Parallelisierung und Vektorisierung numerischer Algorithmen

Gundolf Haase
Johannes Kepler Universität Linz
ghaase@numa.uni-linz.ac.at

WS 1998/99, Linz

Ich möchte an dieser Stelle Dr. Matthias Pester von der TU Chemnitz für die freundliche Überlassung seines Vorlesungsskriptes zur Parallelisierung danken, welches in einigen Teilen des vorliegenden Skripts wiederzufinden ist. Gleichzeitig gingen eigene Vorlesungsmitschriften aus der entsprechenden Vorlesung bei Prof. Volkmar Friedrich ein.

„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 hirnrisse Idee leitete eine neue Ära in der Physik ein.“

Stanisław Lem in „Lokaltermin“

„Es heißt schon seit langem, der Spezialist sei ein Barbar, dessen Unwissenheit nicht allumfassend ist.“

Stanisław Lem in „Die Stimme des Herrn“

Inhaltsverzeichnis

Abbildungsverzeichnis

Algorithmenverzeichnis

Kapitel 1

Einführung

1.1 Wozu braucht man Parallelrechner ?

Mikroelektronik/Technologie
↓
Immer höhere Integrationsdichte der Chips.
Immer schnellere Taktfrequenzen.
↓
Leistung früherer Großrechner wird heute von jedem PC übertroffen !

Aber : Die Signalgeschwindigkeit ist endlich ($3 \cdot 10^8\, m/s$). So waren die einzelnen Prozessoreinheiten der schnellsten Cray (1994) im Kreis angeordnet, damit Nachrichten innerhalb *eines* Taktes übermittelt werden können (Zykluszeiten $\approx 1 \cdot 10^{-9}\, s$). Eine weitere Verkleinerung der Chipstrukturen von gegenwärtig ca. $0.25\, \mu m \;=\; 2.5 \cdot 10^{-7} m$ ist ein technologisches Problem (Reproduktion der Chipmasken). Abgesehen von auftretenden Quanteneffekten ist die Atomgröße ($\approx 10^{-10}\, m$) die letzte, nicht mehr zu unterbietende, Grenze.

Heutiger Stand der Rechenleistung[1] (I/97) bei Sun[2], SGI[3], HP[4], DEC[5] :

[1] http://www.specbench.org/results.html
[2] http://www.sun.de
[3] http://www.sgi.de
[4] http://www.hewlett-packard.de
[5] http://www.dec.de

Prozessor	SpecInt95	SpecFloat95
Ultra4001 (SUN)	10	12
R10000 (SGI)	9	17
HP9000 (HP)	12	20
Alpha (DEC)	15	43

Die Entwicklung sehr schneller Rechner*systeme*, d.h. *Prozessor + Speicher + I/O*, wird immer schwieriger und kostspieliger. Mit der notwendigen Komplexität der Chips steigt auch zwangsläufig die "Chance" von Produktion-/Entwurfsfehlern (Fehlerhafte Division bei Intel[6], Überhitzung von Prozessoren bei SGI).

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.*

$$\Downarrow$$

Hohe Rechenleistung.
Hohe Speicherkapazität (+Datenzugriff)
Schnelle Datenbereitstellung und -auswertung (Grafik)

Einige typische Konsumenten von Rechenleistung :

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

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

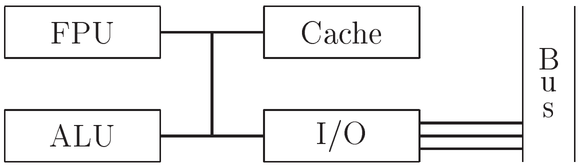
[6]http://www.intel.de

spiele finden. Falls die Rechenleistung einmal ausreichen sollte, will man bestimmt sofort die entsprechende inverse Aufgabe angehen bzw. die eingehenden Parameter bzgl. einer Zielfunktion optimieren.

Klassische von-Neumann-Architektur reicht nicht mehr aus.
↓
Beschleunigung durch *parallele* Verarbeitung.

Parallele Ansätze in von-Neumann-*Rechenwerken*

- Gleitkomma-, Integerarithmetik und I/O parallel möglich.



- Instruction Look Ahead (Befehlscache).
- Memory Interleaving, d.h. getrennter Zugriff auf adressmäßig benachbarte Bytes/Bits, z.B. pro Speicherchip jeweils nur 1 Bit eines Bytes gespeichert.

Pipelining in der Befehlsabarbeitung

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

Das Pipelining 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.

> *Gesetz von H. Gotsch (1953) :*
> Die Kosten eines Rechnersystems steigen nur wie die Quadratwurzel aus seiner Leistung.

Gilt nur für bereits entwickelte Systeme aus technologischer Sicht, da die (fixen, aber rapide steigenden) Kosten für die Entwicklung von Hard- und Software aus Marktgründen auf die Rechenleistung verteilt werden müssen.

Kapitel 2

Parallel- und Vektorrechner

2.1 Klassifikationen

2.1.1 Ebenen der Parallelität

In der Parallelität werden die folgenden Ebenen unterschieden :

Job :
Ganze Jobs laufen parallel zueinander auf verschieden Prozessoren ab. Hierbei tritt keine oder nur eine schwache Interaktion der Jobs zueinander auf.
$\longrightarrow$ bessere Rechnerauslastung
$\longrightarrow$ kürzere Realzeit der Jobs
z.B.: Workstation mit mehreren Prozessoren und Multitasking.

Programm :
Teile eines Programmes laufen auf mehreren Prozessoren ab.
$\longrightarrow$ kürzere Realzeit
z.B.: Parallelrechner

Befehl :
Parallelität zwischen den Phasen (Instruktionen) der Befehlsausführung.
$\longrightarrow$ Beschleunigte Ausführung des gesamten Befehls.
z.B.: serielle Rechner / einzelne Prozessoren

Arithmetik, Bitebene :
Hardwaremäßige Parallelisierung von Integerarithmetik und bitweiser paralleler, aber wortweise serieller Zugriff auf den Speicher bzw. umgekehrt.
$\longrightarrow$ 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.

Pipelining :
Segmentierung von Operationen, welche hintereinander abgearbeitet werden (siehe *Vektorrechner* im nächsten Abschnitt).

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

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

Multiprocessing :
Mehrere voneinander unabhängige Prozessoren mit jeweils eigenem Instruktionssatz. Parallele Ausführung bis hin zu ganzen Programmen oder Jobs.

Bis auf die Variante Prozessorfelder/Multiprocessing können obige Klassifikationen auch kombiniert auftreten.

2.1.3 Grobklassifikation nach Flynn

Im Jahre 1966 unterschied Michael Flynn [Fly66] parallele Architekturen bzgl. des Datenstroms und des Stromes der Programmanweisungen, so daß sich 4 Formen der Parallelität ergeben.

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

Tabelle 2.1: Flynn Klassifikation (Flynn's taxonomy)

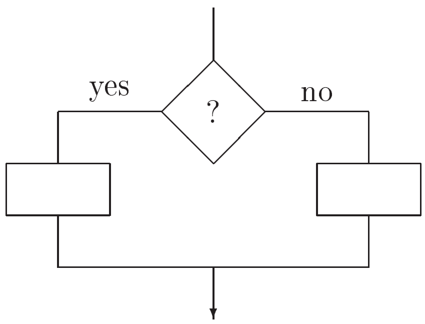
SISD :
Klassische Architektur

- nur prozessorinterne Parallelität (bitparallel, look-ahead)

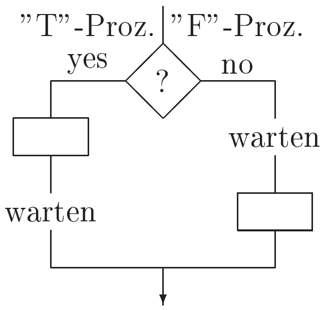
- Pseudeoparallelität mittels Multitasking und Time-Sharing (Durchsatzerhöhung der Anlage)

SIMD :
Parallelität auf Anweisungsebene

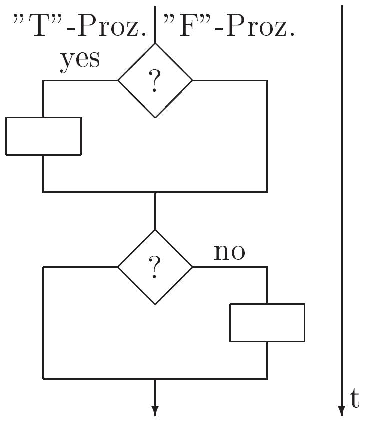
1. Die Daten sind über die Prozessoren verteilt, *ein* Programm wird "im Gleichschritt" auf allen Prozessoren abgearbeitet. Meist sehr *einfache*, dafür aber *viele* Prozessoren (*Prozessorfelder* CM2, MasPar).
   Ein typisches Problem bei SIMD sind Programmalternativen :



Liefert der Test "?" bei 2 Prozessen der SIMD-Maschine unterschiedliche Ergebnisse (T/F), ändert sich das Struktogramm in



,d.h. eigentlich läuft ab :



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 Abb. 2.1, welche auf einen 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" wird i.a. pro Taktzyklus ein Ergebnis y_i geliefert, obwohl in der skalaren Rechnung mehr Takte notwendig sind.

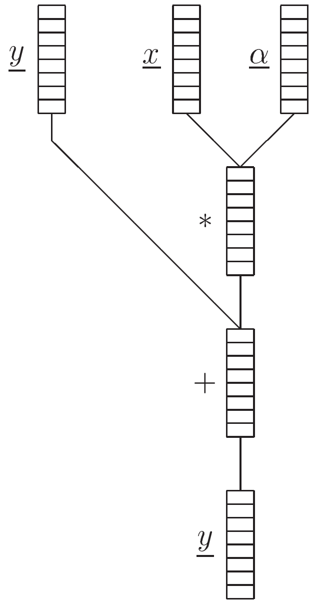


Abbildung 2.1: SAXPY im Vektorrechner

Um hohe MFLOP-Raten zu erreichen, müssen 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 $\underline{y} := \underline{y} + \underline{\alpha}^T \circledast \underline{x}$ effizient behandeln kann.

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

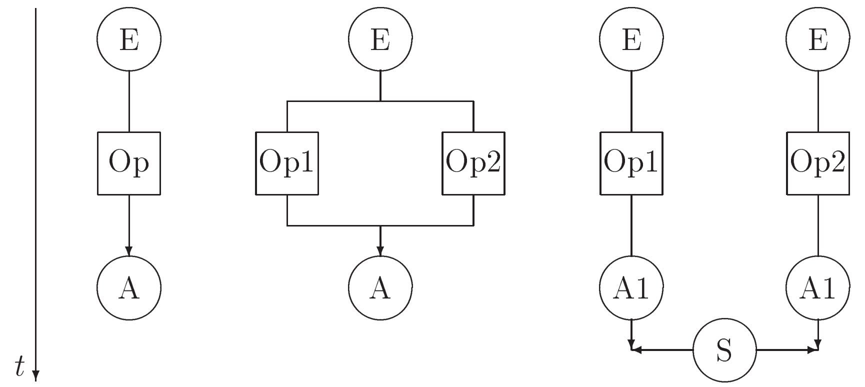
MIMD :
Bedeutet Parallelität auf Programmebene, d.h. jeder Prozessor arbeitet sein eigenes Programm ab. Dies geschieht in der Regel aber nicht unabhängig voneinander, daher unterscheidet man :

1. konkurrierende Prozesse (gemeinsame Ressourcen)

2. kommunizierende Prozesse (Datenfluß, Datenaustausch)

Unterscheide
Programm (statisch) :
Exakte Niederschrift eines Algorithmus

Prozeß (dynamisch) :
Folge von Aktionen, einzige Annahme: positive Geschwindigkeit.



Einzelne Operation

2 parallele Operationen. *Problem :* gemeinsamer Zugriff auf Daten (keine Voraussage über Geschwindigkeit)

2 parallele Operationen mit verteilten Daten $\longrightarrow$ Verlagerung des Problems auf einzelne Synchronisationszeitpunkte. (Gesetz von der Erhaltung der Schwierigkeiten)

Oft repräsentiert jeder Prozeß dasselbe Programm, allerdings mit unterschiedlichen Daten.
$\Longrightarrow$ SPMD als Unterklasse der MIMD

SPMD :
Single Program Multiple Data

- Wird auf größeren Parallelrechnersystemen und vor allem bei massiv parallelen Rechnern verwendet.
- Datenparallelität muß vorhanden sein.
- *Kein* SIMD, da kein Gleichschritt in Anweisungsebene erforderlich. Jedoch müssen im Normalfall in den Synchronisationszeitpunkten Daten angeglichen werden.

2.1.4 Klassifikation nach Speicherzugriff

Shared Memory (konkurrierende Prozesse) :
Speicher, auf den verschiedene Prozesse "gleichzeitig" zugreifen, d.h. sie teilen sich den Speicher.

$\boxed{+}$ Jeder Prozeß kann auf sämtliche Daten zugreifen.
$\Longrightarrow$ Serielles Programm kann ohne größere Schwierigkeiten zum Laufen gebracht werden. Führt in der Regel bei kleinen Prozessorzahlen (2 ... 16) schnell zu einer ersten Leistungssteigerung.

$\boxed{-}$ Anwachsen der Zugriffskonflikte (z.B. Bankkonflikte) bei größeren Prozesszahlen. Somit ist die *Skalierbarkeit* (d.h. Leistung $\sim$ Prozessanzahl) nicht mehr gewährleistet.

$\boxed{-}$ Zur Verringerung der Zugriffskonflikte sind sehr leistungsfähige Bussysteme bzw. Zugriffsverwaltungen notwendig.
$\Longrightarrow$ Verteurung des Gesamtsystems.

Distributed Memory (kommunizierende Prozesse) :
besteht aus Speicherteilen, auf welche im gesamten Programmablauf stets nur jeweils ein Prozeß zugreifen kann. Werden Daten eines anderen Prozesses benötigt kann dies nur über explizite Kommunikation zwischen den Prozessen erfolgen.

$\boxed{+}$ Keine Zugriffskonflikte bei Datenoperationen (Lokalität).

$\boxed{+}$ Relativ preiswerte Hardware (sehr leistungsstarke hat aber auch Ihren Preis)

$\boxed{+}$ Fast beliebig skalierbar.

$\boxed{-}$ Kein direkter Zugriff auf die Daten anderer Prozesse.
$\Longrightarrow$ Nachrichtenaustausch (Kommunikation) über spezielle Kanäle (Links) notwendig.
$\Longrightarrow$ Serielles Programm ist nicht lauffähig, es werden spezielle parallele Algorithmen benötigt.

- Aufwand für Kommunikation noch vor ca. 20 Jahren vernachlässigt, heute aber ist er mitentscheidend für die Qualität eines parallelen Algorithmus (u.a. Verhältnis Kommunikation zu Arithmetik).

- Geschwindigkeit der Verbindungsnetzwerke von großer Bedeutung.

Distributed Shared Memory (DSM) :
(auch *Virtual Shared Memory*) 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").

$\boxed{+}$ 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.

Mittlerweile (Frühjahr 97), hat SGI die Origin2000 mit dem Skalierbaren Symmetrischen Multiprocessing (S2MP) auf den Markt gebracht[1]. 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 Cray[2] (seit 1996 Tochtergesellschaft von SGI).

Bemerkung : Im Rahmen des EUROPORT[3]-Projektes wurden 1995-1997 38 kommerzielle Codes parallelisiert. Bzgl. der Parallelisierbarkeit auf verschiedenen parallelen Programmiermodellen und Parallelrechnern sind

[1] *SGI Performer*, Oct. 96, pp. 4/5
[2] http://www.cray.com
[3] http://www.gmd.de/SCAI/europort/

interessante Vergleiche[4] verfügbar. Insbesondere zeigte sich, daß ein distributed-memory-Konzept zur Datenlokalität in einem shared-memory Programm auf DSM-Hardware sehr erfolgreich eingesetzt werden kann und derart gestaltete Programme den automatisch parallelisierten im allgemeinen überlegen ist.

[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 verteiltem Speicher von Interesse.

Link :
Eine Verbindung zwischen 2 Prozessen.

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

Topologie :
Allgemein eine Vernetzung der Prozesse.

Physikalische Topologie :
Vom Hersteller vorgegebene hardwaremäßige Vernetzung der Recheneinheiten (Knoten)

- Teilweise steckbar (Xplorer, MultiCluster-I).
- Teilweise softwaremäßig konfigurierbar (Crossbar Switches im MultiCluster-II)

Bsp.: Transputer T805 besitzt 4 physische Links.

Logische Topologie [virtuelle Topologie] :
Vom Nutzer gewünschte oder vom Betriebssystem vorgegebene Vernetzung der Prozesse [Knoten]. 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 : Am effizientesten ist natürlich die Identität von logischer und physikalischer Topologie
$\Longrightarrow$ Widerspruch zu Datenstruktur bei vielen Anwendungen.

max. Weglänge :
Max. 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).

2.2.1 Farm / Master-Slave Topologie

Begriff ist vom Aufseherprinzip in den Plantagen des Südens der USA abgeleitet.

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

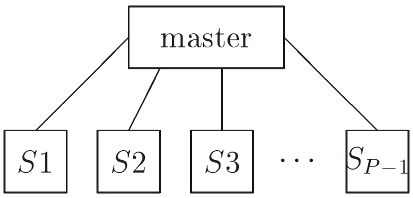


Abbildung 2.2: Master-Slave

Bsp.: Monte-Carlo-Methoden (nicht immer !), Maschinensteuerung, Integralberechnung

2.2.2 Die Pipe

Eindimensionale Anordnung der Knoten.

- 2 Links pro Knoten.
- max. Weglänge: $P - 1$.
- Datenabhängigkeit meist nur in eine Richtung möglich.

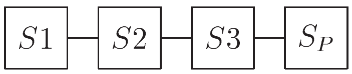


Abbildung 2.3: Pipe

Bsp.:
- Gauß-Algorithmus
- *ein* 1D-Gauß-Seidel-Schritt bei vollbesetzten Systemen
- *mehrere* 1D-Gauß-Seidel-Schritte bei spezieller (FEM/FDM) Matrixstruktur

2.2.3 Der Ring

Erweiterung der Pipe.
- 2 Links pro Knoten.
- max. Weglänge: $P/2$.
- zyklische Datenabhängigkeit realisierbar.

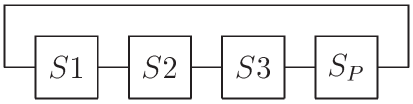


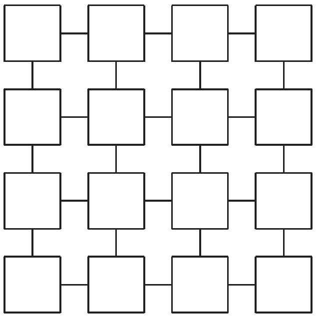
Abbildung 2.4: Ring

Bsp.:
- Gauß-Algorithmus mit anderer Verteilung der Unbekannten als in Pipe.
- *mehrere* 1D-Gauß-Seidel-Schritte auch bei vollbesetzten Matrizen
- Ein- bzw. Auslesen großer Datenmengen zum/vom Parallelrechner über einen, mit dem Hostrechner verbundenen, Knoten (Gesamtdatenmenge übersteigt den verfügbaren lokalen Speicher)

2.2.4 Array und Torus

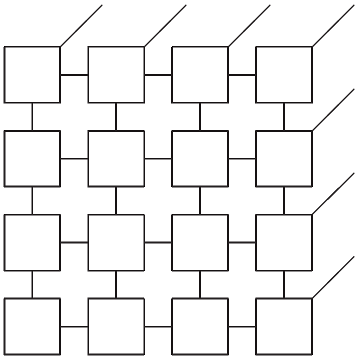
Die logische Weiterenwicklung von Pipe und Ring sind das Array und der Torus welche man als Tensorprodukt von Pipes bzw. Ringen auffassen kann.

2D-Array $P = P_x \cdot P_y$



- 4 Links pro Knoten
- Max. Weglänge: $P_x + P_y - 2$ (= 2 Pipes)
- Hardwaremäßig : MasPar, Parsytec Xplorer

3D-Array $P = P_x \cdot P_y \cdot P_z$



- 6 Links pro Knoten
- Max. Weglänge: $P_x + P_y + P_z - 3$
- Hardwaremäßig : Parsytec Power-GC

Die Bilder der 2D- und 3D-Tori entsprechen obigen Bildern mit der entsprechenden Erweiterung.

2D-Torus

- 4 Links pro Knoten
- Max. Weglänge: $(P_x + P_y)/2$

3D-Torus

- 6 Links pro Knoten
- Max. Weglänge: $(P_x + P_y + P_z)/2$

2.2.5 Der Tree

Der Tree (oder Baum) in Abb. 2.5 ist eine sehr flexible Topologie, mit guten Kommunikationseigenschaften.

Optimaler Baum :
Baum dessen maximale Tiefe und Verästelung (d.h. Verzweigungen pro Knoten) nicht größer als $d = \log_2 P$ ist.

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
- Max. Weglänge: $2d - 1 = 2\log_2 P - 1$

Konstruktionsprinzip

Gray-Code :
Die Binärdarstellungen der Nummern benachbarter Knoten unterscheiden sich in genau einem Bit ($[0, \ldots, d]$) !

Bemerkung : 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.

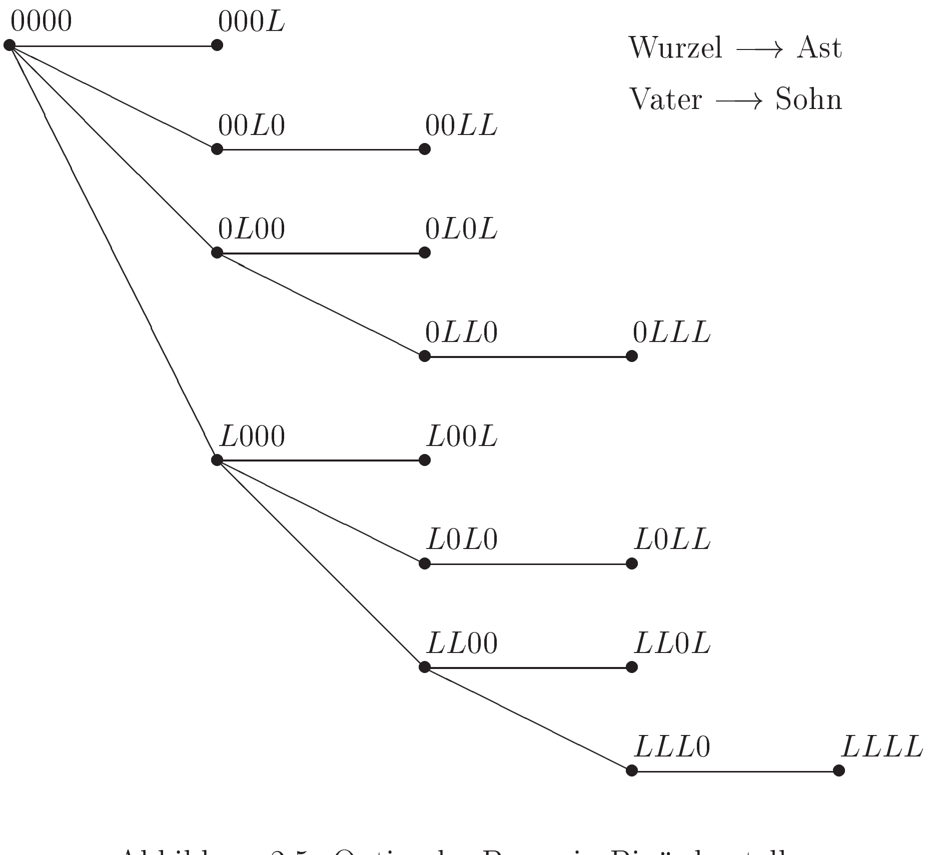


Abbildung 2.5: Optimaler Baum in Binärdarstellung

Hardware: Convex-MPP mit dem Konzept der "Fat Trees", d.h. je näher ein Link an der Wurzel des Baumes, desto leistungsfähiger ist die Datenleitung ausgelegt.

2.2.6 Der Hypercube

Ein d-dimensionaler Hypercube hat genau $P = 2^d$ Knoten.

Rekursiver Aufbau

- Platzierung der Knoten nach dem Gray-Code
- Ein Hypercube der Dimension d ergibt sich durch die entsprechende Verbindung zweier Hypercubes der Dimension $d - 1$.
- Im Hypercube benachbarte Knoten unterscheiden sich in *genau einem* Bit ihrer Knotennummern in Binärdarstellung. Die Position des 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 2.6 in die Subcubes $\{0, 1, 4, 5\}$ und $\{2, 3, 6, 7\}$.

- d Links pro Knoten
- Max. Weglänge: $d = \log_2 P$
- Kürzester Weg im Hypercube zwischen den Knoten p und q ergibt sich aus XOR(p,q) , worin die gesetzten Bits den Linknummern entsprechen, welche die Verbindung bilden.
  Bsp.: $p = 1 = 00L$, $q = 7 = LLL \Longrightarrow$ XOR(1,7) $= LL0$
  Der Weg in obigem Beispiel ist in einen 2-dimensionalen Subcube des Hypercubes eingebettet.
- Hardware: nCube

Die Topologien Ring, Torus und Tree (2.2.2 - 2.2.5) sind im Hypercube eingebettet.
So läßt sich ein Ring mit $P = 2^d$ Knoten ähnlich rekursiv wie der Hypercube

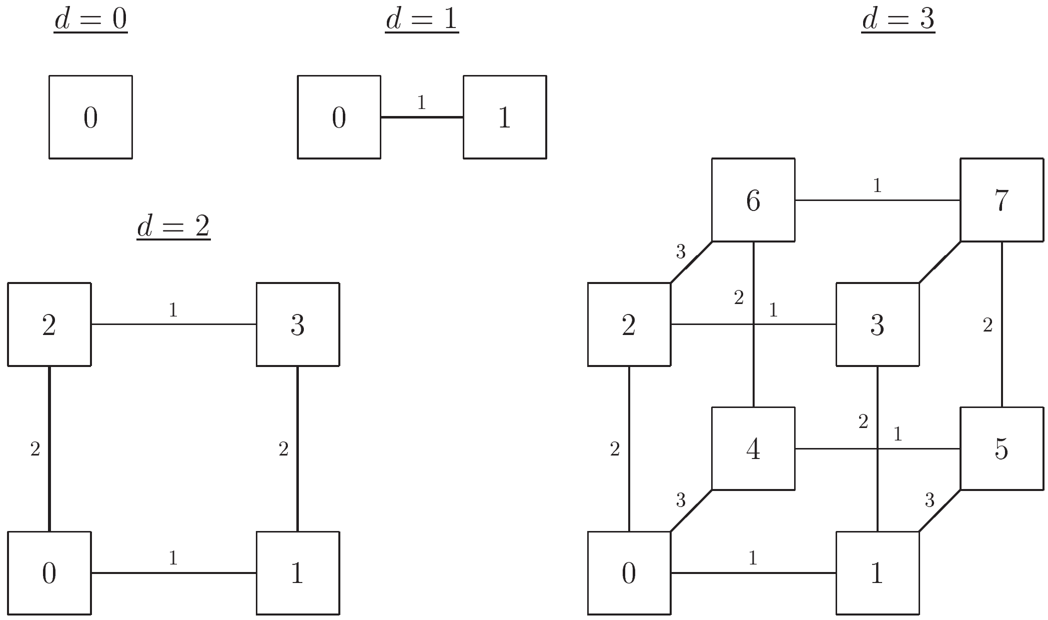
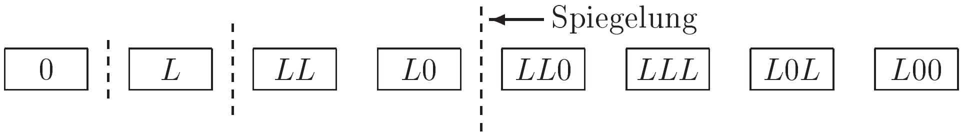


Abbildung 2.6: Hypercube $d = 0, \ldots, 3$

definieren und somit in diesen einbetten:

1. Sei der Ring $P_{d-1} = 2^{d-1}$ bereits im Hypercube eingebettet (d=0 wäre der Induktionsanfang).
2. Die Numerierung der neuen Knoten des Ringes $P_d = 2^d$ erfolgt in gespiegelter Reihenfolge zu den alten Knoten, außerdem wird bei den neuen Knoten das d-te Bit gesetzt.



Algorithmus 2.1: Einbettung des Ringes in Hypercube

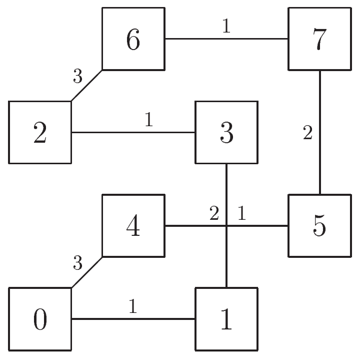


Abbildung 2.7: Einbettung des Ringes im 3D-Hypercube

Aufgabe Ü2.1: Betten Sie einen $2^k \times 2^\ell$-Torus in einen $(k+\ell)$-dimensionalen Hypercube ein.

Aufgabe Ü2.2: Zeigen Sie anhand des Hypercube-3, daß der optimale Baum (Punkt 2.2.5) mit der Wurzel im Knoten 000 im Hypercube eingebettet ist. Wieviele verschiedene Einbettungsmöglichkeiten gibt es ?

Aufgabe Ü2.3: Geben Sie eine Einbettung des optimalen Baums mit der Wurzel im Knoten $L0L$ für den Hypercube-3 an.

Aufgabe Ü2.4: Bilden Sie die Knoten (über ihre Nummern) der beiden Bäume aus den beiden vorangegangenen Aufgaben aufeinander ab.

2.2.7 Das DeBrujin-Netzwerk

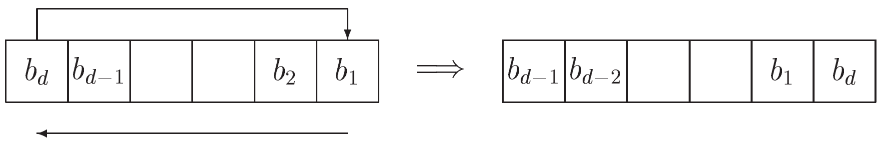
Die bisher betrachteten Topologien besitzen die in Tabelle 2.8 aufgeführten Eigenschaften. Während bei Ring/Torus die Anzahl der Links mit der Raumdimension beschränkt ist (was aber mit einer großen max. Weglänge erkauft werden muß), steigen die pro Knoten benötigten Links bei Tree/Hypercube mit der Anzahl der benötigten Knoten an (was für größere Knotenanzahlen in technologischen Problemen resultiert). Wünschenswert wäre eine Topologie, welche eine max. Weglänge der Ordnung $\mathcal{O}(\log_2(P))$ bei kleiner, konstanter Anzahl der Links pro Knoten besitzt. Das DeBrujin-Netzwerk ist genau ein solches.

Topologie	# Knoten	max. Weglänge	Links pro Knoten
Ring	P	$P/2$	2
Torus	P	$P/2$	$2 *$ Raumdim.
Opt. Baum	$P = 2^d$	$2d - 1$	$1, \ldots, d$
Hypercube	$P = 2^d$	d	d

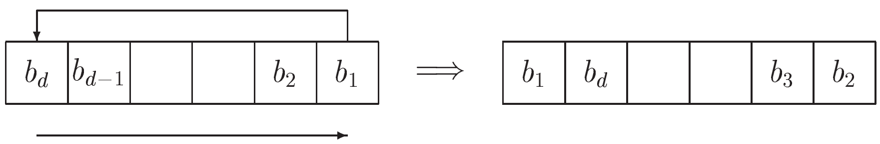
Abbildung 2.8: Eigenschaften Topologien

Zunächst einige Begriffe:

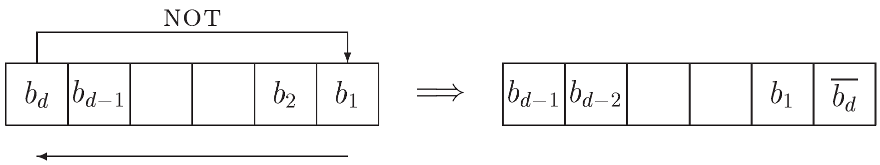
- Left Shuffle [LS] = ISHFTC$(b, 1, d)$ (im weiteren Link 1)



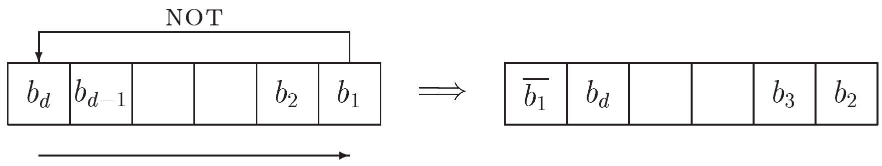
- Right Shuffle [RS] = ISHFTC$(b, -1, d)$ (im weiteren Link 2)



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



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



- Netzwerk der Dimension d besitzt 2^d Knoten und $2^{d+1} - 2$ (gerichtete) Kanten.
- Jeder Knoten (außer Knoten 0 und $2^d - 1$) besitzt genau 4 Links. Die Knoten 0 und $2^d - 1$ benötigen nur 2 Links.
- Nur gerade Knoten im Graphen
  $\Longrightarrow$ Graph ohne Überschneidung durchlaufbar.
  $\Longrightarrow$ Ring ist eingebettet.
- Die max. Weglänge ist d.

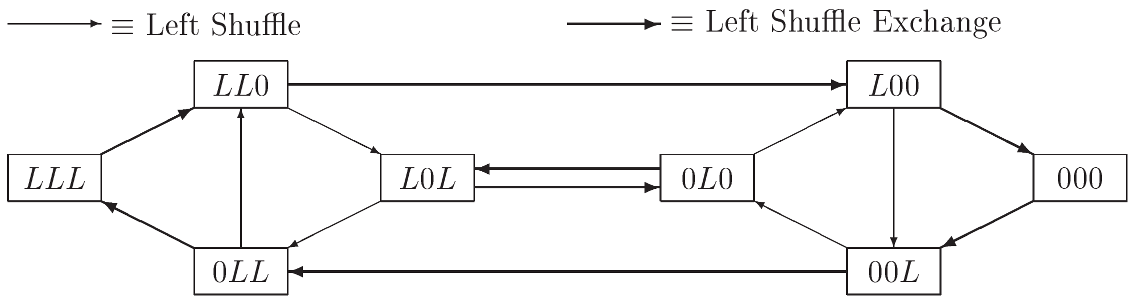


Abbildung 2.9: DeBrujin Netzwerk $d = 3:$ 8 Knoten, 14 Kanten

Es ist keine kommerzielle Hardwarerealisierung des DeBrujin Netzwerks bekannt.

Aufgabe Ü2.5: Für welche Knotennummern $i = 0, \dots, 2^d - 1$ im *de Brujin*-Netzwerk gilt

$$\begin{aligned} \text{RightShuffle}(i) &= \text{LeftShuffle}(i) \quad \text{bzw.} \\ \text{RightShuffleExchange}(i) &= \text{LeftShuffleExchange}(i) \quad ? \end{aligned}$$

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

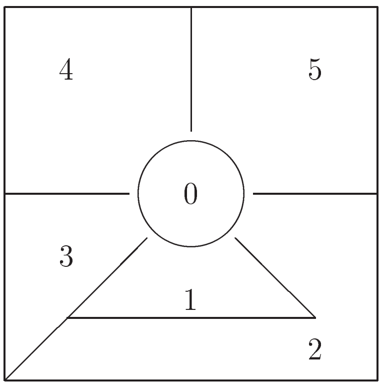
Aufgabe Ü2.6: Stellen Sie ein *de Brujin*-Netzwerk für $d = 4$ auf !

