

Parallelisierung numerischer Algorithmen
für partielle Differentialgleichungen¹

Gundolf Haase

¹Teubner-Verlag, 1999, ISBN 3-519-02970-7

Meinen Eltern und meiner Tochter Rebecca gewidmet.

Vorwort

Das vorliegende Buch wendet sich an Ingenieure und Naturwissenschaftler welche sich mit der Lösung von partiellen Differentialgleichungen beschäftigen, sei es als Entwickler neuer Algorithmen oder als Anwender erprobter Techniken. Zum Verständnis des Buches sind nur geringe Vorkenntnisse bzgl. der behandelten Thematik vonnöten. Der gesamte Inhalt des Buches stammt aus Vorlesungen, welche der Autor an den Universitäten zu Linz (Österreich), Freiberg, Freiburg, Stuttgart (BRD) und Győr (Ungarn) hielt.

Sehr viele Problemstellungen aus den verschiedensten Ingenieurbereichen, aus der Physik und neuerdings auch aus der Finanzmathematik lassen sich mittels partieller Differentialgleichungen, bzw. Systemen von partiellen Differentialgleichungen beschreiben. Diese komplizierten Gleichungen können bei praktischen Problemstellungen nur noch in den seltensten Fällen analytisch, d.h. mit Papier und Bleistift bzw. Formelmanipulationsprogrammen, gelöst werden. Daher überführt man diese Differentialgleichungen durch geeignete Diskretisierungsmethoden (finite Elemente, finite Differenzen, finite Volumen) in ein Gleichungssystem oder eine Folge von Gleichungssystemen, mit reellen Zahlen als Lösung. Die Lösung dieser großdimensionierten Gleichungssysteme erfolgt mit dem Computer.

Dank der Entwicklung moderner, auf Mehrgitter- bzw. Multileveltechniken basierender, iterativer Löser für diese Gleichungssysteme und der rasanten Hardwareentwicklung kann schon ein einzelner Computer komplexe 3D-Probleme lösen. Jedoch wird es immer Problemstellungen geben, welche noch mehr Rechenleistung erfordern. An diesem Punkt setzt das vorliegende Buch an und schlägt parallele Lösungsstrategien für diese Problemklassen vor.

Die kurze Einführung in Kapitel 1 wird in Kapitel 2 durch einen Überblick zu Prinzipien von Parallelrechnern ergänzt. Anschließend werden in Kapitel 3 einige einfache Grundroutinen zum Datenaustausch zwischen zwei oder mehreren Rechenknoten eingeführt und einige Gedanken zum Vergleich von parallelen Algorithmen geäußert. Im anschließenden Kapitel 4 werden einige typische Vorgehensweisen bei der Parallelisierung direkter Lösungsverfahren für Gleichungssysteme betrachtet. Dies dient vorrangig dem Zweck, die völlig anders-

artige Herangehensweise bei der Parallelisierung iterativer Lösungsverfahren in den nachfolgenden Kapiteln hervorzuheben.

Die parallele Lösung von partiellen Differentialgleichungen muß schon bei der Diskretisierung ansetzen. In Kapitel 5 wird die enge Verbindung von in iterativen Verfahren vorkommenden algorithmischen Grundroutinen und der Wahl der Datenaufteilung an Hand einer Diskretisierung mit finiten Elementen analysiert. Es zeigt sich, daß es für den vorgeschlagenen Lösungsweg keinen prinzipiellen Unterschied zwischen einer überlappenden und einer nichtüberlappenden Elementaufteilung gibt. Darauf aufbauend werden in Kapitel 6 die wichtigsten Iterationsverfahren nach einer einheitlichen Strategie parallelisiert, diese Verfahren treten wiederum in Kapitel 7 als Komponenten in der Lösung von Erhaltungsgleichungen auf. Hier zeigt sich, daß das in Kapitel 5 für finite Elemente entwickelte Datenkonzept auch auf finite Volumen übertragbar ist.

Üblicherweise wird die dem Buch zugrunde liegende Vorlesung von einem Praktikum begleitet, welches gemeinsam mit Dr. Michael Kuhn entwickelt wurde. Einige Ausführungen dazu sind im Anhang A zu finden.

Ich möchte an dieser Stelle Dr. Matthias Pester von der TU Chemnitz für die freundliche Überlassung seines Vorlesungsskriptes zur Parallelisierung danken, welches in die Kapitel 1-3 einfloß. Gleichzeitig gingen eigene Vorlesungsmitschriften aus der entsprechenden Vorlesung an der TU Karl-Marx-Stadt bei Prof. Volkmar Friedrich in Kapitel 4 ein. Gleichfalls möchte ich mich bei meinen Linzer Kollegen für die generelle Unterstützung bedanken.

Da sich die Welt des Parallelrechnens, insbesondere die Hardware, rasant entwickelt, werden unter

<http://www.numa.uni-linz.ac.at/Staff/ghaase/parallel.html>
aktuellere Daten zu einigen Punkten des Buches zu finden sein.

Juli 1999, Linz, Österreich

Inhalt

1	Einführung	9
1.1	Wozu braucht man Parallelrechner ?	9
2	Parallel- und Vektorrechner	13
2.1	Klassifikationen	13
2.1.1	Ebenen der Parallelität	13
2.1.2	Hardwareklassifikation	14
2.1.3	Grobklassifikation nach Flynn	14
2.1.4	Klassifikation nach Speicherzugriff	18
2.2	Topologien	20
2.2.1	Definitionen	20
2.2.2	Farm / Master-Slave Topologie	21
2.2.3	Die Pipe	21
2.2.4	Der Ring	22
2.2.5	Array und Torus	22
2.2.6	Der Baum (Tree)	23
2.2.7	Der Hypercube	24
2.2.8	Das DeBrujin-Netzwerk	27
2.2.9	Freie Topologie	28
2.3	Parallele Betriebssystemerweiterungen und Programmiersprachen	29
3	Algorithmische Ebene	31
3.1	Einige Begriffe	31
3.2	Synchronisation paralleler Prozesse	35
3.2.1	Einige Begriffe	35
3.2.2	Semaphoren	36
3.2.3	Datenkohärenz	42
3.3	Grundlegende globale Operationen	43
3.3.1	EXCHANGE	43

3.3.2	Gather-Scatter-Operationen	44
3.3.3	Broadcast	45
3.3.4	Reduce- und All-Reduce Operationen	46
3.3.5	Barrier	48
3.3.6	Bemerkungen zu portablem Code	49
3.4	Leistungsbewertung paralleler Algorithmen	49
3.4.1	Speedup und Scaleup	49
3.4.2	Effizienz	53
3.4.3	Kommunikationsaufwand	54
4	Direkte Verfahren	56
4.1	Die BLAS-Bibliotheken	56
4.1.1	Vektor-Vektor Operationen (BLAS1)	56
4.1.2	Matrix-Vektor Operationen (BLAS2)	59
4.1.3	Matrix-Matrix-Operationen (BLAS3)	62
4.2	Elimination durch Drehungsmatrizen	65
4.2.1	Die Givensrotation	66
4.2.2	Givensrotation auf dem Vektorrechner	67
4.2.3	Givensrotation auf dem Parallelrechner	67
4.3	Die LU-Zerlegung	69
4.3.1	Der serielle Standardalgorithmus	69
4.3.2	Vektorisierung der LU-Zerlegung	69
4.3.3	Parallelisierung der LU-Zerlegung	71
4.4	Gaußelimination für tridiagonale Matrizen	73
4.4.1	Parallelisierung mittels zyklischer Reduktion	73
4.5	FFT	75
4.5.1	Die 1D-Fourieranalyse und Synthese	76
4.5.2	Die FFT-Idee und ihre Parallelisierung	77
5	Gebietszerlegungen und numerische Grundroutinen	80
5.1	Nichtüberlappende Elemente	81
5.1.1	Generierung der Steifigkeitsmatrix	84
5.1.2	Umwandlung der Vektortypen	84
5.1.3	Skalarprodukt	84
5.1.4	Matrix-mal-Vektor Multiplikation	85
5.2	Allgemeiner Ansatz für akkumulierte Matrizen	86
5.2.1	Knoten- und Teilgebietsmengen	86
5.2.2	Matrix-Vektor Produkt	88
5.2.3	Anwendungen der Sätze 5.1. und 5.2.	91

Inhalt		7
5.3	Überlappende Elemente	94
6	Iterative Verfahren	99
6.1	Das CG-Verfahren	99
6.1.1	Das serielle Verfahren	99
6.1.2	Der parallelisierte CG	100
6.2	Das GMRES-Verfahren	102
6.2.1	Das serielle Verfahren	102
6.2.2	Der parallele GMRES	103
6.3	Das ω -Jacobi Verfahren	105
6.3.1	Das serielle Verfahren	105
6.3.2	Datengraph der Jacobi-Iteration	105
6.3.3	Das parallele Verfahren	106
6.4	Das Gauß-Seidel Verfahren	107
6.4.1	Das serielle Verfahren	107
6.4.2	Die Red-Black-Gauß-Seidel-Iteration	108
6.4.3	Das parallele Verfahren	109
6.5	Unvollständige Zerlegungen	115
6.5.1	Der serielle Algorithmus	115
6.5.2	Die parallele ILU-Faktorisierung	117
6.5.3	Die parallele IUL-Faktorisierung	118
6.5.4	Vergleich von ILU- und IUL-Faktorisierung	120
6.5.5	Die IUL-Faktorisierung in 3D	120
6.5.6	Die IUL-Faktorisierung mit reduziertem Besetztheitsmuster	120
6.6	Der Schurkomplement CG	122
6.6.1	Das Schurkomplement	122
6.6.2	Der parallele Schurkomplement-CG	124
6.7	Die Multigridmethode	124
6.7.1	Der serielle Algorithmus	125
6.7.2	Die parallelen Komponenten	126
6.7.3	Der parallele Algorithmus	129
6.8	Vorkonditionierung Iterativer Verfahren	130
6.8.1	Akkumulierte Vorkonditionierer	130
6.8.2	Verteilte Vorkonditionierer	130
7	Erhaltungsgleichungen	131
7.1	Die inkompressiblen Navier-Stokes-Gleichungen	131
7.1.1	Die Differentialgleichungen	131
7.1.2	Die serielle Auflösung	133

7.1.3	Komponenten der Parallelisierung	138
7.2	Die Eulergleichungen	141
7.2.1	Die Differentialgleichungen	141
7.2.2	Die serielle Auflösung	142
7.2.3	Die Parallelisierung mittels verteilter Boxen	145
7.2.4	Die Parallelisierung mittels aufgeteilter Boxen	147
7.3	Die kompressiblen Navier-Stokes-Gleichungen	148
7.3.1	Die Differentialgleichungen	148
7.3.2	Die serielle Auflösung	149
7.3.3	Die Parallelisierung mittels aufgeteilter Boxen und verteilter Elemente	153
A	Begleitendes Praktikum	155
A.1	Durchführung des Praktikums	155
A.2	Programme für das Praktikum	155
A.3	Die Praktikumsaufgaben	156
A.3.1	Vorbereitungen	156
A.3.2	Das erste parallele Programm	158
A.3.3	Blockierende Kommunikation	159
A.3.4	Globale Operationen	159
A.3.5	Lokaler Datenaustausch	160
A.3.6	Iterative Löser	162
B	Einige Internetadressen	164
B.1	MPI: Message Passing Interface	164
B.2	PVM: Parallel Virtual Machine	164
B.3	Weitere Links	164
B.4	Parallelrechner	165
C	Einige Zitate	168
	Literaturverzeichnis	169

1 Einführung

1.1 Wozu braucht man Parallelrechner ?

Während der letzten Dekaden verdoppelte sich die verfügbare Rechenleistung pro Prozessor aller 18 Monate (Moore'sches Gesetz). Nimmt man den beliebten Spec-Index¹ zum Maßstab der Rechenleistung, so erkennt man ein relativ geschlossenes Feld von Spitzencomputern. Gleichzeitig rückt die Rechenleistung des Jedermann-PCs dank der neuen Prozessoren von Intel² und AMD³ in die Bereiche der klassischen Workstationhersteller IBM⁴, Sun⁵, SGI⁶, Compaq⁷ und HP⁸ vor.

Tab. 1.1 Leistungsindex von Workstations mit einem Prozessor (II/99)

Anbieter	Maschine	Prozessor	SpecInt95	SpecFp95
Compaq	AlphaServer ES 40	Alpha 21264, 500 MHz	27	58
Dell	Precision WS 610	PIII-Xeon, 450 MHz	19	15
Hewlett Packard	HP 9000 N4000	PA-RISC 8500, 440 MHz	34	51
IBM	RS/6000 43P-260	POWER-3, 200 MHz	15	30
Silicon Graphics	Origin200	MIPS R12k, 270 MHz	16	25
SUN	Enterprise 3500	UltraSparc-II, 400 MHz	18	30
Intel	Intel-Labor	P-III, 550 MHz	24	15
AMD	AMD-Labor	Athlon, 600 MHz	26	21

Die beiden letzten Zeilen in Tabelle 1.1 sind noch keine offiziellen Spec-Raten, sie wurden in den Herstellerlabors erzielt. Je nach verwendeten Programmen bzw. je nach Testausrichtung ändern sich die Relationen, jedoch bleibt die

¹<http://www.specbench.org/results.html>

²<http://www.intel.de>

³<http://www.amd.com>

⁴<http://www.ibm.de>

⁵<http://www.sun.de>

⁶<http://www.sgi.de>

⁷<http://www.compaq.de>

⁸<http://www.hewlett-packard.de>

Grundtendenz bestehen. Dies heißt nichts anderes, als daß mittlerweile jeder gut (und trotzdem preiswert!) ausgerüstete PC über mehr Rechenleistung und oft auch mehr Speicher als eine sehr teure Workstation von vor 3-5 Jahren verfügt.

Dank der Leistungen von Mikroelektronik und Technologie ($0.18 \mu\text{m}$ -Strukturen) lassen sich immer größere Integrationsdichten und Taktraten erzielen. Da die Technologieentwicklung weitergeht, braucht man also nur so lange zu warten, bis ein Computer mit genügend Rechenleistung produziert wird.

Aber : Die Signalgeschwindigkeit ist endlich ($3 \cdot 10^8 \text{ m/s}$). So waren die einzelnen Prozessoreinheiten der schnellsten Cray (1994) im Kreis angeordnet, damit Nachrichten innerhalb *eines* Taktes übermittelt werden können (Zykluszeiten $\approx 1 \cdot 10^{-9} \text{ s}$). Das technologische Problem der Reproduktion der Chipmasken bei weiterer Verkleinerung der Chipstrukturen von gegenwärtig ca. $0.18 \mu\text{m} = 1.8 \cdot 10^{-7} \text{ m}$ kann mit viel Aufwand gelöst werden. Jedoch entspricht die derzeitige Dicke von 4 nm der SO_2 -Isolationsschicht nur noch 25 Atomlagen, bei 4 Atomlagen verliert das Material seine Isolationswirkung (siehe c't 14/99, S.24). Somit scheint die Atomgröße der weiteren Leistungsentwicklung konventioneller Computer ein Ende zu setzen.

Die Entwicklung sehr schneller Rechnersysteme, d.h. *Prozessor + Speicher + I/O*, wird immer schwieriger und kostspieliger. Mit der notwendigen Komplexität der Chips steigt auch zwangsläufig die "Chance" von Produktions-/Entwurfsfehlern (Fehlerhafte Division bei Intel, Überhitzung von Prozessoren bei anderen Herstellern).

Gleichzeitig steigen die *Anforderung von Wissenschaft und Technik* schneller als die verfügbare Rechenleistung.

Wenn eine geforderte Rechenleistung gebracht wird, ändert der Anwender {Mathematiker, Physiker, Ingenieur} ein ε oder h , um mit dem aktuellen Rechner wieder unzufrieden sein zu dürfen.

⇓

Hohe Rechenleistung.

Hohe Speicherkapazität (+Datenzugriff)

Schnelle Datenbereitstellung und -auswertung (Grafik)

Einige typische Konsumenten von Rechenleistung :

Physik	→	Wiedereintritt in die Erdatmosphäre
	⇒	Bolzmann-Gleichung mit 7 Freiheitsgraden
Chemie	→	Verbrennung im Motor
	⇒	Große Systeme (200 ... 100.000) gewöhnlicher Differentialgleichungen
Meteorologie	→	Globale Wettervorhersage
	⇒	# Meßpunkte ~ Vorhersagedauer
Mechanik	→	Simulation von Crashtests, Elastisch-plastische Verformungen
	⇒	Große (nichtlineare) Gleichungssysteme pro Zeitschritt
Strömungen	→	Windkanalsimulation, Design, Turbulenz
	⇒	gekoppelte Systeme (nichtlinearer) nichtsymmetrischer Differentialgleichungen in 3D

Obige Beispiele erheben keinerlei Anspruch auf Vollständigkeit, fast in jedem technischen Anwendungsgebiet lassen sich ähnlich anspruchsvolle Beispiele finden.

Falls die Rechenleistung einmal ausreichen sollte um ein Problem (z.B., Temperaturverteilung in einer elektrisch beheizten Frontscheibe) zufriedenstellend zu lösen, wird sofort die entsprechende inverse Aufgabe (d.h., bestimme die Lage der Heizdrähte bei gemessener Temperaturverteilung) interessant bzw. die in die Gleichungen eingehenden Parameter (d.h., Form/Lage der Drähte, elektrische Spannung in ihnen) sollen so verändert werden, daß bzgl. einer Zielfunktion das Minimum angenommen wird (z.B., Temperaturgleichverteilung).

Klassische VON-NEUMANN-Architektur reicht nicht mehr aus.



Beschleunigung durch *parallele* Verarbeitung.

- *Parallele Ansätze in VON-NEUMANN-Rechenwerken*

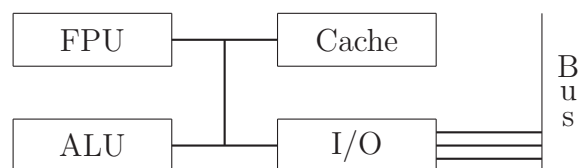


Fig. 1.1 Simplifizierte von-Neumann Architektur

- Gleitkomma-, Integerarithmetik und I/O parallel möglich.
- Instruction Look Ahead (Befehls-cache).
- Memory Interleaving, d.h. getrennter Zugriff auf adressmäßig benachbarte Bytes/Bits, z.B. wird pro Speicherchip jeweils nur 1 Bit eines Bytes gespeichert.

- *Pipelining in der Befehlsabarbeitung*

Tab. 1.2 Pipelining auf Befehlsebene

Zeittakt	i	$i + 1$	$i + 2$	$i + 3$	$i + 4$
Befehl dekodieren	×	•	□	○	
		↘	↘	↘	↘
Adressrechnung		×	•	□	○
			↘	↘	↘
Operanden laden			×	•	□
				↘	↘
Operation				×	•
					↘
Ergebnis speichern					×

Beim Pipelining in Fig. 1.2 wird ein komplexerer Befehl in kleinere Instruktionen aufgespalten, welche unterschiedliche CPU-Einheiten nutzen. Damit ist eine zeitlich überlappende Ausführung aufeinanderfolgender Befehle möglich. Dies tritt auch in Verbindung mit dem Instruction Look Ahead auf. Die RISC-Prozessoren sind nach diesem Prinzip aufgebaut.

- *Weitere Leistungssteigerung*

Im wesentlichen ist eine weitere Leistungssteigerung nur durch Verwendung von mehreren CPUs möglich.

Da viele Probleme der der Physik, Ingenieurwissenschaften, etc. auf partielle Differentialgleichungen führen, fällt der Parallelisierung der entsprechenden Lösungsverfahren eine besondere Rolle zu. Daher konzentrieren wir uns im vorliegenden Buch auf diese Aufgabenklasse.

2 Parallel- und Vektorrechner

2.1 Klassifikationen

2.1.1 Ebenen der Parallelität

In der Parallelität werden die folgenden Ebenen unterschieden :

Definition 2.1. [Job] Ganze Jobs laufen parallel zueinander auf verschiedenen Prozessoren ab. Hierbei tritt keine oder nur eine schwache Interaktion der Jobs zueinander auf.

→ bessere Rechnerauslastung

→ kürzere Realzeit der Jobs

z.B.: Workstation mit mehreren Prozessoren und Multitasking.

Definition 2.2. [Programm] Teile eines Programmes laufen auf mehreren Prozessoren ab.

→ kürzere Realzeit

z.B.: Parallelrechner.

Definition 2.3. [Befehl] Parallelität zwischen den Phasen (Instruktionen) der Befehlsausführung.

→ Beschleunigte Ausführung des gesamten Befehls.

z.B.: serielle Rechner / einzelne Prozessoren.

Definition 2.4. [Arithmetik, Bitebene] Hardwaremäßige Parallelisierung von Integerarithmetik und bitweiser paralleler, aber wortweise serieller Zugriff auf den Speicher bzw. umgekehrt.

→ Weniger Taktzyklen zur Abarbeitung einer Instruktion.

z.B.: 32/64 bit Bussystem.

Die angeführten Ebenen der Parallelität treten auch kombiniert auf.

2.1.2 Hardwareklassifikation

Die Parallelität der Hardware wird wie folgt unterschieden.

Definition 2.5. [Pipelining] Segmentierung von Operationen, welche hintereinander abgearbeitet werden (siehe *Vektorrechner* im nächsten Abschnitt).

Definition 2.6. [Funktionale Einheiten] Verschiedene, funktional unabhängige Einheiten zur Abarbeitung von (verschiedenen) Operationen.
z.B.: Superskalarrechner können Additionen/Multiplikationen/ Logikoperationen gleichzeitig ausführen.

Definition 2.7. [Prozessorfelder] Felder identischer Prozessorelemente zur parallelen Ausführung von (gleichartigen) Operationen.
z.B.: MasPar-Rechner mit 16384 relativ einfachen Prozessoren, systolische Arrays zur Bildverarbeitung.

Definition 2.8. [Multiprocessing] Mehrere voneinander unabhängige Prozessoren mit jeweils eigenem Instruktionssatz. Parallele Ausführung bis hin zu ganzen Programmen oder Jobs. Bis auf die Varianten Prozessorfelder, Multiprocessing können obige Klassifikationen auch kombiniert auftreten.

2.1.3 Grobklassifikation nach Flynn

Im Jahre 1966 unterschied Michael Flynn [Fly 66] parallele Architekturen bzgl. des Datenstroms und des Stromes der Programmanweisungen, so daß sich 4 Formen der Parallelität ergeben.

Tab. 2.1 Flynn's Klassifikation (Flynn's taxonomy)

Single	Multiple		
Instruction Stream			
SISD	MISD	Single	Data
SIMD	MIMD	Multiple	Stream

Im folgenden betrachten wir die einzelnen Klassen im Detail.

Definition 2.9. [SISD] Single Instruction Single Data. Klassische Architektur mit nur einem Anweisungs- und einem Datenstrom.

SISD besitzt nur

- prozessorinterne Parallelität (bitparallel, look-ahead)

- Pseudeoparallelität mittels Multitasking und Time-Sharing (Durchsatz-erhöhung des Computersystems).

Definition 2.10. [SIMD] Single Instruction Multiple Data. Ein einzelner Anweisungsstrom handhabt gleichzeitig mehrere Datenströme.

Hier wird bereits Parallelität auf Anweisungsebene realisiert.

1. Die Daten sind über die Prozessoren verteilt, *ein* Programm wird "im Gleichschritt" auf allen Prozessoren abgearbeitet. Meist sehr *einfache*, dafür aber *vielen* Prozessoren (*Prozessorfelder* CM2, MasPar).

Ein typisches Problem bei SIMD sind Programmalternativen.

Liefert der Test "?" bei 2 Prozessen einer SIMD-Maschine unterschiedliche Ergebnisse (T/F), so ergibt sich formal das linke Struktogramm in Fig.2.1, jedoch spiegelt sich der tatsächliche Zeitablauf auf dem SIMD-Computer in der rechten Darstellung wieder, da beide Zweige der Alternative von beiden Prozessoren abgearbeitet werden müssen.

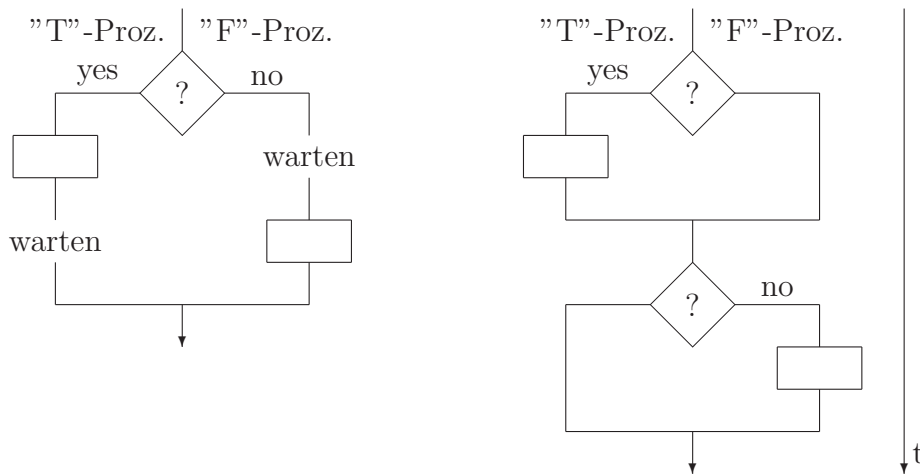


Fig. 2.1 Formaler und zeitlicher Ablauf der Alternative im SIMD-Rechner.

Trotz dieses ungünstigen Verhaltens der SIMD-Rechner wurden bis Anfang der 90er Jahre viele dieser Parallelrechner verkauft, da sie für einige Aufgabenklassen sehr gut angepaßt waren. Durch die Leistungsexplosion der Computerchips und die immer flexibleren Anforderungen an Parallelrechner sind mittlerweile nur noch wenige Spezies dieser Gattung zu finden.

2. Der *Vektorrechner* benutzt das Pipelining derart, daß im Unterschied zum skalaren Rechner die gleiche Befehlssequenzen mit n Datensätzen wiederholt werden. Eine typische Operation ist die SAXPY-Operation $\underline{y} := \underline{y} + \alpha \underline{x}$ in Fig. 2.2, welche zu einem Vektor \underline{y} den mit der reellen Zahl α skalierten Vektor \underline{x} addiert. Die Pipeline besteht in diesem Falle aus Addition, Multiplikation, Lade- und Speicheroperationen welche mit den Vektorregistern durchgeführt werden. Nach einem "Startup" (Berechnung der ersten Komponente) wird i.a. pro Taktzyklus ein Ergebnis y_i geliefert, obwohl in der skalaren Rechnung mehr Takte notwendig sind. Um hohe MFLOP-Raten zu erreichen, müssen

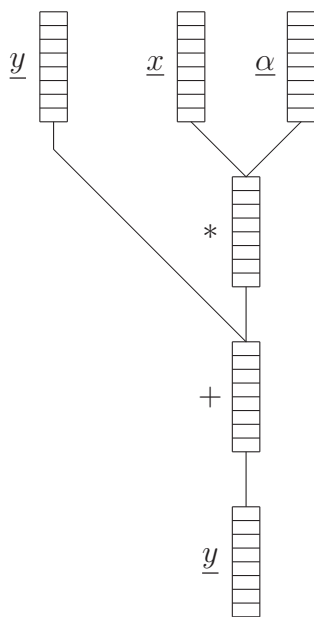


Fig. 2.2
SAXPY im Vektorrechner.

die Vektorregister stets gefüllt sein ! Bei Nichtbeachtung dieser Forderung ist der Einsatz des Vektorrechners sinnlos, da nur die skalare Rechenleistung erreicht wird. In nebenstehender Abbildung tritt ein Vektor $\underline{\alpha}$ auf, welcher nichts anderes als das Auffüllen des entsprechenden Vektorregisters mit dem Skalar α symbolisiert. Dies wird automatisch vom Compiler getan, da der Vektorrechner eigentlich nur die Operation $\underline{y} := \underline{y} + \underline{\alpha}^T \otimes \underline{x}$ effizient behandeln kann. Das Zeichen \otimes bezeichnet die komponentenweise Multiplikation der beiden Vektoren.

Aus der Sicht des Mathematikers, im Sinne der Parallelisierung mathematischer Algorithmen, beschleunigt der Vektorrechner "nur" die sequentiellen Algorithmen ohne etwas an ihnen zu ändern.

Definition 2.11. [MIMD]Multiple Instruction Multiple Data. Dies bedeutet Parallelität auf Programmebene, d.h. jeder Prozessor arbeitet sein eigenes Programm ab.

Die Programme arbeiten in der Regel aber nicht unabhängig voneinander, daher unterscheidet man :

1. konkurrierende Prozesse (gemeinsame Ressourcen)
2. kommunizierende Prozesse (Datenfluß, Datenaustausch)

Wir unterscheiden zwischen Programm und Prozeß.

Definition 2.12. [Programm (statisch)] Exakte Niederschrift eines Algorithmus.

Definition 2.13. [Prozeß (dynamisch)] Folge von Aktionen, einzige Annahme: positive Geschwindigkeit.

Für die beiden Arten von betrachteten Prozessen ergeben sich folgende Varianten eines Programmablaufes.

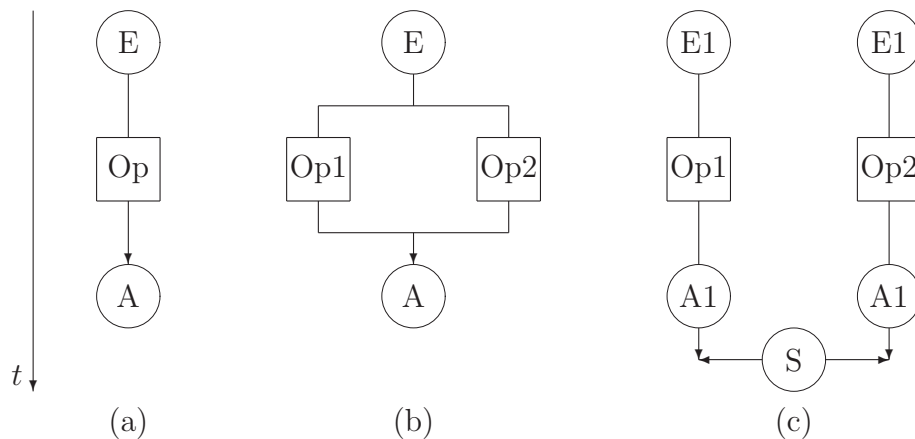


Fig. 2.3 Programmablauf für sequentielle (a), konkurrierende (b) und kommunizierende (c) Prozesse

Die konkurrierenden Prozesse in Fig. 2.3b benutzen die gleichen Eingangsdaten und müssen ihr Ergebnis wiederum gemeinsam abspeichern. Die kommunizierenden Prozesse in Fig. 2.3c können völlig unabhängig voneinander ihre Eingangsdaten verarbeiten und ihre Ergebnisse abspeichern. Da die Prozesse aber im allgemeinen an einer gemeinsamen Problemstellung arbeiten, ist an bestimmten Punkten ein Abgleich/Austausch der Ergebnisse notwendig (Synchronisation).

Oft repräsentiert jeder Prozeß dasselbe Programm, allerdings mit unterschiedlichen Daten. Damit läßt sich eine Unterklasse der MIMD-Computer ableiten, genannt SPMD.

Definition 2.14. [SPMD] Single Program Multiple Data. Auf jedem Prozessor wird das gleiche Programm gestartet, welches individuell seine Daten verarbeitet.

Die Klasse SPMD

- wird auf größeren Parallelrechnersystemen und vor allem bei massiv parallelen Rechnern verwendet.
- benötigt Datenparallelität.
- ist heute *das* Programmiermodell für Parallelrechner.
- ist *kein* SIMD, da kein Gleichschritt in Anweisungsebene erforderlich. Jedoch müssen zu den Synchronisationszeitpunkten Daten abgeglichen werden.

2.1.4 Klassifikation nach Speicherzugriff

Definition 2.15. [Shared Memory] Konkurrierende Prozesse greifen auf einen gemeinsamen Speicher zu, d.h. sie teilen sich den Speicher.

+ Jeder Prozeß kann auf sämtliche Daten zugreifen.

⇒ Serielles Programm kann ohne größere Schwierigkeiten zum Laufen gebracht werden. Führt in der Regel bei kleinen Prozessorzahlen (2 ... 16) schnell zu einer ersten Leistungssteigerung.

− Es ist ein Anwachsen der Zugriffskonflikte (z.B. Bankkonflikte) bei größeren Prozeßanzahlen zu beobachten. Somit ist die *Skalierbarkeit* (d.h. Leistung \sim Prozeßanzahl) nicht mehr gewährleistet.

− Zur Verringerung der Zugriffskonflikte sind sehr leistungsfähige Bussysteme bzw. Zugriffsverwaltungen notwendig.

⇒ Verteuerung des Gesamtsystems.

Definition 2.16. [Distributed Memory] Der Gesamtspeicher ist auf die einzelnen Prozessoren verteilt, so daß die kommunizierenden Prozesse während des gesamten Programmablaufs nur auf ihren lokalen Speicher zugreifen können. Werden Daten eines anderen Prozesses benötigt, kann dies nur über explizite Kommunikation zwischen den Prozessen erfolgen.

+ Keine Zugriffskonflikte bei Datenoperationen (Lokalität).

+ Relativ preiswerte Hardware.

+ Fast beliebig skalierbar.

− Kein direkter Zugriff auf die Daten anderer Prozesse möglich.

⇒ Nachrichtenaustausch (Kommunikation) über spezielle Kanäle (Links) notwendig.

⇒ Serielles Programm ist nicht lauffähig, es werden spezielle parallele Algorithmen benötigt.

- Der Aufwand für Kommunikation wurde noch vor ca. 20 Jahren vernachlässigt, heute aber ist er mitentscheidend für die Qualität eines parallelen Algorithmus (u.a. Verhältnis Kommunikation zu Arithmetik).
- Die Geschwindigkeit der Verbindungsnetzwerke ist von großer Bedeutung.

Definition 2.17. [Distributed Shared Memory (DSM)] DSM, auch *Virtual Shared Memory* genannt, ist der intelligente Versuch eines Kompromisses zwischen Shared und Distributed Memory. Ein dem Distributed Memory aufgesetztes Message-Passing-System simuliert das Vorhandensein eines globalen Shared Memory (KSR: "Meer von Adressen", SGI: "Interconnection fabric").

⊕ Auf diesem Speichermodell können serielle Programme sofort zum Laufen gebracht werden. Wenn die benutzen Algorithmen die Lokalität der Daten ausnutzen (d.h. mehrheitlich auf den dem Prozeß zugeordneten Speicher zugreifen) ist auch eine gute Skalierbarkeit erreichbar.

So wurde, z.B. im Frühjahr 1997 von SGI die Origin2000 mit dem Skalierbaren Symmetrischen Multiprocessing (S2MP) auf den Markt gebracht¹. Jeder Prozessor dieser Maschine besitzt seinen eigenen lokalen Speicher, für die Gesamtmaschine ist der Gesamtspeicher aber als Shared Memory verfügbar. Möglich wurde dies durch den sehr schnellen Crossbar Switch von Cray² (seit 1996 Tochtergesellschaft von SGI).

Bemerkung 2.1. Im Rahmen des EUROPORT³-Projektes wurden 1995-1997 38 kommerzielle Codes parallelisiert. Bzgl. der Parallelisierbarkeit auf verschiedenen parallelen Programmiermodellen und Parallelrechnern sind hierzu interessante Vergleiche⁴ verfügbar. Insbesondere zeigte sich, daß ein distributed-memory-Konzept zur Datenlokazität in einem shared-memory Programm auf DSM-Hardware sehr erfolgreich eingesetzt werden kann und derart gestaltete Programme den automatisch parallelisierten Codes im allgemeinen überlegen ist.

¹*SGI Performer*, Oct. 96, pp. 4/5

²<http://www.cray.com>

³<http://www.gmd.de/SCAI/europort/>

⁴<http://www.gmd.de/SCAI/europort-1/EUROPORT.HTM>

2.2 Topologien

Wie sind parallel arbeitende Recheneinheiten eigentlich miteinander verbunden? Insbesondere ist dies für die Klasse der Parallelrechner mit distributed memory von Interesse.

2.2.1 Definitionen

Definition 2.18. [Link] Ein Link ist eine Verbindung zwischen 2 Prozessen/Prozessoren.

- unidirektionales Link: zu einem Zeitpunkt nur in eine Richtung benutzbar.
- bidirektionales Link: zu jedem Zeitpunkt in beide Richtungen benutzbar.

Definition 2.19. [Topologie] Mit Topologie bezeichnen wir ganz allgemein die Vernetzung der Prozesse/Prozessoren.

Definition 2.20. [Physikalische Topologie] Vom Hersteller vorgegebene hardwaremäßige Vernetzung der Recheneinheiten (Knoten). Diese Vernetzung ist teilweise manuell (Xplorer, MultiCluster-I) oder auch softwaremäßig konfigurierbar (Crossbar Switches im MultiCluster-II).

Beispielsweise besitzt der INMOS-Transputer T805 4 physische Links.

Definition 2.21. [Logische Topologie] Vom Nutzer gewünschte oder vom Betriebssystem vorgegebene Vernetzung der Prozesse [Knoten], manchmal als virtuelle Topologie bezeichnet. Wird meist von der Daten- bzw. Kommunikationsstruktur bestimmt. Die Abbildung der logischen auf die physikalische Topologie geschieht mittels paralleler Betriebssysteme oder Betriebssystemerweiterungen (Abschnitt 2.3).

Bemerkung 2.2. Am effizientesten ist natürlich die Identität von logischer und physikalischer Topologie
 \implies Widerspruch zu Datenstruktur bei vielen Anwendungen.

Definition 2.22. [Durchmesser einer Topologie] Der Durchmesser einer Topologie ist die maximale Anzahl von Linkverbindungen, welche eine Nachricht benutzen muß, um von einem Knoten p zu einem Knoten q zu gelangen (p, q beliebige Knoten im Netzwerk).

Definition 2.23. [Host] Ein Host ist ein, insbesondere bei älteren Parallelrechnern, notwendiger Computer, welcher die Verbindung des Parallelrechners

zu den Ressourcen wie Eingabe-, Ausgabedaten, Grafik, Festplatten etc. herstellt. Bei neueren Systemen wird diese Aufgabe vom Parallelrechner selbst erledigt.

2.2.2 Farm / Master-Slave Topologie

Der Begriff ist angeblich vom Aufseherprinzip in den Plantagen des Südens der USA abgeleitet. Hier beaufsichtigt und verteilt ein Masterprozeß die Arbeit an die Slaveprozesse.

- $P - 1$ Links beim Master notwendig.
- Jede Kommunikation belastet den Master.
- \implies Nur dann praktikabel, wenn kein (oder äußerst geringer) Datenaustausch zwischen den Prozessen notwendig ist.

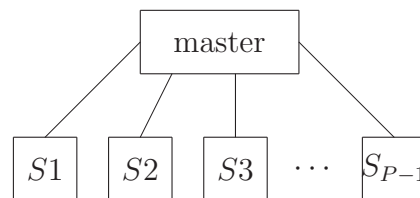


Fig. 2.4
Master-Slave Topologie.

Die Master-Slave Topologie findet in Monte-Carlo-Methoden (höherdimensionale numerische Integralberechnung), bei Teilchenverfolgungen und teilweise bei Bildbearbeitung und -erkennung ihre Anwendung.

2.2.3 Die Pipe

Eine Pipe besteht in einer eindimensionalen Anordnung der Knoten/Prozesse.

- 2 Links pro Knoten notwendig.
- Durchmesser $P - 1$.
- Gut geeignet für eindimensionale Datenabhängigkeiten.



Fig. 2.5
Die Pipe

Die Pipe kann verwendet werden

- im Gauß-Algorithmus für vollbesetzte Gleichungssysteme,
- in *einer* 1D-Gauß-Seidel Iteration bei vollbesetzten Gleichungssystemen,
- bei *mehreren* 1D-Gauß-Seidel Iterationsschritten und spezieller Struktur des Gleichungssystems (FEM/FDM).

2.2.4 Der Ring

Schließt man die Pipe zyklisch ab, so entsteht ein Ring.

- 2 Links pro Knoten.
- Durchmesser: $P/2$.
- Es sind zyklische Datenabhängigkeiten realisierbar.

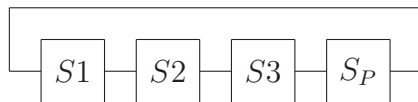


Fig. 2.6
Der Ring.

Die Ringtopologie wird eingesetzt :

- im Gauß-Algorithmus,
- bei *mehreren* 1D-Gauß-Seidel-Schritten, auch mit vollbesetzten Matrizen,
- beim Ein- bzw. Auslesen großer Datenmengen zum/vom Parallelrechner über einen, mit dem Hostrechner verbundenen, Knoten (Gesamtdatenmenge übersteigt den verfügbaren lokalen Speicher)

2.2.5 Array und Torus

Die logische Weiterentwicklung von Pipe und Ring in mehrere Dimensionen sind das Array und der Torus welche man als Tensorprodukt von Pipes bzw. Ringen auffassen kann.

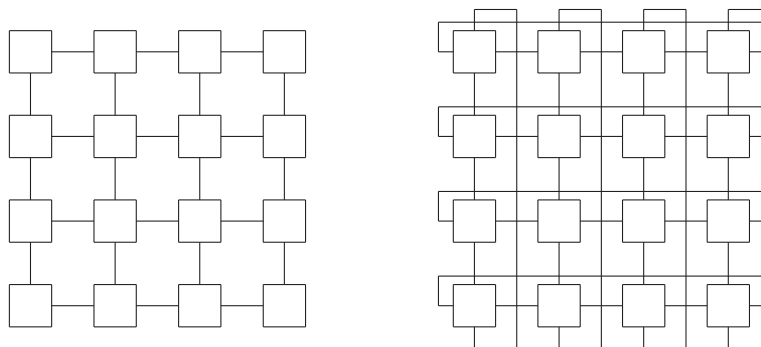


Fig. 2.7 2D-Array (links) und 2D-Torus (rechts) mit $P = P_x \cdot P_y$.

Die Arraytopologie wurde insbesondere in den SIMD-Computern der Firma MasPar verwendet. Jedoch haben auch die Anbieter von MIMD-Rechnern diese Topologie verwandt, als Beispiel seien die Parallelrechner Xplorer (2D) und Power-GC (3D) der Firma Parsytec angeführt.

Tab. 2.2 Charakteristische Größen bei Array und Torus

Topologie	Anzahl der Knoten	Links pro Knoten	Durchmesser
2D-Array	$P_x \cdot P_y$	4	$P_x + P_y - 2$
2D-Torus	$P_x \cdot P_y$	4	$(P_x + P_y)/2$
3D-Array	$P_x \cdot P_y \cdot P_z$	6	$P_x + P_y + P_z - 3$
3D-Torus	$P_x \cdot P_y \cdot P_z$	6	$(P_x + P_y + P_z)/2$

2.2.6 Der Baum (Tree)

Der Baum (engl.: Tree) in Fig. 2.8 ist eine sehr flexible Topologie mit guten Kommunikationseigenschaften und ist daher vielfältig einsetzbar.

Definition 2.24. [Optimaler Baum] Baumtopologie, deren maximale Tiefe und Verästelung (d.h. Anzahl der Verzweigungen pro Knoten) nicht größer als $d = \log_2 P$ ist.

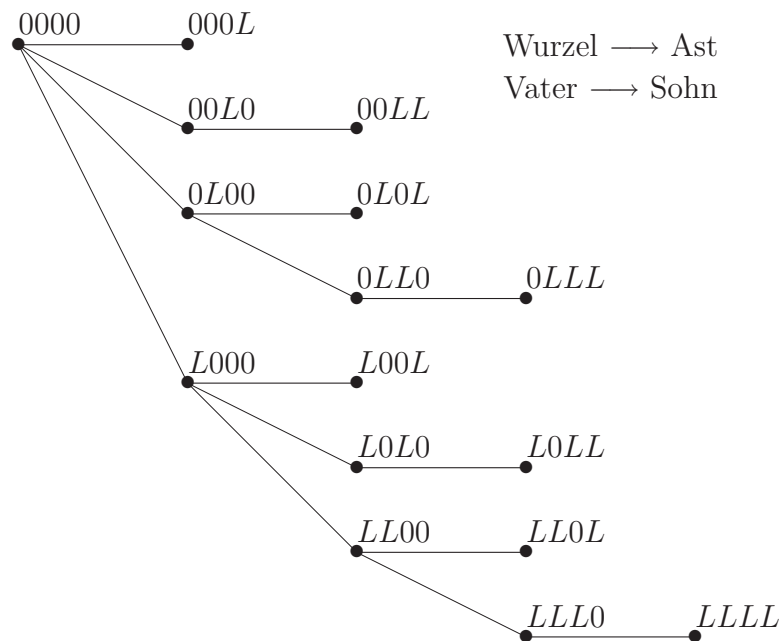


Fig. 2.8 Optimaler Baum in Binärdarstellung

Wir verwenden im weiteren die Bezeichnung Baum stets im Sinne eines optimalen Baumes.

Somit gilt für Bäume stets

- $2^{d-1} < P \leq 2^d$,

- Max. d Links pro Knoten,
- Durchmesser: $2d - 1 = 2 \log_2 P - 1$.

Zur Konstruktion des binären Baumes benötigen wir den Gray-Code.

Definition 2.25. [Gray-Code] Die Binärdarstellungen der Nummern benachbarter Knoten unterscheiden sich in genau einem Bit ($[1, \dots, d]$) !

Bemerkung 2.3. [Konstruktionsprinzip des optimalen Baumes]

Von einem beliebigen Knoten im Baum mit der Wurzel in 000 werden so viele Äste gebildet, wie 0-Bits dem letzten L -Bit der Binärdarstellung der Knotennummer folgen.

Die Firma Convex-MPP führte das Konzept der "Fat Trees" ein, d.h. je näher sich ein Link an der Wurzel des Baumes (Knoten 0000 in Fig. 2.8) befindet, desto leistungsfähiger wird die entsprechende Datenleitung ausgelegt.

2.2.7 Der Hypercube

Ein Würfel hat 8 Knoten, 12 Kanten und kann durch die Verbindung der Knoten zweier (räumlich paralleler) Quadrate erzeugt werden. Bezeichnet man nunmehr diesen Würfel als 3-dimensionalen Hypercube mit $P = 2^3$ Knoten, dann hat man damit bereits das Grundprinzip des *rekursiven Aufbaus* eines d -dimensionalen Hypercubes.

Rekursiver Aufbau des Hypercubes

- Ein Hypercube der Dimension d ergibt sich durch die entsprechende Verbindung zweier Hypercubes der Dimension $d - 1$.
- Die Platzierung der Knoten erfolgt nach dem Gray-Code, d.h. im Hypercube benachbarte Knoten unterscheiden sich in *genau einem* Bit ihrer Knotennummern in Binärdarstellung. Die Position dieses Bits legt eindeutig die Nummer des die beiden Knoten verbindenden Links fest (z.B. sind Knoten $5 = LOL$ und $4 = L00$ über Link 1 im 3-dimensionalen Hypercube miteinander verbunden).
- Die Links mit gleicher Linknummer (=Bitposition) teilen den d -dimensionalen Hypercube in 2 $(d - 1)$ -dimensionale Hypercubes. Je nachdem, ob die zum Link gehörende Bitposition gesetzt ist, ordnen sich die Knoten den beiden *Subcubes* zu. So teilt z.B. Link 2 den 3-dimensionalen Hypercube in Abbildung 2.9 in die Subcubes $\{0, 1, 4, 5\}$ und $\{2, 3, 6, 7\}$.

Der d -dimensionale Hypercube hat damit folgende Eigenschaften :

- $P = 2^d$ Knoten,
- d Links pro Knoten,
- Durchmesser: $d = \log_2 P$,

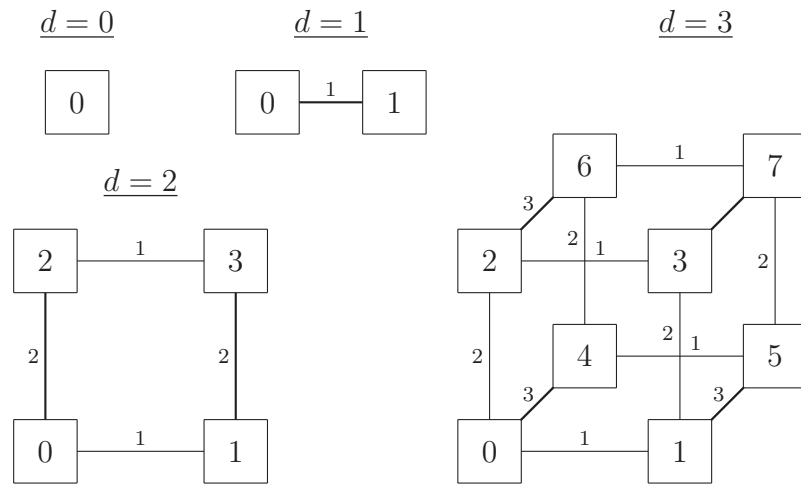


Fig. 2.9 Hypercubes $d = 0, \dots, 3$

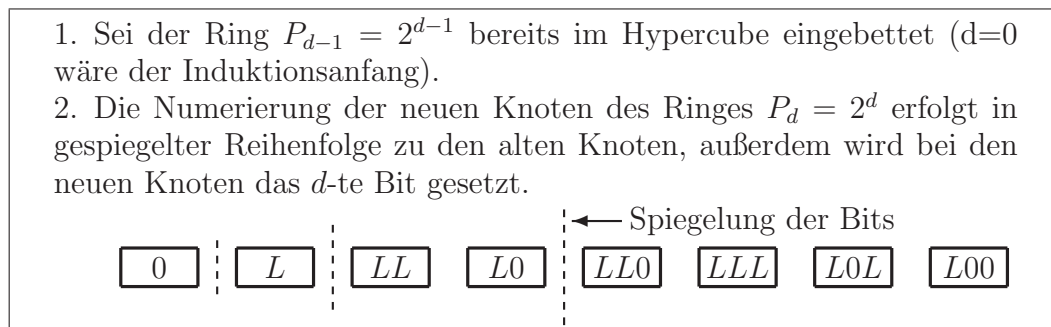
- Der kürzeste Weg im Hypercube zwischen den Knoten p und q ergibt sich aus der bitweisen exklusiv-oder-Operation $XOR(p,q)$, worin die gesetzten Bits den Linknummern entsprechen, welche die Verbindung zwischen den Knoten bilden.

Bsp.: $p = 1 = 00L, q = 7 = LLL \implies XOR(1,7) = LL0$.

Der Weg in obigem Beispiel ist in einen 2-dimensionalen Subcube des Hypercubes eingebettet.

Die Firma nCube hatte einen Hypercube der Dimension 13 mit Spezialprozessoren hardwaremäßig realisiert, konnte allerdings Mitte der 90er Jahre mit der rasanten Leistungssteigerung der Konkurrenten nicht mehr Schritt halten.

Die Topologien Ring, Torus und Tree (2.2.3 - 2.2.6) sind im Hypercube eingebettet. So läßt sich, z.B. ein Ring mit $P = 2^d$ Knoten ähnlich rekursiv wie der Hypercube definieren und somit in diesen einbetten:



Alg. 2.1 Einbettung des Ringes in den Hypercube

In Fig. 2.10 sieht man das Ergebnis von Alg. 2.1.

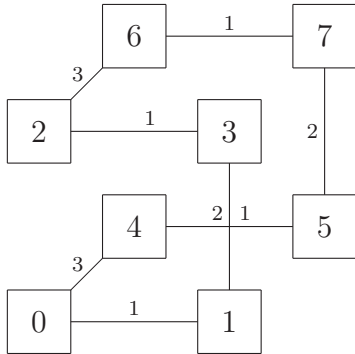


Fig. 2.10
Einbettung des Ringes im 3D-Hypercube

Aufgabe 2.1. Betten Sie einen $2^k \times 2^\ell$ -Torus in einen $(k + \ell)$ -dimensionalen Hypercube ein.

Aufgabe 2.2. Zeigen Sie anhand des d -dimensionalen Hypercube, daß der optimale Baum (Punkt 2.2.6) mit der Wurzel im Knoten 000 im Hypercube eingebettet ist. Wieviele verschiedene Einbettungsmöglichkeiten gibt es ?

Aufgabe 2.3. Geben Sie eine Einbettung des optimalen Baums mit der Wurzel im Knoten $L0L$ für den 3-dimensionalen Hypercube an.

Aufgabe 2.4. Bilden Sie die Knoten der beiden Bäume aus den beiden vorgegangenen Aufgaben mittels einer einfachen Funktion aufeinander ab.

Die bisher betrachteten Topologien besitzen die in Tabelle 2.3 aufgeführten Eigenschaften. Während bei Ring und Torus die Anzahl der Links mit der Raumdimension beschränkt ist (was aber mit einer großen Topologiedurchmesser erkauft werden muß), steigen die pro Knoten benötigten Links bei Tree und Hypercube mit der Anzahl der benötigten Knoten an (was für größere Knotenzahlen in technologischen Problemen resultiert).

Tab. 2.3 Eigenschaften von Topologien

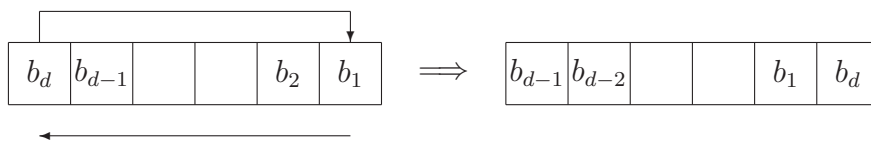
Topologie	# Knoten	Durchmesser	Links pro Knoten
Ring	P	$P/2$	2
Torus	P	$P/2$	$2 * \text{Raumdim.}$
Opt. Baum	$P = 2^d$	$2d - 1$	$1, \dots, d$
Hypercube	$P = 2^d$	d	d

Wünschenswert wäre eine Topologie, welche einen Durchmesser von der Ordnung $\mathcal{O}(\log_2(P))$ bei kleiner, konstanter Anzahl der Links pro Knoten besitzt. Eine solche Topologie wird im nächsten Abschnitt vorgestellt.

