundprimitiven zusammengesetzte Graphikobjekte konfigurieren, sind möglich (z.B. mit den 3D-Graphiksystemen PHIGS [75] und HOOPS [217]).
Mit der graphischen Ausgabe schließlich werden die Daten des Graphiksystems auf Daten des angeschlossenen Graphikgerätes abgebildet. Die wesentliche Transformation ist dabei die Umsetzung in zweidimensionale Graphikgerätedaten. Heute übliche Graphikgeräte stellen Rasterpunkte in einer zweidimensionalen Matrix dar. Zunehmend Verbreitung findet aber auch Graphikhardware, die dreidimensionale Graphikelemente direkt unterstützt. Rendering und Darstellung werden intern realisiert. Das Graphiksystem setzt mit seinem Datenmodell direkt auf der Hardware auf (z.B. OpenGL [187]) und führt die Transformation auf den Darstellungsbereich mit spezieller Hardware durch.
Im folgenden wird vorausgesetzt, daß die Berechnung auf dem Parallelrechner ausgeführt wird. Die Modelldaten werden deshalb verteilt bereitgestellt. Für die weiteren Verarbeitungsschritte sind alternative Lösungsvarianten, sowohl auf seriell als auch parallel arbeitenden Rechnern möglich. Die Auswahl der Lösungsvarianten wird von der Architektur des Graphikgerätes, das die Darstellung leistet, bestimmt. Dabei unterscheidet man im wesentlichen zwei Architekturen - die homogene Visualisierungsumgebung mit direkter Kopplung des Graphikgerätes an den Parallelrechner und die heterogene, verteilte Visualisierungsumgebung, bei der keine direkte Kopplung des Parallelrechners an eine Graphikausgabe erfolgt.

### 7.1.1 Homogene Visualisierungsumgebung

Die Berechnungsergebnisse werden

- direkt auf einen an den Parallelrechner angeschlossenen Framebuffer oder
- über die X-Windows-Schnittstelle [165]

ausgegeben.
Ein in das Parallelrechnersystem integrierter Framebuffer erlaubt in der Regel einen hohen Datendurchsatz, bedeutet aber auch eine hohe Hardwareabhängigkeit der gewählten Programmlösung. Der Framebuffer ist zudem nur an dedizierten Arbeitsplätzen verfügbar.



Die Ausgabe an die X-Windows-Schnittstelle ist demgegenüber wesentlich flexibler, hat dabei allerdings einen geringeren, vom Netzwerk und von der Leistungsfähigkeit des X-Servers abhängigen Datenduchsatz.
Beiden Lösungen gemeinsam ist die parallelisierte Renderingphase. Jede Datenquelle liefert direkt Bilddaten, entweder zur Graphikhard- (Framebuffer) oder -software (X-Server). Die Datenselektion (Clipping) erfolgt ebenfalls parallelisiert. Dabei kann unter Umständen ein Lastbalancierungsproblem eintreten, wenn nur einige Prozessoren Beiträge zur Graphikausgabe liefern müssen. Das Datenaufkommen wird von der Auflösung und der Farbtiefe des Bildes bestimmt. Für die Übertragung eines unkomprimierten Bildes berechnet sich der Speicherbedarf aus Anzahl der Bildpunkte $\times$ Farbtiefe der Darstellung. Bei 600 $\times$ 800 Bildpunkten in Echtfarbdarstellung mit 16 Bit Farbtiefe sind das etwa 1 MByte pro Bild. Die Anforderung an die Datenübertragungsrate zum Ausgabegerät ist dabei konstant.

### 7.1.2 Heterogene, verteilte Visualisierungsumgebung

In einer heterogenen, verteilten Visualisierungsumgebung werden die Ergebnisse der Berechnung zu einer entfernten Graphikworkstation transportiert und dort weiterverarbeitet [149], [177], [219]. Dabei kann mit einer geschickten Datenauswahl der Umfang der zu verarbeitenden Modelldaten und die zu übertragende Datenmenge entscheidend reduziert werden. Die Anforderung an die Datenübertragungsrate zum Ausgabegerät sind deshalb variabel.
Der lokale Aufwand und gegebenenfalls das Lastungleichgewicht zwischen den parallel arbeitenden, datenliefernden Prozessen kann erheblich sein [39]. Rendering und Darstellung auf einer Graphikworkstation erlaubt den Einsatz von vorhandener Standardsoftware. Die Chance, ein geeignetes, bereits vorhandenes Softwarepaket einsetzen zu können steigt bei diesem modularen Ansatz. Die Verfügbarkeit der erforderlichen lokalen Graphikrechenleistung kann zunehmend vorausgesetzt werden. Nur in der heterogenen Visualisierungsumgebung ist Datentransfer als separater Schritt erforderlich. Die Konfiguration des verteilten Visualisierungssystems ist wesentlich flexibler als in der homogenen Umgebung mit direkter Kopplung.

### 7.1.3 Datenselektion und -filterung

Die vom Berechnungsprozeß bereitgestellten Modelldaten müssen vor der Datenübertragung geeignet zusammengestellt werden. Dabei ist eine Datenreduktion und Selektion entsprechend der aktuell am Ausgabegerät eingestellten Parameter - Bildausschnitt und Detaillierung - wünschenswert. Dazu ist allerdings dieser Prozeß als Serverprozeß zu konzipieren. Das ist in der Regel nicht ohne Eingriff in die Logik des Berechnungsprogramms möglich.
Die Selektion und Filterung erfolgt dann im Modelldatenraum und mit den realen Modellobjekten. Resultat dieses Arbeitsschrittes ist ein geeignetes Teilmodell. Dieses Teilmodell wird am besten sofort in das Datenmodell des benutzten Graphiksystems überführt. Der Bearbeitungsschritt ist sehr gut parallelisierbar.
Alternativ ist der Verzicht auf eine geometrische, am Bildausschnitt orientierte Modellreduktion denkbar. Geht man davon aus, das im nachfolgenden Bearbeitungsschritten das Graphiksystem wesentlich effizienter im Graphikdatenraum die Datenselektion vornehmen kann, beschränkt man sich auf dem Parallelrechner auf die selektive Transformation der Modelldaten in Graphikdaten. Dieser Ansatz hat den Vorteil, das immer ein vollständiges Modell zum Graphikprozessor übertragen wird. Dieses Modell kann dann dort autark weiterbearbeitet werden. Bei Veränderung des

Bildausschnitts oder der Orientierung des Objekts sind keine weiteren Datentransfers notwendig. Diese Lösung wird deshalb im folgenden vorzugsweise umgesetzt.

### 7.1.4 Datentransfer

Prinzipielles Problem beim Datemtransfer ist es, die Programmodule zunächst geeignet auf die vorhanden Hardware abzubilden und komfortabel und effizient zu verbinden. Der Datentransfer sollte dabei mit möglichst wenig Daten erfolgen. Die Plazierung von Programmen auf wenig belastete Hardware als Betriebssystemfunktion ist ein lediglich in Ansätzen gelöstes Problem. Es werden dazu Erweiterungen der vorhandenen Betriebssysteme erforderlich. Stand der Technik ist eine statische Zuordnung der Programme zu den Hardwareressourcen.
Die Programmodule sind auf die eintreffenden Daten und die Verfügbarkeit von Datensätzen abzustimmen. Das führt zur Synchronisation der Prozesse in der Visualisierungspipeline. Das Reihenfolgeproblem und die Fehlerbehandlung bei verlorenen Datensätzen sind zu lösen. Während in der homogene Visualisierungsumgebung nur das Zusammenfassen der Teilbilder beim Ausgabeprozeß zu einem Gesamtbild zu realisieren ist, muß für die heterogene, verteilte Visualisierungsumgebung ein vollständiges Übertragungsprotokoll inklusive Prozeßinitialisierung und Beendigung zwischen allen Programmen in der Visualisierungspipeline implementiert werden.
Details der Entwicklung eines stabilen Datentransferprotokolls sind im Abschnitt 7.2 beschrieben.

## 7.2 Kommunikationsbibliothek

### 7.2.1 Entwurfskriterien

Als Basis für die Implementierung einer Kommunikationsbibliothek, die portabel, flexibel und effizient einsetzbar sein soll, wurde eine Implementierung auf UNIX-TCP/IP-Sockets gewählt. Die Wahl wurde durch folgende Randbedingungen und Entscheidungskriterien beeinflußt:

1. Als Parallelrechner kommen neben WorkstationClustern unter MPI vor allem Parsytec-Systeme zum Einsatz, die über eine parallelisierbare Schnittstelle zum UNIX-Hostsystem und die Funktionalität von Sockets verfügen.

2. Sämtliche eingesetzte Software mußte als Sourcecode verfügbar sein, um lauffähige Versionen für verschiedene Hardwarekomponenten (Transputer T800, PowerPC 601, SPARC-Prozessoren und R4400 von Silicon Graphics) erzeugen zu können.

3. Unterschiedliche Systemumgebungen und Softwareumgebungen müssen verbunden werden. Es kann kein Werkzeug eingesetzt werden, das nur auf einem Teil der Systeme verfügbar ist (z.B. RPC-Mechanismus). Die Netzwerkunabhängigkeit der Datendarstellung mittels XDR muß explizit eingebaut werden bzw. bei Bedarf zuschaltbar sein. Praktisch ergeben sich in der avisierten Hardwareumgebung Probleme in der Kommunikation mit dem Transputersystem und der Silicon Graphics. SPARC-Prozessoren und PowerPC benutzen die sogenannte Netzwerk-Byteordnung, sind also im wesentlichen kompatibel. Aus diesem Grund sind lediglich vom Transputer und zur Silicon Graphics Datenkonvertierungen erfoderlich.

4. Die Kommunikationsschnittstelle sollte optimale Geschwindigkeit der Datenübertragung bieten. Damit scheiden protokollintensive, plattformübergreifende Mechanismen (ICCC (*Inter Client Communication Convention*), X-Windows, Tooltalk) aus. Diese Mechanismen werden zudem auf den Parallelrechnern zum Teil nur rudimentär unterstützt.



