Parallele Algorithmen in der Strömungsmechanik

Marcus Hutter 9.9.1990

0. Inhaltsverzeichnis

0. Inhaltsverzeichnis 1

1. Einleitung 1

2. Parallele Rechnerarchitekturen 2

3. Existierende & geplante Parallelrechner 6

4. Der Multigrid-Algorithmus 7

5. Parallelisierung des Multigrid-Algorithmus 8

6. Aufteilung des Gitters auf die Prozessoren 9

7. Literaturverzeichnis 10

1. Einleitung

Heutige Computer sind, entgegen allen Prophezeiungen, noch überwiegend seriell arbeitende von Neumann-Rechner. Dies liegt nicht etwa daran, daß Parallelrechner eine Fehlentwicklung wären, oder sich unüberwindbare Probleme bei der hardwaretechnischen Realisierung ergäben, sondern ist überwiegend ein Softwareproblem, da die automatische Parallelisierung von Algorithmen durch Compiler nur sehr bedingt möglich ist. Da aber die Simulation physikalischer Systeme in natürlicher Weise ein paralleles Problem ist (jedem Teilchen/Raumpunkt sein eigener Prozessor) ist es nicht verwunderlich, daß sich viele numerische Algorithmen leicht (relativ zu Programmen aus anderen Bereichen) parallelisieren lassen. Andererseits haben es bei der gewaltigen Anzahl von Rechenoperationen gerade die numerischen Programme nötig parallelisiert zu werden.

2. Parallele Rechnerarchitekturen

Klassifizierung:
Rechner können grob in 4 Klassen eingeteilt werden:
SISD (Single Instruction Single Data) Rechner sind klassische seriell arbeitende von Neumann-Rechner.
MISD (Multiple Instruction Single Data) Rechner bezeichnet man Systeme, bei denen mehrere Prozessoren an einem BUS auf einen gemeinsamen Speicher zugreifen.
SIMD (Single Instruction Multiple Data) Rechner sind vor allem Vektorrechner, die im engeren Sinn keine Parallelrechner sind. Die Arithmetikeinheit (ALU) besitzt zusätzliche Befehle zur Vektormanipulation, speziell wird die Berechnung des Skalarprodukts unterstützt (mit dem sich dann Vektor- & Matrixmultiplikation programmieren lassen). In (oft) einem Taktzyklus werden zwei der ALU übergebene reelle Zahlen multipliziert und zu einem internen Register hinzuaddiert. Setzt man das Register am Anfang auf Null und übergibt dann seriell beide Vektoren, so enthält das Register nach n Taktzyklen (n = Länge des Vektors) das Skalarprodukt beider Vektoren.