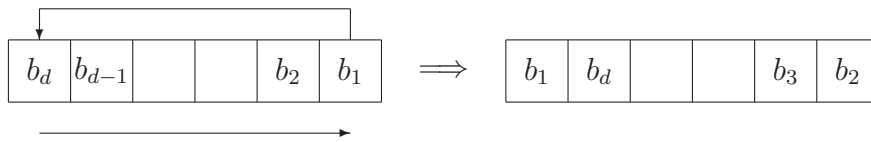
2.2.8 Das DeBrujin-Netzwerk

Zunächst einige Begriffe. Die Shuffle Operationen verschieben die Bits einer Integerzahl in Binärdarstellung, bei der Shuffle Exchange Operation wird zusätzlich ein Bit negiert.

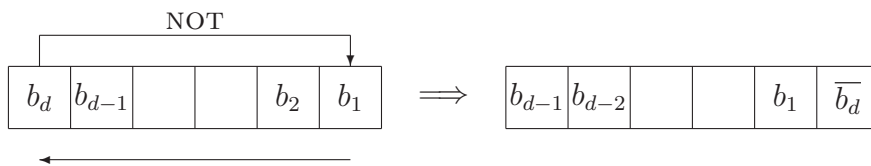
Left Shuffle [LS] = $\text{ISHFTC}(b, 1, d)$ (im weiteren Link 1)



Right Shuffle [RS] = $\text{ISHFTC}(b, -1, d)$ (im weiteren Link 2)



Left Shuffle Exchange [LSE] = $\text{IOR}(\text{ISHFTC}(b, 1, d), 2^{d-1})$ (Link 3)



Right Shuffle Exchange [RSE] = $\text{IOR}(\text{ISHFTC}(b, -1, d), 2^{d-1})$ (Link 4)

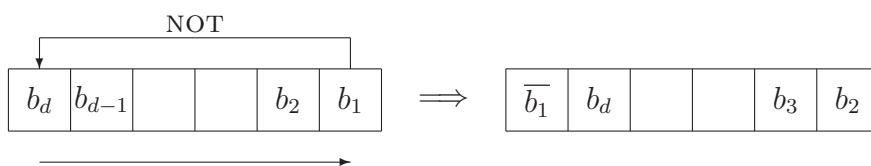


Fig. 2.11 Die Shuffle Operationen

Das DeBrujin-Netzwerk wird unter den folgenden Voraussetzungen mit Hilfe der Shuffle Operationen konstruiert.

- Jeder Knoten (außer Knoten 0 und $2^d - 1$) besitzt genau 4 Links. Die Knoten 0 und $2^d - 1$ benötigen nur 2 Links.
- Ein Netzwerk der Dimension d besitzt 2^d Knoten und $2^{d+1} - 2$ (gerichtete) Kanten.

- Es gibt nur gerade Knoten im Graphen
 \implies Graph ist ohne Überschneidung durchlaufbar.
 \implies Ring ist eingebettet.
- Der Durchmesser ist d .

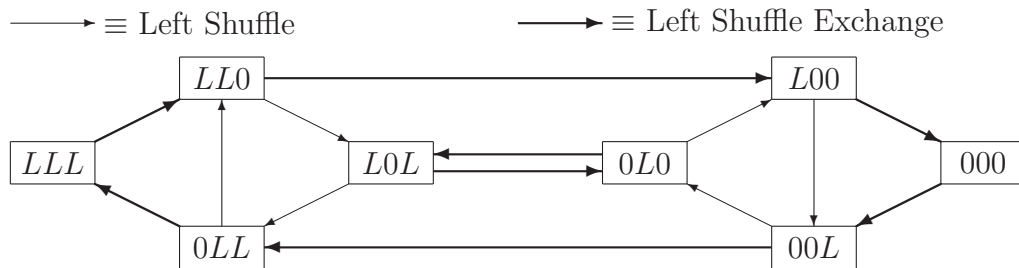


Fig. 2.12 DeBruijn Netzwerk $d = 3$: 8 Knoten, 14 Kanten

Es ist keine kommerzielle Hardwarerealisierung des DeBruijn Netzwerks bekannt.

Aufgabe 2.5. Für welche Knotennummern $i = 0, \dots, 2^d - 1$ im *de Bruijn*-Netzwerk gilt

$$\text{RightShuffle}(i) = \text{LeftShuffle}(i) \text{ bzw.}$$

$$\text{RightShuffleExchange}(i) = \text{LeftShuffleExchange}(i) \text{ ?}$$

Hinweis: Unterscheiden Sie, ob d gerade bzw. ungerade ist.

Aufgabe 2.6. Stellen Sie ein *de Bruijn*-Netzwerk für $d = 4$ auf !

2.2.9 Freie Topologie

Grundidee: Die Beziehungen zwischen den Daten (z.B. Gebietsaufteilung) werden auf das Knotenfeld abgebildet, d.h. die geometrischen Nachbarschaftsbeziehungen werden durch topologischen widerspiegelt.

- Es sind bis zu $P - 1$ Links pro Knoten notwendig.
- Der Durchmesser der Topologie ist bis auf 1 reduzierbar.
- Meist nur als virtuelle Topologie verfügbar.
- Sehr leistungsfähige Hardware notwendig, insbesondere darf das Weiterleiten von Daten über einen Knoten (*Hop*) die Rechenleistung des Knoten nicht beeinflussen.

\implies Kommunikation und Rechnung parallel auf einem Knoten (T805 von Inmos)

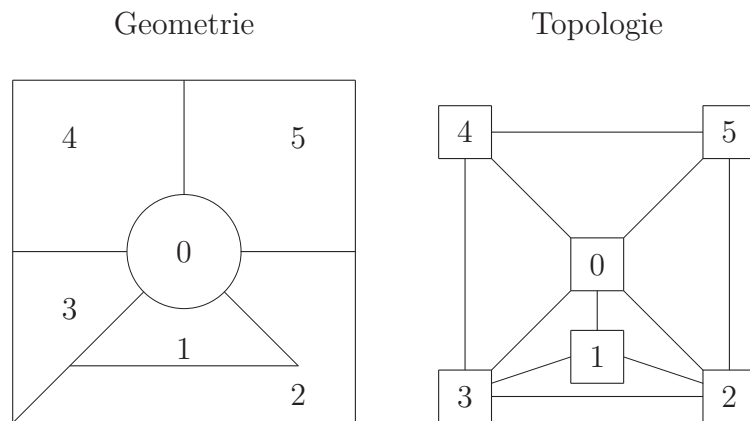


Fig. 2.13 Abbildung von räumlichen Relationen auf eine freie Topologie

- Unterstützung durch parallele Betriebssystemerweiterungen (PARIX, PVM, MPI) erforderlich.

2.3 Parallele Betriebssystemerweiterungen und Programmiersprachen

Zu Beginn der vermehrten Anwendung von Parallelrechnern in der Forschung wurden nur bestimmte, hardwareabhängige Routinen unterstützt. Dahingehend wurde z.B. in

- Occam (C/Assemblerähnlich, sehr hardwarenah)
- Erweiterungen für Fortran, C

programmiert.

⇒ Schlechte Portierbarkeit der Quellen.

⇒ Eigene Bibliotheken, mit sehr wenigen hardwareabhängigen Rufen.

Hersteller von Parallelrechnern bieten z.T. eigene Betriebssystemerweiterungen für UNIX an, welche gewisse Aufgaben zwischen Host und Parallelrechner regeln und die notwendigen Compiler, Bibliotheken etc. enthalten.

Mittlerweile wurden parallele Hochsprachen wie High Performance Fortran (HPF⁵), Vienna Fortran⁶ u.a. entwickelt. Diese Sprachen enthalten gewisse, die Parallelität unterstützende, Sprachkonstrukte, bzw. versucht der Compiler

⁵<http://www.mhpcc.edu/doc/hpf/hpf.html>

⁶<http://www.vcpc.univie.ac.at/activities/language/vf/ViennaFortran.html>

eine automatische Parallelisierung. Leider sind diese parallelisierenden Compiler auf wohlstrukturierte Daten (Tensorproduktgitter, spezielle Matrizen, ältere Gleichungssystemlöser) angewiesen, was ihre praktische Verfügbarkeit auf einschränkt. Für unsere Zwecke, d.h., Lösen von PDEs auf unstrukturierten Gittern, sind sie ungeeignet.

Eine andere Entwicklungsrichtung besteht in der Entwicklung einer allgemeinen, portablen Schnittstelle für MIMD-Rechner. Das Message Passing Interface (**MPI**) ist der aktuelle Abschluß dieser Bemühungen (hervorgegangen aus PVM, Express, u.a.). Da MPI⁷ mittlerweile von allen größeren Herstellern von Parallelrechnern unterstützt wird (meist sitzen Vertreter im MPI-Konsortium) und mittlerweile auf Workstationclustern (LINUX⁸!), ja sogar unter Windows läuft, folgen wir im Buch diesem Ansatz. Zum Verständnis der Rufe in MPI⁹ ist Abschnitt 3.3 unbedingt erforderlich [Fos 94, GLS 94, Pac 97].

⁷<http://www.mcs.anl.gov/mpi/index.html>

⁸<http://www.suse.de/>

⁹<http://www.usfca.edu/mpi>

3 Parallelität auf algorithmischer Ebene

3.1 Einige Begriffe

Definition 3.1. [Skalierbarkeit] Einfache Anpassung eines Programmes an eine real verfügbare Anzahl von Prozessoren.

Die Skalierbarkeit wird mittlerweile höher bewertet als höchste Effizienz (d.h. Gewinn an Rechenzeit durch Parallelisierung) auf einer speziellen Architektur.

Definition 3.2. [Granularität] Größe der Programmabschnitte, welche ohne Kommunikation mit anderen Prozessoren ausführbar sind.

Definition 3.3. [Feinkörniger Algorithmus] Algorithmus, der nach wenigen Arithmetikoperationen wieder Daten von anderen Prozessen benötigt.

Feinkörnige Algorithmen sind besonders geeignet für die sogenannten systolischen Felder (SIMD-Computer).

Beispiel 3.1. [ω -Jacobi-Iteration und Differenzenschema]
Das Laplace-Problem

$$-\Delta u(x) = f(x) \quad x \in (0, 1)^2$$

mit beliebigen Randbedingungen wird mit einem 5-Punkte-Differenzenstern diskretisiert und resultiert in $(N + 1)^2$ Diskretisierungspunkten. Dies führt bei einer zeilenweisen Numerierung der Diskretisierungspunkte (bei 0 beginnend) im Gebietsinneren zu folgender Notation für eine ω -Jacobi-Iteration, $\forall i, j = \overline{1, N-1}$:

$$\begin{aligned} d_{i,j} &:= c_{i,j} + (f_{i,j} - \omega [-c_{i-1,j} - c_{i,j-1} + 4c_{i,j} - c_{i+1,j} - c_{i,j+1}]) / 4 \\ \underline{c} &:= \underline{d} \end{aligned}$$

Hier treten keinerlei Datenabhängigkeiten innerhalb einer Iteration auf.
 \implies Gut parallelisierbar (und vektorisierbar).

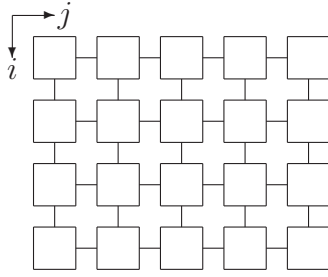


Fig. 3.1
Jacobi-Iteration im systolischen Feld

Beispiel 3.2. [ω -Gauß-Seidel-Iteration (vorwärts)] Wie in Bsp. 3.1. betrachten wir das mit dem 5-Punkte-Differenzenstern diskretisierte Laplace-Problem. Die ω -Gauß-Seidel Iteration kann dann als

$$\widehat{c}_{i,j} := c_{i,j} + (f_{i,j} - \omega [-\widehat{c}_{i-1,j} - \widehat{c}_{i,j-1} + 4c_{i,j} - c_{i+1,j} - c_{i,j+1}]) / 4$$

notiert werden, $\forall i, j = \overline{1, N-1}$.

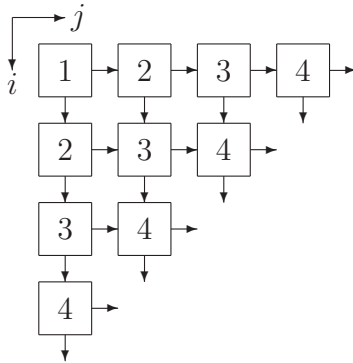


Fig. 3.2
Gauß-Seidel im systolischen Feld

Die neuen Werte $\widehat{c}_{i-1,j}$ und $\widehat{c}_{i,j-1}$ müssen berechnet sein bevor $\widehat{c}_{i,j}$ berechnet werden kann. Daraus ergeben sich die Schlußfolgerungen :

- Die Berechnungen in einer Iteration erfolgen in Form einer sich diagonal bewegenden Front, in Fig. 3.2 sind 1,2,3,4-ter Schritt dargestellt.
- Das systolische Feld habe die Dimension $P = m \times n$.
 → $(m + n - 1)$ Zeitschritte für eine Iteration nötig.
 → mehrere Iterationen kompensieren diesen Overhead der Initialisierung.

1. Iteration :

$$\text{Zeitgewinn} : \frac{t_{\text{seriell}}}{t_{\text{parallel}}} = \frac{mn}{m+n-1} \leq \frac{mn}{2\sqrt{m(n-1)}} = \frac{n}{2(n-1)} \sqrt{m(n-1)} < \sqrt{mn}$$

Dies ist nur ein maximaler Zeitgewinn von \sqrt{P} , d.h., die Verwendung von 100 Rechenknoten würde diese Iteration bestenfalls 10-mal beschleunigen. Ideal wäre aber ein Zeitgewinn von $P = mn$!

2. und folgende Iterationen :

Nach der Initialisierung benötigt eine Iteration nur noch *einen* Zeitschritt, da

die benötigten Daten bereits berechnet sind (Fig. 3.2)

Die k -te (letzte) Iteration benötigt die gleiche Zeit wie die erste Iteration

$$\begin{aligned} \left(\frac{t_{\text{seriell}}}{t_{\text{parallel}}} \right)^{-1} &= \frac{t_{\text{parallel}}}{t_{\text{seriell}}} = \frac{2(m+n-1) + k - 2}{k \cdot mn} \\ &= \frac{2(m+n-1)}{k \cdot mn} + \frac{k}{k \cdot mn} - \frac{2}{k \cdot mn} \\ &\geq \frac{4\sqrt{mn}}{k \cdot mn} + \frac{1}{mn} - \frac{2}{k \cdot mn} \xrightarrow{k \rightarrow \infty} \frac{1}{mn} \end{aligned}$$

Asymptotisch, d.h. bei vielen Iterationen wird der bestmögliche Zeitgewinn $P = m \cdot n$ durch die Parallelisierung erzielt.

Ein vergleichbarer Effekt wird bei cachebasierten Iterationsverfahren [SRWH 97] und bei Vektorrechnern erzielt.

Definition 3.4. [Grobkörniger Algorithmus] Algorithmus, welcher größere Programmabschnitte abarbeiten kann, ohne neuerlich Daten von anderen Prozessen zu benötigen.

Blockvarianten der Algorithmen in Bsp. 3.1. und 3.2. sind grobkörnige Algorithmen. Solche Algorithmen sind prinzipiell für alle Arten von Parallelrechnern gut geeignet, typischerweise benutzt man hierfür MIMD-Rechnermodelle.

Bemerkung 3.1. Heutzutage ist der Zeitbedarf für eine Kommunikation zwischen den Knoten/Prozessoren deutlich höher als für lokalen Speicherzugriff bzw. arithmetische Operationen. Daher sind grobkörnige Algorithmen für die Parallelisierung eher gefragt.

Definition 3.5. [Funktionale Parallelität] Aufspaltung eines Algorithmus in funktional unterschiedliche, parallel abarbeitbare Teilschritte.

Beispiel 3.3. [Funktionale Parallelität] Für die Operation

$$E = (d * e * f + g + c) * b + a + h$$

ergeben sich folgende Abarbeitungszeiten :

seriell (von links nach rechts)	7 Zeitschritte,
parallel auf 2 Prozessoren (Fig. 3.3)	5 Zeitschritte

Aufgabe 3.1. Teilen Sie obige Berechnungsvorschrift auf 3 Prozessoren auf. Nutzen Sie Umformungsregeln. (Ergebnis: 4 Zeitschritte)

- Im parallelen Fall können in Summe mehr arithmetische Operationen auftreten als bei serieller Abarbeitung, dank der Verteilung der Arithmetik ist die parallele Variante trotzdem (potentiell) schneller.

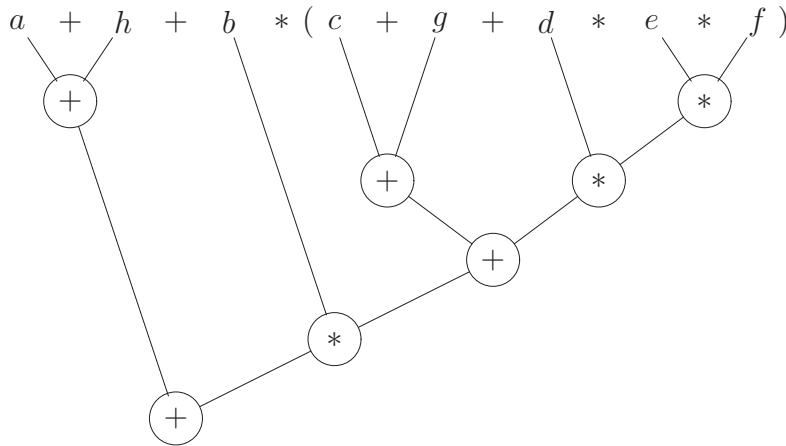


Fig. 3.3 Funktionale Parallelität auf 2 Prozessen

- Funktionale Parallelität ist nur bedingt skalierbar. In obigem Beispiel ergeben mehr als 3 Prozessoren keine weitere Zeitersparnis.

Definition 3.6. [Datenparallelität] Gleiche Programmabschnitte werden parallel über verschiedenen Datenssegmenten ausgeführt.

- Voraussetzung für Datenparallelität: Einfache Aufteilung der Daten in Segmente (High Performance Fortran).
- Algorithmen mit Datenparallelität sind (relativ) leicht skalierbar. Probleme treten bei komplexen Datenzusammenhängen auf, z.B. bei indirekter Adressierung (FEM).

Beispiel 3.4. [Block- ω -Jacobi und Datenparallelität] Die Vektoren und Matrizen werden blockweise aufgeteilt. Hierbei gibt es mehrere Möglichkeiten der Datenaufteilung (siehe Abschnitt 6).

Definition 3.7. [Geometrische Parallelität] Hier erfolgt die Aufspaltung der Daten in Teilmengen entsprechend geometrischer Überlegungen über die räumlichen Anordnung der Objekte (Teilchen, Diskretisierungspunkte etc.), bzw. unter Ausnutzung von Nachbarschaftsbeziehungen bei Diskretisierungsmethoden wie FEM, FDM, FVM.

In unseren Untersuchungen werden wir uns größtenteils auf eine Datenaufteilung stützen, welche auf geometrischer Parallelität basiert.

3.2 Synchronisation paralleler Prozesse

3.2.1 Einige Begriffe

Bemerkung 3.2. Sequentielle Programmierung ist nur Ausdruck unserer Unfähigkeit, die natürliche Parallelität nahezu aller Vorgänge einer "Maschine" klarzumachen.

Probleme treten bei der Verwaltung paralleler Prozesse auf:

- völlig unabhängige Prozesse ("nur gemeinsame Stromversorgung")
 \implies absolut parallel (\implies für uns uninteressant).
- gemeinsame Nutzung (knapper) Ressourcen
 \implies Multiprogrammbetrieb/Time-Sharing
 \implies Verteilungsstrategie notwendig.
- Mehrprozessorsysteme mit verteilten Aufgaben einer Gesamtaufgabe
 \implies Informationsaustausch erfolgt über **shared memory** oder **message passing** (distributed memory).

Definition 3.8. [Undefinierter Zustand] Das Ergebnis einer Programmabschnittes ist von der Abarbeitungsgeschwindigkeit der beteiligten Prozesse abhängig und damit nicht eindeutig vorhersagbar.

Definition 3.9. [Synchronisation] Verhinderung undefinierter Zustände durch "Abstimmung" der Prozesse untereinander zu bestimmten kritischen Zeitpunkten.

Definition 3.10. [Barrier] Punkt im Programm, den alle Prozesse durchlaufen müssen. Garantiert, daß die einzelnen Prozesse solange warten bis alle Prozesse in ihrer Programmabarbeitung diesen Punkt erreicht haben.

Beispiel 3.5. [Undefinierter Zustand] Ein undefinierter Zustand kann, z.B., beim Zugriff auf gemeinsamen Speicher auftreten.

Prozeß A: $N := N + 1$	Zeitpkt.	Prozeß B: $N := N - 1$
LOAD N	(1)	LOAD N
INC N	(2)	DEC N
STORE N	(3)	STORE N

Fig. 3.4 Zwei Prozesse greifen auf die gleiche Variable zu

Das Resultat (d.h. der Wert von N) hängt von der Geschwindigkeit der Prozesse A und B in Fig. 3.4 ab und ist somit undefiniert, wie in Fig. 3.5 zu sehen ist.

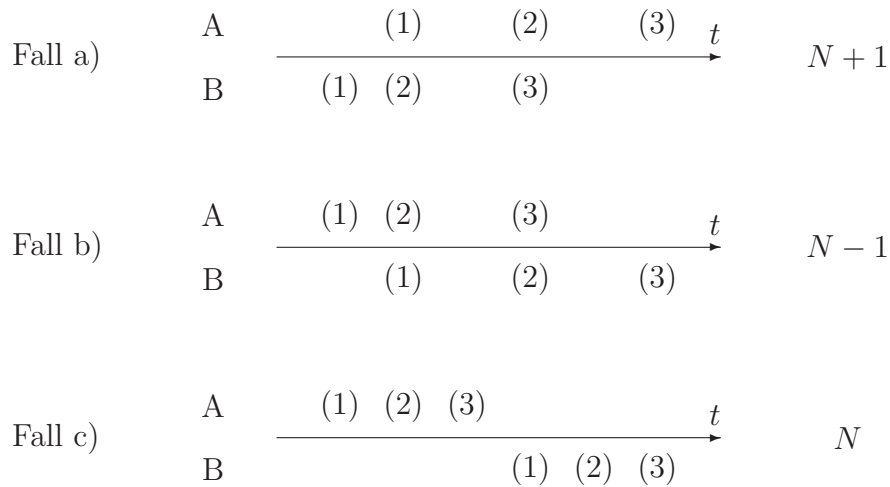


Fig. 3.5 Ergebnisse auf A und B im undefinierten Zustand

Um obigen, undefinierten Zustand zu vermeiden wäre es sinnvoll, die Operationen A und B als nicht teilbar zu betrachten.

\implies Exklusiver Zugriff auf N für einen Prozeß für die erforderliche Zeit.

Ein Konzept zur Realisierung dieses exklusiven Zugriffs wird im nächsten Abschnitt vorgestellt.

3.2.2 Semaphoren

Um bei gemeinsamen Ressourcen den exklusiven Zugriff eines Prozesses auf eine Variable (Speicheradresse) zu gewährleisten, muß diese Variable für alle anderen Prozesse gesperrt werden (engl.: *locked*). Hierbei kann es jedoch passieren, daß das gesamte Mehrprozessorsystem oder Teile davon im Wartestatus verharren. Dieser unerwünschte Effekt wird mit *Deadlock* bezeichnet.

Definition 3.11. [Deadlock] Verklemmung, *mehrere* Prozesse warten auf ein Ereignis, welches aber nur durch einen der wartenden Prozesse ausgelöst werden kann.

Beispiel 3.6. [Dinner for five (Dijkstras oder Hoare, 1971)] "Das Leben der Philosophen besteht aus den sich abwechselnden Tätigkeiten *Denken* und *Essen*. Jeder Philosoph hat seinen Platz am Tisch.

Ihr einziges Problem - abgesehen von denen philosophischer Natur - besteht darin, daß die aufgetragene Mahlzeit eine sehr schwierige Art von Spaghetti ist, zu deren Verzehr man *2 Gabeln benötigt*. Neben jedem Teller liegen 2 Gabeln, so daß grundsätzlich keine Schwierigkeiten auftauchen sollten. Es hat jedoch

zur Folge, daß 2 Nachbarn nicht zur selben Zeit essen können" [Rec 94]

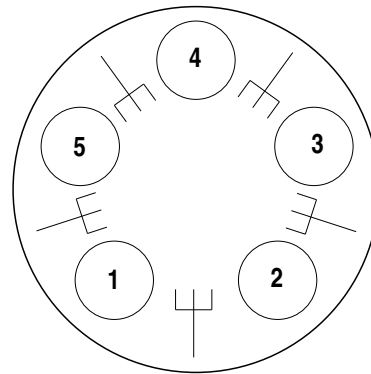


Fig. 3.6
Dinner for five

Die 5 Gabeln stehen hier stellvertretend für knappe Ressourcen in einem großen System.

Lösungsvariante (Handlungsanweisung) 1:

1. Wenn Du hungrig bist, setze Dich an an Deinen Platz.
2. Warte bis links eine Gabel frei ist und ergreife sie.
3. Warte bis rechts eine Gabel frei ist und ergreife sie.
4. Gib nach dem Essen beide Gabeln frei !

⇒ **Deadlock :**

Alle Philosophen verhungern mit einer Gabel in ihrer linken Hand.

Lösungsvariante 2:

1. Wenn Du hungrig bist, setze Dich an an Deinen Platz.
2. Warte bis links und rechts die Gabeln frei sind und ergreife sie.
3. Esse Deine Spaghetti.
4. Gib nach dem Essen beide Gabeln frei !

⇒ "Deadlock für alle" ist ausgeschlossen, aber dem Einzelnen kann der Hungertod drohen (z.B. für Philosoph Nr.1, wenn sich Nr.5 und Nr.2 leicht überlappend beim Essen abwechseln).

Dijkstras Vorschlag : Einführung von Semaphoren (Semaphor ist einem Eisenbahnsignal vergleichbar).

Definition 3.12. [Semaphor] Signal welches von einzelnen Prozessen betätigt wird und nicht von einer Zentralinstanz (Fig. 3.7).

Der Prozeß [Zug] wartet, wenn das Signal auf "WAIT" [Rot] steht. Ansonsten ist der Zugriff [Einfahrt] auf den kritischen Bereich [Fahrtstrecke] frei und

der Prozeß [Zug] stellt das Signal auf "WAIT" [Rot] für alle anderen Prozesse [Züge].

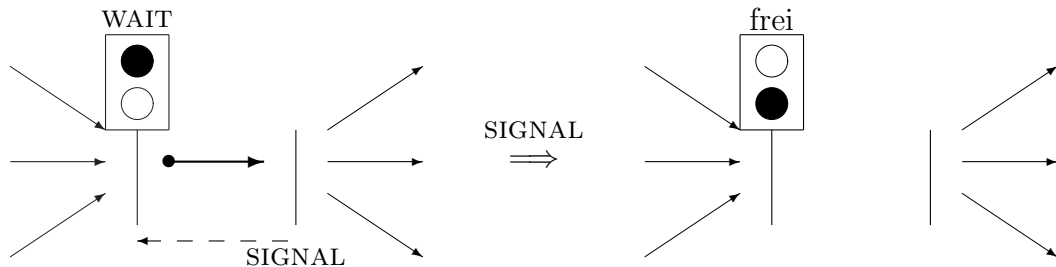


Fig. 3.7 Freigabe einer Semaphore

- Shared Memory \longrightarrow Semaphorenkonzept
- Distributed Memory \longrightarrow Message-Passing Modell \longrightarrow Kommunikation

Semaphorenkonzept

Die Semaphore kann als strukturierter Datentyp aufgefaßt werden mit den Elementen:

- S-Wert (Integer)
- Warteschlange (queue)
- \mathbb{P} (Proberen \equiv WAIT)
- \mathbb{V} (Verhogen \equiv SIGNAL)

Die Operationen \mathbb{P} und \mathbb{V} sind atomar.

Die Operationen sind mit den Elementen wie folgt verknüpft :

- $\mathbb{P}(s)$: S-Wert = 0 \longrightarrow Prozeß in Queue [Zug auf Strecke]
 S-Wert > 0 \longrightarrow (S-Wert) := 0 und Ressourcen ist verfügbar [Strecke ist frei, Einfahren und Signal auf "Rot" setzen]
- $\mathbb{V}(s)$ queue ist leer \longrightarrow (S-Wert) := 1 [Signal auf "Grün" setzen]
 queue nichtleer \longrightarrow Aktiviere einen (wartenden) Prozeß [Wartenden Zug einfahren lassen, Signal bleibt "Rot"]

Die Abfrage $\mathbb{P}(s)$ geschieht nur *einmal* pro Prozeß, daher ist die queue notwendig

Beispiel 3.7. [Semaphoren-Anwendungen]

- exklusiver Zugriff (z.B. $N = N + 1$, $N = N - 1$)

\mathbb{S} Semaphore \longleftarrow INIT(\mathbb{S}) ist notwendig, sonst tritt Deadlock auf.

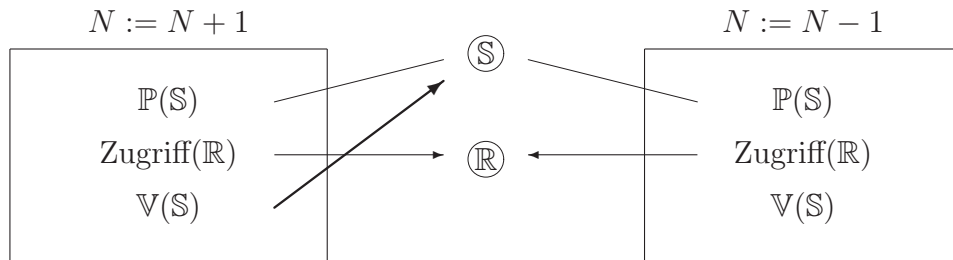


Fig. 3.8 Exklusiver Zugriff mittels Semaphoren

\mathbb{R} bezeichne die Ressourcen.

Das Ergebnis ist eindeutig ($N := (N + 1) - 1$) !!

• Eigenschaftssynchronisation (z.B. Produzent-Konsument)

2 Semaphoren nötig

A \longrightarrow signalisiert vollen Puffer; INIT(A) mit 0

B \longrightarrow signalisiert leeren Puffer; INIT(A) mit 1

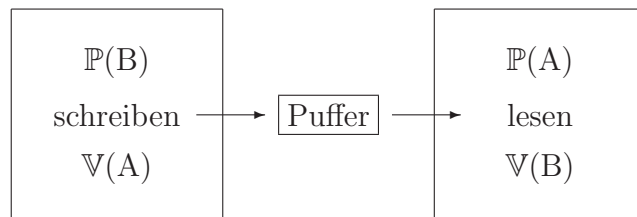


Fig. 3.9 Eigenschaftssynchronisation mittels Semaphoren

Die Eigenschaftssynchronisation tritt bei der ω -Gauß-Seidel-Iteration (vorwärts) aus Bsp. 3.2. auf. In der Iterationsvorschrift

$$\hat{c}_{i,j} := c_{i,j} + (f_{i,j} - \omega [-\hat{c}_{i-1,j} - \hat{c}_{i,j-1} + 4c_{i,j} - c_{i+1,j} - c_{i,j+1}]) / 4$$

tritt $\hat{c}_{i,j}$ als Konsument auf und alle anderen Größen sind Produzenten (auch $\hat{c}_{i-1,j}$ und $\hat{c}_{i,j-1}$!!). Siehe auch Abschnitt 3.1 .

Dijkstras Lösung des "Dinner for Five"

In the universe we assume declared

1. the semaphore `mutex` initially 1
2. the INTEGER ARRAY `C[0:4]` initially all elements 0
3. the semaphore INTEGER ARRAY `prisem[0:4]` initially all elements 0

4. *there exists a procedure*

```

procedure test (integer value k);
  if C[(k-1) mod 5] <> 2  and  C[k] == 1
                        and  C[(k+1) mod 5] <> 2
    begin C[k] := 2,  V(prisem[k])  end
end

```

Bemerkung :

$$C[k] = \begin{cases} 0 & \text{Platz ist frei} \\ 1 & \text{Platz besetzt, Philosoph hungrig} \\ 2 & \text{Platz besetzt, 2 Gabeln in Hand} \end{cases}$$

$prisem[k] == 1$ Philosoph k darf mit dem Essen anfangen.

Obiger Test besteht für den Philosophen k in den Abfragen

- Hat der linke Nachbar keine 2 Gabeln, d.h. hat er keine Gabel in der Hand? (= Er ißt nicht !)
- Sitze ich am Tisch und habe ich Hunger?
- Hat der rechte Nachbar keine 2 Gabeln?

Falls die Abfragen alle mit ja beantwortet werden können, nimmt Philosoph k die 2 Gabeln neben sich und ißt.

5. *In this the live of philosophes can be coded (philosoph w):*

```

1:   cycle begin  think
2:           P(mutex)
3:           C[w] := 1;  test(w)
4:           V(mutex)
5:           P(prisem[w]);  eat
6:           P(mutex)
7:           C[w] := 0;  test[(w+1) mod 5];
                        test[(w-1) mod 5]
8:           V(mutex)
9:   end

```

Erläuterung:

3: Setze Dich und nimm, falls möglich, 2 Gabeln.

5: Habe ich die Erlaubnis zum Essen? Wenn ja dann esse - ansonsten warte, bis die Erlaubnis erteilt wird (d.h., Semaphore $prisem[w]$ gelöscht ist.)

7: Nach dem Essen den Platz freigeben und den Nachbarn mitteilen, daß die beiden Gabeln wieder frei sind.

Nähere Erläuterungen sind in der Zeitschrift c't 12/90 auf Seite 246 zu finden. Eine sehr schöne Demonstration¹ des Diner for Five ist im Internet zu finden.

¹<http://www.javasoft.com/applets/archive/beta/DiningPhilosophers/index.html>

Message Passing

Beim Message Passing (= Kommunikation) wird zwischen blockierender und nichtblockierender Kommunikation unterschieden :

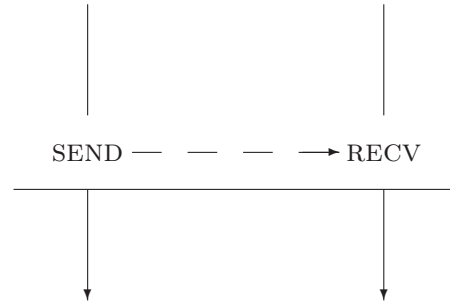


Fig. 3.10
Blockierende Kommunikation

Bei der *blockierenden Kommunikation* warten alle (hier 2) beteiligten Prozesse solange bis alle Prozesse ihre Bereitschaft zum Daten- und Nachrichtenaustausch signalisiert haben. Nebenstehende Kommunikation wirkt für die Prozesse P_1 und P_2 wie 2 Semaphoren :

- $V(P_1)$: Bereit für SEND
- $P(P_1)$: Warte, bis P_1 empfangsbereit ist.
- $V(P_2)$: Bereit für RECV
- $P(P_2)$: Warte, bis P_2 sendebereit ist.

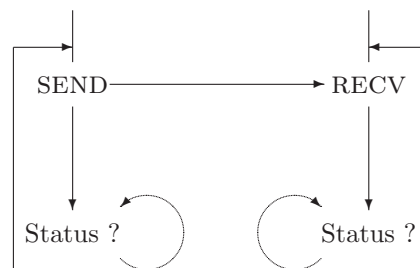


Fig. 3.11
Nichtblockierende Kommunikation

Bei der *nichtblockierenden Kommunikation* senden bzw. empfangen die Prozesse ihre Daten völlig unabhängig vom Status der anderen Prozesse.

- Die nichtblockierenden Kommunikation ist vergleichbar mit der Eigenschaftssynchronisation der Semaphoren.
- Gewährleistet hohe Effektivität in der Kommunikation ohne Verklemmungsgefahr (im Unterschied zur blockierenden Kommunikation).
- Der sendende Prozeß erwartet keine Bestätigung der erfolgreichen Operation. Ausgabeparameter der empfangenden Routine dürfen keinesfalls undefiniert sein - ansonsten kann das gesamte Programm abstürzen. Dies zu verhindern ist die Aufgabe des Programmierers!

Während das Semaphorenkonzept typisch für Parallelrechner mit gemeinsamem Speicher (Shared Memory) ist und meist vom Programmierer unbemerkt das Ressourcenmanagement erledigt, tritt das Message Passing bei Parallelrechnern mit verteiltem Speicher (Distributed Memory) auf und muß zumindest teilweise vom Programmierer selbst erledigt werden.

Monitorkonzept

Eine zentrale Instanz verwaltet die beschränkten Ressourcen (Festplatte, Bildschirmspeicher, ...).

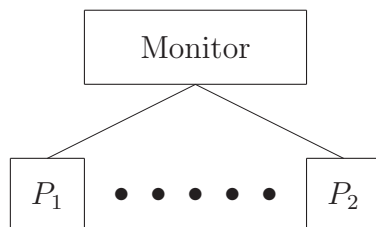


Fig. 3.12
Monitorkonzept

- Insbesondere für konkurrierende Prozesse geeignet.
- Zugriff auf die Ressourcen über spezielle Funktionen, die der Monitor bereitstellt und selbst realisiert (X11, PVM-Daemon).

Unterschiede zwischen Semaphoren und Monitorkonzept :

Bei Semaphoren "unterhalten" sich die Prozesse miteinander, während beim Monitorkonzept eine zentrale Instanz die Aufträge der Prozesse entgegennimmt.

3.2.3 Datenkohärenz

Definition 3.13. [Datenkohärenz] Alle Kopien eines Datensatzes enthalten stets die gleichen Daten.

- Bei distributed-memory Maschinen ist der Programmierer selbst für die Datenkohärenz verantwortlich, siehe Abschnitte 4 und 6.
- Bei den klassischen (alten) shared-memory Maschinen mit einem großen Speicher wurde die Datenkohärenz mittels des Semaphorenkonzepts aus Abschnitt 3.2.2 gewährleistet.
- Mittlerweile haben shared-memory Computer zumindest einen lokalen Cache, wenn nicht sogar gleich ein distributed-shared-memory (2.1.4) Konzept zugrunde liegt.

Hier kommt der *Cache Kohärenz* eine große Bedeutung zu. Zur Absicherung

dieser Kohärenz werden verschiedene Protokolle genutzt.

snoopy-based protocol: Jeder Prozessor verbreitet seinen Zugriff auf Speicherbereiche im gesamten Netzwerk. Dies führt zu Problemen mit der Kommunikationsbandbreite.

directory-based protocol: Jeder Speicherblock (cache-line) besitzt einen Eintrag in einem Verzeichnis (Firma KSR: Meer von Adressen) welcher den Zustand des Blockes und einen Bitvektor mit den Prozessoren, welche Kopien besitzen, speichert. Durch Auswertung dieser Einträge kann jederzeit bestimmt werden, welcher Cache wo aktualisiert werden muß [SGI 97].

3.3 Grundlegende globale Operationen

Die in diesem Abschnitt betrachteten Operationen auf Daten werden im Hypercube (Abschnitt 2.2.7) und den in ihm eingebetteten Topologien betrachtet.

Voraussetzung: Es seien die Routinen

$$\begin{aligned} & \text{SEND_CHAN}(nwords, data, LinkNo) \\ & \text{RECV_CHAN}(nwords, data, LinkNo) \end{aligned}$$

für das Senden und Empfangen von Daten über das Link $LinkNo$ vorhanden. Desweiteren seien die Prozesse p und q über das Link $LinkNo$ verbunden ($XOR(p, q) = 2^{LinkNo-1}$).

Die Bezeichnungen der behandelten Funktionen orientieren sich dabei an einem abstrakten Bibliothekspaket, sind aber ansonsten frei wählbar und zum Beispiel sofort mit den entsprechenden MPI- oder PVM-Rufen belegbar bzw. programmierbar.

3.3.1 EXCHANGE

Die Funktion $\text{EXCHANGE}(LinkNo, nWords, SendData, RecvData)$ tauscht Daten zwischen den Prozessoren p und q über das Link $LinkNo$.

Variante a)

- Funktioniert *nur* bei nichtblockierender Kommunikation, d.h. sendender Prozeß erwartet keine Empfangsbestätigung.
- Blockierende Kommunikation führt zu einem Deadlock.

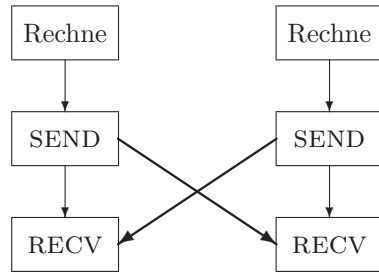


Fig. 3.13
EXCHANGE mit nicht blockierender Kommunikation

Variante b)

Ein verklemmungsfreies EXCHANGE zwischen p und q bei blockierender Kommunikation erfordert eine eindeutige Vorschrift, welcher von beiden Prozessen zuerst senden bzw. empfangen darf. Die Funktion $BTEST(k, link) = XOR(k, 2^{link-1})$ testet, ob das $link-1$ -te Bit gesetzt ist oder ob nicht. Durch die Festlegung, daß der Prozeß, dessen entsprechendes Bit gesetzt ist zuerst senden und dann empfangen darf (beim anderen umgekehrt), ist ein deadlock-freies EXCHANGE realisierbar.

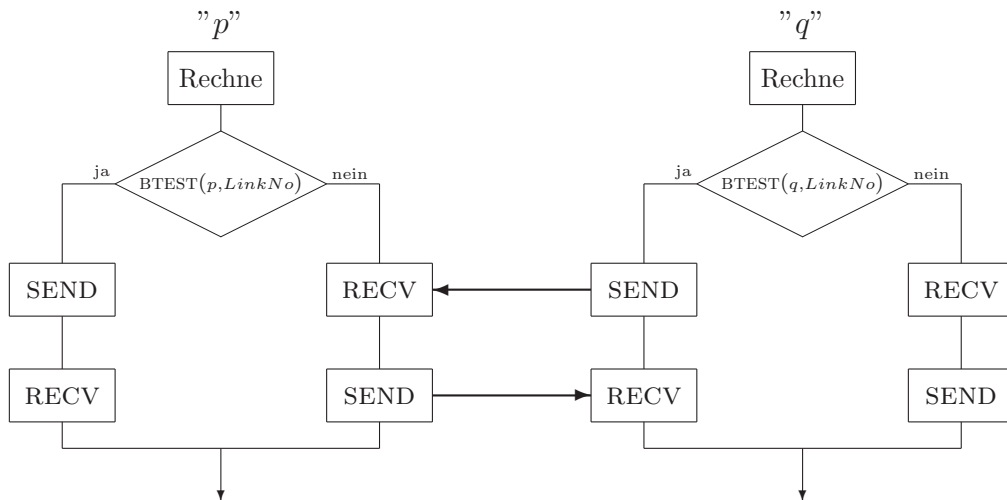


Fig. 3.14 EXCHANGE mit blockierender Kommunikation

3.3.2 Gather-Scatter-Operationen

In parallelen Algorithmen tritt immer wieder das Sammeln (engl.: gather) bzw. Verteilen (engl.: scatter) von Daten durch einen ausgezeichneten Prozeß, im weiteren *Root-Prozeß* genannt, auf.

- Sei Prozeß 0 der Rootprozeß.
- Der optimaler Baum ist im Hypercube eingebettet (Abschnitt 2.2.7)
- Scatter \implies TREE_DOWN(*nWords,Data*)
- Gather \implies TREE_UP(*nLocal,DataIn,nOut,DataOut,MaxData*)

Sei *nCube* die Dimension eines Hypercubes und *ICH* die Nummer eines Prozesses darin. Dann kann TREE_DOWN folgendermaßen realisiert werden.

Im Algorithmus 3.1 steigt die Belastung der Linkverbindungen mit ihrer Nummer.

3.3.3 Broadcast

Datenübergabe von einem Prozeß (oder allen Prozessen) an alle anderen.

- Ein Prozeß an alle anderen: Scatter mit beliebigem Root-Prozeß
- Alle Prozesse an alle anderen.

Variante a) Gather [TREE_UP] sammelt Daten aller Prozesse auf einem Prozeß und verteilt sie anschließend wieder mittels Scatter [TREE_DOWN].

Variante b) *nCube* * EXCHANGE über die Links, wobei die jeweils neuen Daten eingeordnet [CUBE_EXCH] bzw. angehängt [CUBE_CAT] werden.

```

      IF ( ICH.EQ.0 ) THEN
          L = 0
      ELSE
          L = nCube
          DO WHILE ( .NOT. BTEST(ICH,L-1) )
              L = L-1
          END DO
          CALL Recv_Chan(nwords,Data,L)
      ENDIF
      DO Link = L+1, nCube
          CALL Send_Chan(nWords,Data,Link)
      END DO

```

Alg. 3.1 TREE_DOWN(*nWords,Data*)

Aufgabe 3.2. Ändern sie den Algorithmus für TREE_DOWN so ab, daß Sie ein Broadcast von einem beliebigen Prozeß *IRoot* an alle anderen realisieren.

Hinweis: Nutzen Sie die Aufgaben 2.3. und 2.4..

3.3.4 Reduce- und All-Reduce Operationen

Die Reduce Operationen bilden Ergebnisse über Daten aller Prozesse (z.B. $s = \sum_{i=0}^{p-1} a_i$). Das Ergebnis steht zum Schluß einem Root-Prozeß (bei Reduce) bzw. allen Prozessen (bei All-Reduce) zur Verfügung.

- *Reduce* : Es wird die Tree-Topologie mit Root = 000 verwendet. In Fig. 3.15 bezeichnen a) = Link 3, b) = Link 2, c) = Link 1. Die Addition erfolgt immer linkweise parallel, hier in der Reihenfolge a), b), c).

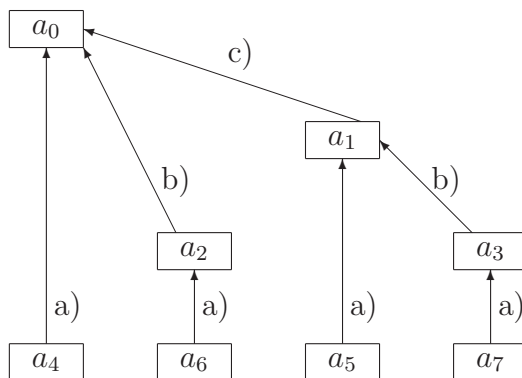


Fig. 3.15 Reduce-Operation im binären Baum

Im Detail wird die Addition wie folgt ausgeführt :

$$s_4 := a_4, s_6 := a_6, s_5 := a_5, s_7 := a_7$$

$$s_2 := (a_2 + a_6), s_3 := (a_3 + a_7)$$

$$s_1 := [(a_1 + a_5) + (a_3 + a_7)]$$

$$s_0 := [(a_0 + a_4) + (a_2 + a_6)] + [(a_1 + a_5) + (a_3 + a_7)] = \sum_{i=0}^{p-1} a_i$$

Somit besitzt am Ende nur Prozeß 000 das richtige Ergebnis, alle anderen Prozesse verfügen nur über Teilergebnisse.

Eine Programmrealisierung könnte über eine spezielle Funktion `TREE_UPDO(n,X,Y,H,VtOp)` erfolgen, welche auch mit Vektoren arbeitet und Operationen auf diesen mit `VtOp` spezifiziert (Hier mittels `CALL Tree_UpDo(1,S,A,AuxArray,VdPlus)`).

- *All-Reduce*

Variante a): Kombination von Reduce- mit einer Scatter-Operation.

$$\left. \begin{array}{l} \text{TREE_UPDO} \\ \text{TREE_DOWN} \end{array} \right\} \text{TREE_DO } \frac{I}{R}$$

Variante b): Akkumulation im Hypercube (*Cubeakkumulation*)

Bei der Cubeakkumulation tauscht `nCube`-mal die eine Hälfte der Prozesse

(nCube-1 Cube) Daten mit der anderen Hälfte aus.

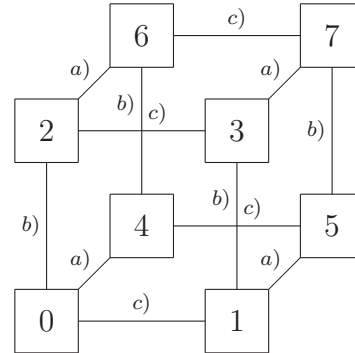


Fig. 3.16 All-Reduce im Hypercube

Die Additionen wird linkweise parallel durchgeführt, in der Abfolge: a) b) c). Bei der All-Reduce Operation im Hypercube besitzen zum Schluß alle Prozesse das Ergebnis.

$$s_0 = [(a_0 + a_4) + (a_2 + a_6)] + [(a_1 + a_5) + (a_3 + a_7)]$$

⋮

$$s_7 = [(a_0 + a_4) + (a_2 + a_6)] + [(a_1 + a_5) + (a_3 + a_7)]$$

⇓

$$s_i = \sum_{i=0}^{p-1} a_i \quad \forall i = 0, p - 1$$

Die Funktion $CUBE_DO \stackrel{I}{R}(n, X, Y, H, VtOp)$ könnte eine Programmrealisierung sein.

Tab. 3.1 Vergleich von All-Reduce in Hypercube und Baum

CUBE_DO	TREE_DO
•	Gleicher arithmetischer Aufwand
• nCube bidirektionale Kommunikationen oder 2*nCube unidirektionale Kommunikationen	2*nCube unidirektionale Kommunikationen
• Gleiche Linkbelastung	Sinkende Linkbelastung mit wachsender/fallender Linknummer
• Ideal bei gleicher Bandbreite aller Links	Auswahl des im Hypercube eingebetteten Baumes nach der Bandbreite der Links
• Ideal im physischen Hypercube, z.B. NCUBE2	Bei größeren Systemen können technologisch bedingte geringere Bandbreiten bei einigen Links auftreten, siehe Bsp. POWER-GC

Beispiel 3.8. [Links im POWER-GC] Im POWER-GC der Firma PARSYTEC (ab 64 Prozessoren lieferbar) sind folgende Relationen bzgl. der Bandbreite der

Links gegeben :

Links 1-4 : Bandbreite a

Links 5-6 : Bandbreite b

Links 7-8 : Bandbreite c

mit $a > b > c$. Da die höheren Links eine geringere Bandbreite aufweisen, sollten sie bei den TREE...-Operation im optimalen Baum nahe am Root-Prozeß benutzt werden. Dort kommunizieren nur wenige Prozesse miteinander.

Entsprechende Tests auf dem POWER-GC mit 128 Prozessoren an der TU Chemnitz (Dr.M.Pester) erbrachten ein signifikant besseres Laufzeitverhalten (\Leftarrow Kommunikationszeit) der entsprechend gewählten TREE_DO-Routinen gegenüber den äquivalenten CUBE_DO-Routinen.

Das hinderliche Problem der unterschiedlichen Bandbreiten in der CUBE_DO-Routine hängt zusammen mit der Bisektionsbandbreite

Definition 3.14. [Bisektionsbandbreite] Kommunikationsbandbreite, wenn die eine Hälfte der Prozessoren mit der anderen Hälfte kommuniziert.

In obigem Beispiel ist diese Bisektionsbandbreite anisotrop, d.h. sie hängt davon ab welche Teile der Maschine miteinander kommunizieren. In der Origin2000 tritt dieses Problem ab 32 Prozessoren bei Nutzung der *Xpress links*² auf ([SGI 97], Seite 34ff). Die erreichte Bisektionsbandbreite einer Origin2000 wächst linear bis zu zu 10 GB/sec auf 32 Prozessoren, verbleibt bei diesem Wert auf 64 Prozessoren und verdoppelt sich wieder bei 128 Prozessoren. Bei einem idealen System wächst die Bisektionsbandbreite mit der Anzahl der Prozessoren.

Aufgabe 3.3. Ändern Sie den Algorithmus für TREE_DOWN auf Seite 45 so ab, daß die Links mit höheren Nummern möglichst wenige Daten transportieren müssen.

3.3.5 Barrier

An einer Barriere erfolgt die Synchronisation aller Prozesse, d.h. alle anderen Prozesse warten, bis der letzte Prozeß diesen Programmpunkt erreicht hat.

- Jede der Operationen 3.3.1-3.3.4 wirkt als Barrier.
- Künstlicher Einbau einer Barrier, z.B. mittels CUBE_DOI(1,I,J,K,ViPlus)
- Die meisten parallelen Algorithmen enthalten ganz natürlich eine Barrier, da an irgendeinem Punkt ein globaler Datenaustausch (z.B. Akkumulation) notwendig wird.

²http://www.sgi.com/origin/2000_specs.html

3.3.6 Bemerkungen zu portablem Code

Die Operationen in 3.3.1 - 3.3.5 bauten auf den folgenden Voraussetzungen auf :

1. Vorhandene logische Topologie, hier Hypercube. Falls der Hypercube physisch nicht vorhanden ist, muß er auf der vorhandenen physischen Topologie aufgebaut werden \rightarrow TRINIT.
2. Sende über Link \rightarrow SEND_CHAN.
3. Empfange über Link \rightarrow SEND_RECV.

Nur diese 3 Routinen sind Hardware- und Betriebssystemabhängig. Alle anderen Routinen könne aus diesen 3 Grundroutinen aufgebaut werden. Eine freiwillige Beschränkung auf gewisse Basisroutinen der verfügbaren parallelen Bibliotheken unter MPI, PVM, Parix, nCube, Paragon, Standalone(3L), Helios, ... sichert einen *portablen Code*.

3.4 Leistungsbewertung paralleler Algorithmen

Wie vergleicht man fair (Was ist fair !?) die Eignung bzw. Nichteignung paralleler Algorithmen für größere oder große Prozessorzahlen ?

3.4.1 Speedup und Scaleup

Definition 3.15. [Speedup] Zeitgewinn eines parallelen Algorithmus gegenüber der seriellen Version bei konstanter globaler Problemgröße N_{ges} .

Sei t_1 - Ausführungszeit des Algorithmus A auf einem Prozessor
 t_P - Ausführungszeit des Algorithmus A auf P Prozessoren,
dann berechnet sich der Speedup zu

$$\boxed{S_P = \frac{t_1}{t_P}} . \quad (3.1)$$

Wünschenswert ist ein Speedup von P , d.h. P Prozessoren sollten P -mal schneller sein als ein einzelner Prozessor. Es gilt jedoch

$$N_{ges} = \text{const.} \implies N_i \sim \frac{1}{P} \quad \forall i = \overline{1, P} ,$$

d.h., je mehr Prozessoren verwendet werden, desto weniger hat der einzelne Prozessor zu tun.