### 7.2.2 Programmierschnittstelle

Die Implementierung erfolgte mit einer Bibliothek, die die Programmierung von Socket-Transportmechanismen realisiert [177]. Dabei sind in der UNIX-Philososphie Server und Client, d.h. Dienstanbieter und Dienstkonsument zu unterscheiden. Das Simulationsprogramm als Datenlieferant wird dabei im folgenden immer als Client arbeiten, während das Visualisierungsprogramm als Server arbeitet. Entsprechend muß das Visualisierungsprogramm zuerst gestartet werden und einen Kommunikationsport anbieten, auf den das Clientprogramm, die eigentliche Simulation, zugreifen kann.
Folgende Funktionen zum Datentransfer stehen zur Verfügung:

**DS_init_socket_write()** Diese Funktion wird zuerst aufgerufen, um einen Kommunikationsport auf der Seite des Clients zu eröffnen. Die Portnummer und der Hostname des Servers (für Internet-Socket) bzw. der Name (für UNIX-Domain-Socket) sind als Parameter anzugeben oder interaktiv einzugeben.

**DS_create_*datatype*_handle()** Mit dieser generischen Funktion werden Strukturen für verschiedene auszutauschen Datenformate initialisiert. *datatype* steht dabei für atomare Daten, ein-, zwei- oder dreidimensionale Felder oder Finite-Element Daten (UCD).

**DS_*prog*_init()** Das ist die eigentliche Initialisierungsroutine zur Kommunikation des Client mit einem bestimmten Servertyp *prog*. Damit werden dem Server die initialisierten Datenformate bekanntgemacht. Außerdem werden Informationen über die Anzahl der gleichzeitig sendenden Datenquellen und die Anzahl der in einer Datenübertragung benutzten Datenformate übertragen.

**DS_send_data_vectors()** Nach erfolgter Initialisierung und der Berechnung der Daten für die Übertragung werden die zuvor initialisierten Datenformate benutzt um die Daten auf der etablierten Socketverbindung an den Server zu schicken. Diese Funktion terminiert, wenn alle Daten erfolgreich übertragen worden sind. Gegebenfalls wird eine unterbrochene Verbindung zum Server wieder aufgebaut.

**DS_close_socket_write()** Wenn alle Daten übertragen sind, wird mit dieser Funktion an den Server das Ende der Datenübertragung gemeldet. Es werden die entsprechenden UNIX-Kommandos zum Schließen der Socketverbindung ausgeführt.

### 7.2.3 Realisierung der Transportschicht

Intern wird für die Datenübertragung der UNIX-Systemruf `write(2V)` benutzt. Probleme ergeben sich bei der Kopplung zwischen Parallelrechner und UNIX-Workstation aus der Abstimmung der beteiligten Softwarekomponenten. Als entscheidend für eine effiziente Datenübertragung erweist sich die Anpassung der Blocklänge beim Sender an die Kapazität der Übertragungskette (vgl. Abbildung 7.2).
Diese Übertragungskette umfaßt die Softwarekomponenten Applikationsprogramm, Message-Passing-Interface, Laufzeitsystem und Routingsystem des Parallelrechners, Schnittstellensoftware zur Umsetzung des spezifischen Protokolls im Parallelrechner auf das Datenprotokoll der UNIX-Workstation und einen Serverprozeß zur Umsetzung von Anforderungen des Parallelrechners in die UNIX-Welt. Ende der Übertragungskette ist wiederum ein Applikationsprogramm auf dem lokalen UNIX-Host

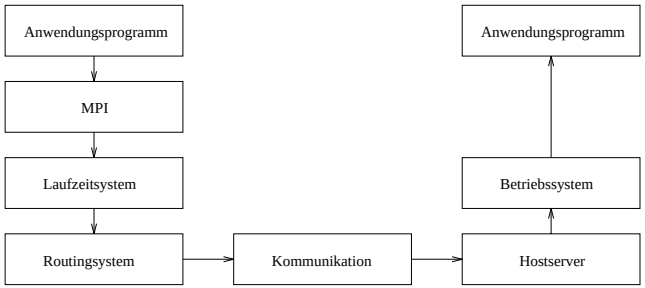


Abbildung 7.2: *Übertragungskette Parallelrechner - Workstation*

(UNIX-Domän-Socket) oder im Netzwerk (Internet-Socket). Dabei sind mindestens im Routingsystem und im Hostserver zusätzliche Kopien der zu übertragenden Daten anzulegen. Um den Speicherplatz effizient auszulasten und trotzdem einen flexiblen Übertragungsmechanismus zu realisieren, wird deshalb eine Nachricht in Blöcke geeigneter Länge gesplittet.
Diese Blocklänge ist in der Regel von der Hardwarecharakteristik abhängig und vom Hersteller der Serversoftware auf einen optimalen Wert fest eingestellt. Untersucht man die erreichbare Datenübertragungsrate für verschiedene Blocklängen (vgl. Abbildung 7.3) wird die Abhängigkeit zwischen gewählter Paketlänge und der maximal erreichbaren Übertragungsrate deutlich. Dabei überlagert sich der Einfluß von nicht zusammenpassender Blocklänge im absendenden Applikationsprogramm und beim UNIX-Systemruf `write(2V)` (Sägezahnkurven für Blocklängen kleiner 8 kByte) mit dem Einfluß der optimalen Blocklänge für das Serverprogramm (Einbrüche in den Kurven für die Blocklängen 8 kByte und 10 kByte). Die Auswahl einer beliebig großen Blocklänge wird vom notwendigen Speicherplatzbedarf im Routingsystem beschränkt.
Variiert man die verwendeten Datenblockgrößen im Bereich oberhalb der vom Serverprogramm benutzten Blocklänge ergeben sich Übertragungsraten entsprechend Abbildung 7.4. Verglichen wird jetzt mit der erreichbaren Übertragungsrate, wenn bereits beim Aufruf der Systemfunktion `write(2V)` im Applikationsprogramm die Nachricht in die 8 kByte Blöcke zerlegt wird.
Es fällt auf, daß sich für bestimmte Nachrichtenlängen im Applikationsprogramm bessere Übertragungsraten erzielen lassen. Diese Nachrichtenlängen korrespondieren zu Vielfachen der Blocklänge im Routingsystem. Allerdings ist die durchschnittlich erreichbare Übertragungsrate nicht wesentlich besser als die mit der festen Blocklänge von 8 kByte erzielbare.

## 7.3 Übertragene Daten

### 7.3.1 Datentypen

Der Umfang und die Struktur der übertragenen Daten ist prinzipiell erweiterbar und an individuelle Bedürfnisse anpaßbar. Zunächst wurde Wert auf eine einfache Implementierung und effiziente, in der Regel auch komprimierte Datenhaltung gelegt. Von der Datenrepräsentation der übertragenen Daten wird in der vorliegenden Implementierung nicht abstrahiert. Die hardwareunabhängige Datenrepräsentation (entsprechend XDR-Spezifikation) kann aber relativ einfach ergänzt werden. Fol-



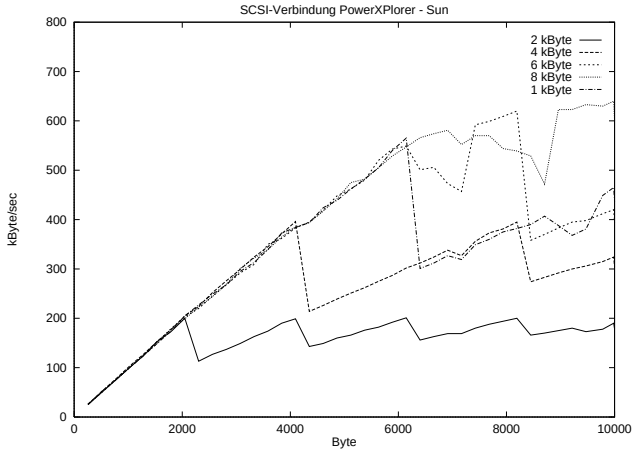


Abbildung 7.3: *Übertragungsrate zwischen Parallelrechner und Workstation in Abhängigkeit von der Blocklänge in der Transferbibliothek*

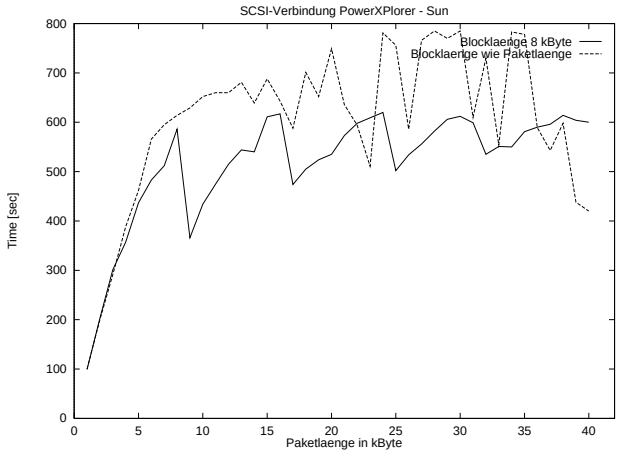


Abbildung 7.4: *Übertragungsrate zwischen Parallelrechner und Workstation in Abhängigkeit von der Nachrichtenlänge*

Tabelle 7.1: *Paketstruktur (8 Byte Tag mit variabel langem Datenteil*

| Eintrag | Länge | Inhalt |
|---|---|---|
| Tag | 12 Bit | Kontrolle, Initialisierung, Daten |
| Tagtyp | 4 Bit | Start, Ende, nur Header, mit Daten |
| Daten | 4 Bit | Daten, Steuerinformation |
| Speichertyp | 4 Bit | Atom, 1D, 2D, 3D, UCD[1] |
| Datentyp | 8 Bit | Enumerator |
| Datenlänge | 4 Byte | Länge des Datenfelds in Byte |
| Datenfeld | variabel | Datenfeld entsprechender Datenlänge |

gende Datentypen sind implementiert und können übertragen werden:

**Elementare Daten** Das sind Steuerinformationen, die eine feste Länge haben, z.B. Informationen über Darstellungsparameter wie Anzahl von Prozessoren, Transformationsparameter usw. Die Problematik liegt in der Kodierung der Bedeutung dieser Informationen.

**Datenfelder** Datenfelder werden als Kopie eines fortlaufenden Speicherbereiches mit vorher festgelegter Struktur übertragen. Die Länge wird im Paketkopf vereinbart. Implementiert sind Feldtypen, d.h. fortlaufend gleiche Elemente eines elementaren Typs (int16, int32, real, real8, char, complex8, complex16, logical). Die Datenfelder haben eine logische Struktur, die geeignet ist, Teilinformationen direkt zu extrahieren.