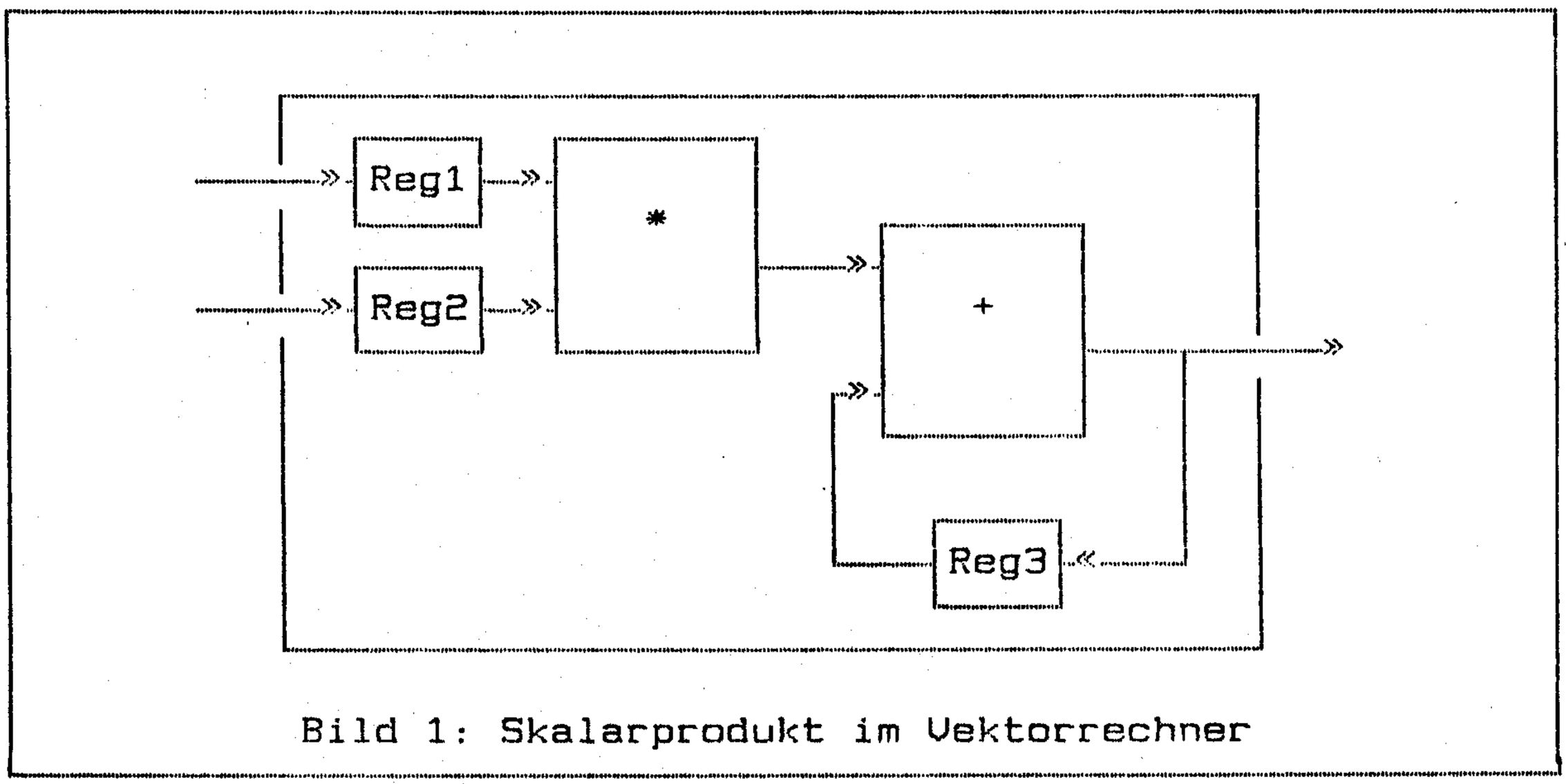


Bild 1: Skalarprodukt im Vektorrechner

Realisiert wird dies durch eine Pipeline, die nicht nur in Vektorrechnern Anwendung findet. Schon einfache Anweisungen, wie die Addition zweier reeller Zahlen, werden häufig in mehreren Phasen (Schritten, Taktzyklen) von je einem dafür ausgelegten Teil der Hardware bearbeitet. Um die anderen Teile nicht Brach liegen zu lassen kann man schon bevor die Anweisung komplett bearbeitet wurde, die nächste in Angriff nehmen und so die Befehle überlappt abarbeiten.
Echt parallelarbeitende SIMD-Rechner werden gerne in der parallelen Komplexitätstheorie (eingeschränkte Klasse von PRAMs) verwendet und wären für viele parallelisierte numerische Algorithmen ausreichend (ein Algorithmus angewendet auf viele verschiedenen Daten).

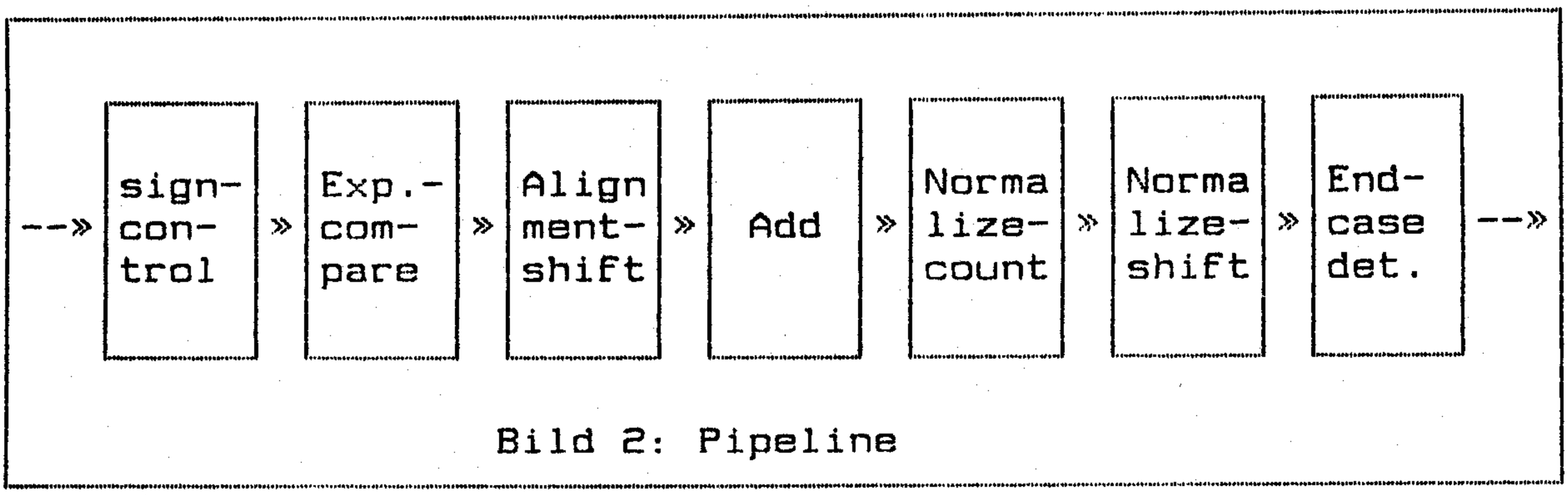


Bild 2: Pipeline

Da aber in der Numerik wesentlich mehr Daten, als Programmmcode zu speichern sind, kann man es sich leisten notfalls auf allen Prozessoren Duplikate des gleichen Programms zu speichern. Man gelangt so zur allgemeinsten Klasse, die MIMD (Multiple Data Multiple Instruction) Rechner, die aus mehreren Prozessoren bestehen mit eigenem lokalem Speicher für Programme und Daten. Im folgenden werden nur noch MIMD-Rechner betrachtet.
Weiterhin können die einzelnen Prozessoren synchron (d.h. gleichgetaktet) oder asynchron arbeiten. Da synchrone Rechner zwar gerne Modelle theoretischer Untersuchungen sind, aber kein reelles Abbild besitzen, werden im folgenden nur noch asynchrone Rechner betrachtet.
Wenn mehrere Prozessoren an einem Problem arbeiten, müssen sie von Zeit zu Zeit Daten austauschen. Geschieht dies über einen gemeinsamen (d.h. allen Prozessoren verfügbaren) Speicher, so spricht man von einem speichergekoppelten System (Prozessoren hängen an gemeinsamen BUS). Da aber (aus technischen Gründen) immer nur eine beschränkte Zahl von Prozessoren (meist nur einer) gleichzeitig auf den globalen Speicher zugreifen kann (auch wenn diese auf verschiedene Zellen zugreifen) stellt diese Kopplung für wachsende Prozessorzahl eine zunehmende Engstelle dar. Für beliebig viele Prozessoren tauglich ist eine Verbindung der Prozessoren durch (serielle) Schnittstellen, auf denen Nachrichten ausgetauscht werden können (Botschaftorientiert). Da aber nicht jeder Knoten mit jedem verbunden werden kann, und eine flexible Verschaltung per Software nur bedingt möglich ist, ist die Frage der Verbindungstopologie entscheidend.