Beispiel 3.9. [Speedup-Arbeiter] $P = 1$ Arbeiter hebt in $t_1 = 1h$ eine Grube von $1 m^3 [N_{ges}]$ aus. Wie lange $[t_P]$ benötigen $P = 1000$ Arbeiter um dieselbe Arbeit zu verrichten ?

→ Die Arbeiter behindern sich gegenseitig, ein zu hoher Organisationsaufwand [Overhead] ist notwendig.

⇒ sehr geringer Speedup.

Satz 3.1. [Amdahlsches Gesetz]

Jeder Algorithmus besteht aus Teilen, welche sich nicht parallelisieren lassen.

Seien s - serieller Anteil des Algorithmus

p - paralleler Anteil des Algorithmus normiert :

$$\boxed{s + p = 1} \quad (3.2)$$

Somit läßt sich die Rechenzeit auf einem Prozessor durch $t_1 \rightarrow s + p$ und (bei idealem Verhalten) die Rechenzeit auf P Prozessoren durch $t_P \rightarrow s + \frac{p}{P}$ ersetzen.

Daraus folgt eine obere Schranke für den maximal erreichbaren Speedup.

$$\boxed{S_{P,max} = \frac{s + p}{s + \frac{p}{P}} = \frac{1}{s + \frac{1-s}{P}} \leq \frac{1}{s}} \quad (3.3)$$

Bemerkung 3.3. Obiger Satz geht auf eine Bemerkung von Gene Amdahl³ im Jahre 1967 zurück. In einer (sehr) alten Reklame von IBM wurde das AMDAHLsche Gesetz bei 1% seriellen Anteils im Algorithmus angewendet:

P	10	100	1000	10000	$\Rightarrow S_{\infty,max} = 100$
S_P	9	50	91	99	

Mehr als 100 Prozessoren erscheinen sinnlos (d.h., daß von 100 auf 1000 Arbeiter zwar der 10-fache Lohn bei nicht einmal verdoppelter Leistung gezahlt wird).

⇒ Entmutigend für die Parallelisierung

⇒ Lohnt es sich weiter zu lesen ?

Ist das AMDAHLsche Gesetz für uns eigentlich relevant ?

- Bei kleineren Prozessorzahlen (< 20) ist bei bestimmten Algorithmen sogar ein Speedup $> P$ erreichbar ! Dies kann z.B. durch das verbesserte Cacheverhalten der Prozessoren mit kleiner werdender Teilproblemgröße oder durch die

³<http://www.amdahl.com>

Ordnung des arithmetischen Aufwands des lokalen Algorithmus mit weniger Daten auftreten.

- Da das AMDAHLsche Gesetz keine Kommunikationszeiten berücksichtigt, sieht das reale Verhalten des Speedup bei zu kleinen Problemgrößen teilweise noch schlechter aus, siehe Fig. 3.17.

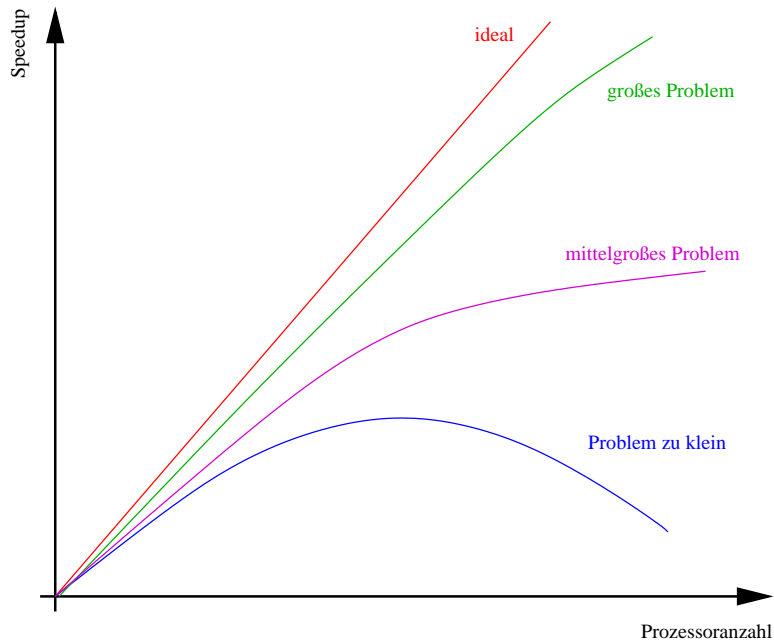


Fig. 3.17 Speedup in Abhängigkeit von der Problemgröße

Bei einer hinreichend großen Problemgröße wird ein akzeptabler Speedup erreicht, da die einzelnen Prozesse nach wie vor wesentlich mehr Arithmetik als Kommunikation ausführen.

Folgerung : Der erreichbare Speedup hängt von der Problemgröße ab. Gleichzeitig erscheint es sinnvoll, auf mehreren Prozessoren auch größerer Gesamtprobleme zu lösen.

Beispiel 3.10. [Scaleup-Arbeiter] Wenn jeder Arbeiter eine Grube von $1 m^3$ aushebt, so schaffen 1000 Arbeiter in einer Stunde fast die 1000-fache Leistung da sie sinnvoll eingesetzt sind.

Ein Skalierung der Problemgröße mit der Prozessoranzahl führt uns aus dem Dilemma des AMDAHLsche Gesetzes.

Definition 3.16. [Scaleup] Erzielter Zuwachs an Problemgröße auf dem Parallelrechner im Vergleich zum sequentiellen Fall bei vergleichbarer Rechenzeit.

Definition 3.17. [Scaled Speedup] Unter der Annahme, daß der parallele Teil des Algorithmus optimal bzgl. der Problemgröße ist (d.h. $\mathcal{O}(N)$) gilt mit den Vereinbarungen

$s_1 + p_1$ - Ausführungszeit auf dem Parallelrechner
 $s_1 + P \cdot p_1$ - Ausführungszeit auf dem seriellen Rechner

$$S_C(P) = \frac{s_1 + P \cdot p_1}{s_1 + p_1} \stackrel{s_1+p_1=1}{=} s_1 + P(1 - s_1) \quad (3.4)$$

Interpretation : 1% serieller Anteil $\implies S_C(P) \approx P$
 da der serielle Anteil mit der Problemgröße abnimmt.
 Diese theoretische Voraussage wird in der Praxis bestätigt.

Bemerkung 3.4. [Speedup/Scaleup]

- Der (Scaled) Speedup darf nicht das einzige Bewertungskriterium sein, z.B. ist in Fig. 3.18 ersichtlich, daß

$$S_P(\text{Gauß Elimination}) > S_P(\text{pcg=vorkonditionierter cg})$$

ABER $t(\text{Gauß Elimination}) \gg t(\text{pcg}) !!$

Bei dieser Untersuchung von M. Pester wurde ein vollbesetztes Gleichungssystem aus der BEM mit verschiedenen parallelen Verfahren gelöst.

- Fairerweise müßten bei der Bewertung auch die Kosten der Rechner in Betracht gezogen werden, z.B. die Kosten pro berechnete Unbekannte.

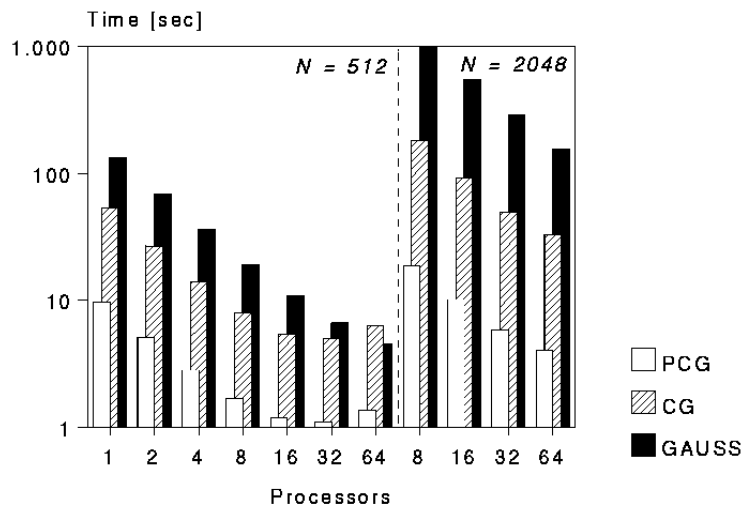


Fig. 3.18 Rechenzeitvergleich Gauß, cg, pcg [M. Pester, Chemnitz]

3.4.2 Effizienz

Definition 3.18. [Parallele Effizienz] Die parallele Effizienz spiegelt wieder, wie gut ein Algorithmus parallelisiert wurde.

Hier sei nur die Formel für die klassische (parallele) Effizienz angegeben, die skalierte Effizienz berechnet sich analog.

$$\boxed{E_{P,par} = \frac{S_P}{P}} \quad (3.5)$$

Eine Effizienz von 100% wäre ideal, d.h. die Verwendung von P Prozessoren beschleunigt die Programmabarbeitung auf das P -fache. Die parallele Effizienz für den Scaleup läßt sich explizit angeben :

$$E_{C,par} = \frac{S_C(P)}{P} = \frac{s_1 + P(1 - s_1)}{P} = 1 - s_1 + \frac{s_1}{P} > 1 - s_1$$

⇒ Ganz leicht sinkende Effizienz bei wachsender Prozessorzahl.

⇒ Effizienz jedoch mindestens so hoch wie paralleler Anteil im Programm.

⇒ Parallelisierung hat doch einen Sinn.

Definition 3.19. [Numerische Effizienz] Setzt den schnellsten seriellen Algorithmus mit dem schnellsten parallelen Algorithmus (ausgeführt auf einem Prozessor, falls möglich) ins Verhältnis.

$$\boxed{E_{num} = \frac{t_{parallel}}{t_{seriell}}} \quad (3.6)$$

Somit ergibt sich die (skalierte) Effizienz eines parallelen Algorithmus

$$\boxed{E = E_{par} \cdot E_{num}} \quad (3.7)$$

Obige Scaled Speedup- und Effizienzaussagen sind sehr optimistisch.

Aber : In Formel (3.4) ist stillschweigend eine Gleichverteilung der Gesamtaufgabe auf alle Prozessoren vorausgesetzt. Dies ist in der Praxis a priori oder im Laufe der Rechnung oft nicht der Fall. Gleichzeitig wird in Formel (3.4) keinerlei Verlust durch die Kommunikation während der Programmabarbeitung berücksichtigt.

⇒ Praktische Effizienz liegt unter der theoretischen.

Definition 3.20. [Loadbalancing] Versuch der gleichmäßigen Auslastung der Prozessoren.

Möglichkeiten dem idealen Loadbalancing nahezukommen :

- Statisches Loadbalancing [a priori]:
 - Aufteilung der Netze, Daten nach Abzählung ist zu ineffektiv und nutzt keinerlei, wie auch immer geartete, Nachbarschaftsbeziehungen aus.
 - Aufteilung der Daten nach gewissen Bisektionstechniken (z.B. Koordinatenbisektion, rekursive spektrale Bisektion, Kerningham-Lin-Algorithmus [TK 96].
 - Je besser die Datenverteilung (auch in Bezug auf die Kommunikation), desto mehr Aufwand muß bei den einzelnen Bisektionstechniken betrieben werden.
 - Besonders geeignet für Optimierungsprobleme, Verteilung der Grobnetzes in Mehrgitteralgorithmen.
 - Die Daten werden gewichtet verteilt, wobei die Gewichte durch eine vorausschauende Abschätzung des Rechenaufwandes, der durch künftige, adaptive Netzverfeinerungen entstehen wird, geschätzt werden. Diese Strategie kann für viele technischen Anwendungen eingesetzt werden.
- Dynamisches Loadbalancing (Umverteilung während der Rechnung) [Bas 96]:
 - Hoher Aufwand für Lastverteilung (oder heuristische Vorgehensweise).
 - Bei hoch adaptiven Verfahren notwendig.
- Neue parallele Betriebssysteme ??

3.4.3 Kommunikationsaufwand

Die in den beiden vorangegangenen Abschnitten dargestellten Speedup- und Effizienzbetrachtungen vernachlässigen die zur Datenkommunikation benötigte Zeit.

Sei t_{Startup} - Zeit zur Initialisierung der Kommunikation (Latency)

t_{Word} - Zeit für die Übertragung eines Wortes (bandbreite)

so ergibt sich die Zeit für eine Kommunikation mit N Byte Information :

$$\boxed{t_k = t_{\text{Startup}} + N \cdot t_{\text{Word}}} \quad (3.8)$$

Nach Formel (3.8) und praktischer Erfahrung ist das Senden *einer* langen Nachricht besser als das Senden mehrerer, entsprechend kurzer, Nachrichten. Betriebssystem oder Hardware teilen große Datenpakete in kleinere auf. Die konkrete Größe hängt vom Hersteller ab ($> 1k\text{Byte}$). Tabelle 3.2 gibt einige Werte für Parallelrechner der letzten 10 Jahre an. Der Parallelrechner

Tab. 3.2 Latency und Bandbreite (Prozessor-Prozessor) verschiedener Parallelrechner

Maschine/OS	Latency	Bandbreite
Xplorer/Parix	100 μ s	(gemessen) 1.3 MB/sec (peak) 8.8 MB/sec
Intel iPsc/860	82.5 μ s	2.5 MB/sec
Meiko, CS-2	10 μ s	45 MB/sec
T9000	10 μ s	(progn.) 80 MB/sec
Convex SPP-1	0.2 μ s	16 MB/sec
nCube2		71 MB/sec
Cray TT3D	6 μ s	120 MB/sec
Origin2000		(real) 680 MB/sec
Parnass ₂		70-850 MB/sec

Parnass₂⁴ an der Uni Bonn steht hier stellvertretend für einen low-budget Parallelrechner auf LINUX-Basis.

Es gibt Algorithmen die nicht unbedingt parallelisierungswürdig sind, obwohl sie so aussehen.

Beispiel 3.11. [FFT-feinkörnig]

Berechnung der FFT-Koeffizienten $e^{2\pi i \frac{k}{N}}$ $k=0, \overline{N-1}$

Sei t_x die Zeit zur Berechnung eines Koeffizienten, dann ist
seriell : $t_{\text{seriell}} = N \cdot t_x$ für einen Vektor der Länge N .

parallel : a) Lokal einen Teilvektor der Länge $\frac{N}{P}$ ausrechnen $\frac{N}{P} \cdot t_x$.

b) Datenaustausch (alle benötigen das Ergebnis)

$$\mathcal{O}(N) \cdot t_{\text{Word}} + \underbrace{\mathcal{O}(\log P)}_{\text{bester Fall}} \cdot t_{\text{Startup}}$$

$$t_{\text{par}} = \frac{N}{P} \cdot t_x + \mathcal{O}(N) \cdot t_{\text{Word}} + \mathcal{O}(\log P) \cdot t_{\text{Startup}}$$

Frage : Wann ist $t_{\text{seriell}} > t_{\text{par}}$?

→ Nur wenn t_x groß genug ist, d.h. bei kleinen t_x lohnt sich auch bei sehr guter Latency und Bandbreite die Parallelisierung nicht.

→ Verhältnis $\frac{\text{Arithmetik}}{\text{Kommunikation}}$ sollte möglichst groß sein.

→ Grobkörnige Algorithmen werden bevorzugt (siehe Abschnitt 4.5).

⁴<http://www.wissrech.iam.uni-bonn.de/research/projects/parnass2/>

4 Vektorisierung und Parallelisierung direkter Verfahren

4.1 Die BLAS-Bibliotheken

Häufig verwendete Routinen bzgl. Vektor- und Matrixoperationen sind in den **Basic Linear Algebra Subroutines** zusammengefaßt. Das besondere an diesen Bibliotheken ist, daß sie für die jeweiligen Hardwareplattformen hochoptimiert sind, und zum Großteil in Assembler programmiert wurden. Mittlerweile werden 3 solcher Bibliotheken unterschieden. Im folgenden gehen wir auf einige Rufe darin ein und streichen Besonderheiten bzgl. der Vektor- und Parallelrechner heraus. Üblicherweise existieren die Routinen für **Single**, **Double** und **Complex** Datenformate.

4.1.1 Vektor-Vektor Operationen (BLAS1)

Folgende Operationen mit den Vektoren \underline{x} , \underline{y} und dem Skalar α sind in BLAS1 enthalten :

1. Komponentenweise
Addition $x_i := x_i + y_i$
Subtraktion $x_i := x_i - y_i$
Multiplikation $x_i := x_i * y_i ; x_i := \alpha * x_i$
Division $x_i := x_i / y_i$
Kopieren $x_i \leftarrow y_i ; x_i \leftrightarrow y_i$
2. Skalarprodukte SDOT, DDOT $s = \sum_{i=1}^N x_i * y_i.$
3. Triaden : $x_i := x_i + \alpha * y_i$ SAXPY/DAXPY
Scalar Alpha times **X** Plus **Y**
bzw. $x_i := x_i * (\alpha + y_i)$

Parallelrechner : Zerlegung der Vektoren in Blöcke. Skalarprodukt benötigt mindestens eine REDUCE Operation (Abschnitt 3.3.4).

Vektorrechner : Außer Skalarprodukt sind alles natürliche Vektoroperationen.

Berechnung des Skalarproduktes auf Vektorrechnern

- *sequentiell* :

```

s := 0
DO i = 1, n
    s := s +  $\underbrace{x_i * y_i}_{\text{vektorisierbar}}$ 
END DO

```

Da obige Summation aber nicht vektorisierbar ist, ist auch der Gesamtalgorithmus nicht vektorisierbar.

- *vektoriert/sequentiell* :

```

DO i = 1, n
    d_i := x_i * y_i
END DO
s := 0
DO i = 1, n
    s := s + d_i
END DO

```

Hier ist die erste Schleife vektorisiert, jedoch die zweite noch nicht. Um auch die zweite Schleife zu vektorisieren, muß man mit der Technik des rekursiven Doppeln arbeiten.

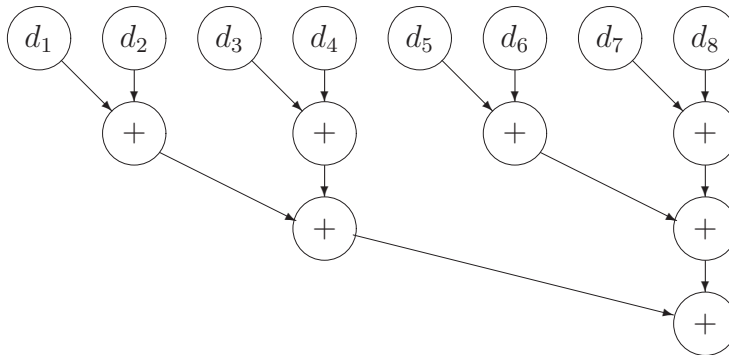
- *vektoriert/ rekursives Doppeln* : Sei $n = 2^d$, ansonsten ist der folgende Algorithmus nur etwas schwerer aufzuschreiben.

```

DO i = 1, n
    d_i := x_i * y_i
END DO
DO l = 1, log2(n)
    is := 2l
    DO i = is, n, is
        d_i := d_i + d_{i-is/2}
    END DO
END DO
s := d_n

```

Das rekursive Doppeln in der zweiten Schleife, auch Kaskadenalgorithmus bzw. zyklische Reduktion genannt, arbeitet zwar mit immer kürzeren Vektoren, nutzt jedoch die Möglichkeiten des Vektorrechners zumindest teilweise aus. In Fig. 4.1 ist deutlich die Äquivalenz zwischen dem rekursiven Doppeln und der REDUCE Operation im binären Baum (Abschnitt 3.3.4) erkennbar. Wir werden dieser zyklischen Reduktion auch bei der Parallelisierung immer wieder begegnen.

Fig. 4.1 Illustration rekursives Doppeln $n = 8$

Berechnung des Skalarproduktes auf einem Parallelrechner mit verteiltem Speicher

Seien die Vektoren \underline{x} , \underline{y} disjunkt in jeweils P Teilvektoren der Längen N_j ($j=\overline{1,P}$) geteilt und auf dem entsprechenden Prozessor P_j verfügbar.

DO IN PARALLEL $j = 1, P$

$s_j := 0$

DO $i = 1, N_j$

$s_j := x_i * y_j$

END DO

END DO

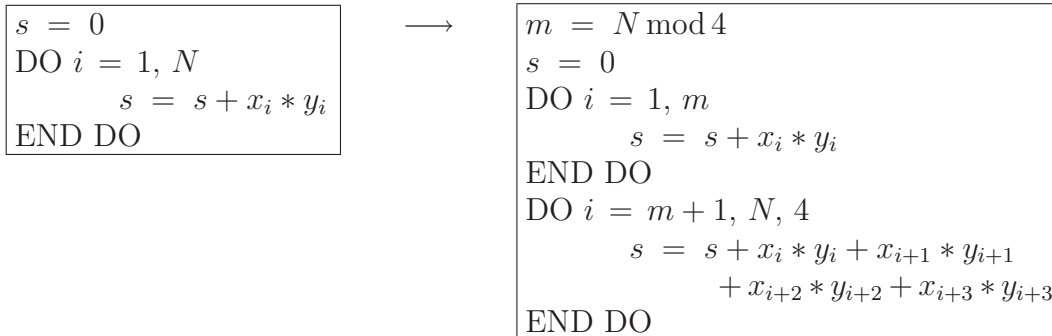
CALL ALL_REDUCE(s_j) $\longrightarrow s$, d.h. $s = \sum_{j=1}^P s_j$

Aufgabe 4.1. Programmieren Sie das globale Skalarprodukt zweier disjunkt verteilter Vektoren auf einem Parallelrechner mit verteiltem Speicher.

Eigene Vektorroutinen

Die Herangehensweise bei der Implementation von eigenen Vektorfunktionen sollte sich an den BLAS-Routinen orientieren. Hierfür ergeben sich zwei Ansätze:

1. Nutzung Loop unrolling in einer Hochsprache wie C oder F77, dh. das Verhältnis Arithmetik zu Schleifenverwaltung wird verbessert. Wir betrachten das Skalarprodukt mit Schrittweite (engl.: stride) 1.



Die Wahl des Moduloparameters (hier 4) ist hardwareabhängig. Mittlerweile sind die optimierenden Compiler so gut, daß bei obigem einfachen Beispiel kein manuelles loop unrolling notwendig ist.

2. Wie 1. mit zusätzlicher Benutzung der BLAS1-Routinen wo dies möglich ist (und die vorhandene BLAS-Bibliothek fehlerfrei ist), z.B. $\text{VDPLUS}(n, X, ix, Y, iy, Z, iz)$, d.h. $\underline{x} = \underline{y} + \underline{z}$ kann unter C bei identischen Übergabevektoren die DAXPY Operation benutzen.

```
IF (adr(X)==adr(Y) AND ix == iy)
    THEN CALL DAXPY(n, 1d0, X, ix, Z, iz)
    ELSE IF (adr(X)==adr(Z) AND ix == iz)
        THEN CALL DAXPY(n, 1d0, X, ix, Z, iz)
        ELSE Loop unrolling wie in 1. Realisierungsebene
    END IF
END IF
```

4.1.2 Matrix-Vektor Operationen (BLAS2)

In diesem Abschnitt werden nur vollbesetzte Matrizen, bzw. Matrizen mit Bandstruktur betrachtet.

Arten der Matrixspeicherung bei vollbesetzter Matrix A .

- i) zeilenweise (row storage) [C,Pascal]
- ii) spaltenweise (column storage) [F77]
- iii) $A = A^T$: zeilenweise oberes Dreieck A_U
- iv) $A = A^T$: spaltenweise unteres Dreieck A_L

Aufgabe 4.2. Stellen Sie die Operation $\underline{v} = A_{n \times n} \underline{x}$ mittels BLAS-Routinen (DDOT, DAXPY) für die Speicherformen i)-iii) dar.

Zusatzaufgabe: Fall ii) ohne BLAS aber mit loop unrolling (Stride 2).

Für die tridiagonale Matrix

$$A = \begin{pmatrix} b_1 & c_1 & & & & \\ a_1 & b_2 & c_2 & & & \\ & \ddots & \ddots & \ddots & & \\ & & & & c_{n-1} & \\ & & & a_{n-1} & b_n & \end{pmatrix}$$

betrachten wir 2 Varianten der Matrix-Vektor Multiplikation $\underline{v} = A_{n \times n} \underline{x}$ auf dem Vektorrechner.

Variante a) Die Matrix wird in den Vektoren $a(1 : n - 1)$, $b(1 : n)$, $c(1 : n - 1)$ gespeichert.

```

v1 = b1 * x1 + c1 * x2
vn = bn * xn + an-1 * xn-1
DO i = 2, n - 1
    yi = ai-1 * xi-1 + bi * xi + ci * xi+1
END DO

```

Variante b) Gegenüber Variante a) werden die Vektoren a, c, x verlängert : $a(0 : n - 1)$, $b(1 : n)$, $c(1 : n)$, $x(0 : n + 1)$.

```

x0 = xn+1 = 0
a0 = cn = 0
DO i = 1, n
    yi = ai-1 * xi-1 + bi * xi + ci * xi+1
END DO

```

In Variante a) müssen die ersten beiden Zeilen seriell abgearbeitet werden, was bei Vektorrechnern eine Geschwindigkeitseinbuße um das 5- 15-fache bedeutet. Daher ist Variante b) auf dem Vektorrechner schneller, obwohl mehr arithmetische Operationen ausgeführt werden müssen.

Parallelrechner mit verteiltem Speicher

Betrachten $\underline{v} = A_{n \times n} \underline{x}$ mit vollbesetzter Matrix A . Je nach Aufteilung der Matrix unterscheidet sich die programmtechnische Realisierung der Multiplikation auf dem Parallelrechner. Zwei Varianten werden betrachtet.

Variante 1 : Die Matrix A wird blockzeilenweise auf die Prozesse verteilt, analog die Teilvektoren.

Variante 1b : Durch einen ALL_TO_ALL_SCATTER-Ruf besitzt jeder Prozeß den gesamten Vektor \underline{x} . Danach läßt sich die Multiplikation ohne weitere Kommunikation ausführen.

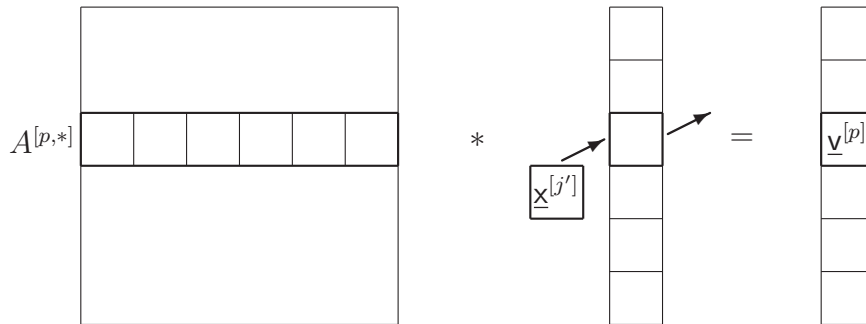


Fig. 4.2 Blockzeilenweise verteilte Matrix

Auf jedem Prozeß p laufen die folgenden Schritte für $j = \overline{1, P}$ ab :

- Berechne $\underline{v}^{[p]} = \underline{v}^{[p]} + A^{[p,j]} \cdot \underline{x}^{[j]}$.
- Sende j und $\underline{x}^{[j]}$ an den vorwärtigen Prozeß im Ring.
- Empfange j' und $\underline{x}^{[j']}$ vom rückwärtigen Prozeß im Ring.

Zum Schluß besitzt jeder Prozeß p den Teilvektor $\underline{v}^{[p]} = A^{[p]} \cdot \underline{x}$ und "seinen" Teilvektor $\underline{x}^{[p]}$.

Alg. 4.1 Parallele Matrix-Vektor Multiplikation, blockzeilenweise

Variante 2 : Wir verteilen A blockspaltenweise auf die Prozesse, Vektor \underline{x} wird entsprechend der Spalten von A verteilt.

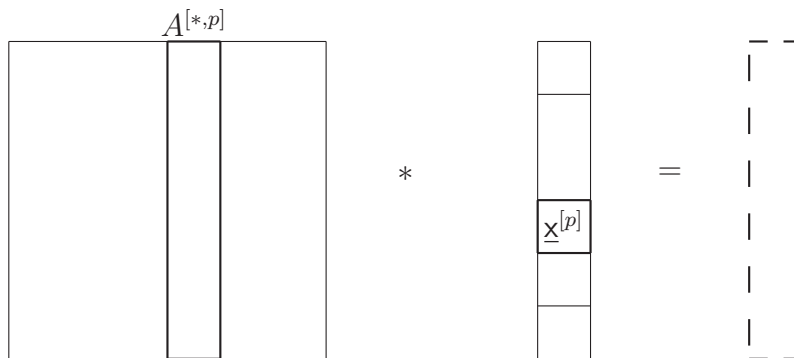


Fig. 4.3 Matrix blockspaltenweise verteilt

Ein weiterer Matrix-Vektor Algorithmus funktioniert analog dem Broadcast-Multiply-Roll Algorithmus (Alg. 4.4) im nächsten Abschnitt.

Auf jedem Prozeß p laufen die folgenden Schritte ab :

- Berechne $\underline{v} = A^{[*p]} \cdot \underline{x}^{[p]}$.
 - REDUCE_ALL(\underline{v}) zur globalen Akkumulation des Vektors \underline{v} .
- Zum Schluß besitzt jeder Prozeß p den gesamten Vektor $\underline{v}^{[p]}$.

Alg. 4.2 Parallele Matrix-Vektor Multiplikation, blockspaltenweise

4.1.3 Matrix-Matrix-Operationen (BLAS3)

Seien A, B, C vollbesetzte Matrizen.

Typische Operationen sind : • $C := A + B$ $C := A - B$

• $C := A * B$

• $C := C + A * B$

• $A^T, A_{N \times N}^{-1}$

• Faktorisierungen, Givensrotationen, ...

Matrix-Multiplikation

Die Multiplikation zweier kompatibler Matrizen wird formal geschrieben als :

$$C_{N \times N} := A_{N \times M} * B_{M \times N} \quad c_{ij} := \sum_{k=1}^M a_{ik} \cdot b_{kj} \quad i,j=\overline{1,N}$$

Zur Berechnung der Matrix C ist demnach eine dreifach Indexschleife erforderlich.

```

DO ..... := 1, .....
  DO ..... := 1, .....
    DO ..... := 1, .....
       $c_{i,j} := c_{i,j} + a_{i,k} \cdot b_{k,j}$ 
    END DO
  END DO
END DO

```

Alg. 4.3 Grundform der Matrix-Matrix Multiplikation

Die 6 verschiedene Algorithmen zur Matrixmultiplikation unterscheiden sich bzgl.

- Zeilen- bzw. Spaltenzugriff auf die Matrixelemente,

- Nutzbare Grundroutinen, z.B. Skalarprodukt, Triade.

Dadurch ist die konkrete Implementierung stark hardwareabhängig:

- Nach jedem Lese/Schreibzugriff auf ein Element einer Speicherbank wird diese Bank für eine gewisse Zeit für weitere Zugriffe gesperrt. Benötigt ein weiterer Prozeß Elemente dieser Bank, so muß er warten, bis die Bank wieder freigegeben ist. Dieser unerwünschte Effekt wird als *Speicherbankkonflikt* bezeichnet.

Wir betrachten als Beispiel das Laden eines Vektors \underline{a} (Matrixzeile / -spalte). Die Sperrzeit der Speicherbanken betrage 4 Lesezyklen.



Fig. 4.4 Speicherung a)

Die Strides mit Konflikten hängen von der Anzahl der Banken und der Sperrzeit ab:

- LOAD(\underline{a}) Stride 1 \implies keine Konflikte
- LOAD(\underline{a}) Stride 2,4,6,8,... \implies Konflikte
- LOAD(\underline{a}) Stride 3,5,7,... \implies keine Konflikte

Eine andere Strategie bei der Abspeicherung des Vektors \underline{a} ist in Fig. 4.5 dargestellt.



Fig. 4.5 Speicherung b)

Hier treten bei Stride 1,2,3 Speicherbankkonflikte auf, der konkrete maximale Stride mit Konflikten hängt von der Länge der Bank ab.

- LOAD(\underline{a}) Stride 1,2,3 \implies Konflikte
- LOAD(\underline{a}) Stride 4,5,6,... \implies keine Konflikte

Speicherbankkonflikte senken schon bei normalen Prozessoren die Leistung, bei Vektorrechnern können sie sich dramatisch auswirken!

- Die angeforderte Adresse/Speicherzelle befindet sich nicht im schnellsten Speicher, dh. die entsprechende Seite des Speichers muß nachgeladen werden. Dieser *page fault* muß bei Prozessoren mit Cache und bei virtuellem Speicher beachtet werden.
- Auf Parallelrechnern mit verteiltem Speicher soll natürlich möglichst wenig Kommunikation erfolgen.
- Ausnutzung der Länge der Vektorregister beim Vektorrechner.

Algorithmen für $C_{n_1 \times n_3} := C_{n_1 \times n_3} + A_{n_1 \times n_2} * B_{n_2 \times n_3}$

1. *inner product*

```
DO i := 1, n3
  DO j := 1, n1
     $c_{i,j} := c_{i,j} + \langle A_{i,*}, B_{*,j} \rangle$ 
  END DO
END DO
```

- Zugriff auf A zeilenweise, auf B spaltenweise \implies Bankkonflikte
- Zugriff auf C ist skalar \implies schlecht vektorisierbar.
- Parallelisierung \Leftrightarrow Aufteilung der Matrizen.

2. *middle product*

```
DO j := 1, n1
  DO k := 1, n2
     $C_{*,j} := C_{*,j} + A_{*,k} * B_{k,j}$ 
  END DO
END DO
```

- $B_{*,j}$ fungiert in innerster Schleife als Skalar, die anderen Größen als Vektoren \implies Triade
- \implies Effiziente Nutzung der Vektorregister und der Cachebandbreite.
- Die k -Schleife ist das Matrix-Vektor Produkt aus BLAS2.
- Spaltenzugriff auf A und C notwendig.
- Liegen A und C in Zeilenspeicherung vor, müssen j und i in obigem Algorithmus vertauscht werden.

3. *outer product*

```
DO k := 1, n2
  DO j := 1, n1
     $C_{*,j} := C_{*,j} + A_{*,k} * B_{k,j}$ 
  END DO
END DO
```

- Wie middle product, jedoch kompakter programmierbar.

Parallelisierung von $C_{n \times n} := C_{n \times n} + A_{n \times n} * B_{n \times n}$

Ausgangspunkt : - outer product mit Zeilenzugriff

- Blockmatrizen $C^{i,j}, A^{i,j}, B^{i,j}$
- 2D-Torus ($n \times n$) Topologie, verteilter Speicher


```

DO k := 1, n
  DO i := 1, n
    Ci,* := Ci,* + Ai,k * Bk,*
  END DO
END DO

```

Der folgende Algorithmus geht auf Fox zurück [Fox 88, KGGK 94].

- $C^{i,j}, A^{i,j}, B^{i,j}$ in Speicher von Prozeß (i,j) =: $\mathbb{P}^{[i,j]}$
- $\tilde{B}^{i,j} := B^{i,j}$
- DO $k := 0, n - 1$
 - broadcast* : $\mathbb{P}^{[i,(k+i) \bmod n]}$ sendet $A^{[i,(k+i) \bmod n]}$ an alle Prozesse $\mathbb{P}^{[i,*]}$
 - $T := A^{[i,(k+i) \bmod n]}$
 - multiply* : $C_{i,j} := C_{i,j} + T * \tilde{B}_{i,j}$
 - roll* : Sende $\tilde{B}_{i,j}$ an $\mathbb{P}^{[i-1,j]}$

Alg. 4.4 Broadcast-multiply-roll Algorithmus

Am Ende von Alg. 4.4 gilt wieder $\tilde{B}_{i,j} = B_{i,j}$.

4.2 Elimination durch Drehungsmatrizen

Zu lösen ist das lineare Gleichungssystem

$$A_{n \times n} \cdot \underline{x} = \underline{b}. \quad (4.1)$$

Die *Gauß-Elimination* (hier nur der erste Schritt)

$$\begin{pmatrix} 1 & 0 & 0 & \dots \\ -\frac{a_{21}}{a_{11}} & 1 & 0 & \dots \\ 0 & 0 & 1 & \dots \\ \vdots & \vdots & \ddots & \ddots \end{pmatrix} \cdot A \cdot \underline{x} = \begin{pmatrix} 1 & 0 & 0 & \dots \\ -\frac{a_{21}}{a_{11}} & 1 & 0 & \dots \\ 0 & 0 & 1 & \dots \\ \vdots & \vdots & \ddots & \ddots \end{pmatrix} \cdot \underline{b} \quad (4.2)$$

ist mittels

- der DAXPY-Operation relativ einfach vektorisierbar und
- bei entsprechender Aufteilung der Matrix auch gut parallelisierbar.

Jedoch ist das Verfahren numerisch instabil !

Zur Stabilisierung wird daher zusätzlich eine *Pivotsuche mit Zeilen/Spaltenvertauschung* durchgeführt. Dieses Verfahren ist

- einfach vektorisierbar, jedoch
- tritt bei Parallelrechnern mit verteiltem Speicher viel Kommunikation auf.

Eine bessere Parallelisierbarkeit kann durch Elimination mittels *Givensrotation* erzielt werden.

4.2.1 Die Givensrotation

Die Transformationsmatrix aus (4.2) wird leicht verändert (wiederum nur der erste Schritt)

$$\begin{pmatrix} c & s & 0 & \cdots \\ -s & c & 0 & \cdots \\ 0 & 0 & 1 & \cdots \\ \vdots & \vdots & & \ddots \end{pmatrix} \cdot A \cdot \underline{x} = \begin{pmatrix} c & s & 0 & \cdots \\ -s & c & 0 & \cdots \\ 0 & 0 & 1 & \cdots \\ \vdots & \vdots & & \ddots \end{pmatrix} \cdot \underline{b}, \quad (4.3)$$

mit dem Ziel

$$-s \cdot a_{11} + c \cdot a_{21} = 0.$$

Die Wahl

$$s = \frac{a_{21}}{\sqrt{a_{11}^2 + a_{21}^2}} \quad \text{und} \quad c = \frac{a_{11}}{\sqrt{a_{11}^2 + a_{21}^2}} \quad (4.4)$$

entspricht durch die Normierung exakt den Einträgen $s = \sin \varphi$ und $c = \cos \varphi$ einer Drehungsmatrix. In der Gaußelimination wurde dagegen $\frac{a_{21}}{a_{11}} = \tan \varphi$ verwendet. Der Vorteil von (4.4) besteht in:

$a_{11} = 0 \implies s = 1, c = 0 \implies$ numerisch stabil \implies keine Pivotsuche.

Da der Hauptaufwand bei den Berechnungen in der Transformation der Matrix A liegt, wird nur die Anwendung der Rotation auf diese betrachtet. Im folgenden bezeichnen die Vektoren \underline{u} und \underline{v} die durch die Rotation zu verändernden Matrixzeilen, $\hat{\underline{u}}$ und $\hat{\underline{v}}$ stellen die resultierenden Matrixzeilen dar. Eine einzelne Givensrotation

$$\begin{aligned} \hat{\underline{u}} &:= c \cdot \underline{u} + s \cdot \underline{v} \\ \hat{\underline{v}} &:= c \cdot \underline{v} - s \cdot \underline{u} \end{aligned} \quad (4.5)$$

ist in manchen BLAS1-Bibliotheken als *ein* Aufruf enthalten (SROT/DROT).

4.2.2 Givensrotation auf dem Vektorrechner

Durch die wechselseitige Datenabhängigkeit der Zeilen \underline{u} und \underline{v} in (4.5) kommt man auf dem Vektorrechner nicht ohne Zwischenspeicherung aus :

$$\begin{aligned} \underline{h} &:= -s \cdot \underline{u} \\ \underline{u} &:= c \cdot \underline{u} + s \cdot \underline{v} \\ \underline{v} &:= c \cdot \underline{v} + \underline{h} \end{aligned} \tag{4.6}$$

4.2.3 Givensrotation auf dem Parallelrechner

Bezeichne $g(k, s)$ die Givensrotation der Zeilen $k - 1$ und k und der Spalte s .

$$\begin{aligned} g(k, s) &: \text{ falls } a_{k,s} = 0 \longrightarrow \text{Return} \\ &\text{sonst } w := \sqrt{a_{k,s}^2 + a_{k-1,s}^2} \\ &c := \frac{a_{k-1,s}}{w}, \quad s := \frac{a_{k,s}}{w} \\ &a_{k-1,s} := w, \quad a_{k,s} := 0 \\ &\text{for } j = s+1, n : \quad h := -s \cdot a_{k-1,j} \\ &\quad a_{k-1,j} := c \cdot a_{k-1,j} + s \cdot a_{k,j} \\ &\quad a_{k,j} := c \cdot a_{k,j} + h \end{aligned}$$

Alg. 4.5 $g(k, s)$ - Givensrotation der Zeilen $k - 1$ und k und der Spalte s

Hierbei tritt folgende Datenabhängigkeit auf:

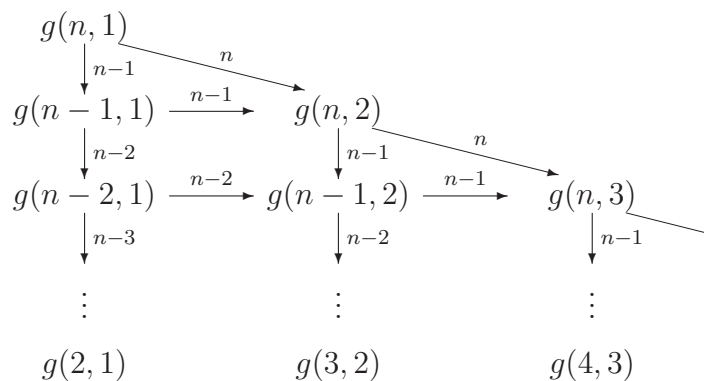


Fig. 4.6 Statische Datenabhängigkeit Givensrotation

Die statische Datenabhängigkeit von Fig. 4.6 wandeln wir durch sogenannte

Marker * in ein *dynamisches Datenflußkonzept* um, mit $m(k, s)$ als der Anzahl der Marker für den Prozeß $g(k, s)$.

Anfangswerte : $m(n, 1) := 2$
 $m(k, 1) := 1 \quad k < n$
 $m(k, s) := 0 \quad s > 1$

Falls $m(k, s) = 2$, dann wird Prozeß $g(k, s)$ gestartet und danach werden $m(k - 1, s)$ und $m(k + 1, s + 1)$ um 1 erhöht.

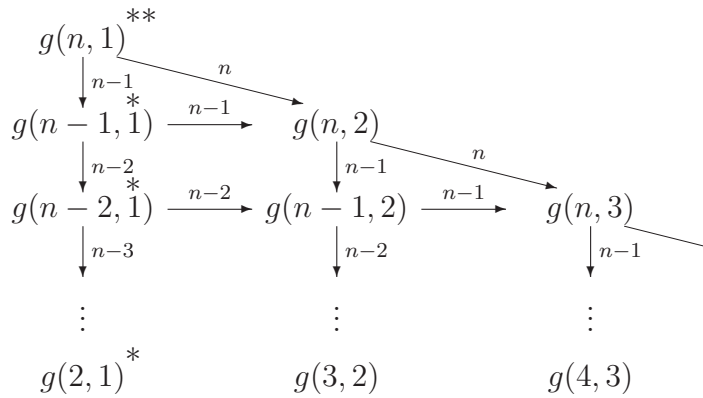


Fig. 4.7 Datenfluß bei gemeinsamem Speicher : Givensrotation

Haben die Prozessoren jedoch nur Zugriff zu ihrem Teil des *verteilten Speichers*, müssen noch die Abhängigkeiten zwischen den Prozessoren untersucht werden. Hierbei bietet sich eine blockzeilenweise Aufteilung der Matrix A an, wie in Fig. 4.8 dokumentiert wird.

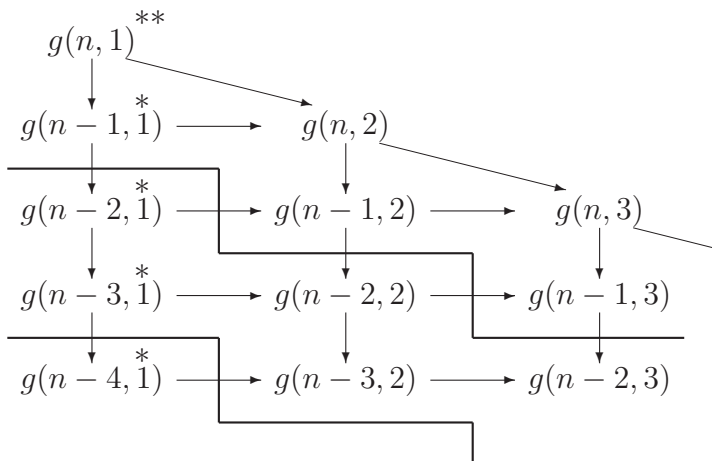


Fig. 4.8 Datenfluß bei verteiltem Speicher : Givensrotation

4.3 Die LU-Zerlegung

4.3.1 Der serielle Standardalgorithmus

Sei $A_{n \times n}$ - vollbesetzt, zeilenweise gespeichert,
 L - untere Dreiecksmatrix, spaltenweise gespeichert,
 U - obere Dreiecksmatrix, zeilenweise gespeichert.

und es soll gelten $\sum_{j=1}^n \ell_{i,j} \cdot u_{j,k} = a_{i,k}$ mit Normierung $\ell_{i,i} = 1$ (Alg. 4.6).

```

DO  $i = 1, n$ 
   $\ell_{i,i} := 1$ 
  DO  $k = 1, i - 1$ 
     $\ell_{i,k} := (a_{i,k} - \sum_{j=1}^{k-1} \underbrace{\ell_{i,j}}_{i\text{-te Zeile}} \cdot \underbrace{u_{j,k}}_{k\text{-te Spalte}}) / u_{k,k}$ 
  END DO
  DO  $k = 1, i$ 
     $u_{k,i} := a_{k,i} - \sum_{j=1}^{k-1} \underbrace{\ell_{k,j}}_{k\text{-te Zeile}} \cdot \underbrace{u_{j,i}}_{i\text{-te Spalte}}$ 
  END DO
END DO

```

Alg. 4.6 LU-Zerlegung ohne Pivotisierung - seriell

4.3.2 Vektorisierung der LU-Zerlegung

Die Speicherung der Matrizen A , L und U bleibt gleich. Alle Vektoroperationen werden in eckigen Klammern dargestellt, mit der Laufvariablen als Index. Falls nicht anders notiert, wird auf einen Vektor mit Stride 1 zugegriffen (Alg. 4.7).

```

[ $\ell_{i,i} := 1$ ] $_{i=1,n}$ 
DO  $i = 1, n$ 
  DO  $k = 1, i - 1$ 
     $h_k := [\ell_{i,j} \cdot u_{j,k}]_{j=1,k-1}$ 
  END DO
  [ $\ell_{i,k} := a_{i,k} - h_k$ ] $_{k=1,i-1}$ 
  DO  $k = 1, i - 1$ 
     $\ell_{i,k} := \ell_{i,k}/u_{k,k}$ 
  END DO
  DO  $k = 1, i$ 
     $h_k := [\ell_{k,j} \cdot u_{j,i}]_{j=1,k-1}$ 
  END DO
  [ $u_{k,i} := \underbrace{a_{k,i}}_{\text{Stride}=n} - h_k$ ] $_{k=1,i}$ 
END DO

```

nicht $[\ell_{i,k} := \ell_{i,k} \cdot u_{k,k}]_{k=1,i-1}$,
da Stride von u gleich k sein müßte !

A zeilenweise gespeichert !

Alg. 4.7 LU-Zerlegung ohne Pivotisierung - vektoriell

```

DO  $i = 1, n$ 
  DO  $k = i, n$ 
     $u_{i,k} := a_{i,k}$ 
  END DO
   $\ell_{i,i} := 1$ 
  DO  $k = i + 1, n$ 
     $\ell_{k,i} := a_{k,i}/u_{i,i}$ 
  END DO
  DO  $k = i + 1, n$ 
    DO  $j = i + 1, n$ 
       $a_{k,j} := a_{k,j} - \ell_{k,i} \cdot u_{i,j}$ 
    END DO
  END DO
END DO

```

Bestimmung i -te Zeile von U

Bestimmung i -te Spalte von L

Transformation der Restmatrix

Alg. 4.8 Rang-r-Modifikation der LU-Zerlegung - seriell

4.3.3 Parallelisierung der LU-Zerlegung

Im folgenden nehmen wir einen Parallelrechner mit *verteilterm Speicher* an. Zur Vorbereitung der Parallelisierung benötigen wir eine Modifikation der LU-Zerlegung (Alg. 4.8), wobei diesmal L spaltenweise, und U zeilenweise gespeichert sein soll. Eine Pivotisierung wird nicht betrachtet.

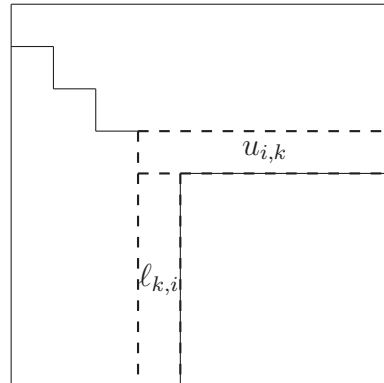


Fig. 4.9
Illustration zur Rang-r-Modifikation

Zum Zwecke der Parallelisierung bietet sich die Blockvariante der Rang-r-Modifikation an. In Alg. 4.9 bezeichnet $A_{k,i}$ den Block der k -ten Zeile und i -ten Spalte der Blockaufteilung von Matrix A .

```

DO i = 1, n
  a) Faktorisierung von  $A_{k,i}$  aus der Beziehung  $A_{k,i} = L_{k,i} \cdot U_{i,i}$  ( $k=\overline{i,n}$ )
      $\longrightarrow U_{i,i}, L_{i,i}, L_{i+1,i}, \dots, L_{n,i}$ 
  b) Bestimme  $U_{i,\ell}$  aus  $A_{i,\ell} = L_{i,i} \cdot U_{i,\ell}$  ( $\ell=\overline{i+1,n}$ )
      $\longrightarrow U_{i,i+1}, \dots, U_{i,n}$ 
  c) Rang-r-Modifikation der Restmatrix  $A_{k,\ell} := A_{k,\ell} - L_{k,i} \cdot U_{i,\ell}$  ( $k,\ell=\overline{i+1,n}$ )
END DO

```

Alg. 4.9 Blockvariante der Rang-r-Modifikation

Jedoch treten bei einer zeilen- bzw. spaltenweisen Aufteilung der Matrix A größere Inbalancen in der Auslastung der Prozessoren auf, da mit sich verkleinernder Restmatrix immer weniger Prozessoren an den Berechnungen teilnehmen, siehe Fig. 4.9.

Eine möglichst gleiche Auslastung aller Prozessoren wird durch die gestreute Aufteilung von quadratischen Blockmatrizen (*engl.: square block scattered decomposition*) erreicht, wie sie in der parallelisierten Version von ScaLAPACK benutzt wird.

$A_{1,1}$ \mathbb{P}_0	$A_{1,2}$ \mathbb{P}_1	$A_{1,3}$ \mathbb{P}_2	$A_{1,4}$ \mathbb{P}_0	$A_{1,5}$ \mathbb{P}_1
$A_{2,1}$ \mathbb{P}_3	$A_{2,2}$ \mathbb{P}_4	$A_{2,3}$ \mathbb{P}_5	$A_{2,4}$ \mathbb{P}_3	$A_{2,5}$ \mathbb{P}_4
$A_{3,1}$ \mathbb{P}_6	$A_{3,2}$ \mathbb{P}_7	$A_{3,3}$ \mathbb{P}_8	$A_{3,4}$ \mathbb{P}_6	$A_{3,5}$ \mathbb{P}_7
$A_{4,1}$ \mathbb{P}_0	$A_{4,2}$ \mathbb{P}_1	$A_{4,3}$ \mathbb{P}_2	$A_{4,4}$ \mathbb{P}_0	$A_{4,5}$ \mathbb{P}_1
$A_{5,1}$ \mathbb{P}_3	$A_{5,2}$ \mathbb{P}_4	$A_{5,3}$ \mathbb{P}_5	$A_{5,4}$ \mathbb{P}_3	$A_{5,5}$ \mathbb{P}_4

5×5 Matrixblöcke
 3×3 Prozessoren ($i=\overline{0,n}$)
 $A_{i,j} \longrightarrow \mathbb{P}_{(i-1) \bmod P_x, (j-1) \bmod P_y}$

Fig. 4.10 Square block scattered decomposition einer vollbesetzten Matrix

Mittels der gestreuten Aufteilung der Matrix A kann nun die Parallelisierung der Blockvariante in Alg. 4.10 aufgeschrieben werden.

$\mathbb{P}(A_{i,j})$ sei derjenige Prozeß, welcher den Block $A_{i,j}$ speichert. DO $i = 1, n$ a) $\mathbb{P}(A_{i,i})$: Bestimmt $L_{i,i}$ und $U_{i,i}$ aus $A_{i,i} = L_{i,i} \cdot U_{i,i}$. Sendet $L_{i,i}$ via Ring in Spaltenrichtung an alle $\mathbb{P}(A_{i,\ell})$ ($\ell=\overline{i+1,n}$). Sendet $U_{i,i}$ via Ring in Zeilenrichtung an alle $\mathbb{P}(A_{k,i})$ ($k=\overline{i+1,n}$). b) $\mathbb{P}(A_{k,i})$ [$k=\overline{i+1,n}$] : Bestimmt $L_{k,i}$ aus $A_{k,i} = L_{k,i} \cdot U_{i,i}$. Sendet $L_{k,i}$ via Ring in Spaltenrichtung an alle $\mathbb{P}(A_{k,\ell})$ ($\ell=\overline{i+1,n}$). $\mathbb{P}(A_{i,\ell})$ [$\ell=\overline{i+1,n}$] Bestimmt $U_{i,\ell}$ aus $A_{i,\ell} = L_{i,i} \cdot U_{i,\ell}$. Sendet $U_{i,\ell}$ via Ring in Zeilenrichtung an alle $\mathbb{P}(A_{k,\ell})$ ($k=\overline{i+1,n}$). c) $\mathbb{P}(A_{k,\ell})$ [$k,\ell=\overline{i+1,n}$] $A_{k,\ell} := A_{k,\ell} - L_{k,i} \cdot U_{i,\ell}$ END DO

Alg. 4.10 Blockweise Rang-r-Modifikation der LU-Zerlegung - parallel

Bemerkung 4.1. Vom Standpunkt der Implementierung empfiehlt sich in Alg. 4.10 eine nichtblockierende Kommunikation. In Richtung der Spalten und Zeilen ist auch eine Verteilung der Matrixblöcke analog der Hypercubenummerierung denkbar.