**Strukturfelder** Strukturfelder werden wie Datenfelder als Kopie eines fortlaufenden Speicherbereiches übertragen. Die Strukturelemente haben aber nicht mehr den gleichen elementaren Typ, sondern beschreiben mit unterschiedlichen Datentypen eine hierarchische Datenstruktur (z.B. Finite-Element Netz mit Elementtoplogie und Knotengeometrie). Die logische Struktur ist geeignet, Teilinformationen zu extrahieren bzw. Informationen von parallelen Datenquellen zusammenzusetzen.

Thornborrow [201] führt einen weiteren Datentyp ein. Es handelt sich um einen Datentyp variabler Länge, der bei der parallelen Erzeugung von Daten entsteht. Das ist zum Beispiel der Fall, wenn eine Selektionsoperation auf allen Prozessoren ausgeführt wird. Die Ergebnisdaten werden auf jedem Prozessor bereitgestellt und sind in ihrem Umfang vom Selektionskriterium abhängig. Es kann passieren, daß einige Prozesse sehr wenig Daten beisteuern, andere dagegen sehr stark einbezogen sind. Das Festlegen der Größe der zu übertragenden Daten kann im Prinzip erst nach der Berechnung der Daten in einem globalen, synchronisierenden Austausch erfolgen. Damit handelt man sich allerdings erhebliche Synchronisationsverluste ein. Der Vorteil, der in der ungleichmäßigen Auslastung der Prozessoren liegt, indem die weniger belasteten Prozesse bereits nachfolgende Anfragen bearbeiten können, wird zunichte gemacht. Deshalb werden die Daten im Datenstrom unbekannter Länge übertragen.

### 7.3.2 Transportprotokoll

Die Datenübertragung wird vollständig in einer Transportschicht realisiert. Diese Transportschicht bedient sich einer festgelegten Paketstruktur (Tabelle 7.1).

[1] Unstructured Cell Data



Die Paketstruktur der Transportschicht erlaubt einen effizienten und flexiblen Austausch verschiedenster Ergebnisdaten. In jedem Datenpaket werden Informationen über den Inhalt und die Länge des Datenfelds übertragen. Damit ist eine einfache Kontrolle des vollständigen Datenaustausches möglich. Der empfangende Prozeß kann anhand des eingehenden Pakets sofort den Inhalt der Daten identifizieren und muß nicht zuvor mit zusätzlichen Informationen initialisiert werden. Außerdem wird es damit möglich, in der Reihenfolge ihres Eintreffens vertauschte Datenpakete zu identifizieren und beim Empfänger wieder in die richtige Reihenfolge zu sortieren. Zur Realisierung des asynchronen Datenaustausches werden folgende Pakettypen verwendet:

**Kontrolldaten** zum Aufbau der Nachrichtenverbindung, zur Fehlerbehandlung und zur Identifikation der Prozesse

**Anwendungsdaten** zur Übertragung der Modellbeschreibung, einschließlich Absenderidentifikation, Datum, Strukturbeschreibung der Daten durch Identifikatoren und Zuordnungsvorschriften bei Datenquellen von parallel arbeitenden Prozessen

**Steuerdaten** zur Initialisierung und zur Kommunikation mit dem Visualisierungsprozeß

## 7.4 Beispielanwendung mit AVS

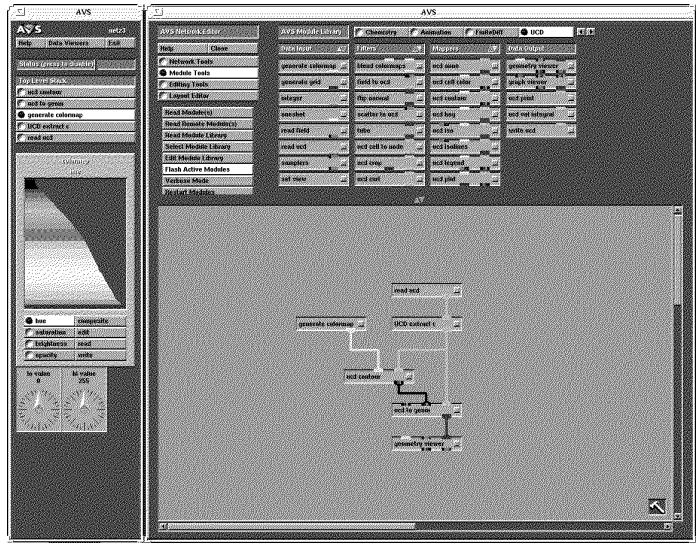

Abbildung 7.5: *Advanced Visualization System, links Kontrollfeld, oben Modulbibliothek, unten Netzwerkeditor*

Das Visualisierungspaket *Advanced Visualization System* AVS [215],[216] ist ein modular aufgebautes Softwaresystem. Auf der Basis eines AVS-Kernsystems sind eine

Vielzahl von AVS-Modulen für

- Datenfilterung und -konvertierung,
- Abbildung von Daten auf geometrische Grundprimitive und
- Darstellung von geometrischen Grundprimitiven auf graphischen Ausgabegeräten

realisiert. Die Modulpalette kann unter Benutzung der mit AVS mitgelieferten Entwicklungsumgebung erweitert und an spezielle Bedürfnisse angepaßt werden.
Die Visualisierungspipeline wird üblicherweise mit Hilfe eines graphischen Benutzerinterfaces, hier dem Netzwerkeditor (engl. *network editor*) von AVS, konfiguriert. Der Netzwerkeditor ist eine X-Windows Applikation, die direkt über den Windowmanager kommuniziert. Dabei werden die Funktionen von X11 benutzt, um proprietäre Bedienelemente zu implementieren. AVS-Module, d.h. Programme, deren Ein- und Ausgabefunktionen mit Hilfe der AVS-Bibliothek als sogenannte Ports beschrieben wurden, können mit diesem Werkzeug ausgewählt und zusammengestellt werden (siehe Abbildung 7.5).
Der Netzwerkeditor interpretiert bei der Auswahl der Module die Beschreibung der Ports. Der Anwender kann zueinander kompatible Ports verbinden. Damit werden Datenkanäle gebildet. Es entsteht ein Netzwerk, in dem die ausgewählten Module in einer Pipeline die eingehenden Daten verarbeiten und in der gewünschten Art und Weise weitergeben. AVS erlaubt für alle Module eine Steuerung der Parameter in einer graphischen Benutzeroberfläche.

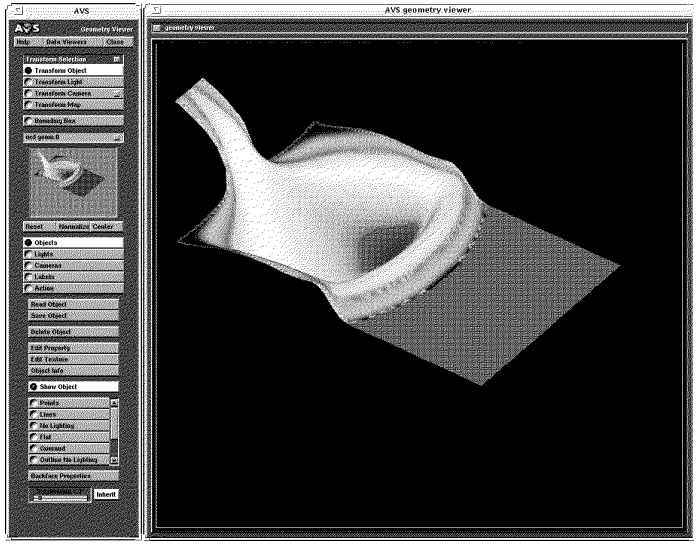


Abbildung 7.6: *Advanced Visualization System, Strömungsvisualisierung mit* geometry viewer

Am Ende der Pipeline steht in der Regel ein Visualisierungsmodul. Für Finite-Element-Ergebnisse geeignet ist zum Beispiel das Modul *AVS geometry viewer*. Für



das Beispiel der Strömungssimulation in Kapitel 4 ist in Abbildung 7.6 die Ausgabe der Wasserhöhe im Berechnungsgebiet in einem Zeitschritt dargestellt. Dabei wird die Wasserhöhe als dritte Koordinate und der Betrag der Ausbreitungsgeschwindigkeit als Farbverlauf in einer sogenannten 4D-Darstellung visualisiert.
Andere Visualisierungsmodule von AVS erlauben die Darstellung und Manipulation von Pixelgrafik und die Darstellung von zweidimensionalen Funktionsgraphen. Die Bedienung dieser Applikationen kann alternativ über die graphische Benutzeroberfläche des Netzwerkeditors oder in einem Kommandozeilenmodus als stand-alone Lösung erfolgen.
Die Synchronisation der Datenübertragung wird mit den AVS eigenen Ein- und Ausgabefunktionen transparent realisiert. Der Entwickler von AVS-Modulen muß lediglich die erfoderlichen AVS-Datenstrukturen inklusive der Steuerungsinformationen in seiner Anwendung generieren.
AVS kann in einem heterogenen Netzwerk arbeiten, wobei die miteinander kommunizierenden Module auf verschiedenen Hardwareplattformen arbeiten. Es bietet damit ideale Voraussetzungen für eine heterogene Visualisierungspipeline.
Die Datenrepräsentation im Visualisierungsmodell unterscheidet sich von der Datenrepräsentation im Simulationsmodell. AVS benutzt definierte Datenschnittstellen und stellt Funktionen zum Aufbau von AVS-Datenfeldern bereit. Diese Funktionen sind aber nur innerhalb einer AVS-Applikation mit dem AVS-Programmierinterface benutzbar. Sie sind mit einer normalen Nutzerlizenz nicht im Quelltext verfügbar und können damit nicht auf einen Parallelrechner portiert werden[2]. Das Austauschdatenformat wurde deshalb so gewählt, daß auf dem Parallelrechner die Datenfelder entsprechend der dann vom AVS-Modul zu benutzenden Parameter aufgebaut werden und nach dem Einlesen und Zusammensetzen ohne zusätzliche Konvertierung direkt für das AVS-Programmierinterface benutzt werden können.