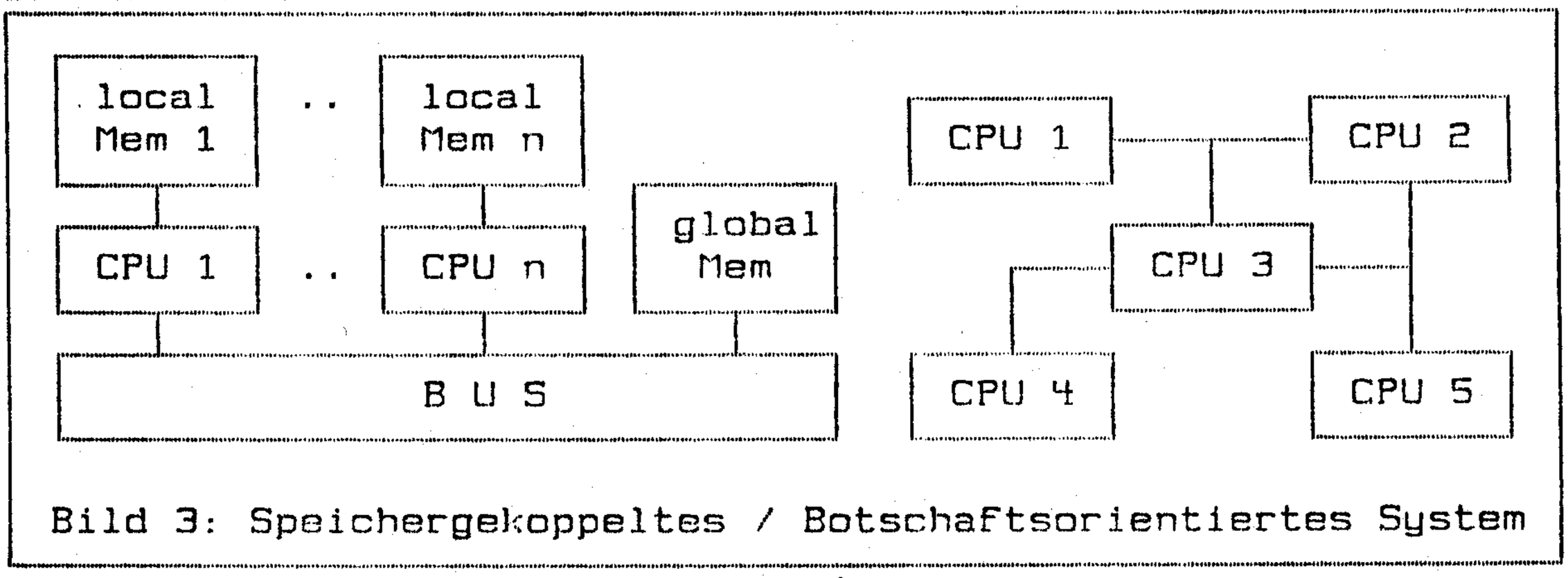


Bild 3: Speichergekoppeltes / Botschaftsorientiertes System

Verbindungsstruktur:
Als Verbindungsstruktur hat sich vor allem der Hypercube bewährt. Hier werden 2^n Prozessoren mit n-stelligen Binärzahlen

durchnummeriert. Zwei Prozessoren sind genau dann miteinander verbunden, wenn deren binäre Nummer sich in höchstens einem Bit unterscheidet (Hammingabstand 1). Die Vorteile dieser Topologie sind:

1. einfach (zu verstehende) Verbindungsstruktur.
2. Verbindungszahl proportional $n2^n$, wächst also nur leicht überllinear mit der Prozessoranzahl.
3. Die größte Entfernung zwischen 2 Prozessoren ist proportional n, wächst also nur logarithmisch mit der Prozessorzahl.
4. Die wichtigsten Strukturen wie k-dimensionale Netze und Bäume sind als Substrukturen enthalten.