4.4 Gaußelimination für tridiagonale Matrizen

Sei A positiv definit und tridiagonal, das Gleichungssystem

$$A \cdot \underline{x} = \underline{f}$$

soll mittels Gaußelimination gelöst werden.

Seriell und vektoriell ist die Gaußelimination bei tridiagonalen Matrizen sofort implementierbar. Es ergibt sich der klassische Eliminationsbaum in Fig. 4.11. Leider ist die in diese klassische Elimination nicht parallelisierbar.

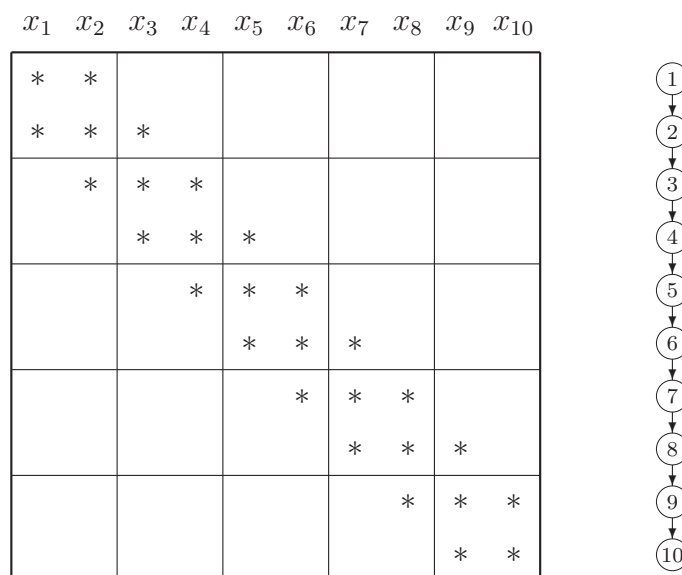


Fig. 4.11 Tridiagonale Matrix und Klassischer Eliminationsbaum

4.4.1 Parallelisierung mittels zyklischer Reduktion

Definition 4.1. [Zyklische Reduktion] Umordnung der Gleichungen und Unbekannten derart, daß mehrere Unbekannte gleichzeitig (parallel) eliminiert werden können.

Grundidee der Umordnung für $N = 2^3 - 1 = 7$ Unbekannte:

Eliminiere alle x_i mit $i \bmod 2 = 1$ aus dem GIS :

$$\begin{array}{rcl}
 b_1x_1 + c_1x_2 & & = f_1 \\
 a_1x_1 + b_2x_2 + c_2x_3 & & = f_2 \\
 & a_2x_2 + b_3x_3 + c_3x_4 & = f_3 \\
 & & a_3x_3 + b_4x_4 + c_4x_5 & = f_4 \\
 & & & a_4x_4 + b_5x_5 + c_5x_6 & = f_5 \\
 & & & & a_5x_5 + b_6x_6 + c_6x_7 & = f_6 \\
 & & & & & a_6x_6 + b_7x_7 & = f_7
 \end{array}$$

$$\begin{array}{rcl}
 \Rightarrow \tilde{b}_2x_2 + \tilde{c}_3x_4 & & = \tilde{f}_2 \\
 \tilde{a}_2x_2 + \tilde{b}_4x_4 + \tilde{c}_5x_6 & & = \tilde{f}_4 \\
 & \tilde{a}_4x_4 + \tilde{b}_6x_6 & = \tilde{f}_3
 \end{array}$$

Eliminiere alle x_i mit $i \bmod 4 = 2 \Rightarrow \hat{b}_4x_4 = \hat{f}_4$

Allg.:

Eliminationsreihenfolge : $2^{k-1} \leq n < 2^k$
 DO $j = 0, k - 1$
 Eliminiere alle x_i mit $i \bmod 2^{j+1} = 2^j$
 END DO

$x_1 \ x_3 \ x_5 \ x_7 \ x_9 \ x_2 \ x_6 \ x_{10} \ x_4 \ x_8$

*			*					*	
	*			*			*		*
		*		*		*		*	*
*	*		*		*			○	
	*	*		*		*		○	○
*	*		○		○		*	○	
		*	*		○	○	○	*	*

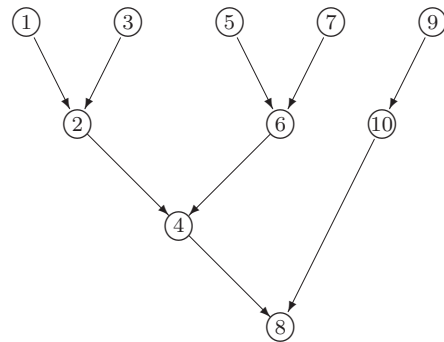


Fig. 4.12 Matrix und Eliminationsbaum nach der Umordnung durch zyklische Reduktion.

Die ○ - Einträge in Fig. 4.12 kennzeichnen das nunmehr auftretende Fill-In.

- Die zyklische Reduktion benötigt doppelt so viel Arithmetik wie die normale Gaußelimination, aber
- sie ist parallel abarbeitbar.
- Eine sequentielle Umordnung zur Fill-In Reduktion hat keinerlei Bedeutung für die parallele Abarbeitung.

Bemerkung 4.2. Es gibt es für die Parallelisierung direkter Verfahren für dünnbesetzte (sparse) Matrizen kein generelles Konzept !

4.5 Die Fast Fourier Transformation

Die Fourier Transformation basiert auf der Zerlegung einer Funktion in bestimmte Frequenzen (Eigenfunktionen) ausgedrückt durch die trigonometrischen Grundfunktionen \sin und \cos . Bei einer Anzahl von $N = 2^P - 1$ Eigenfunktionen läßt sich diese Transformation wesentlich beschleunigen und wird als Fast Fourier Transformation (FFT) bezeichnet.

Anwendung :

- Signalverarbeitung,
- Partielle Differentialgleichungen über Rechteckgebieten,
- Vorkonditionierung zyklischer Matrizen (BEM).

Beispiel 4.1. [Partielle Differentialgleichung im Einheitsquadrat] Aus der partiellen DGL.

$$\begin{array}{l} -\Delta u + cu = f \quad \text{in } \Omega \\ u = \varphi (:= 0) \quad \text{auf } \Gamma = \partial\Omega \end{array}$$

soll in $\Omega = (0, 1)^2$ die Funktion u bestimmt werden. Die restlichen Funktionen und Konstanten sind gegeben

Mit einem äquidistanten Gitternetz ($N + 1$ Linien in jede Dimension) wird das Gebiet diskretisiert und die Differentialgleichung mittels des 5-Punkte Differenzensterns approximiert.

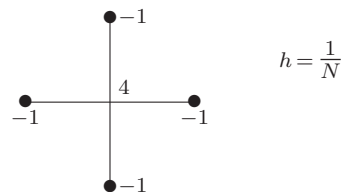


Fig. 4.13
5-Punkte Differenzenstern

Unter Beachtung der homogenen Dirichletrandbedingungen $\varphi = 0$ führt dies auf ein (lineares) Gleichungssystem

$$\begin{aligned} (4 + ch^2)u_{i,j} - (u_{i,j-1} + u_{i,j+1} + u_{i-1,j} + u_{i+1,j}) &= h^2 f_{i,j} & i,j=\overline{1,N-1} \\ &\Downarrow \\ K\underline{u} &= \underline{f} . \end{aligned}$$

Da in obigem Beispiel die Eigenfunktionen des diskretisierten Differentialoperators (hier eine Matrix) K

$$\mu_{k,\ell}(ih, jh) = 2 \sin(k\pi ih) \sin(\ell\pi jh) \quad k,\ell=\overline{1,N-1}$$

sind, und sich die zugehörigen Eigenwerte als

$$\lambda_{k,\ell} = 4 \left[\sin^2 \frac{k\pi h}{2} + \sin^2 \frac{\ell\pi h}{2} \right] + ch^2$$

darstellen lassen, können die rechte Seite f und die gesuchte Funktion u als Linearkombination der Eigenfunktionen dargestellt werden.

$$f_{i,j} = f(ih, jh) = \sum_{k,\ell=1}^{N-1} \gamma_{k,\ell} \cdot \mu_{k,\ell}(ih, jh) \quad (4.7a)$$

$$u_{i,j} = u(ih, jh) = \sum_{k,\ell=1}^{N-1} \beta_{k,\ell} \cdot \mu_{k,\ell}(ih, jh) \quad (4.7b)$$

Die Lösung der Differentialgleichung erhält man nunmehr wie folgt

1. Zerlegung von f in Eigenfrequenzen $\mu_{k,\ell}$ (*Fourieranalyse*)
 \implies Bestimmung der Zerlegungskoeffizienten $\gamma_{k,\ell}$ in (4.7a).
2. Durch die Eigenwerte des diskretisierten Operators dividieren
 $\implies \beta_{k,\ell} := \frac{\gamma_{k,\ell}}{\lambda_{k,\ell}}$.
3. Zusammenbau der diskreten Näherungslösung (*Fouriersynthese*)
 $\implies u(ih, jh)$ in (4.7b).

Die Punkte 1 und 3 werden über die Fouriertransformation realisiert.

4.5.1 Die 1D-Fourieranalyse und Synthese

Wegen der einfacheren Darstellung betrachten wir die Fouriertransformation für die Sinus- und Cosinuentwicklung, obwohl dann komplexe Koeffizienten $\gamma_{k,\ell}$, $\beta_{k,\ell}$ betrachtet werden müssen.

Sei ω die n -te Einheitswurzel von 1 (in \mathbb{C}), dann ist die Fouriertransformationmatrix \mathcal{F} mittels

$$\mathcal{F}_n = \{\mathcal{F}_{j,k}\}_{j,k=0,\overline{n-1}} := \{\omega^{j \cdot k}\}_{j,k=0,\overline{n-1}} \quad (4.8)$$

definiert. Zusammen mit der Normierung $\frac{1}{n}$ schreibt man

- die Fourieranalyse als : $\underline{\gamma} := \frac{1}{n} \cdot \mathcal{F} \cdot \underline{f}$
- die Fouriersynthese als : $\underline{u} := \mathcal{F} \cdot \underline{b}$.

4.5.2 Die FFT-Idee und ihre Parallelisierung

Die hier betrachtete FFT-Idee wird auch als *radix-2 FFT* bezeichnet.

Betrachten wir für $n = 4$ die Matrix \mathcal{F} aus (4.8).

$$\mathcal{F}_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 \\ 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega^9 \end{bmatrix}$$

Wegen $\omega^s = \omega^{s \bmod 4}$ und $\omega_{n=4} = e^{-\frac{2\pi}{4}i} = -i$ ergibt sich

$$\mathcal{F}_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 \\ 1 & \omega^2 & 1 & \omega^2 \\ 1 & \omega^3 & \omega^2 & \omega \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix} . \quad (4.9)$$

Eine Umgruppierung der Einträge in (4.9) mittels der 4×4 -Permutationsmatrix

$$\Pi_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (4.10)$$

liefert

$$\mathcal{F}_4 \Pi_4 = \left[\begin{array}{cc|cc} 1 & 1 & 1 & 1 \\ 1 & -1 & -i & i \\ \hline 1 & 1 & -1 & -1 \\ 1 & -1 & i & -i \end{array} \right] . \quad (4.11)$$

Wegen $\mathcal{F}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$ und der Definition $\Omega_2 := \begin{bmatrix} 1 & 0 \\ 0 & -i \end{bmatrix} = \text{diag}\{1, \omega_{n=4}\}$ folgt

$$\mathcal{F}_4 \Pi_4 = \begin{bmatrix} \mathcal{F}_2 & \Omega_2 \mathcal{F}_2 \\ \mathcal{F}_2 & -\Omega_2 \mathcal{F}_2 \end{bmatrix}, \quad (4.12)$$

d.h. die Fourieranalyse mit $n = 4$ läßt sich auf die Fourieranalyse mit $n = 2$ zurückführen. *Allgemein* :

Sei $n = 2m$, so gilt

$$\mathcal{F}_n \Pi_n = \begin{bmatrix} \mathcal{F}_m & \Omega_m \mathcal{F}_m \\ \mathcal{F}_m & -\Omega_m \mathcal{F}_m \end{bmatrix}, \quad (4.13)$$

mit Π_n als even-odd-Permutation, welche die Vektorkomponenten umordnet

$$\underline{y} := \Pi_n^T \underline{x} \Leftrightarrow \underline{y} = \begin{bmatrix} x_j & j=\overline{0, n-1:2} \\ x_j & j=\overline{1, n-1:2} \end{bmatrix} \quad (4.14)$$

und $\Omega_m := \text{diag}\{\omega_{n=2m}^k, \overline{k=0, m-1}\}$.

Wenn $n = 2^P$ ist, so läßt sich (4.13) $P - 1$ -mal rekursiv anwenden. Damit reduziert sich eine FFT auf das $P - 1$ -malige Umordnen eines Vektors mit etwas zusätzlicher Arithmetik.

Beispiel 4.2. [FFT mit $n = 8$] $x(0, 7 : 2)$ bezeichne im folgenden den Teilvektor von x , der ab Element 0 bis Element 7 jede zweite Komponente enthält, d.h. $x(0, 7 : 2) = [x_j]_{j=\overline{0, 7:2}}$.

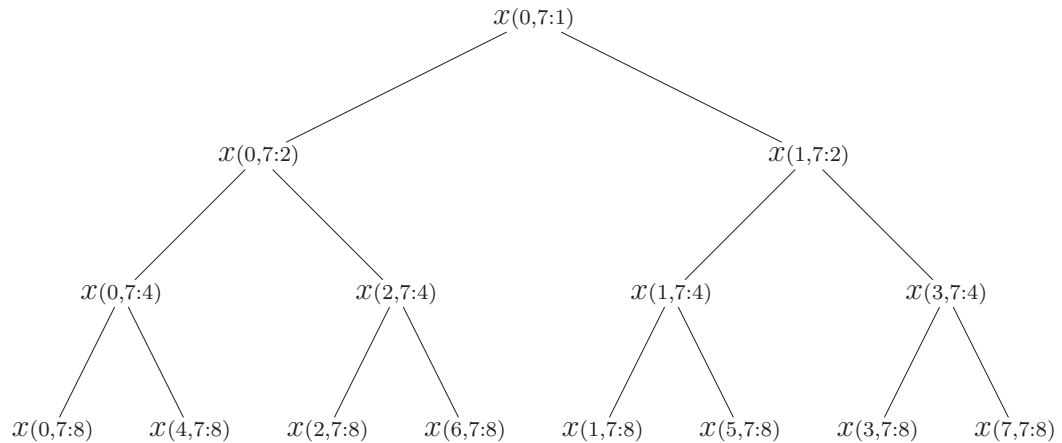


Fig. 4.14 Aufsplittung der Indizes bei der FFT.

Der resultierende Vektor nach der Umordnung in Fig. 4.14 ist nunmehr als $[x_0, x_4, x_2, x_6, x_1, x_5, x_3, x_7]^T$ geordnet.

Die Parallelisierung der Fouriertransformation \mathcal{F} (Umordnung und Arithmetik) auf einem *Parallelrechner* mit verteiltem Speicher ist in Fig. 4.15 als sogenannter *Butterfly-Algorithmus* dargestellt. Hierin sieht man die gute Parallelisierbarkeit der Fouriertransformation, jedoch liegen die resultierenden Komponenten ungeordnet vor, deren Umordnung wiederum mit *Zusatzkommunikation* verbunden ist.

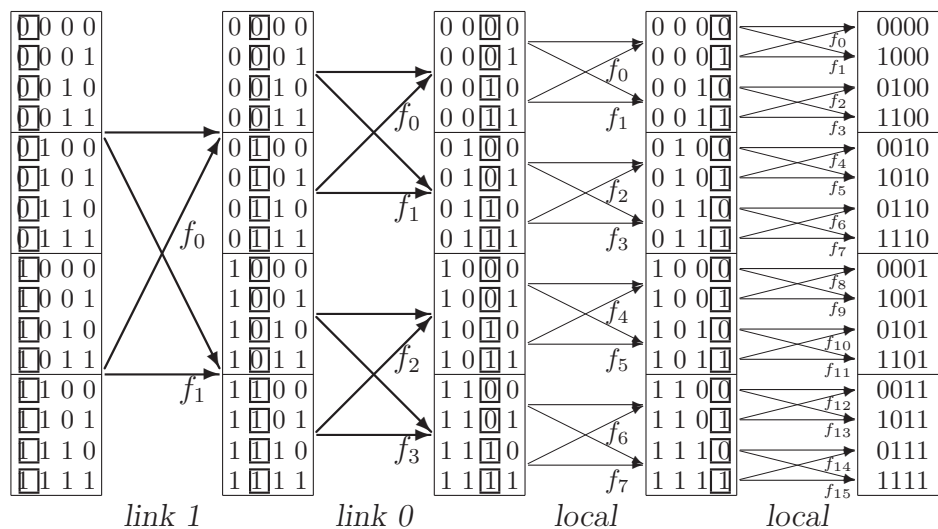


Fig. 4.15 Datenfluß der FFT bei 4 Prozessoren [Dr. Pester Chemnitz].

Die hervorgehobenen Bits der Indizes in Fig. 4.15 bestimmen des entsprechende Bit der nächsten Butterfly-Operation.

Zurückblickend auf das Beispiel zu Beginn dieses Abschnittes 4.5, ist an Stelle der diskreten Gleichung $K\underline{u} = \underline{f}$ die Gleichung

$$\frac{1}{n} \mathcal{F} K \mathcal{F} \mathcal{F} \cdot \underline{u} = \mathcal{F} \cdot \underline{f}$$

zu lösen :

- a) Fourieranalyse : $\underline{\gamma} := \frac{1}{n} \mathcal{F} \underline{f}$ $\underline{\gamma}$ umgeordnet
- b) Operator Λ^{-1} : $\beta_i := \frac{\gamma_i}{\lambda_i}$ $\underline{\beta}$ umgeordnet
- c) Fouriersynthese : $\underline{u} := \mathcal{F} \underline{\beta}$ \underline{u} richtig geordnet !!

Da zweimal die Fouriertransformation angewendet werden mußte, liegen zum Schluß die Komponenten der Lösung wieder in richtiger Reihenfolge vor !

⇒ Die Kombination von Fourieranalyse und -synthese ist sehr gut parallelisierbar.

5 Gebietszerlegungen und numerische Grundroutinen

Die Datenaufteilung bzgl. einer Gebietsaufteilung nutzt geometrische Zusammenhänge für die Parallelisierung aus. Das Rechengebiet Ω wird nun in P Teilgebiete Ω_i ($i=\overline{1,P}$) aufgeteilt, welche jeweils einem Prozeß zugeordnet werden. Das Fig. 5.1 zeigt einen Ausschnitt einer nichtüberlappenden Gebietszerlegung. Die Kanten zwischen den Teilgebieten werden als Interfaces bezeichnet. Die überlappende Gebietszerlegung wird im folgenden nicht betrachtet.

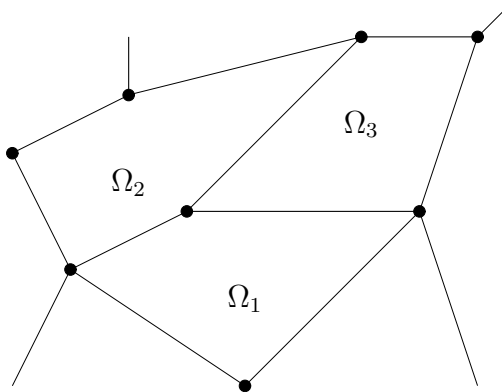


Fig. 5.1
Nichtüberlappende Gebietsaufteilung.

Bezüglich der Datenart unterscheiden wir zwischen element- und knotenbasierten Daten, welche nunmehr überlappend oder nichtüberlappend gespeichert werden (Aber die Gebiete sind nichtüberlappend aufgeteilt!). Somit ergeben sich 4 Arten der Datenverteilung, von denen die Verteilung von (finiten) Elementen eingehend untersucht wird. Generell werden Matrizen betrachtet, welche durch Finite Elemente Methoden (FEM), Finite Differenzen Methoden (FDM) oder Finite Volumen Methoden (FVM) erzeugt wurden. Die hierbei entstehenden dünnbesetzten Matrizen werden im CRS-Format (Compressed Row Storage) gespeichert. Erläuterungen hierzu und weitere Speicherarten für dünnbesetzte Matrizen sind in [BBC⁺ 94]¹ zu finden.

¹http://www.netlib.org/linalg/html_templates/report.html

5.1 Nichtüberlappende Elemente

Sei das Rechengebiet Ω in z.B. 4 Teilgebiete zerlegt und mittels linearen Dreieckselementen diskretisiert. In Fig 5.2 wird dies veranschaulicht.

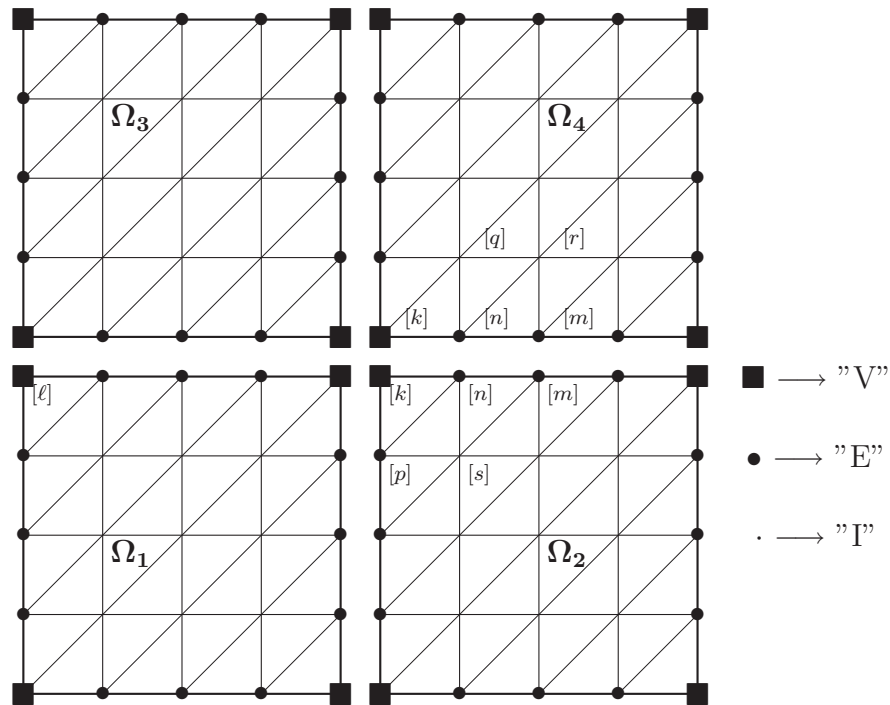


Fig. 5.2 Nichtüberlappende Elemente.

Wir unterscheiden 3 Arten von Knoten mittels der Indizes :

"I" Knoten im Teilgebieteninneren $[N_I := \sum_{i=1}^P N_{I,i}]$,

"E" Knoten auf dem Inneren von Teilgebietenkanten $[N_E := \sum_{j=1}^{n_e} N_{E,j}]$,

"V" Crosspoints (vertices, interfaces), dh. Knoten, welche zu Beginn oder Ende einer Teilgebietenkanten gehören $[N_V]$.

Die beiden letzteren werden oft als Koppelknoten mit dem Index "C" zusammengefaßt $[N_C = N_V + N_E]$. Die Gesamtzahl der Knoten ergibt sich zu $N = N_V + N_E + N_I$.

Zur Vereinfachung der Darstellung werden zuerst die Crosspoints, dann die Kantenknoten und danach die inneren Knoten numeriert. Innerhalb der Kantenknoten besitzen die Knoten einer Kante eine fortlaufende Numerierung,

desgleichen die inneren Knoten eines Teilgebietes. Somit besitzen alle Vektoren und Matrizen eine Blockstruktur der Art

$$\underline{v}^T = (\underline{v}_V, \underline{v}_{E,1}, \dots, \underline{v}_{E,n_e}, \underline{v}_{I,1}, \dots, \underline{v}_{I,P})^T .$$

Entsprechend der Zuordnung der Knoten zu den P Teilgebieten $\overline{\Omega}_i$ ($i = \overline{1, P}$) werden die Einträge der Matrizen und Vektoren auf die entsprechenden Prozesse \mathbb{P}_i verteilt. Koinzidenzmatrizen A_i ($i = \overline{1, P}$) repräsentieren symbolisch diese Knotenzuordnung.

Die $N_i \times N$ Matrix A_i ist eine Boolesche Matrix welche einen globalen Vektor \underline{v} auf den lokalen Vektor \underline{v}_i im Teilgebiet Ω_i abbildet.

Es gilt : – Einträge für innere Knoten erscheinen *genau einmal* pro Zeile und Spalte in den A_i .

– Einträge der Koppelknoten treten so oft in den A_i auf, wie es Teilgebiete gibt, zu denen sie gehören.

Nunmehr kann man 2 Vektortypen definieren, den akkumulierten und den verteilten Vektor :

Definition 5.1. [Akkumulierter Vektor] Die Vektoren $\underline{\mathbf{u}}$ und $\underline{\mathbf{w}}$ werden in Prozessor \mathbb{P}_i ($\hat{=} \overline{\Omega}_i$) als

$$\underline{\mathbf{u}}_i = A_i \underline{\mathbf{u}} \tag{5.1}$$

und $\underline{\mathbf{w}}_i = A_i \underline{\mathbf{w}}$ gespeichert, dh. jeder Prozessor \mathbb{P}_i besitzt den vollen Wert in seinen Knoten.

Definition 5.2. [Verteilter Vektor] Die Vektoren $\underline{\mathbf{r}}, \underline{\mathbf{f}}$ werden in \mathbb{P}_i als $\underline{\mathbf{r}}_i, \underline{\mathbf{f}}_i$ gespeichert, so daß

$$\underline{\mathbf{r}} = \sum_{i=1}^p A_i^T \underline{\mathbf{r}}_i \tag{5.2}$$

gilt, dh. die Knoten auf den Interfaces ($\underline{\mathbf{r}}_{C,i}$) besitzen nur einen Teil des wahren Wertes und erst ihre globale Akkumulation erzeugt den vollen Wert.

Die Matrix K wird im verteilten Sinne, analog zum verteilten Vektor, abgespeichert und daher als verteilte Matrix klassifiziert.

$$\mathbf{K} = \sum_{i=1}^p A_i^T \mathbf{K}_i A_i , \tag{5.3}$$

wobei \mathbf{K}_i die zum Teilgebiet $\overline{\Omega}_i$ gehörige Steifigkeitsmatrix darstellt. Denkt man sich $\overline{\Omega}_i$ als großes finites Element, so stellt die verteilte Matrixspeicherung die

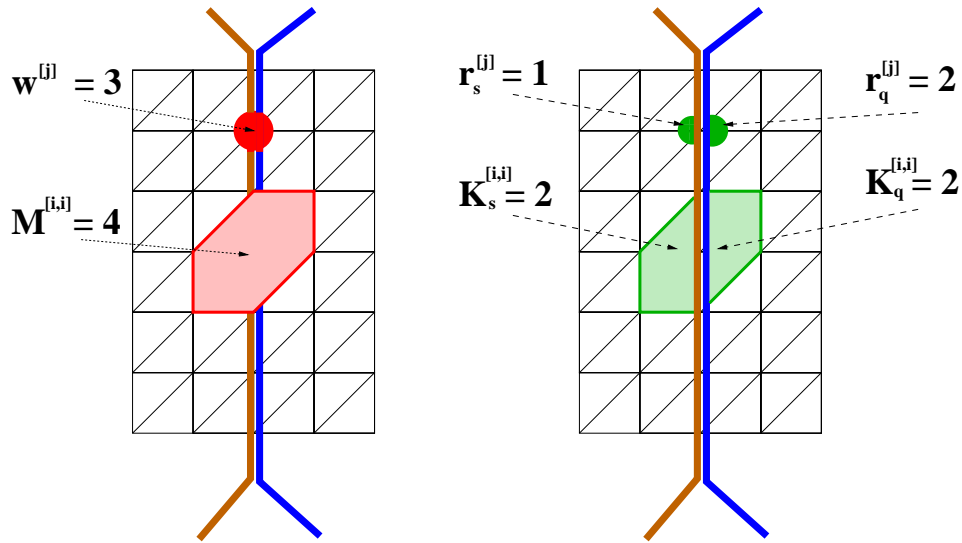


Fig. 5.3 Akkumulierte und verteilte Vektoren \underline{w} , \underline{r} und Matrizen \mathfrak{M} , \mathbf{K}

Steifigkeitsmatrix vor der FE-Akkumulation dar.

Die spezielle globale Knotennummerierung induziert folgende Blockdarstellung der Gleichung $\mathbf{K} \cdot \underline{u} = \underline{f}$:

$$\begin{pmatrix} \mathbf{K}_V & \mathbf{K}_{VE} & \mathbf{K}_{VI} \\ \mathbf{K}_{EV} & \mathbf{K}_E & \mathbf{K}_{EI} \\ \mathbf{K}_{IV} & \mathbf{K}_{IE} & \mathbf{K}_I \end{pmatrix} \cdot \begin{pmatrix} \underline{u}_V \\ \underline{u}_E \\ \underline{u}_I \end{pmatrix} = \begin{pmatrix} \underline{f}_V \\ \underline{f}_E \\ \underline{f}_I \end{pmatrix} . \quad (5.4)$$

Dabei ist \mathbf{K}_I eine Blockdiagonalmatrix mit den Einträgen $\mathbf{K}_{I,i}$, desgleichen sind \mathbf{K}_{IC} , \mathbf{K}_{CI} , \mathbf{K}_{IV} , \mathbf{K}_{VI} Blockmatrizen.

Falls eine globale Akkumulation der Matrix \mathbf{K} ausgeführt wird, bezeichnen wir das Ergebnis mit der akkumulierten Matrix \mathfrak{M} und schreiben

$$\mathfrak{M}_i := A_i \mathfrak{M} A_i^T . \quad (5.5)$$

Obwohl $\mathbf{K} \equiv \mathfrak{M}$ gilt, müssen wir beide wegen der unterschiedlichen Darstellung (und lokalen Abspeicherung) unterscheiden!

Die Diagonalmatrix

$$\boxed{R = \sum_{i=1}^p A_i^T A_i} \quad (5.6)$$

enthält für jeden Knoten die Anzahl der Teilgebiete (Wertigkeit), zu denen er gehört (z.B. in Fig. 5.2 $R^{[k]} := 4$, $R^{[n]} = R^{[m]} = R^{[p]} = 2$, $R^{[q]} = 1$).

Vereinbarung: Im restlichen Abschnitt werden die hoch- und tiefgestellten Indizes wie folgt verwendet : $v_{C,i}^{[n]}$ bezeichnet die n -te Komponente (lokale oder globale Numerierung) eines Vektors \underline{r} abgespeichert im Prozeß \mathbb{P}_i . Der Index bezeichnet einen zum Interface gehörigen Teilvektor. Eine analoge Notation wird für die Matrizen verwandt.

5.1.1 Generierung der Steifigkeitsmatrix

Bei einer lokalen FEM, FDM, FVM Diskretisierung entsteht ganz natürlich eine verteilte Matrix.

Bemerkung 5.1. Bei Konvektionstermen in PDEs benötigen Upwindmethoden höherer Ordnung zusätzliche Information aus den Nachbarelementen (Kommunikation!).

5.1.2 Umwandlung der Vektortypen

Offensichtlich können Addition, Subtraktion und analoge Operationen mit Vektoren gleichen Typs ohne Kommunikation ausgeführt werden. Die Umwandlung eines verteilten in einen akkumulierten Vektor erfordert Kommunikation:

$$\boxed{\underline{\mathbf{w}}_i = A_i \sum_{s=1}^p A_s^T \underline{r}_s .} \quad (5.7)$$

Die andere Umwandlung eines akkumulierten in einen verteilten Vektor ist nicht eindeutig. Eine Möglichkeit besteht in der lokalen Division jeder Vektor-komponente durch die Anzahl der anliegenden Teilgebiete, z.B.

$$\boxed{\underline{r}_i = R_i^{-1} \underline{\mathbf{w}}_i} \quad (5.8)$$

mit $R_i = A_i R A_i^T$ und R in (5.6) definiert.

5.1.3 Skalarprodukt

Das innere Produkt von Vektoren verschiedener Vektortypen erfordert bzgl. der Kommunikation nur die Summation einer reellen Zahl :

$$\boxed{(\underline{\mathbf{w}}, \underline{r}) = \underline{\mathbf{w}}^T \underline{r} = \underline{\mathbf{w}}^T \sum_{i=1}^p A_i^T \underline{r}_i = \sum_{i=1}^p (A_i \underline{\mathbf{w}})^T \underline{r}_i = \sum_{i=1}^p (\underline{\mathbf{w}}_i, \underline{r}_i)}$$

(5.9)

Jede andere Vektorkombination benötigt eine Vektortypumwandlung mit evtl. Kommunikation.

5.1.4 Matrix-mal-Vektor Multiplikation

Im folgenden sehen wir uns die Matrix-mal-Vektor Multiplikationen von Vektoren und Matrizen der verschiedenen Typen an. Eine andere Darstellung findet hierzu man in [Gro 94].

1. Verteilte Matrix mal akkumulierter Vektor ergibt einen verteilten Vektor. Tatsächlich ergibt sich aus Definition (5.3) :

$$\mathbf{K} \cdot \underline{\mathbf{w}} = \sum_{i=1}^p A_i^T \mathbf{K}_i A_i \cdot \underline{\mathbf{w}} = \sum_{i=1}^p A_i^T \underbrace{\mathbf{K}_i \cdot \underline{\mathbf{w}}_i}_{\underline{\mathbf{r}}_i} = \underline{\mathbf{r}} \quad (5.10)$$

Die Ausführung der Summation (d.h. Kommunikation) führt zu einem akkumulierten Vektor.

2. Verteilte Matrix mal verteilter Vektor erfordert eine Vektorkonvertierung bevor obige Multiplikation ausgeführt werden kann.
3. Die Operation akkumulierte Matrix mal akkumulierter Vektor kann nicht mit beliebigen akkumulierten Matrizen \mathfrak{M} ausgeführt werden. Zur genaueren Analyse werfen wir einen Blick auf den Knoten n in Fig. 5.2 bei der Operation $\mathfrak{M} \cdot \underline{\mathbf{w}}$ und untersuchen die lokalen Multiplikationen $\underline{\mathbf{u}}_i = \mathfrak{M}_i \cdot \underline{\mathbf{w}}_i$ im Knoten n .

$$\begin{aligned} u_2^{[n]} &= \mathfrak{M}_2^{[n,n]} \mathbf{w}_2^{[n]} + \mathfrak{M}_2^{[n,m]} \mathbf{w}_2^{[m]} + \mathfrak{M}_2^{[n,k]} \mathbf{w}_2^{[k]} + \mathfrak{M}_2^{[n,p]} \mathbf{w}_2^{[p]} + \mathfrak{M}_2^{[n,s]} \mathbf{w}_2^{[s]} \\ u_4^{[n]} &= \mathfrak{M}_4^{[n,n]} \mathbf{w}_4^{[n]} + \mathfrak{M}_4^{[n,m]} \mathbf{w}_4^{[m]} + \mathfrak{M}_4^{[n,k]} \mathbf{w}_4^{[k]} + \mathfrak{M}_4^{[n,q]} \mathbf{w}_4^{[q]} + \mathfrak{M}_4^{[n,r]} \mathbf{w}_4^{[r]} \end{aligned}$$

In obigen Gleichungen unterscheiden sich die Terme 4 und 5 der rechten Seiten, so daß die Prozessoren 2 und 4 anstelle des eindeutigen Resultats zwei verschiedene besitzen. Das Ergebnis ist weder ein akkumulierter, noch ein verteilter Vektor. Der Grund dafür liegt im Informationstransport durch die Matrix von einem Prozessor a zu einem Knoten welcher zusätzlich auch noch zu Prozessor b gehört. In anderen Worten - der Transport von Information von einem Knoten i zu einem Knoten j ist nur dann erlaubt, wenn die Menge der Prozessoren, zu denen Knoten i gehört, eine Teilmenge der Prozessoren ist, welche Knoten j besitzen. Stellt man die Matrixeinträge als gerichteten Graphen dar, so heißt dies z.B., daß in Abb. 5.2 die Einträge $q \rightarrow k$, $q \rightarrow n$, $s \rightarrow n$, $s \rightarrow m$, $s \rightarrow p$, $p \rightarrow k$, $n \rightarrow k$, $p \rightarrow n$, $n \rightarrow p$, $\ell \rightarrow k$, $k \rightarrow \ell$ in der Matrix nicht erwünscht sind.

Ein analoges Resultat ist bei der Operation akkumulierte Matrix mal verteilter Vektor zu verzeichnen. Die Multiplikation einer akkumulierten Matrix \mathfrak{M} mit einem Vektors wird im nächsten Abschnitt detailliert untersucht.

5.2 Allgemeiner Ansatz für akkumulierte Matrizen

5.2.1 Knoten- und Teilgebietsmengen

Zur einfacheren Beschreibung des allgemeinen Ansatzes benötigen wir einige Definitionen.

Definition 5.3. Die Menge aller Teilgebiete, zu denen der Knoten $x^{[i]}$ gehört, wird mit

$$\sigma^{[i]} := \{s : x^{[i]} \in \overline{\Omega}_s\} \quad (5.11)$$

bezeichnet.

Die Setzung in (5.11) ist äquivalent zu

$$x^{[i]} \in \bigcap_{s \in \sigma^{[i]}} \overline{\Omega}_s . \quad (5.12)$$

Definition 5.4. Alle Knoten mit identischen Teilgebietsmengen σ werden in der Indexmenge

$$\omega(\sigma) := \{i \in \omega : \sigma^{[i]} = \sigma\} \quad (5.13)$$

zusammengefaßt. Desweiteren definiert man die Indexmengen

$$\underline{\omega}(\sigma) := \{i : \sigma^{[i]} \subseteq \sigma\} \quad (5.14)$$

$$\overline{\omega}(\sigma) := \{i : \sigma \subseteq \sigma^{[i]}\} . \quad (5.15)$$

Definition 5.5. $[\omega_s]$ Die Indexmenge $\omega_s := \overline{\omega}(\{s\})$ enthält alle Knoten/Unbekannte des Teilgebietes Ω_s . Wir bezeichnen mit $N_s := |\omega_s|$ die Anzahl der Knoten darin.

Die obige Definition impliziert sofort

$$\sigma^{[i]} = \{s : i \in \omega_s\} . \quad (5.16)$$

Bemerkung 5.2. Offensichtlich gelten die Relationen

$$i \in \omega_s \quad \Leftrightarrow \quad s \in \sigma^{[i]} \quad \Leftrightarrow \quad \{s\} \subseteq \sigma^{[i]} \quad (5.17)$$

$$j \notin \omega_s \quad \Leftrightarrow \quad s \notin \sigma^{[j]} \quad \Leftrightarrow \quad \{s\} \not\subseteq \sigma^{[j]}. \quad (5.18)$$

Zur Illustration dieser Definitionen betrachten wir für die Gebietsaufteilung in Fig. 5.4 einige Mengen.

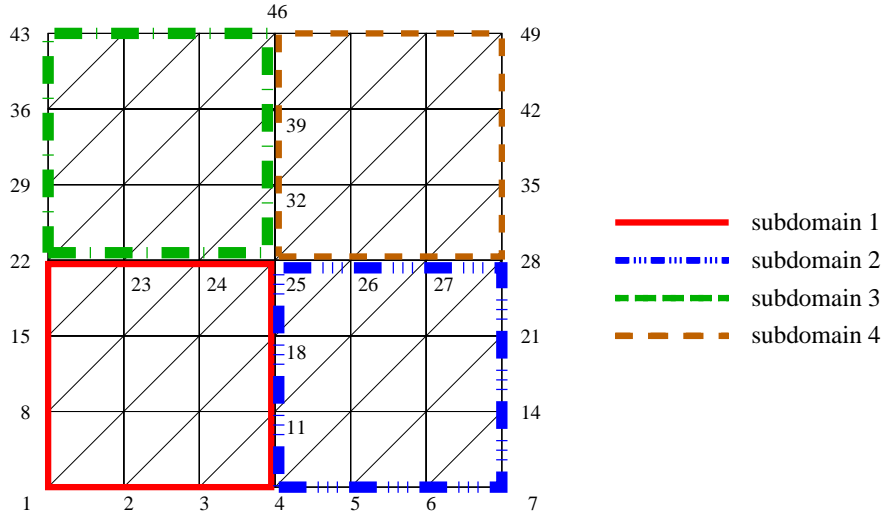


Fig. 5.4 Vier nichtüberlappende Teilgebiete mit Vernetzung und zeilenweiser Numerierung

Seien alle identischen Teilgebietenmengen mittels $\sigma^{[i]}$ über einen repräsentativen Knoten i dargestellt, dann erhalten wir:

$$\begin{aligned} \sigma^{[1]} &= \{1\}, & \sigma^{[7]} &= \{2\}, & \sigma^{[43]} &= \{3\}, & \sigma^{[49]} &= \{4\}, \\ \sigma^{[4]} &= \{1, 2\}, & \sigma^{[22]} &= \{1, 3\}, & \sigma^{[28]} &= \{2, 4\}, & \sigma^{[46]} &= \{3, 4\}, \\ \sigma^{[25]} &= \{1, 2, 3, 4\}. \end{aligned}$$

Einige typische Indexmengen sind:

$$\begin{aligned} \omega(\sigma^{[1]}) &= \omega(\{1\}) &= \{1, 2, 3, 8, 9, 10, 15, 16, 17\}, \\ \omega(\sigma^{[7]}) &= \omega(\{2\}) &= \{5, 6, 7, 12, 13, 14, 19, 20, 21\}, \\ \omega(\sigma^{[4]}) &= \omega(\{1, 2\}) &= \{4, 11, 18\}, \\ \omega(\sigma^{[22]}) & &= \{22, 23, 24\}, \\ \omega(\sigma^{[28]}) & &= \{26, 27, 28\}, \\ \omega(\sigma^{[25]}) & &= \{25\}. \end{aligned}$$

Die folgenden Indexmengen wurden entsprechend (5.14) und (5.15) abgeleitet.

$$\begin{aligned}
\underline{\omega}(\sigma^{[1]}) &= \omega(\sigma^{[1]}), \\
\overline{\omega}(\sigma^{[1]}) &= \omega(\sigma^{[1]}) \cup \omega(\sigma^{[4]}) \cup \omega(\sigma^{[22]}) \cup \omega(\sigma^{[25]}) \\
&= \{\overline{1, 4, 8, 11, 15, 18, 22, 25}\} &= \overline{\omega}(\{1\}), \\
\underline{\omega}(\sigma^{[4]}) &= \omega(\sigma^{[4]}) \cup \omega(\{1\}) \cup \omega(\{2\}) &= \overline{\{1, 21\}}, \\
\overline{\omega}(\sigma^{[4]}) &= \omega(\sigma^{[4]}) \cup \omega(\sigma^{[25]}) &= \{4, 11, 18, 25\}, \\
\underline{\omega}(\sigma^{[25]}) & &= \overline{\{1, 49\}}, \\
\overline{\omega}(\sigma^{[25]}) &= \omega(\sigma^{[25]}) &= \{25\}.
\end{aligned}$$

5.2.2 Matrix-Vektor Produkt

Die Multiplikation einer akkumulierten Matrix \mathfrak{M} mit einem Vektor erfordert detaillierte Untersuchungen. Die folgenden zwei Sätze geben Bedingungen an das Matrixmuster an, sodaß die entsprechenden Matrix-Vektor Produkte ohne Kommunikation ausgeführt werden können.

Satz 5.1. *Genau dann, wenn für sämtliche Matrixeinträge*

$$\forall i, j \in \omega : \quad \sigma^{[i]} \not\subseteq \sigma^{[j]} \implies \mathfrak{M}^{[i,j]} = 0 \quad (5.19)$$

gilt, ergibt die Multiplikation einer akkumulierten Matrix mit einem akkumulierten Vektor wiederum einen akkumulierten Vektor

$$\underline{\mathbf{w}} = \mathfrak{M} \cdot \underline{\mathbf{u}} \quad (5.20)$$

und kann parallel ausgeführt werden, d.h.,

$$\underline{\mathbf{w}}_s = \mathfrak{M}_s \cdot \underline{\mathbf{u}}_s \quad \forall s = \overline{1, P} . \quad (5.21)$$

Beweis. Wir beginnen mit der Definition eines akkumulierten Vektors (5.1) und untersuchen komponentenweise den Ergebnisvektor in globaler Numerierung.

$$\begin{aligned}
& \underline{\mathbf{w}}_s = A_s \underline{\mathbf{w}} & \forall s = \overline{1, p} \\
\stackrel{(5.21)}{\iff} & \mathfrak{M}_s \underline{\mathbf{u}}_s = A_s \mathfrak{M} \underline{\mathbf{u}} & \forall s = \overline{1, p} \\
\stackrel{(5.5)}{\iff} & (A_s \mathfrak{M} A_s^T \cdot \underline{\mathbf{u}}_s)^{[i]} = (A_s \mathfrak{M} \cdot \underline{\mathbf{u}})^{[i]} & \forall s = \overline{1, p}, \quad \forall i \in \omega_s \\
\iff & \sum_{j \in \omega_s} \mathfrak{M}^{[i,j]} \underline{\mathbf{u}}^{[j]} = \sum_{j \in \omega} \mathfrak{M}^{[i,j]} \underline{\mathbf{u}}^{[j]} & \forall s = \overline{1, p}, \quad \forall i \in \omega_s
\end{aligned}$$

Die letzte Zeile ist genau dann wahr, wenn die folgenden Aussagen wahr sind.

$$\begin{array}{llll}
\forall i \in \omega_s & \forall s = \overline{1, p} : & j \notin \omega_s & \implies \mathfrak{M}^{[i,j]} = 0 \\
\stackrel{(5.17), (5.18)}{\iff} & \forall s = \overline{1, p} : & \{s\} \in \sigma^{[i]} \wedge \{s\} \notin \sigma^{[j]} & \implies \mathfrak{M}^{[i,j]} = 0 \\
& \iff & \sigma^{[i]} \not\subseteq \sigma^{[j]} & \implies \mathfrak{M}^{[i,j]} = 0
\end{array}$$

■

Satz 5.2. Genau dann, wenn die Matrixeinträge

$$\forall i, j \in \omega : \quad \sigma^{[i]} \not\subseteq \sigma^{[j]} \implies \mathfrak{M}^{[i,j]} = 0 \quad (5.22)$$

erfüllen, ergibt die Multiplikation einer akkumulierten Matrix mit einem verteilten Vektor wiederum einen verteilten Vektor

$$\underline{\mathbf{r}} = \mathfrak{M} \cdot \underline{\mathbf{v}} \quad (5.23)$$

und kann parallel ausgeführt werden, d.h.,

$$\underline{\mathbf{r}}_s = \mathfrak{M}_s \cdot \underline{\mathbf{v}}_s \quad \forall s = \overline{1, P} . \quad (5.24)$$

Beweis. Wir beginnen mit der Definition eines verteilten Vektors (5.2) und untersuchen unter Nutzung der Beziehung $A_s^T \cdot A_{s \times N \times N} = \text{diag}_{i \in \omega} \begin{cases} 1 & \text{iff } i \in \omega_s \\ 0 & \text{iff } i \notin \omega_s \end{cases}$ komponentenweise den Ergebnisvektor in globaler Numerierung.

$$\begin{aligned}
& \sum_{s=1}^P A_s^T \underline{\mathbf{r}}_s = \underline{\mathbf{r}} \\
& \sum_{s=1}^P A_s^T A_s \mathfrak{M} A_s^T \underline{\mathbf{v}}_s \stackrel{(5.5)}{=} \sum_{s=1}^P A_s^T \mathfrak{M}_s \cdot \underline{\mathbf{v}}_s = \mathfrak{M} \cdot \underline{\mathbf{v}} = \mathfrak{M} \cdot \sum_{s=1}^P A_s^T \underline{\mathbf{v}}_s \\
& \left(\begin{array}{l} \sum_{s=1}^P \sum_{j \in \omega_s} \mathfrak{M}^{[i,j]} \cdot \mathbf{v}_s^{[j]}, i \in \omega_s \\ 0, i \notin \omega_s \end{array} \right) = \left(\sum_{j \in \omega} \mathfrak{M}^{[i,j]} \cdot \sum_{s=1}^P \mathbf{v}_s^{[j]}, i \in \omega \right) \\
& \left(\begin{array}{l} \sum_{j \in \omega} \sum_{s \in \sigma^{[j]}} \mathfrak{M}^{[i,j]} \cdot \mathbf{v}_s^{[j]}, i \in \omega_s \\ 0, i \notin \omega_s \end{array} \right) = \left(\sum_{j \in \omega} \sum_{s=1}^P \mathfrak{M}^{[i,j]} \cdot \mathbf{v}_s^{[j]}, i \in \omega \right)
\end{aligned}$$

Die letzte Zeile ist genau dann wahr, wenn gilt

$$\begin{array}{ll}
s \in \sigma^{[j]} \wedge i \notin \omega_s & \implies \mathfrak{M}^{[i,j]} = 0 \\
\{s\} \subseteq \sigma^{[j]} \wedge \{s\} \notin \sigma^{[i]} & \implies \mathfrak{M}^{[i,j]} = 0 \\
\sigma^{[j]} \not\subseteq \sigma^{[i]} & \implies \mathfrak{M}^{[i,j]} = 0 .
\end{array}$$

■

Offensichtlich erfüllt \mathfrak{M}^T Bedingung (5.22), falls für die Matrix \mathfrak{M} die Relation (5.19) erfüllt ist.

Bemerkung 5.3. Zwischenschritte in den Beweisen der Sätze 5.1. und 5.2. führend direkt zu äquivalenten Formulierungen für (5.19).

$$\mathfrak{M}A_s^T A_s = \mathfrak{M} \quad \forall s = \overline{1, P} \quad (5.25)$$

und für (5.22)

$$A_s^T A_s \mathfrak{M} = \mathfrak{M} \quad \forall s = \overline{1, P} . \quad (5.26)$$

Die Beziehungen (5.25) und (5.26) wurden bereits von Groh [Gro 94] für eine nichtüberlappende Elementverteilung hergeleitet, um (5.20) bzw. (5.23) parallel ausführen zu können.

Bemerkung 5.4. Die Teilgebetsmengen σ induzieren eine Blockstruktur der Unbekannten derart, daß alle Knoten mit derselben Teilgebetsmenge wie der repräsentative Knoten k einen Block bilden. Nunmehr folgt direkt aus den Sätzen 5.1. und 5.2., daß eine akkumulierte Matrix \mathfrak{M} genau dann sowohl auf (5.20) als auch auf (5.23) angewandt werden kann, wenn \mathfrak{M} eine von der Blockstruktur der Knoten abgeleitete Blockdiagonalstruktur besitzt. Wir schreiben

$$\mathfrak{M} = \text{blockdiag} \{ \mathfrak{M}_k \} \quad \text{mit} \quad \mathfrak{M}_k = \{ \mathfrak{M}^{[i,j]} \}_{i,j \in \omega(\sigma^{[k]})} . \quad (5.27)$$

Bemerkung 5.5. Falls für \mathfrak{M} (5.27) gilt, dann besitzt auch die Inverse \mathfrak{M}^{-1} die entsprechende Blockstruktur an kann parallel auf (5.20) und (5.23) angewandt werden.

Satz 5.3. Falls die akkumulierte Matrix \mathfrak{P} die Bedingung (5.19) an das Besetzungsmuster erfüllt, dann ergeben die Matrixmultiplikationen $\mathfrak{P}^T \cdot \mathfrak{K} \cdot \mathfrak{P}$ eine verteilte Matrix \mathfrak{K}^H , welche lokal und völlig parallel berechnet werden kann.

$$\mathfrak{K}^H = \sum_{s=1}^P A_s^T \mathfrak{K}_s^H A_s = \sum_{s=1}^P A_s^T (\mathfrak{P}_s^T \cdot \mathfrak{K}_s \cdot \mathfrak{P}_s) A_s = \mathfrak{P}^T \cdot \mathfrak{K} \cdot \mathfrak{P} . \quad (5.28)$$

Beweis. Die Bedingungen (5.19) und (5.22) an das Matrixmuster sind gleichwertig zu den Beziehungen (5.25) und (5.26), die für alle Teilgebiete erfüllt sein müssen. Daher ergibt sich

$$\begin{aligned}
\mathfrak{P}^T \cdot \mathfrak{K} \cdot \mathfrak{P} &\stackrel{(5.3)}{=} \mathfrak{P}^T \cdot \sum_{s=1}^P A_s^T \mathfrak{K}_s A_s \cdot \mathfrak{P} = \sum_{s=1}^P \mathfrak{P}^T A_s^T \mathfrak{K}_s A_s \mathfrak{P} \\
&\stackrel{(5.25)(5.26)}{=} \sum_{s=1}^P A_s^T A_s \mathfrak{P}^T A_s^T \mathfrak{K}_s A_s \mathfrak{P} A_s^T A_s \\
&= \sum_{s=1}^P A_s^T (\mathfrak{P}_s^T \cdot \mathfrak{K}_s \cdot \mathfrak{P}_s) A_s \\
&=: \sum_{s=1}^P A_s^T \mathfrak{K}_s^H A_s = K^H
\end{aligned}$$

■

5.2.3 Anwendungen der Sätze 5.1. und 5.2.

Wir wenden die Resultate der Sätze 5.1. und 5.2. auf die Datenaufteilung in Fig. 5.4 an. Darin gibt es 9 verschiedene Teilgebetsmengen, welche wir nach der Anzahl der darin enthaltenen Elemente ordnen.

$$\begin{aligned}
\sigma_V &= \sigma^{[25]}, \\
\sigma_{E_1} &= \sigma^{[4]}, & \sigma_{E_2} &= \sigma^{[22]}, & \sigma_{E_3} &= \sigma^{[28]}, & \sigma_{E_4} &= \sigma^{[46]}, \\
\sigma_{I_1} &= \sigma^{[1]}, & \sigma_{I_2} &= \sigma^{[7]}, & \sigma_{I_3} &= \sigma^{[8]}, & \sigma_{I_4} &= \sigma^{[49]}.
\end{aligned}$$

Von obiger Blockstruktur leiten sich die entsprechenden Blockstrukturen der Matrizen und Vektoren ab. Ein bzgl. (5.19) zulässiges Besetztheitsmuster einer akkumulierten Matrix ist dann

$$\mathfrak{M} = \begin{pmatrix}
\mathfrak{M}_V & & & & & & & & & \\
\mathfrak{M}_{E_1V} & \mathfrak{M}_{E_1} & & & & & & & & \\
\mathfrak{M}_{E_2V} & 0 & \mathfrak{M}_{E_2} & & & & & & & \\
\mathfrak{M}_{E_3V} & 0 & 0 & \mathfrak{M}_{E_3} & & & & 0 & & \\
\mathfrak{M}_{E_4V} & 0 & 0 & 0 & \mathfrak{M}_{E_4} & & & & & \\
\mathfrak{M}_{I_1V} & \mathfrak{M}_{I_1E_1} & \mathfrak{M}_{I_1E_2} & 0 & 0 & \mathfrak{M}_{I_1} & & & & \\
\mathfrak{M}_{I_2V} & \mathfrak{M}_{I_2E_1} & 0 & \mathfrak{M}_{I_2E_3} & 0 & 0 & \mathfrak{M}_{I_2} & & & \\
\mathfrak{M}_{I_3V} & 0 & \mathfrak{M}_{I_3E_2} & 0 & \mathfrak{M}_{I_3E_4} & 0 & 0 & \mathfrak{M}_{I_3} & & \\
\mathfrak{M}_{I_4V} & 0 & 0 & \mathfrak{M}_{I_4E_3} & \mathfrak{M}_{I_4E_4} & 0 & 0 & 0 & \mathfrak{M}_{I_4} &
\end{pmatrix}.$$

(5.29)

Dieses Besetztheitsmuster kann verbal durch die folgenden Bedingungen beschrieben werden:

- (a) Keine Verbindung zwischen Crosspoints, welche zu verschiedenen Mengen von Teilgebieten gehören.
- (b) Keine Verbindung zwischen Kanten, welche zu verschiedenen Mengen von Teilgebieten gehören.