### 7.4.1 Unstrukturierte Daten - Finite Element Daten

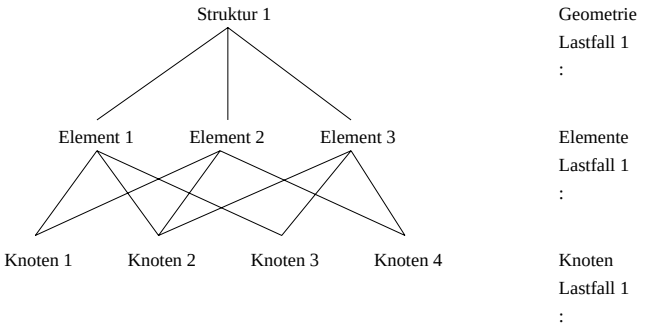


Abbildung 7.7: *Datenstruktur für Finite Element Daten in AVS*

Die Definition der implementierten Strukturfelder für Finite-Element Daten orientiert sich an der Strukturierung des Datentyps Unstructured Cell Data des Visualisierungssystems AVS. In dieser hierarchischen Datenstruktur werden die Topologie

[2] In [212] wird die von Thinking Machines für die Connection Machine CM-5 entwickelte parallele Version von AVS vorgestellt. Die Implementation benutzt intern die Kommunikationsbibliothek der Connection Machine anstatt des AVS-Kommunikationsmechanismus.

der finiten Elemente, die Knotenkoordinaten und auf die Knoten bzw. Elemente bezogene Ergebnisdaten, geordnet nach Lastfällen, unterstützt (vgl. Abbildung 7.7). Die Knotenkoordinaten werden als 2D- bzw. 3D-Daten gemeinsam mit ihren Extremwerten (Boundingbox) verwaltet. Jeder Knoten hat eine Identifikation, die sich aus der lückenlos aufsteigende Numerierung der Knotenkoordinaten ableitet. Auf diese Identifikation referenziert die Beschreibung der Elementtopologie. Elemente werden sogenannten Strukturen zugeordnet. Die Lastfalldaten sind auf die Reihenfolge der Definition von Strukturen, Knoten bzw. Elementen abgestimmt. Das Datenformat vermeidet damit den Verwaltungsaufwand für ein Namensschema, indem durch implizite Namensgebung eine Referenzierung erlaubt wird. Damit ist aber zum Beispiel das Übertragen von Ergebnisdaten nur für ausgewählte Knoten nicht möglich. Die Daten werden als Strukturfeld übertragen, vor dem Transfer ist ein fortlaufender Speicherbereich mit den erforderlichen Daten zu beschreiben.
Die Erstellung der AVS-Datenstrukturen erfolgt direkt auf dem Parallelrechner. Dabei werden die internen Finite-Element Datenstrukturen in AVS-Strukturen konvertiert. Dieser Ansatz ist unter folgenden Gesichtspunkten nicht ideal:

- Nur ein begrenzter Ausschnitt von möglichen Finite-Element Repräsentationen ist übertragbar.
- Die Nutzerdatenstrukturen sind nicht direkt als AVS-Datenstrukturen nutzbar, es ist ein ressourcenintensives Kopieren und Organisieren der Daten erforderlich.
- Adaptivität, unterschiedliche Ansatzfunktionen, inkompatible Elementübergänge, unterschiedliche Elementklassen in einem Modell sind in der Regel nicht darstellbar.

Über die Probleme der Konvertierung bei analoger Herangehensweisen unter Nutzung von AVS wird zum Beispiel in [186] und [219] berichtet. In [149] wird die Verwendung des Visualisierungstools GRAPE [180], [179] beschrieben. Auf der Basis von GRAPE wird in [181] ein alternativer Ansatz zur Datenaufbereitung beschrieben. Dabei hat der Anwender von GRAPE einen Satz von Zugriffsfunktionen auf seine interne Datenstruktur bereitzustellen. Das Visualisierungstool benutzt dann diese prozedurale Schnittstelle, um auf die interessierenden Daten zuzugreifen. Die Problematik der Anbindung eines Parallelrechners mit verteilten Datenanforderungen wird dabei allerdings nicht berücksichtigt.

### 7.4.2 Realisierung im Anwendungsprogramm

Eine Beispielimplementierung im Anwendungsprogramm hat für die Übertragung von Finite-Element Daten mit der UCD-Datenstruktur folgende Gestalt:

```
DS_init_socket_write(&sock, port, host);

DS_create_UCD_handle(&handle,
    DS_REAL, "ucd", CPU_ID, knoten, zellen,
      ndata, cdata, mdata, nodes_element, comp,
    data);

DS_AVSinit(*sock, TOTAL_CPUS, CPU_ID,
           (int *) datavector, NUMBER_OF_DATASETS);

DS_send_data_vectors(*sock, CPU_ID,
           (int *) datavector, NUMBER_OF_DATASETS, iteration);
```



```
DS_close_socket_write(sock);
```

Anhand der Identifikatoren kann der empfangende Prozeß die eintreffenden Daten den entsprechenden Datenquellen und Zeitschritten zuordnen. Das Protokoll der Datenübertragung sichert einen fehlerfreien und vollständigen Datentransport, der gegebenenfalls wiederholt wird, wenn eine Unterbrechung der Verbindung erfolgte. Die Zusammenfassung der ursprünglich verteilt vorliegenden Daten erfolgt auf dem Parallelrechner. Damit wird die schnellere Kommunikation in der homogenen Parallelrechnerumgebung benutzt. Die Daten werden dann über die beschriebene Schnittstelle an den Serverprozeß geschickt, der als Teil des AVS-Netzwerks konfiguriert ist und auf eintreffende Nachrichten wartet.
Mit der Programmierumgebung EXPRESS [186] wird ein ähnlicher Weg beschritten. Als Programmierschnittstelle für den Parallelrechner wird *PVM* [51] benutzt. Ursprünglich für eine reine Workstationanwendung entwickelt, ist auch der Serverprozeß im AVS-Netzwerk Teil der virtuellen parallelen Maschine (PVM). Die Kommunikation im Parallelrechner wird in dieser Lösung direkt im AVS-Server synchronisiert. Diese Herangehensweise ist interessant, wenn der Datendurchsatz im heterogenen Verbund von Parallelrechner und Graphikworkstation ausreichend hoch ist. Das ist für die Anbindung des PowerPC-Systems nicht der Fall.
Für die in Kapitel 4 beschriebene, parallelisierte, zeitabhängige Strömungssimulation wurde die hier angegebene Implementierung benutzt, um den Strömungsverlauf auf einer Graphikworkstation on-line zu visualisieren. Die Berechnung wurde auf dem PowerXPlorer durchgeführt. Das beschriebene AVS-Netzwerk ist um ein weiteres Modul zur Ausgabe von Pixelgrafik im MPEG-Format ergänzt worden. Damit ist es dann möglich, in einem weiteren Verarbeitungsschritt, Videoaufzeichnungen der zeitabhängigen Simulation zu produzieren. Einzelheiten zur verwendeten Software- und Hardwarelösung sind in [34] angegeben.

## 7.5 Zusammenfassung

On-line Visualisierung in einer heterogenen, verteilten Umgebung ermöglicht die Ausnutzung lokaler Rechnerressorcen und die direkt Ergebnisbeurteilung am Arbeitsplatz des Nutzers. Mit bestehender Netzinfrastruktur wird die tatsächliche Lokalisation der einzelnen Ressourcen zum untergeordneten Problem. Zu lösen sind die Probleme

- der Plazierung der Softwarekomponenten,
- des Transportprotokolls,
- des fehlerredundanten Datenaustausches und
- der Auswahl der geeigneten Visualisierungssoftware.

Beispielhaft wurde die Lösung dieser Probleme unter Nutzung von vorhandener Visualisierungssoftware gezeigt. Die Effizienz der Lsöung wird wesentlich vom Datendurchsatz zwischen Parallelrechner und Graphikworkstation bestimmt. Wünschenswert sind skalierbare Übertragungsraten, die der Skalierbarkeit der Rechenleistung eines Multiprozessorsystems entsprechen.

# Kapitel 8

# Ausblick

Statt einer Zusammenfassung der in den einzelnen Abschnitten bereits vorgenommenen Wertung soll hier in einem Ausblick noch einmal die Relevanz der Parallelverarbeitung für den Bauingenieur mit vier Thesen herausgestellt und unterstrichen werden:

1. Paralleles Rechnen ist auf dem Weg zur praktischen Anwendung. Dabei werden im Bauwesen vor allem die Anwendungen der physikalisch und geometrisch nichtlinearen Strukturanalyse, Kontakt-, Crash- und Crushprobleme im Mittelpunkt stehen. Die algorithmischen Grundlagen sind im Bereich der Gleichungslöser in Zusammenarbeit mit der Mathematik für ausgewählte Problemklassen gelegt. Problematisch ist die effiziente Unterstützung von Algorithmen, die eine hohe dynamische Veränderlichkeit der Berechnungsanforderungen im simulierten Modell aufweisen. Das Problem der dynamische Lastanpassung ist noch unzureichend bearbeitet.

2. Bevorzugte Anwendungsfelder des parallelen Rechnens lassen sich dort definieren, wo heutzutage mit der zur Verfügung stehenden Rechentechnik nur unzureichende Problemgrößen bearbeitet werden können und in denen die Verfeinerung der Modelle eine bessere Widerspiegelung des physiklaischen Sachverhaltes liefert. Das ist im Bereich der linearen Strukturmechanik nicht gegeben, gilt aber uneingeschränkt zum Beispiel für die geotechnische und Grundwassermodellierung.

3. Die Anforderungen an geeignete Software zur Parallelisierung sind fomuliert. Das Defizit an Entwicklungswerkzeugen für Parallelrechner verschwindet zunehmend. Effiziente Programmierparadigmen sind mit dem Message-Passing-Konzept vorhanden. Es wird sich in der Zukunft zeigen, ob Alternativkonzepte mit einem höheren Abstraktionsgrad von der parallelen Hardware vergleichbar leistungsfähige und effiziente Lösungen erlauben.

4. Die Parallelverarbeitung hat auch im Bauingenieurwesen eine Zukunft. So muß zum Beispiel die Parallelverarbeitung Einzug in die moderne Bauinformatiklehre halten, besonders bei einer algorithmischen und numerischen Ausrichtung der Lehrinhalte. Auf der Basis moderner FE-Verfahren ist die Parallelverarbeitung unverzichtbarer Bestandteil des Repertoirs zur Erstellung effizienter Ingenieursoftware.

# Literaturverzeichnis

[1] L. Adams, *m-step preconditioned conjugate gradient methods*, SIAM J. Sci. Stat. Comput. **6** (1985), Nr. 2, 452–463.