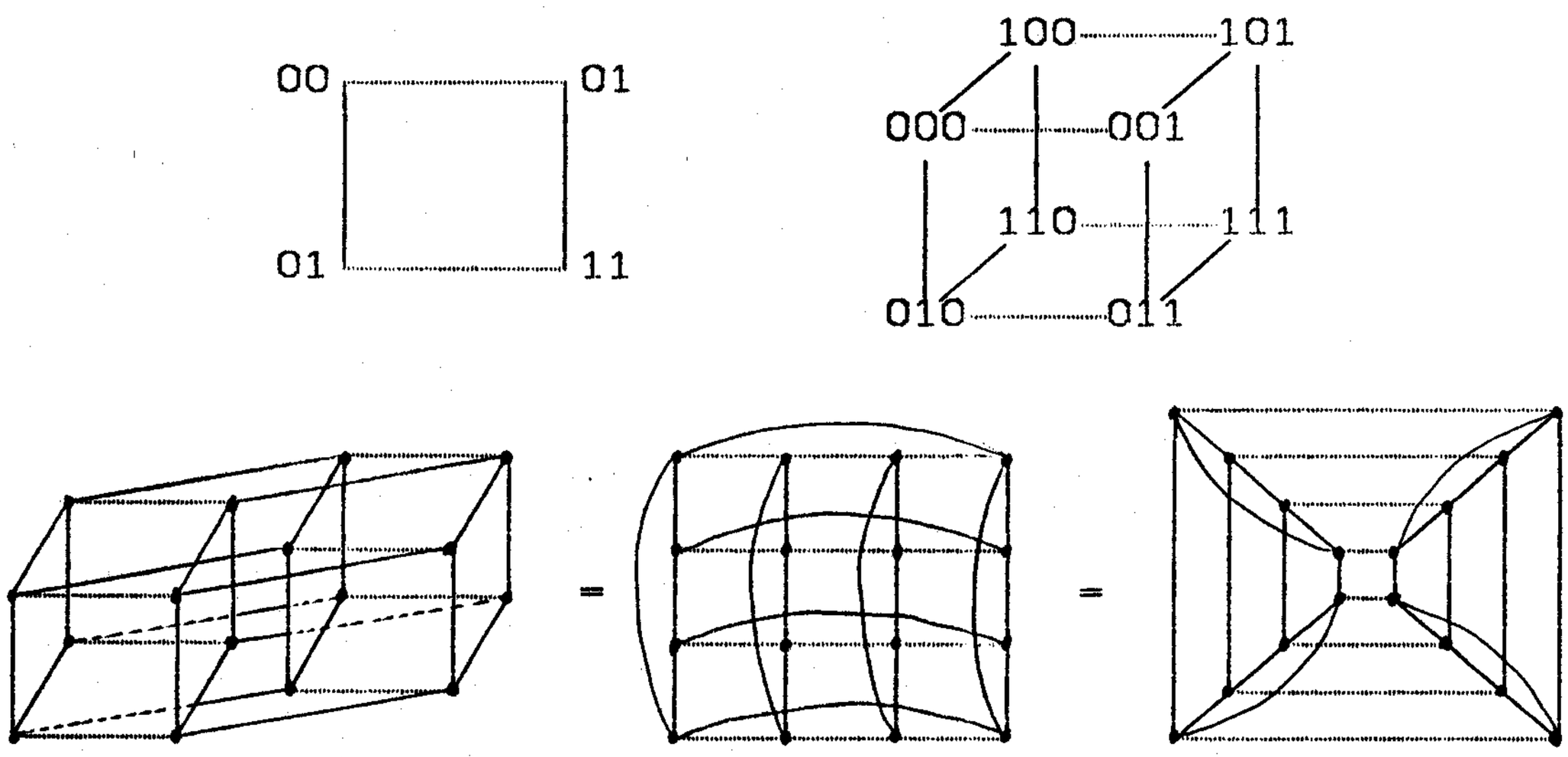


Bild 4: Hypercubes der Dimension 2, 3 und 4

Für lineare Netze, d.h. eine lineare Durchnummerierung der Knoten, verwende man einen 2^n-Gray-Code. Dieser ordnet jeder n-stelligen Binärzahl (also jedem Prozessor) eine (Dezimal)-Zahl in der Weise zu, daß Prozessoren aufeinanderfolgender (Dezimal)-Zahlen miteinander verbunden sind. Für k-dimensionale Netze mit Kantenlängen $2^{n_1} \times .. \times 2^{n_k}$ verwende man k-Tupel von 2^{n_i}-Graycodes ($1 \leq i \leq k$), die als n-stellige Binärzahlen interpretiert werden können. Durch zyklische Gray-Codes erhält man, falls geschünscht, eine Wrapped-Around-Verbindung, d.h. daß Prozessoren auf gegenüberliegenden Seiten verbunden sind. Man erhält so k-dimensionale Tori.

Einen Baum der Tiefe n und maximalem Verzweigungsgrad n erhält man, indem man zwei Knoten genau dann miteinander verbindet, wenn die Binärnummer des einen Knoten (Vater) aus der Nummer des anderen (Sohn) durch löschen des letzten 1-Bits hervorgeht.

00		000		000.00	000.01	000.11	000.10
01		001		001.00	001.01	001.11	001.10
11	×	011	=	011.00	011.01	011.11	011.10
10		010		010.00	010.01	010.11	010.10
		110		110.00	110.01	110.11	110.10
		111		111.00	111.01	111.11	111.10
		101		101.00	101.01	101.11	101.10
		100		100.00	100.01	100.11	100.10