Stellt man die Matrixeinträge als gerichteten Graphen dar, so heißt dies z.B., daß in Fig. 5.2 die Einträge $q \rightarrow k$, $q \rightarrow n$, $s \rightarrow n$, $s \rightarrow m$, $s \rightarrow p$, $p \rightarrow k$, $n \rightarrow k$, $p \rightarrow n$, $n \rightarrow p$, $\ell \rightarrow k$, $k \rightarrow \ell$ in der Matrix nicht erwünscht sind.

Bedingung (a) kann einfach dadurch erreicht werden, daß auf jeder Gebietskante mindestens ein zusätzlicher Kantenknoten plaziert wird (Achtung bei Verbindungen quer durch das Teilgebiet). Ein 2D Netzgenerators für Dreiecke mit linearen Ansatzfunktionen kann leicht so modifiziert werden, daß Bedingung (b) eingehalten wird, z.B. wird die Kante zwischen den Knoten p und n vermieden. Fig. 5.5 zeigt das geänderte Netz.

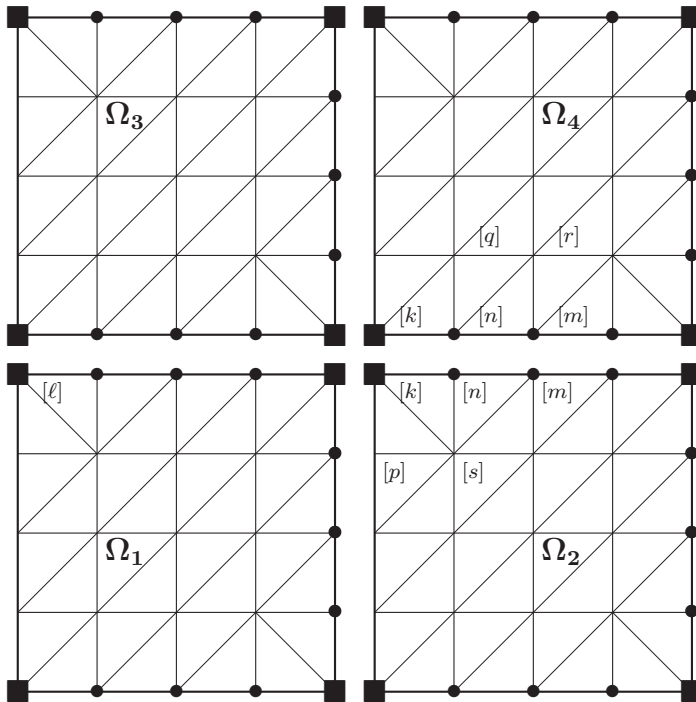


Fig. 5.5 Nichtüberlappende Elemente mit Diskretisierung bei angepaßtem Netz.

- Dieses Vorgehen ist nur bei bestimmten finiten Elementen möglich!

- Bei Verwendung von Tetraedern mit linearen Ansatzfunktionen kann Bedingung 2 auch in 3D Netzgeneratoren eingebaut werden. In kompakterer Schreibweise ergibt dies

$$\mathfrak{M} = \begin{pmatrix} \mathfrak{M}_V & 0 & 0 \\ \mathfrak{M}_{EV} & \mathfrak{M}_E & 0 \\ \mathfrak{M}_{IV} & \mathfrak{M}_{IE} & \mathfrak{M}_I \end{pmatrix} \implies \underline{\mathbf{u}} = \mathfrak{M} \cdot \underline{\mathbf{w}} \quad (5.30)$$

und erfordert die (5.29) entsprechenden Blockstrukturen der Teilmatrizen. Speziell müssen \mathfrak{M}_V , \mathfrak{M}_E und \mathfrak{M}_I Blockdiagonalmatrizen sein, sodaß wir nach Bemerkung 5.5. auch deren Inverse anwenden können.

Das entsprechende transponierte Matrixmuster führt direkt auf

$$\mathfrak{M} = \begin{pmatrix} \mathfrak{M}_V & \mathfrak{M}_{VE} & \mathfrak{M}_{VI} \\ 0 & \mathfrak{M}_E & \mathfrak{M}_{EI} \\ 0 & 0 & \mathfrak{M}_I \end{pmatrix} \implies \underline{\mathbf{f}} = \sum_{i=1}^P A_i^T \underline{\mathbf{f}}_i = \sum_{i=1}^P A_i^T (\mathfrak{M}_i \underline{\mathbf{r}}_i) \quad (5.31)$$

mit Blockdiagonalmatrizen \mathfrak{M}_I , \mathfrak{M}_E , \mathfrak{M}_V . Praktisch heißt dies, z.B., daß in Fig.5.5 die Matrixverbindungen $n \rightarrow k$ und $q \rightarrow k$ erlaubt sind, nicht aber die umgekehrte Richtung.

Bemerkung 5.6. Die Gleichungen (5.30) und (5.31) können unter der Bedingung, daß \mathfrak{M}_V , \mathfrak{M}_E , \mathfrak{M}_I blockdiagonal sind (durch das Netz in Fig. 5.5 garantiert) kombiniert werden. Bezeichnen \mathfrak{M}_L , \mathfrak{M}_U und \mathfrak{M}_D das strikt untere bzw. strikt obere Dreiecksmatrix und die Diagonale von \mathfrak{M} , kann die akkumulierte Matrix-mal-Vektor Operation für alle Vektortypen aufgeschrieben werden.

$$\underline{\mathbf{w}} = \mathfrak{M} \cdot \underline{\mathbf{u}} := (\mathfrak{M}_L + \mathfrak{M}_D) \cdot \underline{\mathbf{u}} + \sum_{i=1}^P A_i^T \mathfrak{M}_{U,i} R_i^{-1} \cdot \underline{\mathbf{u}}_i \quad (5.32a)$$

$$\underline{\mathbf{w}} = \mathfrak{M} \cdot \underline{\mathbf{r}} := (\mathfrak{M}_L + \mathfrak{M}_D) \sum_{i=1}^P A_i^T \cdot \underline{\mathbf{r}}_i + \sum_{i=1}^P A_i^T \mathfrak{M}_{U,i} \cdot \underline{\mathbf{r}}_i \quad (5.32b)$$

$$\underline{\mathbf{f}} = \mathfrak{M} \cdot \underline{\mathbf{u}} := R^{-1}(\mathfrak{M}_L + \mathfrak{M}_D) \cdot \underline{\mathbf{u}} + \mathfrak{M}_U R^{-1} \cdot \underline{\mathbf{u}} \quad (5.32c)$$

$$\underline{\mathbf{f}} = \mathfrak{M} \cdot \underline{\mathbf{r}} := R^{-1}(\mathfrak{M}_L + \mathfrak{M}_D) \sum_{i=1}^P A_i^T \cdot \underline{\mathbf{r}}_i + \mathfrak{M}_U \cdot \underline{\mathbf{r}} \quad (5.32d)$$

Jede dieser Multiplikationen benötigt genau zwei Vektortypumwandlungen, jedoch beeinflußt die konkrete Wahl der Vektortypen die Anzahl der Kommunikationsschritte.

Bemerkung 5.7. Die Faktorisierung der Matrix \mathfrak{M} in eine obere und untere Dreiecksmatrix \mathfrak{L}^{-1} und \mathfrak{U}^{-1} , die den Bedingungen (5.19) und (5.22) genügen müssen, kann auf zwei Weisen geschehen. Zum einem gilt

$$\underline{\mathbf{w}} = \mathfrak{L}^{-1}\mathfrak{U}^{-1} \cdot \underline{\mathbf{r}} := \mathfrak{L}^{-1} \sum_{i=1}^P A_i^T \mathfrak{U}_i^{-1} \cdot \underline{\mathbf{r}}_i \quad (5.33a)$$

$$\underline{\mathbf{f}} = \mathfrak{L}^{-1}\mathfrak{U}^{-1} \cdot \underline{\mathbf{u}} := R^{-1} \mathfrak{L}^{-1} \sum_{i=1}^P A_i^T \mathfrak{U}_i^{-1} R^{-1} \cdot \underline{\mathbf{u}}_i, \quad (5.33b)$$

und andererseits

$$\underline{\mathbf{w}} = \mathfrak{U}^{-1}\mathfrak{L}^{-1} \cdot \underline{\mathbf{r}} := \sum_{i=1}^P A_i^T \mathfrak{U}_i^{-1} R_i^{-1} A_i \mathfrak{L}^{-1} \cdot \left(\sum_{j=1}^P A_j^T \underline{\mathbf{r}}_j \right) \quad (5.34a)$$

$$\underline{\mathbf{f}} = \mathfrak{U}^{-1}\mathfrak{L}^{-1} \cdot \underline{\mathbf{u}} := \mathfrak{U}^{-1} R^{-1} \mathfrak{L}^{-1} \cdot \underline{\mathbf{u}}. \quad (5.34b)$$

Die Gleichungen für die restlichen Vektortypenpaare können leicht mittels Typkonvertierung erhalten werden. Bei der Faktorisierung beeinflusst die konkrete Wahl der Vektortypen die Anzahl und Art der Konvertierungen.

5.3 Überlappende Elemente

Das in Abschnitt 5.1 vorgestellte Konzept der akkumulierten und verteilten Vektor- bzw. Matrixtypen läßt sich auch auf eine Datenaufteilung basierend auf überlappenden Elementen übertragen.

Die Abbildungsmatrix A_s bilde wiederum die globale Knotennumerierung auf die lokale Numerierung im Teilgebiet Ω_s ab. Dann erfolgt die Definition der beiden Vektortypen entsprechend (5.1) und (5.2) mit den dort beschriebenen Eigenschaften.

Damit das Konzept hier anwendbar ist, muß sich die globale Steifigkeitsmatrix K^{FEM} aus der globalen Akkumulation *geeignet gewählter* lokaler Matrizen K_s ergeben, d.h. es wird

$$\mathfrak{K}^{\text{FEM}} = K^{\text{FEM}} \stackrel{!}{=} K = \sum_{s=1}^P A_s^T K_s A_s$$

gefordert. Wie müssen die K_s gewählt werden?

Zur Veranschaulichung betrachten wir folgende Volumenintegrale, welche ele-

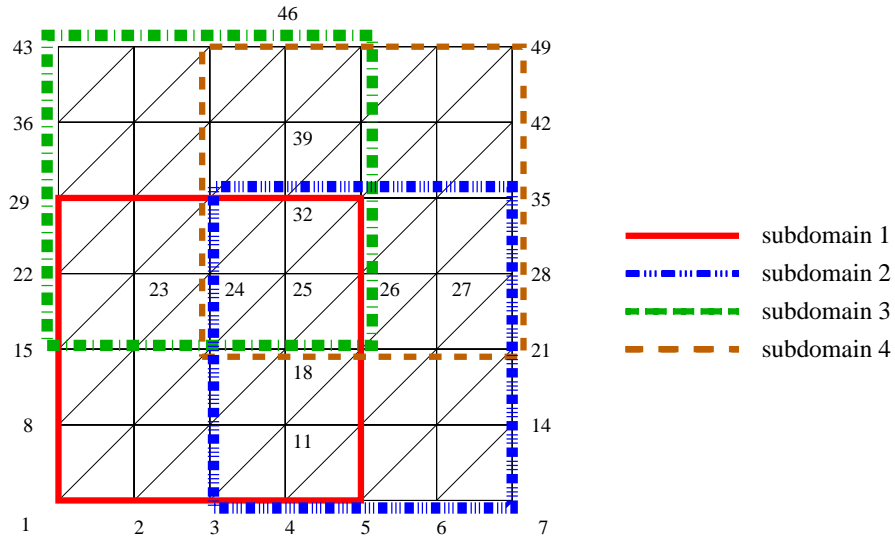


Fig. 5.6 Vier überlappende Teilgebiete mit Vernetzung und zeilenweiser Numerierung

mentweise in den Elementen $\delta^{(r)}$ berechnet werden.

$$V = |\Omega| = \sum_{\delta^{(r)} \subset \Omega} \int_{\delta^{(r)}} dx = \sum_{\delta^{(r)} \subset \Omega} |\delta^{(r)}|$$

$$V_s = |\Omega_s| = \sum_{\delta^{(r)} \subset \Omega_s} |\delta^{(r)}|$$

Im Falle zweier überlappender Teilgebiete ($\Omega_1 \cap \Omega_2 \neq \emptyset$) erhalten wir

$$\begin{aligned} V_1 + V_2 &= \sum_{\delta^{(r)} \subset \Omega_1} |\delta^{(r)}| + \sum_{\delta^{(r)} \subset \Omega_2} |\delta^{(r)}| \\ &= \sum_{\delta^{(r)} \subset \Omega_1 \setminus \Omega_2} |\delta^{(r)}| + \sum_{\delta^{(r)} \subset \Omega_2 \setminus \Omega_1} |\delta^{(r)}| + \mathbf{2} \sum_{\delta^{(r)} \subset \Omega_1 \cap \Omega_2} |\delta^{(r)}| \\ &= \sum_{\delta^{(r)} \subset \Omega_1 \cup \Omega_2} |\delta^{(r)}| + \sum_{\delta^{(r)} \subset \Omega_1 \cap \Omega_2} |\delta^{(r)}| \\ &= V + \sum_{\delta^{(r)} \subset \Omega_1 \cap \Omega_2} |\delta^{(r)}|, \end{aligned}$$

d.h. die Summe der Einzelvolumina übersteigt das Gesamtvolumen.

Dieses Problem läßt sich nur durch Einführung eines Wichtungsfaktors $\frac{1}{W^{(r)}}$

beheben. Für obigen Fall sichert die Setzung

$$W^{(r)} := \begin{cases} 1 & : \delta^{(r)} \subset \Omega_1 \setminus \Omega_2 \\ 2 & : \delta^{(r)} \subset \Omega_1 \cap \Omega_2 \\ 1 & : \delta^{(r)} \subset \Omega_2 \setminus \Omega_1 \end{cases}$$

das gewünschte Ergebnis

$$\begin{aligned} V_1' + V_2' &= \sum_{\delta^{(r)} \subset \Omega_1} \frac{1}{W^{(r)}} |\delta^{(r)}| + \sum_{\delta^{(r)} \subset \Omega_2} \frac{1}{W^{(r)}} |\delta^{(r)}| \\ &= \sum_{\delta^{(r)} \subset \Omega_1 \setminus \Omega_2} \frac{1}{1} |\delta^{(r)}| + \sum_{\delta^{(r)} \subset \Omega_2 \setminus \Omega_1} \frac{1}{1} |\delta^{(r)}| + \mathbf{2} \sum_{\delta^{(r)} \subset \Omega_1 \cap \Omega_2} \frac{1}{2} |\delta^{(r)}| \\ &= \sum_{\delta^{(r)} \subset \Omega_1 \cup \Omega_2} \frac{1}{1} |\delta^{(r)}| = V . \end{aligned}$$

Im allgemeinen Fall stellt $W^{(r)}$ die Anzahl der Teilgebiete dar, zu denen ein Element δ^r gehört und es gilt mit $\sigma^{[i]}$ aus (5.11) die Beziehung

$$W^{(r)} = \left| \bigcap_{x_i \in \bar{\delta}^{(r)}} \sigma^{[i]} \right| , \quad (5.35)$$

d.h. die Wertigkeit des Elementes $\delta^{(r)}$ ist gleich der Anzahl der Teilgebiete, welche alle Elementknoten enthalten.

Bemerkung 5.8. Führt man analog zur Abbildungsmatrix der Knoten A_s eine Matrix A_s^e ein, welche die globale Elementnumerierung auf die lokale abbildet, so gilt analog zu (5.6)

$$R^e = \text{diag}\{W^{(r)}\} = \sum_{s=1}^P (A_s^e)^T \cdot A_s^e . \quad (5.36)$$

Ersetzt man nunmehr das Volumenintegral durch die Bilinearform des Laplaceoperators $\int \nabla^T \varphi_i \cdot \nabla \varphi_j dx$, dann gilt in globaler Numerierung $\forall i, j = \overline{1, N}$

$$\begin{aligned} K_{ij} &= \int_{\Omega} \nabla^T \varphi_i \cdot \nabla \varphi_j dx = \sum_{\delta^{(r)} \subset \Omega} \int_{\delta^{(r)}} \nabla^T \varphi_i \cdot \nabla \varphi_j dx = \sum_{\delta^{(r)} \subset \Omega} K_{ij}^{(r)} \\ &= \sum_{\delta^{(r)} \subset \Omega_1} \frac{1}{W^{(r)}} K_{ij}^{(r)} + \sum_{\delta^{(r)} \subset \Omega_2} \frac{1}{W^{(r)}} K_{ij}^{(r)} + \dots \\ &= \sum_{s=1}^P \sum_{\delta^{(r)} \subset \Omega_s} \frac{1}{W^{(r)}} K_{ij}^{(r)} , \end{aligned}$$

und somit allgemein für die Steifigkeitsmatrix K^{FEM} :

$$K^{\text{FEM}} = \sum_{s=1}^P A_s^T \cdot \underbrace{\sum_{\delta^{(r)} \subset \Omega_s} \frac{1}{W^{(r)}} K^{(r)}}_{=: K_s} \cdot A_s \ .$$

Satz 5.4. *Es liege eine Datenaufteilung mit überlappenden Elementen vor und jedem Element $\delta^{(r)}$ wird ein Gewicht $W^{(r)}$ entsprechend Gleichung (5.35) zugeordnet. Werden desweiteren die lokalen Matrizen*

$$K_s := \sum_{\delta^{(r)} \subset \Omega_s} \frac{1}{W^{(r)}} K^{(r)} \quad (5.37)$$

als lokale Assemblierung der gewichteten Elementmatrizen $K^{(r)}$ berechnet, dann gilt für die damit global akkumulierte Matrix

$$\mathfrak{K}^{\text{FEM}} \equiv \mathbf{K} = \sum_{s=1}^P A_s^T \cdot K_s \cdot A_s \ . \quad (5.38)$$

Die Herleitung enthält den Beweis von Satz 5.4..

Bemerkung 5.9. Die Datenaufteilung der nichtüberlappenden Elemente ist mit $W^{(r)} = 1$, $\forall r$ in (5.37) und (5.38) enthalten.

Bemerkung 5.10. Für alle Größen, welche *direkt* über Volumenintegrale berechnet werden, muß analog zu (5.37) vorgegangen werden. Dies heißt im konkreten Falle des Lastvektors

$$\underline{\mathbf{f}} = \sum_{s=1}^P A_s^T \cdot \underline{\mathbf{f}}_s := \sum_{s=1}^P A_s^T \cdot \sum_{\delta^{(r)} \subset \Omega_s} \frac{1}{W^{(r)}} \underline{\mathbf{f}}^{(r)} \ , \quad (5.39)$$

daß die Last elementweise gewichtet und lokal assembliert wird.

Bemerkung 5.11. Die Teilgebietsmengen $\sigma^{[i]}$ der Datenaufteilung in Fig. 5.6 werden genauso repräsentiert wie die Datenaufteilungen Fig. 5.4 und ergeben analoge Mengen. Die entsprechenden Indexmengen sind nunmehr:

$$\begin{aligned} \omega(\sigma^{[1]}) &= \{1, 2, 8, 9\}, & \omega(\sigma^{[7]}) &= \{6, 7, 13, 14\}, \\ \omega(\sigma^{[43]}) &= \{36, 37, 43, 44\}, & \omega(\sigma^{[49]}) &= \{41, 42, 48, 49\}, \\ \omega(\sigma^{[4]}) &= \{3, 4, 5, 10, 11, 12\}, & \omega(\sigma^{[22]}) &= \{15, 16, 22, 23, 29, 30\}, \\ \omega(\sigma^{[28]}) &= \{20, 21, 27, 28, 34, 35\}, & \omega(\sigma^{[46]}) &= \{38, 39, 40, 45, 46, 47\}, \\ \omega(\sigma^{[25]}) &= \{17, 18, 19, 24, 25, 26, 31, 32, 33\}, \end{aligned}$$

und es folgen die abgeleiteten Indexmengen:

$$\begin{aligned}\underline{\omega}(\sigma^{[1]}) &= \omega(\sigma^{[1]}), \\ \overline{\omega}(\sigma^{[1]}) &= \{\overline{1, 5, 8, 12, 15, 19, 22, 25, 29, 33}\}, \\ \underline{\omega}(\sigma^{[4]}) &= \{\overline{1, 14}\}, \\ \overline{\omega}(\sigma^{[4]}) &= \{3, 4, 5, 10, 11, 12, 17, 18, 19, 24, 25, 26, 31, 32, 33\}, \\ \underline{\omega}(\sigma^{[25]}) &= \{\overline{1, 49}\}, \\ \overline{\omega}(\sigma^{[25]}) &= \omega(\sigma^{[25]}).\end{aligned}$$

Damit lassen sich sämtliche Ergebnisse aus Abschnitt 5.2 auch auf eine Datenaufteilung mit überlappenden Elementen übertragen!

6 Vektorisierung und Parallelisierung iterativer Verfahren

In diesem Kapitel werden lineare Gleichungssysteme der Form

$$K_{n \times n} \cdot \underline{x} = \underline{b} \tag{6.1}$$

mittels iterativer Verfahren gelöst. Die Matrix K und somit obiges Gleichungssystem soll aus der Diskretisierung eines Differentialoperators entstanden sein. Bei Verwendung von FEM (Finite Element Methode), FDM (Finite Differenzen Methode) oder FVM (Finite Volumen Methode) entsteht eine dünnbesetzte (sparse) Matrix K , welche als Grundlage für die folgenden Untersuchungen stehen soll.

Im parallelen Fall wird auf Abschnitt 5 und speziell auf die **nichtüberlappende Elementverteilung** 5.1 Bezug genommen. Die entsprechenden Vektor- und Matrixtypen werden analog gekennzeichnet.

Andere Arten der Datenverteilung in der Parallelisierung werden explizit ausgewiesen.

6.1 Das CG-Verfahren

6.1.1 Das serielle Verfahren

Falls die Matrix K aus (6.1) symmetrisch (d.h. $K = K^T$) und positiv definit (d.h. $(K\underline{v}, \underline{v})_{L_2} > 0$) ist, kann zum Lösen von (6.1) das CG-Verfahren genutzt werden. Üblicherweise wird es in Verbindung mit einer Vorkonditionierung C genutzt, was in Alg. 6.1 dargestellt ist.

Eine Vektorisierung des cg an sich ist trivial, jedoch kann es bei der Vorkonditionierung Probleme geben.

```

Wähle  $\underline{u}^0$ 
 $\underline{r} := \underline{f} - K \cdot \underline{u}^0$ 
 $\underline{w} := C^{-1} \cdot \underline{r}$ 
 $\underline{s} := \underline{w}$ 
 $\sigma := \sigma_{old} := \sigma_0 := (\underline{w}, \underline{r})$ 
repeat
     $\underline{v} := K \cdot \underline{s}$ 
     $\alpha := \sigma / (\underline{s}, \underline{v})$ 
     $\underline{u} := \underline{u} + \alpha \cdot \underline{s}$ 
     $\underline{r} := \underline{r} - \alpha \cdot \underline{v}$ 
     $\underline{w} := C^{-1} \cdot \underline{r}$ 
     $\sigma := (\underline{w}, \underline{r})$ 
     $\beta := \sigma / \sigma_{old}$ 
     $\sigma_{old} := \sigma$ 
     $\underline{s} := \underline{w} + \beta \cdot \underline{s}$ 
until  $\sqrt{\sigma / \sigma_0} < \text{tolerance}$ 

```

Alg. 6.1 Serieller CG mit Vorkonditionierung

6.1.2 Der parallelisierte CG

Wählt man in Algorithmus 6.1 $C = I$ erhält man den klassischen CG. Betrachtet man die darin enthaltenen Operationen und vergleicht sie mit den Ausführungen in Abschnitt 5.1, so kann man folgende *Parallelisierungsstrategie* aufbauen, wobei die Anzahl der notwendigen Kommunikationen möglichst klein gehalten werden soll.

- Die Matrix K muß als \mathbf{K} (5.3) verteilt gespeichert werden, da im anderen Fall Einschränkungen an die Matrixstruktur erfolgen müssen.

⇒ Vektoren $\underline{s}, \underline{u} \rightarrow \underline{\mathfrak{s}}, \underline{\mathfrak{u}}$ sind akkumuliert

⇒ Matrix-mal-Vektor liefert einen verteilten Vektor, ohne daß Kommunikation notwendig ist (5.10).

⇒ Vektoren $\underline{v}, \underline{r}, \underline{f} \rightarrow \underline{\mathfrak{v}}, \underline{\mathfrak{r}}, \underline{\mathfrak{f}}$ sind verteilt gespeichert und werden im Verlauf der Iteration nicht akkumuliert.

⇒ Wählt man auch $\underline{w} \rightarrow \underline{\mathfrak{w}}$ akkumuliert, so lassen sich alle DAXPY-Operationen ohne Kommunikation durchführen.

- Da in den beiden Skalarprodukten der CG-Schleife unterschiedliche Vektortypen verwendet werden, können diese Operationen mit sehr geringem Kommunikationsaufwand ausgeführt werden (5.9).

!! Die Zuweisung $\underline{\mathfrak{w}} := \underline{\mathfrak{r}}$ beinhaltet eine Vektortypumwandlung mittels Akku-

mulation (5.7) und damit eine Kommunikation über die zu mehreren Prozessen gehörenden Daten.

```

Wähle  $\underline{\mathbf{u}}^0$ 
 $\underline{\mathbf{r}} := \underline{\mathbf{f}} - \mathbf{K} \cdot \underline{\mathbf{u}}^0$ 
 $\underline{\mathbf{w}} := \sum_{j=1}^P A_j^T \underline{\mathbf{r}}_j$ 
 $\underline{\mathbf{s}} := \underline{\mathbf{w}}$ 
 $\sigma := \sigma_{old} := \sigma_0 := (\underline{\mathbf{w}}, \underline{\mathbf{r}})$ 
repeat
     $\underline{\mathbf{v}} := \mathbf{K} \cdot \underline{\mathbf{s}}$ 
     $\alpha := \sigma / (\underline{\mathbf{s}}, \underline{\mathbf{v}})$ 
     $\underline{\mathbf{u}} := \underline{\mathbf{u}} + \alpha \cdot \underline{\mathbf{s}}$ 
     $\underline{\mathbf{r}} := \underline{\mathbf{r}} - \alpha \cdot \underline{\mathbf{v}}$ 
     $\underline{\mathbf{w}} := \sum_{j=1}^P A_j^T \underline{\mathbf{r}}_j$ 
     $\sigma := (\underline{\mathbf{w}}, \underline{\mathbf{r}})$ 
     $\beta := \sigma / \sigma_{old}$ 
     $\sigma_{old} := \sigma$ 
     $\underline{\mathbf{s}} := \underline{\mathbf{w}} + \beta \cdot \underline{\mathbf{s}}$ 
until  $\sqrt{\sigma / \sigma_0} < \text{tolerance}$ 

```

Alg. 6.2 Parallelisierter CG

Der resultierende parallele CG-Algorithmus 6.2 erfordert pro Iteration 2 ALL_REDUCE-Operationen mit einer reellen Zahl und eine Vektorakkumulation.

Man kann auch über andere Ansätze zu Algorithmus 6.2 gelangen, so sind z.B. die verteilt gespeicherten Vektoren $\underline{\mathbf{f}}$, $\underline{\mathbf{r}}$ und $\underline{\mathbf{v}}$ identisch mit den funktionalen Größen im CG-Algorithmus (“Energie“). Eine Einteilung der Vektoren in funktionale Größen und Funktionswerte liefert einen guten Ansatz für die Zuordnung der 2 Vektortypen.

Bemerkung 6.1. Es gibt cg-Varianten, in denen beide Skalarprodukte und damit die entsprechende Kommunikationen zusammengefaßt werden können. Dadurch spart man eine Startup-Zeit ein. Jedoch sind diese Varianten unter Umständen instabil.

Zur Vorkonditionierungsproblematik siehe Abschnitt 6.8.

6.2 Das GMRES-Verfahren

6.2.1 Das serielle Verfahren

Zur Auflösung linearer Gleichungssysteme mit nichtsymmetrischer Matrix K existiert, die General Conjugate Residuals Methode. Ist die Matrix darüber hinaus auch noch nicht positiv definit (d.h. $(K\underline{v}, \underline{v})_{L_2} \not\geq 0$), existiert ein ganzes Spektrum von Iterationsverfahren (GMRES, QMR, Bi-cg, Bi-cgstab). Wir betrachten das GMRES-Verfahren welches auf dem Arnoldi-Verfahren (einer Verallgemeinerung der Lanczos-Methode) basiert. Da die Anzahl der zu

```

Wähle  $\underline{u}^0$ 
 $\underline{r} := \underline{f} - K \cdot \underline{u}^0$ 
 $\underline{w}^1 := C^{-1} \cdot \underline{r}$ 
 $z_1 := \sqrt{(\underline{w}^1, \underline{r})}$ 
 $k := 0$ 
repeat  $k := k + 1$ 
     $\underline{r} := K \cdot \underline{w}^k$ 
    for  $i := 1$  to  $k$  do
         $h_{i,k} := (\underline{w}^i, \underline{r})$ 
         $\underline{r} := \underline{r} - h_{i,k} \cdot \underline{w}^i$ 
    end
     $\underline{w}^{k+1} := C^{-1} \cdot \underline{r}$ 
     $h_{k+1,k} := \sqrt{(\underline{w}^{k+1}, \underline{r})}$ 
     $\underline{w}^{k+1} := \underline{w}^{k+1} / h_{k+1,k}$ 
    for  $i := 1$  to  $k - 1$  do  $\begin{pmatrix} h_{i,k} \\ h_{i+1,k} \end{pmatrix} := \begin{pmatrix} c_{i+1} & s_{i+1} \\ s_{i+1} & -c_{i+1} \end{pmatrix} \cdot \begin{pmatrix} h_{i,k} \\ h_{i+1,k} \end{pmatrix}$ 
         $\alpha := \sqrt{h_{k,k}^2 + h_{k+1,k}^2}$ 
         $s_{k+1} := h_{k+1,k} / \alpha$  ;  $c_{k+1} := h_{k,k} / \alpha$  ;  $h_{k,k} := \alpha$ 
         $z_{k+1} := s_{k+1} z_k$  ;  $z_k := c_{k+1} z_k$ 
    until  $|z_{k+1} / z_1| < \text{tolerance}$ 
     $z_k := z_k / h_{k,k}$ 
    for  $i := k - 1$  down to  $1$  do  $z_i := (z_i - \sum_{j=i+1}^k h_{i,j} z_j) / h_{i,i}$ 
 $\underline{u}^k := \underline{u}^0 + \sum_{i=1}^k z_i \cdot \underline{w}^i$ 

```

Alg. 6.3 Serieller GMRES mit Vorkonditionierung

speichernden Vektoren \underline{w}^k und die Größe der Matrix $H = \{h_{i,j}\}_{i,j=\overline{1,k}}$ mit der Anzahl der Schleifendurchläufe zunimmt, können beim GMRES Speicherüberläufe auftreten. Zu deren Verhinderung bricht man die REPEAT-Schleife nach m Durchläufen ab und startet mit der erhaltenen Lösung \underline{u}^m neu. Dieser GMRES(m) kann allerdings instabil werden.

6.2.2 Der parallele GMRES

Bei Algorithmus 6.3 muß man neben den funktionalen Größen \underline{f} , \underline{r} und den Funktionswerten \underline{u} , \underline{w}^k noch die Vektoren $\{z_i\}_{i=\overline{1,k+1}}$, $\{s_i\}_{i=\overline{1,k+1}}$, $\{c_i\}_{i=\overline{1,k+1}}$ und die Matrix $\{h_{i,j}\}_{i,j=\overline{1,k}}$ in Betracht ziehen, wobei diese innerhalb des Algorithmus als skalare Größen betrachtet werden. Folgende *Parallelisierungsstrategie* wird auf den GMRES ohne Vorkonditionierung ($C = I$) angewandt.

- \underline{f} , $\underline{r} \rightarrow \underline{\mathbf{f}}$, $\underline{\mathbf{r}}$ werden verteilt gespeichert.
- \underline{u} , $\underline{w}^k \rightarrow \underline{\mathbf{u}}$, $\underline{\mathbf{w}}^k$ werden akkumuliert gespeichert.
- Die skalaren Größen $\{z_i\}_{i=\overline{1,k+1}}$, $\{s_i\}_{i=\overline{1,k+1}}$, $\{c_i\}_{i=\overline{1,k+1}}$ und $\{h_{i,j}\}_{i,j=\overline{1,k}}$ werden redundant auf jedem Prozessor lokal gespeichert.

\implies Skalarprodukte können mit sehr geringem Kommunikationsaufwand ausgeführt werden (5.9). Matrix-mal-Vektor benötigt keine Kommunikation (5.10).

!! Die Zuweisung $\underline{\mathbf{w}} := \underline{\mathbf{r}}$ beinhaltet eine Vektortypumwandlung mittels Akkumulation (5.7) und damit eine Kommunikation über die zu mehreren Prozessen gehörenden Daten.

- Alle Manipulationen an und mit den skalaren Größen c_i , s_i , z_i und $h_{i,j}$ werden redundant auf allen Prozessoren lokal ausgeführt.

!! *Fast alle* DAXPY-Operationen arbeiten mit den gleichen Vektortypen (und skalaren Größen), *außer* der Operation

$$\underline{\mathbf{r}} := \underline{\mathbf{r}} - h_{i,k} \cdot \underline{\mathbf{w}}^i ,$$

welche verschiedenen Vektortypen enthält. Eine Umwandlung des akkumulierten Vektors $\underline{\mathbf{w}}^i$ in einen verteilten ist nach (5.8) aber ohne Kommunikation möglich !

\implies Alle DAXPY-Operationen kommen ohne Kommunikation aus.

Der resultierende parallele GMRES-Algorithmus 6.4 erfordert in der k -ten Iteration $(k+1)$ ALL_REDUCE-Operationen mit einer reellen Zahl und eine Vektorakkumulation. Durch die notwendige Typumwandlung des Vektors $\underline{\mathbf{w}}^i$ kommen noch zusätzlich $k \cdot n$ Multiplikationen hinzu (n - Länge des Vektors $\underline{\mathbf{w}}$). Die Variante GMRES(m), mit Neustart nach m Iterationen, läßt sich völlig analog parallelisieren.

```

Wähle  $\underline{\mathbf{u}}^0$ 
 $\underline{\mathbf{r}} := \underline{\mathbf{f}} - \mathbf{K} \cdot \underline{\mathbf{u}}^0$ 
 $\underline{\mathbf{w}}^1 := \sum_{s=1}^P A_s^T \underline{\mathbf{r}}_s$ 
 $z_1 := \sqrt{(\underline{\mathbf{w}}^1, \underline{\mathbf{r}})}$ 
 $k := 0$ 
repeat  $k := k + 1$ 
   $\underline{\mathbf{r}} := \mathbf{K} \cdot \underline{\mathbf{w}}^k$ 
  for  $i := 1$  to  $k$  do
     $h_{i,k} := (\underline{\mathbf{w}}^i, \underline{\mathbf{r}})$ 
     $\underline{\mathbf{r}} := \underline{\mathbf{r}} - h_{i,k} \cdot R^{-1} \cdot \underline{\mathbf{w}}^i$ 
  end
   $\underline{\mathbf{w}}^{k+1} := \sum_{s=1}^P A_s^T \underline{\mathbf{r}}_s$ 
   $h_{k+1,k} := \sqrt{(\underline{\mathbf{w}}^{k+1}, \underline{\mathbf{r}})}$ 
   $\underline{\mathbf{w}}^{k+1} := \underline{\mathbf{w}}^{k+1} / h_{k+1,k}$ 
  for  $i := 1$  to  $k - 1$  do  $\begin{pmatrix} h_{i,k} \\ h_{i+1,k} \end{pmatrix} := \begin{pmatrix} c_{i+1} & s_{i+1} \\ s_{i+1} & -c_{i+1} \end{pmatrix} \cdot \begin{pmatrix} h_{i,k} \\ h_{i+1,k} \end{pmatrix}$ 
   $\alpha := \sqrt{h_{k,k}^2 + h_{k+1,k}^2}$ 
   $s_{k+1} := h_{k+1,k} / \alpha$  ;  $c_{k+1} := h_{k,k} / \alpha$  ;  $h_{k,k} := \alpha$ 
   $z_{k+1} := s_{k+1} z_k$  ;  $z_k := c_{k+1} z_k$ 
until  $|z_{k+1} / z_1| < \text{tolerance}$ 
 $z_k := z_k / h_{k,k}$ 
for  $i := k - 1$  down to 1 do  $z_i := (z_i - \sum_{j=i+1}^k h_{i,j} z_j) / h_{i,i}$ 
 $\underline{\mathbf{u}}^k := \underline{\mathbf{u}}^0 + \sum_{i=1}^k z_i \cdot \underline{\mathbf{w}}^i$ 

```

Alg. 6.4 Paralleler GMRES

6.3 Das ω -Jacobi Verfahren

6.3.1 Das serielle Verfahren

Das ω -Jacobi Verfahren benötigt intern eine Division durch die Hauptdiagonalelemente der Matrix K , daher setzen wir $D = \text{diag}(K)$.

```

Wähle  $\underline{u}^0$ 
 $\underline{r} := \underline{f} - K \cdot \underline{u}^0$ 
 $\sigma := \sigma_0 := (\underline{r}, \underline{r})$ 
 $k := 0$ 
while  $\sigma > \text{tolerance}^2 \cdot \sigma_0$  do
   $k := k + 1$ 
   $\underline{u}^k := \underline{u}^{k-1} + \omega \cdot D^{-1} \cdot \underline{r}$ 
   $\underline{r} := \underline{f} - K \cdot \underline{u}^k$ 
   $\sigma := (\underline{r}, \underline{r})$ 
end

```

Alg. 6.5 Serielle Jacobi-Iteration

6.3.2 Datengraph der Jacobi-Iteration

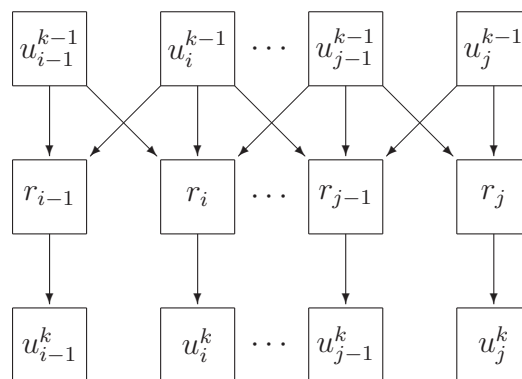


Fig. 6.1 Datengraph der Jacobi-Iteration

Da während einer Jacobi-Iteration keine Datenabhängigkeiten zwischen den Komponenten des Vektors \underline{u}^k auftreten, sind Vektorisierung und Parallelisierung leicht durchführbar (vgl. Abschnitt 4.1), der entsprechende Datengraph ist in Fig. 6.1 dargestellt.

6.3.3 Das parallele Verfahren

Auch im parallelen Fall benötigt das ω -Jacobi Verfahren die Hauptdiagonalelemente der assemblierten Steifigkeitsmatrix. Da diese verteilt gespeichert sind, muß die Hauptdiagonale separat assembliert und gespeichert werden. Die *Parallelisierungsstrategie* ähnelt dem Vorgehen beim CG-Verfahren in Abschnitt 6.1.2 .

- Die invertierte assemblierte Diagonale wird als zusätzlicher Vektor abgespeichert $\underline{\mathfrak{d}} = \mathfrak{D}^{-1} = \text{diag}^{-1}(\sum_{s=1}^P A_s^T \mathbf{K}_s A_s)$.

- Die Matrix K ist als \mathbf{K} (5.3) verteilt gespeichert.

⇒ Vektor $\underline{u} \rightarrow \underline{\mathbf{u}}$ ist akkumuliert, Vektoren $\underline{r}, \underline{f} \rightarrow \underline{\mathbf{r}}, \underline{\mathbf{f}}$ sind verteilt gespeichert.

⇒ Matrix-mal-Vektor benötigt keine Kommunikation (5.10).

!! Das Skalarprodukt benötigt unterschiedliche Vektortypen

⇒ Akkumulation von $\underline{\mathbf{r}} \rightarrow \underline{\mathbf{w}}$.

!! Mit dem akkumulierten Vektor $\underline{\mathbf{w}}$ läßt sich nunmehr die verbliebene DAXPY-Operation kommunikationsfrei ausführen.

```

 $\underline{\mathfrak{d}} := \text{diag}(\mathbf{K}^{[i,i]})_{i=1,n}$ 
 $\underline{\mathfrak{d}} := \sum_{s=1}^P A_s^T \underline{\mathfrak{d}}_s$ 
 $\underline{\mathfrak{d}} := \{1/\underline{\mathfrak{d}}^{[i]}\}_{i=1,n}$ 
Wähle  $\underline{\mathbf{u}}^0$ 
 $\underline{\mathbf{r}} := \underline{\mathbf{f}} - \mathbf{K} \cdot \underline{\mathbf{u}}^0$ 
 $\underline{\mathbf{w}} := \sum_{s=1}^P A_s^T \underline{\mathbf{r}}_s$ 
 $\sigma := \sigma_0 := (\underline{\mathbf{w}}, \underline{\mathbf{r}})$ 
 $k := 0$ 
while  $\sigma > \text{tolerance}^2 \cdot \sigma_0$  do
   $k := k + 1$ 
   $\underline{\mathbf{u}}^k := \underline{\mathbf{u}}^{k-1} + \omega \cdot \underline{\mathfrak{d}} \otimes \underline{\mathbf{w}}$ 
   $\underline{\mathbf{r}} := \underline{\mathbf{f}} - \mathbf{K} \cdot \underline{\mathbf{u}}^k$ 
   $\underline{\mathbf{w}} := \sum_{s=1}^P A_s^T \underline{\mathbf{r}}_s$ 
   $\sigma := (\underline{\mathbf{w}}, \underline{\mathbf{r}})$ 
end

```

Alg. 6.6 Parallele Jacobi-Iteration : JACOBI($\mathbf{K}, \underline{\mathbf{u}}^0, \underline{\mathbf{f}}$)

Die in Algorithmus 6.6 vorkommende Operation $\underline{\mathfrak{d}} \circledast \underline{\mathfrak{w}}$ bezeichnet die komponentenweise Multiplikation zweier Vektoren, d.h. $\{\underline{\mathfrak{d}}_i \cdot \underline{\mathfrak{w}}_i\}_{i=\overline{1,n}}$, und ist nichts anderes als die Multiplikation einer Diagonalmatrix mit einem Vektor. Falls der Relaxationsparameter ω im Verlaufe der Iteration konstant bleibt, kann man ihn im Vorbereitungsschritt mit dem Vektor $\underline{\mathfrak{d}}$ multiplizieren.

Falls kein Abbruchtest nötig ist, z.B. bei Verwendung als Glätter mit einer fixen Anzahl von Schritten, dann kann man die Berechnung der Skalarprodukte weglassen und $\underline{\mathfrak{r}}$ kann identisch $\underline{\mathfrak{w}}$ gewählt werden.

Per Jacobi-Iterationsschritt tritt somit eine Vektorakkumulation und ein ALL_REDUCE einer reellen Zahl auf.

6.4 Das Gauß-Seidel Verfahren

Bezeichnen L und U die strikt untere bzw. obere Dreiecksmatrix von K aus (6.1) und D die entsprechende Diagonale, so läßt sich die Steifigkeitsmatrix eindeutig als $K = L + D + U$ darstellen. Des weiteren gehen wir von der Dünnbesetztheit von K aus.

6.4.1 Das serielle Verfahren

```

Wähle  $\underline{u}^0$ 
 $\tilde{\underline{r}} := \underline{f} - K \cdot \underline{u}^0$ 
 $\sigma := \sigma_0 := (\tilde{\underline{r}}, \tilde{\underline{r}})$ 
 $k := 0$ 
while  $\sigma > \text{tolerance}^2 \cdot \sigma_0$  do
   $k := k + 1$ 
   $\underline{u}^k := \underline{u}^{k-1} + D^{-1} \cdot (\underline{f} - L \cdot \underline{u}^k - (D + U) \cdot \underline{u}^{k-1})$ 
   $\tilde{\underline{r}} := \underline{f} - K \cdot \underline{u}^k$ 
   $\sigma := (\tilde{\underline{r}}, \tilde{\underline{r}})$ 
end

```

Alg. 6.7 Serieller Gauß-Seidel vorwärts

Der wesentliche algorithmische Unterschied zwischen dem ω -Jacobi Verfahren (Alg. 6.5) und der Gauß-Seidel Iteration besteht im ständigen Update des Residuums zur Bestimmung der Iterierten \underline{u}^k , dadurch ist die Konvergenzgeschwindigkeit gegenüber dem ω -Jacobi Verfahren höher. Diesen Schritt

betrachten wir in Komponentenschreibweise :

$$r_i := f_i - \sum_{j=1}^{i-1} K_{i,j} u_j^k - \sum_{j=i}^n K_{i,j} u_j^{k-1} \quad (6.2)$$

$$u_i^k := u_i^{k-1} + K_{i,i}^{-1} \cdot r_i$$

Die Iterierte u_i^k in (6.2) hängt von der Numerierung der Vektorkomponenten, d.h. ihrer Abarbeitungsreihenfolge ab. Damit ist das Gauß-Seidel-Verfahren numerierungsabhängig. Um Rechenaufwand zu sparen, wird üblicherweise im Abbruchtest von Alg. 6.7 das punktweise Residuum \underline{r} aus (6.2) anstelle des Residuums \tilde{r} benutzt. Wir werden uns in weiteren Abschnitt stets auf diesen Abbruchtest $(\underline{r}, \underline{r})$ beziehen. Alle Algorithmen sind dann auch auf den Abbruchtest $(D^{-1}\underline{r}, \underline{r})$ anwendbar, wobei $D^{-1}\underline{r}$ die Korrektur im k -ten Iterationsschritt darstellt.

Wie aus Alg. 6.7 und dem Graphen in Fig. 6.2 ersichtlich, hängen die Komponenten von \underline{u}^k voneinander ab, z.B. kann u_i^k erst berechnet werden wenn u_{i-1}^k vorliegt. Allgemein müssen erst alle u_s^k ($s < i$) vorliegen, für welche $K_{i,s} \neq 0$ (Graph der Matrix !) ist, ehe u_i^k berechnet werden kann.

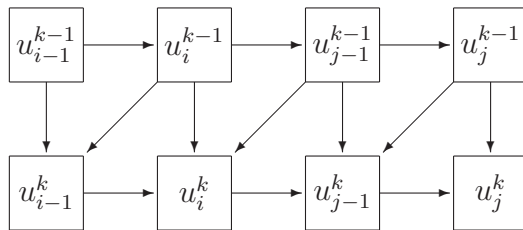


Fig. 6.2 Datengraph der Gauß-Seidel-Iteration

Diese Datenabhängigkeit ist für die Parallelisierung hinderlich und macht eine Vektorisierung unmöglich.

6.4.2 Die Red-Black-Gauß-Seidel-Iteration

Um trotz der ungünstigen Datenabhängigkeiten das Gauß-Seidel-Verfahren auf Vektor- und Parallelrechnern nutzen zu können, teilt man die Indizes der Vektoren in (mindestens) zwei disjunkte Teilmengen ω_{red} und ω_{black} auf, so daß gilt $K_{i,j} \equiv 0 \forall i \neq j \in \omega_{\text{red}}$ und analog für ω_{black} , d.h. innerhalb der Teilmengen existieren keine Datenabhängigkeiten.

Somit läßt sich der Updateschritt in Alg. 6.7 umformulieren zu

$$\begin{aligned} \underline{u}_{\text{red}}^k &:= \underline{u}_{\text{red}}^{k-1} + D_{\text{red}}^{-1} \cdot \left(\underline{f}_{\text{red}} - (L_{rb} + U_{rb}) \cdot \underline{u}_{\text{black}}^{k-1} - D_{\text{red}} \underline{u}_{\text{red}}^{k-1} \right) \\ \underline{u}_{\text{black}}^k &:= \underline{u}_{\text{black}}^{k-1} + D_{\text{black}}^{-1} \cdot \left(\underline{f}_{\text{black}} - (L_{br} + U_{br}) \cdot \underline{u}_{\text{red}}^k - D_{\text{black}} \underline{u}_{\text{black}}^{k-1} \right) \end{aligned}$$

Alg. 6.8 Updateschritt im Red-Black-Gauß-Seidel vorwärts

Der entsprechende Datengraph ist in Fig. 6.3 dargestellt.

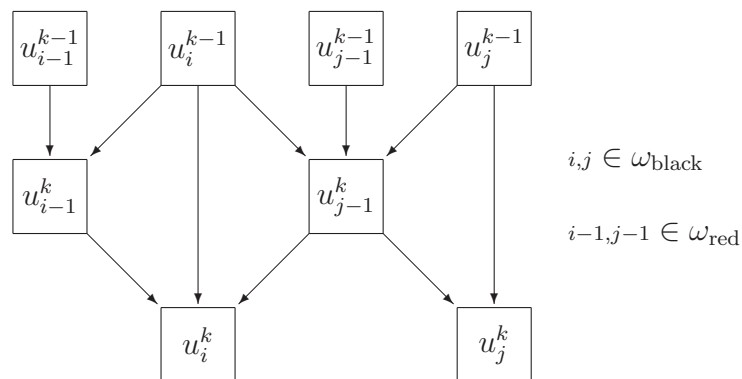


Fig. 6.3 Datengraph der Red-Black-Gauß-Seidel-Iteration

Da innerhalb der "roten" und "schwarzen Daten" keine Abhängigkeiten mehr existieren, ist Algorithmus 6.8 nunmehr leicht vektorisierbar. Diese knotenweise Red/Black-Datenaufteilung ist bei einer Diskretisierung mit dem 5-Punkte Differenzenstern in 2D anwendbar, bei Verwendung eines 7-Punkte Differenzensterns auch in 3D. Komplexere Datenabhängigkeiten erfordern entsprechend mehr "Farben".

Für die Parallelisierung empfiehlt sich eine Blockvariante obiger Iteration, in welcher die Bearbeitung "gleichfarbiger" Datenblöcke keinerlei Kommunikation erfordert. Beim Einheitsquadrat ergibt sich eine schachbrettartige Einfärbung des Gebietes (hier reichen im günstigsten Fall 2 Farben in 2D !!), welches einer disjunkten Datenaufteilung entspricht.

6.4.3 Das parallele Verfahren

Da der Updateschritt der wesentliche Unterschied gegenüber dem Jacobi-Verfahren ist, beschränken wir uns auf dessen Analyse. Als Datenverteilung werden die nichtüberlappenden Elemente vorausgesetzt (Abschnitt 5.1).

Eine formale Anwendung der Parallelisierungsstrategie des ω -Jacobi Verfahrens ergibt eine nichtakkumulierte Matrix \mathbf{K} , eine akkumulierte Diagonalmatrix \mathfrak{D} , die verteilt gespeicherten Vektoren $\underline{\mathbf{f}}$, $\underline{\mathbf{r}}$ und die akkumulierten Vektoren $\underline{\mathbf{u}}^k$, $\underline{\mathbf{m}}$. Im Unterschied zum Jacobi-Verfahren ergeben sich mehrere Varianten.