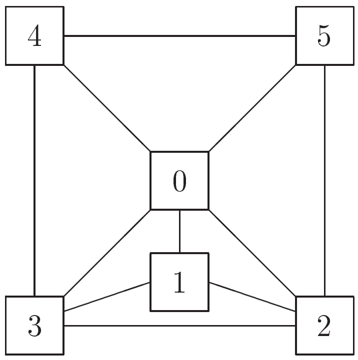
Aufgabe Ü2.7: Beweisen Sie, daß im *de Brujin*-Netzwerk die maximale Weglänge zwischen zwei beliebigen Knoten p, q gleich der Dimension d ist !

2.2.8 Freie Topologie

Grundidee: Abbildung der Beziehungen in der Datenaufteilung (z.B. Gebietsaufteilung) auf das Knotenfeld, d.h. die geometrischen Nachbarschaftsbeziehungen werden durch topologische wiedergespiegelt.

Geometrie *Topologie*





- Bis zu $P - 1$ Links pro Knoten notwendig.
- Max. Weglänge 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.
  $\Longrightarrow$ Kommunikation und Rechnung parallel auf einem Knoten (T805 von Inmos)
- Unterstützung durch paralleles Betriebssystem (PARIX, PVM, MPI)

2.3 Parallele Betriebssystemerweiterungen und Programmiersprachen

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

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

programmiert.
$\Longrightarrow$ Schlechte Portierbarkeit der Quellen.
$\Longrightarrow$ Eigene Bibliotheken, mit sehr wenigen, hardwareabhängigen Rufen (*libCubecom.a*).

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. Falls entsprechende Rechner der Firma Parsytec[5] verfügbar sind, wird im Praktikum auf die OS-Erweiterung PARIX eingegangen.

Mittlerweile wurden parallele Hochsprachen wie High Performance Fortran (HPF[6]), Vienna Fortran[7] u.a. entwickelt. Diese Sprachen enthalten gewisse, die Parallelität unterstützende, Sprachkonstrukte, bzw. versucht der Compiler eine automatische Parallelisierung.

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 MPI[8] mittlerweile von allen größeren Herstellern von Parallelrechnern unterstützt wird (meist sitzen Vertreter im MPI-Konsortium) und mittlerweile auf Workstationclustern (LINUX[9] !), ja sogar unter Windows läuft, gehen wir im Praktikum speziell auf die Programmierung unter MPI ein. Zum Verständnis der Rufe in MPI[10] ist Abschnitt 3.3 unbedingt erforderlich [Fos94, GLS94, Pac97].

[5] http://www.parsytec.de/
[6] http://www.mhpcc.edu/doc/hpf/hpf.html
[7] http://www.vcpc.univie.ac.at/activities/language/vf/ViennaFortran.html
[8] http://www.mcs.anl.gov/mpi/index.html
[9] http://www.suse.de/
[10] http://www.usfca.edu/mpi

Kapitel 3

Parallelität auf algorithmischer Ebene

3.1 Einige Begriffe

Skalierbarkeit :
Einfache Anpassung eines Programmes an eine real verfügbare Anzahl von Prozessoren.

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

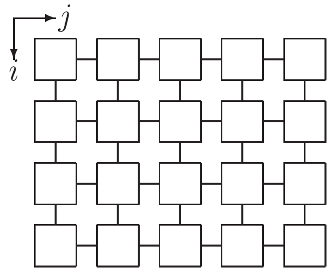
Granularität :
Größe der Programmabschnitte, welche ohne Kommunikation mit anderen Prozessoren ausführbar sind.

Feinkörniger Algorithmus : $\longrightarrow$ Systolische Felder [SIMD]

Bsp.: ω-*Jacobi-Iteration*

Laplace-Problem, mit dem 5-Punkte-Differenzenstern diskretisiert.

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



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

Bsp.: ω-*Gauß-Seidel-Iteration (vorwärts)*

Laplace-Problem, mit dem 5-Punkte-Differenzenstern diskretisiert.

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

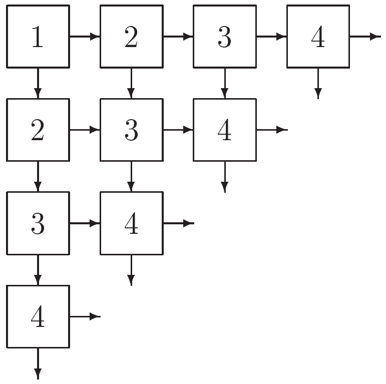


Abbildung 3.1: Gauß-Seidel im systolischen Feld

Die Werte $\widehat{c}_{i-1,j}$ und $\widehat{c}_{i,j-1}$ müssen berechnet sein, bevor $\widehat{c}_{i,j}$ berechnet werden kann.

– Berechnung als Wellenfront {1,2,3,4-ter Schritt}.

– $m \times n$ systolisches Feld
$\longrightarrow$ $(m+n-1)$ Zeitschritte für eine *Iteration* nötig.
$\longrightarrow$ mehrere Iterationen kompensieren diesen Overhead beim Startup.

1. Iteration :
Zeitgewinn : $\frac{t_{parallel}}{t_{seriell}} = \frac{m+n-1}{mn} \approx \frac{m+n}{mn} \geq \frac{2\sqrt{mn}}{mn} = \frac{2}{\sqrt{mn}}$
Ideal wäre aber ein Verhältnis $\frac{1}{P} = \frac{1}{mn}$!

2. Iteration :

– Nach Initialisierung benötigt eine Iteration nur noch *einen* Zeitschritt.

– k-te (letzte) Iteration benötigt gleiche Zeit wie Initialisierung

$$\begin{aligned}\frac{t_{parallel}}{t_{seriell}} &= \frac{2(m+n-1)+k-2}{k \cdot mn} = \frac{2(m+n-1)}{k \cdot mn} + \frac{k}{k \cdot mn} - \frac{2}{k \cdot mn} \\ &\geq \frac{4\sqrt{mn}}{k \cdot mn} + \frac{1}{mn} - \frac{2}{k \cdot mn} \stackrel{k \mapsto \infty}{\longrightarrow} \frac{1}{mn}\end{aligned}$$

Asymptotisch, d.h. bei vielen Iterationen wird der bestmögliche Zeitgewinn durch die Parallelisierung erzielt.

Grobkörniger Algorithmus : $\longrightarrow$ Mehrprozessorrechner [MIMD]
z.B. Blockvarianten von ω-Jacobi- bzw. Gauß-Seidel-Iteration

Bemerkung : 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 gefragt.

Funktionale Parallelität :
Aufspaltung eines Algorithmus in funktional unterschiedliche, parallel abarbeitbare Teilschritte.

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

- seriell (von links nach rechts) 7 Zeitschritte
- parallel 2 Prozessoren 5 Zeitschritte

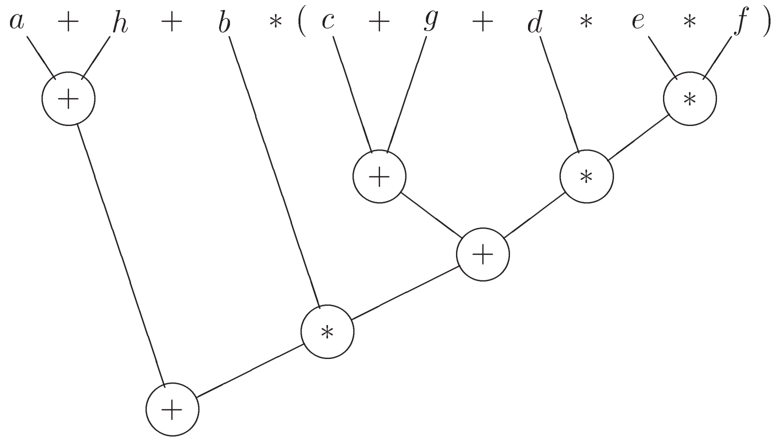


Abbildung 3.2: Funktionale Parallelität

Aufgabe Ü3.8: Teilen Sie obige Berechnungsvorschrift auf 3 Prozessoren auf. Nutzen Sie Umformungsregeln. (Ergebnis: 4 Zeitschritte)

- Im parallelen Fall können in Summe mehr arithmetische Operationen auftreten, trotzdem ist die parallele Variante (theoretisch) schneller.
- Funktionale Parallelität ist nur bedingt skalierbar. In obigem Beispiel ergeben mehr als 3 Prozessoren keine weitere Zeitersparnis.

Datenparallelität :
Gleiche Programmabschnitte werden parallel über verschiedenen Datensegmenten ausgeführt.

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

Bsp.: *Block-ω-Jacobi*

  - Aufteilung der Vektoren und (Block-)Matrizen.
  - Mehrere Möglichkeiten der Datenaufteilung (siehe Abschnitt 5)

Spezialfall : Geometrische Parallelität.

Die Aufspaltung der Daten in Teilmengen entsprechend geometrischer Überlegungen über der räumlichen Anordnung der Objekte (Teilchen, Diskretisierungspunkte etc.), bzw. Ausnutzung von Nachbarschaftsbeziehungen bei Diskretisierungsmethoden wie FEM, FDM, FVM.

3.2 Synchronisation paralleler Prozesse

3.2.1 Einige Begriffe

Bemerkung : 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")
  $\Longrightarrow$ absolut parallel ($\Longrightarrow$ für uns uninteressant)

- gemeinsame Nutzung (knapper) Ressourcen
  $\Longrightarrow$ Multiprogrammbetrieb/Time-Sharing
  $\Longrightarrow$ Verteilungsstrategie notwendig

- Mehrprozessorsysteme mit verteilten Aufgaben einer Gesamtaufgabe
  $\Longrightarrow$ Informationsaustausch erfolgt über **shared memory** oder **message passing** (distributed memory).

Synchronisation :
Verhinderung undefinierter Zustände durch "Abstimmung" der Prozesse untereinander zu bestimmten kritischen Zeitpunkten.

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.

Bsp.: *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

Abbildung 3.3: Undefinierter Zustand

Das Resultat (d.h. der Wert von N) hängt von der Geschwindigkeit der Prozesse A und B ab und ist somit undefiniert.

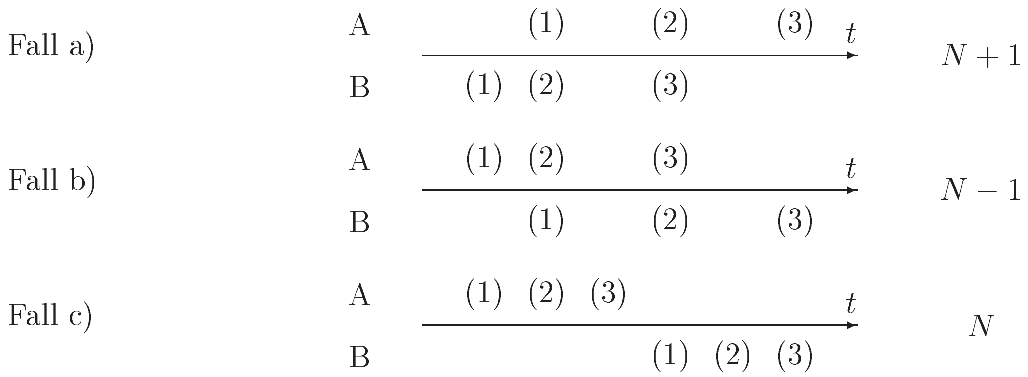


Abbildung 3.4: Ergebnisse im undefinierten Zustand

Um obigen undefinierten Zustand zu vermeiden wäre es sinnvoll, die Operationen A und B als nicht teilbar zu betrachten.
$\Longrightarrow$ Exklusiver Zugriff auf N für einen Prozeß für die erforderliche Zeit.
Ein Konzept für diesen exklusiven Zugriff wird im nächsten Abschnitt vorgestellt.

3.2.2 Semaphoren

Ein generelles Problem beim Zugriff auf gemeinsame Ressourcen ist die Deadlock-Freiheit.

Deadlock :
Verklemmung, *mehrere* Prozesse warten auf je ein Ereignis welches aber nur durch einen der wartenden Prozesse ausgelöst werden kann.

Bsp.: *Dinner for five [Dijkstras oder Hoare, 1971]*

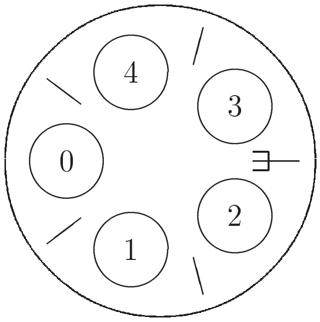


Abbildung 3.5: Dinner for five

> "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" [Rec94]
>
> Die 5 Gabeln stehen hier stellvertretend für knappe Resourcen in einem großen System.

Lösungsvariante (Handlungsanweisung) 1:

1. Wenn Du hungrig bist, setzte 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 !

$\Longrightarrow$ **Deadlock** :

Alle Philosophen verhungern mit einer Gabel in ihrer linken Hand.

Lösungsvariante 2:

1. Wenn Du hungrig bist, setzte 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 !

$\Longrightarrow$ "Deadlock für alle" ist ausgeschlossen, aber dem Einzelnen kann der Hungertod drohen (z.B. für Philosoph Nr.1, wenn sich Nr.0 und Nr.2 leicht überlappend beim Essen abwechseln).

Dijkstras Vorschlag : Einführung von Semaphoren (Semaphore ist ein Signal bei der holländischen Eisenbahn).

Semaphor :
Signal welches von einzelnen Prozessen betätigt wird und nicht von einer Zentralinstanz (Abb. 3.6).

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].

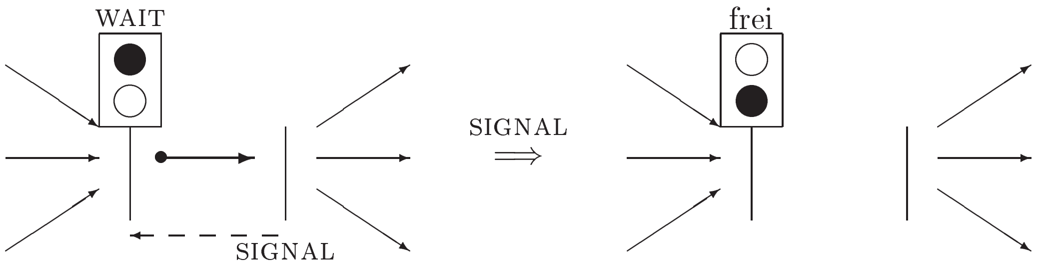


Abbildung 3.6: Freigabe einer Semaphore

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

Semaphorenkonzept

Die Semaphore kann als strukturierter Datentyp aufgefaßt werden mit den

Elementen:
- $\mathbb{S}$-Wert (Integer)
- Warteschlange (queue)

und Operationen (• init):
- $\mathbb{P}$roberen $\equiv$ WAIT
- $\mathbb{V}$erhogen $\equiv$ SIGNAL

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

Die Operationen sind mit den Elementen wie folgt verknüpft :

$\mathbb{P}$(s):	$\mathbb{S}$-Wert = 0	$\longrightarrow$	Prozeß in Queue [Zug auf Strecke]
	$\mathbb{S}$-Wert > 0	$\longrightarrow$	($\mathbb{S}$-Wert) := 0 und Resourcen ist verfügbar [Strecke ist frei, Einfahren und Signal auf "Rot" setzen]
$\mathbb{V}$(s)	queue ist leer	$\longrightarrow$	($\mathbb{S}$-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.

Beispiele für Semaphoren-Anwendungen

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

  $\mathbb{S}$ Semaphore $\longleftarrow$ INIT($\mathbb{S}$) ist notwendig, sonst Deadlock

  $\mathbb{R}$ Resourcen

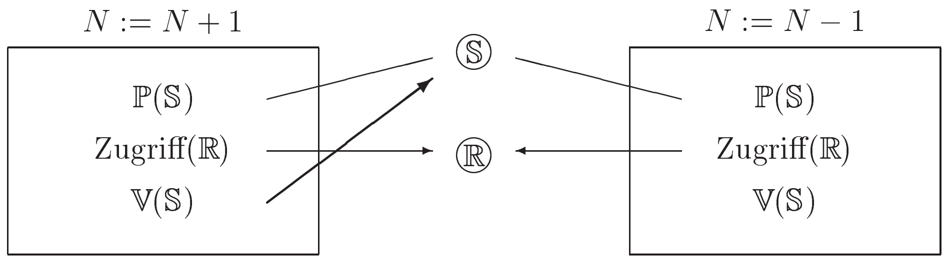


Abbildung 3.7: Exklusiver Zugriff mittels Semaphoren

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

- Eigenschaftssynchronisation (z.B. Produzent-Konsument)

  2 Semaphoren — A $\longrightarrow$ signalisiert vollen Puffer; INIT(A) mit 0
B $\longrightarrow$ signalisiert leeren Puffer; INIT(A) mit 1

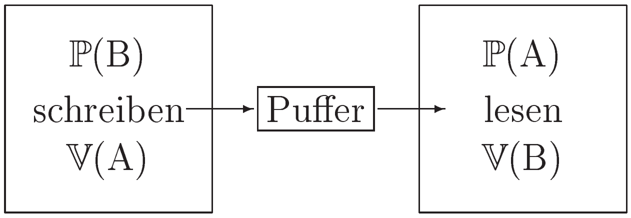


Abbildung 3.8: Eigenschaftssynchronisation mittels Semaphoren

Tritt auf bei der ω-Gauß-Seidel-Iteration (vorwärts) für das Laplace-Problem, mit dem 5-Punkte-Differenzenstern diskretisiert.

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

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

Obige Procedure 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 fängt an zu Essen.

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: Setzen und falls möglich 2 Gabeln nehmen.

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.

Message Passing

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

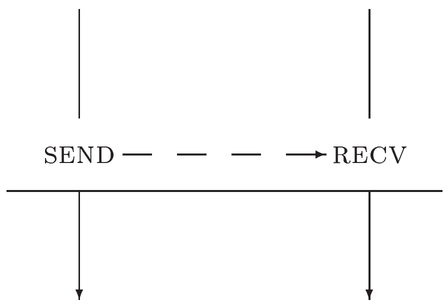


Abbildung 3.9: Blockierende Kommunikation

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

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

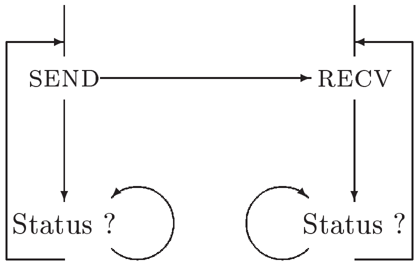


Abbildung 3.10: Nichtblockierende Kommunikation

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

- 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 Resourcenmanagement erledigt, tritt das Message Passing bei Parallelrechnern mit verteiltem Speicher (Distributed Memory) auf und muß zumindest teilweise vom Programmierer selbst erledigt werden.

Monitorkonzept

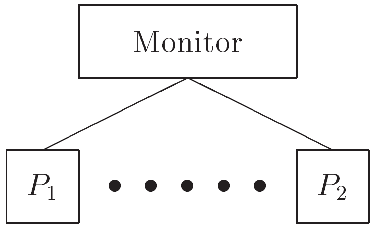


Abbildung 3.11: Monitorkonzept

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

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

Unterschiede zwischen Semaphoren und Monitorkonzept :

Semaphoren : Prozesse "unterhalten sich miteinander".
Monitor : Zentrale Instanz nimmt Aufträge der Prozesse entgegen (Bsp.: Pizzabäcker).

3.2.3 Datenkohärenz

Datenkohärenz :
Kopien eines Datensatzes enthalten stets die gleichen Daten.

- Bei distributed-memory Maschinen ist der Programmierer selbst für die Datenkohärenz verantwortlich, siehe Abschnitte 4 und 5.
- 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 Kommunikationsbandweite.

directory-based protocol: Jeder Speicherblock (cache-line) besitzt einen Eintrag in einem Verzeichnis (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 [SGI97].

3.3 Grundlegende globale Operationen

Die in diesem Abschnitt betrachteten Operationen auf Daten werden im Hypercube (Abschnitt 2.2.6) und den in ihm eingebetteten Topologie betrachtet.

Vor.: Es seien die Routinen

$$\text{SEND\_CHAN}(nwords,\ data,\ LinkNo)$$
$$\text{RECV\_CHAN}(nwords,\ data,\ LinkNo)$$

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

Die Bezeichnungen der behandelten Funktionen orientieren sich dabei an einem konkreten (eigenen) Bibliothekspaket, sind aber ansonsten frei wählbar.

3.3.1 Exchange

$$\text{EXCHANGE}(LinkNo,\ nWords,\ SendData,\ RecvData)$$

Austausch von 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.
- Synchrone Kommunikation führt zu einem Deadlock.

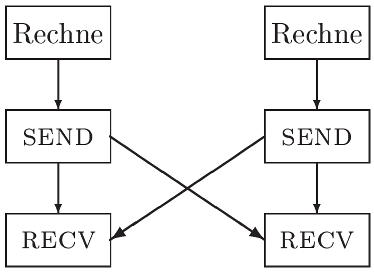


Abbildung 3.12: Exchange non-blocking

Variante b)

Verklemmungsfreies Exchange zwischen p und q bei blockierender Kommunikation erfordert eine eindeutige Vorschrift, welcher von beiden Prozessen zuerst senden bzw. empfangen darf. Die Funktion $\text{BTEST}(k, link) =$

$\text{XOR}(k, 2^{link-1})$ 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.

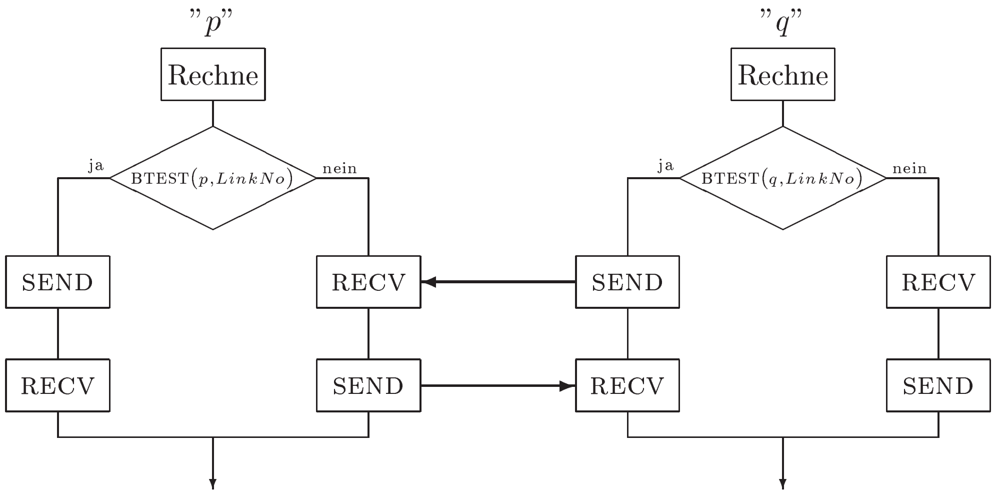


Abbildung 3.13: EXCHANGE blockierend

3.3.2 Gather-Scatter-Operationen

Sammeln (engl.: gather) bzw. Verteilen (engl.: scatter) von Daten durch ausgezeichneten Prozeß, im weiteren *Root-Prozeß* genannt.

- Sei der Rootprozeß der Prozeß 0.
- Optimaler Baum ist im Hypercube eingebettet (Abschnitt 2.2.6)
- Scatter $\Longrightarrow$ TREE_DOWN($nWords$,$Data$)
- Gather $\Longrightarrow$ 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
```

Algorithmus 3.1: Allg. TREE_DOWN(*nWords*,*Data*)

E1: 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 Ü2.3 und Ü2.4.

3.3.4 Reduce- und All-Reduce Operationen

Bildung von Ergebnissen über Daten aller Prozesse (z.B. $s = \sum_{i=0}^{p-1} a_i$). Das Ergebnis steht am Ende einem Root-Prozeß (bei Reduce) bzw. allen Prozessen (bei All-Reduce) zur Verfügung.

- *Reduce :* Es wird die Tree-Topologie mit Root = 000 verwendet.

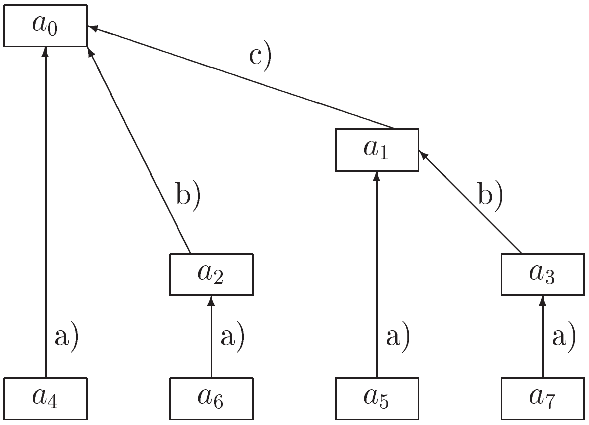


Abbildung 3.14: Reduce-Operation im binären Baum

Abfolge a), b), c): a) = Link 3, b) = Link 2, c) = Link 1
$s_4 := a_4,\ s_6 := a_6,\ s_5 := a_5,\ s_7 := a_7$
$s_2 := (a_2 + a_6),\ s_3 := (a_3 + a_7)$
$s_1 := [(a_1 + a_5) + (a_3 + a_7)]$
$s_0 := [(a_0 + a_4) + (a_2 + a_6)] \ + \ [(a_1 + a_5) \ + \ (a_3 + a_7)] \ = \ \sum_{i=0}^{p-1} a_i$
Somit besitzt am Ende nur Prozeß 000 das richtige Ergebnis, alle anderen Prozesse verfügen nur über Teilergebnisse.

Programmrealisierung über die Function TREE_UPDO(n,X,Y,H,$VtOp$), welche auch mit Vektoren arbeitet und Operation auf diesen in $VtOp$ spezifiziert (Hier mittels `CALL Tree_UpDo(1,S,A,AuxArray,VdPlus)`).

- *All-Reduce*

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

$$\left.\begin{array}{l}\text{TREE\_UPDO}\\ \text{TREE\_DOWN}\end{array}\right\} \ \text{TREE\_DO}\,{}^{I}_{\substack{D\\R}}$$

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.

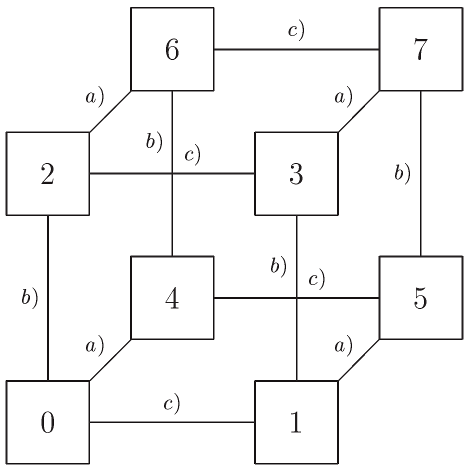


Abbildung 3.15: All-Reduce im Hypercube

Zeitliche Abfolge: a) b) c), jeweils den Links 3, 2 und 1 zugeordnet.

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

$\vdots$

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

$\Downarrow$

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

Programmrealisierung über die Function $\textsc{Cube\_Do}\,{}^{I}_{\substack{D\\R}}(n,X,Y,H,VtOp)$.

Vergleich der Varianten a) und b)

	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

Bsp.: *Links im* POWER-GC

Im POWER-GC der Firma PARSYTEC (ab 64 Prozessoren lieferbar) sind folgende Relationen bzgl. der Bandbreite 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 **Bisektionsbandweite** :
Bandweite, wenn die eine Hälfte der Prozessoren mit der anderen Hälfte kommuniziert.

In obigem Beispiel ist diese Bisektionsbandweite anisotrop, d.h. sie hängt davon ab welche Teile der Maschine miteinander kommunizieren. In der Origin2000 tritt dieses Problem ab 128 Prozessoren auf ([SGI97], Seite 34ff).
Bei einem idealen System wächst die Bisektionsbandweite mit der Anzahl der Prozessoren.

Aufgabe Ü3.9: Ändern Sie den Algorithmus für TREE_DOWN auf Seite 43 so um, daß die Links mit höheren Nummern möglichst wenige Daten transportieren müssen.

3.3.5 Barrier

Synchronisation aller Prozesse.

- 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.

3.3.6 Bemerkungen zu portablem Code

Die Operationen in 3.3.1 - 3.3.5 bauten auf den Voraussetzungen

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

auf. Nur diese 3 Routinen sind Hardware- und Betriebssystemabhängig. Alle anderen Routine sollten auf diesen 3 Grundroutinen aufbauen !
$\Rightarrow$ Portabler Code
Bsp.: Von uns genutzte Bibliothek *libCubecom.a* unter MPI, PVM, Parix, nCube, Paragon, Standalone(3L), Helios, ... verfügbar.

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

Speedup :
Sei t_1 - Ausführungszeit des Algorithmus A auf einem Prozessor
t_P - Ausführungszeit des Algorithmus A auf P Prozessoren

$$\boxed{S_P = \frac{t_1}{t_P}} \tag{3.1}$$

Wünschenswert ist ein Speedup von P, d.h. P Prozessoren sind P-mal schneller als ein einzelner.

Die Größe des zu lösenden Problems N_{ges} bleibt konstant.

$N_{ges} = \text{const.} \quad \Longrightarrow \quad N_i \sim \dfrac{1}{P} \;\; \forall i = \overline{1, P}$,
d.h. je mehr Prozessoren verwendet werden, desto weniger hat der einzelne Prozessor zu tun.
Bsp.: *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 ?
> $\longrightarrow$ Die Arbeiter behindern sich gegenseitig, ein zu hoher Organisationsaufwand [Overhead] ist notwendig.
> $\Longrightarrow$ sehr geringer Speedup.

AMDAHL*sches Gesetz :*

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

Seien s - serieller Anteil des Algorithmus
p - paralleler Anteil des Algorithmus normiert :

$$\boxed{s + p \;=\; 1} \tag{3.2}$$

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

Daraus folgt eine obere Schranke für den maximal erreichbaren Speedup.

$$\boxed{S_{P,max} \;=\; \frac{s+p}{s+\frac{p}{P}} \;=\; \frac{1}{s+\frac{1-s}{P}} \;\leq\; \frac{1}{s}} \tag{3.3}$$

Bemerkung : Obiger Satz geht auf eine Bemerkung von Gene Amdahl[1] im Jahre 1967 zurück

Interpretation des AMDAHL*schen Gesetzes* (sehr alte Reklame von IBM)

1% serieller Anteil im Algorithmus $\Longrightarrow S_{\infty,max} = 100$

P	10	100	1000	10000
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).

$\Longrightarrow$ Entmutigend

$\Longrightarrow$ Vorlesungsende ?

Ist das AMDAHLsche Gesetz für uns 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 Ordnung des arithmetischen Aufwands des lokalen Algorithmus mit weniger Daten auftreten.

- Da das AMDAHLsche Gesetz keine Kommunikationszeiten berücksichtigt, sieht das reale Verhalten des Speedup noch schlechter aus, siehe Abbildung 3.16.

[1] http://www.amdahl.com

Abbildung 3.16: Realer Speedup

Folgerung : Der erreichbare Speedup hängt von der Problemgröße ab.

zu Bsp.: 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.

Scaleup :
Welcher Zuwachs an Problemgröße wird auf dem Parallelrechner im Vergleich zum sequentiellen Fall bei vergleichbarer Rechenzeit erzielt.

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

$$\boxed{S_C(P) \;=\; \frac{s_1 + P \cdot p_1}{s_1 + p_1} \overset{s_1+p_1=1}{=} s_1 + P(1 - s_1)} \tag{3.4}$$

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

Bemerkungen zum Speedup/Scaleup

- (Scaled) Speedup darf nicht einziges Bewertungskriterium sein, z.B. ist in Abbildung 3.17 ersichtlich, daß

$$
\begin{array}{ll}
 & S_P(\text{Gauß Elimination}) > S_P \text{ (pcg=vorkonditionierter cg)} \\
ABER & t(\text{Gauß Elimination}) >> t(\text{pcg}) \text{ !!}
\end{array}
$$

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

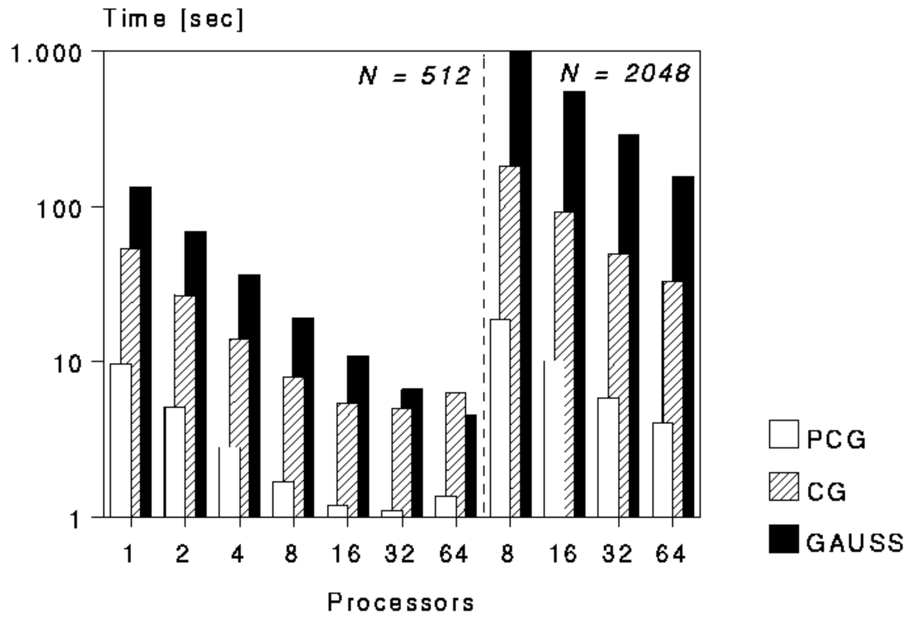


Abbildung 3.17: Rechenzeitvergleich Gauß, cg, pcg [Dr. Pester, Chemnitz]

3.4.2 Effizienz

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

$$
\boxed{E_{P,par} = \frac{S_P}{P}} \tag{3.5}
$$

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 :

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

$\Longrightarrow$ Ganz leicht sinkende Effizienz bei wachsender Prozessorzahl.
$\Longrightarrow$ Effizienz jedoch mindestens so hoch wie paralleler Anteil im Programm.
$\Longrightarrow$ Vorlesung hat doch einen Sinn.

Numerische Effizienz :
Setzt den schnellsten seriellen Algorithmus mit dem schnellsten parallelen Algorithmus (ausgeführt auf einem Prozessor) ins Verhältnis.

$$\boxed{E_{num} \;=\; \frac{t_{parallel}}{t_{seriell}}} \tag{3.6}$$

Somit ergibt sich die (skalierte) Effizienz eines parallelen Algorithmus

$$\boxed{E \;=\; E_{par} \cdot E_{num}} \tag{3.7}$$

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.
$\Longrightarrow$ Praktische Effizienz liegt unter der theoretischen.

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 [TK96].

$\longrightarrow$ Je besser die Datenverteilung (auch in Bezug auf die Kommunikation), desto mehr Aufwand muß bei den einzelnen Bisektionstechniken betrieben werden.
$\longrightarrow$ Besonders geeignet für Optimierungsprobleme, Verteilung der Grobnetzes in Mehrgitteralgorithmen.

- Dynamisches Loadbalancing [Umverteilung während der Rechnung] [Bas96] :
  - Hoher Aufwand für Lastverteilung (Heuristische Vorgehensweise notwendig).
  - 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} - Startupzeit (Latency)
t_{Word} - Zeit für die Übertragung eines Wortes (Bandweite)
so ergibt sich die Zeit für eine Kommunikation mit N Byte Information :

$$\boxed{t_k \ = \ t_{\text{Startup}} + N \cdot t_{\text{Word}}} \tag{3.8}$$

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 ($> 1kByte$).
Die erreichte Bisektionsbandweite einer Origin2000 wächst linear bis zu zu 10 GB/sec auf 32 Prozessoren, verbleibt bei diesem Wert auf 64 Prozessoren und verdoppelt sich bei 128 Prozessoren.

Maschine/OS	Latency	Bandbreite
Xplorer/Parix	100 μs	(gemessen) 1.3 MB/sec (peak) 8.8 MB/sec
T9000	10 μs	(progn.) 80 MB/sec
Intel iPsc/860	82.5 μs	2.5 MB/sec
Convex SPP-1	0.2 μs	16.0 MB/sec
nCube2		71.0 MB/sec
Cray TT3D	6 μs	120.0 MB/sec
IntelParagon		200.0 MB/sec
Origin2000		(real) 680.0 MB/sec

Tabelle 3.1: Vergleich Latency und Bandbreite

Es gibt Algorithmen die nicht unbedingt parallelisierungswürdig sind, obwohl sie so aussehen. Bsp.: *FFT-feinkörnig*

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

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 $t_{\text{seriell}} > t_{\text{par}}$?

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

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

$\longrightarrow$ Grobkörnige Algorithmen bevorzugen (siehe Abschnitt (6.4)).

Kapitel 4

Grundroutinen numerischer Algorithmen

4.1 Die BLAS-Bibliotheken

Häufig verwendete Routinen bzgl. Vektor- und Matrixoperationen sind in den **B**asic **L**inear **A**lgebra **S**ubroutines 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 **S**ingle, **D**ouble und **C**omplex 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 $\quad s = \sum\limits_{i=1}^{N} x_i * y_i .$

3. Triaden : $x_i := x_i + \alpha * y_i$ SAXPY/DAXPY **S**calar **A**lpha times **X** **P**lus **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 $i = 1, n$
  $\quad s := s + \underbrace{x_i * y_i}_{vektorisierbar}$
  END DO

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

- *vektorisiert/sequentiell :*

  DO $i = 1, n$
  $\quad d_i := x_i * y_i$
  END DO
  $s := 0$
  DO $i = 1, n$
  $\quad s := s + d_i$
  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 Doppelns arbeiten.

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

  DO $i = 1, n$
  $\quad d_i := x_i * y_i$
  END DO
  DO $\ell = 1, \log_2(n)$
  $\quad is := 2^\ell$
  $\quad$ DO $i = is, n, is$
  $\quad\quad d_i := d_i + d_{i-is/2}$
  $\quad$ 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 Abbildung 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.

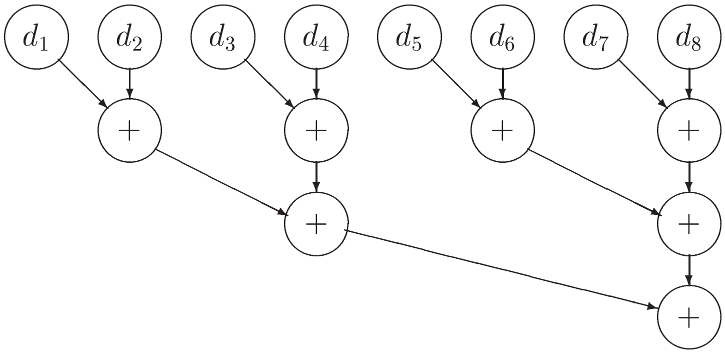


Abbildung 4.1: Illustration rekursives Doppeln $n = 8$

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 $({\scriptstyle j=\overline{1,P}})$ 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 Ü4.10: Programmieren Sie das globale Skalarprodukt zweier disjunkt verteilter Vektoren auf einem Parallelrechner mit verteiltem Speicher.

Bemerkung : Eigene Bibliothek *libvbasmod.a*, welche als Untermenge BLAS1-Funktionalität enthält.

1. Realisierungsebene :

Hochsprache C, F77 und Loop unrolling, dh. das Verhältnis Arithmetik zu Schleifenverwaltung wird verbessert. Betrachten das Skalarprodukt mit Stride 1.

$s = 0$
DO $i = 1, N$
    $s = s + x_i * y_i$
END DO
	$\longrightarrow$	$m = N \bmod 4$
		$s = 0$
		DO $i = 1, m$
		    $s = s + x_i * y_i$
		END DO
		DO $i = m+1, N, 4$
		    $s = s + x_i * y_i + x_{i+1} * y_{i+1}$
		        $+ x_{i+2} * y_{i+2} + x_{i+3} * y_{i+3}$
		END DO

Die Wahl des Moduloparameters (hier 4) ist hardwareabhängig.

2. Realisierungsebene :

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

```
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 Ü4.11: Stellen Sie die Operation $\underline{\mathsf{v}} = A_{n\times n}\underline{\mathsf{x}}$ mittels BLAS-Routinen (DDOT, DAXPY) für die Speicherformen i)-iii) dar.
$*$ 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 $\underline{\mathsf{v}} = A_{n\times n}\underline{\mathsf{x}}$ auf dem Vektorrechner.