Bild 5: $2^2, 2^3$ und $(2^2, 2^3)$-Gray-Code

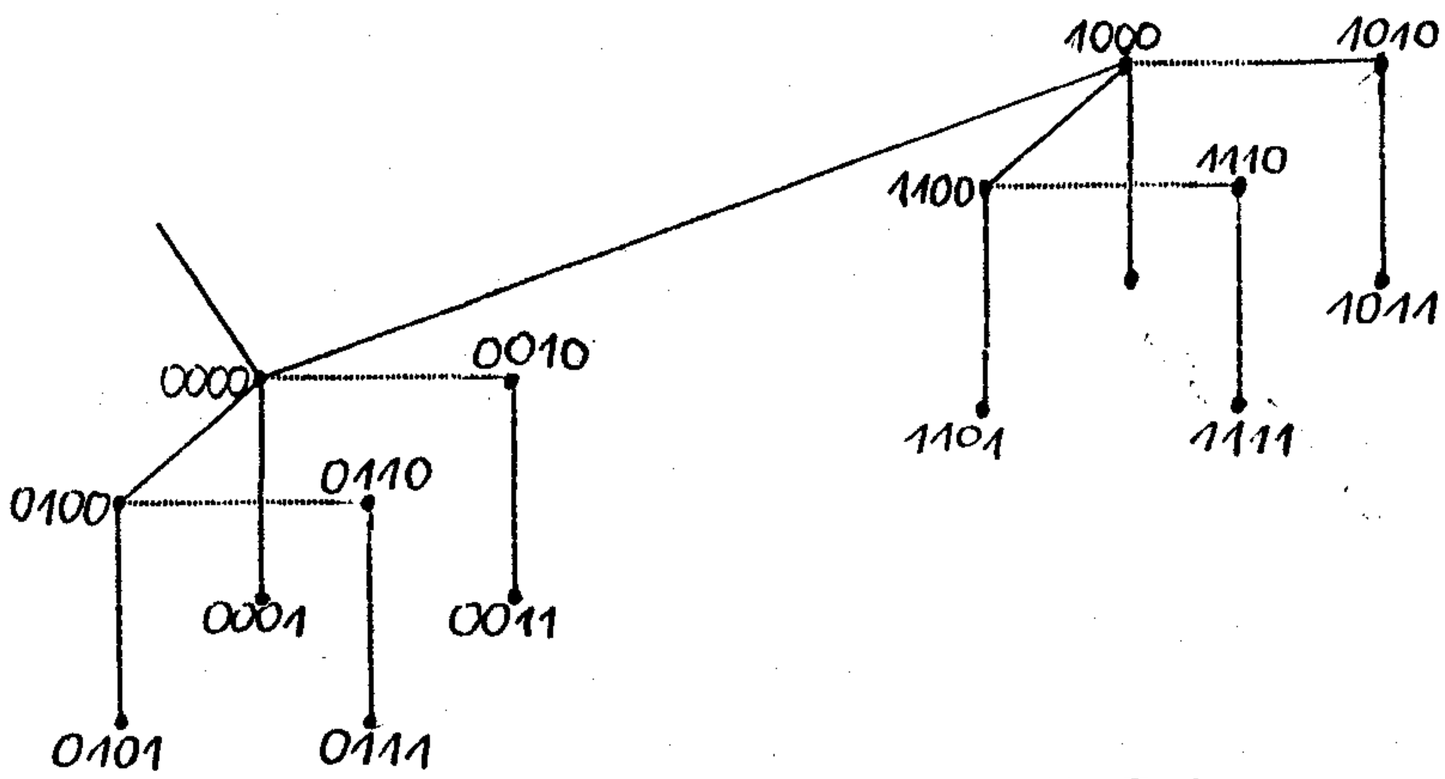


Bild 6: Baum in einem 4-Cube

Laufzeit paralleler Algorithmen:
Arbeitet man mit P Prozessoren statt mit einem, so kann man maximal eine Geschwindigkeitssteigerung (Speedup) um den Faktor S=T(1)/T(P) = 'Zeit die ein Prozessor zum Lösen des Problems benötigt' / 'Zeit die P Prozessoeren dazu benötigen' erwarten. In der Praxis wird dieser Wert aus folgenden Gründen nicht erreicht. Zum einen müssen die Prozesse durch einen Master-Prozess erzeugt und auf die verschiedenen Prozessoren verteilt und die Ergebnisse wieder abgeholt werden; hier spricht man von statischem Overhead, zum anderen kostet der Datentransfer während der Laufzeit oft erhebliche Zeit, da die Transferraten (Byte/sec = Baud) um den Faktor 100-1000 niedriger als die Befehlszykluszeit (in der Numerik im wesentlichen die Fließkommaoperationen/sec = Flops) sind. Hinzu kommen noch Totzeiten der Prozessoren, in denen sie nur auf die Ergebnisse anderer Prozessoren warten; hier spricht man von dynamischem Overhead. Diese sogenannte Wasted Time ergibt sich aus W=T*T(P)-T(1). Die Effizienz eines parallelen Algorithmus ist gegeben durch E=S/P die im besten Fall (ohne Overhead) 1 ist. Manchmal passiert es auch, daß ein (vielleicht optimaler) serieller Algorithmus ungeeignet zur Parallelisierung ist so, daß auf einen (vielleicht sogar Ornungsmäßig) langsameren zurückgegriffen werden muß, der sich parallelisieren läßt.

Level der Parallelisierung:
Algorithmen lassen sich auf verschiedenen Programmierebenen parallelisieren. Auf oberster Ebene befindet sich die Paralle-

lisierung mittels Tasks (Task=Prozesse). Die Tasks entsprechen auf der Softwareseite den nebeneinander arbeitenden Prozessoren auf Hardwareseite. Ein Taskmanager verteilt die Prozesse automatisch auf die Prozessoren. Der Vorteil dieser Methode liegt darin, daß Änderungen in der Hardwarekonfiguration den Softwarehersteller nicht berühren; Werden mehr Prozessoren zur Verfügung gestellt so wird das Programm (ohne irgendeine Modifikation), solange es genügend viele Tasks enthält, entsprechend schneller. Da bei sehr vernetzten Problemen die geschickte Aufteilung der Tasks auf die Prozessoren wesentlich zur Effizienz beiträgt, ist diese Methode ungeeignt.
Die Aufteilung auf unterster Ebene, auf Instruction-Level zielt in Richtung Datenflußrechner und ist heute noch eher Gegenstand theoretischer Untersuchungen.
Für uns am interessantesten ist die Parallelisierung auf Loop-Level, da Numerikprogramme von Loops ziemlich übersät sind. Oftmals sind die Berechnungen in den einzelnen Loop-durchläufen unabhängig voneinander (oder können auf einfache Weise unabhängig gemacht werden), d.h. daß die Reihenfolge der Berechnung keine Rolle spielt. Ist dies der Fall, so können sämtliche Durchläufe auch parallel berechnet werden.
Der Einfachheit halber geht man in der Weise vor, daß man einen Algorithmus zuerst vollständig parallelisiert (oder zumindest weitgehend), um ihn dann auf die zur Verfügung stehenden Prozessoren so zu verteilen, daß die Kommunikation zwischen verschiedenen Prozessoren auf ein Minimum reduziert wird, da jene (zeitlich) teuer sind.