[2] M.A. Ajiz und A. Jennings, *A robust incomplete Cholesky-conjugate gradient algorithm*, International Journal for Numerical Methods in Engineering **20** (1984), 949–966.

[3] M. Al-Nasra und D.J. Nguyen, *An algorithm for domain decomposition in finite element analysis.*, Computers & Structures **39** (1991), 227–289.

[4] R. Alasdair, A. Bruce, James G. Mills und A. Gordon Smith, *CHIMP version 2.0 user guide*, Tech. Report EPCC-KTP-CHIMP-V2-USER, Edinburgh Parallel Computing Centre, University of Edinburgh, Marz 1994.

[5] G.S. Almasi und A. Gottlieb, *Highly parallel computing*, 2 Ausg., Benjamin Cummings series in computer science and engineering, Benjamin Cummings Publ. Company, Inc., Redwood City, CA, 1994.

[6] G.M. Amdahl, *Validity of the single-processor approach to achieving large-scale computer capabilities.*, Proc. AFIPS conf., 1967, S. 483–485.

[7] C. Ashcraft, S. Eisenstat und J. Liu, *A fan-in algorithm for distributed sparse numerical factorization*, SIAM Journal of scintific and statistical computations (1990), 593–599.

[8] B. Asprall und J.R. Gilbert, *Graph coloring using eigenvalue decomposition*, SIAM J. Algebraic discrete methods **5** (1984), 526–538.

[9] O. Axelsson, *Iterative solution methods*, Cambridge University Press, 1994.

[10] I. Babuska, *The p- and hp-Version of the finite element method. State of the art*, Finite Elements. Theory and Application (R. G. Voigt D. L. Dwoyer, M. Y. Hussaini, Hrsg.), Springer Verlag, 1988.

[11] I. Babuska und E. Rank, *An expert system for optimal mesh design in the hp-version of finite element method*, International Journal for Numerical Methods in Engineering **24** (1987), 2087–2106.

[12] I. Babuska, B. A. Szabo und I. N. Katz, *The p-version of the finite element method*, SIAM J. Num. Anal. **18** (1981).

[13] I. Babuska und B.A. Szabo, *On the rates of convergence of the finite element method*, International Journal for Numerical Methods in Engineering **18** (1982), 323–343.

[14] V. Bala, S. Kipnis, L. Rudolph und M. Snir, *Designing efficient, scalable, and portable collective communication libraries*, Tech. Report, IBM T.J. Watson Research Center, Oktober 1992.

[15] S.T. Barnard und H.D. Simon, *A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems*, Concurrency: Practice and Experience **6** (1994), Nr. 2, 101–117.

[16] E.R. Barnes und A.J. Hoffmann, *Partitioning, spectra and linear programming.*, Tech. Report RC 9511 (No. 42058), IBM T.J. Watson Research Center, 1982.

[17] M. Barnett, R. Littlefield, D.G. Payne und R. van de Geijn, *On the efficiency of global combine algorithms for 2d-meshes with wormhole routing*, Tech. Report, Univ. of Texas at Austin, 1993.

[18] M. Barnett, D.G. Payne und R. van de Geijn, *Optimal broadcasting im mesh-connected architectures*, Tech. Report, Univ. of Texas at Austin, Dezember 1991.

[19] M. Barnett, D.G. Payne, R. van de Geijn und J. Watts, *Broadcasting on meshes with wormhole routing*, Tech. Report, Univ. of Texas at Austin, November 1993.

[20] P. Bartelt, *Finite element procedures on vector/tightly coupled parallel computers*, Diss., ETH Zürich., 1989.

[21] J.-L. Batoz, K.J Bathe und W.H. Ho, *A study of three-node triangular bending element*, International Journal for Numerical Methods in Engineering **15** (1980), 1771–1812.

[22] J.-L. Batoz und M.B. Tahar, *Evaluation of a new quadrilateral thin plate bending element*, International Journal for Numerical Methods in Engineering **18** (1982), 1655–1677.

[23] K.P. Belkhale und P. Prithviraj, *Recursive partitions on multiprocessors*, 5th Distributed Memory Computing Conference, 1990, S. 930–938.

[24] J. Bellmann und E. Rank, *Die p- und hp-Version der Finite-Elemente-Methode oder: Lohnen sich höherwertige Elemente*, Bauingenieur 64 (1989), 67–72.

[25] M.J. Berger und S.H. Bokhari, *A partitioning strategy for nonuniform problems on multiprocessors*, IEEE Transactions on Computers **C-36** (1987), Nr. 5, 570–580, mesh partitioning.

[26] J.C. Bermond und C. Peyrat, *De Bruijn and Kautz networks: a competitor for the hypercube?*, 1st European Workshop on Hypercube and Distributed Computers, North Holland, F. Andre and J.P. Verjus, 1989, S. 279–293.

[27] J.E. Boillat, *Load balancing and poisson equation in a graph*, Concurrency: Practice and Experience **2** (1990), 289–313.

[28] S.H. Bokhari, *On the mapping problem*, IEEE Transactions on Computers **3** (1981), 207–213.

[29] L. Bomans und R. Hempel, *The argonne/gmd macros in fortran for portable parallel programming and their implementation on the Intel iPSC/2.*, Parallel Computing **15** (1990), 119–132.

[30] J. Bowen, *Hypercubes*, Practical Computing **5** (1982), Nr. 4, 97–99.

[31] D. Braess, *Finite Elemente*, Springer Verlag, Berlin, Heidelberg, New York, 1991.



[32] C. Bremer, *Algorithmen zum effizienten Einsatz der Finite-Element-Methode*, Diss., Braunschweig, Techn. Univ., 1986.

[33] R. Brent, A. Cleary, M. Hegland, J. Jenkinson, Z. Leyk, M. Osborne, P. Price, S. Roberts, D. Singleton und M. Nakanishi, *Implementation and performance of scalable scientific library subroutines on Fujitsu VPP500 parallel-vector supercomputer*, Scalable High-Performance Computing Conf. 1994 (Australian National University, Fujitsu), IEEE Computer Society Press, Mai 1994, S. 526–533.

[34] M. Burghardt, *Visualisierung der Ergebnisdaten von parallelen FE-Berechnungen auf Workstations*, Dipl.arbeit, TH Darmstadt, 1995.

[35] R. Butler und E. Lusk, *Monitors, messages, and clusters: the p4 parallel programming system*, Tech. Report, Argonne National Laboratory, Mathematics and Computer Science Division, ANL, Argonne, IL 60439, 1992.

[36] R. Calkin, R. Hempel, H.-C. Hoppe und P. Wypior, *Portable programming with the PARMACS message-passing library.*, Parallel Computing **20** (1994), Nr. 4, 615–632.

[37] P. Carlin, C. Kesselman und K.M. Chandi, *The compositional C++ language definition*, Tech. Report CS-TR-92-02, California Institute of Technology, Pasadena CA, 1992.

[38] U. D. Carlini und U. Villano, *The routing problem in transputer-based parallel systems*, Microprocessors and Microsystems **15** (1991), Nr. 1.

[39] J.A. Challinger, *Scalable parallel direct volume rendering for nonrectilinear computational grids*, Diss., Computer and Information science, Univ. of California, Santa Cruz, Dezember 1993.

[40] N. Chrisochoides, E. Houstis und J. Rice, *Mapping algorithms and software environment for data parallel PDE iterative solvers*, Parallel and Distributed Computing **21** (1994), Nr. 1.

[41] N. Chrisochoides, N. Mansour und G. Fox, *Performance evaluation of load balancing algorithms for parallel single-phase iterative PDE solvers*, Scalable High-Performance Computing Conf. 1994, IEEE Computer Society Press, Mai 1994, S. 764–772.

[42] M. Cosnard und D. Trystram, *Parallel algorithms and archtectures*, International Thomson Computer Press, London, 1995.

[43] A.L.G.A. Coutinho, J.L.D. Alves, N.F.F. Ebecken und L.M. Troina, *Conjugate gradient solution of finite element equations on the IBM 3090 vector computer utilizing polynomial preconditionings.*, Computer Methods in Applied Mechanics and Engineering **84** (1990), 129–145.

[44] D.M. Cvetkovic, M. Doob und H. Sachs, *Spektrum von Graphen*, Akademie-Verlag, Berlin, 1981.

[45] G. Cybenko, *Dynamic load balancing for distributed memory multiprocessors*, Parallel and Distributed Computing **7** (1989), 279–301.

[46] W. J. Dally und C. Seitz, *Deadlock free message routing in multiprocessor interconnection networks*, IEEE Transactions on Computers **C-36** (1987), Nr. 5, 547–553.

[47] R. Das, D.J. Mavriplis, J. Saltz, S. Gupta und R. Ponnusamy, *The design and implementation of a parallel unstructured euler solver using software primitives*, Tech. Report 92-12, ICASE NASA Langley Research Centre, Hampton, VA 23665-5225, Marz 1992.

[48] R. Davey, J. Parker, M. Parsons und M. Sawyer, *Unstructured mesh partitioning and improvement on the AP1000*, 4th International Parallel Computing Workshop, Imperial College London, 1995, S. 219–223.

[49] M. Debbage, M. Hill und D. Nicole, *Virtual channel router-Version 2.0 User Guide*, Tech. Report PUMA-033, University of Southampton, Oktober 1991.

[50] R. Diekmann, B. Monien und R. Preis, *Using helpfull sets to improve graph bisections.*, Tech. Report RF-008-94, Universität Paderborn, Juni 1994.

[51] J. Dongarra, A. Geist, R. Manchek und V. Sunderam, *Integrated PVM framework supports heterogeneous network computing*, Computers in Physics **7** (1993), Nr. 2, 166–175.

[52] J.J. Dongarra und T.H. Dunigan, *Message-passing performance of various computers*, Tech. Report ORNL/TM-13006, ORNL, Februar 1996.

[53] R.v. Driesche und D. Roose, *Dynamic load balancing of iteratively refined grids by an enhanced spectral bisection algorithm*, Workshop Dynamic load balancing on MPP systems, Daresbury, November 1995.

[54] R. Van Driessche und D. Roose, *Dynamic load balancing with a spectral bisection algorithm for the constrained graph partitioning problem*, High-Performance Computing and Networking (B. Hertzberger und G. Serazzi, Hrsg.), LNCS 919, Springer, 1995, S. 392–397.