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

$$
\begin{aligned}
v_1 &= b_1 * x_1 + c_1 * x_2 \\
v_n &= b_n * x_n + a_{n-1} * x_{n-1}
\end{aligned}
$$

DO $i = 2, n-1$

$\qquad y_i = a_{i-1} * x_{i-1} + b_i * x_i + c_i * x_{i+1}$

END DO

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

$$
\begin{aligned}
x_0 &= x_{n+1} = 0 \\
a_0 &= c_n = 0
\end{aligned}
$$

DO $i = 1, n$

$\qquad y_i = a_{i-1} * x_{i-1} + b_i * x_i + c_i * x_{i+1}$

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 : Verteilen A blockzeilenweise auf die Prozesse, analog die Teilvektoren.

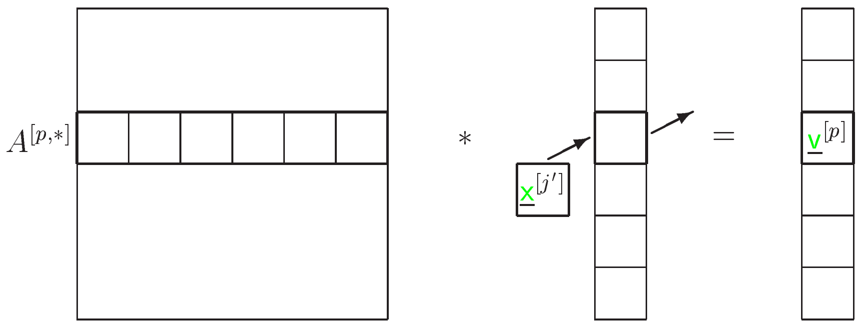


Abbildung 4.2: Matrix blockzeilenweise verteilt

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 $\underline{x}^{[p]}$.

Algorithmus 4.1: Parallele Matrix-Vektor Multiplikation, blockzeilenweise

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.

Variante 2 : Verteilen A blockspaltenweise auf die Prozesse, Vektor $\underline{x}$ entsprechend der Spalten von A.

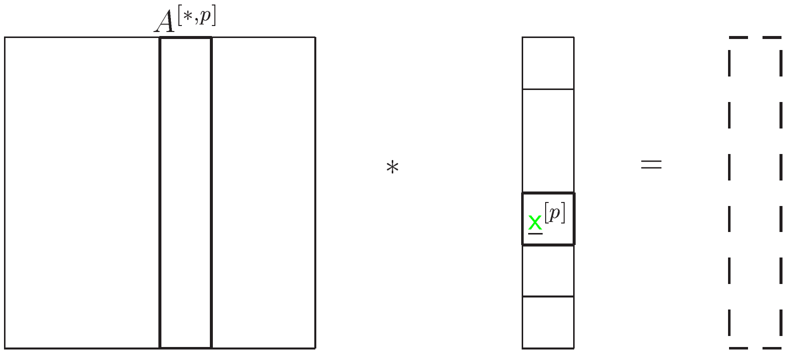


Abbildung 4.3: Matrix blockspaltenweise verteilt

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 $\underline{v}^{[p]}$.

Algorithmus 4.2: Parallele Matrix-Vektor Mult., blockspaltenweise

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

4.1.3 Matrix-Matrix-Operationen (BLAS3)

Seien A, B, C vollbesetzte Matrizen.
Typische Operationen sind :
- $C := A + B \qquad C := A - B$
- $C := A * B$
- $C := C + A * B$
- A^T, $A^{-1}_{N\times N}$
- Faktorisierungen, Givensrotationen, ...

Matrix-Multiplikation

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

```
DO ...... := 1, ......
    DO ...... := 1, ......
        DO ...... := 1, ......
            c_{i,j} := c_{i,j} + a_{i,k} · b_{k,j}
        END DO
    END DO
END DO
```

Algorithmus 4.3: Grundform der Matrix-Matrix Multiplikation

$\Longrightarrow$ 6 verschiedene Algorithmen zur Matrixmultiplikation mit den Unterschieden

- Zeilen- bzw. Spaltenzugriff auf die Matrixelemente
- Nutzbare Grundroutinen, z.B. Skalarprodukt, Triade

$\Longrightarrow$ Konkrete Implementierung stark hardwareabhängig :

- Speicherbankkonflikte :
  Nach jedem Lese/Schreibzugriff auf ein Element einer Speicherbank wird diese Bank für eine gewisse Zeit für Zugriffe gesperrt.

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

Speicherung a)

Bank 1	Bank 2	Bank 3	Bank 4
a_1 a_5 a_9	a_2 a_6 a_{10}	a_3 a_7 a_{11}	a_4 a_8 a_{12}

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

Die Strides mit Konflikten hängen von der Anzahl der Bänke und der Sperrzeit ab.

Speicherung b)

Bank 1	Bank 2	Bank 3	Bank 4
a_1 a_2 a_3	a_4 a_5 a_6	a_7 a_8 a_9	a_{10} a_{11} a_{12}

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	$\Longrightarrow$ Konflikte
LOAD($\underline{a}$)	Stride 4,5,6, ...	$\Longrightarrow$ keine Konflikte

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

- Page fault :
  Die angeforderte Adresse/Speicherzelle befindet sich nicht im schnellsten Speicher, dh. dieser muß nachgeladen werden.

  Muß bei Prozessoren mit Cache und bei virtuellem Speicher beachtet werden.

- Auf Parallelrechnern mit verteiltem Speicher soll natürlich möglichst wenig Kommunikation erfolgen.

- 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 $i := 1, n_3$
   DO $j := 1, n_1$
   $c_{i,j} := c_{i,j} + \langle A_{i,*}, B_{*,j}\rangle$
   END DO
   END DO

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

2. *middle product*

   DO $j := 1, n_1$
   DO $k := 1, n_2$
   $C_{*,j} := C_{*,j} + A_{*,k} * B_{k,j}$
   END DO
   END DO

   - $B_{*,j}$ fungiert in innerster Schleife als Skalar, die anderen Größen als Vektoren $\Longrightarrow$ Triade
   - $\Longrightarrow$ Effiziente Nutzung der Vektorregister, Ausnutzung 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 , n_2
      DO j := 1 , n_1
            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
            C_{i,*} := C_{i,*} + A_{i,k} * B_{k,*}
      END DO
END DO
```

Dieser Algorithmus geht auf Fox zurück [Fox88, KGGK94].
Bemerkung : Am Ende gilt wieder $\widetilde{B}_{i,j} \;=\; B_{i,j}$.

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

Algorithmus 4.4: broadcast-multiply-roll Algorithmus

4.2 Operationen mit dünnbesetzten Matrizen

Zum Sprachgebrauch:

- engl. *dense* $\widehat{=}$ dt. regelmäßig dünnbesetzt (tridiagonal, ...)
- engl. *sparse* $\widehat{=}$ dt. unregelmäßig dünnbesetzt (FEM-Matrizen, ...)

In der Vorlesung steht "dünnbesetzt" stehts im Sinne von *sparse*.
Vorsicht, bei Softwareanbietern wird der Begriff "dünnbesetzt" zu ihren Gunsten ausgelegt, d.h. es handelt sich dort meist um Bandmatrizen, Tridiagonalmatrizen etc. .

4.2.1 Speicherung dünnbesetzter Matrizen

Allgemein werden nur die Nichtnullemente der Matrix gespeichert, daher ist eine indirekte Adressierung der Matrixeinträge über irgendwie geartete Zeiger notwendig.

Compressed Row Storage (CRS)

Zur Speicherung der Matrix

$$A_{n\times m} = \begin{pmatrix} 10 & 0 & 0 & -2 \\ 3 & 9 & 0 & 0 \\ 0 & 7 & 8 & 7 \\ 3 & 0 & 8 & 7 \end{pmatrix}$$

werden zwei INTEGER-Felder und ein REAL/DOUBLE-Feld benötigt.

Werte	:	*val* =	10, −2, 3, 9, 7, 8, 7, 3, 8, 7
Spaltenindex	:	*col_ind* =	1, 4, 1, 2, 2, 3, 4, 1, 3, 4
Zeilenpointer (auf Zeilenanfang)	:	*row_ptr* =	1, 3, 5, 8, 11

Der Zeilenpointer kann auch auf das Zeilenende oder bei symmetrischen Matrizen auf die Diagonale zeigen.

Analog funktioniert die Speicherform Compressed Column Storage (CCS)

Skyline Storage

Wird vorrangig bei Matrizen mit variabler Band- bzw. Profilbreite verwandt. Günstig bei direkten Lösern (z.B. Gauß, LU-Zerlegung), da das Band/Profil erhalten bleibt.

Weitere Speicherarten für dünnbesetzte Matrizen siehe Dongarra[1] u.a. [BBC+94]

4.2.2 Operationen mit dünnbesetzten Matrizen im Vektorrechner

Verschiedene Matrixspeicherungen
$\Longrightarrow$ verschiedene Implementierungen für die gleiche Operation.

Bsp.: $\underline{y} = A_{n\times n} \cdot \underline{x}$ *in CRS*