Variante 1 : formales Übertragen

Ein formales Umschreiben von Alg. 6.7 mit den parallelen Datentypen ergibt die erste Variante.

$$\underline{\mathbf{u}}^k := \underline{\mathbf{u}}^{k-1} + \mathfrak{D}^{-1} \cdot \sum_{s=1}^P A_s^T (\underline{\mathbf{f}}_s - \mathbf{L}_s \underline{\mathbf{u}}_s^k - (\mathbf{D}_s + \mathbf{U}_s) \cdot \underline{\mathbf{u}}_s^{k-1})$$

Alg. 6.9 Paralleler Updateschritt Gauß-Seidel : Variante 1

Auf den ersten Blick ist man mit Algorithmus 6.9 bereits am Ziel seiner Wünsche, jedoch werden sich bei der Implementierung Schwierigkeiten einstellen, welche eine effiziente Parallelisierung fast unmöglich machen. Die gewählte Datenverteilung mit nichtüberlappenden Elementen ist in Fig. 5.5 dargestellt.

$$\widehat{\underline{\mathbf{r}}}_V := \underline{\mathbf{f}}_V - \mathbf{K}_{VE} \underline{\mathbf{u}}_E^{k-1} - \mathbf{K}_{VI} \underline{\mathbf{u}}_I^{k-1} \quad (6.3a)$$

$$i=\overline{1, N_V} : \begin{cases} r_{V,i} := \widehat{r}_{V,i} - \sum_{j=1}^{i-1} \mathbf{K}_{V,ij} \underline{\mathbf{u}}_{V,j}^k - \sum_{j=i}^{N_V} \mathbf{K}_{V,ij} \underline{\mathbf{u}}_{V,j}^{k-1} \\ \underline{\mathbf{u}}_{V,i}^k := \underline{\mathbf{u}}_{V,i}^{k-1} + \mathfrak{D}_{V,i}^{-1} \cdot \sum_{s=1}^P A_{s,i}^T r_{V,i} \end{cases} \quad (6.3b)$$

$$\widehat{\underline{\mathbf{r}}}_E := \underline{\mathbf{f}}_E - \mathbf{K}_{EV} \underline{\mathbf{u}}_V^k - \mathbf{K}_{EI} \underline{\mathbf{u}}_I^{k-1} \quad (6.3c)$$

$$i=\overline{N_V+1, N_C} : \begin{cases} r_{E,i} := \widehat{r}_{E,i} - \sum_{j=N_V+1}^{i-1} \mathbf{K}_{E,ij} \underline{\mathbf{u}}_{E,j}^k - \sum_{j=i}^{N_C} \mathbf{K}_{E,ij} \underline{\mathbf{u}}_{E,j}^{k-1} \\ \underline{\mathbf{u}}_{E,i}^k := \underline{\mathbf{u}}_{E,i}^{k-1} + \mathfrak{D}_{E,i}^{-1} \cdot \sum_{s=1}^P A_{s,i}^T r_{E,i} \end{cases} \quad (6.3d)$$

$$\widehat{\underline{\mathbf{r}}}_I := \underline{\mathbf{f}}_I - \mathbf{K}_{IV} \underline{\mathbf{u}}_V^k - \mathbf{K}_{IE} \underline{\mathbf{u}}_E^k \quad (6.3e)$$

$$i=\overline{N_C+1, N_I} : \begin{cases} r_{I,i} := \widehat{r}_{I,i} - \sum_{j=N_C+1}^{i-1} \mathbf{K}_{I,ij} \underline{\mathbf{u}}_{I,j}^k - \sum_{j=i}^N \mathbf{K}_{I,ij} \underline{\mathbf{u}}_{I,j}^{k-1} \\ \underline{\mathbf{u}}_{I,i}^k := \underline{\mathbf{u}}_{I,i}^{k-1} + \mathfrak{D}_{I,i}^{-1} \cdot r_{I,i} \end{cases} \quad (6.3f)$$

Die Gleichungen (6.3) schlüsseln die Berechnungen in Algorithmus 6.9 entsprechend der, durch die Knotennumerierung analog zu Punkt 5.1 erzeugten, Blockstruktur der Matrix und Vektoren auf. Die Summe $\sum_{s=1}^P A_{s,i}^T(\cdot)$ bedeutet in *jedem* Update einer Komponente i die Akkumulation des Residuums dieser Komponente. Die Gesamtanzahl der übertragenen Worte ($N_C = N_V + N_E$) ist hierbei genauso groß wie beim entsprechenden Updateschritt im ω -Jacobi-Verfahren, jedoch sind dies hier N_C *einzelne* Kommunikationen für welche auch N_C Startupzeiten hinzukommen. Es gelte $t_{\text{Startup}} = c \cdot t_{\text{Word}}$, dann ergibt sich im Vergleich :

$$\begin{aligned} t_{\text{Jacobi}} &:= t_{\text{Startup}} + N_C \cdot t_{\text{Word}} \leq N_C \cdot (t_{\text{Startup}} + t_{\text{Word}}) =: t_{\text{GS}} \\ &\quad (N_C + c) \cdot t_{\text{Word}} \leq N_C(c + 1) \cdot t_{\text{Word}} \end{aligned}$$

In obiger Ungleichung bekommt der Faktor c mit wachsender Datenmenge auf dem Koppelrand einen dominierenden Einfluß auf das Verhältnis der Kommunikationszeiten der zwei Verfahren und es gilt

$$t_{\text{GS}} \geq N_C \left[\frac{c + 1}{c + N_C} \right] \cdot t_{\text{Jacobi}} \xrightarrow{N_C \rightarrow \infty} (c + 1) \cdot t_{\text{Jacobi}} ,$$

d.h. die Effizienz des Gauß-Seidel-Verfahrens mit dem Updateschritt (6.3) wird weit unter der des Jacobi-Verfahren liegen.

Durch Ausnutzung von Nachbarschaftsbeziehungen kann dieses Kommunikationsverhältnis N_C bis auf die maximale Anzahl von Daten auf einer Interfacekante reduziert werden.

Damit (6.3) überhaupt vernünftig parallelisierbar ist, müssen die Anforderungen 1 und 2 (Seite 92) an das FE-Netz erfüllt sein, siehe Fig. 5.5 . Ist dies nicht der Fall, so müssen (6.3b) und (6.3d) sequentiell abgearbeitet werden! Dies kann nur durch eine Red-Black-Numerierung der Interfacekanten erreicht werden, was bei allgemeinen Gebieten und Diskretisierungen mehr als 2 Farben erfordert.

Variante 2 : Gauß-Seidel- ω -Jacobi

Die schlechte Parallelisierung von Variante 1 war eng mit der Datenabhängigkeit der Iterierten auf dem Koppelrand verbunden. Benutzt man in (6.3b) und (6.3d) nur die jeweiligen $(k-1)$ -ten Iterierten, vermeidet man diese Datenabhängigkeit und benötigt auch keinerlei Voraussetzungen an die Vernetzung. Genauso wie zwischen den Blöcken "V, E, I" wird im Teilgebietsinneren die Gauß-Seidel-Iteration ausgeführt. Durch die partielle Anwendung der ω -Jacobi Iteration ist die Konvergenzgeschwindigkeit etwas geringer als beim vollen Gauß-Seidel Verfahren.

$$\underline{d} := \text{diag}(K^{[i,i]})_{i=\overline{1,n}}$$

$$\underline{d}_V := \sum_{s=1}^P A_s^T \underline{d}_s$$

$$\underline{d}_E := \{1/d^{[i]}\}_{i=\overline{1,n}}$$

Wähle \underline{u}^0

$$\underline{r} := \underline{f} - \mathbf{K} \cdot \underline{u}^0$$

$$\underline{w} := \sum_{s=1}^P A_s^T \underline{r}_s$$

$$\sigma := \sigma_0 := (\underline{w}, \underline{r})$$

$$k := 0$$

while $\sigma > \text{tolerance}^2 \cdot \sigma_0$ do

$$k := k + 1$$

$$\underline{r}_V := \underline{f}_V - \mathbf{K}_V \cdot \underline{u}_V^{k-1} - \mathbf{K}_{VE} \cdot \underline{u}_E^{k-1} - \mathbf{K}_{VI} \cdot \underline{u}_I^{k-1}$$

$$\underline{w}_V := \sum_{s=1}^P A_{V,s}^T \underline{r}_{V,s}$$

$$\underline{u}_V^k := \underline{u}_V^{k-1} + \underline{d}_V \otimes \underline{w}_V$$

$$\underline{r}_E := \underline{f}_E - \mathbf{K}_{EV} \cdot \underline{u}_V^k - \mathbf{K}_E \cdot \underline{u}_E^{k-1} - \mathbf{K}_{EI} \cdot \underline{u}_I^{k-1}$$

$$\underline{w}_E := \sum_{s=1}^P A_{E,s}^T \underline{r}_{E,s}$$

$$\underline{u}_E^k := \underline{u}_E^{k-1} + \underline{d}_E \otimes \underline{w}_E$$

$$\underline{\hat{r}}_I := \underline{f}_I - \mathbf{K}_{IV} \cdot \underline{u}_V^k - \mathbf{K}_{IE} \cdot \underline{u}_E^k$$

$${}_{i=\overline{N_C+1,N}} : \begin{cases} r_{I,i} := \underline{\hat{r}}_{I,i} - \sum_{j=N_C+1}^{i-1} K_{I,ij} \cdot u_{I,j}^k - \sum_{j=i}^N K_{I,ij} \cdot u_{I,j}^{k-1} \\ u_{I,i}^k := u_{I,i}^{k-1} + d_{I,i} \cdot r_{I,i} \end{cases}$$

$$\underline{w}_I := \underline{r}_I$$

$$\sigma := (\underline{w}, \underline{r})$$

end

Alg. 6.10 Paralleler Gauß-Seidel- ω -Jacobi vorwärts

Die in Algorithmus 6.10 vorkommende Operation $\underline{\mathbf{d}} \otimes \underline{\mathbf{m}}$ bezeichnet die komponentenweise Multiplikation zweier Vektoren. Außerdem wurde berücksichtigt, daß im Teilgebietsinneren ("I") alle akkumulierten Teilvektoren und -matrizen stets identisch den verteilt gespeicherten Varianten sind.

Berücksichtigt man, daß die Crosspoints ("V") und die Interfacedaten ("E") in jedem Falle getrennt akkumuliert werden, dann wird pro Iterationsschritt nur genausoviel Kommunikation wie beim ω -Jacobi Verfahren (Algorithmus 6.6) benötigt !

Falls kein Abbruchttest nötig ist, z.B. bei Verwendung als Glätter mit einer fixen Anzahl von Schritten, dann kann man die Berechnung der Skalarprodukte weglassen und $\underline{\mathbf{r}}$ identisch $\underline{\mathbf{m}}$ wählen.

Variante 3 : Gauß-Seidel-Blocklöser

Im Unterschied zu Variante 2, welche (6.3b) und (6.3d) durch ein etwas schlechteres Verfahren ersetzte, werden nunmehr die entsprechenden Blocksysteme direkt gelöst. Voraussetzung für eine parallele Auflösbarkeit ist die in Abschnitt 5.1 geforderte Blockstruktur der akkumulierten Matrizen \mathfrak{K}_E und \mathfrak{K}_V (siehe auch Netzanforderungen 1 und 2). Die Matrixeinträge $\mathfrak{K}_{E,j}$ aus $\mathfrak{K}_E = \text{blockdiag}\{\mathfrak{K}_{E,j}\}_{j=1, \text{NumEdges}}$ sind für Kanten und lineare Ansatzfunktionen tridiagonal und lassen sich somit problemlos invertieren (Thompson/Progonka). Die Matrix \mathfrak{K}_V ist normalerweise eine Diagonalmatrix.

Genauso wie zwischen den Blöcken "V, E, I" wird im Teilgebietsinneren die Gauß-Seidel-Iteration ausgeführt. Durch die partielle Anwendung eines direkten Löser ist die Konvergenzgeschwindigkeit etwas besser als beim Gauß-Seidel Verfahren.

Berücksichtigt man, daß die Crosspoints ("V") und die Interfacedaten ("E") in jedem Falle getrennt akkumuliert werden, dann wird pro Iterationsschritt nur genausoviel Kommunikation wie beim ω -Jacobi Verfahren (Algorithmus 6.6) benötigt, allerdings ist der Vorbereitungsschritt durch die Akkumulation der Matrix (tridiagonal) fast 3-mal so aufwendig.

Die akkumulierten Matrizen \mathfrak{K}_E und \mathfrak{K}_V müssen in obigem Algorithmus zusätzlich gespeichert werden, da die nichtakkumulierten Matrizen \mathbf{K}_E und \mathbf{K}_V noch in den Matrix-mal-Vektor Operationen benötigt werden.

Falls kein Abbruchttest nötig ist, z.B. bei Verwendung als Glätter mit einer fixen Anzahl von Schritten, kann man die Berechnung der Skalarprodukte weglassen und $\underline{\mathbf{r}}$ identisch $\underline{\mathbf{m}}$ wählen.

Aufgabe 6.1. Geben Sie eine Variante von Algorithmus 6.11 an, welche ohne die doppelte Abspeicherung von K_E und K_V auskommt !

$$\mathfrak{K}_V := \sum_{s=1}^P A_{V,s}^T K_{V,s} A_{V,s}$$

$$\mathfrak{K}_E := \sum_{s=1}^P A_{E,s}^T K_{E,s} A_{E,s}$$

Wähle \underline{u}^0

$$\underline{r} := \underline{f} - \mathbf{K} \cdot \underline{u}^0$$

$$\underline{w} := \sum_{s=1}^P A_s^T \underline{r}_s$$

$$\sigma := \sigma_0 := (\underline{w}, \underline{r})$$

$$k := 0$$

while $\sigma > \text{tolerance}^2 \cdot \sigma_0$ do

$$k := k + 1$$

$$\underline{r}_V := \underline{f}_V - \mathbf{K}_V \cdot \underline{u}_V^{k-1} - \mathbf{K}_{VE} \cdot \underline{u}_E^{k-1} - \mathbf{K}_{VI} \cdot \underline{u}_I^{k-1}$$

$$\underline{w}_V := \sum_{s=1}^P A_{V,s}^T \underline{r}_{V,s}$$

$$\underline{u}_V^k := \underline{u}_V^{k-1} + \mathfrak{K}_V^{-1} \cdot \underline{w}_V$$

$$\underline{r}_E := \underline{f}_E - \mathbf{K}_{EV} \cdot \underline{u}_V^k - \mathbf{K}_E \cdot \underline{u}_E^{k-1} - \mathbf{K}_{EI} \cdot \underline{u}_I^{k-1}$$

$$\underline{w}_E := \sum_{s=1}^P A_{E,s}^T \underline{r}_{E,s}$$

$$\underline{u}_E^k := \underline{u}_E^{k-1} + \mathfrak{K}_E^{-1} \cdot \underline{w}_E$$

$$\widehat{r}_I := \underline{f}_I - \mathbf{K}_{IV} \cdot \underline{u}_V^k - \mathbf{K}_{IE} \cdot \underline{u}_E^k$$

$${}_{i=\overline{N_C+1, N}}: \begin{cases} r_{I,i} := \widehat{r}_{I,i} - \sum_{j=N_C+1}^{i-1} K_{I,ij} \cdot u_{I,j}^k - \sum_{j=i}^N K_{I,ij} \cdot u_{I,j}^{k-1} \\ u_{I,i}^k := u_{I,i}^{k-1} + d_{I,i} \cdot r_{I,i} \end{cases}$$

$$\underline{w}_I := \underline{r}_I$$

$$\sigma := (\underline{w}, \underline{r})$$

end

Alg. 6.11 Paralleler Gauß-Seidel-Blocklöser vorwärts

6.5 Unvollständige Zerlegungen

Schon seit Jahrzehnten wird zum Lösen von (6.1) die Matrix K in eine obere Dreiecksmatrix \tilde{U} und eine untere Dreiecksmatrix \tilde{L} zerlegt. Während die Auflösung der resultierenden, gestaffelten Gleichungssysteme mit diesen Dreiecksmatrizen sehr schnell ausführbar ist, explodiert die Rechenzeit für die Faktorisierung

$$K = \tilde{L} \cdot \tilde{U} , \quad (6.4)$$

da sich die Anzahl der arithmetischen Operationen wie $\mathcal{O}(N^3)$ verhält. Bei der Faktorisierung dünnbesetzter Matrizen tritt ein Fill-In auf, welches bei einem sehr ausgedünnten Besetztheitsmuster der Matrix (FEM, FDM, ...) zu einem gravierenden Mehraufwand an Speicher zur Abspeicherung der zerlegten Matrix führt.

Aus diesen beiden Gründen heraus wird versucht, die Matrix K in Dreiecksmatrizen L und U mit dem gleichen Besetztheitsmuster zu zerlegen. Die Gleichung (6.4) gilt dadurch nicht mehr, zur exakten Beschreibung muß noch eine Restmatrix P in Betracht gezogen werden:

$$K = L \cdot U - P . \quad (6.5)$$

Die entstandene Faktorisierung $L \cdot U$ bzw. $U \cdot L$ kann in der einfachen Iteration zur Approximation des Defektes genutzt werden (auch anstelle von D^{-1} in der Jacobi-Iteration in Alg. 6.5).

Es gibt spezielle Faktorisierungen für symmetrische, positiv definite Matrizen (ICC= Incomplete Cholesky) und Faktorisierungen, die ein gewisses Fill-In zulassen (ILU(m)). Auch Lumpingtechniken, d.h. wegzulassende Einträge werden auf die Hauptdiagonale addiert, sind in Verwendung (MAF).

Im folgenden betrachten wir die unvollständige Faktorisierung (6.5) für nicht-symmetrische Matrizen K ohne Fill-In, d.h. die klassische ILU(0) Faktorisierung, und den entsprechenden Auflösungsschritt $L \cdot U \cdot \underline{w} = \underline{r}$.

6.5.1 Der serielle Algorithmus

Unter Benutzung der Blockstruktur (5.4) der Matrix und Vektoren aus Abschnitt 5.1 schreiben wir die blockweise LU-Zerlegung mit einer typlosen Matrix K und typlosen Vektoren auf.

Die Restmatrizen P dienen nur der saubereren Notation des Algorithmus, in der praktischen Rechnung werden sie weggelassen. Innerhalb der einzelnen Blöcke wird die punktweise ILU verwandt.

Bestimme	
L_V, U_V	$K_V = L_V \cdot U_V - P_V$
L_{EV}	$K_{EV} = L_{EV} \cdot U_V - P_{EV}$
L_{IV}	$K_{IV} = L_{IV} \cdot U_V - P_{IV}$
U_{VE}	$K_{VE} = L_V \cdot U_{VE} - P_{VE}$
U_{VI}	$K_{VI} = L_V \cdot U_{VI} - P_{VI}$
	$K_E := K_E - L_{EV} \cdot U_{VE}$
	$K_{EI} := K_{EI} - L_{EV} \cdot U_{VI}$
	$K_{IE} := K_{IE} - L_{IV} \cdot U_{VE}$
	$K_I := K_I - L_{IV} \cdot U_{VI}$
Bestimme	
L_E, U_E	$K_E = L_E \cdot U_E - P_E$
L_{IE}	$K_{IE} = L_{IE} \cdot U_E - P_{IE}$
U_{EI}	$K_{EI} = L_E \cdot U_{EI} - P_{EI}$
	$K_I := K_I - L_{IE} \cdot U_{EI}$
Bestimme	
L_I, U_I	$K_I = L_I \cdot U_I - P_I$

Alg. 6.12 Serielle Block-ILU-Faktorisierung

I)	Solve $L \cdot \underline{u} = \underline{r}$
	$\underline{u}_V := L_V^{-1} \underline{r}_V$
	$\underline{u}_E := L_E^{-1} (\underline{r}_E - L_{EV} \underline{u}_V)$
	$\underline{u}_I := L_I^{-1} (\underline{r}_I - L_{IE} \underline{u}_E - L_{IV} \underline{u}_V)$
II)	Solve $U \cdot \underline{w} = \underline{u}$
	$\underline{w}_I := U_I^{-1} \underline{u}_I$
	$\underline{w}_E := U_E^{-1} (\underline{u}_E - U_{EI} \underline{w}_I)$
	$\underline{w}_V := U_V^{-1} (\underline{u}_V - U_{VE} \underline{w}_E - U_{VI} \underline{w}_I)$

Alg. 6.13 Serieller blockweiser Auflösungsschritt $L \cdot U \cdot \underline{w} = \underline{r}$

6.5.2 Die parallele ILU-Faktorisierung

Die erste Idee zur Parallelisierung der ILU ist die einfache Übertragung von Algorithmus 6.12 auf eine akkumulierte Matrix \mathfrak{K} . Natürlicherweise sind \mathfrak{L} und \mathfrak{U} ebenfalls akkumuliert. Jedoch müssen nun wegen (5.30) und (5.31) die Forderungen 5.19 und 5.22 an die Vernetzung aus Abschnitt 5.1 erfüllt sein.

Start	$\mathfrak{K} = \sum_{i=1}^P A_i^T K_i A_i$	
Determine		Why parallel ?
$\mathfrak{L}_V, \mathfrak{U}_V$	$\mathfrak{K}_V = \mathfrak{L}_V \cdot \mathfrak{U}_V - P_V$	mesh \rightarrow diagonal matrix
\mathfrak{L}_{EV}	$\mathfrak{K}_{EV} = \mathfrak{L}_{EV} \cdot \mathfrak{U}_V - P_{EV}$	DD, mesh \rightarrow block matrices
L_{IV}	$K_{IV} = L_{IV} \cdot \mathfrak{U}_V - P_{IV}$	DD, mesh \rightarrow block matrices
\mathfrak{U}_{VE}	$\mathfrak{K}_{VE} = \mathfrak{L}_V \cdot \mathfrak{U}_{VE} - P_{VE}$	DD, mesh \rightarrow block matrices
U_{VI}	$\mathfrak{K}_{VI} = \mathfrak{L}_V \cdot U_{VI} - P_{VI}$	DD, mesh \rightarrow block matrices
Modify		
	$\mathfrak{K}_E := \mathfrak{K}_E - \mathfrak{L}_{EV} \cdot \mathfrak{U}_{VE}$	same matrix type
	$K_{EI} := K_{EI} - \mathfrak{L}_{EV} \cdot U_{VI}$	same matrix type
	$K_{IE} := K_{IE} - L_{IV} \cdot \mathfrak{U}_{VE}$	same matrix type
	$K_I := K_I - L_{IV} \cdot U_{VI}$	same matrix type
Determine		
$\mathfrak{L}_E, \mathfrak{U}_E$	$\mathfrak{K}_E = \mathfrak{L}_E \cdot \mathfrak{U}_E - P_E$	DD, mesh \rightarrow block matrices
L_{IE}	$K_{IE} = L_{IE} \cdot \mathfrak{U}_E - P_{IE}$	DD \rightarrow block matrices
U_{EI}	$K_{EI} = \mathfrak{L}_E \cdot U_{EI} - P_{EI}$	DD \rightarrow block matrices
Modify		
	$K_I := K_I - L_{IE} \cdot U_{EI}$	same matrix type
Determine		
$\mathfrak{L}_I, \mathfrak{U}_I$	$\mathfrak{K}_I = \mathfrak{L}_I \cdot \mathfrak{U}_I - P_I$	DD \rightarrow block matrices

Alg. 6.14 Parallelisierte $\mathfrak{L}\mathfrak{U}$ -Faktorisierung

Da die redundant gespeicherten Matrizen $\mathfrak{K}_V, \mathfrak{K}_E, \mathfrak{K}_E, \mathfrak{K}_{EV}$ und \mathfrak{K}_{VE} nach der Akkumulation bis zur Zerlegung nicht mehr modifiziert werden, kann und muß die Akkumulation vor der Faktorisierung ausgeführt werden.

In 3D kommt noch ein Block mit den Faces "F" hinzu. Die entsprechende Faktorisierung ist analog, jedoch ergeben sich weitere Forderungen an die Vernetzung.

Betrachtet man den Auflösungsschritt $\mathfrak{m} = \mathfrak{U}^{-1} \cdot \mathfrak{L}^{-1} \cdot \mathfrak{r}$ näher und trägt den mit akkumulierten Matrizen erlaubten Matrix-Vektor-Multiplikationen (5.30) und (5.31) Rechnung, so sieht man aus Gleichung (5.34a), daß 3 Vektortypumwandlungen auftreten müssen. Die Diagonalmatrix R mit der Anzahl der am Knoten anliegenden Teilgebiete wurde in (5.6) definiert.

I) $\underline{\mathbf{r}} := \sum_{i=1}^P A_i^T \underline{\mathbf{r}}_i$ II) $\underline{\mathbf{u}}_V := \mathcal{L}_V^{-1} \underline{\mathbf{r}}_V$ $\underline{\mathbf{u}}_E := \mathcal{L}_E^{-1} (\underline{\mathbf{r}}_E - \mathcal{L}_{EV} \underline{\mathbf{u}}_V)$ $\underline{\mathbf{u}}_I := L_I^{-1} (\underline{\mathbf{r}}_I - L_{IE} \underline{\mathbf{u}}_E - L_{IV} \underline{\mathbf{u}}_V)$ III) $\underline{\mathbf{u}} := R^{-1} \underline{\mathbf{u}}$	IV) $\underline{\mathbf{w}}_I := U_I^{-1} \underline{\mathbf{u}}_I$ $\underline{\mathbf{w}}_E := \mathcal{U}_E^{-1} (\underline{\mathbf{u}}_E - U_{EI} \underline{\mathbf{w}}_I)$ $\underline{\mathbf{w}}_V := \mathcal{U}_V^{-1} (\underline{\mathbf{u}}_V - \mathcal{U}_{VE} \underline{\mathbf{w}}_E - U_{VI} \underline{\mathbf{w}}_I)$ V) $\underline{\mathbf{w}} := \sum_{i=1}^P A_i^T \underline{\mathbf{w}}_i$
---	---

Alg. 6.15 Paralleler Auflösungsschritt $\mathcal{L} \cdot \mathcal{U} \cdot \underline{\mathbf{w}} = \underline{\mathbf{r}}$

Verbesserung : Der Nachteil von Algorithmus 6.15 besteht vor allem darin, daß jeweils gerade ein anderer als der gegebene Vektortyp benötigt wird. Dadurch sind 3 Vektortypumwandlungen mit 2 Akkumulationsschritten notwendig. Dies bedeutet in einem Iterationsverfahren mit der *ILLU* als Vorkonditionierer doppelten Kommunikationsaufwand gegenüber dem ω -Jacobi-Verfahren.

\implies Ein Auflösungsschritt $\underline{\mathbf{w}} = \mathcal{L}^{-1} \cdot \mathcal{U}^{-1} \cdot \underline{\mathbf{r}}$ würde laut (5.33a) nur eine Vektortypumwandlung mit einer Akkumulation benötigen.

6.5.3 Die parallele IUL-Faktorisierung

Zur Überwindung der doppelten Kommunikation in Algorithmus 6.15 wird eine unvollständige $\mathcal{U}\mathcal{L}$ -Faktorisierung benutzt.

Der Durchlaufsinns kehrt sich gegenüber der ILU-Faktorisierung um. Dadurch werden die redundant gespeicherten Matrizen \mathfrak{K}_V , \mathfrak{K}_E , \mathfrak{K}_{EV} und \mathfrak{K}_{VE} während der Faktorisierung lokal verändert, so daß ihre Akkumulation erst kurz vor der entsprechenden Faktorisierung ausgeführt werden darf !

Der Auflösungsschritt ist nur noch eine Anwendung von (5.33a).

Start	K	
Determine		Why parallel ?
U_I, L_I	$K_I = U_I \cdot L_I - P_I$	DD \rightarrow blocks matrices
L_{IE}	$K_{IE} = U_I \cdot L_{IE} - P_{IE}$	DD \rightarrow blocks matrices
L_{IV}	$K_{IV} = U_I \cdot L_{IV} - P_{IV}$	DD \rightarrow blocks matrices
U_{EI}	$K_{EI} = U_{EI} \cdot L_I - P_{EI}$	DD \rightarrow blocks matrices
U_{VI}	$K_{VI} = U_{VI} \cdot L_I - P_{VI}$	DD \rightarrow blocks matrices
Modify		
	$K_E := K_E - U_{EI} \cdot L_{IE}$	same matrix type
	$K_{EV} := K_{EV} - U_{EI} \cdot L_{IV}$	same matrix type
	$K_{VE} := K_{VE} - U_{VI} \cdot L_{IE}$	same matrix type
	$K_V := K_V - U_{VI} \cdot L_{IV}$	same matrix type
Accumulate	$\mathfrak{K}_E, \mathfrak{K}_{EV}, \mathfrak{K}_{VE}$	
e.g.	$\mathfrak{K}_{EV} := \sum_{i=1}^P A_{E,i}^T K_{EV,i} A_{V,i}$	—
Determine		
$\mathfrak{U}_E, \mathfrak{L}_E$	$\mathfrak{K}_E = \mathfrak{U}_E \cdot \mathfrak{L}_E - P_E$	DD, mesh \rightarrow blocks matrices
\mathfrak{U}_{VE}	$\mathfrak{K}_{VE} = \mathfrak{U}_{VE} \cdot \mathfrak{L}_E - P_{VE}$	DD, mesh \rightarrow blocks matrices
\mathfrak{L}_{EV}	$\mathfrak{K}_{EV} = \mathfrak{U}_E \cdot \mathfrak{L}_{EV} - P_{EV}$	DD, mesh \rightarrow blocks matrices
Modify		
	$K_V := K_V - \mathfrak{U}_{VE} \cdot R_E^{-1} \cdot \mathfrak{L}_{EV}$	same matrix type (5.34a)
Accumulate	\mathfrak{K}_V	—
Determine		
$\mathfrak{U}_V, \mathfrak{L}_V$	$\mathfrak{K}_V = \mathfrak{U}_V \cdot \mathfrak{L}_V - P_V$	mesh \rightarrow diagonal matrix

Alg. 6.16 Parallelisierte IUL-Faktorisierung

I) $\underline{u}_I := U_I^{-1} \underline{r}_I$ $\underline{u}_E := \mathfrak{U}_E^{-1} (\underline{r}_E - U_{EI} \underline{u}_I)$ $\underline{u}_V := \mathfrak{U}_V^{-1} (\underline{r}_V - \mathfrak{U}_{VE} \underline{u}_E - U_{VI} \underline{u}_I)$	III) $\underline{w}_V := \mathfrak{L}_V^{-1} \underline{u}_V$ $\underline{w}_E := \mathfrak{L}_E^{-1} (\underline{u}_E - \mathfrak{L}_{EV} \underline{w}_V)$ $\underline{w}_I := L_I^{-1} (\underline{u}_I - L_{IE} \underline{w}_E - L_{IV} \underline{w}_V)$
II) $\underline{u} := \sum_{i=1}^P A_i^T \underline{u}_i$	

Alg. 6.17 Paralleler Auflösungsschritt $\mathfrak{U} \cdot \mathfrak{L} \cdot \underline{w} = \underline{r}$

6.5.4 Vergleich von ILU- und IUL-Faktorisierung

Die Lösungsansätze in den beiden vorangegangenen Abschnitten unterscheiden sich kaum bzgl. der Arithmetik, dagegen etwas im Kommunikationsverhalten, Tabelle 6.1 faßt das Kommunikationsverhalten zusammen.

Tab. 6.1 Kommunikation in ILU und IUL

	ILU : $\mathcal{L} \cdot \mathcal{U}$	IUL : $\mathcal{U} \cdot \mathcal{L}$
Vorbereitung	Matrixakkumulation	—
Faktorisierung	—	Matrixakkumulation
Auflösung	$2 \times$ Vektorakkumulation	$1 \times$ Vektorakkumulation

Die für die einmalige Assemblierung der Matrix notwendige Kommunikation muß in jedem Falle für die Kanten- und Crosspointknoten separat durchgeführt werden. Somit ist die Kommunikationszeit in den Algorithmen 6.14 und 6.16 identisch. Entsprechend der FE-Diskretisierung liegt die Steifigkeitsmatrix \mathbf{K} in jedem Falle verteilt gespeichert vor, so daß diese Kommunikation nicht eingespart werden kann (Abschnitt 5.1).

Mit Hinblick auf die Kommunikation ist die IUL-Faktorisierung (6.16) im parallelen Fall zu empfehlen.

6.5.5 Die IUL-Faktorisierung in 3D

In 3D kommt mit den Faces, d.h. 2D-Interfaces, noch eine weitere Gruppe von Knoten hinzu. Damit sieht ein Vektor wie $(\underline{u}_V^T, \underline{u}_E^T, \underline{u}_F^T, \underline{u}_I^T)^T$ aus. Analoge neue Blöcke treten in den Matrizen auf. Während der Auflösungsschritt in Alg. 6.18 analog zu Alg. 6.17 ist, muß man bei der Faktorisierung etwas aufpassen.

6.5.6 Die IUL-Faktorisierung mit reduziertem Besetztheitsmuster

Im 3d-Fall bzw. bei Verwendung von bilinearen/quadratischen/... Elementen in 2d können die Matrixanforderungen der Sätze 5.1. und 5.2. aus Abschnitt 5.1 nicht mehr eingehalten werden. Hier muß an Stelle der Originalmatrix \mathfrak{K} die Forderungen erfüllende Matrix $\tilde{\mathfrak{K}}$ faktorisiert werden, d.h. alle störenden Einträge werden entfernt bzw. auf die Hauptdiagonalen verteilt. In jedem Falle muß aber auch das (meist als Indexfeld gespeicherte) Besetztheitsmuster entsprechend geändert werden !

Start	K	
Determine		Why parallel ?
U_I, L_I	$K_I = U_I \cdot L_I$	DD \rightarrow blocks matrices
L_{IE}	$K_{IE} = U_I \cdot L_{IE}$	DD \rightarrow blocks matrices
L_{IV}	$K_{IV} = U_I \cdot L_{IV}$	DD \rightarrow blocks matrices
U_{EI}	$K_{EI} = U_{EI} \cdot L_I$	DD \rightarrow blocks matrices
U_{VI}	$K_{VI} = U_{VI} \cdot L_I$	DD \rightarrow blocks matrices
	\vdots	
Modify		
	$K_E := K_E - U_{EI} \cdot L_{IE}$	same matrix type
	$K_{EV} := K_{EV} - U_{EI} \cdot L_{IV}$	same matrix type
	$K_{VE} := K_{VE} - U_{VI} \cdot L_{IE}$	same matrix type
	$K_V := K_V - U_{VI} \cdot L_{IV}$	same matrix type
	\vdots	
Accumulate	$\mathfrak{K}_F, \mathfrak{K}_{FE}, \mathfrak{K}_{EF}, \mathfrak{K}_{FV}, \mathfrak{K}_{VF}$	
e.g.	$\mathfrak{K}_{FV} := \sum_{i=1}^P A_{F,i}^T K_{FV,i} A_{V,i}$	—
Determine		
$\mathfrak{U}_F, \mathfrak{L}_F$	$\mathfrak{K}_F = \mathfrak{U}_F \cdot \mathfrak{L}_F$	DD, mesh \rightarrow blocks
\mathfrak{U}_{EF}	$\mathfrak{K}_{EF} = \mathfrak{U}_{EF} \cdot \mathfrak{L}_F$	DD, mesh \rightarrow blocks
\mathfrak{L}_{FE}	$\mathfrak{K}_{FE} = \mathfrak{U}_F \cdot \mathfrak{L}_{FE}$	DD, mesh \rightarrow blocks
\mathfrak{U}_{VF}	$\mathfrak{K}_{VF} = \mathfrak{U}_{VF} \cdot \mathfrak{L}_F$	DD, mesh \rightarrow blocks
\mathfrak{L}_{FV}	$\mathfrak{K}_{FV} = \mathfrak{U}_F \cdot \mathfrak{L}_{FV}$	DD, mesh \rightarrow blocks
Modify		
	$K_E := K_E - \mathfrak{U}_{EF} \cdot R_F^{-1} \cdot \mathfrak{L}_{FE}$	same matrix type
	$K_{EV} := K_{EV} - \mathfrak{U}_{EF} \cdot R_F^{-1} \cdot \mathfrak{L}_{FV}$	same matrix type
	$K_{VE} := K_{VE} - \mathfrak{U}_{VF} \cdot R_F^{-1} \cdot \mathfrak{L}_{FE}$	same matrix type
	$K_V := K_V - \mathfrak{U}_{VF} \cdot R_F^{-1} \cdot \mathfrak{L}_{FV}$	same matrix type
Accumulate	$\mathfrak{K}_E, \mathfrak{K}_{EV}, \mathfrak{K}_{VE}$	
e.g.	$\mathfrak{K}_{EV} := \sum_{i=1}^P A_{E,i}^T K_{EV,i} A_{V,i}$	—
Determine		
$\mathfrak{U}_E, \mathfrak{L}_E$	$\mathfrak{K}_E = \mathfrak{U}_E \cdot \mathfrak{L}_E$	DD, mesh \rightarrow blocks
\mathfrak{U}_{VE}	$\mathfrak{K}_{VE} = \mathfrak{U}_{VE} \cdot \mathfrak{L}_E$	DD, mesh \rightarrow blocks
\mathfrak{L}_{EV}	$\mathfrak{K}_{EV} = \mathfrak{U}_E \cdot \mathfrak{L}_{EV}$	DD, mesh \rightarrow blocks
Modify	$K_V := K_V - \mathfrak{U}_{VE} \cdot R_E^{-1} \cdot \mathfrak{L}_{EV}$	same matrix type
Accumulate	\mathfrak{K}_V	—
Determine		
$\mathfrak{U}_V, \mathfrak{L}_V$	$\mathfrak{K}_V = \mathfrak{U}_V \cdot \mathfrak{L}_V$	mesh \rightarrow diagonal matrix

Alg. 6.18 Parallelisierte IUL-Faktorisierung in 3D

6.6 Der Schurkomplement CG

6.6.1 Das Schurkomplement

Der Begriff des Schurkomplements geht auf den Mathematiker Isaii Schur¹ (1894-1939 in Deutschland arbeitend) zurück. Wir führen den Begriff von einem anderen Ansatz her ein.

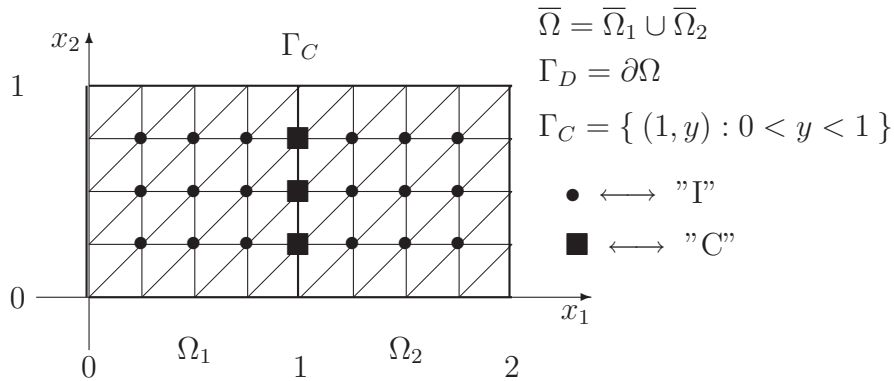


Fig. 6.4 2 Superelemente (=Teilgebiete) mit Diskretisierung

Die FE-Knotenverteilung in Fig.6.4 induziert im zu lösenden Gleichungssystem (6.1) eine Blockstruktur :

$$\begin{pmatrix} K_C & K_{CI,1} & K_{CI,2} \\ K_{IC,1} & K_{I,1} & 0 \\ K_{IC,2} & 0 & K_{I,2} \end{pmatrix} \cdot \begin{pmatrix} \underline{u}_C \\ \underline{u}_{I,1} \\ \underline{u}_{I,2} \end{pmatrix} = \begin{pmatrix} \underline{f}_C \\ \underline{f}_{I,1} \\ \underline{f}_{I,2} \end{pmatrix}. \quad (6.6)$$

Eine Gaußelimination des oberen Dreiecks der Matrix in (6.6) ergibt das gestaffelte Gleichungssystem

$$\begin{pmatrix} S_C & 0 \\ K_{IC} & K_I \end{pmatrix} \cdot \begin{pmatrix} \underline{u}_C \\ \underline{u}_I \end{pmatrix} = \begin{pmatrix} \underline{g}_C \\ \underline{f}_I \end{pmatrix}, \quad (6.7)$$

mit dem Schurkomplement

$$\begin{aligned} S_C &= K_C - K_{CI}K_I^{-1}K_{IC} \\ &= (K_{C,1} - K_{CI,1}K_{I,1}^{-1}K_{IC,1}) + (K_{C,2} - K_{CI,2}K_{I,2}^{-1}K_{IC,2}) \end{aligned} \quad (6.8)$$

und der modifizierten rechten Seite

$$\underline{g}_C = \underline{f}_C - K_{CI}K_I^{-1}\underline{f}_I.$$

¹1875 (Mohilev am Dnjepr) - 1941 (Tel Aviv)

Hierbei ist $K_{IC} = \begin{pmatrix} K_{IC,1} \\ K_{IC,2} \end{pmatrix}$ und $K_I = \text{blockdiag}\{K_{I,1}, K_{I,2}\}$. Die Matrix S_C ist i.a. vollbesetzt, in bestimmten Fällen läßt sich das Schurkomplement explizit angeben.

Der Hauptnutzen obigen Gleichungssystems (6.7) besteht darin, daß durch die Blockdiagonalstruktur von K_I die Invertierung der lokalen Matrizen $K_{I,1}$ und $K_{I,2}$ hintereinander für jedes Superelement ausgeführt werden kann, wodurch den Hauptspeicher übersteigende Probleme durch Auslagerung der nicht benötigten Daten auf externe Datenträger trotzdem gelöst werden können. Dieses Vorgehen wurde in den 60/70-er Jahren als Element-by-Element Methode (EBE) bekannt.

```

I)   $\underline{g}_C := \underline{f}_C$ 
    for  $s := 1$  to  $P$  do
         $\underline{g}_C := \underline{g}_C - K_{CI,s} \cdot K_{I,s}^{-1} \cdot \underline{f}_{I,s}$ 
    end

II)  Löse  $S_C \cdot \underline{x}_C = \underline{g}_C$ 

III) for  $s := 1$  to  $P$  do
         $\underline{x}_{I,s} := K_{I,s}^{-1} \cdot (\underline{f}_{I,s} - K_{IC,s} \cdot \underline{x}_C)$ 
    end

```

Alg. 6.19 Element-by-Element Methode

Da das Schurkomplement nur in den seltensten Fällen explizit angebar ist, löst man oft das reduzierte Gleichungssystem (mit vollbesetzter Matrix !) in II) mittels eines geeigneten Iterationsverfahrens. Der Vorteil besteht darin, daß die in jedem Iterationsverfahren notwendige Matrix-mal-Vektor Multiplikation parallel ausgeführt werden kann :

$$\begin{aligned} \underline{r}_C &:= S_C \underline{w}_C \\ &= \sum_{s=1}^P (K_{C,s} - K_{CI,s} K_{I,s}^{-1} K_{IC,s}) \cdot \underline{w}_{C,s} . \end{aligned}$$

Die superelementweise Invertierung der $K_{I,s}$ geschieht durch einmalige LU- bzw. Choleskyzerlegung, wodurch die weitere Anwendung von $K_{I,s}^{-1}$ nur noch in den Rücksubstitutionsschritten besteht.

6.6.2 Der parallele Schurkomplement-CG

Falls die Matrix in (6.1) symmetrisch und positiv definit ist, so ist auch das entsprechende Schurkomplement symmetrisch und positiv definit. Damit ist Zeile II) des Algorithmus 6.19 mittels eines CG iterativ lösbar.

Aufgabe 6.2. Schreiben Sie den parallelisierten Schurkomplement-CG auf, betrachten Sie insbesondere die Matrix-Vektor Multiplikation.

Für den Vorkonditionierungsschritt im Schurkomplement-CG wurden verschiedene Techniken zur Approximation des Schurkomplements entwickelt, einige sind in [Dry 84, BPS 89, TCK 92] zu finden. Ein Ansatz über BEM und Multigridmethoden ist in [CKL 96] zu finden.

6.7 Die Multigridmethode

Es existiere eine Folge von regulären (FE-) Netzen \mathfrak{T}_k ($k=\overline{1,\ell}$), wobei das feinere Netz/Gitter \mathfrak{T}_{k+1} aus dem gröberen \mathfrak{T}_k hervorgegangen ist (einfachster Fall: Viertelung aller Dreiecke in 2D). Für diese Gitterhierarchie soll $\mathfrak{T}_1 \subset \mathfrak{T}_2 \subset \dots \subset \mathfrak{T}_\ell$ gelten.

Die gegebene Differentialgleichung wird auf jedem Gitter \mathfrak{T}_k diskretisiert, so daß eine Folge von Gleichungssystemen

$$K_k \cdot \underline{u}_k = \underline{f}_k \quad (6.9)$$

mit den jeweiligen dünnbesetzten, symmetrischen und positiv definiten Steifigkeitsmatrizen K_k entsteht.

Eine Analyse des (nichtoptimalen) Konvergenzverhaltens der Iterationsverfahren in den Abschnitten 6.3 und 6.4, mittels einer Zerlegung des Fehlers in die Eigenfrequenzen der Matrix K , liefert :

- Die Fehleranteile mit hohen Frequenzen werden sehr schnell reduziert.
- Die niedrigfrequenten Fehleranteile bestimmen das langsame Konvergenzverhalten der klassischen Verfahren.

⇒ Man braucht ein Verfahren, welches die niedrigfrequenten Fehleranteile e_{low} kostengünstig reduziert, bei Beibehaltung der schnellen Reduktion hochfrequenten Fehleranteile e_{high} .

⇒ *Zweigitteridee* :

Reduziere e_{high} auf dem Gitter \mathfrak{T}_ℓ (Glättungsschritt) projiziere den Restfehler auf das gröbere Gitter $\mathfrak{T}_{\ell-1}$, löse dort exakt, interpoliere die erhaltene Korrektur (Defektkorrektur) wieder auf das Feingitter \mathfrak{T}_ℓ und addiere sie zur bereits

erhaltenen Lösung. Oft werden hinterher die verbliebenen hochfrequenten Fehleranteile nochmals geglättet.

- Die *Multigriddidee* ersetzt nun den exakten Löser bei der Defektkorrektur wiederum durch eine Zweigitteridee auf dem Gitter $\ell - 1$, usw., bis hinunter zum größten Gitter \mathfrak{T}_1 , welches exakt gelöst wird.

6.7.1 Der serielle Algorithmus

Zur Formulierung des Multigridalgorithmus benötigen wir die folgenden zusätzlichen Matrizen und Vektoren :

- I_k^{k-1} : Restriktionsoperator um Daten vom feinen auf das grobe Gitter zu transferieren.
- I_{k-1}^k : Interpolationsoperator um Daten vom groben auf das feine Gitter zu transferieren.
- $S_{\text{pre}}^{\nu_{\text{pre}}}$: Vorglättungsoperator zur Reduktion von e_{high} , ν_{pre} -mal angewandt.
- $S_{\text{post}}^{\nu_{\text{post}}}$: Nachglättungsoperator zur Reduktion von e_{high} , ν_{post} -mal angewandt.
- \underline{d}_k : Defekt auf k -tem Gitter.
- \underline{w}_k : Korrektur auf k -tem Gitter.
- MGM^γ : Rekursive Multigridprozedur, γ -mal aufgerufen ($\gamma = 1$ - V-Zyklus, $\gamma = 2$ - W-Zyklus)

Als Vor- bzw. Nachglättungsoperatoren können z.B. die Iterationsverfahren aus 6.3 - 6.5 gewählt werden.

if ($k == 1$) then	
Solve $K_1 \cdot \underline{u}_1 = \underline{f}_1$	Grobgridlöser
else	
$\widehat{\underline{u}}_k := S_{\text{pre}}^{\nu_{\text{pre}}}(K_k, \underline{u}_k, \underline{f}_k)$	Vorglättung
$\underline{d}_k := \underline{f}_k - K_k \cdot \widehat{\underline{u}}_k$	Defektberechnung
$\underline{d}_{k-1} := I_k^{k-1} \cdot \underline{d}_k$	Restriktion des Defektes
$\underline{w}_{k-1}^0 := 0$	
$\underline{w}_{k-1} := \text{MGM}^\gamma(K_{k-1}, \underline{w}_{k-1}^0, \underline{d}_{k-1}, k - 1)$	Defektsystem
$\underline{w}_k := I_{k-1}^k \cdot \underline{w}_{k-1}$	Interpolation der Korrektur
$\widetilde{\underline{u}}_k := \widehat{\underline{u}}_k + \underline{w}_k$	Addition der Korrektur
$\underline{u}_k := S_{\text{post}}^{\nu_{\text{post}}}(K_k, \widetilde{\underline{u}}_k, \underline{f}_k)$	Nachglättung
endif	

Alg. 6.20 Seriell Multigrid : $\text{MGM}(K_k, \underline{u}_k, \underline{f}_k, k)$

Falls Multigrid als Vorkonditionierer im CG verwandt werden soll, muß der entsprechende Multigriditerationsoperator symmetrisch sein. In diesem Falle werden Interpolation und Restriktion, sowie Vor- und Nachglättung so gewählt, daß $I_k^{k-1} = (I_{k-1}^k)^T$ und $S_{\text{pre}} = (S_{\text{post}})^T$ mit $\nu_{\text{pre}} = \nu_{\text{post}}$ gelten. Das Defektsystem auf dem größten Gitter wird direkt gelöst.

6.7.2 Die parallelen Komponenten

Trivialerweise ist der Multigridalgorithmus parallelisierbar, falls seine einzelnen Komponenten Interpolation, Restriktion, Glättung und Grobgitterlöser parallelisierbar sind. Wenn wir die nichtüberlappende Elementaufteilung aus Abschnitt 5.1 bereits auf dem größten Gitter \mathfrak{T}_1 benutzen, so bleibt diese Aufteilung auf sämtlichen feineren Gittern erhalten.

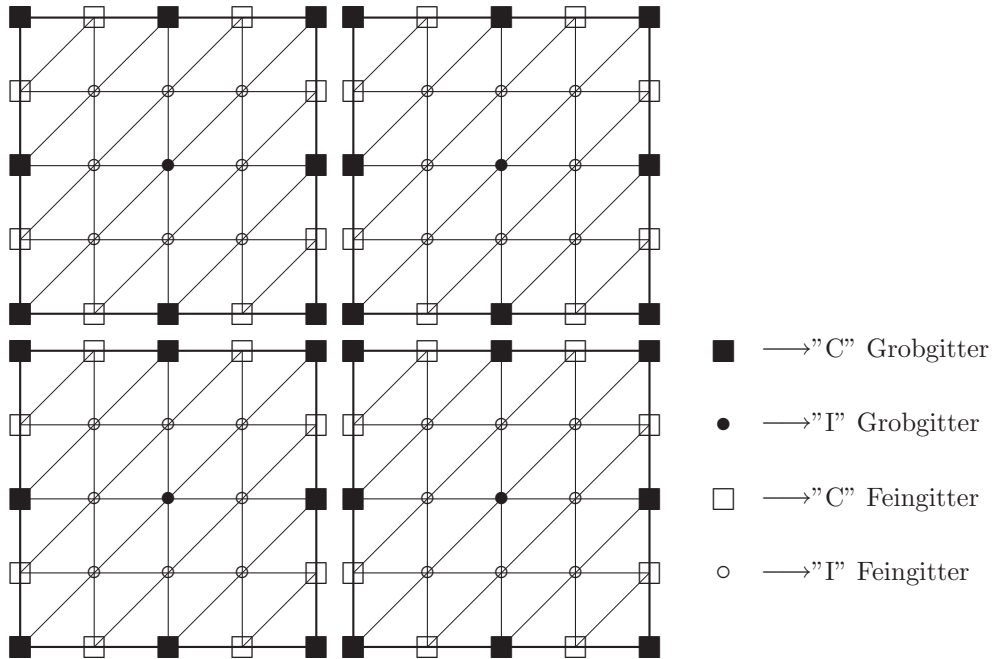


Fig. 6.5 Nichtüberlappende Elemente bei zwei Gittern

Die Steifigkeitsmatrizen K_k seien verteilt gespeichert. Aus den Erfahrungen der vorangegangenen Abschnitten (insbesondere 6.1) ist in Alg. 6.20 folgende Zuordnung der Vektoren zu den zwei Typen a priori erkennbar:

- Akkumulierte Vektoren : $\hat{\mathbf{u}}_k, \tilde{\mathbf{u}}_k, \hat{\mathbf{w}}_k$.
- Verteilte Vektoren : $\underline{\mathbf{f}}_k, \underline{\mathbf{d}}_k$.
- Noch frei: $\underline{\mathbf{u}}_k, \underline{\mathbf{w}}_{k-1}, \underline{\mathbf{w}}_{k-1}^0, \underline{\mathbf{d}}_{k-1}$.

Die Interpolation

Betrachtet man Fig. 6.5 und die entsprechende, aus der Netzverfeinerung hervorgehende Interpolation, so ist für die 3 Knotenklassen V, E, I folgendes erkennbar :

1. Die Interpolation $I_{V,k-1}^k$ innerhalb der Crosspoints ist die (evtl. skalierte) Identität I .
2. Crosspoints (V) gehören vollständig zum größten Gitter und werden somit nie durch Kanten- (E) oder innere (I) Knoten interpoliert
 $\implies I_{VE,k-1}^k = I_{VI,k-1}^k = 0$.
3. Die Interpolationmatrix auf den Kanten zerfällt in Blöcke
 $\implies I_{E,k-1}^k = \text{blockdiag}\{I_{E_j,k-1}^k\}_{j=1, \overline{\text{NumEdges}}}$.
4. Kantenknoten werden nie durch innere Knoten interpoliert
 $\implies I_{EI,k-1}^k = 0$.
5. Offensichtlich zerfällt auch die Interpolationmatrix auf den inneren Knoten (I) in Blöcke
 $\implies I_{I,k-1}^k = \text{blockdiag}\{I_{I_s,k-1}^k\}_{s=1, \overline{P}}$.

Somit besitzt die Interpolationsmatrix I_{k-1}^k eine Blockstruktur, in welcher nur das untere Blockdreieck ungleich 0 ist. Die Blockstruktur ist insbesondere so geartet, daß (5.30) anwendbar wird.

- \implies Interpolation ist akkumuliert : \mathfrak{J}_{k-1}^k .
 \implies Vektor $\underline{\mathbf{w}}_{k-1}$ ist akkumuliert.

Man hätte die Interpolationsmatrix prinzipiell auch vom verteilten Typ wählen können, hätte sich dann aber in der Interpolation bzw. in der nachfolgenden Addition eine Kommunikation bei der erforderlichen Typumwandlung eingehandelt.

Keine Kommunikation in der Interpolation !

Die Restriktion

Wählt man die Restriktion als die transponierte Interpolation, d.h. $I_k^{k-1} = (I_{k-1}^k)^T$, ergibt sich die Dreiecksstruktur der Matrix, sodaß (5.31) anwendbar ist.

- \implies Restriktion ist akkumuliert : \mathfrak{J}_k^{k-1} .
 \implies Vektor $\underline{\mathbf{d}}_{k-1}$ ist verteilt gespeichert, $\underline{\mathbf{d}}_k$ sowieso.