[55] J. Dubois, A. Greenbaum und G. Rodrigue, *Approximating the inverse of a matrix for use in iterative algorithms on vector processors*, Computing **22** (1979), 257–268.

[56] I. Duff, *Parallel implementation of multifrontal schemes*, Parallel Computing (1986), 193–204.

[57] T.H. Dunigan, *Early experiences and performance of the Intel paragon*, Tech. Report ORNL/TM-12194, ORNL, Oktober 1994.

[58] C. Farhat, *A simple and efficient automatic FEM domain decomposer*, Computers & Structures **28** (1988), Nr. 5, 579–602.

[59] C. Farhat und M. Lesoinne, *Automatic partitioning of unstructured meshes for the parallel solution of problems in computational mechanics*, International Journal for Numerical Methods in Engineering **36** (1993), 745–764.

[60] C. Farhat und F.-X. Roux, *A method of finite element tearing and interconnecting and its parallel solution algorithm.*, International Journal for Numerical Methods in Engineering **32** (1991), Nr. 1, 1205–1227.

[61] ———, *Implicit parallel processing in structural mechanics.*, Computational Mechanics Advances **2** (1994), Nr. 1, 1–124.

[62] J. Favenesi, A. Danial, J. Tombrello und J. Watson, *Distributed finite element analysis using a transputer network*, Computing Systems in Engineering **1** (1991), Nr. 2-4, 171–182.



[63] M. Fiduccia und R.M. Mattheyses, *A linear time heuristic for improving network partitions*, Proc. 19th IEEE Design Automation Conf., 1982, S. 175–181.

[64] M. Fiedler, *Algebraic connectivity of graphs*, Czech. Math. J. **98** (1973), Nr. 23, 298–305.

[65] ———, *A property of eigenvectors of nonnegative symmetric matrices and its applications to graph theory*, Czech. Math J. **25** (1975), Nr. 100, 619–633.

[66] J. Flowera, S. Otto und M. Salama, *Optimal mapping of irregular finite element domains to parallel processors*, Parallel computations and their impact on mechanics (A.K. Noor, Hrsg.), Vol. 86, ASME, New York, 1987, S. 239–252.

[67] M.J. Flynn, *Very high speed computing systems*, Proc. of the IEEE **54** (1966), Nr. 12, 1901–1909.

[68] Message Passing Interface Forum, *Document for a standard message-passing interface*, Tech. Report CS-93-214, University of Tennessee, November 1993, Available on **netlib**.

[69] G.C. Fox, M.A. Johnson, G.A. Lyzenga, S.W. Otto, J.K. Salmon und D.W. Walker, *Solving problems on concurrent processors*, Prentice Hall, Englewood Cliffs, NJ, 1988.

[70] G.C. Fox, R.D. Williams und P.C. Messina (Hrsg.), *Parallel computing works!*, Morgan Kaufman, San Francisco, CA, 1994.

[71] R.L. Fox und E.L. Stanton, *Developments in structural analysis by direct energy minimization*, AIAA Journal **6** (1968), Nr. 6, 1036–1042.

[72] I. Fried, *A gradient computational procedure for the solution of large problems arising from the finite element discretization method*, International Journal for Numerical Methods in Engineering **2** (1970), 477–494.

[73] J.P. Gago, D.W. Kelly, O.C. Zienkiewicz und I. Babuska, *A-posteriori error analysis and adaptive processes in the finite element method. part II: Adaptive processes*, International Journal for Numerical Methods in Engineering **19** (1983), 1621–1656.

[74] F. Gao und B. Parlett, *Communication cost of sparse Cholesky factorization on a hypercube*, Tech. Report PAM-436, Center for Pure and Applied Mathematics, University of California, Berkeley, CA, 1988.

[75] Tom Gaskins, *PHIGS Programming Manual*, O'Reilly & Associates, Inc., Sebastopol, CA, 1992.

[76] G.A. Geist, M.T. Heath, B.W. Peyton und P.H. Worley, *A user's guide to PICL: a portable instrumented communication library.*, Tech. Report TM-11616, Oak Ridge National Laboratory, Oktober 1990.

[77] A. George, M.T. Heath, J.W.-H. Liu und E.G.-Y. Ng, *Sparse Cholesky factorization on a local memory multiprocessor*, SIAM Journal on Scientific and Statistical Computation **9** (1988), 327–340.

[78] ———, *Solution of sparse posititive definite systems on a hypercube.*, J. Comp. Appl. Math. **27** (1989), 129–156.

[79] A. George und J.W.-H. Liu, *An automatic nested dissection algorithm for irregular finite element problems*, SIAM J. Numer. Anal. **15** (1978), 1053–1069.

[80] ———, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice Hall, Englewood Cliffs, NJ, 1981.

[81] J.R. Gilbert und R. Schreiber, *Highly parallel sparse Cholesky factorization*, SIAM Journal on Scientific and Statistical Computation **13** (1992), 1151–1172.

[82] J.R. Gilbert und E. Zmijewski, *A parallel graph partitioning algorithm for a message-passing multiprocessor.*, Tech. Report 87-803, Cornell University, Ithaca, NY, 1987.

[83] D. Goldberg, *Sizing populations for serial and parallel genetic algorithms.*, Proc. of the Third Int. Conf. on Genetic Algorithms (J.D. Schaffer, Hrsg.), Morgan Kaufmann, San Francisco, CA, 1989, S. 70–79.

[84] J.J. Grefenstette, *Parallel adaptive algorithms for function optimization.*, Tech. Report CS-81-19, Vanderbilt University, Nashville, TE, 1981.

[85] W. Gropp und E. Lusk, *An abstract device definition to support the implentation of a high-level point-to-point message-passing interface*, Tech. Report MCS-P342-119, Mathematics and Computer Science Division, Argonne Natl. Laboratory, lusk@mcs.anl.gov, Oktober 1994.

[86] K. D. Gunther, *The routing problem in transputer-based parallel systems*, IEEE Transactions on Computers **C-29** (1989), 512–524.

[87] A. Gupta, G. Karypis und V. Kumar, *Highly scalable parallel algorithms for sparse matrix factorization.*, Tech. Report 94-63, Dept. Computer Science, University of Minnesota, Minneapolis, MN, Dezember 1994.

[88] A. Gupta und V. Kumar, *Parallel algorithms for forward elimination and backward substitution in direct solution of sparse linear systems.*, Tech. Report, Dept. Computer Science, University of Minnesota, Minneapolis, MN, Juni 1995.

[89] J.L. Gustafson, *Re-evaluating Amdahl's law*, Comm of the ACM **31** (1988), 532–533.

[90] W. Hackbusch, *Iterative solution of large systems of equations*, Springer Verlag, Berlin,, 1994.

[91] W. Haendler, *The impact of classification schemes on computer architectures.*, Proc. of the 1977 Int. Conf. on Parallel Processing, IEEE, 1977, S. 7–15.

[92] M.T. Heath, E. Ng und B.W. Peyton, Parallel algorithms for sparse linear systems, Parallel Algorithms for Matrix Computation, 83–124, Parallel Algorithms for Matrix Computation, SIAM Press, 1990, S. 83–124.

[93] A. Heirich, *Scalable load balancing by diffusion*, Diss., California Institute of Technology, Oktober 1994.

[94] B. Hendrickson, *Can static load balancing algorithms be appropriate in a dynamic setting?*, Workshop Dynamic load balancing on MPP systems, Daresbury Labs, Warrington, November 1995.

[95] B. Hendrickson und R. Leland, *Multidimensional spectral load balancing.*, Tech. Report TR SAND 93-0074, Sandia Natl. Lab., Albuquerque, NM, Juni 1993.



[96] ———, *A multilevel algorithm for partitioning graphs*, Tech. Report SAND 93-1301, Sandia Natl. Lab., Albuquerque, NM, Juni 1993.

[97] ———, *The Chaco Users guide - Version 1.0*, Tech. Report SAND93-2339, Sandia Natl. Lab., Albuquerque, NM, Oktober 1993.

[98] M.R. Hestenes, *Conjugate direction methods in optimization*, Springer-Verlag, New York, 1980.

[99] M.R. Hestenes und E. Stiefel, *Methods of conjugate gradients for solving linear systems*, Journal of Research of the National Bureau of Standards **49** (1952), 409–436.

[100] C.A.R. Hoare, *Communicating sequential processes*, Prentice Hall International, 1985.

[101] J. Holland, *Outline for a logical theory of adaptive systems*, J. ACM **3** (1962), 297–314.

[102] ———, *Adaptation in natural and artificial systems*, AnnArbor, The University of Michigan Press, 1975.

[103] G. Horton, *A multi-level diffusion method for dynamic load balancing*, Parallel Computing **19** (1993), 209–218.

[104] S.-H. Hsieh, G.H. Paulino und J.F. Abel, *Recursive spectral algorithms for automatic domain paritioning in parallel finite element analysis*, Computer Methods in Applied Mechanics and Engineering **121** (1995), 137–162.

[105] T.J.R. Hughes und R.M. Ferencz, *Fully vectorized ebe preconditioners for nonlinear solid mechanics: Applications to large-scale three-dimensional continuum, shell and contact/impact problems*, Domain Decomposition Methods for Partial Differential Equations (R. Glowinski, G.H. Golub, G.A. Meurant, und J. Periaux, Hrsg.), SIAM Philadelphia, 1988, S. 261–280.

[106] T.J.R. Hughes, I. Levit und J.M. Winget, *Implicit stable algorithms for heat conduction analysis*, J. Engn. Mech. Div., ASCE **109** (1983), 576–585.

[107] S. Jensen, *Adaptive dimensional reduction numerical solution of monotone quasilinear boundary value problems*, SIAM J. Numerical Analysis **29** (1992), 1294–1320.

[108] M. Jerrum und G. Sorkin, *Simulated annealing for graph bisection*, Tech. Report, Dept. Computer Science, University of Edinburgh, April 1993.

[109] C.B. Jiang, M. Kawahara, K. Hatanaka und K. Kashiyama, *A three-step finite element method for convection dominated incompressible flow*, Comp. Fluid Dyn. J. **1** (1993), 443–446.

[110] Z. Johan, K.K. Mathur und S.L. Johnsson, *An efficient communication strategy for finite element methods on the connection machine CM-5 system*, Tech. Report TR 256, Thinking Machines, 245 First Street, Cambridge MA 02142, 1993.

[111] O.L. Johnson, A.C. Micchelli und G. Paul, *Polynomial preconditioners for conjugate gradient calculations.*, SIAM J. Num. Analysis **20** (1983), Nr. 2, 362–376.