```
DO i = 1, n
    y(i) := 0
    DO j = row_ptr(i), row_ptr(i+1) - 1
        y(i) = y(i) + val(j) * x(col_ind(j))
    END DO
END DO
```

Wie obige Doppelschleife vektorisieren ?

1. Zeile i in A aufblasen und mit Nullen füllen
   $\Longrightarrow$ innere Schleife $\widehat{=}\ y_i = y_i + (A_{i,*}, \underline{x})$
   Widerspruch ist bei 3D-FEM Matrix kompletter Unsinn.

2. GATHER/SCATTER (sammeln/verteilen) auf der Matrixzeilen mittels des Indexvektor

   ```
   y = 0
   DO i = 1, n
       length := row_ptr(i+1) + 1 - row_ptr(i)      [Register mit Vektorlänge]
       j1 := row_ptr(i)
       y(i) := y(i) + val(j1) * x(col_ind(j1))
   END DO
   ```

   Der Vektorrechner lädt das Indexfeld $col\_ind(j1)$ und adressiert damit den entsprechenden Eintrag in $\underline{x}$ (GATHER). Analog läßt sich ein entsprechender Vektor speichern (SCATTER).

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

- Ähnlich werden Matrix-Matrix-Operationen ausgeführt.
- Durch die indirekte Adressierung sind Speicherbankkonflikte und Page faults nicht zu vermeiden.
- Die Parallelisierung des Matrix-Vektor-Produkts für dünnbesetzte Matrizen sollte die (lokale) Besetztheitsstruktur der Matrix ausnutzen (Abschnitt 4.3).

4.3 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 wereden. Das Bild 4.4 zeigt einen Ausschnitt einer nichtüberlappenden Gebietszerlegung. Die Kanten zwischen den Teilgebieten werden als Interfaces bezeichnet. Die überlappende Gebietszerlegung wird im folgenden nicht betrachtet.

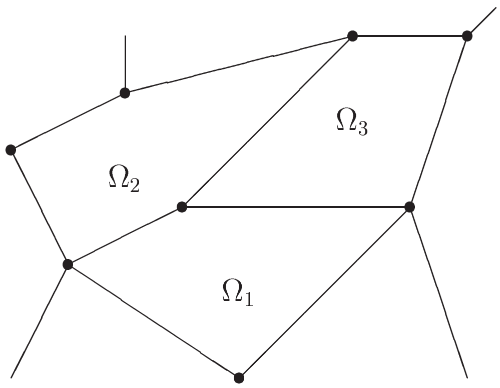


Abbildung 4.4: Nichtüberlappende Gebietsaufteilung.

Bezüglich der Datenart unterscheiden wir zwischen element- und knotenbasierten Daten, welche nunmehr überlappend oder nichtüberlappend gespeichert werden (Aber die Gebiete an sich sind nichtüberlappend aufgeteilt !). Somit ergeben sich 4 Arten der Datenverteilung, von denen die nichtüberlappende 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.

4.3.1 Nichtüberlappende Elemente

Sei das Rechengebiet Ω in z.B. 4 Teilgebiete zerlegt und mittels lineareren Dreieckselementen diskretisiert. In Bild 4.5 wird dies veranschaulicht.

Wir unterscheiden 3 Arten von Knoten mittel der Indizes :

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

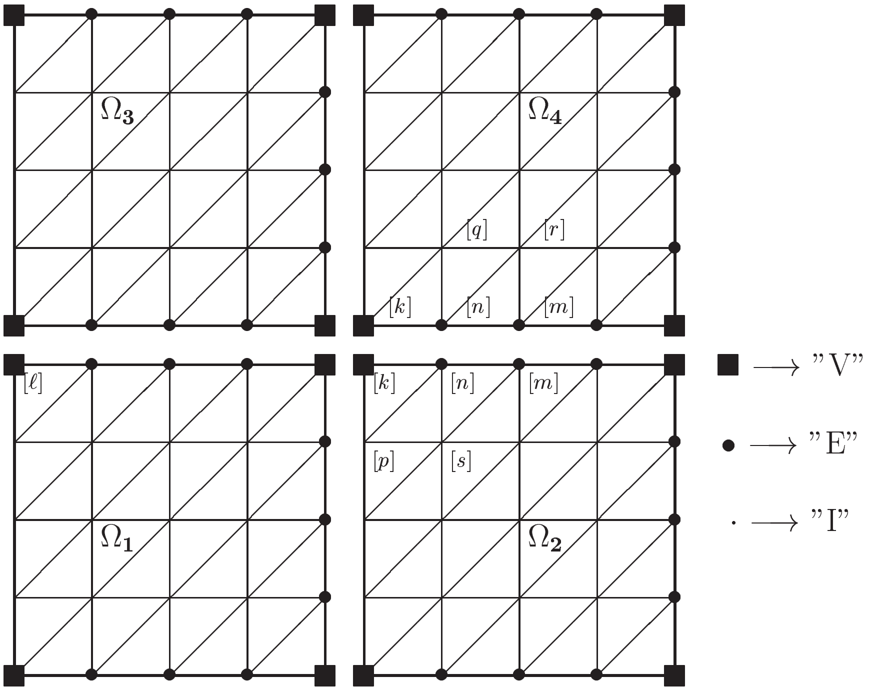


Abbildung 4.5: Nichtüberlappende Elemente.

"E" Knoten auf dem Inneren von Teilgebietskanten (edges, interfaces) $[N_E := \sum_{j=1}^{n_e} N_{E,j}]$,

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

Die beiden letzteren werden oft als Koppelknoten mit dem Index "C" $[N_C = N_V + N_E]$ zusammengefaßt. Die Gesamtzahl der Knoten ergibt sich zu $N = N_V + N_E + N_I = \sum_{i=1}^{P} N_i$.

Zur Vereinfachung der Darstellung werden zuerst die Crosspoints, dann die Kantenknoten und danach die inneren Knoten numeriert. Innerhalb der Kantenknoten sind besitzen die Knoten einer Kante eine fortlaufende Numerierung, desgleichen die inneren Knoten eines Teilgebietes. Somit besitzen alle Vektoren und Matrizen eine Blockstruktur der Art

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

Entsprechend der Zuordnung der Knoten zu den P Teilgebieten $\overline{\Omega}_i$ $(i = 1, 2, \ldots, P)$ werden die Einträge der Matrizen und Vektoren auf die entsprechenden Prozesse $\mathbb{P}_i$ verteilt. Koinzidenzmatrizen A_i $({\scriptstyle i=\overline{1,P}})$ repräsentieren symbolisch diese Knotenzuordnung.

Die $N \times N_i$ Matrix A_i ist eine Boolesche Matrix welche einen globalen Vektor $\underline{v}$ auf den lokalen Vektor $\underline{v}_i$ 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 (Typ I) und den verteilten (Typ-II) Vektor :

Typ I : $\underline{\mathfrak{u}}$ und $\underline{\mathfrak{w}}$ werden in Prozessor $\mathbb{P}_i$ ($\hat{=} \ \overline{\Omega}_i$) als $\underline{\mathfrak{u}}_i = A_i \underline{\mathfrak{u}}$ und $\underline{\mathfrak{w}}_i = A_i \underline{\mathfrak{w}}$ gespeichert, dh. jeder Prozessor $\mathbb{P}_i$ besitzt den vollen Wert seiner Knoten.

Typ II : $\underline{\mathsf{r}}, \underline{\mathsf{f}}$ wird in $\mathbb{P}_i$ als $\underline{\mathsf{r}}_i, \underline{\mathsf{f}}_i$ gespeichert, so daß $\underline{\mathsf{r}} = \sum\limits_{i=1}^{p} A_i^T \underline{\mathsf{r}}_i$ gilt, dh. die Knoten auf den Interfaces ($\underline{\mathsf{r}}_{C,i}$) besitzen nur einen Teil des wahren Wertes.

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

$$\mathsf{K} = \sum_{i=1}^{p} A_i^T \mathsf{K}_i A_i \quad , \tag{4.1}$$

wobei K_i die zum Teilgebiet $\overline{\Omega}_i$ gehörige Steifigkeitsmatrix darstellt. Denkt man sich $\overline{\Omega}_i$ als finites Element, so stellt die verteilte Matrixspeicherung genau die Darstellung einer Steifigkeitsmatrix vor der FE-Akkumulation dar. Die spezielle globale Knotennumerierung induziert folgende Blockdarstellung der Gleichung $\mathsf{K} \cdot \underline{\mathfrak{u}} = \underline{\mathsf{f}}$:

$$\begin{pmatrix} \mathsf{K}_V & \mathsf{K}_{VE} & \mathsf{K}_{VI} \\ \mathsf{K}_{EV} & \mathsf{K}_E & \mathsf{K}_{EI} \\ \mathsf{K}_{IV} & \mathsf{K}_{IE} & \mathsf{K}_I \end{pmatrix} \cdot \begin{pmatrix} \underline{\mathfrak{u}}_V \\ \underline{\mathfrak{u}}_E \\ \underline{\mathfrak{u}}_I \end{pmatrix} = \begin{pmatrix} \underline{\mathsf{f}}_V \\ \underline{\mathsf{f}}_E \\ \underline{\mathsf{f}}_I \end{pmatrix} \quad . \tag{4.2}$$

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

Falls eine globale Akkumulation der Matrix K ausgeführt wird, bezeichnen wir das Ergebnis mit der Typ-I Matrix $\mathfrak{M}$ und schreiben

$$\mathfrak{M}_i \ := \ A_i \mathfrak{M} A_i^T \quad . \tag{4.3}$$

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

Die Diagonalmatrix

$$\boxed{R \;=\; \sum_{i=1}^{p} A_i^T A_i} \tag{4.4}$$

enthält für jeden Knoten die Anzahl der Teilgebiete (Wertigkeit), zu denen er gehört (z.B. in Bild 4.5 $R^{[k]} := 4$, $R^{[n]} = R^{[m]} = R^{[p]} = 2$, $R^{[q]} = 1$).

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

Generierung der Steifigkeitsmatrix.

Bei einer lokalen FEM, FDM, FVM Diskretisierung ensteht ganz natürlich eine Typ-II Matrix.
Achtung : Bei Konvektionstermen in pdes ist bei Upwind-Methoden höherer Ordnung (besser als lineares Upwind) zusätzliche Information aus den Nachbarelementen notwendig (Kommunikation!).

Umwandlung der Vektortypen.

Offensichtlich können Addition/Subtraktion und analoge Operationen mit Vektoren gleichen Typs ohne Kommunikation ausgeführt werden.
Die Umwandlung eines Typ II in einen Typ I Vektor erfordert Kommunikation :

$$\boxed{\underline{\mathfrak{w}}_i \;=\; A_i \sum_{i=1}^{p} A_i^T \underline{\mathsf{r}}_i \;\; .} \tag{4.5}$$

Die andere Umwandlung eines Typ I in einen Typ II Vektor ist nicht eindeutig. Eine Möglichkeit besteht in der lokalen Division jeder Vektorkomponente durch die Anzahl der anliegenden Teilgebiete, z.B.

$$\boxed{\underline{\mathsf{r}}_i \;=\; R^{-1} \underline{\mathfrak{w}}_i} \tag{4.6}$$

mit R in (4.4) definiert.

Skalarprodukt.

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

$$\boxed{(\underline{\mathfrak{w}}, \underline{\mathfrak{r}}) = \underline{\mathfrak{w}}^T \underline{\mathfrak{r}} = \underline{\mathfrak{w}}^T \sum_{i=1}^{p} A_i^T \underline{\mathfrak{r}}_i = \sum_{i=1}^{p} (A_i \underline{\mathfrak{w}})^T \underline{\mathfrak{r}}_i = \sum_{i=1}^{p} (\underline{\mathfrak{w}}_i, \underline{\mathfrak{r}}_i)} \tag{4.7}$$

Jede andere Vektorkombination erfordert eine Vektortypumwandlung mit evtl. Kommunikation.

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 man in [Gro94].

1. Typ-II Matrix × Typ-I Vektor ergibt einen Type-II Vektor.
   Tatsächlich ergibt sich aus Definition (4.1) :

   $$\mathsf{K} \cdot \underline{\mathfrak{w}} = \sum_{i=1}^{p} A_i^T \mathsf{K}_i A_i \cdot \underline{\mathfrak{w}} = \sum_{i=1}^{p} A_i^T \underbrace{\mathsf{K}_i \cdot \underline{\mathfrak{w}}_i}_{\underline{\mathfrak{r}}_i} = \underline{\mathfrak{r}} \tag{4.8}$$

   Die Ausführung der Summation (d.h. Kommunikation) führt zu einem Typ-I Vektor.

2. Typ-II Matrix × Typ-II Vektor erfordert eine Vektorkonvertierung bevor obige Multiplikation ausgeführt werden kann.

3. Typ-I Matrix × Typ-I Vektor kann nicht mit beliebigen Typ-I Matrizen $\mathfrak{M}$ berechnet werden.
   Zur genaueren Analyse werfen wir einen Blick auf den Knoten n in Abb. 4.5 und die Operation $\underline{\mathfrak{u}} = \mathfrak{M} \cdot \underline{\mathfrak{w}}$. Die lokale Multiplikation $\underline{\mathfrak{u}}_i = \mathfrak{M}_i \cdot \underline{\mathfrak{w}}_i$ sollte als Ergebnis einen Typ-I Vektor $\underline{\mathfrak{u}}$ im Knoten n hervorbringen.

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

   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. 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 ($\omega_i \supseteqq \omega_j$). Stellt man die Matrixeinträge als gerichteten Graphen dar, so heißt dies z.B., daß in Abb. 4.5 die Einträge $q \to k$, $q \to n$, $s \to n$, $s \to m$, $s \to p$, $p \to k$, $n \to k$, $p \to n$, $n \to p$, $\ell \to k$, $k \to \ell$ in der Matrix nicht erwünscht sind.
Somit vermeidet nur eine Matrix der Form

$$\mathfrak{M} = \begin{pmatrix} \mathfrak{M}_V & 0 & 0 \\ \mathfrak{M}_{EV} & \mathfrak{M}_E & 0 \\ \mathfrak{M}_{IV} & \mathfrak{M}_{IE} & \mathfrak{M}_I \end{pmatrix} \quad \Longrightarrow \quad \underline{\mathfrak{u}} = \mathfrak{M} \cdot \underline{\mathfrak{w}} \tag{4.9}$$

mit Blockdiagonalmatrizen $\mathfrak{M}_I$, $\mathfrak{M}_E$, $\mathfrak{M}_V$ solche unerwünschten Einträge. Während die Blockdiagonalstruktur von $\mathfrak{M}_I$ durch die Datenaufteilung garantiert ist, muß das FE-Netz für die restlichen Matrizen die folgenden Anforderungen erfüllen :

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

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

Bedingung 3a kann einfach durch dadurch erreicht werden, daß auf jeder Gebietskante mind. ein zusätzlicher Kantenknoten plaziert wird (Achtung bei Verbindung quer durch das Teilgebiet). Ein 2D Netzgenerators für Dreiecke mit linearen Ansatzfunktionen kann leicht so modifiziert werden, daß Bedingung 3b eingehalten wird, z.B. wird die Kante zwischen den Knoten p und n vermieden. Abb. 4.6 zeigt das geänderte Netz.
• Diese Vorgehen ist nur bei bestimmten finiten Elementen möglich !
• Bei Verwendung von Tetraedern mit linearen Ansatzfunktionen kann Bedingung 3b auch in 3D Netzgeneratoren eingebaut werden.

4. Wie im vorangegangen Punkt kann Typ-I Matrix × Typ-II Vektor nicht mit allgemeinen Typ-I Matrizen $\mathfrak{M}$ durchgeführt werden. Das Ergebnis sollte ein Typ-II Vektor sein.
Nehmen wir an, die (auch hier notwendigen) Bedingungen 3a and 3b seien durch das Netz (Abb. 4.6) erfüllt. Richten wir unsere Aufmerksamkeit auf die Operation $\underline{\mathsf{f}}_I = \mathfrak{M}_{IC} \cdot \underline{\mathsf{r}}_C$. Lokal auf Prozessor 4 ausgeführt liefert

$$\mathsf{f}_4^{[r]} = \mathfrak{M}_{IC,4}^{[r,n]} \cdot \mathsf{r}_4^{[n]} + \mathfrak{M}_{IC,4}^{[r,m]} \cdot \mathsf{r}_4^{[m]} \ .$$

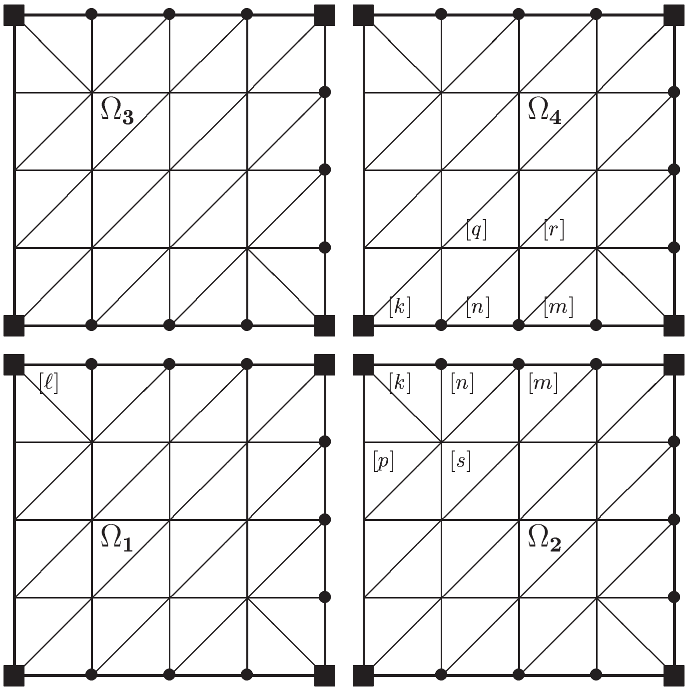


Abbildung 4.6: Nichtüberlappende Elemente mit Diskretisierung bei angepaßtem Netz.

Wegen der Eigenschaften des Typ-II Vektors r fehlen im Resultat die Teile $\mathfrak{M}_{IC,2}^{[r,n]} \cdot \mathsf{r}_2^{[n]}$ und $\mathfrak{M}_{IC,2}^{[r,m]} \cdot \mathsf{r}_2^{[m]}$. Im Unterschied zum vorigen Punkt ist der Transport von Information von Knoten i zu Knoten j nur zulässig, falls die Menge der Prozessoren zu denen j gehört eine Teilmenge der Prozessoren von Knoten i ist $(\omega_i \subseteqq \omega_j)$. In Abb.4.6 sind z.B. $n \to k$ und $q \to k$ erlaubt, aber nicht die umgekehrte Richtung. Damit kann nur eine Matrix der Form

$$\mathfrak{M} = \begin{pmatrix} \mathfrak{M}_V & \mathfrak{M}_{VE} & \mathfrak{M}_{VI} \\ 0 & \mathfrak{M}_E & \mathfrak{M}_{EI} \\ 0 & 0 & \mathfrak{M}_I \end{pmatrix} \implies \underline{\mathsf{f}} = \sum_{i=1}^{P} A_i^T \underline{\mathsf{f}}_i = \sum_{i=1}^{P} A_i^T \left(\mathfrak{M}_i \underline{\mathsf{r}}_i\right) \tag{4.10}$$

mit Blockdiagonalmatrizen $\mathfrak{M}_I$, $\mathfrak{M}_E$, $\mathfrak{M}_V$ für diese Art der Matrix-mal-Vektor Multiplikation benutzt werden.

Bemerkung: Die Punkte 3 und 4 können unter der Bedingung, daß $\mathfrak{M}_V$, $\mathfrak{M}_E$, $\mathfrak{M}_I$ blockdiagonal sind (durch das Netz in Abb. 4.6 garantiert) kombiniert werden. Bezeichnen $\mathfrak{M}_L$, $\mathfrak{M}_U$ und $\mathfrak{M}_D$ das streng untere bzw. streng obere Dreiecks und die Diagonale von $\mathfrak{M}$, kann die Typ I Matrix-mal-Vektor

Operation für alle Vektortypen aufgeschrieben werden.

$$\underline{\mathfrak{w}} = \mathfrak{M} \cdot \underline{\mathfrak{u}} \;:=\; (\mathfrak{M}_L + \mathfrak{M}_D) \cdot \underline{\mathfrak{u}} + \sum_{i=1}^{P} A_i^T \mathfrak{M}_{U,i} R_i^{-1} \cdot \underline{\mathfrak{u}}_i \tag{4.11a}$$

$$\underline{\mathfrak{w}} = \mathfrak{M} \cdot \underline{\mathfrak{r}} \;:=\; (\mathfrak{M}_L + \mathfrak{M}_D) \sum_{i=1}^{P} A_i^T \cdot \underline{\mathfrak{r}}_i + \sum_{i=1}^{P} A_i^T \mathfrak{M}_{U,i} \cdot \underline{\mathfrak{r}}_i \tag{4.11b}$$

$$\underline{\mathfrak{f}} = \mathfrak{M} \cdot \underline{\mathfrak{u}} \;:=\; R^{-1}(\mathfrak{M}_L + \mathfrak{M}_D) \cdot \underline{\mathfrak{u}} + \mathfrak{M}_U R^{-1} \cdot \underline{\mathfrak{u}} \tag{4.11c}$$

$$\underline{\mathfrak{f}} = \mathfrak{M} \cdot \underline{\mathfrak{r}} \;:=\; R^{-1}(\mathfrak{M}_L + \mathfrak{M}_D) \sum_{i=1}^{P} A_i^T \cdot \underline{\mathfrak{r}}_i + \mathfrak{M}_U \cdot \underline{\mathfrak{r}} \;. \tag{4.11d}$$

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

Die Faktorisierung der Matrix $\mathfrak{M}$ in eine obere und untere Dreiecksmatrix $\mathfrak{L}^{-1}$ und $\mathfrak{U}^{-1}$, die den Bedingungen 3 bzw. 4 genügen müssen, kann auf zwei Weisen geschehen. Zum einem gilt

$$\underline{\mathfrak{w}} = \mathfrak{L}^{-1}\mathfrak{U}^{-1} \cdot \underline{\mathfrak{r}} \;:=\; \mathfrak{L}^{-1} \sum_{i=1}^{P} A_i^T \mathfrak{U}_i^{-1} \cdot \underline{\mathfrak{r}}_i \tag{4.12a}$$

$$\underline{\mathfrak{f}} = \mathfrak{L}^{-1}\mathfrak{U}^{-1} \cdot \underline{\mathfrak{u}} \;:=\; R^{-1}\mathfrak{L}^{-1} \sum_{i=1}^{P} A_i^T \mathfrak{U}_i^{-1} R^{-1} \cdot \underline{\mathfrak{u}}_i \;, \tag{4.12b}$$

und andererseits

$$\underline{\mathfrak{w}} = \mathfrak{U}^{-1}\mathfrak{L}^{-1} \cdot \underline{\mathfrak{r}} \;:=\; \sum_{i=1}^{P} A_i^T \mathfrak{U}_i^{-1} R_i^{-1} A_i \mathfrak{L}^{-1} \cdot \Big(\sum_{j=1}^{P} A_j^T \underline{\mathfrak{r}}_j\Big) \tag{4.13a}$$

$$\underline{\mathfrak{f}} = \mathfrak{U}^{-1}\mathfrak{L}^{-1} \cdot \underline{\mathfrak{u}} \;:=\; \mathfrak{U}^{-1} R^{-1} \mathfrak{L}^{-1} \cdot \underline{\mathfrak{u}} \;. \tag{4.13b}$$

Die fehlenden Gleichungen können leicht mittels Typkonvertierung erhalten werden. Bei der Faktorisierung beeinflußt die konkrete Wahl der Vektortypen die Anzahl und Art der der Konvertierungen.

4.3.2 Nichtüberlappende Knoten

Bei der nichtüberlappenden Knotenaufteilung werden die Knoten disjunkt aufgeteilt.

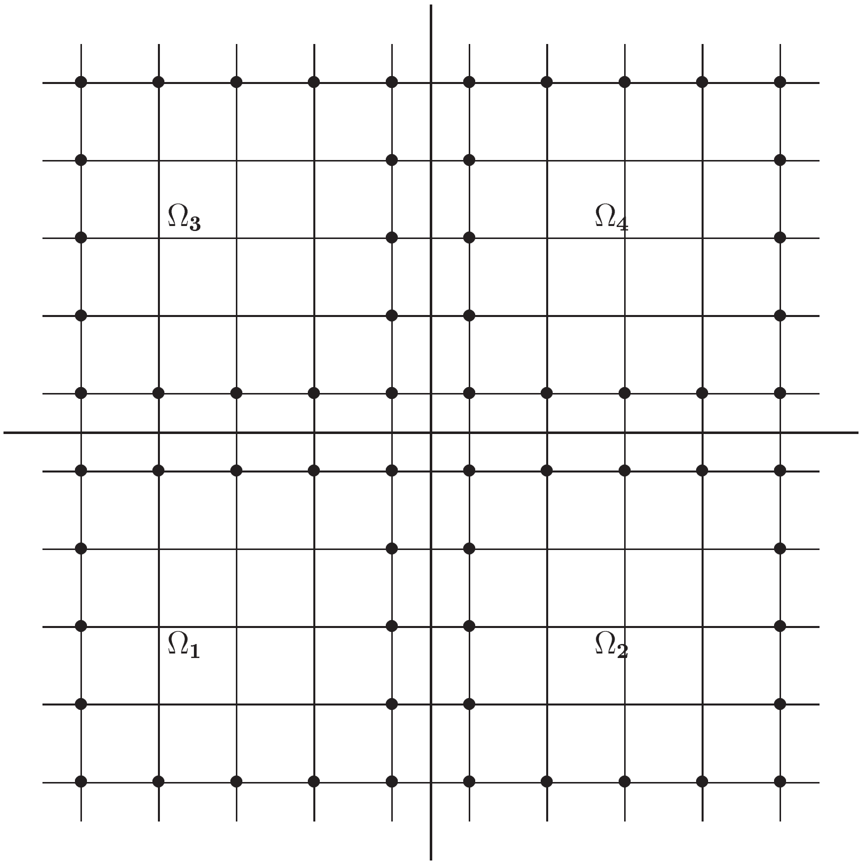


Abbildung 4.7: Nichtüberlappende Knoten.

Generierung der Steifigkeitsmatrix.

Bei FVM entspricht diese Aufteilung einer natürlichen Verteilung der finiten Volumina. Im FEM- und FDM-Fall wird Information aus dem Nachbarprozessor benötigt. In jedem Falle muß die lokale Steifigkeitsmatrix K_i alle Verbindungen zu den ersten Knotenschichten (bei linearen Ansatzelementen nur die erste) der Nachbarprozessoren beinhalten.

Umwandlung der Vektortypen.

Hier treten nur Typ-I Vektoren welche disjunkt gespeichert werden.

Skalarprodukt.

Es muß wiederum nur die globale Summe (AllReduce) einer reellen Zahl berechnet werden. Die Teilvektoren sind disjunkt gespeichert !

$$(\underline{\mathfrak{w}}, \underline{\mathfrak{r}}) \;=\; \underline{\mathfrak{w}}^T \underline{\mathfrak{r}} \;=\; \sum_{i=1}^{p} (\underline{\mathfrak{w}}_i, \underline{\mathfrak{r}}_i)$$

Matrix-Vektor Multiplikation.

Die Matrizen sind vom Typ-I, enthalten aber Verbindungen zu den Nachbargebieten. Deren entsprechende Knoteninformation eines Vektors $\underline{\mathfrak{r}}$ muß vorher via Datenaustausch geholt werden, d.h. temporär wird dieser Vektor mit überlappenden Knoten gespeichert. Die restliche lokale Matrix-Vektor Multiplikation läuft völlig parallel ab.

4.3.3 Überlappende Elemente

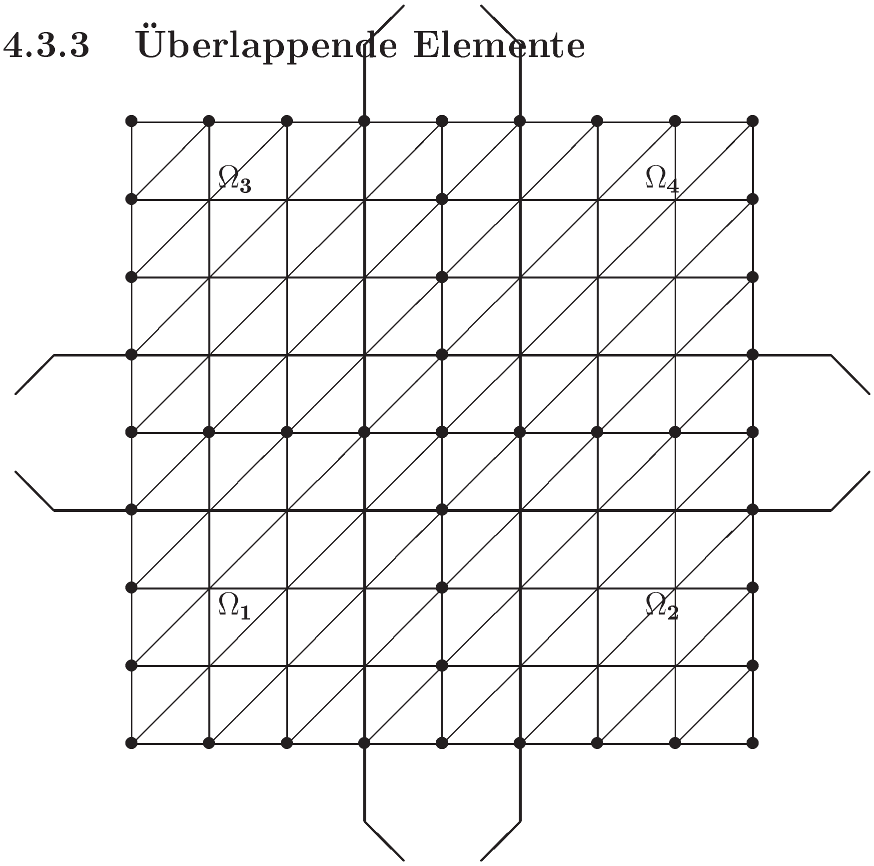


Abbildung 4.8: Überlappende Elemente.

Bei dieser Datenverteilung gibt es eine Menge von Knoten, welche im Überlappungsbereich zu mehreren Teilgebieten gehören : $x_{C_{i,j}} \in \overline{\Omega}_{h,i} \cap \overline{\Omega}_{h,j}$. Diese Verteilungsart hat inbesondere bei FEM mit Upwind-Techniken Vorteile.

Generierung der Steifigkeitsmatrix.

Sei $\widetilde{\mathfrak{K}}_i$ der Teil der globalen Steifigkeitsmatrix, der alle Einträge von Knoten des Teilgebietsinneren ($x \in \overline{\Omega}_{h,i} \subset \Gamma_{h,i}$) mit Knoten des Teilgebietes ($x \in$

$\overline{\Omega}_{h,i}$) verbindet, d.h. durch das Fehlen der äußeren Knotenschicht ist $\widetilde{\mathfrak{K}}_i$ eine Rechteckmatrix vom Typ-I (akkumuliert). In Skizze 4.8 besitzt Prozeß $\mathbb{P}_i$ die Einträge $\widetilde{\mathfrak{K}}^{[k,k]}$ und $\widetilde{\mathfrak{K}}^{[k,\ell]}$, während $\widetilde{\mathfrak{K}}^{[\ell,\ell]}$, $\widetilde{\mathfrak{K}}^{[\ell,k,]}$ und $\widetilde{\mathfrak{K}}^{[k,k]}$ in $\mathbb{P}_j$ gespeichert sind.
Es sind andere Arten der Matrixspeicherung denkbar.

Umwandlung der Vektortypen.

Alle Vektoren werden mit ihrem vollen Wert gespeichert (Typ-I). Jedoch müssen diese Werte immer wieder zwischen den beteiligten Prozessen abgeglichen werden, da die lokalen Rechnungen in der Regel unterschiedliche Ergebnisse liefern. Dieser Abgleich kann als zweifache Vektortypumwandlung unter Nutzung von (4.6) und (4.7) geschrieben werden :

$$\underline{\mathfrak{w}}_i \;:=\; A_i \sum_{i=1}^{p} A_i^T R^{-1} \underline{\mathfrak{w}}_i \tag{4.14}$$

Skalarprodukt.

Vor Ausführung des Skalarproduktes, analog zu 4.3.1, muß einer der Vektoren wiederum in einen verteilt gespeicherten umgewandelt werden. Somit ergibt sich das Skalarprodukt zu

$$(\underline{\mathfrak{w}}, \underline{\mathfrak{r}}) \;=\; \sum_{i=1}^{p} (\underline{\mathfrak{w}}_i, R^{-1} \underline{\mathfrak{r}}_i) \tag{4.15}$$

Matrix-Vektor Multiplikation.

Die Matrix-Vektor Multiplikation $\underline{\mathfrak{w}} = \widetilde{\mathfrak{K}} \underline{\mathfrak{u}}$ mit den Typ-I Vektoren $\underline{\mathfrak{w}}$ und $\underline{\mathfrak{u}}$ ist im Gegensatz zu Abschnitt 4.3.1 ausführbar, da die äußere Knotenschicht eines Teilgebietes zwar in $\underline{\mathfrak{u}}$ aber nicht in $\underline{\mathfrak{w}}$ gespeichert wird. Die Werte $\underline{\mathfrak{w}}$, welche zu Knoten der äußeren Randschicht des Teilgebietes gehören, müssen über eine zusätzlichen Kommunikationsschritt ermittelt werden.

4.3.4 Überlappende Knoten

Die überlappende Knotenverteilung ist bei Konvektionsproblemen und einer FEM-, FVM-Diskretisierung beliebt. Die Durchführung der Grundroutinen erfolgt analog zu Punkt 4.3.3 .

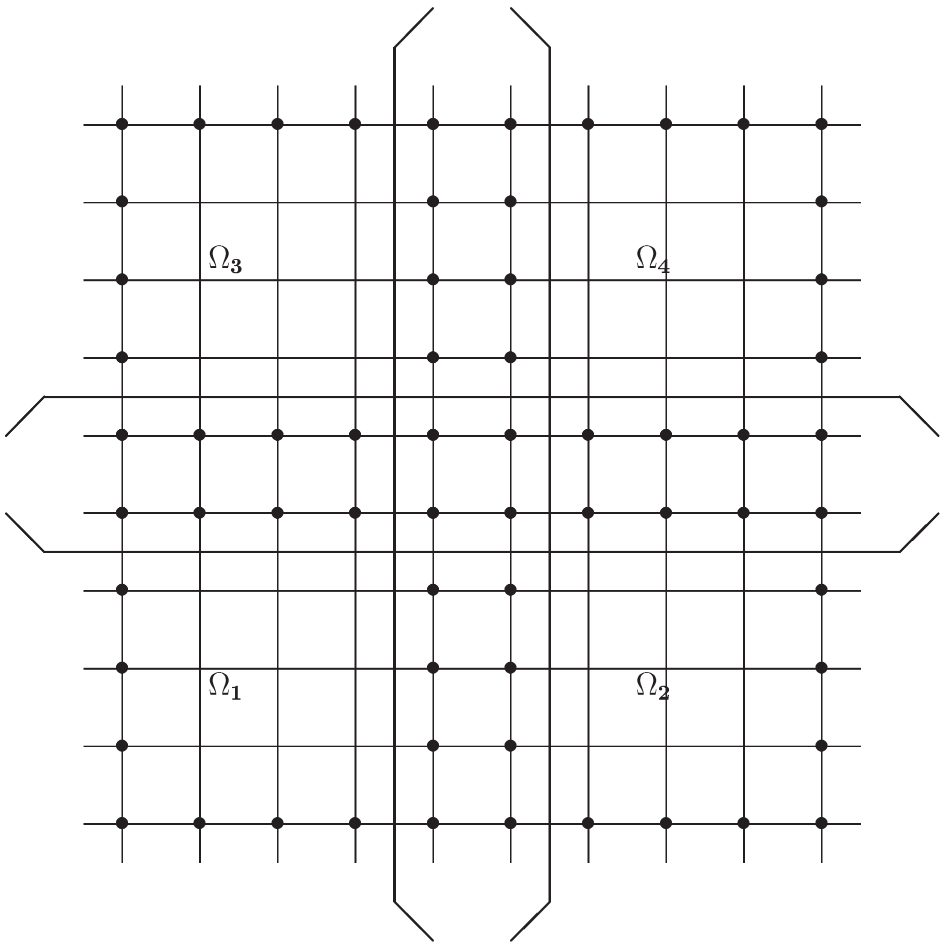


Abbildung 4.9: Überlappende Knoten.

Kapitel 5

Vektorisierung und Parallelisierung iterativer Verfahren

In diesem Kapitel werden lineare Gleichungssysteme der Form

$$K_{n\times n} \cdot \underline{x} \;=\; \underline{b} \tag{5.1}$$

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) ensteht eine dünnbesetzte (sparse) Matrix K, welche als Grundlage für die folgenden Untersuchungen stehen soll.

Im parallelen Fall wird auf Abschnitt 4.3 und speziell auf die **nichtüberlappende Elementverteilung** 4.3.1 Bezug genommen. Die entsprechenden Vektor- und Matrixtypen werden analog gekennzeichnet.
Andere Arten der Datenverteilung in der Parallelisierung werden explizit ausgewiesen.

5.1 Das CG-Verfahren

5.1.1 Das serielle Verfahren

Falls die Matrix K aus (5.1) symmetrisch (d.h. $K = K^T$) und positiv definit (d.h. $(K\underline{v}, \underline{v})_{L_2} > 0$) ist, kann zum Lösen von (5.1) das CG-Verfahren genutzt werden. Üblicherweise wird es in Verbindung mit einer Vorkonditionierung C

genutzt, was in Alg. 5.1 dargestellt wird

$$
\boxed{
\begin{array}{l}
\text{Wähle } \underline{u}^0 \\
\underline{r} \;:=\; \underline{f} - K \cdot \underline{u}^0 \\
\underline{w} \;:=\; C^{-1} \cdot \underline{r} \\
\underline{s} \;:=\; \underline{w} \\
\sigma \;:=\; \sigma_{old} \;:=\; \sigma_0 \;:=\; (\underline{w}, \underline{r}) \\
\textsf{repeat} \\
\qquad \begin{array}{lcl}
\underline{v} & := & K \cdot \underline{s} \\
\alpha & := & \sigma / (\underline{s}, \underline{v}) \\
\underline{u} & := & \underline{u} + \alpha \cdot \underline{s} \\
\underline{r} & := & \underline{r} - \alpha \cdot \underline{v} \\
\underline{w} & := & C^{-1} \cdot \underline{r} \\
\sigma & := & (\underline{w}, \underline{r}) \\
\beta & := & \sigma / \sigma_{\text{old}} \\
\sigma_{\text{old}} & := & \sigma \\
\underline{s} & := & \underline{w} + \beta \cdot \underline{s}
\end{array} \\
\textsf{until} \quad \sqrt{\sigma / \sigma_0} \; < \; \text{tolerance}
\end{array}
}
$$

Algorithmus 5.1: Serieller CG mit Vorkonditionierung

Eine Vektorisierung des cg an sich ist trivial, jedoch kann es bei der Vorkonditionierung Probleme geben (siehe auch 5.8).

5.1.2 Der parallelisierte CG

Wählt man in Algorithmus 5.1 $C = I$ erhält man den klassischen CG. Betrachtet man die darin enthaltenen Operationen und vergleicht sie mit den Ausführungen in Abschnitt 4.3.1, so kann man folgende *Parallelisierungsstrategie* aufbauen, wobei die Anzahl der notwendigen Kommunikationen möglichst klein gehalten werden soll.

- Die Matrix K muß als K (4.1) verteilt (Typ-II) gespeichert werden, da im anderen Fall Einschränkungen an die Matrixstruktur erfolgen müssen.

$\Longrightarrow$ Vektoren $\underline{s}$, $\underline{u}$ $\to$ $\underline{\mathfrak{s}}$, $\underline{\mathfrak{u}}$ sind akkumuliert (Typ-I)
$\Longrightarrow$ Matrix-mal-Vektor liefert Typ-II Vektor, ohne daß Kommunikation notwendig ist (4.8).

$\Longrightarrow$ Vektoren $\underline{v}$, $\underline{r}$, $\underline{f}$ $\to$ $\underline{\mathsf{v}}$, $\underline{\mathsf{r}}$, $\underline{\mathsf{f}}$ sind verteilt gespeichert (Typ-II) und werden im Verlauf der Iteration nicht akkumuliert.

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

- Da in den beiden Skalarprodukte der CG-Schleife unterschiedliche Vektortypen verwendet werden, können diese Operationen mit sehr geringem Kommunikationsaufwand ausgeführt werden (4.7).

!! Die Zuweisung $\underline{\mathfrak{w}} := \underline{\mathsf{r}}$ beinhaltet eine Vektortypumwandlung mittels Akkumulation (4.5) und damit eine Kommunikation über die zu mehreren Prozessen gehörenden Daten.

$$
\begin{array}{l}
\text{Wähle } \underline{\mathfrak{u}}^0 \\
\underline{\mathsf{r}} \;:=\; \underline{\mathsf{f}} - \mathsf{K}\cdot\underline{\mathfrak{u}}^0 \\
\underline{\mathfrak{w}} \;:=\; \sum\limits_{j=1}^{P} A_j^T \underline{\mathsf{r}}_j \\
\underline{\mathfrak{s}} \;:=\; \underline{\mathfrak{w}} \\
\sigma \;:=\; \sigma_{old} \;:=\; \sigma_0 \;:=\; (\underline{\mathfrak{w}}, \underline{\mathsf{r}}) \\
\mathsf{repeat} \\
\qquad \begin{array}{lcl}
\underline{\mathsf{v}} & := & \mathsf{K}\cdot\underline{\mathfrak{s}} \\
\alpha & := & \sigma/(\underline{\mathfrak{s}}, \underline{\mathsf{v}}) \\
\underline{\mathfrak{u}} & := & \underline{\mathfrak{u}} + \alpha\cdot\underline{\mathfrak{s}} \\
\underline{\mathsf{r}} & := & \underline{\mathsf{r}} - \alpha\cdot\underline{\mathsf{v}} \\
\underline{\mathfrak{w}} & := & \sum\limits_{j=1}^{P} A_j^T \underline{\mathsf{r}}_j \\
\sigma & := & (\underline{\mathfrak{w}}, \underline{\mathsf{r}}) \\
\beta & := & \sigma/\sigma_{\text{old}} \\
\sigma_{\text{old}} & := & \sigma \\
\underline{\mathfrak{s}} & := & \underline{\mathfrak{w}} + \beta\cdot\underline{\mathfrak{s}}
\end{array} \\
\mathsf{until}\quad \sqrt{\sigma/\sigma_0} \;<\; \text{tolerance}
\end{array}
$$

Algorithmus 5.2: Parallelisierter CG

Der resultierende parallele CG-Algorithmus 5.2 erfordert pro Iteration 2 ALL_REDUCE-Operationen mit einer reellen Zahl und eine Vektorakkumulation.

Man kann auch über andere Ansätze zu Algorithmus 5.2 gelangen, so sind z.B. die verteilt gespeicherten Vektoren $\underline{\mathsf{f}}$, $\underline{\mathsf{r}}$ und $\underline{\mathsf{v}}$ 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 : 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 5.8.

5.2 Das GMRES-Verfahren

5.2.1 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. $(K\underline{v}, \underline{v})_{L_2} \not> 0$), 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.

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

Algorithmus 5.3: Serieller GMRES mit Vorkonditionierung

Da die Anzahl der zu 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 erhalten Lösung $\underline{u}^m$ neu. Dieser GMRES(m) kann allerdings instabil werden.

Zur Vorkonditionierungsproblematik siehe Abschnitt 5.8.

5.2.2 Der parallele GMRES

Bei Algorithmus 5.3 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.

- $\underline{f}$, $\underline{r} \to \underline{\mathsf{f}}$, $\underline{\mathsf{r}}$ werden verteilt gespeichert.
- $\underline{u}$, $\underline{w}^k \to \underline{\mathfrak{u}}$, $\underline{\mathfrak{w}}^k$ 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.

$\Longrightarrow$ Skalarprodukte können mit sehr geringem Kommunikationsaufwand ausgeführt werden (4.7). Matrix-mal-Vektor benötigt keine Kommunikation (4.8).

!! Die Zuweisung $\underline{\mathfrak{w}} := \underline{\mathsf{r}}$ beinhaltet eine Vektortypumwandlung mittels Akkumulation (4.5) 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

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

welche verschieden Vektortypen enthält. Eine Umwandlung des akkumulierten Vektors $\underline{\mathfrak{w}}^i$ in einen verteilten ist nach (4.6) aber ohne Kommunikation möglich !

$\Longrightarrow$ Alle DAXPY-Operationen kommen ohne Kommunikation aus.

Wähle $\underline{\mathfrak{u}}^0$

$$
\begin{array}{lll}
\underline{\mathsf{r}} & := & \underline{\mathsf{f}} - \mathsf{K}\cdot\underline{\mathfrak{u}}^0 \\
\underline{\mathfrak{w}}^1 & := & \sum\limits_{s=1}^{P} A_s^T \underline{\mathsf{r}}_s \\
z_1 & := & (\underline{\mathfrak{w}}^1, \underline{\mathsf{r}}) \\
k & := & 0
\end{array}
$$

$$
\begin{array}{ll}
\text{repeat} & k := k+1 \\
& \underline{\mathsf{r}} := \mathsf{K}\cdot\underline{\mathfrak{w}}^k \\
& \text{for } i := 1 \text{ to } k \text{ do} \\
& \qquad h_{i,k} := (\underline{\mathfrak{w}}^i, \underline{\mathsf{r}}) \\
& \qquad \underline{\mathsf{r}} := \underline{\mathsf{r}} - h_{i,k}\cdot R^{-1}\cdot\underline{\mathfrak{w}}^i \\
& \text{end} \\
& \underline{\mathfrak{w}}^{k+1} := \sum\limits_{s=1}^{P} A_s^T \underline{\mathsf{r}}_s \\
& h_{k+1,k} := (\underline{\mathfrak{w}}^{k+1}, \underline{\mathsf{r}}) \\
& \underline{\mathfrak{w}}^{k+1} := \underline{\mathfrak{w}}^{k+1}/h_{k+1,k} \\
& \text{for } i := 1 \text{ to } k-1 \text{ do} \quad \begin{pmatrix} h_{i,k} \\ h_{i+1,k} \end{pmatrix} := \begin{pmatrix} c_{i+1} & s_{i+1} \\ s_{i+1} & -c_{i+1} \end{pmatrix} \cdot \begin{pmatrix} h_{i,k} \\ h_{i+1,k} \end{pmatrix} \\
& \alpha := \sqrt{h_{k,k}^2 + h_{k+1,k}^2} \\
& s_{k+1} := h_{k+1,k}/\alpha \ ; \ c_{k+1} := h_{k,k}/\alpha \ ; \ h_{k,k} := \alpha \\
& z_{k+1} := s_{k+1} z_k \ ; \ z_k := c_{k+1} z_k \\
\text{until} & |z_{k+1}/z_1| < \text{tolerance}
\end{array}
$$

$$z_k := z_k/h_{k,k}$$

$$\text{for } i := k-1 \text{ down to } 1 \text{ do} \quad z_i := (z_i - \sum_{j=i+1}^{k} h_{i,j} z_j)/h_{i,i}$$

$$\underline{\mathfrak{u}}^k := \underline{\mathfrak{u}}^0 + \sum_{i=1}^{k} z_i \cdot \underline{\mathfrak{w}}^j$$

Algorithmus 5.4: Paralleler GMRES

Der resultierende parallele GMRES-Algorithmus 5.4 erfordert in der k-ten Iteration (k+1) ALL_REDUCE-Operationen mit einer reellen Zahl und eine Vektorakkumulation. Durch die notwendige Typumwandlung des Vektors $\underline{\mathfrak{w}}^i$ kommen noch zusätzlich $k \cdot n$ Multiplikationen hinzu (n - Länge des Vektors $\underline{\mathfrak{w}}$).

Die Variante GMRES(m), mit Neustart nach m Iterationen, läßt sich völlig analog parallelisieren.

5.3 Das ω-Jacobi Verfahren

5.3.1 Das serielle Verfahren

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

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

$\quad k := k + 1$

$\quad \underline{u}^k := \underline{u}^{k-1} + \omega \cdot D^{-1} \cdot \underline{r}$

$\quad \underline{r} := \underline{f} - K \cdot \underline{u}^k$

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

end

Algorithmus 5.5: Serielle Jacobi-Iteration

5.3.2 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 Abb. 5.1 dargestellt.

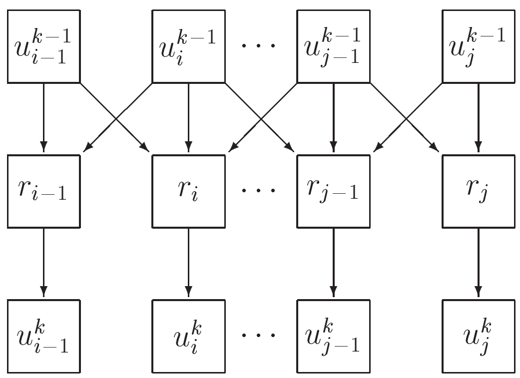


Abbildung 5.1: Datengraph der Jacobi-Iteration

5.3.3 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 5.1.2 .

- Die invertierte assemblierte Diagonale wird als Vektor abgespeichert $\underline{\mathfrak{d}} = \mathfrak{D}^{-1} = \operatorname{diag}^{-1}(\sum_{i=1}^{P} A_s^T \mathsf{K}_s A_s)$.
- Die Matrix K ist als K (4.1) verteilt (Typ-II) gespeichert.

$\Longrightarrow$ Vektor $\underline{u} \to \underline{\mathfrak{u}}$ ist akkumuliert (Typ-I), Vektoren $\underline{r}$, $\underline{f} \to \underline{\mathsf{r}}$, $\underline{\mathsf{f}}$ sind verteilt gespeichert.
$\Longrightarrow$ Matrix-mal-Vektor benötigt keine Kommunikation (4.8).

!! Das Skalarprodukt benötigt unterschiedliche Vektortypen
$\Longrightarrow$ Akkumulation von $\underline{\mathsf{r}} \longrightarrow \underline{\mathfrak{w}}$.

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

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

Algorithmus 5.6: Parallele Jacobi-Iteration : JACOBI($\mathsf{K}, \underline{\mathfrak{u}}^0, \underline{\mathsf{f}}$)

Die in Algorithmus 5.6 vorkommende Operation $\underline{\mathfrak{d}} \circledast \underline{\mathfrak{w}}$ bezeichnet die komponentenweise Multiplikation zweier Vektoren, d.h. $\{\mathfrak{d}_i \cdot \mathfrak{w}_i\}_{i=\overline{1,n}}$, 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 in den 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.

5.4 Das Gauß-Seidel Verfahren

Bezeichnen L und U die strikt untere bzw. obere Dreiecksmatrix von K aus (5.1) 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.

5.4.1 Das serielle Verfahren

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
  $k := k+1$
  $\underline{u}^k := \underline{u}^{k-1} + D^{-1} \cdot \left(\underline{f} - L \cdot \underline{u}^k - (D+U) \cdot \underline{u}^{k-1}\right)$
  $\underline{r} := \underline{f} - K \cdot \underline{u}^k$
  $\sigma := (\underline{r}, \underline{r})$
end

Algorithmus 5.7: Serieller Gauß-Seidel vorwärts

Der wesentliche algorithmische Unterschied zwischen dem ω-Jacobi Verfahren (Alg. 5.5) 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 :

$$u_i^k \;:=\; u_i^{k-1} + K_{i,i}^{-1} \left(f_i - \sum_{j=1}^{i-1} K_{i,j} u_j^k - \sum_{j=i}^{n} K_{i,j} u_j^{k-1} \right) \tag{5.2}$$

Die Iterierte u_i^k Gleichung (5.2) hängt jetzt von der Numerierung des Vektorkomponenten, d.h. ihrer Abarbeitungsreihenfolge ab. Damit ist das Gauß-Seidel-Verfahren numerierungsabhängig.

5.4.2 Datengraph der Gauß-Seidel-Iteration

Wie aus Alg. 5.7 und dem Graphen in Abb. 5.2 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

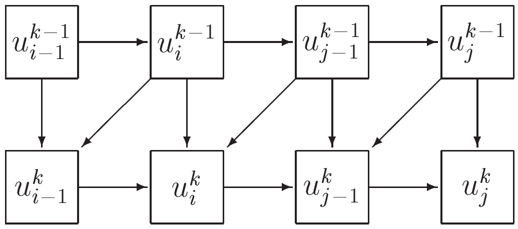


Abbildung 5.2: Datengraph der Gauß-Seidel-Iteration

$K_{i,s} \neq 0$ (Graph der Matrix !) ist, ehe u_i^k berechnet werden kann.
Diese Datenabhängigkeit ist für die Parallelisierung hinderlich und macht eine Vektorisierung unmöglich.

5.4.3 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.
Dann läßt sich der Updateschritt in Alg. 5.7 umformulieren zu

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

Algorithmus 5.8: Updateschritt im Red-Black-Gauß-Seidel vorwärts

Der entsprechende Datengraph ist in Abb. 5.3 dargestellt.

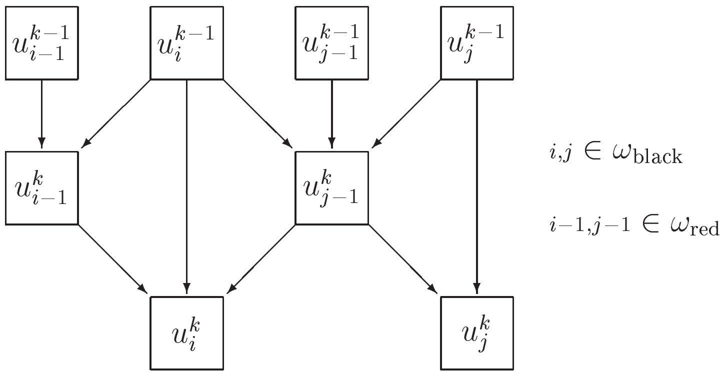


Abbildung 5.3: Datengraph der Red-Black-Gauß-Seidel-Iteration

Da innerhalb der "roten" und "schwarzen Daten" keine Abhängigkeiten mehr existieren, ist Algorithmus 5.8 nunmehr leicht vektorisierbar (mit Stride 2 !!). Diese knotenweise Red/Black-Datenaufteilung funktioniert unter bestimmten Umständen (5-Punkte Differenzenstern bei Diskretisierung auf gleichmäßigem Gitter), auch in 2D (wieviele Farben mindestens in 3D ? $\Longrightarrow$ 2 !).

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 !!), welche mit der disjunkten Datenaufteilung in Abschnitt 4.3.2 konform geht.

5.4.4 Das parallele Verfahren

Da der Updateschritt der wesentliche Unterschied gegenüber dem Jacobi-Verfahren ist, beschränken wir uns wiederum auf dessen Analyse. Als Datenverteilung werden die nichtüberlappenden Elemente vorausgesetzt (Abschnitt 4.3.1).

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

Variante 1 : formales Übertragen

Schreibt man den Updateschritt aus Algorithmus 5.7 mit den parallelen Datentypen, ist folgender Algorithmus denkbar :

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

Algorithmus 5.9: Paralleler Updateschritt Gauß-Seidel : Variante 1

Auf den ersten Blick ist man mit Algorithmus 5.9 bereits am Ziel seiner Wünsche, jedoch werden sich bei der Implementierung Schwierigkeiten einstellen, welche eine effiziente Parallelisierung fast unmöglich machen. Die gewählte Datenverteilung ist in Abb. 5.4 dargestellt.
Durch die Knotennumerierung analog zu Punkt 4.3.1 wird eine Matrix und Vektorstruktur analog zu (4.2) erzeugt.

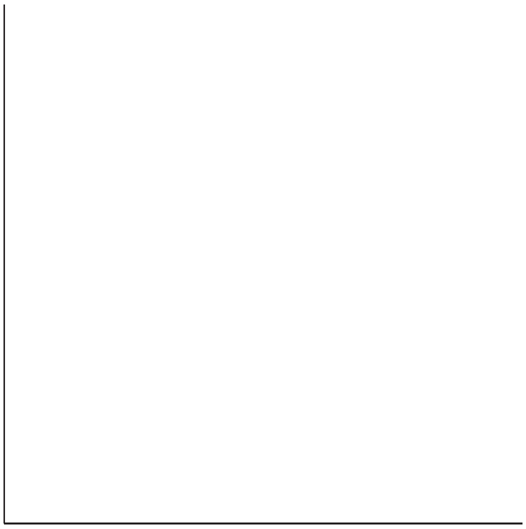


Abbildung 5.4: Gebietsaufteilung Einheitsquadrat

Für eine detailierte Analyse schlüsselt man die Berechnungen in Algorithmus 5.9 entsprechend der entstehenden Blockstruktur auf:

$$\begin{aligned}
\underline{\mathsf{r}}_V &:= \underline{\mathsf{f}}_V - \mathsf{K}_{VE}\underline{\mathfrak{u}}_E^{k-1} - K_{VI}\underline{\mathfrak{u}}_I^{k-1} && \text{(5.3a)}\\
\mathfrak{u}^k_{V,i} &:= \mathfrak{u}^{k-1}_{V,i} + \mathfrak{D}^{-1}_{V,i}\sum_{\boldsymbol{s=1}}^{\boldsymbol{P}} A^T_{s,i}\left(\mathsf{r}_{V,i} - \sum_{j=1}^{i-1}\mathsf{K}_{V,ij}\mathfrak{u}^k_{V,j} - \sum_{j=i}^{N_V}\mathsf{K}_{V,ij}\mathfrak{u}^{k-1}_{V,j}\right) && \text{(5.3b)}\\
\underline{\mathsf{r}}_E &:= \underline{\mathsf{f}}_E - \mathsf{K}_{EV}\underline{\mathfrak{u}}_V^{k} - K_{EI}\underline{\mathfrak{u}}_I^{k-1} && \text{(5.3c)}\\
\mathfrak{u}^k_{E,i} &:= \mathfrak{u}^{k-1}_{E,i} + \mathfrak{D}^{-1}_{E,i}\sum_{\boldsymbol{s=1}}^{\boldsymbol{P}} A^T_{s,i}\left(\mathsf{r}_{E,i} - \sum_{j=N_V+1}^{i-1}\mathsf{K}_{E,ij}\mathsf{u}^k_{E,j} - \sum_{j=i}^{N_E}\mathsf{K}_{E,ij}\mathsf{u}^{k-1}_{E,j}\right) && \text{(5.3d)}\\
\underline{\mathsf{r}}_I &:= \underline{\mathsf{f}}_I - K_{IV}\underline{\mathfrak{u}}_V^{k} - K_{IE}\underline{\mathfrak{u}}_E^{k} && \text{(5.3e)}\\
u^k_{I,i} &:= u^{k-1}_{I,i} + D^{-1}_{I,i}\left(r_{I,i} - \sum_{j=N_E+1}^{i-1} K_{I,ij}u^k_{I,j} - \sum_{j=i}^{N} K_{I,ij}u^{k-1}_{I,j}\right) && \text{(5.3f)}
\end{aligned}$$

Die Summe $\sum_{\boldsymbol{s=1}}^{\boldsymbol{P}} A^T_{s,i}(\cdot)$ bedeutet in *jedem* Update einer Komponente i die Akkumulation des Residuums dieser Komponente. Die Anzahl 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 :

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

In obiger Ungleichung hat der Faktor c keinerlei Einfluß auf das Verhältnis der Kommunikationszeiten der zwei Verfahren, es gilt vielmehr

$$t_{\mathrm{GS}} \;\geq\; N_C \left[\frac{c+1}{c+N_C}\right] \cdot t_{\mathrm{Jacobi}} \quad \stackrel{N_c \to \infty}{\longrightarrow} (c+1) \cdot t_{\mathrm{GS}} \;\; ,$$