Hier ist eine Abspeicherung als verteilte Matrix wiederum prinzipiell möglich, würde jedoch vor der Restriktion eine Akkumulation von $\underline{\mathbf{d}}_k$ erfordern.

Eine Verwendung der Injektion als Restriktion ändert nichts an diesen Aussagen.

Keine Kommunikation in der Restriktion !

Die Glättung

Die Glättungsoperatoren S_{pre} und S_{post} wie ω -Jacobi, Gauß-Seidel und ILU (6.3 - 6.5) benötigen als Input stets eine akkumulierte Startlösung $\underline{\mathbf{u}}^0$, eine verteilt gespeicherte rechte Seite $\underline{\mathbf{f}}$ und eine verteilte Matrix \mathbf{K} . Die iterierte Lösung $\underline{\mathbf{u}}$ ist stets akkumuliert. Innerhalb des Iterationsverfahrens ist in jedem Falle mindestens eine Akkumulation nötig.

⇒ Vektor $\underline{\mathbf{u}}_k$ ist akkumuliert, $\widehat{\underline{\mathbf{u}}}_k$ und $\widetilde{\underline{\mathbf{u}}}_k$ sowieso.

⇒ Vektor $\underline{\mathbf{w}}_k^0$ ist akkumuliert.

Kommunikation innerhalb der Glättung !

Der Grobgitterlöser

Das eigentliche Parallelisierungsproblem in (Multilevel- und) Mehrgitterverfahren ist das exakt zu lösende Grobgitterproblem

$$\sum_{s=1}^P A_{s,1}^T \mathbf{K} A_{s,1} \underline{\mathbf{u}}_1 = \sum_{s=1}^P A_{s,1}^T \underline{\mathbf{f}}_{s,1} . \quad (6.10)$$

Folgende Lösungsansätze sind denkbar und werden verwendet :

1. Direkter Löser - seriell

Größtes Gitter ist so klein, daß es ein Prozessor \mathbb{P}_0 zusätzlich speichern kann.

- Assemblierung \mathbf{K}_1 (REDUCE).
- Assemblierung $\underline{\mathbf{f}}_1$ (REDUCE).
- Prozessor \mathbb{P}_0 löst $\mathbf{K}_1 \cdot \underline{\mathbf{u}}_1 = \underline{\mathbf{f}}_1$.
- Verteilen von $\underline{\mathbf{u}}_1$ (SCATTER).

Diese Vorgehen ist allerdings rein sequentiell !!

2. Direkter Löser - parallel

Auch das größte Gitter wird verteilt gespeichert und das Grobgittersystem direkt parallel gelöst. Im Auflösungsprozeß treten Loadinbalancen auf.

3. Iterative Löser - parallel

Hier wird keine Assemblierung von \mathbf{K}_1 und $\underline{\mathbf{f}}_1$ benötigt, jedoch tritt in jedem Iterationsschritt mindestens eine Akkumulation auf (siehe 6.1 - 6.6).

Achtung : Die evtl. beabsichtigte Symmetrie des Multigridoperators wird bei den CG-ähnlichen Verfahren zerstört. Um sie zu erhalten, muß z.B. eine lexikographisch vorwärtige Gauß-Seidel Iteration stets mit einer rückwärtigen gekoppelt werden.

Prinzipiell ergibt sich beim Grobgitterlöser durch das ungünstige Verhältnis von Arithmetik zu Kommunikation eine Verschlechterung der Parallelisierung. Dies äußert sich bei 32 und mehr Prozessoren ganz deutlich im W-Zyklus, der sehr oft auf die groben und insbesondere das größte Gitter zugreift.

Ein Lösungsversuch besteht in der Verteilung des Grobgitters auf weniger als P Prozessoren, jedoch ist dann bei der Interpolation und Restriktion zwischen bestimmten Gittern ℓ_0 und $\ell_0 + 1$ Kommunikation notwendig.

Kommunikation im Grobgitterlöser !

6.7.3 Der parallele Algorithmus

Unter Beachtung im vorigen Abschnitt behandelten Anforderungen an die Multigridkomponenten, ist die Parallelisierung von Alg. 6.20 recht einfach.

if ($k == 1$) then		
Solve $\sum_{s=1}^P A_{s,1}^T K A_{s,1} \underline{u}_1 = \sum_{s=1}^P A_{s,1}^T \underline{f}_{s,1}$		Grobgitterlöser
else		
$\hat{\underline{u}}_k := S_{\text{pre}}^{\nu}(\mathbf{K}_k, \underline{u}_k, \underline{f}_k)$		Vorglättung
$\underline{d}_k := \underline{f}_k - \mathbf{K}_k \cdot \hat{\underline{u}}_k$		Defektberechnung
$\underline{d}_{k-1} := \mathcal{J}_k^{k-1} \cdot \underline{d}_k$		Restriktion des Defektes
$\underline{w}_{k-1}^0 := 0$		
$\underline{w}_{k-1} := \text{PMGM}^\gamma(\mathbf{K}_{k-1}, \underline{w}_{k-1}^0, \underline{d}_{k-1}, k-1)$		Defektsystem
$\underline{w}_k := \mathcal{J}_{k-1}^k \cdot \underline{w}_{k-1}$		Interpolation der Korrektur
$\tilde{\underline{u}}_k := \hat{\underline{u}}_k + \underline{w}_k$		Addition der Korrektur
$\underline{u}_k := S_{\text{post}}^{\nu}(\mathbf{K}_k, \tilde{\underline{u}}_k, \underline{f}_k)$		Nachglättung
endif		

Alg. 6.21 Paralleles Multigrid : $\text{PMGM}(\mathbf{K}_k, \underline{u}_k, \underline{f}_k, k)$

6.8 Vorkonditionierung Iterativer Verfahren

Die Typumwandlung $\underline{\mathbf{w}} = \sum_{j=1}^P A_j^T \underline{\mathbf{r}}_j$ im CG (Fig. 6.2) als Matrix geschrieben, stellt sich als akkumulierte Einheitsmatrix dar und ist somit äquivalent zu Gleichung (5.32b) mit den Komponenten $\mathfrak{M}_D = \mathfrak{I}$ und $\mathfrak{M}_L = \mathfrak{M}_U = 0$. Daher suchen wir nach Vorkonditionierern welcher nicht wesentlich mehr Kommunikation benötigen als diese Umwandlung von verteilten zu einem akkumulierten Vektor.

6.8.1 Akkumulierte Vorkonditionierer

Nunmehr läßt sich ein akkumulierter Vorkonditionierer aus den Gleichungen (5.32b), (5.33a) und (5.34a) auswählen. Insbesondere letzterer ist interessant.

Bemerkung 6.2. Die in den Abschnitten 6.3 bis 6.7 behandelten Iterationsverfahren sind jeweils als Vorkonditionierer im CG (spd) bzw. GMRES geeignet. Im Falle einer symmetrischen Steifigkeitsmatrix paßt die ASM-DD-Vorkonditionierung [HLM 91] unter Benutzung von Schurkomplementvorkonditionierern [BPS 89, TCK 92] exakt ins Schema von Gleichung (5.33a).

6.8.2 Verteilte Vorkonditionierer

Möchte man eine verteilte Vorkonditionierungsmatrix $\mathbf{C}^{-1} = \sum_{i=1}^p A_i^T \mathbf{C}_i^{-1} A_i$ benutzen ist eine zweifache Vektortypumwandlung notwendig (Abschnitt 5.1). Deshalb sieht der Vorkonditionierungsschritt wie folgt aus :

$$\underline{\mathbf{w}} = \mathbf{C}^{-1} \sum_{j=1}^P A_j^T \underline{\mathbf{r}}_j := \sum_{i=1}^P A_i^T \left(\mathbf{C}_i^{-1} \cdot A_i \sum_{j=1}^P A_j^T \underline{\mathbf{r}}_j \right) . \quad (6.11)$$

Die Matrix \mathbf{K} ist nicht assembliert und daher stellen die Teilmatrizen \mathbf{K}_i eine PDE 2. Ordnung in Ω_i mit homogenen Neumann Randbedingungen an den inneren Rändern $\partial\Omega_i \setminus \partial\Omega$ dar. Für den Laplaceoperator kann dies zu einer singulären Matrix \mathbf{K}_i führen! Daher ist die Wahl $\mathbf{C}_i = \mathbf{K}_i$ ungünstig.

Eine andere Möglichkeit besteht in der Akkumulation der Steifigkeitsmatrix \mathbf{K} und der Wahl $\mathbf{C}_i = \mathfrak{K}_i$. Der lokale Vorkonditionierer \mathbf{C}_i repräsentiert nunmehr eine PDE 2. Ordnung im Gebiet $\tilde{\Omega}_i$ mit homogenen Dirichlet Randbedingungen, wobei $\tilde{\Omega}_i$ das um die nächste Schicht von Elementen vergrößerte Gebiet Ω_i darstellt. Damit ist dieser Vorkonditionierer vom überlappenden Typ, siehe [SBG 96].

7 Die Parallele Lösung von Erhaltungsgleichungen

Die in diesem Kapitel dargestellten Auflösungstechniken für Erhaltungsgleichungen sind bewußt simplifiziert dargestellt, um den eigentlichen Zweck dieses Kapitels, *die Parallelisierungsansätze* nicht in den Hintergrund treten zu lassen.

7.1 Die inkompressiblen Navier-Stokes-Gleichungen

Die Darstellung der Navier-Stokes-Gleichungen in konvektiver Form und ihr serieller Auflösungsprozeß folgen im wesentlichen der Dissertation von Volker John [Joh 97].

7.1.1 Die Differentialgleichungen

Aus der Impulsbilanz und der Massenerhaltung einer inkompressiblen (homogenen) Flüssigkeit ergeben sich die **inkompressiblen Navier-Stokes-Gleichungen**.

Gesucht ist $u(x, t) \in [C^2(\Omega) \cup C(\bar{\Omega})] \times [C^1((0, T]) \cup C([0, T])]$, so daß gilt :

$u_t - \nu \Delta u + (u \cdot \nabla) u + \nabla p = f$	in $\Omega \times (0, T]$	(7.1)
$\nabla \cdot u = 0$	in $\Omega \times (0, T]$	
$u(t) = g(t)$	auf $\partial\Omega \times (0, T]$	
$u(x) = u_0(x)$	für $t = 0$	

In den Gleichungen 7.1 bezeichnen :

- u - Vektor der Geschwindigkeitskomponenten,
- p - Druck (genauer $p := p/\rho$),
- f - Kraftfeld
- ν - kinematische Viskosität
- g - Dirichlet-RB
- u_0 - Anfangsbedingungen.

Falls Geschwindigkeit und Druck zeitunabhängig sind, erhält man die **stationären Navier-Stokes-Gleichungen**. Gesucht ist $u(x) \in C^2(\Omega) \cup C(\bar{\Omega})$, so daß gilt :

$$\begin{array}{rcl} -\nu\Delta u + (u \cdot \nabla)u + \nabla p = f & \text{in } \Omega & \\ \nabla \cdot u = 0 & \text{in } \Omega & \\ u = g & \text{auf } \partial\Omega & \end{array} \quad (7.2)$$

Eine weitere Vereinfachung ergibt durch Weglassen des Konvektionsterms $(u \cdot \nabla)u$ die **Stokes-Gleichungen**

$$\begin{array}{rcl} -\nu\Delta u + \nabla p = f & \text{in } \Omega & \\ \nabla \cdot u = 0 & \text{in } \Omega & \\ u = g & \text{auf } \partial\Omega & \end{array} \quad (7.3)$$

Die zu (7.1) - (7.3) passenden Variationsformulierungen mit $u \in H^1$, $p \in L_2$ und passenden Ansatzräumen lassen sich leicht herleiten (siehe [Joh 97],...). Die anschließende Diskretisierung mittels FEM, FVM liefert

- eine Folge (7.1)
- von nichtlinearen, nichtsymmetrischen (7.2)
- und indefiniten (7.3) Gleichungssystemen.

Um die Stabilität der Diskretisierung zu sichern (Erfüllung der diskreten inf sup-Bedingung), werden u.a. die in Abschnitt 7.1.3 erwähnten Elemente benutzt.

7.1.2 Die serielle Auflösung

Zur Zeitdiskretisierung von (7.1) benutzen wir das 2-schichtige gewichtete Differenzschema (L bezeichne den Differentialoperator)

$$\frac{y(t_{K+1}) - y(t_K)}{t_{K+1} - t_K} + \sigma L(y(t_{K+1})) + (1 - \sigma)L(y(t_K)) = \sigma f(t_{K+1}) + (1 - \sigma)f(t_K) .$$

Für $\sigma = 0$ erhält man das explizite Schema (Stabilität muß gesichert werden!), für $\sigma = 1$ das rein implizite und für $\sigma = \frac{1}{2}$ das Crank-Nicolson Schema.

Es sind auch 3-schichtige Differenzschemata anwendbar.

Bezeichne

$\tau = t_{K+1} - t_K$: Zeitschrittweite		
M	: Massenmatrix	$\leftarrow (u, v)_{L_2}$	$\leftarrow u$
D	: Diffusionsmatrix	$\leftarrow a(u, v)$	$\leftarrow \Delta u$
$C(u)$: Konvektionsmatrix	$\leftarrow b(u, u, v)$	$\leftarrow (u \cdot \nabla)u$
B	: Gradientenmatrix	$\leftarrow (p, \nabla \cdot v)$	$\leftarrow \nabla p$
B^T	: Divergenzmatrix	$\leftarrow (q, \nabla \cdot u)$	$\leftarrow \nabla \cdot u$
$\underline{u}^K = \underline{u}(t_K)$: Geschwindigkeitsvektor		
$\underline{p}^K = \underline{p}(t_K)$: Druckvektor,		

dann liefert die anschließende Ortsdiskretisierung eine Folge von nichtlinearen, nichtsymmetrischen und indefiniten Gleichungssystemen ($K = 0, 1, \dots$) :

$$\begin{pmatrix} \frac{1}{\tau}M + \sigma(D + C(\underline{u}^{K+1})) & B \\ B^T & 0 \end{pmatrix} \cdot \begin{pmatrix} \underline{u}^{K+1} \\ \underline{p}^{K+1} \end{pmatrix} = \begin{pmatrix} \underline{f} \\ 0 \end{pmatrix} \quad (7.4)$$

mit

$$\underline{\hat{f}} := \sigma \underline{f}(t_{K+1}) + (1 - \sigma)\underline{f}(t_K) + \left[\frac{1}{\tau}M - (1 - \sigma)(D + C(\underline{u}^K)) \right] \underline{u}^K .$$

Ein nichtlineares, aber quasilineares GIS $A(\underline{v})\underline{v} = \underline{g}$ kann u.a. mittels Fixpunktiteration gelöst werden [Idee: $A(\underline{v}^{n+1}) \approx A(\underline{v}^n)$, $\underline{v}_0 = ?$] :

$$\text{Löse } A(\underline{v}^n) \underbrace{(\underline{v}^{n+1} - \underline{v}^n)}_{\underline{v}_\delta} = \underline{g} - A(\underline{v}^n) \cdot \underline{v}^n$$

$$\underline{v}^{n+1} := \underline{v}^n + \underline{v}_\delta$$

Diese Idee ändert in (7.4) $C(\underline{u}^{K+1})$ in $C(\underline{u}^K)$ (dies sind dann die diskreten Oseen-Gleichungen) und somit führt die Setzung

$$A(\underline{u}) := \frac{1}{\tau}M + \sigma(D + C(\underline{u})) \quad (7.5)$$

zur Fixpunktiteration bei der Auflösung von (7.4).

$$\begin{array}{l}
\left(\underline{u}^0\right) := \left(\underline{u}^K\right) \quad , \quad n := 0 \\
\text{do} \\
n := n + 1 \\
\text{Löse das lineare, nichtsymmetrische und indefinite GLS} \\
\left(\begin{array}{cc} A(\underline{u}^n) & B \\ B^T & 0 \end{array}\right) \cdot \left(\begin{array}{c} \underline{u}_\delta^n \\ \underline{p}_\delta^n \end{array}\right) = \left(\begin{array}{c} \widehat{f} - A(\underline{u}^n)\underline{u}^n - B\underline{p}^n \\ -B^T\underline{u}^n \end{array}\right) \quad (7.6) \\
\left(\begin{array}{c} \underline{u}^{n+1} \\ \underline{p}^{n+1} \end{array}\right) := \left(\begin{array}{c} \underline{u}^n \\ \underline{p}^n \end{array}\right) + \left(\begin{array}{c} \underline{u}_\delta^n \\ \underline{p}_\delta^n \end{array}\right) \\
\text{until } \left\| \left(\begin{array}{c} \underline{u}_\delta^n \\ \underline{p}_\delta^n \end{array}\right) \right\| < \varepsilon_{\text{fix}} \\
\left(\begin{array}{c} \underline{u}^{K+1} \\ \underline{p}^{K+1} \end{array}\right) := \left(\begin{array}{c} \underline{u}^{n+1} \\ \underline{p}^{n+1} \end{array}\right)
\end{array}$$

Alg. 7.1 Linear implizite Fixpunktiteration (seriell)

Mit

$$\begin{aligned}
A &:= A(\underline{u}^n) \\
\underline{r} &:= \widehat{f} - A(\underline{u}^n)\underline{u}^n - B\underline{p}^n \\
\underline{s} &:= -B^T\underline{u}^n
\end{aligned}$$

läßt sich das lineare Sattelpunktproblem (7.6) als

$$\left(\begin{array}{cc} A & B \\ B^T & 0 \end{array}\right) \cdot \left(\begin{array}{c} \underline{u}_\delta \\ \underline{p}_\delta \end{array}\right) = \left(\begin{array}{c} \underline{r} \\ \underline{s} \end{array}\right) \quad (7.7)$$

schreiben.

Eigenschaften von A , nach John [Joh 97] Seite 30 :

- Stokes (7.3) $\implies A$ ist symmetrisch, positiv definit.
- Navier Stokes (7.2) mit Stabilisierung durch scharfes Upwind oder Streamline-Diffusion-Upwind-Petrov-Galerkin (SUPG)
 $\implies A = A(\underline{u})$ ist regulär (d.h. $\exists A^{-1}$), falls $u(x)$ die Massenbilanzgleichung $\nabla \cdot u(x) = 0$ erfüllt.

In speziellen Fällen (keine stumpfen Dreiecke in Vernetzung) ist A sogar eine M -Matrix, was die Klasse der verfügbaren Löser vergrößert.

- Navier-Stokes instationär (7.1) $\implies A = A(\underline{u})$ ist regulär bei entsprechend kleiner Zeitschrittweite τ (Stabilität des Schemas).

Zur Lösung von (7.7) verwenden wir das Druckkorrekturverfahren, welches gewisse Ähnlichkeiten zur SIMPLE-Methode und zur Schur-Komplement Methode aufweist [Zul 97].

Idee : Faktorisier die Blockmatrix A und approximiere gegebenenfalls zu invertierende Teilmatrizen :

$$\begin{pmatrix} A & B \\ B^T & 0 \end{pmatrix} \approx \begin{pmatrix} \hat{A} & 0 \\ B^T & I \end{pmatrix} \begin{pmatrix} \hat{A}^{-1} & 0 \\ 0 & \hat{C}^{-1} \end{pmatrix} \begin{pmatrix} \hat{A} & B \\ 0 & I \end{pmatrix} ,$$

mit $\hat{A} \approx A$ und $\hat{C} \approx B^T A^{-1} B$ approximiert das negative Schurkomplement. Hier und im weiteren bezeichne $G \approx H$ die Spektraläquivalenz der Matrizen G und H mit gleichem Rang n , d.h., es existieren positive Konstanten \underline{c} , \bar{c} so daß gilt

$$\underline{c}(G \cdot \underline{v}, \underline{v}) \leq (H \cdot \underline{v}, \underline{v}) \leq \bar{c}(G \cdot \underline{v}, \underline{v}) \quad \forall \underline{v} \in \mathbb{R}^n .$$

Löse	$\hat{A}\underline{u}_\delta = \underline{r}$	(7.8)
Löse	$\hat{C}\underline{p}_\delta = B^T \underline{u}_\delta - \underline{s}$	
Löse	$\hat{A}\underline{u}_\delta = \underline{r} - B\underline{p}_\delta$	

Alg. 7.2 Druckkorrekturverfahren (seriell, eine Iteration)

Bemerkung 7.1. [Arrow-Hurwicz] Ein Weglassen der 3.Zeile in Algorithmus 7.2 ergibt den vorkonditionierten Arrow-Hurwicz Algorithmus.

Löse	$\hat{A}\underline{u}_\delta = \underline{r}$
Löse	$\hat{C}\underline{p}_\delta = B^T \underline{u}_\delta - \underline{s} = B^T \underline{u}^{n+1}$

Alg. 7.3 Vorkonditionierter Arrow-Hurwicz Algorithmus

Bemerkung 7.2. [Uzawa] Wählt man $\hat{C} = \frac{1}{\tau} I_{N_p}$, $\hat{A} = A$ und läßt die 3.Zeile in Algorithmus 7.2 weg, so erhält man den klassischen Uzawa-Algorithmus [BF 91]. Die hierbei entstehende Iterationsmatrix ist symmetrisch bzgl. eines speziell gewählten Skalarproduktes. Üblicherweise verwendet man den vorkonditionierten Uzawa-Algorithmus.

$\begin{aligned} \text{Löse} \quad A\underline{u}^{n+1} &= \underline{f} - B\underline{p}^n \\ \text{Löse} \quad \widehat{C}\underline{p}_\delta &= B^T\underline{u}_\delta - \underline{s} = B^T\underline{u}^{n+1} \end{aligned}$

Alg. 7.4 Vorkonditionierter Uzawa-Algorithmus

Bemerkung 7.3. Da die Druckkorrektur in (7.8) möglichst genau gelöst werden soll, wird häufig die Fixpunktiteration

$$\underline{p}_\delta^{m+1} - \underline{p}_\delta^m := \alpha \left(B^T \widetilde{A}^{-1} B \right)^{-1} \left(-B^T \widehat{A}^{-1} B \underline{p}_\delta^m + B^T \underline{u}_\delta - \underline{s} \right) \quad (7.9)$$

mit $\widetilde{A} \approx A$ benutzt.

Mögliche Wahl von \widetilde{A} : Einheitsmatrix I_{N_u} , Massenmatrix M (nichtkonforme Elemente), $\text{diag}(A)$.

Mögliche Wahl von $(B^T \widetilde{A}^{-1} B)$: Einheitsmatrix I_{N_p} , Massenmatrix M_{N_p} (Diagonalmatrix bei konstanten Druckansätzen).

Die Matrix $B^T \widetilde{A}^{-1} B$ ist im symmetrischen Fall positiv semidefinit (d.h. alle Eigenwerte größer oder *gleich* 0). Durch Festlegung einer Komponente von \underline{p}_δ kann das entsprechende Gleichungssystem in (7.9) mittels eines geeigneten Iterationsverfahrens (gmres, cg, mg) gelöst werden. Darin wird nur die Multiplikation der Matrix $B^T \widetilde{A}^{-1} B$ mit einem Vektor benötigt, nicht jedoch ihre Inverse!

Bemerkung 7.4. Üblicherweise wird auch \widehat{A}^{-1} über Iterationsverfahren realisiert.

Bemerkung 7.5. Die Fixpunktiteration (Algorithmus 7.1) entstand durch die Linearisierung von (7.4) mittels

$$C(\underline{u}^{K+1})\underline{u}^{K+1} \xrightarrow[\text{implizit}]{\text{linear}} C(\underline{u}^K)\underline{u}^{K+1} .$$

Wählt man stattdessen die Ersetzung

$$C(\underline{u}^{K+1})\underline{u}^{K+1} \xrightarrow[\text{explizit}]{\text{Konvektion}} C(\underline{u}^K)\underline{u}^K ,$$

erhält man sofort das Sattelpunktproblem (7.7) mit

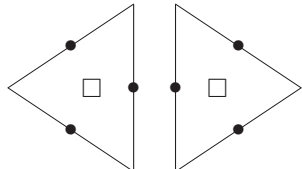
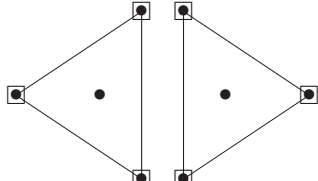
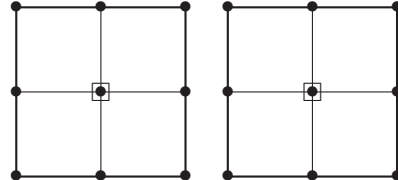
$$\begin{aligned} A &:= \frac{1}{\tau} M + \sigma D && \text{spd!} \\ \underline{r} &:= \sigma \underline{f}(t_{K+1}) + (1 - \sigma) \underline{f}(t_K) + \left[\frac{1}{\tau} M - (1 - \sigma) D - C(\underline{u}^K) \right] \underline{u}^K \\ \underline{s} &:= 0 , \end{aligned}$$

7.1.3 Komponenten der Parallelisierung

Die Datenaufteilung

Bezeichne \bullet die Knoten der Geschwindigkeitskomponenten und \square die Knoten des Druckes. Zur Diskretisierung können u.a. die in Tab. 7.1 dargestellten Elemente verwendet werden. Für die Parallelisierung (= Datenaufteilung) bietet

Tab. 7.1 Verwendete Elementtypen

(a)	nichtkonformes $[P_1, P_{0-}]$ Element	
(b)	in u, p konformes $[P_1+\text{Bubble}, P_1]$ MINI-Element	
(c)	in u konformes $[Q_1^h, Q_0^H]$ Element, Druckkomponente in Mittelpunkt des Makroelements	

sich eine nichtüberlappende Elementaufteilung entsprechend Abschnitt 5.1 an. Beim Elementtyp (c) sollten die Makroelemente verteilt werden.

Da die Matrizen in (7.4) elementweise berechnet werden, bietet sich hier eine verteilte Speicherung an, d.h. $M = \sum_{s=1}^p A_s^T M_s A_s$ und analog für D, C, B und genauso \underline{f} . Da diese Matrizen teilgebietsweise berechnet und gespeichert werden, fällt außer evtl. für C_s keine Kommunikation an. Die Koinzidenzmatrizen A_i werden im weitem weggelassen. Wird zur Berechnung der Konvektionsmatrix C_s ein Upwind-Schema 1. Ordnung verwandt, benötigt man keine Kommunikation in der Matrixgenerierung. Bei der Verwendung von Upwind-Schemata höherer Ordnung erfordert die Kenntnis der Geschwindigkeitskomponenten weiterer Nachbarelemente einen vorbereitenden Kommunikationsschritt.

Die parallele Fixpunktiteration

Da sämtliche Teilmatrizen in (7.4) vom verteilten Typ sind (siehe Abschnitt 5.1), bietet sich die Definition der Zustandgrößen $\underline{\mathbf{u}}$, $\underline{\mathbf{p}}$ als akkumulierte Vektoren und der integralen Größen $\underline{\mathbf{f}}$ ($\underline{\mathbf{r}}$, $\underline{\mathbf{s}}$) als verteilte Vektoren an.

$$\begin{array}{l}
 \left(\begin{array}{c} \underline{\mathbf{u}}^0 \\ \underline{\mathbf{p}}^0 \end{array} \right) := \left(\begin{array}{c} \underline{\mathbf{u}}^K \\ \underline{\mathbf{p}}^K \end{array} \right) \quad , \quad n := 0 \\
 \text{do} \\
 \quad n := n + 1 \\
 \quad \boxed{\text{Löse}} \text{ das lineare, nichtsymmetrische und indefinite GLS} \\
 \quad \quad \left(\begin{array}{cc} A(\underline{\mathbf{u}}^n) & B \\ B^T & 0 \end{array} \right) \cdot \left(\begin{array}{c} \underline{\mathbf{u}}_\delta^n \\ \underline{\mathbf{p}}_\delta^n \end{array} \right) = \left(\begin{array}{c} \widehat{\underline{\mathbf{f}}} - A(\underline{\mathbf{u}}^n)\underline{\mathbf{u}}^n - B\underline{\mathbf{p}}^n \\ -B^T\underline{\mathbf{u}}^n \end{array} \right) \\
 \quad \quad \left(\begin{array}{c} \underline{\mathbf{u}}^{n+1} \\ \underline{\mathbf{p}}^{n+1} \end{array} \right) := \left(\begin{array}{c} \underline{\mathbf{u}}^n \\ \underline{\mathbf{p}}^n \end{array} \right) + \left(\begin{array}{c} \underline{\mathbf{u}}_\delta^n \\ \underline{\mathbf{p}}_\delta^n \end{array} \right) \\
 \text{until} \quad \sum_{s=1}^p \left[\left(R_{\underline{\mathbf{u}},s}^{-1} \underline{\mathbf{u}}_{\delta,s}^n, \underline{\mathbf{u}}_{\delta,s}^n \right) + \left(R_{\underline{\mathbf{p}},s}^{-1} \underline{\mathbf{p}}_{\delta,s}^n, \underline{\mathbf{p}}_{\delta,s}^n \right) \right] < \varepsilon_{\text{fix}}^2 \\
 \left(\begin{array}{c} \underline{\mathbf{u}}^{K+1} \\ \underline{\mathbf{p}}^{K+1} \end{array} \right) := \left(\begin{array}{c} \underline{\mathbf{u}}^{n+1} \\ \underline{\mathbf{p}}^{n+1} \end{array} \right)
 \end{array}$$

Alg. 7.6 Linear implizite Fixpunktiteration (parallel)

Die neuen Größen in Alg. 7.6 sind

$$\begin{aligned}
 \widehat{\underline{\mathbf{f}}} &:= \sigma \mathbf{f}(t_{K+1}) + (1 - \sigma) \mathbf{f}(t_K) + \left[\frac{1}{\tau} \mathbf{M} - (1 - \sigma) (\mathbf{D} + \mathbf{C}(\underline{\mathbf{u}}^K)) \right] \underline{\mathbf{u}}^K \\
 A(\underline{\mathbf{u}}^n) &:= \frac{1}{\tau} \mathbf{M} + \sigma \left(\mathbf{D} + \boxed{\mathbf{C}(\underline{\mathbf{u}}^n)} \right) .
 \end{aligned}$$

- Kommunikation erfordernde Algorithmenteile werden mit $\boxed{\phantom{\text{code}}}$ gekennzeichnet, bzw. sind durch die Akkumulationssumme $\sum_{s=1}^p$ ersichtlich.
 - Im teilgebietsweisen inneren Produkt muß die notwendige, kommunikationslose Vektortypumwandlung (5.8) an den akkumulierten Vektoren $\underline{\mathbf{u}}_{\delta,s}^n$ und $\underline{\mathbf{p}}_{\delta,s}^n$ vorgenommen werden.
- Bei den Elementtypen (a) und (c) mit Verteilung der Makroelemente ist $R_{\underline{\mathbf{p}}} = I_{N_p}$.

Das parallele Druckkorrekturverfahren

Das Sattelpunktproblem (7.7) (linear, nichtsymmetrisch, indefinit)

$$\begin{pmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{B}^T & 0 \end{pmatrix} \cdot \begin{pmatrix} \underline{\mathbf{u}}_\delta \\ \underline{\mathbf{p}}_\delta \end{pmatrix} = \begin{pmatrix} \underline{\mathbf{r}} \\ \underline{\mathbf{s}} \end{pmatrix}$$

mit $\underline{\mathbf{r}} := \hat{\mathbf{f}} - \mathbf{A}(\underline{\mathbf{u}}^n)\underline{\mathbf{u}}^n - \mathbf{B}\underline{\mathbf{p}}^n$
 $\underline{\mathbf{s}} := -\mathbf{B}^T\underline{\mathbf{u}}^n$

wird mit dem parallelen Druckkorrekturverfahren gelöst.

Löse	$\hat{\mathbf{A}} \cdot \hat{\underline{\mathbf{u}}}_\delta = \underline{\mathbf{r}}$	mittels geeigneter IV aus Kap. 6
Löse	$\hat{\mathbf{C}} \cdot \hat{\underline{\mathbf{p}}}_\delta = \mathbf{B}^T \hat{\underline{\mathbf{u}}}_\delta - \underline{\mathbf{s}}$	(7.10)
Löse	$\hat{\mathbf{A}} \cdot \underline{\mathbf{u}}_\delta = \underline{\mathbf{r}} - \mathbf{B}\hat{\underline{\mathbf{p}}}_\delta$	mittels geeigneter IV aus Kap. 6

Alg. 7.7 Druckkorrekturverfahren (parallel)

Zur Auflösung von (7.10) :

- $\hat{\mathbf{C}} := \mathcal{J}_{N_p}$ (Uzawa) $\implies \hat{\underline{\mathbf{p}}}_\delta = \sum_{s=1}^p (\mathbf{B}_s^T \underline{\mathbf{u}}_\delta - \underline{\mathbf{s}}_s)$

Im Falle konstanter Druckansätze pro Element (Diskretisierungen a) und b)) ist keine Typumwandlung/Akkumulation des Druckvektors nötig.

- $\hat{\mathbf{A}} \approx \mathfrak{A} = \sum_{s=1}^p \mathbf{A}_s(\underline{\mathbf{u}}^n)$, $\hat{\mathbf{C}} := (\mathbf{B}^T \tilde{\mathfrak{A}}^{-1} \mathbf{B})$, $\tilde{\mathfrak{A}} \approx \mathfrak{A}$
- Löst man die Gleichung (7.10) mittels der Fixpunktiteration (7.9)

$$(\mathbf{B}^T \tilde{\mathfrak{A}}^{-1} \mathbf{B}) (\underline{\mathbf{p}}_\delta^{m+1} - \underline{\mathbf{p}}_\delta^m) = \alpha \left((\mathbf{B}^T \hat{\mathbf{A}}^{-1} \mathbf{B}) \underline{\mathbf{p}}_\delta^m + \mathbf{B}^T \underline{\mathbf{u}}_\delta - \underline{\mathbf{s}} \right), \quad (7.11)$$

so wird bei der Berechnung der rechten Seite in der Auflösung von

$$\hat{\mathbf{A}} \cdot \underline{\mathbf{w}} = \mathbf{B}\underline{\mathbf{p}}_\delta^m$$

wiederum ein geeignetes parallelisiertes IV aus Kap. 6 benutzt.

- Zur Auflösung von (7.11) setzen wir ein Iterationsverfahren ein, welches in jedem Schritt der Matrix-mal-Vektor Operation

$$\underline{\mathbf{v}} := \mathbf{B}^T \underbrace{\tilde{\mathfrak{A}}^{-1} \mathbf{B} \cdot \underline{\mathbf{w}}}_{\hat{\underline{\mathbf{v}}}}$$

die schnelle parallele (und direkte) Auflösung von

$$\tilde{\mathfrak{A}} \cdot \hat{\mathbf{v}} = \mathbf{B} \cdot \mathbf{w}$$

erfordert.

Für $\tilde{\mathfrak{A}} := \mathfrak{I}_{N_u}$ bzw. $\tilde{\mathfrak{A}} := \sum_{s=1}^p \text{diag} \mathbf{A}_s$ ist diese Forderung mit einer Vektorakkumulation per Iteration leicht realisierbar.

Bei einer Diskretisierung mit dem nichtkonformen Element (a) sichert die Diagonalität von M ([Joh 97], pp.27) bei der Wahl $\tilde{\mathfrak{A}} := \mathfrak{M} = \sum_{s=1}^p \mathbf{M}_s$ die schnelle parallele Auflösbarkeit.

Bei Verwendung des konformen Elements (c) ist diese günstige Diagonaleigenschaft nicht mehr gegeben, wodurch eine Verwendung von $\tilde{\mathfrak{A}} = \mathfrak{M}$ parallel ineffizient wird. Ersetzt man jedoch \mathfrak{M} durch eine Matrix $\tilde{\mathfrak{M}}$ wie in Abschnitt 6.5.6 beschrieben, und wählt $\tilde{\mathfrak{A}}$ gleich der entsprechenden unvollständigen Zerlegung (Kapitel 6.5), ist mittels eines Iterationsverfahrens eine effiziente parallele Auflösung möglich.

7.2 Die Eulergleichungen

Die Darstellung der ersten beiden Abschnitte dieses Kapitels lehnt sich stark an Kröner [Krö 97], pp. 372-394, an

7.2.1 Die Differentialgleichungen

Die Eulergleichungen der Gasdynamik haben über dem Integrationsgebiet $\Omega \times (0, T] \subset \mathbb{R}^2$ das Aussehen

$$\partial_t \begin{pmatrix} \varrho \\ \varrho u \\ \varrho v \\ e \end{pmatrix} + \partial_x \begin{pmatrix} \varrho u \\ \varrho u^2 + p \\ \varrho uv \\ u \cdot (e + p) \end{pmatrix} + \partial_y \begin{pmatrix} \varrho v \\ \varrho uv \\ \varrho v^2 + p \\ v \cdot (e + p) \end{pmatrix} = 0, \quad (7.12)$$

zusätzlich $p = (\gamma - 1) \left[e - \frac{\varrho}{2} \cdot (u^2 + v^2) \right]$
+ AB + RB .

Hierin sind ϱ - Dichte
 u, v - Geschwindigkeitskomponenten
 p - Druck
 e - Energie
 $\gamma = c_p/c_v$.

Bei konstanter Dichte und dem Weglassen der 4. Gleichung ergeben sich aus (7.12) die inkompressiblen Navier-Stokes-Gleichungen (7.1).

Allgemein schreibt man obige Gleichungen in der Form (Gleichung für p eingesetzt) mit $\vec{u} = (\varrho, \varrho u, \varrho v, e)^T$:

$$\begin{array}{l} \partial_t \vec{u} + \partial_x f_1(\vec{u}) + \partial_y f_2(\vec{u}) = 0 \quad \text{in } \Omega \times (0, T] \subset \mathbb{R}^2 \\ \quad \quad \quad + \text{AB} \quad \quad + \text{RB} . \end{array} \quad (7.13)$$

7.2.2 Die serielle Auflösung

• Eine Zeitdiskretisierung über ein explizites 2-schichtiges Differenzenschema, mit $\vec{u}^K := \vec{u}(t_K)$ und $\tau = t_{K+1} - t_K$, führt auf :

$$\begin{array}{l} \vec{u}^0 \quad \text{aus AB gegeben,} \\ \vec{u}^{K+1} := \vec{u}^K - \tau \cdot \text{div} f(\vec{u}^K) \quad \text{in } \Omega \quad \forall K = 0, 1, \dots \\ \quad \quad \quad + \text{RB} \end{array} \quad (7.14)$$

• Eine Ortsdiskretisierung mittels Volumenmethode (Freiheitsgrade im Schwerpunkt der Volumina $T_j, j = \overline{1, N}$), mit $u_j^K := \vec{u}^K(x_j), x_j \in T_j$, führt auf das Gleichungssystem :

$$\begin{array}{l} u_j^0 = \frac{1}{|T_j|} \int_{T_j} \vec{u}^0(x) dx \quad j = \overline{1, N} \\ u_j^{K+1} := u_j^K - \frac{1}{|T_j|} \int_{T_j} \text{div} f(\vec{u}^K(x)) dx \quad j = \overline{1, N}, \forall K = 0, 1, \dots \end{array} \quad (7.15)$$

• Der Gaußsche Integralsatz überführt das Volumenintegral aus (7.15) in ein Oberflächenintegral, welches sich als Summe der $n_e(j)$ Integrale auf den Begrenzungsflächen (-flächen) S_{jl} des Volumenelements T_j darstellen läßt. n_{jl} ist der nach außen gerichtete Normalenvektor von S_{jl} .

$$\int_{T_j} \text{div} f(\vec{u}^K) dx = \int_{\partial T_j} f(\vec{u}^K(s)) \vec{n}(s) ds = \sum_{l=1}^{n_e(j)} \int_{S_{jl}} f(\vec{u}^K(s)) \vec{n}_{jl} ds$$

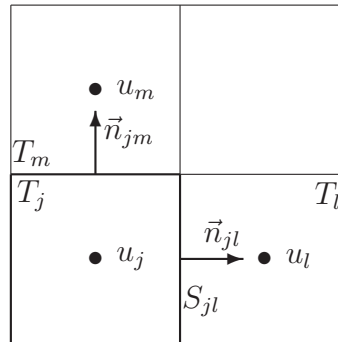


Fig. 7.1
Bezeichnungen am Volumenelement T_j .

- Der kritische Punkt für eine numerische Lösung und auch deren Parallelisierung ist, neben der Stabilität des expliziten Schemas, die numerische Approximation des Flusses über den Rand $\int_{S_{jl}}$ durch einen numerischen Fluß

$$g_{jl}(\underline{u}^K) \approx \int_{S_{jl}} f(\underline{u}^K(s)) \vec{n}_{jl} ds . \quad (7.16)$$

Das numerische Schema zum Lösen von (7.13) ist dann

$$\begin{aligned} u_j^0 &= \frac{1}{|T_j|} \int_{T_j} \vec{u}^0(x) dx & j = \overline{1, N} \\ u_j^{K+1} &:= u_j^K - \frac{\tau}{|T_j|} \sum_{l=1}^{n_e(j)} g_{jl}(\underline{u}^K) & j = \overline{1, N} \quad \forall K = 0, 1, \dots \end{aligned} \quad (7.17)$$

Die Berechnung von g_{jl} wird im folgenden kurz skizziert.

Berechnung von g_{jl} mittels Flux-Vector-Splitting

Die Vorgehensweise entspricht der in Kröner [Krö 97], p.374f.

Es gelte, wie z.B. in (7.12),

$$f_1(\vec{u}) = f'_1(\vec{u}) \cdot \vec{u} \quad \text{und} \quad f_2(\vec{u}) = f'_2(\vec{u}) \cdot \vec{u} .$$

Die Definition

$$C_{jl}(\vec{u}) := \vec{n}_{jl} f'(\vec{u})$$

führt über ($Q(\vec{u})$ - nichtsingulär) $Q^{-1} C_{jl} Q = D = \text{diag} \{ \lambda_1, \dots, \lambda_m \}$ und $\lambda_i^+ := \max \{ \lambda_i, 0 \}$, $\lambda_i^- := \min \{ \lambda_i, 0 \}$, $i = \overline{1, m}$ zu

$$C_{jl}^+(\vec{u}) = Q D^+ Q^{-1} \quad \text{und} \quad C_{jl}^-(\vec{u}) = Q D^- Q^{-1} .$$

Wegen $\vec{n}_{jl} = -\vec{n}_{lj}$ gilt $C_{jl}(\vec{u}) = -C_{lj}(\vec{u})$ und insbesondere $C_{jl}^-(\vec{u}) = -C_{lj}^+(\vec{u})$. Es wird nun der numerische Fluß $g_{jl} := \int_{S_{jl}} \tilde{g}_{jl}(s) ds$ mittels

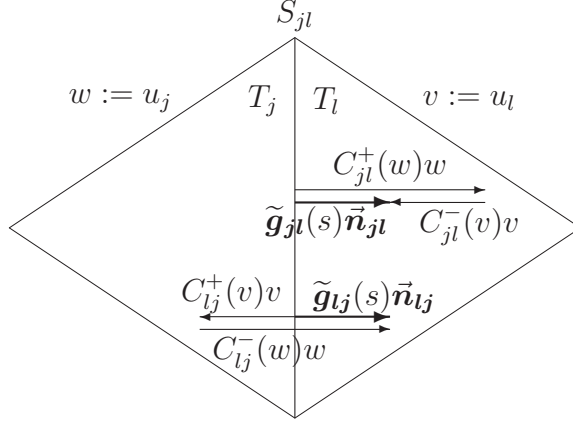


Fig. 7.2 Fluß zwischen den Elementen T_j und T_l nach Steger und Warming.

$$g_{jl}(w, v) := |S_{jl}| (C_1(w, v)w + C_2(w, v)v) \quad (7.18)$$

berechnet.

Für die Berechnung der C_1, C_2 gibt es verschiedene Ansätze :

1. Nach Steger und Warming [SW 81]

$$\begin{aligned} C_1(w, v) &:= C_{jl}^+(w) \\ C_2(w, v) &:= C_{jl}^-(v) \left[\stackrel{!}{=} -C_{lj}^+(v) \right] \end{aligned} \quad (7.19)$$

2. Nach Vijayasundaram [Vij 86]

$$\begin{aligned} C_1(w, v) &:= C_{jl}^+\left(\frac{w+v}{2}\right) \\ C_2(w, v) &:= C_{jl}^-\left(\frac{w+v}{2}\right) \left[\stackrel{!}{=} -C_{lj}^+\left(\frac{w+v}{2}\right) \right] \end{aligned} \quad (7.20)$$

3. Nach van Leer [Lee 92]

$$\begin{aligned} C_1(w, v) &:= C_{jl}(w) + \left| C_{jl}^+\left(\frac{w+v}{2}\right) \right| \\ C_2(w, v) &:= C_{jl}(v) - \left| C_{jl}^-\left(\frac{w+v}{2}\right) \right| \\ &\left[\stackrel{!}{=} -C_{lj}(v) - \left| C_{lj}^+\left(\frac{w+v}{2}\right) \right| \right] \end{aligned} \quad (7.21)$$

Bemerkung 7.8. Die Matrizen $C_{jl}^+(w)$, $C_{jl}^-(w)$ und $C_{jl}(w)$ sind lokal in T_j berechenbar. Analog sind $C_{ij}^+(v)$, $C_{ij}^-(v)$ und $C_{ij}(v)$ lokal in T_i berechenbar. Für die Anwendung des Argumentes $\frac{w+v}{2}$ muß als Vorbereitungsschritt der Mittelwert gebildet werden.

Die in (7.19)-(7.21) mit [] gekennzeichneten Formeln für C_2 berechnen C_2 lokal in T_l . Allgemein gilt dann $\underbrace{C_2(w, v)}_{T_j} = - \underbrace{C_1(v, w)}_{T_l}$.

```

for j =  $\overline{1, N}$  do
  for l ∈ {Nachbarn} do
     $\sigma_{jl} := C_1(u_j, u_l) \cdot u_j$ 
  od
od
for j =  $\overline{1, N}$  do
  g := 0
  for l ∈ {Nachbarn} do
    g := g + |Sjl| (σjl - σlj)
  od
   $u_j^{K+1} := u_j^K - \frac{\tau}{|T_j|} \cdot g$ 
od

```

Alg. 7.8 Effiziente serielle Berechnung des (K+1)-ten Zeitschritts

7.2.3 Die Parallelisierung mittels verteilter Boxen

Die Verteilung von kompletten Boxen auf die Prozessoren entspricht einer Aufteilung mit nichtüberlappenden Knoten. Der einzige kritische Punkt bei der Parallelisierung von (7.17) besteht dann in der effizienten Berechnung der Summe $C_1(w, v)w + C_2(w, v)v$ in (7.18).

Die Unbekannten u_j kann man sich als in den Boxschwerpunkten gespeichert denken. Die lokalen Flußanteile σ_{jl} in Algorithmus 7.8 liegen an den Kanten (Flächen) vor, sind jedoch inkonsistent (i.a. $\sigma_{jl} \neq \sigma_{lj}$). Für die parallele Implementierung empfiehlt sich bzgl. der Speicherung von u und σ eine Aufteilung in überlappende Elemente mit einem Element Überlappungsbreite. Das erweiterte Teilgebiet von Ω_i sei mit $\tilde{\Omega}_i$ bezeichnet, der erweiterte Lösungsvektor sei $\tilde{u}_i = \left(\underbrace{\{u_j\}_{T_j \in \Omega_i}}_{u_i}, \underbrace{\{u_l\}_{T_l \in \tilde{\Omega}_i \setminus \Omega_i}}_{\text{externe Komp.}} \right)$, analog $\tilde{\sigma}$.

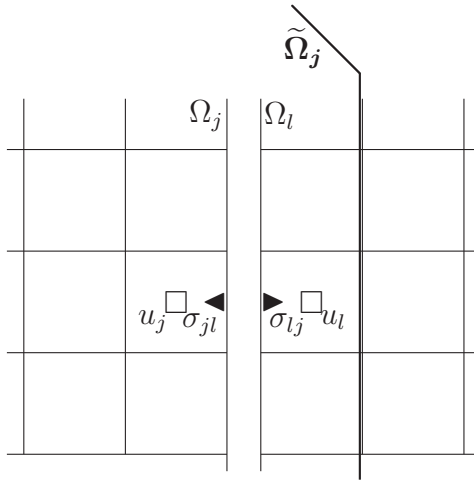


Fig. 7.3
Verteilte Boxen

- 1.) Tausche die Randkomponenten von \underline{u} mit den Nachbarn über die Kanten aus [EXCHANGED].
Aktualisiere die externen Komponenten von $\tilde{\underline{u}}_i$, d.h. $\{\tilde{u}_l \notin \underline{u}_i\}$.
- 2.) for $j = \overline{1, N}$ do
 for $l \in \{\text{neighbours}\}$ do
 $\sigma_{jl} := C_1(u_j, \tilde{u}_l) \cdot u_j$
 od
 od
- 3.) Tausche die Randkomponenten von σ , d.h. $\{\sigma_{jl} | \tilde{u}_l \notin \underline{u}_i\}$ mit den Nachbarn über die Kanten aus [EXCHANGED].
Aktualisiere die externen Komponenten $\{\tilde{\sigma}_{lj} | \tilde{u}_l \notin \underline{u}_i \wedge \tilde{u}_j \in \underline{u}_i\}$.
- 4.) for $j = \overline{1, N}$ do
 $g := 0$
 for $l \in \{\text{neighbours}\}$ do
 $g := g + |S_{jl}| (\sigma_{jl} - \tilde{\sigma}_{lj})$
 od
 $u_j^{K+1} := u_j^K - \frac{\tau}{|T_j|} \cdot g$
 od

Alg. 7.9 (K+1)-ter Zeitschritt in Ω_i bei verteilten Boxen

Bemerkung 7.9.

1. Algorithmus 7.9 ist für alle 3 Formeln (7.19)-(7.21) anwendbar.
2. Bei Benutzung von (7.19) kann Schritt 1.) weggelassen werden, da \tilde{u}_l nicht benötigt wird.
3. Bei Verwendung von (7.20) und (7.21) kann man den Datenaustausch in Schritt 3.) einsparen, indem in $\tilde{\Omega}_j$ die Größe $\tilde{\sigma}_{lj} := -C_2(u_j, \tilde{u}_l)$ redundant (zu $C_1(u_l, \tilde{u}_j)$ in $\tilde{\Omega}_l$) berechnet wird.

Nachteile: • Höherer arithmetischer Aufwand (fast verdoppelt).

- Falls man nur in den Randelementen redundant rechnet, muß man im Programm Fallunterscheidungen einbauen.

7.2.4 Die Parallelisierung mittels aufgeteilter Boxen

Repräsentiert man jede Box durch ihren Mittelpunkt und verbindet diese Mittelpunkte ergibt sich eine Art FEM-Netz, welches analog zu den nichtüberlappenden Elementen aus Abschnitt 5.1 verteilt wird. Somit sind sämtliche Ergebnisse bezüglich der dort definierten Vektor- und Matrizentypen anwendbar.

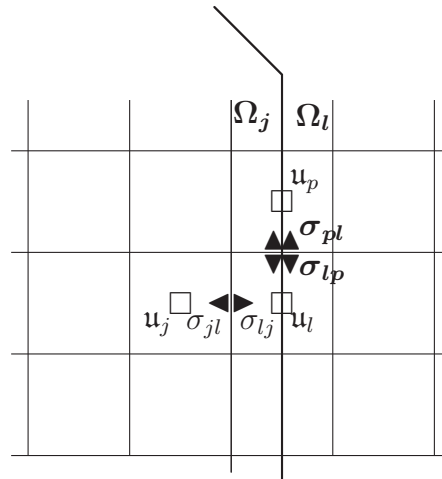


Fig. 7.4
Aufgeteilte Boxen

Man wählt die Funktionalgröße \mathbf{g} als verteilt gespeichert und die Unbekannten \mathbf{u} als akkumulierten Vektor. Jedoch liegen auch die Größen σ_{pl} aus denen sich \mathbf{g} ergibt, in akkumulierter Form vor und dürfen nicht mehrfach gezählt werden. Da der Flußanteil \mathbf{g}_{jl} eine integrale Größe (Funktionalgröße) ist, darf man nur den Anteil der Kante S_{jl} bei der lokalen Berechnung in Ω_j berücksichtigen welcher sich in diesem Teilgebiet befindet. Dies wird durch die verteilte Größe \mathbf{s}_{jl} ausgedrückt.

```

0.) Vorbereitungsschritt a: Berechnung der Boxvolumina.
   for  $j = \overline{1, N}$  do    $\mathbf{t}_j := |T_j \cap \Omega_i|$    od
    $\underline{\mathbf{t}} := \sum_{s=1}^P \mathbf{t}_s$ 

   Vorbereitungsschritt b: Berechnung der Teilkantenlängen.
   for  $j = \overline{1, N}$  do
     for  $l \in \{\text{neighbours}\}$  do    $\mathbf{s}_{jl} := |S_{jl} \cap \Omega_i|$ 
   od od
1.) for  $j = \overline{1, N}$  do
    $\mathbf{g}_j := 0$ 
   for  $l \in \{\text{neighbours}\}$  do
      $\mathbf{g}_j := \mathbf{g}_j + \mathbf{s}_{jl} (C_1(\mathbf{u}_j, \mathbf{u}_l) \cdot \mathbf{u}_j - C_2(\mathbf{u}_j, \mathbf{u}_l) \cdot \mathbf{u}_l)$ 
   od
    $\mathbf{g}_j := \mathbf{g}_j / \mathbf{t}_j$ 
od
2.)  $\underline{\mathbf{u}}^{K+1} := \underline{\mathbf{u}}^K - \tau \cdot \sum_{s=1}^P \mathbf{g}_s$ 

```

Alg. 7.10 (K+1)-ter Zeitschritt in Ω_i bei aufgeteilten Boxen**Bemerkung 7.10.**

1. Algorithmus 7.10 benötigt neben dem einmaligen Berechnen der Boxvolumina pro Zeitschritt nur noch eine Kommunikation.
2. Die zur Berechnung der C_1 und C_2 benötigten Daten sind alle lokal verfügbar ohne daß zusätzliche Kommunikation notwendig ist.
3. Unter Benutzung der Relation $C_2(w, v) = -C_1(v, w)$ läßt sich analog zu Algorithmus 7.10 Arithmetik sparen, **ohne** daß Kommunikation nötig ist !

7.3 Die kompressiblen Navier-Stokes-Gleichungen**7.3.1 Die Differentialgleichungen**

Die Darstellung der kompressiblen Navier-Stokes-Gleichungen und ihr serieller Auflösungsprozeß folgt im wesentlichen dem Buch von Kröner [Krö 97].