3. Existierende & geplante Parallelrechner

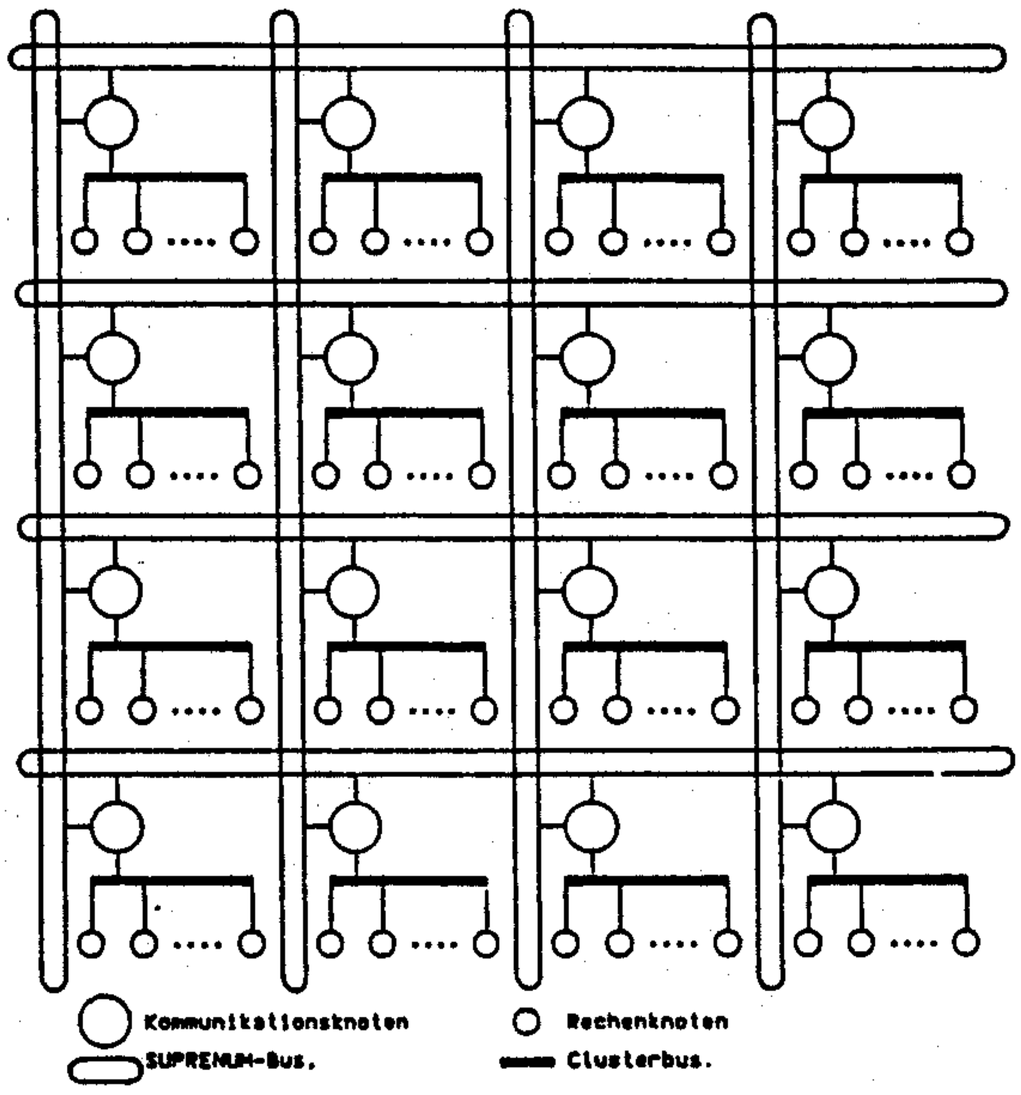
Cray:
Die Cray 2 ist ein Vektorrechner mit 4 Background-CPUs in ECL-Logik mit 4,1 ns Zykluszeit und 64KByte lokalem Speicher. Sie besitzt einen 64Bit-Foreground-Prozessor mit 2GByte RAM und zählt mit 2GFlops zu einem der schnellsten heutigen Rechner (trotz geringer Parallelisierung). Sie hat die Form einer Tonne um die internen Kabellängen kurz halten zu können und ist im Gegensatz zu ihrem Vorgänger Flüssigkeitsgekühlt. Wartungsexperten bedauern sehr, daß das Konzept der 'integrierten Sitzbank' der Cray 1 (aus Kostengründen ??) nicht übernommen wurde.

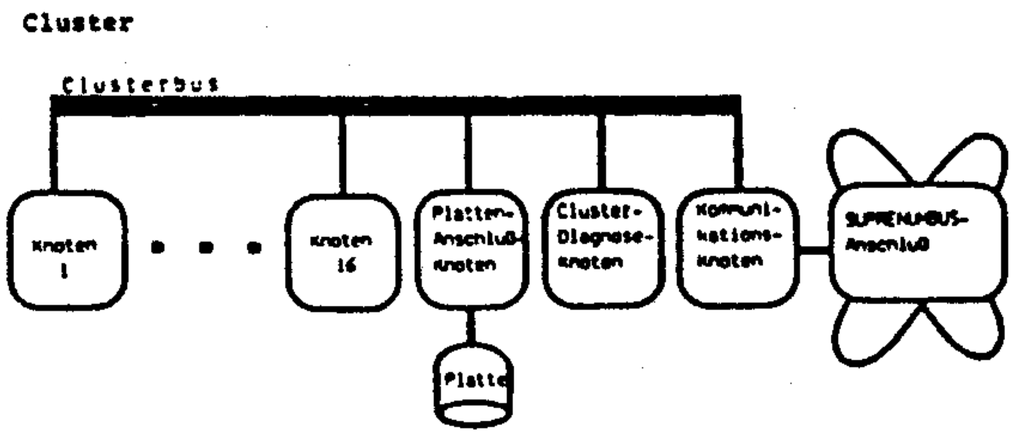
Transputer:
Transputer sind RISC-Prozessoren von der Firma INMOS die speziell für Parallelrechner entwickelt wurden. Die Prozessoren besitzen je 4 Links (Serielle Schnittstellen) mit denen sie zu beliebigen Topologien (natürlich mit maximalem Verzweigungsgrad 4) verbunden werden können. Da Links & Coprozessor im T818 bereits integriert sind, kann ein Rechnerknoten mit relativ wenigen Bauteilen aufgebaut werden. Erweiterungskarten mit je 4 Transputern gibt es schon für PCs.

Alliant:
Alliant entwickelt auf der Basis des Vektorprozessors i860 von Intel einen Parallelrechner. Allein ein i860 (64Bit-Prozessor + Coprozessor) weist schon die Leistung einer Cray1 auf. Der i860 soll auf dem Gebiet der Großrechner der Standardprozessor (PAX-Standard) werden, so wie die 80x86 im PC-Bereich

Suprenum:

Der Suprenum-Rechner ist eine Entwicklung der GMD (Gesellschaft für Mathematik und Datenverarbeitung). Er besteht aus 16 Clustern, die in einer Hypercube-Topologie (der Dimension 4) verbunden sind. Jeder Cluster besteht aus 16 32-Bit Prozessoren von Motorola (68020) plus Coprozessor (68882) plus je 8 MByte lokalem Speicher, die über einen Bus miteinander und verschiedenen anderen Komponenten verbunden sind. Die Verwendung von Standardchips soll dieses System relativ billig machen. Es wurde versucht durch eine heterogene Struktur die Nachteile von BUS- bzw. botschaftsorientierten Systemen zu vermeiden. Entwickelt wurde der Rechner vor allem für Mehrgitterverfahren.