d.h. die Effizienz des Gauß-Seidel-Verfahrens mit dem Updateschritt (5.3) wird weit unter dem 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 (5.3) überhaupt vernünftig parallelisierbar ist, müssen die Anforderungen 3a und 3a an das FE-Netz erfüllt sein, siehe Abb. 4.6 . Ist dies nicht der Fall, so müssen (5.3b) und (5.3d)) sequentiell abgearbeitet werden ! Ansonsten muß eine Red-Black-Numerierung der Interfacekanten durchgeführt werden, was bei allgemeinen Gebieten zu Problemen führt.

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 (5.3b) und (5.3d) 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.

$$
\begin{array}{lll}
\underline{\mathsf{d}} & := & \operatorname{diag}(K^{[i,i]})_{i=\overline{1,n}} \\
\underline{\mathfrak{d}} & := & \sum\limits_{s=1}^{P} A_s^T \underline{\mathsf{d}}_s \\
\underline{\mathfrak{d}} & := & \{1/d^{[i]}\}_{i=\overline{1,n}} \\
\multicolumn{3}{l}{\text{Wähle } \underline{\mathfrak{u}}^0} \\
\underline{\mathsf{r}} & := & \underline{\mathsf{f}} - \mathsf{K} \cdot \underline{\mathfrak{u}}^0 \\
\underline{\mathfrak{w}} & := & \sum\limits_{s=1}^{P} A_s^T \underline{\mathsf{r}}_s \\
\sigma & := & \sigma_0 \; := \; (\underline{\mathfrak{w}}, \underline{\mathsf{r}}) \\
k & := & 0 \\
\multicolumn{3}{l}{\textsf{while} \quad \sigma > \text{tolerance}^2 \cdot \sigma_0 \quad \textsf{do}} \\
 & & \begin{array}{lll}
k & := & k+1 \\
\underline{\mathsf{r}}_V & := & \underline{\mathsf{f}}_V - \mathsf{K}_V \cdot \underline{\mathfrak{u}}_V^{k-1} - \mathsf{K}_{VE} \cdot \underline{\mathfrak{u}}_E^{k-1} - K_{VI} \cdot \underline{u}_I^{k-1} \\
\underline{\mathfrak{w}}_V & := & \sum\limits_{s=1}^{P} A_{V,s}^T \underline{\mathsf{r}}_{V,s} \\
\underline{\mathfrak{u}}_V^k & := & \underline{\mathfrak{u}}_V^{k-1} + \underline{\mathfrak{d}}_V \circledast \underline{\mathfrak{w}}_V \\
\underline{\mathsf{r}}_E & := & \underline{\mathsf{f}}_E - \mathsf{K}_{EV} \cdot \underline{\mathfrak{u}}_V^k - \mathsf{K}_E \cdot \underline{\mathfrak{u}}_E^{k-1} - K_{EI} \cdot \underline{u}_I^{k-1} \\
\underline{\mathfrak{w}}_E & := & \sum\limits_{s=1}^{P} A_{E,s}^T \underline{\mathsf{r}}_{E,s} \\
\underline{\mathfrak{u}}_E^k & := & \underline{\mathfrak{u}}_E^{k-1} + \underline{\mathfrak{d}}_E \circledast \underline{\mathfrak{w}}_E \\
\underline{r}_I & := & \underline{f}_I - K_{IV} \cdot \underline{\mathfrak{u}}_V^k - K_{IE} \cdot \underline{\mathfrak{u}}_E^k \\
u_{I,i}^k & := & u_{I,i}^{k-1} + d_{I,i} \cdot \left(r_{I,i} - \sum\limits_{j=N_E+1}^{i-1} K_{I,ij} \cdot u_{I,j}^k - \sum\limits_{j=i}^{N} K_{I,ij} \cdot u_{I,j}^{k-1} \right) \\
\underline{w}_I & := & \underline{r}_I - K_I \cdot \underline{u}_I^k \\
\sigma & := & (\underline{\mathfrak{w}}, \underline{\mathsf{r}})
\end{array} \\
\multicolumn{3}{l}{\textsf{end}}
\end{array}
$$

Algorithmus 5.10: Paralleler Gauß-Seidel-ω-Jacobi vorwärts

Die in Algorithmus 5.10 vorkommende Operation $\underline{\mathfrak{d}} \circledast \underline{\mathfrak{w}}$ 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 werdem, dann wird pro Iterationsschritt nur genausoviel Kommunikation wie beim ω-Jacobi Verfahren (Algorithmus 5.6) benötigt !
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 $\underline{\mathsf{r}}$ identisch $\underline{\mathfrak{w}}$ wählen.

Variante 3 : Gauß-Seidel-Blocklöser

Im Unterschied zu Variante 2, welche (5.3b) und (5.3d) durch ein etwas schlechteres Verfahren ersetzte, werden nunmehr die entsprechenden Blocksysteme direkt gelöst. Voraussetzung für eine parallele Auflösbarkeit ist die in Abschnitt 4.3.1 geforderte Blockstruktur der akkumulierten Matrizen $\mathfrak{K}_E$ und $\mathfrak{K}_V$ (siehe auch Netzanforderungen 3a und 3b). Die Matrixeinträge $\mathfrak{K}_{E,j}$ aus $\mathfrak{K}_E = \text{blockdiag}\{\mathfrak{K}_{E,j}\}_{j=\overline{1,\text{NumEdges}}}$ 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 Lösers ist die Konvergenzgeschwindigkeit etwas besser als beim Gaus-Seidel Verfahren.

$$
\begin{aligned}
\mathfrak{K}_V &:= \sum_{s=1}^{P} A_{V,s}^T \mathsf{K}_{V,s} A_{V,s} \\
\mathfrak{K}_E &:= \sum_{s=1}^{P} A_{E,s}^T \mathsf{K}_{E,s} A_{E,s}
\end{aligned}
$$

Wähle $\underline{\mathfrak{u}}^0$

$$
\begin{aligned}
\underline{\mathsf{r}} &:= \underline{\mathsf{f}} - \mathsf{K} \cdot \underline{\mathfrak{u}}^0 \\
\underline{\mathfrak{w}} &:= \sum_{s=1}^{P} A_s^T \underline{\mathsf{r}}_s \\
\sigma &:= \sigma_0 := (\underline{\mathfrak{w}}, \underline{\mathsf{r}}) \\
k &:= 0
\end{aligned}
$$

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

$$
\begin{aligned}
k &:= k+1 \\
\underline{\mathsf{r}}_V &:= \underline{\mathsf{f}}_V - \mathsf{K}_V \cdot \underline{\mathfrak{u}}_V^{k-1} - \mathsf{K}_{VE} \cdot \underline{\mathfrak{u}}_E^{k-1} - K_{VI} \cdot \underline{u}_I^{k-1} \\
\underline{\mathfrak{w}}_V &:= \sum_{s=1}^{P} A_{V,s}^T \underline{\mathsf{r}}_{V,s} \\
\underline{\mathfrak{u}}_V^k &:= \underline{\mathfrak{u}}_V^{k-1} + \mathfrak{K}_V^{-1} \cdot \underline{\mathfrak{w}}_V \\
\underline{\mathsf{r}}_E &:= \underline{\mathsf{f}}_E - \mathsf{K}_{EV} \cdot \underline{\mathfrak{u}}_V^k - \mathsf{K}_E \cdot \underline{\mathfrak{u}}_E^{k-1} - K_{EI} \cdot \underline{u}_I^{k-1} \\
\underline{\mathfrak{w}}_E &:= \sum_{s=1}^{P} A_{E,s}^T \underline{\mathsf{r}}_{E,s} \\
\underline{\mathfrak{u}}_E^k &:= \underline{\mathfrak{u}}_E^{k-1} + \mathfrak{K}_E^{-1} \cdot \underline{\mathfrak{w}}_E \\
\underline{r}_I &:= \underline{f}_I - K_{IV} \cdot \underline{\mathfrak{u}}_V^k - K_{IE} \cdot \underline{\mathfrak{u}}_E^k \\
u_{I,i}^k &:= u_{I,i}^{k-1} + d_{I,i} \cdot \left(r_{I,i} - \sum_{j=N_E+1}^{i-1} K_{I,ij} \cdot u_{I,j}^k - \sum_{j=i}^{N} K_{I,ij} \cdot u_{I,j}^{k-1} \right) \\
\underline{w}_I &:= \underline{r}_I - K_I \cdot \underline{u}_I^k \\
\sigma &:= (\underline{\mathfrak{w}}, \underline{\mathsf{r}})
\end{aligned}
$$

end

Algorithmus 5.11: Paralleler Gauß-Seidel-Blocklöser vorwärts

Berücksichtigt man, daß die Crosspoints (”V”) und die Interfacedaten (”E”) in jedem Falle getrennt akkumuliert werdem, dann wird pro Iterationsschritt nur genausoviel Kommunikation wie beim ω-Jacobi Verfahren (Algorithmus 5.6) 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 K_E und K_V noch in den Matrix-mal-Vektor Operationen benötigt werden.

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

Aufgabe Ü5.12: Geben Sie eine Variante von Algorithmus 5.11 an, welche ohne die doppelte Abspeicherung von K_E und K_V auskommt !

5.5 Unvollständige Zerlegungen

Schon seit Jahrzehnten wird zum Lösen von (5.1) die Matrix K in eine obere Dreiecksmatrix $\widetilde{U}$ und eine untere Dreiecksmatrix $\widetilde{L}$ zerlegt. Während die Auflösung der resultierenden, gestaffelten Gleichungssysteme mit diesen Dreiecksmatrizen sehr schnell ausführbar ist, explodiert die Rechenzeit für die Faktorisierung

$$K \;=\; \widetilde{L} \cdot \widetilde{U} \ , \tag{5.4}$$

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 Gründen wird versucht, die Matrix K in die Dreiecksmatrizen L und U mit dem gleichen Besetztheitsmuster zu zerlegen. Die Gleichung (5.4) gilt dadurch nicht mehr, zur exakten Beschreibung muß noch eine Restmatrix P in Betracht gezogen werden :

$$K \;=\; L \cdot U \,-\, P \ . \tag{5.5}$$

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).

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 (5.5) für nichtsymmetrische Matrizen K ohne Fill-In, d.h. die klassische ILU(0) Faktorisierung und den entsprechenden Auflösungsschritt $L \cdot U \cdot \underline{w} = \underline{r}$.

5.5.1 Der serielle Algorithmus

Unter Benutzung der Blockstruktur der Matrix und Vektoren in (4.2) aus Abschnitt 4.3.1 schreiben wir die blockweise LU-Zerlegung mit einer typlosen Matrix K und typlosen Vektoren auf.

$$
\begin{array}{|ll|}
\hline
\text{Bestimme} & \\
L_V, U_V & K_V \;=\; L_V \cdot U_V - P_V \\
L_{EV} & K_{EV} \;=\; L_{EV} \cdot U_V - P_{EV} \\
L_{IV} & K_{IV} \;=\; L_{IV} \cdot U_V - P_{IV} \\
U_{VE} & K_{VE} \;=\; L_V \cdot U_{VE} - P_{VE} \\
U_{VI} & K_{VI} \;=\; L_V \cdot U_{VI} - P_{VI} \\
\hline
 & K_E \;:=\; K_E - L_{EV} \cdot U_{VE} \\
 & K_{EI} \;:=\; K_{EI} - L_{EV} \cdot U_{VI} \\
 & K_{IE} \;:=\; K_{IE} - L_{IV} \cdot U_{VE} \\
 & K_I \;:=\; K_I - L_{IV} \cdot U_{VI} \\
\hline
\text{Bestimme} & \\
L_E, U_E & K_E \;=\; L_E \cdot U_E - P_E \\
L_{IE} & K_{IE} \;=\; L_{IE} \cdot U_E - P_{IE} \\
U_{EI} & K_{EI} \;=\; L_E \cdot U_{EI} - P_{EI} \\
\hline
 & K_I \;:=\; K_I - L_{IE} \cdot U_{EI} \\
\hline
\text{Bestimme} & \\
L_I, U_I & K_I \;=\; L_I \cdot U_I - P_I \\
\hline
\end{array}
$$

Algorithmus 5.12: Serielle Block-ILU-Faktorisierung

Die Restmatrizen P dienen nur der sauberen Notation des Algorithmus, in der praktischen Rechnung werden sie weggelassen. Innerhalb der einzelnen Blöcke wird die punktweise ILU verwandt.

$$
\begin{array}{|llll|}
\hline
\text{I)} & \text{Solve } L \cdot \underline{u} \;=\; \underline{r} & & \\
 & \underline{u}_V & := & L_V^{-1} \underline{r}_V \\
 & \underline{u}_E & := & L_E^{-1}(\underline{r}_E - L_{EV}\underline{u}_V) \\
 & \underline{u}_I & := & L_I^{-1}(\underline{r}_I - L_{IE}\underline{u}_E - L_{IV}\underline{u}_V) \\
\text{II)} & \text{Solve } U \cdot \underline{w} \;=\; \underline{u} & & \\
 & \underline{w}_I & := & U_I^{-1} \underline{u}_I \\
 & \underline{w}_E & := & U_E^{-1}(\underline{u}_E - U_{EI}\underline{w}_I) \\
 & \underline{w}_V & := & U_V^{-1}(\underline{u}_V - U_{VE}\underline{w}_E - U_{VI}\underline{w}_I) \\
\hline
\end{array}
$$

Algorithmus 5.13: Serieller blockweiser Auflösungsschritt $L \cdot U \cdot \underline{w} = \underline{r}$

5.5.2 Die parallele ILU-Faktorisierung

Die erste Idee zur Parallelisierung der ILU ist die einfache Übertragung von Algorithmus 5.12 auf eine akkumulierte Matrix $\mathfrak{K}$. Natürlicherweise sind $\mathfrak{L}$ und $\mathfrak{U}$ ebenfalls akkumuliert. Jedoch müssen nun wegen (4.9) und (4.10) die Forderungen 3a und 3b an die Vernetzung aus Abschnitt 4.3.1 erfüllt sein.

Start	$\mathfrak{K} = \sum_{i=1}^{P} A_i^T \mathsf{K}_i A_i$	
Determine		Why parallel ?
$\mathfrak{L}_V, \mathfrak{U}_V$	$\mathfrak{K}_V = \mathfrak{L}_V \cdot \mathfrak{U}_V - P_V$	mesh $\to$ diagonal matrix
$\mathfrak{L}_{EV}$	$\mathfrak{K}_{EV} = \mathfrak{L}_{EV} \cdot \mathfrak{U}_V - P_{EV}$	DD, mesh $\to$ block matrices
L_{IV}	$K_{IV} = L_{IV} \cdot \mathfrak{U}_V - P_{IV}$	DD, mesh $\to$ block matrices
$\mathfrak{U}_{VE}$	$\mathfrak{K}_{VE} = \mathfrak{L}_V \cdot \mathfrak{U}_{VE} - P_{VE}$	DD, mesh $\to$ block matrices
U_{VI}	$\mathfrak{K}_{VI} = \mathfrak{L}_V \cdot U_{VI} - P_{VI}$	DD, mesh $\to$ 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 $\to$ block matrices
L_{IE}	$K_{IE} = L_{IE} \cdot \mathfrak{U}_E - P_{IE}$	DD $\to$ block matrices
U_{EI}	$K_{EI} = \mathfrak{L}_E \cdot U_{EI} - P_{EI}$	DD $\to$ 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 $\to$ block matrices

Algorithmus 5.14: Parallelisierte $\mathfrak{LU}$-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 die Akkumulation vor der Faktorisierung ausgeführt werden.
In 3D kommt noch ein Block mit den Faces "F" hinzu. Die entsprechende Faktorisierung ist analog, jedoch müssen an die Vernetzung weitere Forderungen gestellt werden.

Betrachtet man den Auflösungsschritt $\underline{\mathfrak{w}} = \mathfrak{U}^{-1} \cdot \mathfrak{L}^{-1} \cdot \underline{\mathsf{r}}$ näher und trägt den mit akkumulierten Matrizen erlaubten Matrix-Vektor-Multiplikationen (4.9) und (4.10) Rechnung, so sieht man aus Gleichung (4.13a), daß 3 Vektortypumwandlungen auftreten müssen.

I)	$\underline{\mathfrak{r}} := \sum_{i=1}^{P} A_i^T \underline{\mathsf{r}}_i$	IV)	$\underline{w}_I := U_I^{-1} \underline{u}_I$
II)	$\underline{\mathfrak{u}}_V := \mathfrak{L}_V^{-1} \underline{\mathfrak{r}}_V$		$\underline{\mathsf{w}}_E := \mathfrak{U}_E^{-1}(\underline{\mathsf{u}}_E - U_{EI}\underline{w}_I)$
	$\underline{\mathfrak{u}}_E := \mathfrak{L}_E^{-1}(\underline{\mathfrak{r}}_E - \mathfrak{L}_{EV}\underline{\mathfrak{u}}_V)$		$\underline{\mathsf{w}}_V := \mathfrak{U}_V^{-1}(\underline{\mathsf{u}}_V - \mathfrak{U}_{VE}\underline{\mathsf{w}}_E - U_{VI}\underline{w}_I)$
	$\underline{u}_I := L_I^{-1}(\underline{r}_I - L_{IE}\underline{\mathfrak{u}}_E - L_{IV}\underline{\mathfrak{u}}_V)$		
III)	$\underline{\mathsf{u}} := R^{-1}\underline{\mathfrak{u}}$	V)	$\underline{\mathfrak{w}} := \sum_{i=1}^{P} A_i^T \underline{\mathsf{w}}_i$

Algorithmus 5.15: Paralleler Auflösungsschritt $\mathfrak{L} \cdot \mathfrak{U} \cdot \underline{\mathfrak{w}} = \underline{\mathsf{r}}$

Die Diagonalmatrix R mit der Anzahl der am Knoten anliegenden Teilgebiete wurde in (4.4) definiert.

Verbesserung : Der Nachteil von Algorithmus 5.15 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 ILU als Vorkonditionierer doppelten Kommunikationsaufwand gegenüber dem ω-Jacobi-Verfahren.

$\Longrightarrow$ Ein Auflösungsschritt $\underline{\mathfrak{w}} = \mathfrak{L}^{-1} \cdot \mathfrak{U}^{-1} \cdot \underline{\mathsf{r}}$ würde laut (4.12a) nur eine Vektortypumwandlung mit einer Akkumulation benötigen.

$\downarrow$

5.5.3 Die parallele IUL-Faktorisierung

Zur Überwindung der doppelten Kommunikation in Algorithmus 5.15 wird eine unvollständige $\mathfrak{U}\mathfrak{L}$-Faktorisierung benutzt.

Der Durchlaufsinn kehrt sich gegenüber der ILU-Faktorisierung um. Dadurch werden die redundant gespeicherten Matrizen $\mathfrak{K}_V$, $\mathfrak{K}_E$, $\mathfrak{K}_E$, $\mathfrak{K}_{EV}$ und $\mathfrak{K}_{VE}$ während der Faktorisierung lokal verändert, so daß ihre Akkumulation erst kurz vor der entsprechenden Faktorisierung ausgeführt werden darf !

Start	K	
Determine		Why parallel ?
$U_I,\ L_I$	$K_I \ = \ U_I \cdot L_I - P_I$	DD $\to$ blocks matrices
L_{IE}	$K_{IE} \ = \ U_I \cdot L_{IE} - P_{IE}$	DD $\to$ blocks matrices
L_{IV}	$K_{IV} \ = \ U_I \cdot L_{IV} - P_{IV}$	DD $\to$ blocks matrices
U_{EI}	$K_{EI} \ = \ U_{EI} \cdot L_I - P_{EI}$	DD $\to$ blocks matrices
U_{VI}	$K_{VI} \ = \ U_{VI} \cdot L_I - P_{VI}$	DD $\to$ blocks matrices
Modify		
	$\mathsf{K}_E \ := \ \mathsf{K}_E - U_{EI} \cdot L_{IE}$	same matrix type
	$\mathsf{K}_{EV} \ := \ \mathsf{K}_{EV} - U_{EI} \cdot L_{IV}$	same matrix type
	$\mathsf{K}_{VE} \ := \ \mathsf{K}_{VE} - U_{VI} \cdot L_{IE}$	same matrix type
	$\mathsf{K}_V \ := \ \mathsf{K}_V - U_{VI} \cdot L_{IV}$	same matrix type
Accumulate	$\mathfrak{K}_E,\ \mathfrak{K}_{EV},\ \mathfrak{K}_{VE}$	
e.g.	$\mathfrak{K}_{EV} \ := \ \sum\limits_{i=1}^{P} A_{E,i}^T \mathsf{K}_{EV,i} A_{V,i}$	—
Determine		
$\mathfrak{U}_E,\ \mathfrak{L}_E$	$\mathfrak{K}_E = \mathfrak{U}_E \cdot \mathfrak{L}_E - P_E$	DD, mesh $\to$ blocks matrices
$\mathfrak{U}_{VE}$	$\mathfrak{K}_{VE} = \mathfrak{U}_{VE} \cdot \mathfrak{L}_E - P_{VE}$	DD, mesh $\to$ blocks matrices
$\mathfrak{L}_{EV}$	$\mathfrak{K}_{EV} = \mathfrak{U}_E \cdot \mathfrak{L}_{EV} - P_{EV}$	DD, mesh $\to$ blocks matrices
Modify		
	$\mathsf{K}_V \ := \ \mathsf{K}_V - \mathfrak{U}_{VE} \cdot R_E^{-1} \cdot \mathfrak{L}_{EV}$	same matrix type (4.13a)
Accumulate	$\mathfrak{K}_V$	—
Determine		
$\mathfrak{U}_V,\ \mathfrak{L}_V$	$\mathfrak{K}_V = \mathfrak{U}_V \cdot \mathfrak{L}_V - P_V$	mesh $\to$ diagonal matrix

Algorithmus 5.16: Parallelisierte IUL-Faktorisierung

Der Auflösungsschritt ist nur noch eine Anwendung von (4.12a).

I) $\underline{u}_I := U_I^{-1} \underline{r}_I$ $\underline{\mathsf{u}}_E := \mathfrak{U}_E^{-1}(\underline{\mathsf{r}}_E - U_{EI}\underline{u}_I)$ $\underline{\mathsf{u}}_V := \mathfrak{U}_V^{-1}(\underline{\mathsf{r}}_V - \mathfrak{U}_{VE}\underline{\mathsf{u}}_E - U_{VI}u_I)$ II) $\underline{\mathfrak{u}} := \sum\limits_{i=1}^{P} A_i^T \underline{\mathsf{u}}_i$	III) $\underline{\mathfrak{w}}_V := \mathfrak{L}_V^{-1}\underline{\mathfrak{u}}_V$ $\underline{\mathfrak{w}}_E := \mathfrak{L}_E^{-1}(\underline{\mathfrak{u}}_E - \mathfrak{L}_{EV}\underline{\mathfrak{w}}_V)$ $\underline{w}_I := L_I^{-1}(\underline{u}_I - L_{IE}\underline{\mathfrak{w}}_E$ $- L_{IV}\underline{\mathfrak{w}}_V)$

Algorithmus 5.17: Paralleler Auflösungsschritt $\mathfrak{U} \cdot \mathfrak{L} \cdot \underline{\mathfrak{w}} = \underline{\mathsf{r}}$

5.5.4 Vergleich von ILU- und IUL-Faktorisierung

Die Lösungsansätze in den beiden vorangegangenen Abschnitten unterscheiden sich kaum bzgl. der Arithmetik, dagegen etwas im Kommunikationsverhalten, Tabelle 5.1 faßt das Kommunikationsverhalten zusammen.

	ILU : $\mathfrak{L} \cdot \mathfrak{U}$	IUL : $\mathfrak{U} \cdot \mathfrak{L}$
Vorbereitung	Matrixakkumulation	—
Faktorisierung	—	Matrixakkumulation
Auflösung	$2 \times$ Vektorakkumulation	$1 \times$ Vektorakkumulation

Tabelle 5.1: Kommunikation in ILU und IUL

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 5.16 und 5.14 identisch. Entsprechend der FE-Diskretisierung liegt die Steifigkeitsmatrix K in jedem Falle verteilt gespeichert vor, so daß diese Kommunikation nicht eingespart werden kann (Abschnitt 4.3.1).

Mit Hinblick auf die Kommunikation ist die IUL-Faktorisierung im parallelen Fall zu empfehlen.

5.5.5 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 Auflösungschritt analog zu Algorithmus 5.17 ist, muß man bei der Faktorisierung etwas aufpassen.

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		
	$\mathsf{K}_E \;:=\; \mathsf{K}_E - U_{EI} \cdot L_{IE}$	same matrix type
	$\mathsf{K}_{EV} \;:=\; \mathsf{K}_{EV} - U_{EI} \cdot L_{IV}$	same matrix type
	$\mathsf{K}_{VE} \;:=\; \mathsf{K}_{VE} - U_{VI} \cdot L_{IE}$	same matrix type
	$\mathsf{K}_V \;:=\; \mathsf{K}_V - U_{VI} \cdot L_{IV}$	same matrix type
	$\vdots$	
Accumulate	$\mathfrak{K}_F$, $\mathfrak{K}_{FE}$, $\mathfrak{K}_{EF}$, $\mathfrak{K}_{FV}$, $\mathfrak{K}_{VF}$	
e.g.	$\mathfrak{K}_{FV} \;:=\; \sum_{i=1}^{P} A_{F,i}^T \mathsf{K}_{FV,i} A_{V,i}$	—
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	$\mathsf{K}_E \;:=\; \mathsf{K}_E - \mathfrak{U}_{EF} \cdot R_F^{-1} \cdot \mathfrak{L}_{FE}$	same matrix type
	$\mathsf{K}_{EV} \;:=\; \mathsf{K}_{EV} - \mathfrak{U}_{EF} \cdot R_F^{-1} \cdot \mathfrak{L}_{FV}$	same matrix type
	$\mathsf{K}_{VE} \;:=\; \mathsf{K}_{VE} - \mathfrak{U}_{VF} \cdot R_F^{-1} \cdot \mathfrak{L}_{FE}$	same matrix type
	$\mathsf{K}_V \;:=\; \mathsf{K}_V - \mathfrak{U}_{VF} \cdot R_F^{-1} \cdot \mathfrak{L}_{FV}$	same matrix type
Accumulate	$\mathfrak{K}_E$, $\mathfrak{K}_{EV}$, $\mathfrak{K}_{VE}$	
e.g.	$\mathfrak{K}_{EV} \;:=\; \sum_{i=1}^{P} A_{E,i}^T \mathsf{K}_{EV,i} A_{V,i}$	—
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	$\mathsf{K}_V \;:=\; \mathsf{K}_V - \mathfrak{U}_{VE} \cdot R_E^{-1} \cdot \mathfrak{L}_{EV}$	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

Algorithmus 5.18: Parallelisierte IUL-Faktorisierung in 3D

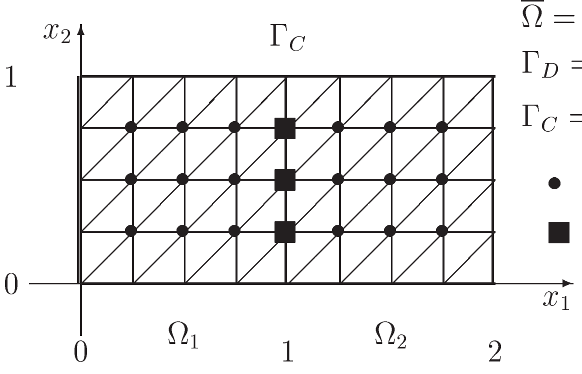
5.5.6 Die IUL-Faktorisierung mit reduziertem Besetztheitsmuster

Im 3d-Fall bzw. bei Verwendung von bilinearen/quadratischen/... Elementen in 2d können die Matrixanforderungen (a) und (b) aus Abschnitt 4.3.1 nicht mehr eingehalten werden. Hier muß an Stelle der Originalmatrix $\mathfrak{K}$ die die Forderungen erfüllende Matrix $\widetilde{\mathfrak{K}}$ 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 !
Numerische Tests zeigen die Berechtigung dieses Vorgehens.

5.6 Der Schurkomplement CG

5.6.1 Das Schurkomplement

Der Begriff des Schurkomplements geht auf den Mathematiker Isaai Schur[1] (1894-1939 in Deutschland arbeitend) zurück. Wir führen den Begriff von einem anderen Ansatz her ein.



$\overline{\Omega} = \overline{\Omega}_1 \cup \overline{\Omega}_2$

$\Gamma_D = \partial\Omega$

$\Gamma_C = \{\, (1, y) : 0 < y < 1 \,\}$

• ⟷ "I"

■ ⟷ "C"

Abbildung 5.5: 2 Superelemente (=Teilgebiete) mit Diskretisierung

Die FE-Knotenverteilung in Abb.5.5 induziert im zu lösenden Gleichungssystem (5.1) eine Blockstruktur :

$$\begin{pmatrix} K_C & K_{CI,1} & K_{CI,2} \\ K_{IC,1} & K_{I,1} & 0 \\ K_{IC,2} & 0 & K_{I,2} \end{pmatrix} \cdot \begin{pmatrix} \underline{u}_C \\ \underline{u}_{I,1} \\ \underline{u}_{I,2} \end{pmatrix} = \begin{pmatrix} \underline{f}_C \\ \underline{f}_{I,1} \\ \underline{f}_{I,2} \end{pmatrix} . \tag{5.6}$$

Eine Gaußelimination des oberen Dreiecks der Matrix in (5.6) ergibt das gestaffelte Gleichungssystem

$$\begin{pmatrix} S_C & 0 \\ K_{IC} & K_I \end{pmatrix} \cdot \begin{pmatrix} \underline{u}_C \\ \underline{u}_I \end{pmatrix} = \begin{pmatrix} \underline{g}_C \\ \underline{f}_I \end{pmatrix} , \tag{5.7}$$

mit dem Schurkomplement

$$\begin{aligned} S_C &= K_C - K_{CI} K_I^{-1} K_{IC} \qquad (5.8) \\ &= \left(K_{C,1} - K_{CI,1} K_{I,1}^{-1} K_{IC,1}\right) + \left(K_{C,2} - K_{CI,2} K_{I,2}^{-1} K_{IC,2}\right) \end{aligned}$$

und der modifizierten rechten Seite

$$\underline{g}_C = \underline{f}_C - K_{CI} K_I^{-1} \underline{f}_I \ .$$

[1] 1875 (Mohilev am Dnjepr) - 1941 (Tel Aviv)

Hierbei ist $K_{IC} = \binom{K_{IC,1}}{K_{IC,2}}$ 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 (5.7) 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-ger 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
    $\underline{x}_{I,s} := K_{I,s}^{-1} \cdot \left(\underline{f}_{I,s} - K_{IC,s} \cdot \underline{x}_C\right)$
  end

Algorithmus 5.19: Element-by-Element Methode

Da das Schurkomplement nur in den seltensten Fällen explizit angebbar ist, löst man oft das reduzierte Gleichungsystem (mit vollbesetzter Matrix !) in II) mittels eines geeigneten Iterationsverfahrens. Der Vorteil besteht darin, daß die in jedem Iterationsverfahren notwendige Matrix-mal-Vektor Multiplikation ausgeführt werden kann :

$$\begin{aligned} \underline{r}_C &:= S_C \underline{w}_C \\ &= \sum_{s=1}^{P} \left(K_{C,s} - K_{CI,s} K_{I,s}^{-1} K_{IC,s}\right) \cdot \underline{w}_{C,s} \ . \end{aligned}$$

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.

5.6.2 Der parallele Schurkomplement-CG

Falls die Matrix in (5.1) symmetrisch und positiv definit ist, so ist auch das entsprechende Schurkomplement symmetrisch und positiv definit. Damit ist

Zeile II) des Algorithmus 5.19 mittels eines CG iterativ lösbar.

Aufgabe Ü5.13: 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 [Dry84, BPS89, TCK92] zu finden. Ein Ansatz über BEM und Multigridmethoden ist in [CKL96] zu finden.

5.7 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 $\mathfrak{T}_1 \subset \mathfrak{T}_2 \subset \cdots \subset \mathfrak{T}_\ell$ gelten.

Die gegebene Differentialgleichung wird auf jedem Gitter $\mathfrak{T}_k$ diskretisiert so daß einen Folge von Gleichungssystemen

$$K_k \cdot \underline{u}_k \;=\; \underline{f}_k \tag{5.9}$$

mit den jeweiligen dünnbesetzten, symmetrischen und positiv definiten Steifigkeitsmatrizen K_k entsteht.

Eine Analyse des (nichtoptimalen) Konvergenzverhaltens der Iterationsverfahren in den Abschnitten 5.3 und 5.4 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 gängigen Verfahren.

$\Longrightarrow$ Man braucht ein Verfahren, welches die niedrigfrequenten Fehleranteile e_{low} kostengünstig reduziert, bei Beibehaltung der schnellen Reduktion hochfrequenter Fehleranteile e_{high}.

$\Longrightarrow$ *Zweigitteridee :*

Reduziere e_{high} auf dem Gitter $\mathfrak{T}_\ell$ (Glättungsschritt) projiziere den Restfehler auf das gröbere Gitter $\mathfrak{T}_{\ell-1}$, löse dort exakt, interpoliere die erhaltene Korrektur (Defektkorrektur) wieder auf das Feingitter $\mathfrak{T}_\ell$ und addiere sie zur bereits erhaltenen Lösung. Oft werden hinterher die verbliebenen hochfrequenten Fehleranteile nochmals geglättet.

- Die *Multigrididee* ersetzt nun den exakten Löser bei der Defektkorrektur wiederum durch eine Zweigitteridee auf dem Gitter $\ell - 1$ bis hinunter zum gröbsten Gitter $\mathfrak{T}_1$, welches exakt gelöst wird.

5.7.1 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 5.3 - 5.5 gewählt werden.

if $(k == 1)$ then			
Solve $K_1 \cdot \underline{u}_1 \;=\; \underline{f}_1$			Grobgitterlöser
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			

Algorithmus 5.20: Serielles 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 $I_k^{k-1} = \left(I_{k-1}^k\right)^T$ und $S_{\text{pre}} = \left(S_{\text{post}}\right)^T$ mit $\nu_{\text{pre}} = \nu_{\text{post}}$ gelten. Das Defektsystem auf dem gröbsten Gitter wird direkt gelöst.

5.7.2 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 4.3.1 bereits auf dem gröbsten Gitter $\mathfrak{T}_1$ benutzen, so bleibt diese Aufteilung auf sämtlichen feineren Gittern erhalten.

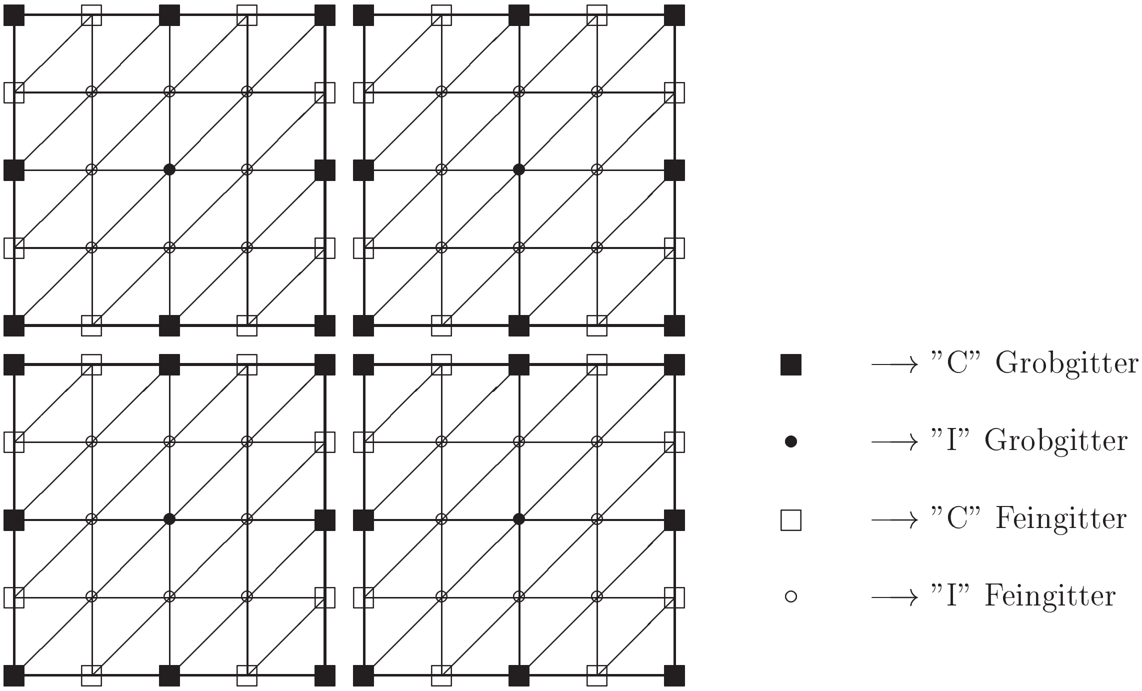


Abbildung 5.6: Nichtüberlappende Elemente bei zwei Gittern

Die Steifigkeitsmatrizen K_k seien verteilt gespeichert. Aus den Erfahrungen der vorangegangenen Abschnitten (insbesondere 5.1) ist in Alg. 5.20 folgende Zuordnung der Vektoren zu den zwei Typen a priori erkennbar:

- Akkumulierte Vektoren : $\underline{\widehat{\mathfrak{u}}}_k$, $\underline{\widetilde{\mathfrak{u}}}_k$, $\underline{\widehat{\mathfrak{w}}}_k$.
- Verteilte Vektoren : $\underline{\mathsf{f}}_k$, $\underline{\mathsf{d}}_k$.
- Noch frei: $\underline{u}_k$, $\underline{w}_{k-1}$, $\underline{w}^0_{k-1}$, $\underline{d}_{k-1}$.

Die Interpolation

Betrachtet man Abb. 5.6 und die entsprechende, aus der Netzverfeinerung hervorgehende Interpolation, so ist für die 3 Knotenklassen V, E, I folgendes erkennbar :

1. Die Interpolation $I^k_{V,k-1}$ innerhalb der Crosspoints ist die (evtl. skalierte) Identität I .

2. Crosspoints (V) gehören vollständig zum gröbsten Gitter und werden somit nie durch E oder I interpoliert
$\Longrightarrow I^k_{VE,k-1} = I^k_{VI,k-1} = 0$.

3. Die Interpolationmatrix auf den Kanten zerfällt in Blöcke
$\Longrightarrow I^k_{E,k-1} = \text{blockdiag}\{I^k_{E_j,k-1}\}_{i=\overline{1,\text{NumEdges}}}$.

4. Kantenknoten werden nie durch innere Knoten interpoliert
$\Longrightarrow I^k_{EI,k-1} = 0$.

5. Offensichtlich zerfällt auch die Interpolationmatrix auf den inneren Knoten (I) in Blöcke
$\Longrightarrow I^k_{I,k-1} = \text{blockdiag}\{I^k_{I_s,k-1}\}_{s=\overline{1,P}}$.

Somit besitzt die Interpolationsmatrix I^k_{k-1} eine Blockstruktur, in welcher nur das untere Blockdreieck ungleich 0 ist. Die Blockstruktur ist insbesondere so geartet, daß (4.9) anwendbar wird.

$\Longrightarrow$ Interpolation ist akkumuliert : $\mathfrak{I}^k_{k-1}$.

$\Longrightarrow$ Vektor $\underline{\mathfrak{w}}_{k-1}$ 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. $I^{k-1}_k = \left(I^k_{k-1}\right)^T$, ergibt sich die Dreiecksstruktur in (4.10).

$\Longrightarrow$ Restriktion ist akkumuliert : $\mathfrak{I}^{k-1}_k$.

$\Longrightarrow$ Vektor $\underline{\mathsf{d}}_{k-1}$ ist verteilt gespeichert, $\underline{\mathsf{d}}_k$ sowieso.

Hier ist eine Abspeicherung als verteilte Matrix wiederum prinzipiell möglich, würde jedoch vor der Restriktion eine Akkumulation von $\underline{\mathsf{d}}_k$ 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 (5.3 - 5.5) benötigen als Input stets eine akkumulierte Startlösung $\underline{\mathfrak{u}}^0$, eine verteilt gespeicherte rechte Seite $\underline{\mathsf{f}}$ und eine verteilte Matrix K. Die iterierte Lösung $\underline{\mathfrak{u}}$ ist stets akkumuliert. Innerhalb des Iterationsverfahrens ist in jedem Falle mindestens eine Akkumulation nötig.

$\Longrightarrow$ Vektor $\underline{\mathfrak{u}}_k$ ist akkumuliert, $\widehat{\underline{\mathfrak{u}}}_k$ und $\widetilde{\underline{\mathfrak{u}}}_k$ sowieso.

$\Longrightarrow$ Vektor $\underline{\mathfrak{w}}_k^0$ ist akkumuliert.

Kommunikation innerhalb der Glättung !

Der Grobgitterlöser

Das eigentliche Parallelisierungsproblem in (Multilevel- und) Mehrgitterverfahren ist das exakt zu lösende Grobgitterproblem

$$\sum_{s=1}^{P} A_{s,1}^T \mathsf{K} A_{s,1}\, \underline{\mathfrak{u}}_1 \quad = \quad \sum_{s=1}^{P} A_{s,1}^T\, \underline{\mathsf{f}}_{s,1} \quad . \tag{5.10}$$

Folgende Lösungsansätze sind denkbar und werden verwendet :

1. *Direkter Löser - seriell*

   Gröbstes Gitter ist so klein, daß es ein Prozessor $\mathbb{P}_0$ zusätzlich speichern kann.
   - Assemblierung K_1 (Reduce).
   - Assemblierung $\underline{\mathsf{f}}_1$ (Reduce).
   - Prozessor $\mathbb{P}_0$ löst $\mathfrak{K}_1 \cdot \underline{\mathfrak{u}}_1 = \underline{\mathfrak{f}}_1$.
   - Verteilen von $\underline{\mathfrak{u}}_1$ (Scatter).

   Diese Vorgehen ist allerdings rein sequentiell !!

2. *Direkter Löser - parallel*

   Auch das gröbste Gitter wird verteilt gespeichert und das Grobgittersystem direkt parallel gelöst. Im Auflösungsprozeß treten Loadinbalancen auf.

3. *Iterative Löser - parallel*

   Hier wird keine Assemblierung von K_1 und $\underline{\mathsf{f}}_1$ benötigt, jedoch tritt in jedem Iterationsschritt mind. eine Akkumulation auf (siehe 5.1 - 5.6).

   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 gröbste 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 !

5.7.3 Der parallele Algorithmus

if $(k == 1)$ then

Solve $\sum_{s=1}^{P} A_{s,1}^T \mathsf{K} A_{s,1} \underline{\mathfrak{u}}_1$	$=$	$\sum_{s=1}^{P} A_{s,1}^T \underline{\mathsf{f}}_{s,1}$	Grobgitterlöser

else

$\widehat{\underline{\mathfrak{u}}}_k$	$:=$	$S_{\text{pre}}^{\nu_{\text{pre}}}(\mathsf{K}_k, \underline{\mathfrak{u}}_k, \underline{\mathsf{f}}_k)$	Vorglättung
$\underline{\mathsf{d}}_k$	$:=$	$\underline{\mathsf{f}}_k - \mathsf{K}_k \cdot \widehat{\underline{\mathfrak{u}}}_k$	Defektberechnung
$\underline{\mathsf{d}}_{k-1}$	$:=$	$\mathfrak{I}_k^{k-1} \cdot \underline{\mathsf{d}}_k$	Restriktion des Defektes
$\underline{\mathfrak{w}}_{k-1}^0$	$:=$	0	
$\underline{\mathfrak{w}}_{k-1}$	$:=$	$\text{PMGM}^{\gamma}(\mathsf{K}_{k-1}, \underline{\mathfrak{w}}_{k-1}^0, \underline{\mathsf{d}}_{k-1}, k-1)$	Defektsystem
$\underline{\mathfrak{w}}_k$	$:=$	$\mathfrak{I}_{k-1}^{k} \cdot \underline{\mathfrak{w}}_{k-1}$	Interpolation der Korrektur
$\widetilde{\underline{\mathfrak{u}}}_k$	$:=$	$\widehat{\underline{\mathfrak{u}}}_k + \underline{\mathfrak{w}}_k$	Addition der Korrektur
$\underline{\mathfrak{u}}_k$	$:=$	$S_{\text{post}}^{\nu_{\text{post}}}(\mathsf{K}_k, \widetilde{\underline{\mathfrak{u}}}_k, \underline{\mathsf{f}}_k)$	Nachglättung

endif

Algorithmus 5.21: Paralleles Multigrid : $\text{PMGM}(\mathsf{K}_k, \underline{\mathfrak{u}}_k, \underline{\mathsf{f}}_k, k)$

5.8 Vorkonditionierung Iterativer Verfahren

Die Typumwandlung $\underline{\mathfrak{w}} = \sum_{j=1}^{P} A_j^T \underline{\mathsf{r}}_j$ im CG (Abb. 5.2) als Matrix geschrieben, stellt sich als Typ-I Einheitsmatrix dar und ist somit äquivalent zu Gleichung (4.11b) 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 Typ-II zu einem Typ-I Vektor.

5.8.1 Typ-I Vorkonditionierer

Nunmehr läßt sich ein Typ-I Vorkonditionierer aus den Gleichungen (4.11b), (4.12a) und (4.13a) auswählen. Insbesondere letzterer ist besonders interessant.

Bemerkung : Die in den Abschnitten 5.3 bis 5.7 behandelten Iterationsverfahren sind jeweils als Vorkonditionierer im CG (spd) bzw. GMRES geeignet. Im Falle einer symmetrischen Steifigkeitsmatrix paßt die ASM-DD-Vorkonditionierung [HLM91] unter Benutzung von Schurkomplementvorkonditionierern [BPS89, TCK92] exakt ins Schema von Gleichung (4.12a).

5.8.2 Typ-II Vorkonditionierer

Möchte man eine Typ-II Vorkonditionierungsmatrix $\mathsf{C}^{-1} = \sum\limits_{i=1}^{p} A_i^T \mathsf{C}_i^{-1} A_i$ benutzen ist eine zweifache Vektortypumwandlung notwendig (Abschnitt 4.3.1). Deshalb sieht der Vorkonditionierungsschritt wie folgt aus :

$$\underline{\mathfrak{w}} \;=\; \mathsf{C}^{-1} \sum_{j=1}^{P} A_j^T \underline{\mathsf{r}}_j \;:=\; \sum_{i=1}^{P} A_i^T \left(\mathsf{C}_i^{-1} \cdot A_i \sum_{j=1}^{P} A_j^T \underline{\mathsf{r}}_j \right) \quad . \tag{5.11}$$

Die Matrix K ist nicht assembliert und daher stellen die Teilmatrizen 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 K_i führen !

Andererseits muß C_i nicht auf die gleiche Weise gewählt werden. Eine Möglichkeit besteht in der Akkumulation der Steifigkeitsmatrix K und der Wahl $\mathsf{C}_i = \mathfrak{K}_i$. Der lokale Vorkonditionierer C_i repräsentiert eine PDE 2. Ordnung im Gebiet $\widetilde{\Omega}_i$ mit homogenen Dirichlet Randbedingungen, wobei $\widetilde{\Omega}_i$ das Gebiet Ω_i vergrößert um die nächste Schicht von Elementen darstellt. Damit ist dieser Vorkonditionierer vom überlappenden Typ, siehe [SBG96].

Kapitel 6

Vektorisierung und Parallelisierung direkter Verfahren

6.1 Elimination durch Drehungsmatrizen

Zu lösen ist das lineare Gleichungssystem

$$A_{n\times n} \cdot \underline{x} \;=\; \underline{b} \;. \tag{6.1}$$

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

$$\begin{pmatrix} 1 & 0 & 0 & \cdots \\ -\frac{a_{21}}{a_{11}} & 1 & 0 & \cdots \\ 0 & 0 & 1 & \cdots \\ \vdots & \vdots & & \ddots \end{pmatrix} \cdot A \cdot \underline{x} \;=\; \begin{pmatrix} 1 & 0 & 0 & \cdots \\ -\frac{a_{21}}{a_{11}} & 1 & 0 & \cdots \\ 0 & 0 & 1 & \cdots \\ \vdots & \vdots & & \ddots \end{pmatrix} \cdot \underline{b} \tag{6.2}$$

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 viel Kommunikation bei Parallelrechnern mit verteiltem Speicher auf.

Ausweg ⇒ Elimination mittels *Givensrotation*

6.1.1 Die Givensrotation

Die Transformationsmatrix aus (6.2) wird leicht verändert (wiederum nur der erste Schritt)

$$\begin{pmatrix} c & s & 0 & \cdots \\ -s & c & 0 & \cdots \\ 0 & 0 & 1 & \cdots \\ \vdots & \vdots & & \ddots \end{pmatrix} \cdot A \cdot \underline{x} \;=\; \begin{pmatrix} c & s & 0 & \cdots \\ -s & c & 0 & \cdots \\ 0 & 0 & 1 & \cdots \\ \vdots & \vdots & & \ddots \end{pmatrix} \cdot \underline{b} \;, \tag{6.3}$$

mit dem Ziel

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

Die Wahl

$$s \;=\; \frac{a_{21}}{\sqrt{a_{11}^2 + a_{21}^2}} \qquad \text{und} \qquad c \;=\; \frac{a_{11}}{\sqrt{a_{11}^2 + a_{21}^2}} \tag{6.4}$$

entspricht durch die Normierung exakt den Einträgen $s = \sin\varphi$ und $c = \cos\varphi$ einer Drehungsmatrix.
In der Gaußelimination wurde die Größe $\frac{a_{21}}{a_{11}} = \tan\varphi$ verwendet.

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

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

Eine einzelne Givensrotation

$$\begin{aligned} \widehat{\underline{u}} &:= c \cdot \underline{u} + s \cdot \underline{v} \\ \widehat{\underline{v}} &:= c \cdot \underline{v} - s \cdot \underline{u} \end{aligned} \tag{6.5}$$

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

6.1.2 Givensrotation auf dem Vektorrechner

Durch die wechselseitige Datenabhängigkeit der Zeilen $\underline{u}$ und $\underline{v}$ in (6.5) kommt man auf dem Vektorrechner nicht ohne Zwischenspeicherung aus :

$$\begin{aligned} \underline{h} &:= -s \cdot \underline{u} \\ \underline{u} &:= c \cdot \underline{u} + s \cdot \underline{v} \\ \underline{v} &:= c \cdot \underline{v} + \underline{h} \end{aligned} \tag{6.6}$$

6.1.3 Givensrotation auf dem Parallelrechner

Bezeichne $g(k,s)$ die Givensrotation der Zeilen $k-1$ und k und der Spalte s . Hierbei tritt folgende Datenabhängigkeit auf :

$$
\boxed{
\begin{array}{llll}
g(k,s) & : \text{ falls} & a_{k_s} = 0 & \longrightarrow \quad \text{Return} \\
 & \text{sonst} & w := \sqrt{a_{k,s}^2 + a_{k-1,s}^2} & \\
 & & c := \dfrac{a_{k-1,s}}{w} \quad , & s := \dfrac{a_{k,s}}{w} \\
 & & a_{k-1,s} := w \quad , & a_{k,s} := 0 \\
 & & {\scriptstyle j=\overline{s+1,n}} : & h := -s a_{k-1,j} \\
 & & & a_{k-1,j} := c a_{k-1,j} + s a_{k,j} \\
 & & & a_{k,j} := c a_{k,j} + h
\end{array}
}
$$

Algorithmus 6.1: $g(k,s)$ - Givensrotation der Zeilen $k-1$ und k und der Spalte s

$$
\begin{tikzcd}
g(n,1) \arrow[d, "n-1"] \arrow[dr, "n"] & & & \\
g(n-1,1) \arrow[r, "n-1"] \arrow[d, "n-2"] & g(n,2) \arrow[d, "n-1"] \arrow[dr, "n"] & & \\
g(n-2,1) \arrow[r, "n-2"] \arrow[d, "n-3"] & g(n-1,2) \arrow[r, "n-1"] \arrow[d, "n-2"] & g(n,3) \arrow[d, "n-1"] \arrow[r, dash] & {} \\
\vdots & \vdots & \vdots & \\
g(2,1) & g(3,2) & g(4,3) &
\end{tikzcd}
$$

Abbildung 6.1: Statische Datenabhängigkeit Givensrotation

Die statische Datenabhängigkeit von Bild 6.1 wandeln wir durch sogenannte Marker $*$ in ein *dynamisches Datenflußkonzept* um, miteiner beliebigen Anzahl von Prozessoren.

Sei in $m(k,s)$ die Anzahl der Marken für den Prozeß $g(k,s)$.