Das Verhalten einer kompressiblen Flüssigkeit in einem beschränkten Gebiet der Ebene wird analog zu (7.12) durch die **kompressiblen Navier-Stokes-Gleichungen** im Gebiet $\Omega \times (0, T] \subset \mathbb{R}^2$ beschrieben:

$$\begin{aligned}
& \partial_t \begin{pmatrix} \varrho \\ \varrho u \\ \varrho v \\ e \end{pmatrix} + \partial_x \begin{pmatrix} \varrho u \\ \varrho u^2 + p \\ \varrho uv \\ u \cdot (e + p) \end{pmatrix} + \partial_y \begin{pmatrix} \varrho v \\ \varrho uv \\ \varrho v^2 + p \\ v \cdot (e + p) \end{pmatrix} \\
& + \partial_x \begin{pmatrix} 0 \\ \tau_{11} \\ \tau_{12} \\ \tau_{11}u + \tau_{12}v + k\partial_x\Theta \end{pmatrix} + \partial_y \begin{pmatrix} 0 \\ \tau_{21} \\ \tau_{22} \\ \tau_{21}u + \tau_{22}v + k\partial_y\Theta \end{pmatrix} = 0
\end{aligned}$$

Zusätzlich mit $v_1 := u, v_2 := v$:

$$\begin{aligned}
p &= (\gamma - 1) \left[e - \frac{\varrho}{2} \cdot (u^2 + v^2) \right] \\
e &= \varrho \left[c_v \Theta + \frac{1}{2} (u^2 + v^2) \right] \\
\tau_{ij} &= \lambda \operatorname{div} \begin{pmatrix} u \\ v \end{pmatrix} \delta_{ij} + \mu (\partial_j v_i + \partial_i v_j) \quad i, j = 1, 2 \\
&+ \text{AB} \quad + \text{RB}
\end{aligned}$$

(7.22)

Zusätzlich zu den bereits aus Abschnitt 7.2.1 bekannten Größen sind :

- τ_{ij} - Komponenten des viskosen Anteil des Spannungstensors
- λ, μ - Viskositätskoeffizienten
- Θ - absolute Temperatur
- c_v - spezifische Wärme bei konstanten Volumen
- k - Wärmeleitfähigkeit
- $\delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$ - Diracsche Deltafunktion.

Außerdem wird $c_v, k, \mu > 0$ und $\lambda = -\frac{3}{2}\mu$ angenommen.

Aus den Gleichungen (7.22) ergeben sich die kompressiblen Navier-Stokes (7.1) und die Euler-Gleichungen (7.12) als Spezialfälle.

7.3.2 Die serielle Auflösung

Zur Auflösung beschreibt man (7.22) als Summe von Gleichungen der Konvektion und solchen Gleichungen, welche die viskosen Eigenschaften beinhalten, und betrachtet diese beiden Systeme zunächst getrennt. Mit den Abkürzungen

$$\theta_1 = \tau_{11}u + \tau_{12}v + k\partial_x\Theta \quad \text{und} \quad \theta_2 = \tau_{21}u + \tau_{22}v + k\partial_y\Theta$$

ergibt sich somit

$$\frac{1}{2} \partial_t \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ e \end{pmatrix} + \partial_x \begin{pmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ u \cdot (e + p) \end{pmatrix} + \partial_y \begin{pmatrix} \rho v \\ \rho uv \\ \rho v^2 + p \\ v \cdot (e + p) \end{pmatrix} = 0 \quad (7.23)$$

$$\frac{1}{2} \partial_t \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ e \end{pmatrix} + \partial_x \begin{pmatrix} 0 \\ \tau_{11} \\ \tau_{12} \\ \theta_1 \end{pmatrix} + \partial_y \begin{pmatrix} 0 \\ \tau_{21} \\ \tau_{22} \\ \theta_2 \end{pmatrix} = 0 . \quad (7.24)$$

Die Gleichungen (7.23) sind mit den Euler-Gleichungen aus Abschnitt 7.2 (bis auf den Faktor $\frac{1}{2}$) identisch und wurden dort bereits behandelt. Die Diskretisierung erfolgte örtlich über die FVM und zeitlich explizit. Der Auflösungsschritt in jedem Zeitschritt wurde in Algorithmus 7.8 dargestellt und resultiert in einer im j -ten Volumenelement konstanten diskreten Lösung $U_j^{K+1/2}$ im $K + 1$ -ten Zeitschritt.

Die Gleichungen (7.24) beschreiben zeitliche Spannungszustände des Fluids und beinhalten 2. Ableitungen der gesuchten Größen. Aus diesem Grunde wird die örtliche Diskretisierung dieser Gleichungen über FEM realisiert. Bei Verwendung linearer Ansatzfunktionen ergibt sich damit die stückweise lineare Lösungsfunktion u_h^{K+1} im $K + 1$ -ten Zeitschritt.

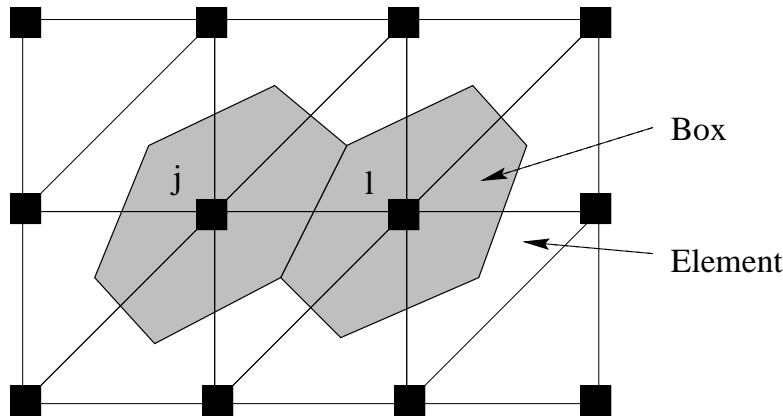


Fig. 7.5 Duale Vernetzung

Zur Koppelung der diskretisierten Gleichungen (7.23) und (7.24) ist eine duale Vernetzung des Gebietes mit finiten Volumina und finiten Elementen (\mathcal{T}_h) notwendig. Eine mögliche Vernetzung [FFLMW 97, FFLM 97, FFLMW 97] ist in Fig. 7.5 gegeben.

Mit den Testfunktionen (Dreiecksvernetzung) $\varphi \in \mathbb{V}_h : \Omega \mapsto \mathbb{R}^4$ und den Definitionen

$$(u_h, \varphi)_h := \frac{1}{3} \sum_{T \in \mathcal{T}_h} |T| \sum_{i=1}^3 u_h(P_T^i) \cdot \varphi(P_T^i) \approx \int_{\Omega} u_h \cdot \varphi \, dx$$

$$a_h(u_h, \varphi)_h := \text{Variationsformulierung Elastizitätsoperator}$$

$$b_h(u_h, \varphi)_h := \sum_j \varphi(P_j) \sum_{l=1}^k g_{jl}(u_h(P_l), u_h(P_j)) \, ,$$

mit g aus (7.18) ergeben sich für die diskrete Auflösung des gekoppelten Systems (7.23) und (7.24) in jedem Zeitschritt die folgenden 3 Ansätze :

1. Inviscid (7.23) - viscous (7.24) operator splitting

$U_j^{K+1/2} := U_j^K - \frac{\tau}{ T_j } \sum_{l=1}^{n_e(j)} g_{jl}(U^K)$	
Setze $u_h^{K+1/2}(P_i) = U_i^{K+1/2}$	$\forall P_i$
Löse $(u_h^{K+1}, \varphi)_h = (u_h^{K+1/2}, \varphi)_h - \tau a_h(u_h^{K+1/2}, \varphi)_h$	$\forall \varphi \in \mathbb{V}_h$
Setze $U_i^{K+1} = u_h^{K+1}(P_i)$	$\forall P_i$

(7.25)

Drückt man die 4 Komponenten von u_h und φ durch die FE-Basisfunktionen ψ_i aus, d.h.,

$$u_h(x) = (p(x), u(x), v(x), e(x))^T = \sum_{i=1}^4 (p_i, u_i, v_i, e_i)^T \cdot \psi_i(x)$$

$$\varphi(x) = (\varphi_1(x), \varphi_2(x), \varphi_3(x), \varphi_4(x))$$

$$= \sum_{i=1}^4 (\varphi_{1,i}, \varphi_{2,i}, \varphi_{3,i}, \varphi_{4,i})^T \cdot \psi_i(x) \, ,$$

so stellt sich die 3.Zeile in (7.25) als Gleichungssystem

$$M \cdot \underline{u}^{K+1} = M \cdot \underline{u}^{K+1/2} - \tau A \cdot \underline{u}^{K+1/2} \quad (7.26)$$

dar, mit A als Elastizitätsmatrix und M als der Massenmatrix. Bei entsprechender Wahl der Basisfunktionen bzw. nach dem Lumping ist M eine Diagonalmatrix und somit leicht zu invertieren.

Üblicherweise gilt $\psi_i(P_j) = \delta_{ij}$, so daß die Interpolationsforderungen der 2. und 4.Zeile automatisch erfüllt sind.

2. Explizites Schema

Unter Nutzung der Bilinearform b_h wird ein Zeitschritt in (7.23) als

$$(u_h^{K+1/2}, \varphi)_h = (u_h^K, \varphi)_h - \tau \cdot b_h(u_h^K, \varphi)_h$$

ausgedrückt und somit läßt sich (7.22) in einer diskreten Gleichung schreiben :

$$(u_h^{K+1}, \varphi)_h = (u_h^K, \varphi)_h - \tau \cdot [b_h(u_h^K, \varphi)_h + a_h(u_h^K, \varphi)_h] \quad \forall \varphi \in \mathbb{V}_h .$$

Mit B als der Matrix der Bilinearform b_h ist somit das GLS

$$M \cdot \underline{u}^{K+1} = M \cdot \underline{u}^K - \tau(B + A) \cdot \underline{u}^K \quad (7.27)$$

zu lösen.

3. Semi-implizites Schema

Im Unterschied zum expliziten Schema wird hier in der Bilinearform a_h die aktuelle Lösung u_h^{K+1} eingesetzt.

$$(u_h^{K+1}, \varphi)_h = (u_h^K, \varphi)_h - \tau \cdot [b_h(u_h^K, \varphi)_h + a_h(u_h^{K+1}, \varphi)_h] \quad \forall \varphi \in \mathbb{V}_h .$$

Das resultierende GLS

$$(M + \tau A) \cdot \underline{u}^{K+1} = M \cdot \underline{u}^K - \tau B \cdot \underline{u}^K \quad (7.28)$$

ist relativ aufwendig aufzulösen, da wegen der Besetztheitsstruktur von A die Matrix $M + \tau A$ keinesfalls eine Diagonalmatrix ist. Der Nutzen des semi-impliziten Schemas besteht in der größer wählbaren Zeitschrittweite τ . Hat man *schnelle* iterative Löser zur Verfügung (Eigenschaften von $M + \tau A$ beachten !) bzw. zeitlich konstante Matrizen A und M (einmalige Faktorisierung), dann resultiert dieses Schema in einer kürzeren Rechenzeit im Vergleich zum rein impliziten Schema.

7.3.3 Die Parallelisierung mittels aufgeteilter Boxen und verteilter Elemente

Die Aufteilung des Gebietes in Teilgebiete erfolgt entlang der Grenzen der finiten Elemente. Entsprechend den in Abschnitten 7.2.4 und 4.1.3 für Boxen und Elemente untersuchten Datenaufteilungen ist zur parallelen Realisierung von (7.25) - (7.28) die Wahl von \underline{u} als akkumulierten Vektor und von \underline{g} , M , A und B als verteilten Vektoren und Matrizen naheliegend.

0.) Vorbereitung: Berechnung Boxvolumina/Teilkantenlängen.

for $j = \overline{1, N}$ do $\mathbf{t}_j := |T_j \cap \Omega_i|$ od ; $\underline{t} := \sum_{s=1}^P \mathbf{t}_s$
 for $j = \overline{1, N}$ do $s_{jl} := |S_{jl} \cap \Omega_i| \quad \forall l \in \{\text{Nachbarn}\}$ od

1.) for $j = \overline{1, N}$ do $\mathbf{g}_j := 0$
 for $l \in \{\text{Nachbarn}\}$ do
 $\mathbf{g}_j := \mathbf{g}_j + s_{jl} (C_1(\mathbf{u}_j, \mathbf{u}_l) \cdot \mathbf{u}_j - C_2(\mathbf{u}_j, \mathbf{u}_l) \cdot \mathbf{u}_l)$
 od
 od

2.) Generiere/modifiziere M , A und B falls nötig.

3.) Je nach Zeitschema führe aus:

a) **Inviscid-viscous splitting**

$$\underline{u}^{K+1/2} := \underline{u}^K - \tau \cdot \sum_{s=1}^P \underbrace{\mathbf{g}_s / \mathbf{t}_s}_{\text{komponentenweise}}$$

$$\text{Akkumuliere } \mathfrak{M} := \sum_{s=1}^P M_s$$

$$\text{Löse } \mathfrak{M} \cdot \underline{u}^{K+1} = \sum_{s=1}^P (M_s - \tau A_s) \cdot \underline{u}^K$$

b) **Explizites Schema**

$$\text{Akkumuliere } \mathfrak{M} := \sum_{s=1}^P M_s$$

$$\text{Löse } \mathfrak{M} \cdot \underline{u}^{K+1} = \sum_{s=1}^P (M_s - \tau [B_s + A_s]) \cdot \underline{u}^K$$

c) **Semi-implizites Schema**

$$\text{Akkumuliere } \mathfrak{K} := \sum_{s=1}^P K_s = \sum_{s=1}^P (M_s + \tau A_s)$$

$$\text{Löse } \mathfrak{K} \cdot \underline{u}^{K+1} = \sum_{s=1}^P (M_s - \tau B_s) \cdot \underline{u}^K$$

Alg. 7.11 Zeitschritt in Ω_i für kompressible Navier-Stokes

Bemerkung 7.11.

- Da, außer bei zeitabhängigen Netzen oder Basisfunktionen, die Massenmatrix M zeitunabhängig ist, kann i.a. die Akkumulation zu \mathfrak{M} in den Vorbereitungsschritt verlegt werden.
- Falls auch die Elastizitätsmatrix A zeitunabhängig ist, kann auch deren Akkumulation zu \mathfrak{K} in den Vorbereitungsschritt verlegt werden.
- Für Massematrizen \mathfrak{M} mit Diagonalstruktur besteht das Lösen der entsprechende GIS nur in einer Multiplikation mit der akkumulierten Inversen dieser Diagonalmatrix.
- Falls \mathfrak{M} keine Diagonalstruktur besitzt, wird üblicherweise durch Lumping eine Diagonalmatrix $\tilde{\mathfrak{M}}$ erzeugt, mit welcher das GIS gelöst wird.
- Verwendet man Iterationsverfahren zur Auflösung des GIS in 3c) von Alg. 7.11, so wird K nicht akkumuliert und das GIS

$$K \cdot \underline{u}^{K+1} = \underline{f}$$

kann mit den Methoden aus Abschnitt 6 gelöst werden. Hierbei tritt in jedem Fall Kommunikation auf.

- Falls M keine Diagonalstruktur aufweist, so ist $K = M + \tau A$ keine M-Matrix mehr, so daß einige Iterationsverfahren gar nicht oder nur modifiziert anwendbar sind.

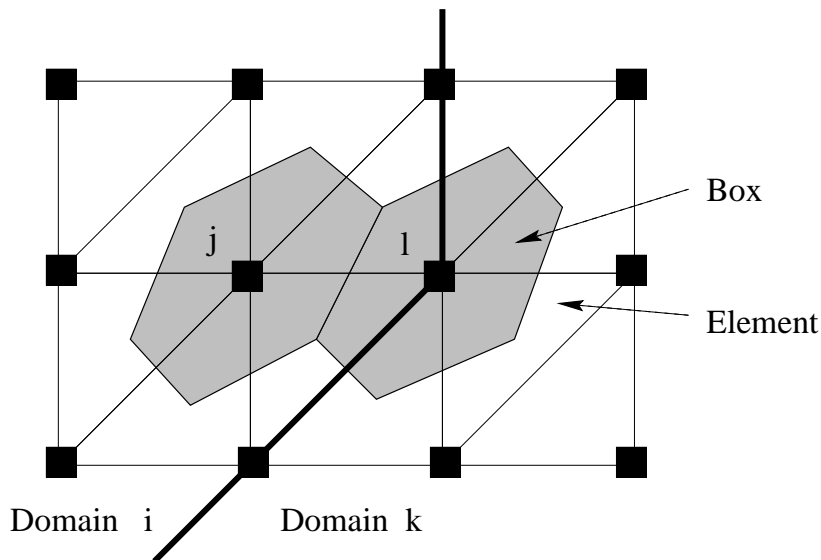


Fig. 7.6 Duale Vernetzung und Gebietsaufteilung

A Begleitendes Praktikum

A.1 Durchführung des Praktikums

Es gibt mittlerweile einige Einführungsbücher in MPI, u.a. [GLS 94], [Pac 97]¹. Darin wird die Handhabung von MPI in den Vordergrund gestellt. Bei unserem Praktikum benutzen wir zwar auch MPI, jedoch erstens nur eine kleine Untermenge der verfügbaren Rufe und zweitens könnte das Praktikum auch genauso mit PVM (oder einer anderen parallelen Bibliothek) durchgeführt werden.

Das Praktikum selbst konzentriert sich nur auf die für die Implementierung von iterativen Verfahren notwendigen Parallelisierungsansätze und führt den Anwender schrittweise bis zu einfachen iterativen Lösern und der Parallelisierung eines Löser für die Stokesgleichung in 2D. Zur Erreichung dieses Ziels sind 5 Praktika á 3 h vorgesehen, wobei die tatsächlich benötigte Zeit stark von der individuellen Programmiererfahrung abhängt. Zur Durchführung des Praktikums reicht schon ein einfacher LINUX-PC.

Das Praktikum wird in FORTRAN und C durchgeführt. Für einige Praktikumsaufgaben können vorbereitete Programmteile benutzt werden, insbesondere wird der Code des seriellen Stokeslösers bereitgestellt (Dank an Dr. Michael Kuhn). Bei Bedarf werden die vorbereitete Musterlösungen benutzt. Im gesamten Praktikum werden nur blockierende Kommunikationsrufe verwendet, da diese einfacher zu Verstehen sind als die größere Anzahl von nichtblockierenden Rufen.

A.2 Programme für das Praktikum

Die vorbereiteten Programmteile sowie Musterlösungen können via Internet <http://www.numa.uni-linz.ac.at/Staff/ghaase/praktikum.html> bzw. per email ghaase@numa.uni-linz.ac.at an den Autor angefordert werden.

¹<http://www.usfca.edu/mpi>

Gegenwärtig unterstützen wir die MPI-Plattformen (vorzugweise die LAM-Distribution) unter LINUX, Solaris, Aix und Irix, da diese Programme dann auch auf großen Parallelrechnern laufen. Jedoch sind das Praktikum und die Codes darin nicht von den angegebenen Plattformen abhängig. Eine Praktikumsversion für Windows ist in Vorbereitung.

A.3 Die Praktikumsaufgaben

A.3.1 Vorbereitungen

MPI unter SOLARIS, LINUX, AIX, IRIX etc.

Das Message Passing Interface enthält mehr als 140 Funktionen jeweils in F77, C und C++ verfügbar. Schon mit nur 6 Funktionen kann man parallele Programme schreiben, die meisten anderen Funktionen basieren auf diesen. Im Praktikum verwenden wir die folgenden Funktionen.

Basisfunktionen	MPI_Init
	MPI_Finalize
	MPI_Send
	MPI_Recv
	MPI_Comm_rank
	MPI_Comm_size
Zusatzfunktionen	MPI_Barrier
	MPI_Bcast
	MPI_Gather
	MPI_Scatter
	MPI_Reduce
	MPI_Allreduce

Online Hilfe

MPI Rufe	http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html
MPI Homepage	http://www.mcs.anl.gov/mpi/index.html
LAM Implementierung	http://www.mpi.nd.edu/lam
mit dem ftp-download	http://www.mpi.nd.edu/lam/download/

Start in einem Pool von Workstations

- In den Pool einloggen (Computername *myhost*).
- Überprüfen ob man sich auf einem anderen Computer des Pools via `rlogin yourhost`

einloggen kann ohne ein Passwort eingeben zu müssen.

Falls ein Passwort verlangt wird - dieses eingeben und *.rhosts* editieren

```
vi ~/.rhosts
```

durch Hinzufügen einer Zeile (hier: symbolische Namen!)

```
myhost my_user_name
```

Installieren der Beispielcodes

- Neues Verzeichnis anlegen:

```
mkdir Course
```

```
cd Course
```

- Beispiel kopieren und auspacken:

```
cp tmp/Example.tar .
```

```
tar xf Example.tar
```

- Natürlich kann auch die Lösung kopiert werden - aber erst selbst probieren!

```
cp tmp/Solution.tar .
```

Start von LAM-MPI

■ Setzen der Umgebungsvariablen *archi* (bash-shell) mit einer Plattform aus {LINUX, IRIX, SOLARIS, AIX},

```
export archi=LINUX
```

bzw. obige Zeile in das File `$(HOME)/.bashrc` schreiben.

■ Initialisiere das LAM-MPI.

```
lamboot
```

Nunmehr können mehrere Prozesse parallel auf dem lokalen Computer oder auf einem vordefinierten Pool von Computern laufen.

■ Falls bereits eine eigenes File (wie *Example/src/mynode*) mit den Namen der Poolcomputer existiert, dann kann man MPI auch wie folgt starten:

```
lamboot -v mynode
```

Nunmehr laufen mehrere Prozesse parallel auf den in *mynode* angegebenen Computern.

Beenden von LAM-MPI

- Beenden aller MPI-Prozesse, welche nicht ordnungsgemäß beendet wurden.

`lamclean`

- Beenden der MPI-Sitzung.

`wipe`

A.3.2 Das erste parallele Programm

E1 Kompiliere das Programm in *Example/src/firstc* (*Example/src/firstf*) für `archi = LINUX` (bzw. entsprechende Plattform) via `make`.
Starte das Programm!

Die nachfolgenden MPI-Funktionen erfordern einen *communicator* als Parameter. Dieser Kommunikator beschreibt die Menge (*group*) von Prozessen, welche an der entsprechenden MPI-Funktion teilnehmen. Der Standardfall, daß alle Prozesse teilnehmen, wird durch den Kommunikator *MPI_COMM_WORLD* beschrieben. Im Praktikum beschränken wir uns auf diesen einfachen Fall. Hierzu müssen wir eine Variable eines speziellen MPI-Types

MPI_Comm `icomm = MPI_COMM_WORLD;`

(in FORTRAN: INTEGER) anlegen, welche als Parameter benutzt wird!

E2 Schreibe Dein erstes paralleles Programm unter Nutzung der Rufe

MPI_Init und **MPI_Finalize**,

kompile es und starte 4 Prozesse (evtl. ohne Option `-lamd`)
`mpirun -c 4 -lamd first.LINUX`

E3 Benutze die Funktionen

MPI_Comm_rank und **MPI_Comm_size**,

um die Anzahl der laufenden Prozesse und die eigene Nummer in der Topologie (*rank*) zu bestimmen. Nur der Masterprozeß (0) soll die Anzahl der laufenden Prozesse ausgeben.

E4 Nutze die in *greetings.c(greetings.f)* gegebene Funktion

Greetings(myid,numprocs,icomm)

Schaue die dortigen Rufzeilen von

MPI_Send und **MPI_Recv** !

genau an!

A.3.3 Blockierende Kommunikation

E5 Schreibe eine Funktion

Send_ProcD(to,nin,xin,icomm)

welche *nin* Double-Precision-Zahlen des Feldes *xin* zum Prozeß *to* sendet. Beachte, daß der empfangende Prozeß *to* im allgemeinen nicht weiß, wie lang das zu empfangende Datenpaket sein wird.

E6 Schreibe eine Funktion

Recv_ProcD(from,nout,xout,maxbuf,icomm),

entsprechend zu E5, welche *nout* Double-Precision-Zahlen in einem Feld *xout* von Prozeß *from* empfängt. Der empfangende Prozeß besitzt keinerlei Information über die Länge des zu empfangenden Datenpakets, daher ist *nout* ein Ausgabeparameter! *maxbuf* beinhaltet die maximale Länge des Feldes *xout*.

E7 Teste die Funktionen aus E5 und E6 zuerst mit 2 Prozessen, sodaß Prozeß 1 sendet und Prozeß 0 empfängt. Erweitere den Test mit mehreren Prozessen.

E8 Kombiniere E5 und E6 zur Funktion

ExchangeD(yourid,nin,xin,nout,xout,maxbuf,icomm),

welche Double-Precision-Zahlen zwischen dem eigenen Prozeß und einem anderen Prozeß *yourid* austauscht. Die restlichen Parameter sind die gleichen wie in E5, E6. Teste die Funktion mit 2 und mehr Prozessen.

Was war doch gleich ein Deadlock (S. 36)?

A.3.4 Globale Operationen

Der Double-Precision-Vektor \underline{x} sei blockweise disjunkt verteilt, d.h., verteilt über alle Prozesse s ($s=0, P-1$), daß $\underline{x} = (\underline{x}_0^T, \dots, \underline{x}_s^T)^T$ gilt.

E9 Schreibe eine Funktion

DebugD(nin,xin,icomm)

welche *nin* Double-Precision-Zahlen des Vektors *xin* auf dem Bildschirm ausgibt. Starte das Programm mit mehreren Prozessen.

⇒ Alle Prozesse schreiben durcheinander ihre lokalen Vektoren aus, d.h., eine Fehlersuche wird sehr erschwert.

Verbessere die Funktion **DebugD** derart, daß Prozeß 0 vom Terminal die Nummer des Prozesses einliest, welcher Daten ausgeben soll. Nutze

MPI_Bcast

um diese Information allen Prozessen mitzuteilen, sodaß diese entsprechend reagieren können. Evtl. muß **MPI_Barrier** benutzt werde, um die Ausgabe zu synchronisieren.

E10 Vertausche das globale Minimum und Maximum des Vektors \underline{x} !
Benutze hierbei die Funktionen

MPI_Gather, MPI_Scatter/MPI_Bcast und Exchanged.

Wie kann man den Kommunikationsaufwand reduzieren?

Hinweis : Berechne zuerst die lokalen Minima und Maxima und laß danach einen Prozeß die globalen Größen bestimmen.

Man kann alternativ auch gleich die Funktion **MPI_Allreduce** mit den Operationen **MPI_Minloc/MPI_Maxloc** benutzen.

E11 Schreibe eine Funktion, welches das globale Skalarprodukt

Skalar(n,x,y,icomm)

zweier Double-Precision-Vektoren x und y der lokalen Länge n berechnet. Benutze

MPI_Allreduce mit der Operation **MPI_SUM**.

A.3.5 Lokaler Datenaustausch

Das Einheitsquadrat $[0, 1]^2$ wird gleichmäßig zeilenweise in $procx \times procy$ Rechtecke Ω_i unterteilt. Die Prozeßnummer ($rank$) stimmt mit der Teilgebietsnummer überein.

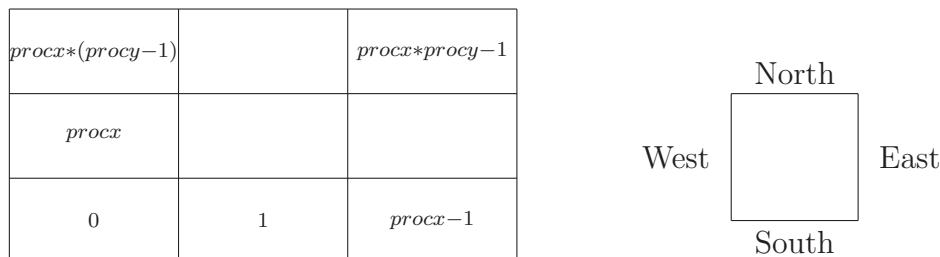


Fig. A.1 Gebietsaufteilung und lokale Richtungen

Die Funktion

IniGeom(myid,procx,procy,neigh,color)

aus *example/src/accuc* (*example/src/accuf*) generiert entsprechend obiger Gebietsaufteilung die topologischen Informationen, welche im Integerfeld *neigh(4)* gespeichert werden. Durch *color* wird eine Red-Black-Einfärbung der Prozesse definiert. Die Funktion

IniCoord(myid,procx,procy,xl,xr,yb,yt)

generiert die Koordinaten der unteren linken Ecke (xl, yb) und der oberen rechten Ecke (xr, yt) jedes Teilgebietes.

E12 Realisiere einen lokalen Datenaustausch einer Double-Precision-Zahl zwischen jedem Prozessor und dessen (über eine gemeinsame Kante verbundenen) Nachbarn. Benutze die Funktion **ExchangeD** aus E8.

Jedes Teilgebiet Ω_i sei wiederum gleichmäßig in $(nx - 1) * (ny - 1)$ Rechtecke unterteilt, welche ein Dreiecksnetz beinhalten (nx, ny seien für alle Teilgebiete gleich!). Jeder Mittelpunkt (Knoten) einer Dreiecksseite entspricht 2 Kom-

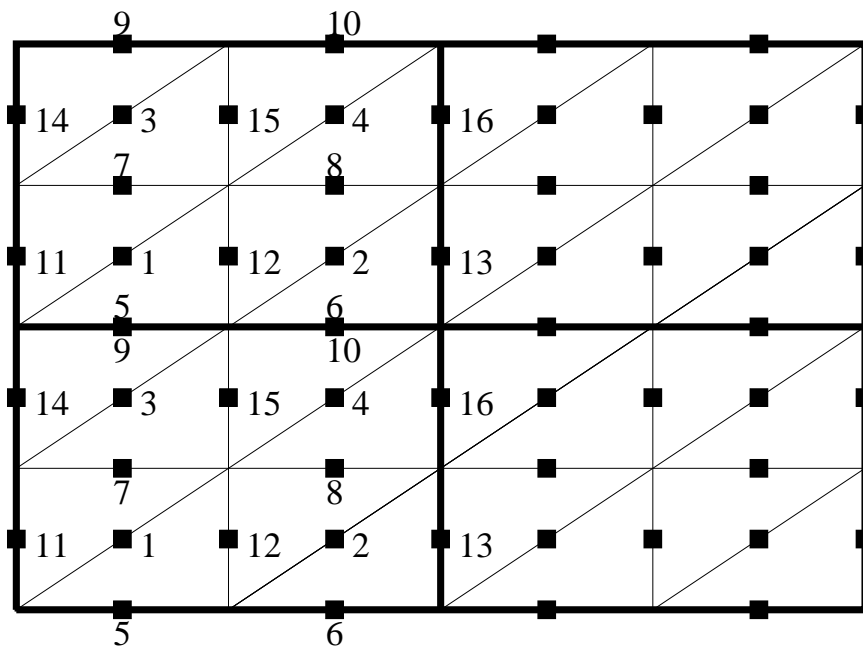


Fig. A.2 Diskretisierung in lokaler Numerierung mit 4 Teilgebieten und $nx = ny = 3$

ponenten eines Vektors, z.B., der Geschwindigkeit einer Flüssigkeit in diesem Punkt. Somit haben wir

- $2 * (nx - 1) * (ny - 1)$ Unbekannte auf der Diagonalen

- $2 * (nx - 1) * ny$ Unbekannte auf den horizontalen Linien
- $2 * nx * (ny - 1)$ Unbekannte auf den vertikalen Linien,

d.h., $nd := 2 * (3 * nx * ny - 2 * nx - 2 * ny + 1)$ lokale Unbekannte pro Teilgebiet. Wir benutzen eine spezielle Numerierung, welche zeilenweise mit den Werten auf der Diagonalen beginnt, gefolgt von den horizontalen Linien und schließlich den vertikalen Linien. Die Funktion

GetBound(id,nx,ny,w,s)

kopiert die dem Rand South(id=1), East (id=2), North (id=3), West (id=4) entsprechenden Werte des Vektor w in den Hilfsvektor s . Umgekehrt addiert die Funktion

AddBound(id,nx,ny,w,s)

die Werte in s auf die entsprechenden Komponenten von w . Diese Funktion kann zur Akkumulation von Daten auf den Teilgebietsrändern benutzt werden, was eine typische Operation in parallelen Programmen ist.

E13 Schreibe eine Funktion, welche einen verteilten Vektor w (Double Precision) (S. 82) akkumuliert. Die Rufzeile könnte folgendermaßen aussehen

VecAccu(nx,ny,w,neigh,color,myid,icomm),

wobei w sowohl Eingabe- als auch Ausgabeparameter ist.

A.3.6 Iterative Löser

Wir betrachten das Stokesproblem in schwacher Formulierung (Variationsformulierung).

Finde $(\mathbf{u}, p) \in \mathbf{X} \times \mathbf{M} := \mathbf{H}^1(\Omega) \times L_2^*(\Omega)$ sodaß

$$\begin{aligned} \int_{\Omega} \nabla \mathbf{u} \nabla \mathbf{v} \, dx - \int_{\Omega} \nabla \cdot \mathbf{v} p \, dx &= \int_{\Omega} \mathbf{f} \mathbf{v} \, dx \quad \forall \mathbf{v} \in \mathbf{X} \\ \int_{\Omega} \nabla \cdot \mathbf{u} q \, dx &= 0 \quad \forall q \in \mathbf{M}, \end{aligned}$$

in $\Omega := (0, 1)^2$ gilt, mit den Randbedingungen $\mathbf{u} = (1, 0)^T$ für $y = 1$ und $\mathbf{u} = 0$ sonst auf $\partial\Omega$. Die $(\mathbf{H}^1(\Omega))$ nichtkonformen P_1 - P_0 Elemente sind eine stabile Diskretisierung dieses Problems. Damit erhalten wir das Sattelpunkt-

problem (7.7)

$$\begin{pmatrix} A & B \\ B^T & 0 \end{pmatrix} \begin{pmatrix} \mathbf{u} \\ p \end{pmatrix} = \begin{pmatrix} \mathbf{f} \\ 0 \end{pmatrix} .$$

Mit gegebenen Vorkonditionierern \hat{A} für A und \hat{C} für $-B^T A^{-1} B$ kann das Sattelpunktproblem mittels des vorkonditionierten Arrow–Hurwitz Algorithmus (S. 135) gelöst werden.

$$\begin{aligned} \hat{A}(\mathbf{u}^{k+1} - \mathbf{u}^k) &= \mathbf{f} - A\mathbf{u}^k - Bp^k \\ \hat{C}(p^{k+1} - p^k) &= B^T \mathbf{u}^{k+1} \end{aligned}$$

Eine serielle Programmversion ist in *seqc* (*seqf*) zu finden. Sie benutzt cg-Iterationen mit Diagonalskalierung für \hat{A} und $\hat{C} := M$, wobei M die Massenmatrix (Gramsche Matrix) bzgl. des Druckes p darstellt. Die Matrizen A, B, \hat{C} werden in

```
GetGradMatrix(nx, ny, 0, 0, A, id, ik)
GetDivMatrix(nx, ny, 0, 1, B, idb, ikb)
GetMassMatrix(nx, ny, 1, 1, M)
```

generiert. Die Diagonale von A erhält man aus

```
AccuDiag(nx, ny, sk, id, d)
```

Der Vektor der rechten Seite wird mit

```
GetRhs(nx, ny, f, 0, 1, 0, 1) , GetRhsPressure(nx, ny, g, 0, 1, 0, 1)
```

initialisiert. Randbedingungen werden mittels

```
DirichletBc(nx, ny, A, id, ik, f, u)
```

eingebaut und die Anfangslösung wird mit

```
SetU(nx, ny, u, 0, 1, 0, 1)
```

initialisiert. Die Operationen $w := w + \alpha A u$, $p := p + \alpha B^T u$, $u := u + \alpha B p$ sind implementiert in

```
CrsMult(1, nu, w, u, id, ik, A, alpha)
BTCrsMult(np, nu, p, u, idb, ikb, B, alpha)
BCrsMult(np, nu, p, u, idb, ikb, B, alpha) .
```

E14 Leite aus dem sequentiellen Code eine parallele Version ab!

B Einige Internetadressen

B.1 MPI: Message Passing Interface

- Die MPI Homepage¹ mit vielen weiteren Links.
- Die Beschreibung sämtlicher MPI Rufe².
- Die MPICH Implementierung von MPI³.
- Die LAM Implementierung von MPI⁴, mit dem ftp-download⁵.

B.2 PVM: Parallel Virtual Machine

- PVM Homepage⁶ mit vielen weiteren Tools.
- Kurzbeschreibung PVM⁷.
- Vereinigung von MPI und PVM : PVMPI⁸.
- Neues Projekt der PVM-Gruppe HARNESS⁹.

B.3 Weitere Links

- Lecture on Scientific Supercomputing¹⁰, mit vielen technischen Details über Vektor- und Parallelrechner.

¹<http://www.mcs.anl.gov/mpi/index.html>

²<http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html>

³<http://www.mcs.anl.gov/mpi/mpich/index.html>

⁴<http://www.mpi.nd.edu/lam>

⁵<http://www.mpi.nd.edu/lam/download/>

⁶<http://www.epm.ornl.gov/pvm>

⁷<http://www.uni-karlsruhe.de/Hartmut.Haefner/pvm3.html>

⁸<http://www.netlib.org/mpi/pvmpi/pvmpi.html>

⁹<http://www.epm.ornl.gov/harness/>

¹⁰<http://www.uni-karlsruhe.de/Uni/RZ/Personen/rz03/book>

- VCPC¹¹ : European Centre for Parallel Computing in Wien.
- GSCI: German Scientific Computing Page¹².
- Konferenzen und Informationen zu Gebietszerlegungsmethoden¹³.
- MGNet¹⁴ : Multigrid und Gebietszerlegungsmethoden.
- Netlib¹⁵ : Mathematische Software, Artikel, etc.
- Tools und Benchmarks für die Parallelisierung¹⁶.
- EUROPORT¹⁷ Portierung kommerzieller Software auf Parallelrechner.
- Top 500 Supercomputer¹⁸.

B.4 Parallelrechner

Einen guten Überblick über gängige Hochleistungscomputer vermittelt eine Internetseite von Ken Hawick, John Dongarra u.a.¹⁹. Hier noch einige persönliche Anmerkungen zu einigen Parallelrechnern.

- Dank des sehr guten Preis-Leistungs-Verhältnisses der PCs und der freien Verfügbarkeit von LINUX²⁰ und da, z.B., bei der S.u.S.E.²¹-Distribution MPI und PVM mitgeliefert werden, läßt sich relativ preiswert ein Parallelrechner im Eigenbau herstellen. In analoger Weise können vorhandene Rechner zu einem Cluster von Workstations vereinigt werden.

Einige Beispiele hierzu aus den Jahren 1998/99 sind Parnass₂²² an der Universität Bonn, CLOWN²³ in Paderborn, die Rechner Loki²⁴ und Avalon²⁵ in Los Alamos.

- Seit einigen Jahren bieten die Workstationhersteller IBM, HP, SGI, SUN, Compaq (DEC) ihre Parallelrechner an, welche alle einen shared memory auf dem mit bis zu 4 Prozessoren bestückten Board haben und zwischen den Boards ein message passing realisieren.

¹¹<http://www.vcpc.univie.ac.at>

¹²<http://scicomp.math.uni-augsburg.de/scicomp>

¹³<http://www.ddm.org/>

¹⁴<http://www.mgnet.org/>

¹⁵<http://www.netlib.org/>

¹⁶<http://www.nhse.org/ptlib>

¹⁷<http://www.gmd.de/SCAI/europort/>

¹⁸<http://www.top500.org>

¹⁹<http://nhse.npac.syr.edu/hpccsurvey/>

²⁰<http://www.cs.helsinki.fi/linux/>

²¹<http://www.suse.de/>

²²<http://www.wissrech.iam.uni-bonn.de/research/projects/parnass2/>

²³<http://www.ix.de/ix/artikel/1999/01/010/>

²⁴<http://loki-www.lanl.gov/>

²⁵<http://cnls.lanl.gov/avalon/>

- Die Firma Thinking Machines²⁶ hat die Produktion von eigenen Parallelrechnern eingestellt und sich auf die Auswertung großer Datenmengen (data mining) spezialisiert. Auf dem gleichen Gebiet arbeitet auch der klassische Hersteller von SIMD-Rechnern, die Firma MasPar²⁷, unter dem neuen Firmennamen NeoVista²⁸.
- Der einzige Parallelrechnerhersteller Deutschlands, die Firma Parsytec²⁹, ist ebenfalls nicht mehr im Bereich der Numerik vertreten. Vielmehr ist diese Firma mittlerweile sehr erfolgreich in der Echtzeitbilderkennung (Qualitätskontrolle) tätig und setzt dort ihre Parallelrechner-Technologie ein.
- Der letzte Parallelrechner der Firma nCube³⁰ war der nCUBE2 mit einer klassischen Hypercube-Architektur. Leider verzögerte sich die Produktion der neuen Prozessoren (Kommunikation und Arithmetik in einem) für den nCUBE3³¹ derartig, daß seit 1993 keine Neuinstallationen im Berechnungsbereich mehr bekannt sind. Die Firma ist heute auf große Datenmengen im Multi-Mediabereich spezialisiert und nutzt dazu ihr nCUBE-System mit bis zu 512 Prozessoren.
- Die Firma Kendall Square Research (KSR) existiert seit Mitte der 90er Jahre nicht mehr.
- Das Parallelrechnerprogramm von Convex wird in Form der SPP-Reihe der Firma Hewlett-Packard³² fortgeführt.
- Die Parallelrechnerentwicklung von Cray Research³³ ist von SGI³⁴ übernommen worden.
- Quadrics Supercompers World³⁵ ist eine Firma in Großbritannien. Hervorgegangen ist die Firma aus dem APEmille Projekt³⁶, welches SIMD-Rechner mit einem 3D-Torus als Topologie konzipierte, und der Firma Meiko³⁷. Leider ist nichts über Realisierungen dieser Computer bekannt. Das letzte Parallelrechnermodell von Meiko aus dem Jahre 1992, die CS-2, basierte auf einer Fat Tree Topologie.
- Einen neuen Versuch die klassischen shared memory Parallelrechner wieder zu produzieren, stellt die Firma Tera³⁸ dar. Der mit viel US-Regierungsgeldern

²⁶<http://www.think.com>

²⁷<http://nhse.npac.syr.edu/hpccsurvey/orgs/maspar/maspar.html>

²⁸<http://www.neovista.com/>

²⁹<http://www.parsytec.de>

³⁰<http://www.ncube.com>

³¹<http://www.ncube.com/products/cpu.html>

³²<http://www.hp.com>

³³<http://www.cray.com>

³⁴<http://www.sgi.com>

³⁵<http://www.quadrics.com/>

³⁶<http://chimera.roma1.infn.it/ape.html>

³⁷<http://www.meiko.com/>

³⁸<http://www.tera.com>

unterstützte Start scheint einen konkurrenzfähigen Parallelrechner hervorgebracht zu haben. Der Hintergrund dieser Rechnerentwicklung scheint darin zu bestehen, daß 20-30 Jahre alte Softwarepakete aus dem Ingenieurbereich (mit viel Entwicklungsarbeit darin!) einen Parallelrechner benötigen der ihr Programmiermodell unterstützt.

C Einige Zitate

- Ada Augusta, Countess of Lovelace (1815-1851). Sie schrieb Programme für die ersten Computer von Babbage.

*It can do only what we know to order it to do.*¹

- Peter Cochrane (Forschungschef der British Telecom), 1998, über Minderheitenschutz.

*Es gibt 6 Milliarden Menschen auf der Welt - und 14 Milliarden Mikroprozessoren. Wir sind jetzt schon in der Minderheit.*²

- Edsger W. Dijkstra (1969).

Program testing can be used to show the presence of bugs, but never to show their absence.

- Edsger W. Dijkstra (1986).

The question of whether computers can think is just like the question of whether submarines can swim.

- Tony Hoare.

I don't know what the programming language of the year 2000 will look like, but I know it will be called FORTRAN.

- John v. Neumann.

If people do not believe that mathematics is simple, it is only because they do not realize how complicated life is.

- Stanisław Lem in „Lokaltermin“.

Es hieß, 49 Prozent seiner Leute seien verrückt. ... Man brauchte sich nur ein paar der letzten Ideen seiner Leute vorzunehmen : Da man sich nicht in der Zeit bewegen kann, muß man die Zeit bewegen. Wenn Energie nicht zwischen Orten gleicher Temperatur fließen will, muß man sie zwingen, indem man Löcher macht. ... Diese zweifellos hirnrissige Idee leitete eine neue Ära in der Physik ein.

- Stanisław Lem in „Die Stimme des Herrn“.

Es heißt schon seit langem, der Spezialist sei ein Barbar, dessen Unwissenheit nicht allumfassend ist.

¹Gefunden im Arithmeum der Stadt Bonn.

²Die Woche, 49/1998, pp.11

Literaturverzeichnis

- [Bär 93] Bär, S.: *Forschen auf Deutsch: Der Macchiavelli für Forscher und solche, die es noch werden wollen*. Deutsch 1993
- [Bas 96] Bastian, P.: *Parallele adaptive Mehrgitterverfahren*. Teubner Scripten zur Numerik. Teubner 1996
- [BBC⁺ 94] Barrett, R.; Berry, M.; Chan, T.; Demmel, J.; Donato, J.; Dongarra, J.; Eijkhout, V.; Pozo, R.; Romine, C.; der Vorst, H. V.: *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*³. Philadelphia, PA: SIAM 1994
- [BDFN 93] Burkhardt, S.; Drey, K.-D.; Friedrich, V.; Nowak, O.: *Parallele Rechnersysteme: Programmierung und Anwendung*. Berlin/München: Verlag Technik 1993
- [BF 91] Brezzi, F.; Fortin, M.: *Mixed and Hybrid Finite Element Methods*, Bd. 15 von *Springer Series in Computational Mathematics*. Springer-Verlag 1991
- [BPS 89] Bramble, J. H.; Pasciak, J. E.; Schatz, A. H.: The construction of preconditioners for elliptic problems by substructuring I – IV. *Math. Comp.* (1986, 1987, 1988, 1989). 47, 103–134, 49, 1–16, 51, 415–430, 53, 1–24
- [CKL 96] Carstensen, C.; Kuhn, M.; Langer, U.: *Fast Parallel Solvers for Symmetric Boundary Element Domain Decomposition Equations*. Report 500, Institute for Mathematics, Johannes Kepler University Linz 1996. Submitted for publication.
- [Dij 65] Dijkstra, E. W.: *Cooperating Sequential Processes*. Techn. Ber. Technical Report EWD-123, Technological University of Eindhoven 1965. Reprinted in [Gen 68]
- [Dry 84] Dryja, M.: A finite element-capacitance Matrix Method for Elliptic Problems on Regions Partitioned into Substructures. *Numer. Math.* **34** (1984) 153–168

³http://www.netlib.org/linalg/html_templates/report.html

- [EJL 92] Evans, D.; Joubert, G.; Lidell, H. (Hrsg.): Parallel Computing '91. North-Holland 1992. Proceedings of the Int. Conference on Parallel Computing, London, U.K., 3-6 September 1991
- [FFLM 97] Feistauer, M.; Felcman, J.; Lukacova-Medvidova, M.: On the convergence of a Combined Finite Volume - Finite Element Method for Nonlinear Convection - Diffusion Problems. *Num.Methods for Part.Diff.Eqs* **13** (1997) 163–190
- [FFLMW 97] Feistauer, M.; Felcman, J.; Lukacova-Medvidova, M.; Warnecke, G.: Error Estimates of a Combined Finite Volume - Finite Element Method for Nonlinear Convection - Diffusion Problems. Preprint 96-27⁴, University of Magdeburg 1997
- [Fly 66] Flynn, M.: Very high-speed computing systems. In: *Proceedings of the IEEE*, Bd. 54. December 1966 S. 1901–1909
- [Fos 94] Foster, I.: Designing and Building Parallel Programs. Addison-Wesley 1994
- [Fox 88] Fox, G.: Solving Problems on Concurrent Processors. Prentice Hall 1988
- [Ge 90] Gallivan, K.; et.al.: Parallel Algorithms for Matrix Computations. Philadelphia: SIAM 1990
- [Gen 68] Genuys, F. (Hrsg.): Programming Languages. London, England: Academic Press 1968
- [GLS 94] Gropp, W.; Lusk, E.; Skjellum, A.: Using MPI. Cambridge, London: The MIT Press 1994
- [GO 96] Golub, G.; Ortega, J. M.: Scientific Computing: Eine Einführung in das wissenschaftliche Rechnen und Parallele Numerik. Stuttgart: Teubner 1996. Dt. Übersetzung
- [Gro 94] Groh, U.: Local realization of vector operations on parallel computers. Preprint SPC 94-2⁵, TU Chemnitz 1994. In german
- [Gus 78] Gustafsson, I.: A class of first order factorization methods. *BIT* **18** (1978) 142–156
- [Hac 91] Hackbusch, W.: Iterative Lösung großer schwachbesetzter Gleichungssysteme. Studienbücher Mathematik. Nr. Bd. 69. Teubner 1991
- [HB 91] Heermann, D.; Burkitt, A.: Parallel Algorithms in Computational Science, Bd. 24 von *Information Sciences*. Springer, Berlin 1991
- [HLM 91] Haase, G.; Langer, U.; Meyer, A.: The Approximate Dirichlet Decomposition Method. Part I,II. *Computing* **47** (1991) 137–167

⁴<http://www.math.uni-magdeburg.de/preprints/shadows/96-27report.html>

⁵http://www.tu-chemnitz.de/ftp-home/pub/Local/mathematik/SPC/spc94_2.ps.gz

- [Hwa 93] Hwang, K.: Advanced computer architecture: parallelism, scalability, programmability. McGraw-Hill computer engineering series. McGraw-Hill 1993
- [Joh 97] John, V.: Parallele Lösung der inkompressiblen Navier-Stokes Gleichungen auf adaptiv verfeinerten Gittern. Dissertation, Otto-von-Guericke-Universität Magdeburg August 1997
- [Jun 96] Jung, M.: On the parallelization of multi-grid methods using a non-overlapping domain decomposition data structure. Applied Numerical Mathematics **20** (1996)
- [KGGK 94] Kumar, V.; Grama, A.; Gupta, A.; Karypis, G.: Introduction to Parallel Computing: Design and Analysis of Algorithms. Redwood City, California: Benjamin/Cummings Publishing Company 1994
- [Krö 97] Kröner, D.: Numerical Schemes for Conservation Laws. WILEY and TEUBNER 1997
- [KST 95] Keyes, D. E.; Saad, Y.; Truhlar, D. G.: Domain-Based Parallelism and Problem Decomposition Methods in Computational Science and Engineering. SIAM 1995
- [KX 94] Keyes, D. E.; Xu, J. (Hrsg.): Domain Decomposition Methods in Scientific and Engineering Computing, Providence, Rhode Island 1994. AMS. Proceedings of the Seventh Int. Conference on Domain Decomposition, October 27-30, 1993, The Pennsylvania State University
- [Law 89] Law, K. H.: A parallel finite element solution method. Computer and Structures **23** (1989) 6 845–858
- [Lee 92] Leer, B.: Flux vector splitting for the Euler equations. In: *Proc. 8th International Conference on Numerical Methods in Fluid Dynamics*. Springer, Berlin, 1992
- [LP 98] Lucquin, B.; Pironneau, O.: Introduction to Scientific Computing. Wiley 1998
- [Mey 90] Meyer, A.: A Parallel Preconditioned Conjugate Gradient Method Using Domain Decomposition and Inexact Solvers on Each Subdomain. Computing **45** (1990) 217–234
- [MS 96] Miller, R.; Stout, Q. F.: Parallel Algorithms for Regular Architectures: Meshes and Pyramids. The MIT Press 1996
- [MvdV 77] Meijerink, J.; van der Vorst, H.: A Iterative Solution for Linear Systems of Which the Coefficient Matrix is a Symmetric M-Matrix. Math. Comp. **31** (1977) 148–162
- [Pac 97] Pacheco, P.: Parallel Programming with MPI. Morgan Kaufmann Publishers 1997
- [Rec 94] Rechenberg, P.: Was ist Informatik ? : Eine allgemeinverständliche Einführung. München/Wien: Carl Hanser 1994

- [Saa 92] Saad, Y.: Iterative methods for sparse linear systems. *Frontiers in Applied Mathematics*. Boston: PWS Publishing Company 1992
- [SBG 96] Smith, B.; Bjørstad, P.; Gropp, W.: *Domain Decomposition : Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press 1996
- [SGI 97] SGI: Origin Servers. Technical report, Silicon Graphics Computer Systems April 1997
- [SRWH 97] Stals, L.; Råde, U.; Weiß, C.; Hellwanger, H.: Data Local Iterative Methods for the Efficient Solution of Partial Differential Equations⁶. In: *Proceedings of The Eight Biennial Computational Techniques and Applications Conference*. Adelaide, Australia, Sept. 1997
- [SW 81] Steger, J.; Warming, R.: Flux vector splitting of the inviscid gasdynamic equations with applications to finite difference methods. *J. Comput. Phys.* **40** (1981) 263–293
- [TCK 92] Tong, C. H.; Chan, T. F.; Kuo, C. J.: Multilevel Filtering Preconditioners: Extensions to More General Elliptic Problems. *SIAM J. Sci. Stat. Comput.* **13** (1992) 227–242
- [TK 96] Topping; Khan: *Parallel Finite Element Computations*. Edinburgh: Saxe-Coburg Publications 1996
- [TU 94] Thomas, G.; Ueberhuber, C. W.: *Visualization of Scientific Parallel Programs*. Springer, Berlin 1994
- [Vij 86] Vijayasundaram, G.: Transonic flow simulation using an upstream centered scheme of Godunov in finite elements. *J. Comput. Phys.* **63** (1986) 416–433
- [vL 92] van Loan, C.: *Computational Framework for the Fast Fourier Transformation*. *Frontiers in Applied Mathematics*. Philadelphia: SIAM 1992
- [Zul 97] Zulehner, W.: *Numerische Methoden in der Strömungsmechanik*. University of Linz⁷ 1997

⁶<http://wwwbode.informatik.tu-muenchen.de/Par/arch/cache>

⁷<http://nathan.numa.uni-linz.ac.at/Teaching/cfd.ps>