Die 16 Cluster des Suprenum-Prototyp-Rechners sind matrixförmig über das Suprenum-Bus-System miteinander verbunden.

Bild 7: Suprenum-Rechnerarchitektur

4. Der Multigrid-Algorithmus

Als Beispiel zur Parallelisierung soll auch hier das Modellproblem, die Laplace-Differenzialgleichung

$$\frac{\partial^2 \Phi}{\partial x^2} + \frac{\partial^2 \Phi}{\partial y^2} = 0$$

auf einem Rechteckgebiet mit $\Phi(x,y)=\Phi_0(x,y)$ als Randbedingung betrachtet werden. Diskretisiert man die Dgl auf N*N Punkten, so geht die Dgl in eine Differenzengleichung der Form

$$\Phi_{i+1,j} + \Phi_{i-1,j} + \Phi_{i,j+1} + \Phi_{i,j-1} - 4\Phi_{i,j} = 0 \text{ mit } 0<i,j<n \text{ und}$$

$\Phi_{0,j}$ $\Phi_{n,j}$ $\Phi_{i,0}$ $\Phi_{i,n}$ gegeben durch die Randbedingung

über. Daraus läßt sich die Iterationsvorschrift

$$\Phi_{i,j}^{(n+1)} := (\Phi_{i+1,j}^{(n)} + \Phi_{i-1,j}^{(n)} + \Phi_{i,j+1}^{(n)} + \Phi_{i,j-1}^{(n)})/4$$

gewinnen.

Dies ist das Jakobi-Verfahren. Ersetzt man den alten Wert von Φ nicht ganz durch den neuen, sondern gewichtet ihn mit dem alten, so läßt sich die Konvergenz erheblich beschleunigen.

$$\Phi_{i,j}^{(n+1)} := \omega(\Phi_{i+1,j}^{(n)} + \Phi_{i-1,j}^{(n)} + \Phi_{i,j+1}^{(n)} + \Phi_{i,j-1}^{(n)})/4 + (1-\omega)\Phi$$

Doch auch dieses Verfahren konvergiert relativ schlecht gegen die Lösung $O(N \log N)$, da (bildet man das Fourierspektrum) je nach Wahl von ω entweder die hochfrequenten Anteile oder die Niederfrequenten schlecht konvergieren. Die Idee ist nun ein paar Schritte mit für die hochfrequenten Anteile optimalem ω (≈ 0.7) zu iterieren und dann die niederfrequenten Anteile zu verbessern. Dazu kann auf ein gröberes Gitter (Restriktion) übergegangen werden, bzgl dessen die niederfrequenten Anteile aufgrund des gröberen Gitters wieder hochfreuent sind und so das Verfahren iteriert werden kann. Durch Interpolation und nochmalige Anwendung ein paar Jakobi-Schritte erhält man nun eine verbesserte Lösung auf dem feinen Gitter. Dies ist der Mehrgittter-V-Zyklus.
Um einen Startwert für den V-Zyklus zu erhalten zieht man sich nahezu am eigenen Kopf auf dem Sumpf. Man wählt ein sehr grobes Gitter mit Startwert $\Phi \equiv 0$. Wendet man nun den V-Zyklus an und interpoliert auf ein nächst feineres Gitter, so erhält man einen Startwert für dieses feinere Gitter. So kann successiv durch V-Zyklen die Lösung auf dem feinsten Gitter approximiert werden.
Details können in [2] nachgelesen werden. Trotz dieses recht aufwendig anmutenden Verfahren hat es nur die Ordnung $O(N^2)$ und ist somit eines der schnellsten.

5. Parallelisierung des Multigrid-Algorithmus

Einen (allein aus physikalischen Gründen) geeigneten Ansatz zur Parallelisierung des Multigrid-Algorithmus bietet die Parallelverarbeitung der einzelnen Gitterpunkte. Dazu werden parallele Varianten des Jakobi-Verfahrens (mit Relaxation), der Interpolation, der Restriktion und eventuell einer globalen Fehlerbestimmung benötigt.

Paralleler Gauß-Seidel-Algorithmus:
Das Jakobi-Verfahren ist direkt geeignet jeden Gitterpunkt parallel zu berechnen, falls man die neu berechneten Werte zuerst in einem weiteren Array zwischenspeichert. Würde man die Werte direkt überschreiben, so kann es passieren, daß ein Nachbarpunkt nicht den alten, sondern den neuen Wert zur Berechnung verwendet. Wie sich zeigen läßt ist dies aber sogar von Vorteil, wenn teilweise zur Berechnung schon die neuen Werte verwendet werden. Die Iterationsformel

$$\Phi_{i,j}^{(n+1)} := \omega(\Phi_{i+1,j}^{(n)} + \Phi_{i-1,j}^{(n+1)} + \Phi_{i,j+1}^{(n)} + \Phi_{i,j-1}^{(n+1)})/4 + (1-\omega)\Phi$$

von Gauß & Seidel konvergiert doppelt so schnell wie das Jakobiverfahren, läßt sich nur leider nicht mehr vollständig parallelisieren.
Die gebräuchlichte Variante ist die Red-Black-Färbung. Das Gitter wird Schachbrettmusterartig eingefärbt (man beachte: rot, nicht weiß) und zuerst alle roten Punkte iteriert

$$\Phi_{i,j}^{(n+1)} := (\Phi_{i+1,j}^{(n)} + \Phi_{i-1,j}^{(n)} + \Phi_{i,j+1}^{(n)} + \Phi_{i,j-1}^{(n)})/4 \text{ mit } i+j \text{ gerade}$$

und dann alle schwarzen

$$\Phi_{i,j}^{(n+1)} := (\Phi_{i+1,j}^{(n+1)} + \Phi_{i-1,j}^{(n+1)} + \Phi_{i,j+1}^{(n+1)} + \Phi_{i,j-1}^{(n+1)})/4 \text{ mit } i+j \text{ ungerade,}$$

wobei die neuen roten Werte verwendet werden.