$m(k,s) = 2 \qquad \Longrightarrow \qquad$ Prozeß $g(k,s)$ startet.

$$\begin{array}{lll} \text{Anfangswerte :} & m(n,1) := 2 & \\ & m(k,1) := 1 & k < n \\ & m(k,s) := 0 & s > 1 \end{array}$$

Bei *gemeinsamem Speicher* ergibt sich der Datenfluß in Bild 6.2

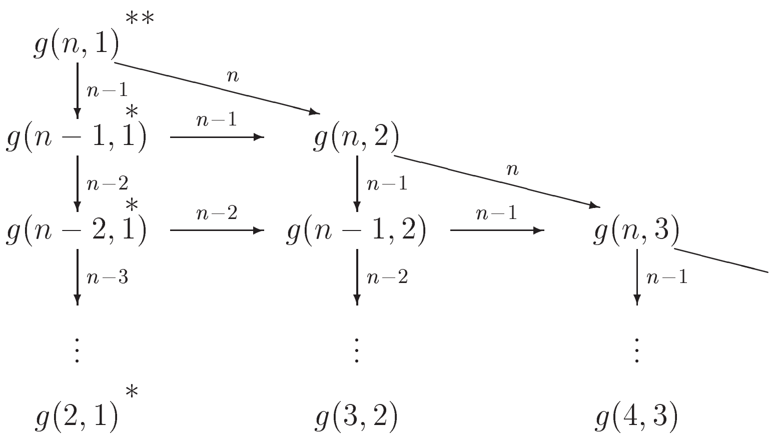


Abbildung 6.2: Datenfluß bei gemeinsamem Speicher : Givensrotation

Nach Abschluß von Prozeß $g(k,s)$ werden $m(k-1,s)$ und $m(k+1,s+1)$ jeweils um 1 erhöht.

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, was in Bild 6.3 dokumentiert wird.

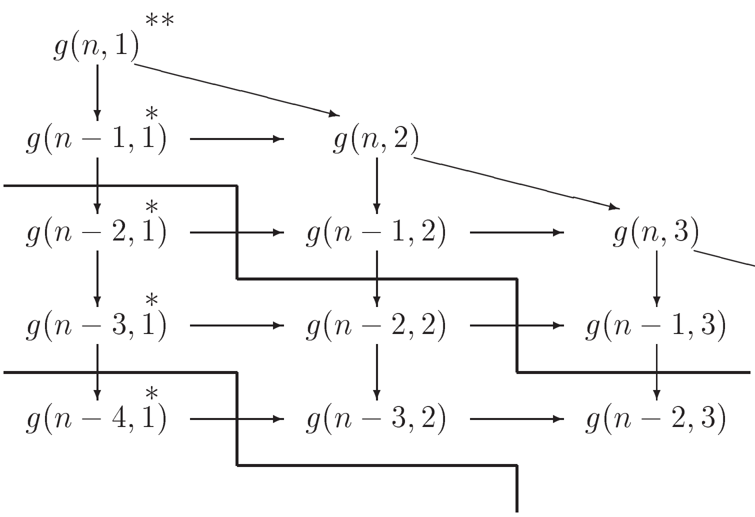


Abbildung 6.3: Datenfluß bei verteiltem Speicher : Givensrotation

Bemerkung : Eine analoge Aufteilung der Zeilen ist auch beim Gaußalgorithmus denkbar.

6.2 Die LU-Zerlegung

Sei $A_{n \times n}$ - vollbesetzt, zeilenweise gespeichert,

L - untere Dreiecksmatrix mit Normierung $\ell_{i,i} = 1$, spaltenweise gespeichert,

U - obere Dreiecksmatrix, zeilenweise gespeichert.

und es soll gelten $\sum\limits_{j=1}^{n} \ell_{i,j} \cdot u_{j,k} \;=\; a_{i,k}$.

DO $i \;=\; 1,\, n$

  $\ell_{i,i} \;:=\; 1$

  DO $k \;=\; 1,\, i-1$

$$\ell_{i,k} \;:=\; \Big(a_{i,k} - \sum_{j=1}^{k-1} \underbrace{\ell_{i,j}}_{\substack{i-te\\ Zeile}} \cdot \underbrace{u_{j,k}}_{\substack{k-te\\ Spalte}} \Big) \,/\, u_{k,k}$$

  END DO

  DO $k \;=\; 1,\, i$

$$u_{k,i} \;:=\; a_{k,i} - \sum_{j=1}^{k-1} \underbrace{\ell_{k,j}}_{\substack{k-te\\ Zeile}} \cdot \underbrace{u_{j,i}}_{\substack{i-te\\ Spalte}}$$

  END DO

END DO

Algorithmus 6.2: LU-Zerlegung ohne Pivotisierung - seriell

6.2.1 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.

$[\,\ell_{\mathbf{i},\mathbf{i}} := 1\,]_{\mathbf{i}=1,n}$
DO $i = 1, n$
  DO $k = 1, i-1$
    $h_k := [\,\ell_{i,\mathbf{j}} \cdot u_{\mathbf{j},k}\,]_{\mathbf{j}=1,k-1}$
  END DO
  $[\,\ell_{i,\mathbf{k}} := a_{i,\mathbf{k}} - h_{\mathbf{k}}\,]_{\mathbf{k}=1,i-1}$
  DO $k = 1, i-1$ — nicht $[\ell_{i,\mathbf{k}} := \ell_{i,\mathbf{k}} \cdot u_{\mathbf{k},\mathbf{k}}]_{\mathbf{k}=1,i-1}$,
    $\ell_{i,k} := \ell_{i,k}/u_{k,k}$ — da Stride von u gleich k sein müßte !
  END DO
  DO $k = 1, i$
    $h_k := [\,\ell_{k,\mathbf{j}} \cdot u_{\mathbf{j},i}\,]_{\mathbf{j}=1,k-1}$
  END DO
  $[\,u_{\mathbf{k},i} := \underbrace{a_{\mathbf{k},i}}_{Stride=n} - h_{\mathbf{k}}\,]_{\mathbf{k}=1,i}$ — A zeilenweise gespeichert !
END DO

Algorithmus 6.3: LU-Zerlegung ohne Pivotisierung - vektoriell

6.2.2 Parallelisierung der LU-Zerlegung

Im folgenden nehmen wir einen Parallelrechner mit *verteiltem Speicher* an.

Zur Vorbereitung der Parallelisierung benötigen wir einen anderen Algorithmus der LU-Zerlegung, wobei diesmal L spaltenweise, und U zeilenweise gespeichert sein soll. Eine Pivotisierung wird nicht betrachtet.

DO $i = 1, n$	
DO $k = i, n$	Bestimmung i-te Zeile von U
$u_{i,k} := a_{i,k}$	
END DO	
$\ell_{i,i} := 1$	
DO $k = i+1, n$	Bestimmung i-te Spalte von L
$\ell_{k,i} := a_{k,i}/u_{i,i}$	
END DO	
DO $k = i+1, n$	Transformation der Restmatrix
DO $j = i+1, n$	
$a_{k,j} := a_{k,j} - \ell_{k,i} \cdot u_{i,j}$	
END DO	
END DO	
END DO	

Algorithmus 6.4: Rang-r-Modifikation der LU-Zerlegung - seriell

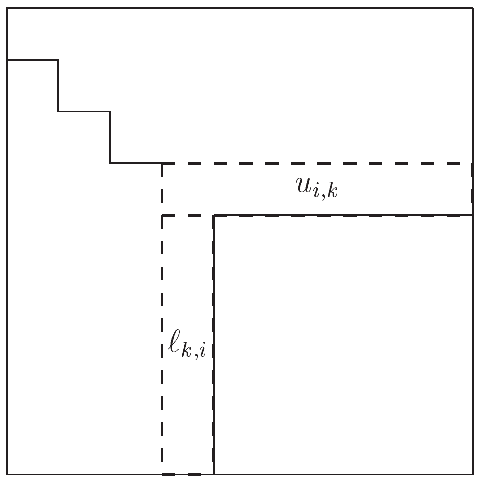


Abbildung 6.4: Illustration zur Rang-r-Modifikation

Zum Zwecke der Parallelisierung bietet sich die Blockvariante der Rang-r-Modifikation an. Hier bezeichnet $A_{k,i}$ die Zeilen und Spalten der Blockaufteilung der Matrix A.

DO $i = 1, n$

a) Faktorisierung von $A_{k,i}$ aus der Beziehung $A_{k,i} = L_{k,i} \cdot U_{i,i}$ $\quad (k=\overline{i,n})$

$\longrightarrow U_{i,i},\ L_{i,i},\ L_{i+1,i},\ \ldots,\ L_{n,i}$

b) Bestimme $U_{i,\ell}$ aus $A_{i,\ell} = L_{i,i} \cdot U_{i,\ell}$ $\quad (\ell=\overline{i+1,n})$

$\longrightarrow U_{i,i+1},\ \ldots,\ U_{i,n}$

c) Rang-r-Modifikation der Restmatrix $A_{k,\ell} := A_{k,\ell} - L_{k,i} \cdot U_{i,\ell}$ $\quad (k,\ell=\overline{i+1,n})$

END DO

Algorithmus 6.5: Blockvariante der Rang-r-Modifikation

Bei einer zeilen- bzw. spaltenweisen *Aufteilung der Matrix A* werden desto weniger der verfügbaren Prozessoren eingesetzt je kleiner die Restmatrix wird (siehe Bild 6.4).

$$\Downarrow$$

Gestreute Aufteilung von quadratischen Blockmatrizen (*square block scattered decomposition*), wie sie in der parallelisierten Version von ScaLAPACK benutzt wird.

Bsp.: *Gestreute Aufteilung*

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

5×5 Matrixblöcke

3×3 Prozessoren $\left(_{i=\overline{0,n}}\right)$

$A_{i,j} \longrightarrow \mathbb{P}_{(i-1) \mod P_x, (j-1) \mod P_y}$

Mittels der gestreuten Aufteilung der Matrix A kann nun die *Parallelisierung der Blockvariante* aufgeschrieben werden. *Bemerkung :* Vom Standpunkt der Implementierung empfiehlt sich hier eine nichtblockierende Kommunikation. In Richtung der Spalten und Zeilen ist auch eine Verteilung analog der Hypercubenumerierung denkbar.

6.3 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 Bild 6.5 Leider ist die in diese klassische Elimination nicht parallelisierbar.

$\mathbb{P}(A_{i,j})$ 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

Algorithmus 6.6: Blockweise Rang-r-Modifikation der LU-Zerlegung - parallel

6.3.1 Parallelisierung mittels zyklischer Reduktion

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

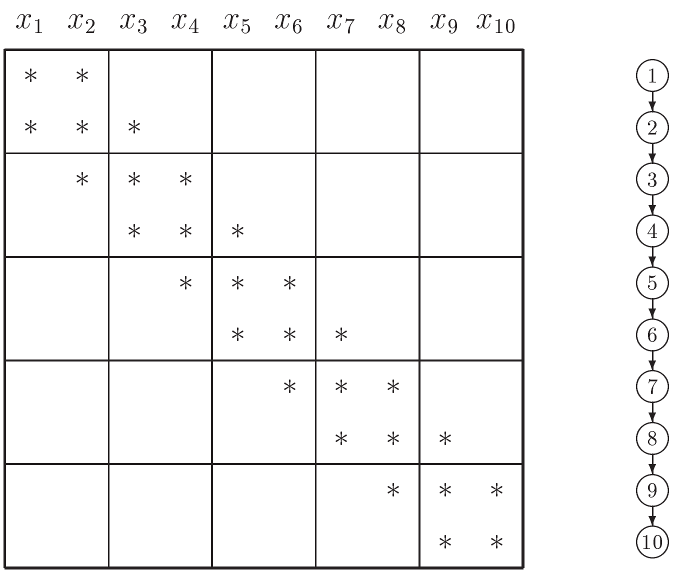


Abbildung 6.5: Tridiagonale Matrix und Klassischer Eliminationsbaum

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

Eliminiere alle x_i mit $i \mod 2 = 1$ aus dem GlS :

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

$$\Longrightarrow \begin{array}{ccccccc}
\widetilde{b}_2x_2 & + & \widetilde{c}_3x_4 & & & = & \widetilde{f}_2 \\
\widetilde{a}_2x_2 & + & \widetilde{b}_4x_4 & + & \widetilde{c}_5x_6 & = & \widetilde{f}_4 \\
 & & \widetilde{a}_4x_4 & + & \widetilde{b}_6x_6 & = & \widetilde{f}_3
\end{array}$$

Eliminiere alle x_i mit $i \mod 4 = 2$ $\quad\Longrightarrow\quad$ $\widehat{b}_4x_4 = \widehat{f}_4$

Allg.:

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

Der in Bild 6.6 dargestellte Eliminationsbaum (und damit die zyklische Reduktion)

$$
\begin{array}{cccccccccc}
x_1 & x_3 & x_5 & x_7 & x_9 & x_2 & x_6 & x_{10} & x_4 & x_8 \\
\hline
* & & & & & * & & & & \\
& * & & & & * & & & * & \\
& & * & & & & * & & * & \\
& & & * & & & * & & & * \\
& & & & * & & & * & & * \\
* & * & & & & * & & & \bigcirc & \\
& & * & * & & & * & & \bigcirc & \bigcirc \\
& & & & * & & & * & & \bigcirc \\
& * & * & & & \bigcirc & \bigcirc & & * & \bigcirc \\
& & & * & * & & \bigcirc & \bigcirc & \bigcirc & *
\end{array}
$$

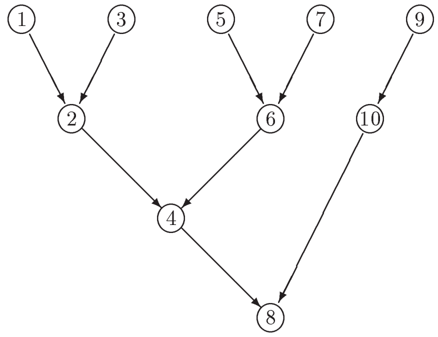


Abbildung 6.6: Matrix und Eliminationsbaum nach der Umordnung durch zyklische Reduktion. Die $\bigcirc$ - Einträge kennzeichnen das nunmehr auftretende Fill-In.

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

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

6.4 Die Fast Fourier Transformation

Die Fourier Transformation ist eine Methode 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.

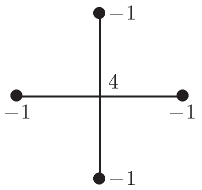
Anwendung :
- Signalverarbeitung,
- Partielle Differentialgleichungen über Rechteckgebieten,
- Vorkonditionierung zyklischer Matrizen (BEM).

Bsp.: *Partielle Differentialgleichung im Einheitsquadrat*

Aus der partiellen DGl.
in $\Omega = (0,1)^2$ soll die Funktion u bestimmt werden.

$$\boxed{\begin{array}{rcll} -\triangle u + cu &=& f & \text{in } \Omega \\ u &=& \varphi\,(:= 0) & \text{auf } \Gamma = \partial\Omega \end{array}}$$

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



$h = \frac{1}{N}$

$$\Longrightarrow \quad (4 + ch^2)u_{i,j} - (u_{i,j-1} + u_{i,j+1} + u_{i-1,j} + u_{i+1,j}) \;=\; h^2 f_{i,j} \quad {\scriptstyle i,j=\overline{1,N-1}}$$

$$\Longrightarrow \quad \Lambda \underline{u} \;=\; \underline{f}$$

Da in obigem Beispiel die Eigenfunktionen des diskretisierten Differentialoperators Λ

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

sind, und sich die zugehörigen Eigenwerte als

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

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

$$f_{i,j} = f(ih, jh) \;=\; \sum_{k,\ell=1}^{N-1} \gamma_{k,\ell} \cdot \mu_{k,\ell}(ih, jh) \tag{6.7a}$$

$$u_{i,j} = u(ih, jh) \;=\; \sum_{k,\ell=1}^{N-1} \beta_{k,\ell} \cdot \mu_{k,\ell}(ih, jh) \tag{6.7b}$$

Die Lösung der Differentialgleichung erhält man nunmehr wie folgt

1. Zerlegung von f in Eigenfrequenzen $\mu_{k,\ell}$ (*Fourieranalyse*)
   $\Longrightarrow$ Bestimmung der Zerlegungskoeffizienten $\gamma_{k,\ell}$ in (6.7a).
2. Durch die Eigenwerte des diskr. Operators dividieren
   $\Longrightarrow$ $\beta_{k,\ell} := \frac{\gamma_{k,\ell}}{\lambda_{k,\ell}}$
3. Zusammenbau der diskreten Näherungslösung (*Fouriersynthese*)
   $\Longrightarrow$ $u(ih, jh)$ in (6.7b)

Die Punkte 1 und 3 werden über die Fouriertransformation realisiert.

6.4.1 Die 1D-Fourieranalyse und Synthese

Wegen der einfacheren Darstellung betrachten wir die Fouriertransformation für die Sinus- und Cosinusentwicklung, 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 Fouriertransformationsmatrix $\mathcal{F}$ mittels

$$\mathcal{F}_n \;=\; \{\mathcal{F}_{j,k}\}_{j,k=\overline{0,n-1}} \;:=\; \left\{\omega^{j\cdot k}\right\}_{j,k=\overline{0,n-1}} \tag{6.8}$$

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 : $\underline{u} \;:=\; \mathcal{F} \cdot \underline{b}$.

$$\Pi_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.10}$$

liefert

$$\mathcal{F}_4 \Pi_4 = \left[\begin{array}{cc|cc} 1 & 1 & 1 & 1 \\ 1 & -1 & -i & i \\ \hline 1 & 1 & -1 & -1 \\ 1 & -1 & i & -i \end{array}\right] . \tag{6.11}$$

6.4.2 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 (6.8).

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

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

$$\mathcal{F}_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 \\ 1 & \omega^2 & 1 & \omega^2 \\ 1 & \omega^3 & \omega^2 & \omega \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix} . \tag{6.9}$$

Eine Umgruppierung der Einträge in (6.9) mittels der 4×4-Permutationsmatrix

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

$$\mathcal{F}_4 \Pi_4 = \begin{bmatrix} \mathcal{F}_2 & \Omega_2 \mathcal{F}_2 \\ \mathcal{F}_2 & -\Omega_2 \mathcal{F}_2 \end{bmatrix} , \tag{6.12}$$

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

Sei $n = 2m$, so gilt

$$\mathcal{F}_n \Pi_n = \begin{bmatrix} \mathcal{F}_m & \Omega_m \mathcal{F}_m \\ \mathcal{F}_m & -\Omega_m \mathcal{F}_m \end{bmatrix} , \tag{6.13}$$

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

$$\underline{y} \;:=\; \Pi_n^T \underline{x} \quad \Leftrightarrow \quad \underline{y} \;=\; \begin{bmatrix} x_j & {\scriptstyle j=\overline{0,n-1:2}} \\ x_j & {\scriptstyle j=\overline{1,n-1:2}} \end{bmatrix} \tag{6.14}$$

und $\Omega_m \;:=\; \text{diag}\{\omega_{n=2m}^k,\; \overline{k=0,m-1}\}$.
Wenn $n = 2^P$ ist, so läßt sich (6.13) $P-1$-mal rekursiv anwenden. Damit reduziert sich eine FFT auf das $P-1$-malige Umordnen eines Vektors mit etwas zusätzlicher Arithmetik.

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. $x(0, 7 : 2) = [x_j]_{j=\overline{0,7:2}}$

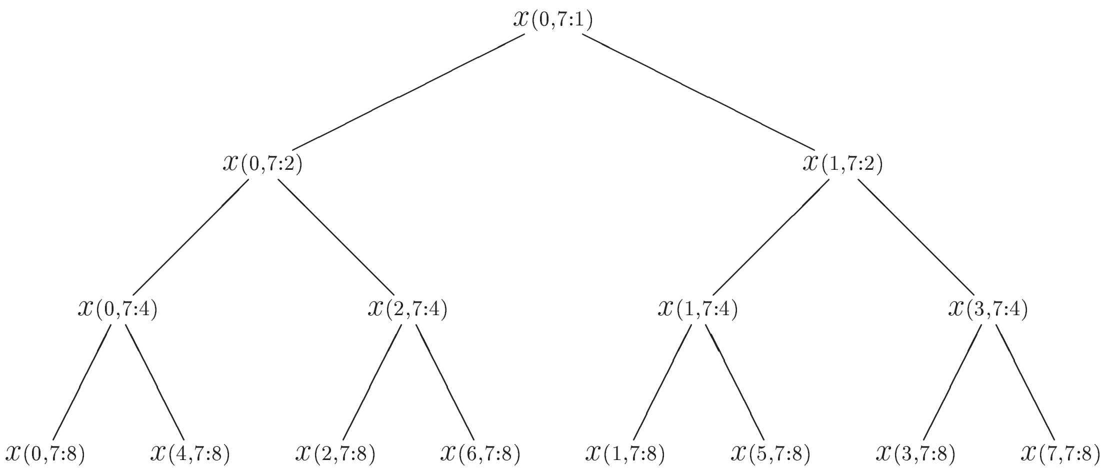


Abbildung 6.7: Aufsplittung der Indizes

Der resultierende Vektor nach der Umordnung in Abbildung 6.7 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 Bild 6.8 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.
Zurückblickend auf das Beispiel zu Beginn dieses Abschnittes 6.4, ist an Stelle der diskreten Gleichung $\Lambda \underline{u} = \underline{f}$ die Gleichung

$$\tfrac{1}{n} \mathcal{F} \Lambda \mathcal{F} \mathcal{F} \cdot \underline{u} \;=\; \mathcal{F} \cdot \underline{f}$$

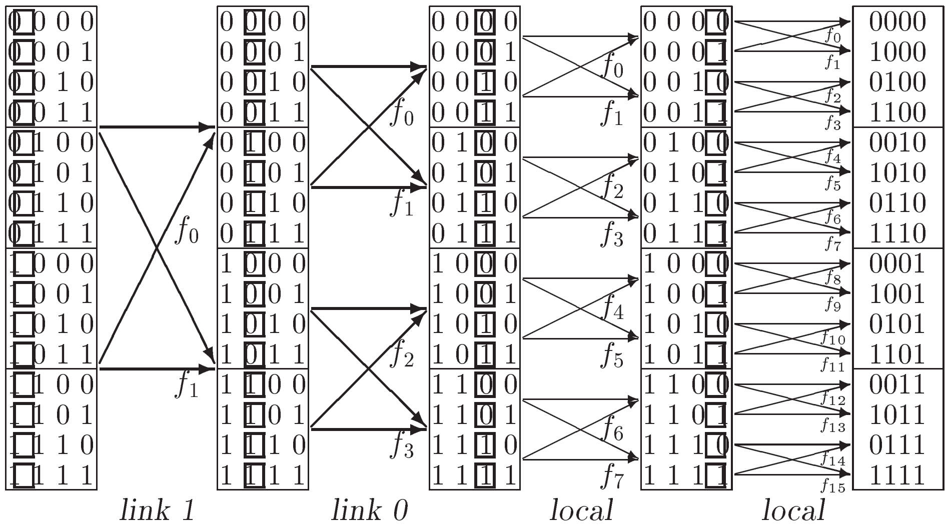


Abbildung 6.8: Datenfluß der FFT bei 4 Prozessoren
Die hervorgehobenen Bits der Indizes bestimmen des entsprechende Bit der nächsten Butterfly-Operation [Dr. Pester Chemnitz].

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 !

$\Longrightarrow$ Die Kombination von Fourieranalyse und -synthese ist sehr gut parallelisierbar.

Kapitel 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 [Joh97].

7.1.1 Die Differentialgleichungen

Aus der Impulsbilanz und der Massenerhaltung einer inkompressiblen (homogenen) Flüssigkeit ergeben sich die **inkompressiblen Navier-Stokes-Gleichungen**.
Gesucht ist $u(x,t) \in [C^2(\Omega) \cup C(\overline{\Omega})] \times [C^1((0,T]) \cup C([0,T])]$, so daß gilt :

$$\boxed{\begin{array}{rcll} u_t - \nu\Delta u + (u\cdot\nabla)\, u + \nabla p &=& f & \text{in } \Omega \times (0,T] \\ \nabla \cdot u &=& 0 & \text{in } \Omega \times (0,T] \\ u(t) &=& g(t) & \text{auf } \partial\Omega \times (0,T] \\ u(x) &=& u_0(x) & \text{für } t = 0 \end{array}} \tag{7.1}$$

Hier bezeichnen : u - Vektor der Geschwindigkeitskomponenten,
p - Druck (genauer $p := p/\varrho$),
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 $u(x) \in C^2(\Omega) \cup C(\overline{\Omega})$, so daß gilt :

$$\boxed{\begin{aligned} -\nu\Delta u + (u \cdot \nabla)\, u + \nabla p &= f && \text{in } \Omega \\ \nabla \cdot u &= 0 && \text{in } \Omega \\ u &= g && \text{auf } \partial\Omega \end{aligned}} \tag{7.2}$$

Eine weitere Vereinfachung ergibt durch Weglassen des Konvektionsterms $(u \cdot \nabla)\, u$ die **Stokes-Gleichungen**

$$\boxed{\begin{aligned} -\nu\Delta u + \nabla p &= f && \text{in } \Omega \\ \nabla \cdot u &= 0 && \text{in } \Omega \\ u &= g && \text{auf } \partial\Omega \end{aligned}} \tag{7.3}$$

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 [Joh97],...). 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 Differenzenschema (L bezeichne den elliptischen 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 !!),
für $\sigma = 1$ das rein implizite und
für $\sigma = \frac{1}{2}$ das Crank-Nicolson Schema.
Es sind auch 3-schichtige Differenzenschemata anwendbar.

Bezeichne

$\tau = t_{K+1} - t_K$	:	Zeitschrittweite				
M	:	Massenmatrix	$\longleftarrow$	$(u,v)_{L_2}$	$\longleftarrow$	u
D	:	Diffusionsmatrix	$\longleftarrow$	$a(u,v)$	$\longleftarrow$	Δu
$C(u)$	:	Konvektionsmatrix	$\longleftarrow$	$b(u,u,v)$	$\longleftarrow$	$(u\cdot\nabla)u$
B	:	Gradientenmatrix	$\longleftarrow$	$(p, \nabla\cdot v)$	$\longleftarrow$	∇p
B^T	:	Divergenzmatrix	$\longleftarrow$	$(q, \nabla\cdot u)$	$\longleftarrow$	$\nabla\cdot u$
$\underline{u}^K = \underline{u}(t_K)$	:	Geschwindigkeitsvektor				
$\underline{p}^K = \underline{p}(t_K)$	:	Druckvektor,				

dann liefert die anschließende Ortsdiskretisierung eine Folge von nichtlinearen, nichtsymmetrischen und indefiniten Gleichungssystemen ($K = 0, 1, \dots$) :

$$\begin{pmatrix} \frac{1}{\tau}M + \sigma\left(D + C(\underline{u}^{K+1})\right) & B \\ B^T & 0 \end{pmatrix} \cdot \begin{pmatrix} \underline{u}^{K+1} \\ \underline{p}^{K+1} \end{pmatrix} \;=\; \begin{pmatrix} \widehat{\underline{f}} \\ 0 \end{pmatrix} \tag{7.4}$$

mit

$$\widehat{\underline{f}} \;:=\; \sigma \underline{f}(t_{K+1}) + (1-\sigma)\underline{f}(t_K) + \left[\tfrac{1}{\tau}M - (1-\sigma)\left(D + C(\underline{u}^K)\right)\right]\underline{u}^K \ .$$

Ein nichtlineares, aber quasilineares GlS $A(\underline{v})\underline{v} = \underline{g}$ kann u.a. mittels Fixpunktiteration gelöst werden [Idee: $A(\underline{v}^{n+1}) \approx A(\underline{v}^n)$, $\underline{v}_0 = ?$] :

$$\text{Löse } A(\underline{v}^n)\underbrace{\left(\underline{v}^{n+1} - \underline{v}^n\right)}_{\underline{v}_\delta} \;=\; \underline{g} - A(\underline{v}^n)\cdot\underline{v}^n$$

$$\underline{v}^{n+1} \;:=\; \underline{v}^n + \underline{v}_\delta$$

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

$$A(\underline{u}) \;:=\; \tfrac{1}{\tau}M + \sigma\left(D + C(\underline{u})\right) \tag{7.5}$$

zur Fixpunktiteration bei der Auflösung von (7.4).

$$\begin{pmatrix}\underline{u}^0\\ \underline{p}^0\end{pmatrix} := \begin{pmatrix}\underline{u}^K\\ \underline{p}^K\end{pmatrix} \quad , \quad n := 0$$

do

$n := n+1$

Löse das lineare, nichtsymmetrische und indefinite GlS

$$\begin{pmatrix} A(\underline{u}^n) & B \\ B^T & 0 \end{pmatrix} \cdot \begin{pmatrix}\underline{u}_\delta^n\\ \underline{p}_\delta^n\end{pmatrix} = \begin{pmatrix}\underline{\widehat{f}} - A(\underline{u}^n)\underline{u}^n - B\underline{p}^n \\ -B^T\underline{u}^n\end{pmatrix} \tag{7.6}$$

$$\begin{pmatrix}\underline{u}^{n+1}\\ \underline{p}^{n+1}\end{pmatrix} := \begin{pmatrix}\underline{u}^n\\ \underline{p}^n\end{pmatrix} + \begin{pmatrix}\underline{u}_\delta^n\\ \underline{p}_\delta^n\end{pmatrix}$$

until $\left\| \begin{pmatrix}\underline{u}_\delta^n\\ \underline{p}_\delta^n\end{pmatrix} \right\| < \varepsilon_{\text{fix}}$

$$\begin{pmatrix}\underline{u}^{K+1}\\ \underline{p}^{K+1}\end{pmatrix} := \begin{pmatrix}\underline{u}^{n+1}\\ \underline{p}^{n+1}\end{pmatrix}$$

Algorithmus 7.1: Linear implizite Fixpunktiteration (seriell)

Mit

$$\begin{aligned} A &:= A(\underline{u}^n) \\ \underline{r} &:= \underline{\widehat{f}} - A(\underline{u}^n)\underline{u}^n - B\underline{p}^n \\ \underline{s} &:= -B^T\underline{u}^n \end{aligned}$$

läßt sich das lineare Sattelpunktproblem (7.6) als

$$\begin{pmatrix} A & B \\ B^T & 0 \end{pmatrix} \cdot \begin{pmatrix}\underline{u}_\delta\\ \underline{p}_\delta\end{pmatrix} = \begin{pmatrix}\underline{r}\\ \underline{s}\end{pmatrix} \tag{7.7}$$

schreiben.

Eigenschaften von A, nach John [Joh97] Seite 30 :

- Stokes (7.3) $\Longrightarrow$ A ist symmetrisch, positiv definit.
- Navier Stokes (7.2) mit Stabilisierung durch scharfes Upwind oder Streamline-Diffusion-Upwind-Petrov-Galerkin (SUPG)
  $\Longrightarrow$ $A = A(\underline{u})$ 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) $\Longrightarrow$ $A = A(\underline{u})$ 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 [Zul97].
<u>Idee :</u> Faktorisiere die Blockmatrix A und approximiere gegebenfalls zu invertierende Teilmatrizen :

$$\begin{pmatrix} A & B \\ B^T & 0 \end{pmatrix} \approx \begin{pmatrix} \widehat{A} & 0 \\ B^T & I \end{pmatrix} \begin{pmatrix} \widehat{A}^{-1} & 0 \\ 0 & \widehat{C}^{-1} \end{pmatrix} \begin{pmatrix} \widehat{A} & B \\ 0 & I \end{pmatrix} \quad ,$$

mit $\widehat{A} \approx A$ und $\widehat{C} \approx B^T A^{-1} B$ 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 $\underline{c}$, $\overline{c}$ so daß gilt

$$\underline{c}\,(G \cdot \underline{v}, \underline{v}) \;\leq\; (H \cdot \underline{v}, \underline{v}) \;\leq\; \overline{c}\,(G \cdot \underline{v}, \underline{v}) \qquad \forall \underline{v} \in \mathbb{R}^n \quad .$$

$$\boxed{\begin{array}{lrcl} \text{Löse} & \widehat{A}\widehat{\underline{u}}_\delta & = & \underline{r} \\ \text{Löse} & \widehat{C}\underline{p}_\delta & = & B^T \widehat{\underline{u}}_\delta - \underline{s} \\ \text{Löse} & \widehat{A}\underline{u}_\delta & = & \underline{r} - B\underline{p}_\delta \end{array}} \tag{7.8}$$

Algorithmus 7.2: Druckkorrekturverfahren (seriell, eine Iteration)

Bemerkungen :

(i) Ein Weglassen der 3.Zeile in Algorithmus 7.2 ergibt den vorkonditionierten Arrow-Hurwicz Algorithmus

$$\boxed{\begin{array}{lrcl} \text{Löse} & \widehat{A}\underline{u}_\delta & = & \underline{r} \\ \text{Löse} & \widehat{C}\underline{p}_\delta & = & B^T \underline{u}_\delta - \underline{s} \;=\; B^T \underline{u}^{n+1} \end{array}}$$

Algorithmus 7.3: Vorkonditionierter Arrow-Hurwicz Algorithmus

(ii) Wählt man $\widehat{C} = \frac{1}{\tau} I_{N_p}$, $\widehat{A} = A$ und läßt die 3.Zeile in Algorithmus 7.2 weg, so erhält man den klassischen Uzawa-Algorithmus [BF91]. Die hierbei entstehende Iterationsmatrix ist symmetrisch bzgl. eines speziell gewählten Skalarproduktes.

$$\begin{array}{lrcl} \text{Löse} & A\underline{u}^{n+1} & = & \underline{f} - B\underline{p}^{n} \\ \text{Löse} & \widehat{C}\underline{p}_{\delta} & = & B^T\underline{u}_{\delta} - \underline{s} \; = \; B^T\underline{u}^{n+1} \end{array}$$

Algorithmus 7.4: Vorkonditionierter Uzawa Algorithmus

(iii) Da die Druckkorrektur in (7.8) möglichst genau gelöst werden soll, wird häufig die Fixpunktiteration

$$\underline{p}_{\delta}^{m+1} - \underline{p}_{\delta}^{m} \;:=\; \alpha \left(B^T\widetilde{A}^{-1}B\right)^{-1} \left(-B^T\widehat{A}^{-1}B\underline{p}_{\delta}^{m} + B^T\underline{u}_{\delta} - \underline{s}\right) \tag{7.9}$$

mit $\widetilde{A} \approx A$ benutzt.
Mögliche Wahl von $\widetilde{A}$: Einheitsmatrix I_{N_u}, Massenmatrix M (nichtkonforme Elemente), $\mathrm{diag}(A)$.
Mögliche Wahl von $\left(B^T\widetilde{A}^{-1}B\right)$: 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 $\underline{p}_{\delta}$ 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 !

(iv) Üblicherweise wird auch $\widehat{A}^{-1}$ über Iterationsverfahren realisiert.

(v) Die Fixpunktiteration (Algorithmus 7.1) entstand durch die Linearisierung von (7.4) mittels

$$C(\underline{u}^{K+1})\underline{u}^{K+1} \quad \xrightarrow[\text{implizit}]{\text{linear}} \quad C(\underline{u}^{K})\underline{u}^{K+1} \ .$$

Wählt man stattdessen die Ersetzung

$$C(\underline{u}^{K+1})\underline{u}^{K+1} \quad \xrightarrow[\text{explizit}]{\text{Konvektion}} \quad C(\underline{u}^{K})\underline{u}^{K} \ ,$$

erhält man sofort das Sattelpunktproblem (7.7) mit

$$\begin{array}{rcl} A & := & \frac{1}{\tau}M + \sigma D \qquad\qquad (spd!) \\ \underline{r} & := & \sigma\underline{f}(t_{K+1}) + (1-\sigma)\underline{f}(t_K) + \left[\frac{1}{\tau}M - (1-\sigma)D - C(\underline{u}^K)\right]\underline{u}^K \\ \underline{s} & := & 0 \ , \end{array}$$

welches als Lösung $\underline{u}^{K+1} := \underline{u}_\delta$, $\underline{p}^{K+1} := \underline{p}_\delta$ liefert.
Da der Konvektionsanteil explizit behandelt wird, ist dieses Verfahren in der Zeititeration instabiler (insbesondere bei dominantem Konvektionsanteil).

(vi) Die stationären Navier-Stokes-Gleichungen (7.2) werden durch (7.4) mit

$$M := 0 \ , \qquad \sigma := 1 \ , \qquad \underline{u} := \underline{u}^{K+1} = \underline{u}^K \ , \qquad \underline{p} := \underline{p}^{K+1} = \underline{p}^K$$

repräsentiert.

(vii) Die Stokes-Gleichungen (7.3) drücken sich in dem Sattelpunktproblem (7.7) mit den Komponenten

$$A := D \ , \qquad \underline{r} := \underline{f} \ , \qquad s := 0 \ , \qquad \underline{u} := \underline{u}_\delta \ , \qquad \underline{p} := \underline{p}_\delta$$

aus. Die Matrix A ist symmetrisch und positiv definit (spd).

for $K = 0, 1, \ldots$ do
- Fixpunktiteration (Alg. 7.1)
  - Druckkorrekturverfahren (Alg. 7.2)
    - evtl. Fixpunktiteration (7.9)

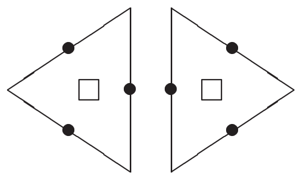
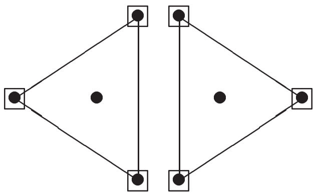
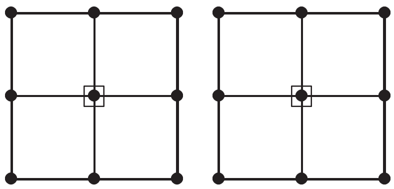
od

Algorithmus 7.5: Gesamtalgorithmus zur numerischen Lösung der instationären Navier-Stokes-Gleichungen

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 folgenden Elemente verwendet werden :

(a)	nichtkonformes $[\mathrm{P}_1,\ \mathrm{P}_0\text{-}]$ Element	
(b)	konformes (in u, p) $[\mathrm{P}_1\text{+Bubble},\ \mathrm{P}_1]$ MINI-Element	
(c)	konformes (in u) $[\mathrm{Q}_1^h,\ \mathrm{Q}_0^H]$ Element, Druckkomponente in Mittelpunkt des Makroelements	

Für die Parallelisierung bietet sich eine nichtüberlappende Elementaufteilung entsprechend Abschnitt 4.3.1 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. $\mathsf{M} = \sum_{s=1}^{p} A_s^T \mathsf{M}_s A_s$ und analog für D, C,B und genauso $\underline{\mathsf{f}}$. Da diese Matrizen teilgebietsweise berechnet und gespeichert werden, fällt außer evtl. für C_s keine Kommunikation an.
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 ($\sum$)
die Kenntnis der Geschwindigkeitskomponenten weiterer Nachbarelemente einen vorbereitenden Kommunikationsschritt.

Die parallele Fixpunktiteration

Da sämtliche Teilmatrizen in (7.4) vom verteilten Typ sind (siehe Abschnitt 4.3.1), bietet sich die Definition der Zustandgrößen $\underline{\mathfrak{u}}$, $\underline{\mathfrak{p}}$ als akkumulierte Vektoren und der integralen Größen $\underline{\mathsf{f}}$ ($\underline{\mathsf{r}}$, $\underline{\mathsf{s}}$) als verteilte Vektoren an.

$$\begin{pmatrix}\underline{\mathfrak{u}}^0\\ \underline{\mathfrak{p}}^0\end{pmatrix} := \begin{pmatrix}\underline{\mathfrak{u}}^K\\ \underline{\mathfrak{p}}^K\end{pmatrix} \quad , \quad n := 0$$

do

$n := n+1$

$\boxed{\text{Löse}}$ das lineare, nichtsymmetrische und indefinite GlS

$$\begin{pmatrix}\mathsf{A}(\underline{\mathfrak{u}}^n) & \mathsf{B}\\ \mathsf{B}^T & 0\end{pmatrix} \cdot \begin{pmatrix}\underline{\mathfrak{u}}^n_\delta\\ \underline{\mathfrak{p}}^n_\delta\end{pmatrix} = \begin{pmatrix}\underline{\widehat{\mathsf{f}}} - \mathsf{A}(\underline{\mathfrak{u}}^n)\underline{\mathfrak{u}}^n - \mathsf{B}\underline{\mathfrak{p}}^n\\ -\mathsf{B}^T\underline{\mathfrak{u}}^n\end{pmatrix}$$

$$\begin{pmatrix}\underline{\mathfrak{u}}^{n+1}\\ \underline{\mathfrak{p}}^{n+1}\end{pmatrix} := \begin{pmatrix}\underline{\mathfrak{u}}^n\\ \underline{\mathfrak{p}}^n\end{pmatrix} + \begin{pmatrix}\underline{\mathfrak{u}}^n_\delta\\ \underline{\mathfrak{p}}^n_\delta\end{pmatrix}$$

until $\displaystyle\sum_{s=1}^{p}\left[\left(R^{-1}_{\mathfrak{u},s}\underline{\mathfrak{u}}^n_{\delta,s}, \underline{\mathfrak{u}}^n_{\delta,s}\right) + \left(R^{-1}_{\mathfrak{p},s}\underline{\mathfrak{p}}^n_{\delta,s}, \underline{\mathfrak{p}}^n_{\delta,s}\right)\right] < \varepsilon^2_{\text{fix}}$

$$\begin{pmatrix}\underline{\mathfrak{u}}^{K+1}\\ \underline{\mathfrak{p}}^{K+1}\end{pmatrix} := \begin{pmatrix}\underline{\mathfrak{u}}^{n+1}\\ \underline{\mathfrak{p}}^{n+1}\end{pmatrix}$$

Algorithmus 7.6: Linear implizite Fixpunktiteration (parallel)

Mit

$$\begin{aligned}\underline{\widehat{\mathsf{f}}} &:= \sigma\mathsf{f}(t_{K+1}) + (1-\sigma)\mathsf{f}(t_K) + \left[\tfrac{1}{\tau}\mathsf{M} - (1-\sigma)\left(\mathsf{D} + \mathsf{C}(\underline{\mathfrak{u}}^K)\right)\right]\underline{\mathfrak{u}}^K\\ \mathsf{A}(\underline{\mathfrak{u}}^n) &:= \tfrac{1}{\tau}\mathsf{M} + \sigma\left(\mathsf{D} + \boxed{\mathsf{C}(\underline{\mathfrak{u}}^n)}\right) \quad .\end{aligned}$$

- Kommunikation erfordernde Algorithmenteile werden mit $\boxed{}$ gekennzeichnet, bzw. sind durch die Akkumulationssumme $\sum_{s=1}^{p}$ ersichtlich.

- Im teilgebietsweisen inneren Produkt muß die notwendige, kommunikationslose Vektortypumwandlung (4.6) an den akkumulierten Vektoren $\underline{\mathfrak{u}}^n_{\delta,s}$ und $\underline{\mathfrak{p}}^n_{\delta,s}$ vorgenommen werden.
  Bei den Elementtypen (a) und (c) mit Verteilung der Makroelemente ist $R_{\mathfrak{p}} = I_{N_p}$.

Das parallele Druckkorrekturverfahren

Das Sattelpunktproblem (7.7) (linear, nichtsymmetrisch, indefinit)

$$\begin{pmatrix} \mathsf{A} & \mathsf{B} \\ \mathsf{B}^T & 0 \end{pmatrix} \cdot \begin{pmatrix} \underline{\mathfrak{u}}_\delta \\ \underline{\mathfrak{p}}_\delta \end{pmatrix} = \begin{pmatrix} \underline{\mathsf{r}} \\ \underline{\mathsf{s}} \end{pmatrix}$$

$$\text{mit} \qquad \begin{aligned} \underline{\mathsf{r}} &:= \underline{\widehat{\mathsf{f}}} - \mathsf{A}(\underline{\mathfrak{u}}^n)\underline{\mathfrak{u}}^n - \mathsf{B}\underline{\mathfrak{p}}^n \\ \underline{\mathsf{s}} &:= -\mathsf{B}^T \underline{\mathfrak{u}}^n \end{aligned}$$

wird mit dem parallelen Druckkorrekturverfahren gelöst.

Löse	$\widehat{A} \cdot \underline{\widehat{\mathfrak{u}}}_\delta$	$=$	$\underline{\mathsf{r}}$	mittels geeigneter IV aus Kap. 5
Löse	$\widehat{C} \cdot \underline{\mathfrak{p}}_\delta$	$=$	$\mathsf{B}^T \underline{\widehat{\mathfrak{u}}}_\delta - \underline{\mathsf{s}}$	(7.10)
Löse	$\widehat{A} \cdot \underline{\mathfrak{u}}_\delta$	$=$	$\underline{\mathsf{r}} - \mathsf{B}\underline{\mathfrak{p}}_\delta$	mittels geeigneter IV aus Kap. 5

Algorithmus 7.7: Druckkorrekturverfahren (parallel)

Zur Auflösung von (7.10) :

- $\widehat{C} := \mathfrak{I}_{N_p}$ (Uzawa) $\Longrightarrow \underline{\mathfrak{p}}_\delta = \sum\limits_{s=1}^{p} A_s^T \left(\mathsf{B}_s^T \underline{\mathfrak{u}}_\delta - \underline{\mathsf{s}}_s \right)$

  Im Falle konstanter Druckansätze pro Element (Diskretisierungen a) und b)) ist keine Typumwandlung/Akkumulation des Druckvektors nötig.

- $\widehat{A} \approx \mathfrak{A} = \sum\limits_{s=1}^{p} A_s^T \cdot \mathsf{A}_s(\underline{\mathfrak{u}}^n) \cdot A_s, \qquad \widehat{C} := \left(\mathsf{B}^T \widetilde{\mathfrak{A}}^{-1} \mathsf{B} \right), \qquad \widetilde{\mathfrak{A}} \approx \mathfrak{A}$

- Löst man die Gleichung (7.10) mittels der Fixpunktiteration (7.9)

  $$\left(\mathsf{B}^T \widetilde{\mathfrak{A}}^{-1} \mathsf{B} \right) \left(\underline{\mathfrak{p}}_\delta^{m+1} - \underline{\mathfrak{p}}_\delta^m \right) = \alpha \left(\left(\mathsf{B}^T \widehat{A}^{-1} \mathsf{B} \right) \underline{\mathfrak{p}}_\delta^m + \mathsf{B}^T \underline{\mathfrak{u}}_\delta - \underline{\mathsf{s}} \right) \quad , \tag{7.11}$$

  so wird bei der Berechnung der rechten Seite in der $\boxed{\text{Auflösung}}$ von

  $$\widehat{A} \cdot \underline{\mathfrak{w}} = \mathsf{B}\underline{\mathfrak{p}}_\delta^m$$

  wiederum ein geeignetes parallelisiertes IV aus Kap. 5 benutzt.

- Zur Auflösung von (7.11) setzen wir ein Iterationsverfahren ein, welches in jedem Schritt der Matrix-mal-Vektor Operation

  $$\underline{\mathsf{v}} := \mathsf{B}^T \underbrace{\widetilde{\mathfrak{A}}^{-1} \mathsf{B} \cdot \mathfrak{w}}_{\widehat{\underline{\mathfrak{v}}}}$$

die schnelle parallele (und direkte) $\boxed{\text{Auflösung}}$ von

$$\widetilde{\mathfrak{A}} \cdot \underline{\widehat{\mathfrak{v}}} = \mathsf{B}\underline{\mathfrak{w}}$$

erfordert.
Für $\widetilde{\mathfrak{A}} := \mathfrak{I}_{N_u}$ bzw. $\widetilde{\mathfrak{A}} := \sum_{s=1}^{p} \mathrm{diag}\mathsf{A}_s$ ist diese Forderung mit einer Vektorakkumulation per Iteration leicht realisierbar.
Bei einer Diskretisierung mit dem nichtkonformen Element (a) sichert die Diagonalität von M ([Joh97],pp.27) bei der Wahl $\widetilde{\mathfrak{A}} := \mathfrak{M} = \sum_{s=1}^{p} \mathsf{M}_s$ die schnelle parallele Auflösbarkeit.
Bei Verwendung des konformen Elements (c) ist diese günstige Diagonaleigenschaft nicht mehr gegeben, wodurch eine Verwendung von $\mathfrak{A} = \mathfrak{M}$ parallel ineffizient wird. Ersetzt man jedoch $\mathfrak{M}$ durch eine Matrix $\widetilde{\mathfrak{M}}$ wie in Abschnitt 5.5.6 beschrieben, und wählt $\widetilde{\mathfrak{A}}$ gleich der entsprechenden unvollständigen Zerlegung (Kapitel 5.5), ist mittels eines Iterationsverfahrens eine effiziente parallele Auflösung möglich.

7.2 Die Euler-Gleichungen

Die Darstellung der ersten beiden Abschnitte dieses Kapitels lehnt sich stark an Kröner[Krö97], pp. 372-394, an

7.2.1 Die Differentialgleichungen

Die Euler-Gleichungen der Gasdynamik haben das Aussehen

$$\boxed{\begin{array}{l} \partial_t \begin{pmatrix} \varrho \\ \varrho u \\ \varrho v \\ e \end{pmatrix} + \partial_x \begin{pmatrix} \varrho u \\ \varrho u^2 + p \\ \varrho uv \\ u \cdot (e+p) \end{pmatrix} + \partial_y \begin{pmatrix} \varrho v \\ \varrho uv \\ \varrho v^2 + p \\ v \cdot (e+p) \end{pmatrix} \;=\; 0 \qquad \text{in } \mathbb{R}^2 \supset \Omega \times (0,T] \\ \text{Zusätzlich} \quad p \;=\; (\gamma - 1)\left[e - \dfrac{\varrho}{2} \cdot (u^2+v^2)\right] \\ \qquad\qquad\quad + \text{AB} \qquad + \text{RB} \end{array}} \tag{7.12}$$

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) :

$$\boxed{\begin{array}{l} \partial_t \vec{u} + \partial_x f_1(\vec{u}) + \partial_y f_2(\vec{u}) \;=\; 0 \qquad \text{in } \mathbb{R}^2 \supset \Omega \times (0,T] \\ \qquad\qquad + \text{AB} \qquad + \text{RB} \;, \end{array}} \tag{7.13}$$

mit $\vec{u} = \begin{pmatrix} \varrho \\ \varrho u \\ \varrho v \\ e \end{pmatrix}$.

7.2.2 Die serielle Auflösung

- Zeitdiskretisierung über ein explizites 2-schichtiges Differenzenschema, mit $\vec{u}^K := \vec{u}(t_K)$ und $\tau = t_{K+1} - t_K$ führt auf :

$$\boxed{\begin{array}{lll} \vec{u}^0 & \text{aus AB gegeben,} & \\ \vec{u}^{K+1} & := \ \vec{u}^K - \tau \cdot \operatorname{div} f(\vec{u}^K) \qquad \text{in } \Omega & \forall K = 0,1,\ldots \\ + \text{ RB} & & \end{array}} \tag{7.14}$$

- 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 :

$$\boxed{\begin{array}{lll} u_j^0 \ = \ \dfrac{1}{|T_j|} \displaystyle\int_{T_j} \vec{u}^0(x)dx & j = \overline{1,N} & \\ u_j^{K+1} \ := \ u_j^K - \dfrac{1}{|T_j|} \displaystyle\int_{T_j} \operatorname{div} f(\vec{u}^K(x))dx & j = \overline{1,N} & \forall K = 0,1,\ldots \end{array}} \tag{7.15}$$

- Der Gaußsche Integralsatz überführt das Volumenintegral aus (7.15) in ein Oberflächenintegral, welches sich als Summe der $n_e(j)$ Integrale auf den Begrenzungskanten (-flächen) S_{jl} des Volumenelements T_j darstellen läßt. n_{jl} ist der nach außen gerichtete Normalenvektor von S_{jl}.

$$\int_{T_j} \operatorname{div} f(\vec{u}^K)\, dx \ = \ \int_{\partial T_j} f(\vec{u}^K(s))\vec{n}(s)\, ds \ = \ \sum_{l=1}^{n_e(j)} \int_{S_{jl}} f(\vec{u}^K(s))\vec{n}_{jl}\, ds$$

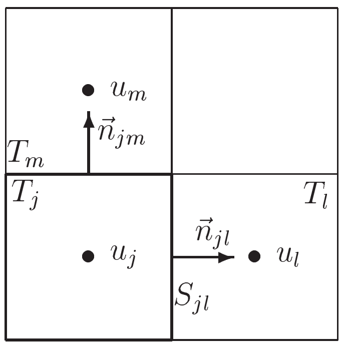


Abbildung 7.1: Bezeichnungen am Volumenelement T_j.

- 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 $\int_{S_{jl}} \cdot$ durch einen numerischen Fluß

$$g_{jl}(\underline{u}^K) \;\approx\; \int_{S_{jl}} f(\underline{u}^K(s))\vec{n}_{jl}\, ds \;. \tag{7.16}$$

Das numerische Schema zum Lösen von (7.13) ist dann

$$\boxed{\begin{aligned} u_j^0 &= \frac{1}{|T_j|}\int_{T_j} \vec{u}^0(x)dx \qquad\qquad j = \overline{1,N} \\ u_j^{K+1} &:= u_j^K - \frac{\tau}{|T_j|}\sum_{l=1}^{n_e(j)} g_{jl}(\underline{u}^K) \qquad j = \overline{1,N} \quad \forall K = 0,1,\ldots \end{aligned}} \tag{7.17}$$

Die Berechnung von g_{jl} wird im folgenden kurz skizziert.

<u>Berechnung von g_{jl} mittels Flux-Vector-Splitting</u>

Die Vorgehensweise entspricht der in Kröner [Krö97], p.374f.
Es gelte, wie z.B. in (7.12),

$$f_1(\vec{u}) \;=\; f_1'(\vec{u})\cdot\vec{u} \qquad \text{und} \qquad f_2(\vec{u}) \;=\; f_2'(\vec{u})\cdot\vec{u} \;.$$

Die Definition

$$C_{jl}(\vec{u}) \;:=\; \vec{n}_{jl} f'(\vec{u})$$

führt über ($Q(\vec{u})$ - nichtsingulär) $Q^{-1}C_{jl}Q = D = \operatorname{diag}\{\lambda_1,\ldots,\lambda_m\}$ und $\lambda_i^+ := \max\{\lambda_i,\, 0\}$, $\lambda_i^- := \min\{\lambda_i,\, 0\}$, $i = \overline{1,m}$ zu

$$C_{jl}^+(\vec{u}) \;=\; QD^+Q^{-1} \qquad \text{und} \qquad C_{jl}^-(\vec{u}) \;=\; QD^-Q^{-1} \;.$$

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 $g_{jl} := \int_{S_{jl}} \widetilde{g}_{jl}(s)ds$ mittels

$$g_{jl}(w,v) \;:=\; |S_{jl}|\,(C_1(w,v)w + C_2(w,v)v) \tag{7.18}$$

berechnet.

Für die Berechnung der C_1, C_2 gibt es verschiedene Ansätze :

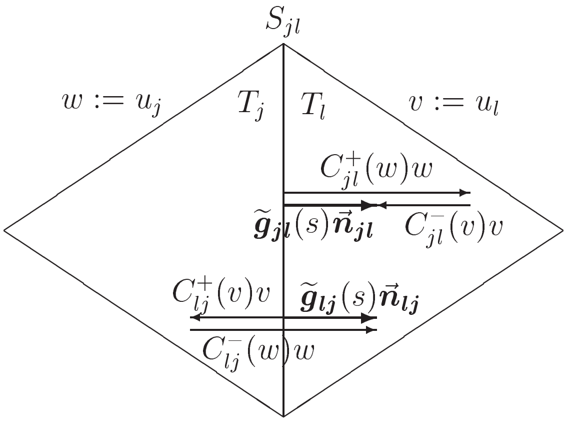


Abbildung 7.2: Fluß zwischen den Elementen T_j und T_l nach Steger und Warming.

1. Nach Steger und Warming [SW81]

$$\begin{aligned} C_1(w,v) &:= C_{jl}^+(w) \\ C_2(w,v) &:= C_{jl}^-(v) \quad \left[\overset{!}{=} -C_{lj}^+(v) \right] \end{aligned} \tag{7.19}$$

2. Nach Vijayasundaram [Vij86]

$$\begin{aligned} C_1(w,v) &:= C_{jl}^+(\frac{w+v}{2}) \\ C_2(w,v) &:= C_{jl}^-(\frac{w+v}{2}) \quad \left[\overset{!}{=} -C_{lj}^+(\frac{w+v}{2}) \right] \end{aligned} \tag{7.20}$$

3. Nach van Leer [Lee92]

$$\begin{aligned} C_1(w,v) &:= C_{jl}(w) + \left| C_{jl}^+(\frac{w+v}{2}) \right| \\ C_2(w,v) &:= C_{jl}(v) - \left| C_{jl}^-(\frac{w+v}{2}) \right| \\ & \left[\overset{!}{=} -C_{lj}(v) - \left| C_{lj}^+(\frac{w+v}{2}) \right| \right] \end{aligned} \tag{7.21}$$

Bemerkung :
Die Matrizen $C_{jl}^+(w)$, $C_{jl}^-(w)$ und $C_{jl}(w)$ sind lokal in T_j berechenbar. Analog sind $C_{lj}^+(v)$, $C_{lj}^-(v)$ und $C_{lj}(v)$ lokal in T_l berechenbar. Für die Anwendung des Argumentes $\frac{w+v}{2}$ muß als Vorbereitungsschritt der Mittelwert gebildet $\sum$

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 = 1,N do
    for l ∈ {neighbours} do
        σ_jl := C_1(u_j, u_l) · u_j
    od
od
for j = 1,N do
    g := 0
    for l ∈ {neighbours} do
        g := g + |S_jl| (σ_jl − σ_lj)
    od
    u_j^{K+1} := u_j^K − τ/|T_j| · g
od
```