[112] M.T. Jones und P.E. Plassmann, *Computational results for parallel unstructured mesh computations*, Computing Systems in Engineering **5** (1994), Nr. 4-6, 297–309.

[113] ———, *Parallel algorithms for the adaptive refinement and partitioning of unstructured meshes*, Scalable High-Performance Computing Conf. 1994 (Argonne National Laboratory), IEEE Computer Society Press, Mai 1994, S. 478–485.

[114] José Encarnação and Wolfgang Straßer, *Computer graphics*, Akademie Verlag Berlin, 1988.

[115] D.K. Kahaner, *Parallel processing activities in japan*, comp.research.japan, Oktober 1994.

[116] G. Karypis und V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, Tech. Report 95-035, Dept. Computer Science, University of Minnesota, Minneapolis, MN, 1995.

[117] ———, *METIS - Unstructured graph partitioning and sparse matrix ordering system. User guide*, Dept. Computer Science, University of Minnesota, Minneapolis, MN, Juni 1995.

[118] K. Kashiyama, H. Ito, M. Behr und T.E. Tezduyar, *Three-step explicit finite element computation of shallow water flow on a massively parallel computer*, International Journal for Numerical Methods in Fluids **21** (1995), Nr. 10, 885–900.

[119] M. Kawahara, H. Hirano und K. Tsubota, *Selective lumping finite element method for shallow water flow*, International Journal for Numerical Methods in Fluids **2** (1982), 89–112.

[120] M. Kawahara und T. Umetsu, *Finite element method for moving boundary problems in river flow*, International Journal for Numerical Methods in Fluids **6** (1986), 365–386.

[121] D.W. Kelly, J.P. Gago, O.C. Zienkiewicz und I. Babuska, *A posteriori error analysis and adaptive processes in the finite element method. part I: Error analysis*, International Journal for Numerical Methods in Engineering **19** (1983), 1593–1619.

[122] B. Kernighan und S. Lin, *An efficient heuristic procedure for partitioning graphs.*, Bell System Technical Journal **29** (1970), 291–307.

[123] A.I. Khan und B.H.V. Topping, *Subdomain generation for parallel finite element analysis*, Computing systems in Engineering **4** (1993), Nr. 4-6, 473–488.

[124] ———, *A transputer routing algorithm for non-linear or dynamic finite element analysis*, Engineering Computations **11** (1994), 549–564.

[125] S. Kirkpatrick, C.D. Gelatt und M.P. Vecchi, *Optimization by simulated annealing.*, Science **220** (1983), 671–680.

[126] A. Knowles und T. Kantchev, *Message passing in a transputer system*, Microprocessors and Microsystems **13** (1989), Nr. 2.

[127] V. Kumar, A.Y. Grama, A. Gupta und G. Karypis, *Introduction to parallel computing: Design and analysis of algorithms*, Benjamin-Cummings, Redwood City, CA, 1994.



[128] V. Kumar und A. Gupta, *Analyzing scalability of parallel algorithms and architectures*, Tech. Report 91-18, Dept. Computer Science, University of Minnesota, Minneapolis, MN, 1991.

[129] L. Lämmer und U. Meißner, *Improved parallel approach to moving boundary problems in river flow*, Proc. of the Second Intl. Conf. on Hydroscience and Engineering (S. Wang, Hrsg.), Vol. II, Tsinghua University Press, Beijing, 1995, S. 1799–1806.

[130] G.v. Laszewski und H. Mühlenbein, *Partitioning a graph with a parallel genetic algorithm.*, Parallel problem solving from nature (H.-P. Schwefel und R. Männer, Hrsg.), Lecture Notes in Computer Sciences, Nr. 469, Springer Berlin, Heidelberg u.a., 1991, S. 165–169.

[131] K. Law und D.R. Mackay, *A parallel row-orientied sparse solution method for finite element structural analysis*, International Journal for Numerical Methods in Engineering **36** (1993), 2895–2919.

[132] K.H. Law, *A parallel finite element solution method*, Computers & Structures **23** (1986), Nr. 6, 845–858.

[133] D.H. Lawrie, *Access and alignment of data in an array processor*, IEEE Transactions on Computers **24** (1975), Nr. 12, 1145–1155.

[134] C. Leiserson und J. Lewis, *Orderings for parallel sparse symmetric factorization*, Tech. Report ETA-TR-85, Boeing Computer Services, Marz 1988.

[135] C.E. Leiserson, *Fat-trees: Universal networks for hardware efficient supercomputing*, 1985 International Conference on Parallel Processing, 1985, S. 393–402.

[136] R. Leland und B. Hendrickson, *An empirical study of static load balancing algorithms*, Scalable High-Performance Computing Conf. 1994, IEEE Computer Society Press, Mai 1994, S. 682–685.

[137] T. Lengauer, *Combinatorical algorithms for integrated circuit layout*, Teubner-Verlag, Stuttgart, 1990.

[138] J.W.-H. Liu, *The minimum degree ordering with constraints*, SIAM J. Sci. Stat. Comput. **10** (1989), 1136–1145.

[139] R.F. Lucas, *Solving planar systems of equations on distributed-memory multiprocessors*, Diss., Dept. of Electrical Engineering, Stanford University, Palo Alto, CA, 1987.

[140] R.F. Lucas, T. Blank und J.J. Tiemann, *A parallel solution method for large sparse systems of equations*, IEEE Transactions on Computer Aided Design **CAD-6** (1987), Nr. 6, 981–991.

[141] R. Lusk, *An easy introduction to MPI*, 1994.

[142] T.A. Manteuffel, *An incomplete factorization technique for positive definite linear systems*, Mathematics of Computation **34** (1980), 473–497.

[143] H.M. Markowitz, *The elimination form of the inverse and its application to linear programming*, Management Science **3** (1957), 255–269.

[144] J.A. Meijerink und H.A. van der Vorst, *An iterative solution method for linear systems of which the coefficient matrix is a symmetric m-matrix*, Mathematics of Computation **31** (1977), 148–162.

[145] U. Meißner und H. Wibbeler, *A least square principle for the a posteriori computation of finite element approximation errors*, Computer Methods in Applied Mechanics and Engineering **85** (1991), 89–108.

[146] P. M. Merlin und P. J. Schweitzer, *Deadlock avoidance in store-and-forward networks*, IEEE Transactions on Computers **28** (1980), Nr. 3, 345–360.

[147] N. Metropolis, A.W. Rosenbluth und M.N. Rosenbluth et.al., *Equation of state calculations by fast computing machines*, J. Chem. Physics **21** (1953), Nr. 6, 1087–1092.

[148] A. Meyer, *A parallel preconditioned conjugate gradient method using domain decomposition and inexact solvers on each subdomain*, Computing **45** (1990), 217–234.

[149] M. Meyer, *Grafik-Ausgabe vom Parallelrechner für 3D-Gebiete*, Tech. Report, Fakultät für Mathematik, Technische Universität Chemnitz-Zwickau, Mai 1995.

[150] J.G. Mills, L.J. Clarke und A.S. Trew, *CHIMP concepts*, Tech. Report EPCC-KTP-CHIMP-CONC, Edinburgh Parallel Computing Centre, University of Edinburgh, April 1991.

[151] B. Mohar, The Laplacian spectrum of graphs, 871–898, J. Wiley, New York, 1991, S. 871–898.

[152] B. Monien, R. Diekmann und R. Lueling, *Communication throughput of interconnection networks*, Proc. of MFCS '94, Springer LNCS, 1994.

[153] D. Moore, *A round-robin parallel partitioning algorithm*, Tech. Report TR 88-916, Cornell University, Ithaca, NY, 1988.

[154] M. Mu und J. Rice, *A grid based subtree-subcube assignment strategy for solving partial differential equations on hypercubes*, SIAM Journal on Scientific and Statistical Computation **13** (1992), 826–839.

[155] nCUBE Corporation, *nCUBE2 programmers guide, r2.0*, Dezember 1990.

[156] E. Ng, *Supernodal symbolic Cholesky factorization on a local-memory multiprocessor*, Parallel Computing **19** (1993), 153–162.

[157] B. Nour-Omid, *Solving large linearized systems in mechanics*, Solving large-scale problems in mechanics (M. Papadrakakis, Hrsg.), J. Wiley, Chichester, 1993, S. 39–64.

[158] B. Nour-Omid und B.N. Parlett, *Element preconditioning using splitting techniques*, SIAM J. Sci. Stat. Computing **6** (1985), 761–770.

[159] B. Nour-Omid, A. Raefsky und G. Lyzenga, *Solving finite element equations on concurrent computers*, Parallel computations and their impact on mechanics (A.K. Noor, Hrsg.), Vol. 86, ASME, New York, 1987, S. 209–228.

[160] J. Olden, *Finite-Element-Analyse von Plattentragwerken durch adaptive Software-Techniken*, Diss., Darmstadt, Techn. Hochsch., 1996.

[161] A.M. Ostrowski, *Über die Determinanten mit überwiegender Hauptdiagonale*, In Commentarii Mathematici Helvetici [90], 69–96.



[162] M. Papadrakakis, *Solving large-scale linear problems in solid and structural mechanics*, Solving large-scale problems in mechanics (M. Papadrakakis, Hrsg.), J. Wiley, Chichester, 1993, S. 1–37.

[163] Parasoft corporation, *Express version 1.0: A communication environment for parallel computers*, 1988.

[164] K. Parviz und L. Kienrock, *Virtual cut-through: a new computer communication switching technique*, Computer Networks **3** (1979), 267–268.

[165] M. Pester, *Grafik-Ausgabe vom Parallelrechner für 2D-Gebiete*, Tech. Report Preprint SPC95_24, Fakultät für Mathematik, Technische Universität Chemnitz-Zwickau, November 1994.

[166] P. Pierce, *The NX/2 operating system*, Proceedings of the Third Conf. on Hypercube Concurrent Computers and Applications., ACM Press, 1988, S. 384–390.

[167] A. Pollmann, *HELIOS, Anwendungs-, Funktionalitäts- und Leistungsaspekte*, BI-Wissenschaftsverlag, Mannheim, Wien und Zürich, 1991.

[168] A. Pothen, H.D. Simon und K.-P. Liou, *Partitioning sparse matrice with Eigenvectors of graphs*, SIAM J. Matrix Appl. **11** (1990), Nr. 3, 430–452.