$$
\begin{array}{l}
\textsf{for } j = \overline{1,N} \textsf{ do}\\
\quad \textsf{for } l \in \{\text{neighbours}\} \textsf{ do}\\
\quad\quad \sigma_{jl} \; := \; C_1(u_j, u_l) \cdot u_j\\
\quad \textsf{od}\\
\textsf{od}\\
\textsf{for } j = \overline{1,N} \textsf{ do}\\
\quad g \; := \; 0\\
\quad \textsf{for } l \in \{\text{neighbours}\} \textsf{ do}\\
\quad\quad g \; := \; g + |S_{jl}| \, (\sigma_{jl} - \sigma_{lj})\\
\quad \textsf{od}\\
\quad u_j^{K+1} \; := \; u_j^K - \dfrac{\tau}{|T_j|} \cdot g\\
\textsf{od}
\end{array}
$$

Algorithmus 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 der Aufteilung mit nichtüberlappenden Knoten aus Abschnitt 4.3.2.
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 $\widetilde{\Omega}_i$ bezeichnet, der erweiterte Lösungsvektor sei $\underline{\widetilde{u}}_i = \Big(\underbrace{\{u_j\}_{T_j \in \Omega_i}}_{\underline{u}_i}, \underbrace{\{u_l\}_{T_l \in \widetilde{\Omega}_i \setminus \Omega_i}}_{\text{externe Komp.}} \Big)$, analog $\widetilde{\sigma}$.

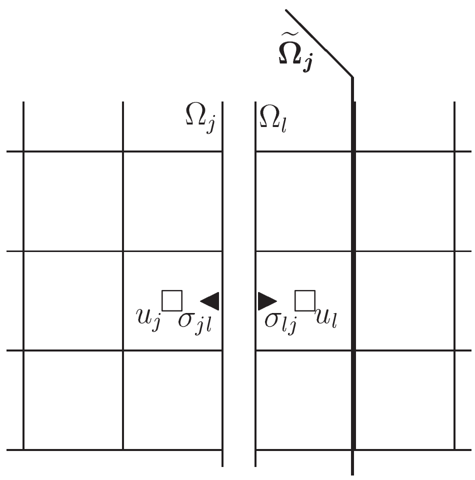


Abbildung 7.3: Verteilte Boxen

1.) Tausche die Randkomponenten von $\underline{u}$ mit den Nachbarn über die Kanten aus [EXCHANGED].
Aktualisiere die externen Komponenten von $\widetilde{\underline{u}}_i$, d.h. $\{\widetilde{u}_l \notin \underline{u}_i\}$.

2.) for $j = \overline{1, N}$ do
    for $l \in \{\text{neighbours}\}$ do
        $\sigma_{jl} \;:=\; C_1(u_j, \widetilde{u}_l) \cdot u_j$
    od
od

3.) Tausche die Randkomponenten von σ, d.h. $\{\sigma_{jl} | \widetilde{u}_l \notin \underline{u}_i\}$ mit den Nachbarn über die Kanten aus [EXCHANGED].
Aktualisiere die externen Komponenten $\{\widetilde{\sigma}_{lj} | \widetilde{u}_l \notin \underline{u}_i \wedge \widetilde{u}_j \in \underline{u}_i\}$.

4.) for $j = \overline{1, N}$ do
    $g \;:=\; 0$
    for $l \in \{\text{neighbours}\}$ do
        $g \;:=\; g + |S_{jl}| \, (\sigma_{jl} - \widetilde{\sigma}_{lj})$
    od
    $u_j^{K+1} \;:=\; u_j^K - \dfrac{\tau}{|T_j|} \cdot g$
od

Algorithmus 7.9: (K+1)-ter Zeitschritt in Ω_i bei verteilten Boxen

Bemerkungen :

1. Algorithmus 7.10 ist für alle 3 Formeln (7.19)-(7.21) anwendbar.
2. Bei Benutzung von (7.19) kann Schritt 1.) weggelassen werden, da $\widetilde{u}_l$ nicht benötigt wird.
3. Bei Verwendung von (7.20) und (7.21) kann man den Datenaustausch in Schritt 3.) einsparen, indem in $\widetilde{\Omega}_j$ die Größe $\widetilde{\sigma}_{lj} \;:=\; -C_2(u_j, \widetilde{u}_l)$ redundant (zu $C_1(u_l, \widetilde{u}_j)$ in $\widetilde{\Omega}_l$) 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 4.3.1 verteilt wird. Somit sind sämtliche Ergebnisse bezüglich der dort definierten Vektor- und Matrizentypen anwendbar.

Man wählt die Funktionalgröße $\mathfrak{g}$ als verteilt gespeichert und die Unbekannten $\mathfrak{u}$ als akkumulierten Vektor. Jedoch liegen auch die Größen $\boldsymbol{\sigma_{pl}}$ aus denen sich $\mathfrak{g}$ ergibt, in akkumulierter Form vor und dürfen nicht mehrfach gezählt werden. Da der Flußanteil $\mathfrak{g}_{jl}$ 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 $\mathfrak{s}_{jl}$ ausgedrückt.

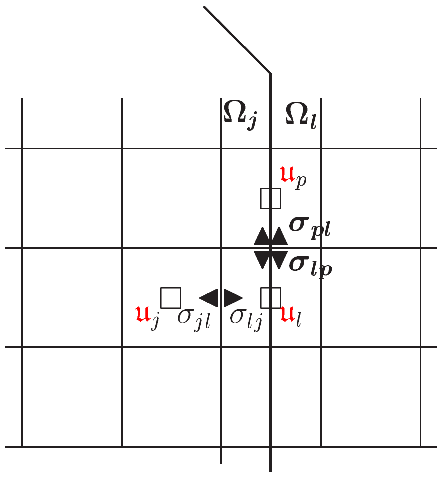


Abbildung 7.4: Aufgeteilte Boxen

0.) Vorbereitungsschritt a: Berechnung der Boxvolumina.
for $j = \overline{1, N}$ do $\quad \mathsf{t}_j := |T_j \cap \Omega_i|$ $\quad$ od

$\underline{\mathsf{t}} := \sum_{s=1}^{P} A_s^T \underline{\mathsf{t}}_s$

Vorbereitungsschritt b: Berechnung der Teilkantenlängen.
for $j = \overline{1, N}$ do
  for $l \in \{\text{neighbours}\}$ do $\quad \mathsf{s}_{jl} := |S_{jl} \cap \Omega_i|$
od od

1.) for $j = \overline{1, N}$ do
  $\mathsf{g}_j := 0$
  for $l \in \{\text{neighbours}\}$ do
    $\mathsf{g}_j := \mathsf{g}_j + \mathsf{s}_{jl} \left(C_1(\mathfrak{u}_j, \mathfrak{u}_l) \cdot \mathfrak{u}_j - C_2(\mathfrak{u}_j, \mathfrak{u}_l) \cdot \mathfrak{u}_l \right)$
  od
  $\mathsf{g}_j := \mathsf{g}_j / \mathsf{t}_j$
od

2.) $\underline{\mathfrak{u}}^{K+1} := \underline{\mathfrak{u}}^{K} - \tau \cdot \sum_{s=1}^{P} \underline{\mathsf{g}}_s$

Algorithmus 7.10: (K+1)-ter Zeitschritt in Ω_i bei aufgeteilten Boxen

Bemerkungen :

1. Algorithmus 7.9 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 7.9 Arithmetik sparen, **ohne** daß Kommunikation nötig ist !

7.3 Die kompressiblen Navier-Stokes-Gleichungen

Die Darstellung der kompressiblen Navier-Stokes-Gleichungen und ihr serieller Auflösungsprozeß folgt im wesentlichen dem Buch von Kröner [Krö97].

7.3.1 Die Differentialgleichungen

Das Verhalten einer kompressiblen Flüssigkeit in einem beschränkten Gebiet der Ebene wird analog zu (7.12) durch die **kompressiblen Navier-Stokes-Gleichungen** beschrieben:

$$\boxed{\begin{aligned}
&\partial_t \begin{pmatrix} \varrho \\ \varrho u \\ \varrho v \\ e \end{pmatrix} + \partial_x \begin{pmatrix} \varrho u \\ \varrho u^2 + p \\ \varrho uv \\ u\cdot(e+p) \end{pmatrix} + \partial_y \begin{pmatrix} \varrho v \\ \varrho uv \\ \varrho v^2 + p \\ v\cdot(e+p) \end{pmatrix} \\
&+ \partial_x \begin{pmatrix} 0 \\ \tau_{11} \\ \tau_{12} \\ \tau_{11}u + \tau_{12}v + k\partial_x\Theta \end{pmatrix} + \partial_y \begin{pmatrix} 0 \\ \tau_{21} \\ \tau_{22} \\ \tau_{21}u + \tau_{22}v + k\partial_y\Theta \end{pmatrix} \;=\; 0 \qquad \text{in } \mathbb{R}^2 \supset \Omega \times (0,T] \\
&\text{Zusätzlich} \quad p \;=\; (\gamma-1)\left[e - \frac{\varrho}{2}\cdot(u^2+v^2)\right] \\
&\qquad\qquad\quad\; e \;=\; \varrho\left[c_v\Theta + \tfrac{1}{2}(u^2+v^2)\right] \\
&\qquad\qquad\quad\; \tau_{ij} \;=\; \lambda \mathrm{div}\begin{pmatrix} u \\ v \end{pmatrix}\delta_{ij} + \mu\left(\partial_j v_i + \partial_i v_j\right) \qquad i,j = 1,2\,,\; v_1 := u,\; v_2 := v \\
&\qquad\qquad\quad\; + \text{AB} \qquad + \text{RB}
\end{aligned}} \tag{7.22}$$

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 - Wärmeleitzahl
$\delta_{ij} \;=\; \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$ - 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.

$$\frac{1}{2}\partial_t \begin{pmatrix} \varrho \\ \varrho u \\ \varrho v \\ e \end{pmatrix} + \partial_x \begin{pmatrix} \varrho u \\ \varrho u^2 + p \\ \varrho uv \\ u \cdot (e+p) \end{pmatrix} + \partial_y \begin{pmatrix} \varrho v \\ \varrho uv \\ \varrho v^2 + p \\ v \cdot (e+p) \end{pmatrix} = 0 \qquad (7.23)$$

$$\frac{1}{2}\partial_t \begin{pmatrix} \varrho \\ \varrho u \\ \varrho v \\ e \end{pmatrix} + \partial_x \begin{pmatrix} 0 \\ \tau_{11} \\ \tau_{12} \\ \tau_{11} u + \tau_{12} v + k\partial_x \Theta \end{pmatrix} + \partial_y \begin{pmatrix} 0 \\ \tau_{21} \\ \tau_{22} \\ \tau_{21} u + \tau_{22} v + k\partial_y \Theta \end{pmatrix} = 0 \qquad (7.24)$$

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.

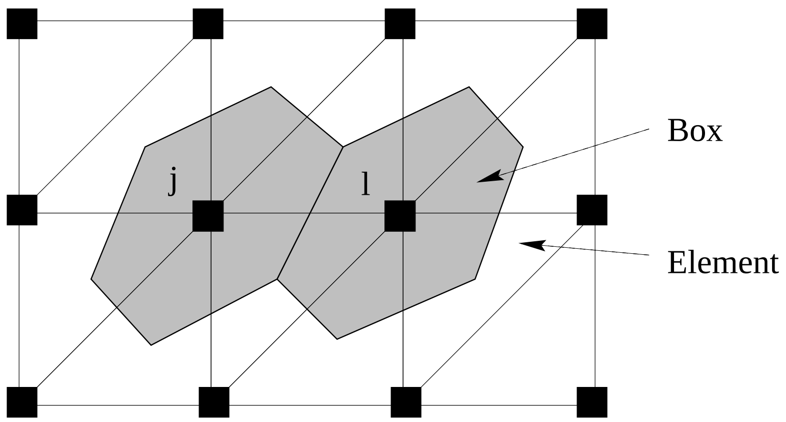


Abbildung 7.5: Duale Vernetzung

Zur Koppelung der diskretisierten Gleichungen (7.23) und (7.24) ist eine duale Vernetzung des Gebietes mit finiten Volumen und finiten Elementen ($\mathcal{T}_h$) notwendig. *Eine* mögliche Vernetzung [FFLMW97, FFLM97, FFLMW97] ist in Abbildung 7.5 gegeben.

Mit den Testfunktionen (Dreiecksvernetzung) $\varphi \in \mathbb{V}_h : \Omega \mapsto \mathbb{R}^4$ und den Definitionen

$$
\begin{aligned}
(u_h, \varphi)_h &:= \frac{1}{3} \sum_{T \in \mathcal{T}_h} |T| \sum_{i=1}^{3} u_h(P_T^i) \cdot \varphi(P_T^i) \approx \int_\Omega u_h \cdot \varphi \, dx \\
a_h(u_h, \varphi)_h &:= \text{Variationsformulierung Elastizitätsoperator} \\
b_h(u_h, \varphi)_h &:= \sum_j \varphi(P_j) \sum_{l=1}^{k} g_{jl}(u_h(P_i), u_h(P_j))
\end{aligned}
$$

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. <u>Inviscid (7.23) - viscous (7.24) operator splitting</u>

$$
\boxed{
\begin{array}{llll}
U_j^{K+1/2} & := U_j^K - \dfrac{\tau}{|T_j|} \displaystyle\sum_{l=1}^{n_e(j)} g_{jl}(\underline{U}^K) & & \\
\text{Set} & u_h^{K+1/2}(P_i) \; = \; U_i^{K+1/2} & & \forall P_i \\
\text{Solve} & (u_h^{K+1}, \varphi)_h \; = \; (u_h^{K+1/2}, \varphi)_h - \tau a_h(u_h^{K+1/2}, \varphi)_h & & \forall \varphi \in \mathbb{V}_h \\
\text{Set} & U_i^{K+1} \; = \; u_h^{K+1}(P_i) & & \forall P_i
\end{array}
}
\tag{7.25}
$$

Drückt man die 4 Komponenten von u_h und φ durch die FE-Basisfunktionen ψ_i aus, d.h.,

$$
\begin{aligned}
u_h(x) &= (p(x), u(x), v(x), e(x))^T = \sum_{i=1}^{4} (p_i, u_i, v_i, e_i)^T \cdot \psi_i(x) \\
\varphi(x) &= (\varphi_1(x), \varphi_2(x), \varphi_3(x), \varphi_4(x)) = \sum_{i=1}^{4} (\varphi_{1,i}, \varphi_{2,i}, \varphi_{3,i}, \varphi_{4,i})^T \cdot \psi_i(x) \ ,
\end{aligned}
$$

so stellt sich die 3.Zeile in (7.25) als Gleichungssystem

$$
M \cdot \underline{u}^{K+1} = M \cdot \underline{u}^{K+1/2} - \tau A \cdot \underline{u}^{K+1/2} \tag{7.26}
$$

dar, mit A als Elastizitätsmatrix und M als der Massenmatrix. Bei entsprechender Wahl der Basisfunktionen bzw. der Integrationsregel wie oben (entspricht 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 :

$$(u_h^{K+1}, \varphi)_h \;=\; (u_h^K, \varphi)_h - \tau \cdot \left[b_h(u_h^K, \varphi)_h + a_h(u_h^K, \varphi)_h\right] \qquad \forall \varphi \in \mathbb{V}_h \;\; .$$

Mit B als der Matrix der Bilinearform b_h ist somit das GlS

$$M \cdot \underline{u}^{K+1} \;=\; M \cdot \underline{u}^K - \tau(B + A) \cdot \underline{u}^K \tag{7.27}$$

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.

$$(u_h^{K+1}, \varphi)_h \;=\; (u_h^K, \varphi)_h - \tau \cdot \left[b_h(u_h^K, \varphi)_h + a_h(u_h^{K+1}, \varphi)_h\right] \qquad \forall \varphi \in \mathbb{V}_h \;\; .$$

Das resultierende GlS

$$(M + \tau A) \cdot \underline{u}^{K+1} \;=\; M \cdot \underline{u}^K - \tau B \cdot \underline{u}^K \tag{7.28}$$

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 $\underline{\mathfrak{u}}$ als akkumulierten Vektor und von $\underline{\mathsf{g}}$, M, A und B als verteilten Vektoren und Matrizen naheliegend.

0.) Vorbereitungsschritt: Berechnung Boxvolumina/Teilkantenlängen.

$\mathsf{for}\ j = \overline{1,N}\ \mathsf{do}\quad \mathsf{t}_j := |T_j \cap \Omega_i| \quad \mathsf{od}\ ; \qquad \underline{\mathfrak{t}} := \sum_{s=1}^{P} A_s^T \underline{\mathsf{t}}_s$

$\mathsf{for}\ j = \overline{1,N}\ \mathsf{do}\quad \mathsf{s}_{jl} := |S_{jl} \cap \Omega_i| \quad \forall l \in \{\text{neighbours}\} \quad \mathsf{od}$

1.) $\mathsf{for}\ j = \overline{1,N}\ \mathsf{do}\quad \mathsf{g}_j := 0$

$\quad \mathsf{for}\ l \in \{\text{neighbours}\}\ \mathsf{do}$

$\qquad \mathsf{g}_j := \mathsf{g}_j + \mathsf{s}_{jl}\left(C_1(\mathfrak{u}_j,\mathfrak{u}_l)\cdot \mathfrak{u}_j - C_2(\mathfrak{u}_j,\mathfrak{u}_l)\cdot \mathfrak{u}_l)\right\}$

$\quad \mathsf{od}$

od

2.) Generiere/modifiziere M, A und B falls nötig.

3.) Je nach Zeitschema führe aus:

a) Inviscid-viscous splitting

$$\underline{\mathfrak{u}}^{K+1/2} := \underline{\mathfrak{u}}^K - \tau \cdot \sum_{s=1}^{P} \underbrace{\underline{\mathsf{g}}_s / \underline{\mathfrak{t}}_s}_{\text{komponetenweise}}$$

$$\text{Accumulate} \quad \mathfrak{M} := \sum_{s=1}^{P} \mathsf{M}_s$$

$$\text{Solve} \quad \mathfrak{M} \cdot \underline{\mathfrak{u}}^{K+1} = \sum_{s=1}^{P} (\mathsf{M}_s - \tau \mathsf{A}_s) \cdot \underline{\mathfrak{u}}^K$$

b) Explizites Schema

$$\text{Accumulate} \quad \mathfrak{M} := \sum_{s=1}^{P} \mathsf{M}_s$$

$$\text{Solve} \quad \mathfrak{M} \cdot \underline{\mathfrak{u}}^{K+1} = \sum_{s=1}^{P} (\mathsf{M}_s - \tau\,[\mathsf{B}_s + \mathsf{A}_s]) \cdot \underline{\mathfrak{u}}^K$$

c) Semi-implizites Schema

$$\text{Accumulate} \quad \mathfrak{K} := \sum_{s=1}^{P} \mathsf{K}_s = \sum_{s=1}^{P} (\mathsf{M}_s + \tau \mathsf{A}_s)$$

$$\text{Solve} \quad \mathfrak{K} \cdot \underline{\mathfrak{u}}^{K+1} = \sum_{s=1}^{P} (\mathsf{M}_s - \tau \mathsf{B}_s) \cdot \underline{\mathfrak{u}}^K$$

Algorithmus 7.11: Zeitschritt in Ω_i für kompressible Navier-Stokes

Bemerkungen :

- 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 die Akkumulation zu $\mathfrak{K}$ in den Vorbereitungsschritt verlegt werden.

- Für Massematrizen $\mathfrak{M}$ mit Diagonalstruktur besteht das Lösen der entsprechende GlS nur in einer Multiplikation mit der akkumulierten Inversen dieser Diagonaleinträge.

- Falls $\mathfrak{M}$ keine Diagonalstruktur besitzt, wird üblicherweise durch Lumping eine Diagonalmatrix $\widetilde{\mathfrak{M}}$ erzeugt, mit welcher das GlS gelöst wird.

- Verwendet man Iterationsverfahren zur Auflösung des GlS in 3c) von Alg. 7.11, so wird K nicht akkumuliert und das GlS
$$\mathsf{K} \cdot \underline{\mathfrak{u}}^{K+1} = \underline{\mathsf{f}}$$
kann mit den Methoden aus Abschnitt 5 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.

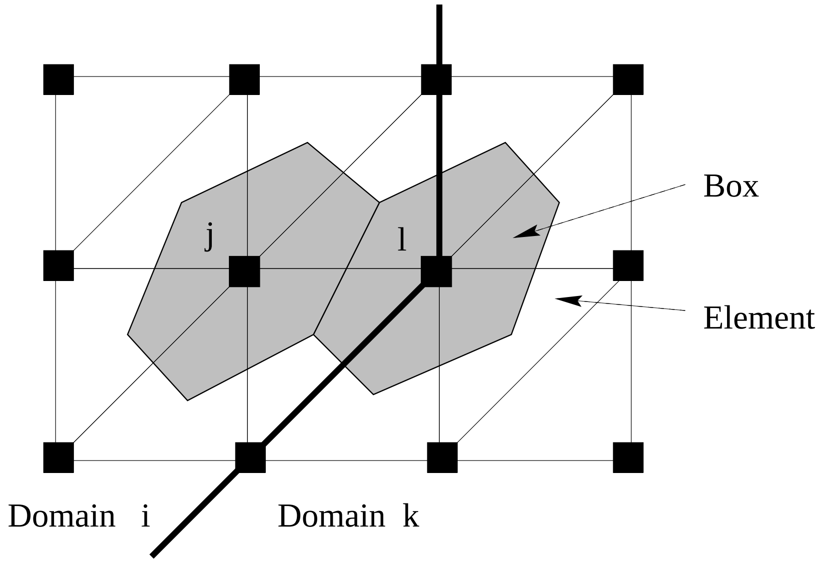


Abbildung 7.6: Duale Vernetzung und Gebietsaufteilung

Anhang A

Internet addresses

A.1 MPI

- MPI-homepage[1]
- Calls in MPI[2], or as postscript[3]
- Pacheco[4] [Pac97] : with good hints on books and software
- LAM implementarion of MPI[5] with ftp download[6].

A.2 Further links

- Lecture on Scientific Supercomputing[7], includes a lot of information on vector and parallel computers.
- History of Parallel Computing[8]
- GSCI: German Scientific Computing Page[9]
- Conferences/Information on domain decomposition methods[10]

[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.mpi-forum.org/docs/mpi-11.ps
[4] http://www.usfca.edu/mpi
[5] http://www.mpi.nd.edu/lam
[6] http://www.mpi.nd.edu/lam/download/
[7] http://www.uni-karlsruhe.de/Uni/RZ/Personen/rz03/book
[8] http://www.par.univie.ac.at/literature/parallel_history.html
[9] http://scicomp.math.uni-augsburg.de/scicomp
[10] http://www.ddm.org/

- MGNet: Multigrid and domain decomposition methods[11]
- Netlib: Mathematical Software, Papers etc.[12]
- Tools, benchmarks for parallelization[13]
- EUROPORT[14] Porting of commercial software on parallel computers
- Top 500 Supercomputer (Server U.S.A.)[15]

[11] http://www.mgnet.org/
[12] http://www.netlib.org/
[13] http://www.nhse.org/ptlib
[14] http://www.gmd.de/SCAI/europort/
[15] http://www.top500.org

Anhang B

Some quotations

- Ada Augusta, Countess of Lovelace (1815-1851). Wrote programms for the first computers by Babbage.
  It can do only what we know to order it to do. [1]

- 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 comlicated life is.

- Peter Cochrane (Forschungschef der British Telecom), 1998 über Minderheitenschutz[2]
  Es gibt 6 Milliarden Menschen auf der Welt - und 14 Milliarden Mikroprozessoren. Wir sind jetzt schon in der Minderheit.

[1] Found in the Arithmeum of Bonn, Germany
[2] Die Woche, 49/1998, pp.11

Literaturverzeichnis

[Bär93] Siegfried Bär. *Forschen auf Deutsch: Der Macchiavelli für Forscher und solche, die es noch werden wollen.* Deutsch, 1993.

[Bas96] Peter Bastian. *Parallele adaptive Mehrgitterverfahren.* Teubner Scripten zur Numerik. Teubner, 1996.

[BBC+94] R. Barrett, M. Berry, T.F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*[3]. SIAM, Philadelphia, PA, 1994.

[BDFN93] Steffen Burkhardt, Klaus-Dieter Drey, Volkmar Friedrich, and Oliver Nowak. *Parallele Rechnertsysteme: Programmierung und Anwendung.* Verlag Technik, Berlin/München, 1993.

[BF91] F. Brezzi and M. Fortin. *Mixed and Hybrid Finite Element Methods*, volume 15 of *Springer Series in Computational Mathematics.* Springer-Verlag, 1991.

[BPS89] J. H. Bramble , J. E. Pasciak , and A. H. Schatz. 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.

[CKL96] Carsten Carstensen, Michael Kuhn, and Ulrich Langer. Fast parallel solvers for symmetric boundary element domain decomposition equations. Report 500 , Institute for Mathematics, Johannes Kepler University Linz, 1996. Submitted for publication.

[3] http://www.netlib.org/linalg/html_templates/report.html

[Dij65] E. W. Dijkstra. Cooperating sequential processes. Technical Report Technical Report EWD-123, Technological University of Eindhoven, 1965. Reprinted in [Gen68].

[Dry84] M. Dryja. A finite element-capacitance matrix method for elliptic problems on regions partitioned into substructures. *Numer. Math.*, 34:153–168, 1984.

[EJL92] D.J. Evans, G.R. Joubert, and H. Lidell, editors. *Parallel Computing '91*. North-Holland, 1992. Proceedings of the Int. Conference on Parallel Computing, London, U.K., 3-6 Septembert 1991.

[FFLM97] Miroslav Feistauer, Jiri Felcman, and Maria Lukacova-Medvidova. On the convergence of a combined finite volume - finite element method for nonlinear convection - diffusion problems. *Num.Methods for Part.Diff.Eqs*, 13:163–190, 1997.

[FFLMW97] Miroslav Feistauer, Jiri Felcman, Maria Lukacova-Medvidova, and Gerald Warnecke. Error estimates of a combined finite volume - finite element method for nonlinear convection - diffusion problems. Preprint 96-27[4], University of Magdeburg, 1997.

[Fly66] Michael Flynn. Very high-speed computing systems. In *Proceedings of the IEEE*, volume 54, pages 1901–1909, December 1966.

[Fos94] Ian Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1994.

[Fox88] Geoffrey Fox. *Solving Problems on Concurrent Processors*. Prentice Hall, 1988.

[Ge90] K.A. Gallivan and et.al. *Parallel Algorithms for Matrix Computations*. SIAM, Philadelphia, 1990.

[Gen68] F. Genuys, editor. *Programming Languages*. Academic Press, London, England, 1968.

[GLS94] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI*. The MIT Press, Cambridge, London, 1994.

[4] http://www.math.uni-magdeburg.de/preprints/shadows/96-27report.html

[GO96] Gene Golub and James M. Ortega. *Scientific Computing: Eine Einführung in das wissenschaftliche Rechnen und Parallele Numerik*. Teubner, Stuttgart, 1996. dt. Übersetzung.

[Gro94] Ulrich Groh. Local realization of vector operations on parallel computers. Preprint SPC 94-2[5], TU Chemnitz, 1994. in german.

[Gus78] I. Gustafsson. A class of first order factorization methods. *BIT*, 18:142–156, 1978.

[Hac91] Wolfgang Hackbusch. *Iterative Lösung großer schwachbesetzter Gleichungssysteme*. Number Bd. 69 in Studienbücher Mathematik. Teubner, 1991.

[HB91] D.W. Heermann and A.N. Burkitt. *Parallel Algorithms in Computational Science*, volume 24 of *Information Sciences*. Springer, Berlin, 1991.

[HLM91] Gundolf Haase, Ulrich Langer, and Arnd Meyer. The approximate Dirichlet decomposition method. part I,II. *Computing*, 47:137–167, 1991.

[Hwa93] Kai Hwang. *Advanced computer architecture: parallelism, scalability, programmabilty*. McGraw-Hill computer engineering series. McGraw-Hill, 1993.

[Joh97] Volker John. *Parallele Lösung der inkompressiblen Navier-Stokes Gleichungen auf adaptiv verfeinerten Gittern*. PhD thesis, Otto-von-Guericke-Universität Magdeburg, August 1997.

[Jun96] Michael Jung. On the parallelization of multi-grid methods using a non-overlapping domain decomposition data structure. *Applied Numerical Mathematics*, 20, 1996.

[KGGK94] Vipin Kumar, Anath Grama, Anshul Gupta, and Georeg Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin/Cummings Publishing Company, Redwood City, California, 1994.

[Krö97] D. Kröner. *Numerical Schemes for Conservation Laws*. WILEY and TEUBNER, 1997.

[5] http://www.tu-chemnitz.de/ftp-home/pub/Local/mathematik/SPC/spc94_2.ps.gz

[KST95] David E. Keyes, Yousef Saad, and Donald G. Truhlar. *Domain-Based Parallelism and Problem Decomposition Methods in Computational Science and Engineering*. SIAM, 1995.

[KX94] David E. Keyes and Jinchao Xu, editors. *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.

[Law89] K. H. Law. A parallel finite element solution method. *Computer and Structures*, 23(6):845–858, 1989.

[Lee92] B. van Leer. Flux vector splitting for the euler equations. In *Proc. 8th International Conference on Numerical Methods in Fluid Dynamics*. Springer, Berlin, 1992.

[Mey90] Arnd Meyer. A parallel preconditioned conjugate gradient method using domain decomposition and inexact solvers on each subdomain. *Computing*, 45:217–234, 1990.

[MS96] Russ Miller and Quentin F. Stout. *Parallel Algorithms for Regular Architectures: Meshes and Pyramides*. The MIT Press, 1996.

[MvdV77] J. Meijerink and H. van der Vorst. A iterative solution for linear systems of which the coefficient matrix is a symmetric m-matrix. *Math. Comp.*, 31:148–162, 1977.

[Pac97] Peter Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers, 1997.

[Rec94] Peter Rechenberg. *Was ist Informatik ? : Eine allgemeinverständliche Einführung*. Carl Hanser, München/Wien, 1994.

[Saa92] Yousef Saad. *Iterative methods for sparse linear systems*. Frontiers in Applied Mathematics. PWS Publishing Company, Boston, 1992.

[SBG96] Barry Smith, Petter Bjørstad, and William Gropp. *Domain Decomposition : Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press, 1996.

[SGI97] SGI. Origin servers. Technical report, Silicon Graphics Computer Systems, April 1997.

[SW81] J.L. Steger and R.F. Warming. Flux vector splitting of the inviscid gasdynamic equations with applications to finite difference methods. *J. Comput. Phys.*, 40:263–293, 1981.

[TCK92] Charles H. Tong, Tony F. Chan, and C.C. Jay Kuo. Multilevel filtering preconditioners: Extensions to more general elliptic problems. *SIAM J. Sci. Stat. Comput.*, 13:227–242, 1992.

[TK96] Topping and Khan. *Parallel Finite Element Computations.* Saxe-Coburg Publications, Edinburgh, 1996.

[TU94] Gerald Thomas and Christoph W. Ueberhuber. *Visualization of Scientific Parallel Programs.* Springer, Berlin, 1994.

[Vij86] G. Vijayasundaram. Transonic flow simulation using an upstream centered scheme of godunov in finite elements. *J. Comput. Phys.*, 63:416–433, 1986.

[vL92] Charles van Loan. *Computational Framework for the Fast Fourier Transformation.* Frontiers in Applied Mathematics. SIAM, Philadelphia, 1992.

[Zul97] Walter Zulehner. *Numerische Methoden in der Strömungsmechanik.* University of Linz[6], 1997.

[6] http://nathan.numa.uni-linz.ac.at/Teaching/cfd.ps