[169] A. Pothen und C. Sun, *Distributed multifrontal factorization using clique trees*, Proc. 5th SIAM Conf. on Parallel Processing for Scientific Computing, 1991, S. 34–40.

[170] D. Powers, *Graph partitioning by eigenvectors*, Lin. Alg. Appl. **101** (1988), 121–133.

[171] R. Pozo und S.L. Smith, *Performance evaluation of the parallel multifrontal method in a distributed-memory environment*, Proc. 6th SIAM Conf. on Parallel Processing for Scientific Computing, 1993, S. 453–456.

[172] F.P. Preparata und J. Vuillemin, *The cube-connected cycles: A versatile network for parallel computation*, Communication of the ACM **24** (1981), Nr. 5, 300–309.

[173] J.K. Reid, *On the method of conjugate gradients for the solution of large sparse systems of linear equations*, Large sparse sets of linear equations (J.K. Reid, Hrsg.), Academic Press, 1971, S. 231–254.

[174] M. Roettger, U.-P. Schroeder und J. Simon, *Virtual topology library for PARIX*, Tech. Report 005-93, Paderborn Center for Parallel Computing, Uni-GH Paderborn, November 1993.

[175] D.J. Rose, A graph-theoretic study of the solution of sparse positive definite systems of linear equations, Graph theory and Computing, Graph theory and Computing, Academic Press, New York, 1972.

[176] Ralf Rosenberg, *EURO Telebits Journal*, Tech. Report, Projektträger des BMFT für Informationstechnik bei der Deutschen Forschungsanstalt für Luft- und Raumfahrt e.V. Abteilung EG-Forschungsprogramme, Oktober 1993.

[177] J.S. Rowlan und B.T. Wightman, *Portal: A communication library for run-time visualization of distributed, asynchronous data*, Scalable High-Performance Computing Conference, Knoxville, TN, Mai 1994, S. 350–356.

[178] C. Rubbia, *Report of the EEC working group on high-performance computing*, Tech. Report, Europäische Gemeinschaft, Brüssel, 1991.

[179] M. Rumpf und M. Geiben, Visualization of finite elements and tools for numerical analysis, Advances in Scientific Visualization, Advances in Scientific Visualization, Springer, New York, 1993.

[180] M. Rumpf und A. Schmidt, *GRAPE graphics programming environment*, Tech. Report 8, SFB 256, Universität Bonn, 1990.

[181] M. Rumpf, A. Schmidt und K.G. Siebert, *Functions defining arbitrary meshes - a flexible interface between numerical data and visualization routines*, Tech. Report, Universität Freiburg, Februar 1995.

[182] F.C. Sang und I.H. Sudborough, *Embedding large meshes into small ones*, Tech. Report, Department of Computer Science, University of Texas at Dallas, Richardson, TX 75083, 1990.

[183] J.E. Savage und M.G. Wloka, *Parallelism in graph partitioning*, Computing Systems in Engineering **1** (1991), 135–148.

[184] R. Schreiber, *Scalibility of sparse direct solvers*, Graph theory and sparse matrix computation (A. George, J.R. Gilbert und J.W.-H. Liu, Hrsg.), Springer, 1993, S. 191–209.

[185] H.R. Schwarz, *Methode der finiten Elemente*, Teubner-Verlag, Stuttgart, 1984.

[186] U. Schwarz, *EXTENDER - Ein generisches Werkzeug zur netzverteilten Simulation und Visualisierung*, http://www.zib-berlin.de/German/VisPar/vis/BRTB/EXTENDER/XTD.html, Juli 1995.

[187] SGI (Hrsg.), *OpenGL Programming Guide*, Addison-Wesley, 1994.

[188] J.R. Shewchuk, *An introduction to the conjugate gradient method without the agonizing pain*, Tech. Report CMU-CS-94-125, School of Computer Science, Carnegie Mellon Univ., Pittsburgh, PA 15213, Marz 1994.

[189] J.E. Shore, *Second thoughts on parallel processing.*, Computers and Electrical Engineering **1** (1973), Nr. 1, 95–109.

[190] M. Simmen, *Comments on broadcast algorithms for two-dimensional grids*, Parallel Computing **17** (1991), 109–112.

[191] H.D. Simon, *Partitioning of unstructured problems for parallel processing.*, Computing Systems in Engineering **2** (1991), Nr. 2/3, 135–148.

[192] A. Skjellum, S.G. Smith, A.P. Leung und M. Morari, *The design and evolution of Zipcode*, Tech. Report, LLNL, 1992.

[193] B. Smith, P. Brstad und W. Gropp, *Domain decomposition*, Cambridge University Press, Cambridge, 1996.

[194] Perihelion Software (Hrsg.), *The Helios operating system*, Prentice Hall, 1989.

[195] N.T. Son und Y. Parker, *Adaptive deadlock-free packet routeing in transputer-based multiprocessor interconnection networks*, Computer Journal **34** (1991), Nr. 6, 493.

[196] E. Stein, W. Rust und S. Ohnimus, *h- and d-adaptive fe element methods for two dimensional structural problems including post-buckling of shells*, Computer Methods in Applied Mechanics and Engineering **101** (1992), 315–354.



[197] P. Suaris und G. Kedem, *An algorithm for quadrisection and its application to standard cell placement*, IEEE Transactions on Circuits and Systems **35** (1988), 294–303.

[198] M. Suarjana, *Conjugate gradient method for linear and nonlinear structural analysis on sequential and parallel computers*, Diss., Stanford University, Dept. Civil Engineering, 1994.

[199] C. Sun, *Efficient parallel solutions of large SPD systems on distributed-memory multiprocessors*, Tech. Report CTC-92-102, Advanced Computing Research Institute, Center for Theory and Simulation in Science and Engineering, Cornell University, Ithaca, NY, August 1992.

[200] Thinking Machines Corporation, *Connection machine cm-5 technical summary*, 1993.

[201] C. Thornborrow und C. Faigle, *Parallel modules in modular visualisation environments.*, Tech. Report, EPCC, Edinburgh, Scotland, 1993.

[202] M. Timenetsky, *A new preconditioning technique for solving large sparse linear systems*, Linear Algebra and its Applications **154-156** (1991), 331–353.

[203] W.F. Tinney, *Comments on using sparsity techniques for power system problems*, Tech. Report RAI 3-12-69, IBM Research, Marz 1969.

[204] S. M. Trewin, *Mesh decomposition and distribution concepts*, Tech. Report Draft, EPCC, April 1994.

[205] S.M. Trewin, *PUL-MD prototyp user guide*, EPCC, 0.2 Ausg., September 1994.

[206] ———, *PUL-SM prototype user guide*, EPCC, Februar 1995.

[207] T. Umetsu, *Applications of moving boundary simulation for river flow due to configuration bank environment*, 8th Int. Conference on Finite Elements in Fluids on New Trends and Applications, 1993.

[208] ———, *A boundary condition technique of moving boundary simulation for broken da problem by tree-step explicit finite element method.*, Proc. of the Second Intl. Conf. on Hydroscience and Engineering (S. Wang, Hrsg.), Vol. II, Tsinghua University Press, Beijing, 1995, S. 394–399.

[209] T. Umetsu, L. Lämmer und U. Meißner, *Two-step explicit finite element method for shallow water flow using a scalable parallel approach*, X. Intl. Conf on Computational Methods in Water Resources, 1994, S. 1541–1548.

[210] R.A. van de Geijn, *On global combine operations*, Tech. Report, Dept. of Computer Science, Univ. of Texas at Austin, Austin, Texas 78712, Oktober 1992.

[211] E.A. Varvarigos und D.P. Bertsekas, *Communication algorithms for isotropic tasks in hypercubes and wraparound meshes*, Parallel Computing **18** (1992), 1233–1257.

[212] A. Vaziri, M. Kremenetsky, M. Fitzgibbon und C. Levit, *Experiences with CM/AVS to visualize and compute simulation data on the CM-5*, Tech. Report, NASA Ames Research Center, Moffett Field, CA, Marz 1994.

[213] C. Walshaw und M. Berzins, *Dynamic load-balancing for PDE solvers on adaptive unstructured meshes*, CPE **7** (1995), Nr. 1, 17–28.

[214] C. Walshaw, M. Cross und M.G. Everett, *A localized algorithm for optimising unstructured mesh partitions*, Int. J. Supercomputer Appl. **9** (1996), Nr. 4, 280–295, to appear.

[215] Waltham, *AVS developer guide*, Advanced Visual Systems Inc., 1992.

[216] ———, *AVS user guide*, Advanced Visual Systems Inc., 1992.

[217] G. Wiegand und R. Covey, *HOOPS Reference Manual, Vers. 3.0*, Ithaca Software, Almeida, CA, 1994.

[218] D.R. Williams, *Performance of dynamic load balancing algorithms for unstructured mesh calculations.*, Concurrency: Practice and Experience **3** (1991), Nr. 5, 457–491.

[219] S. Williams, M.L. Sawley und D. Cobut, *On-line visualization for scientific applications on the T3D*, Tech. Report, EPFL, Januar 1996.

[220] G. Wittum und F. Liebau, *On truncated incomplete decompositions*, BIT **29** (1989), 719–740.

[221] C.-L. Wu und T.-Y. Feng, *On a class of multistage interconnection networks*, IEEE Transactions on Computers **29** (1980), Nr. 8, 694–702.

[222] J. Wu, J. Saltz, H. Berryman und S. Hiranandani, *Distributed memory compiler design for sparse problems*, Tech. Report 91-13, ICASE, 1991.

[223] O.C. Zienkiewicz und J.Z. Zhu, *The superconvergent patch recovery and a-posteriori error estimates. part 1: The recovery technique*, International Journal for Numerical Methods in Engineering **33** (1992), 1331–1364.

[224] ———, *The superconvergent patch recovery and a-posteriori error estimates. part 1: Error estimates and adaptivity*, International Journal for Numerical Methods in Engineering **33** (1992), 1365–1382.

[225] E. Zmijewski, *Limiting communication in parallel sparse Cholesky factorization*, Tech. Report 89-18, University of California, Dept. of Computer Science, Santa Barbara, CA, 1989.

[226] M. Zubair und M. Ghose, *A performance study of sparse Cholesky factorization on Intel iPSC/860*, Tech. Report 92-13, ICASE NASA Langley Research Centre, Hampton, VA 23665-5225, Marz 1992.