Parallele Restriktion und Interpolation:
Restriktion und Interpolation bereiten keine Schwierigkeiten bei der Parallelisierung. Man verfeinert das Gitter meist, indem man die Zeilenzahl gleichzeitig mit der Spaltenzahl verdoppelt und somit in jedem Schritt die vierfache Punktzahl erhält. Manchmal erhöht man Zeilen- und Spaltenzahl alternierend so, daß die Netzgröße um den Faktor 2 zunimmt. Die einfachste Art der Interpolation im ersten Fall ist die lineare Interpolation

$$\Phi_{2i,2j}^{(h)} := \Phi_{i,j}^{(2h)} \qquad \Phi_{2i+1,2j+1}^{(h)} = (\Phi_{i,j}^{(2h)} + \Phi_{i,j+1}^{(2h)} + \Phi_{i+1,j}^{(2h)} + \Phi_{i+1,j+1}^{(2h)})/4$$

$$\Phi_{2i,2j+1}^{(h)} = (\Phi_{i,j}^{(2h)} + \Phi_{i,j+1}^{(2h)})/2 \qquad \Phi_{2i+1,j}^{(h)} := (\Phi_{i,j}^{(2h)} + \Phi_{i+1,j}^{(2h)})/2$$

Restriktion erfolgt am einfachsten durch Weglassen der nichtbenötigten Punkte

$$\Phi_{i,j}^{(2h)} := \Phi_{2i,2j}^{(h)}$$

oder durch Mittelbildung.

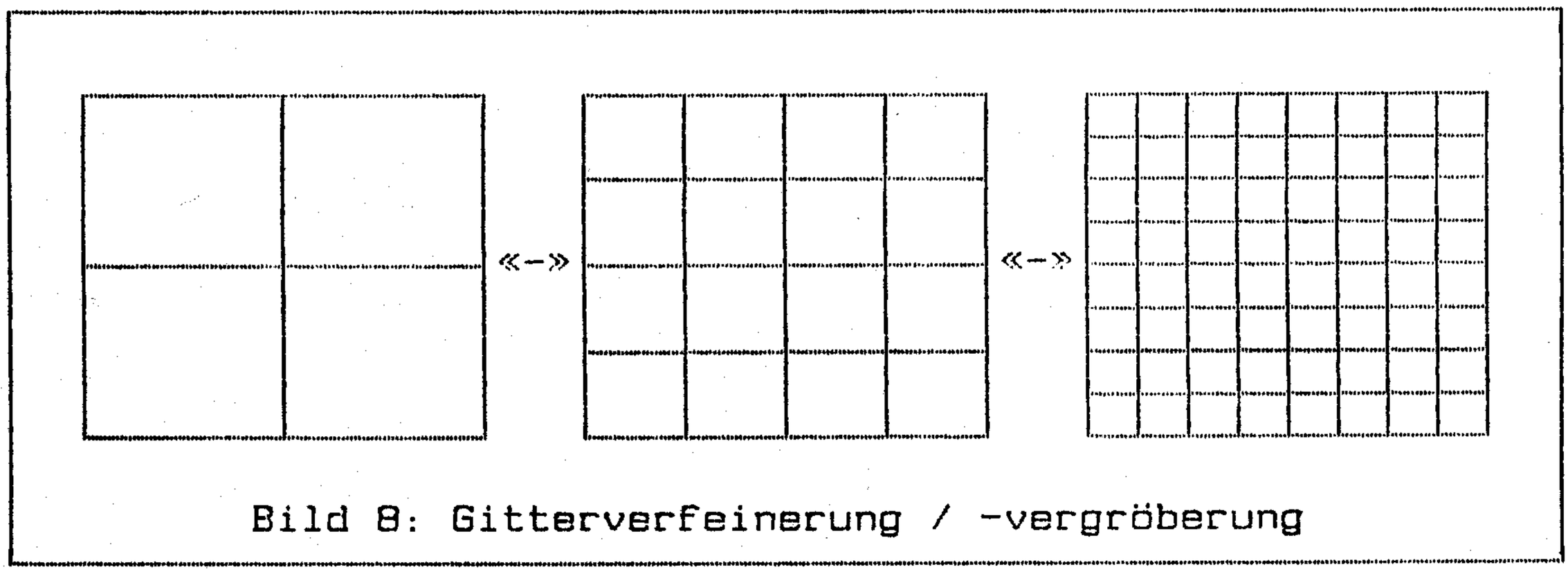


Bild 8: Gitterverfeinerung / -vergröberung

6. Aufteilung des Gitters auf die Prozessoren

Da i.a. wesentlich weniger als N^2 Prozessoren zur Verfügung stehen muß das N*N-Gitter in $P < N^2$ Teile zerschnitten werden. Dabei sollte auf folgendes geachtet werden:

1. Ähnliche Gitterverteilung für die verschiedenen Verfeinerungsstufen.
2. Geringe Kommunikation zwischen den Prozessoren.
3. Gleichmäßige Arbeitsverteilung.

In unserem Fall sind die Bedingungen leicht zu erfüllen. Man teilt das Gitter in P Quadrate, wobei P Quadratzahl sei. Da der Rechenaufwand pro Prozessorknoten porportional zur Anzahl der Gitterknoten und der Kommunikationsaufwand zwischen den Prozessoren proportional zur Zahl der Randknoten ist bieten Quadrate (Kreise gehen leider nicht) die beste Effizienz.
Um einen einheitlichen Algorithmus auch für Randpunkte zu gewähren, ist es von Vorteil zusätzlich die Randpunkte der Nachbarprozessoren (da diese zur Berechnung benötigt werden) als

Kopie zu speichern und nach einem Iterationsschritt durch Austausch die Konsistenz wieder herzustellen.

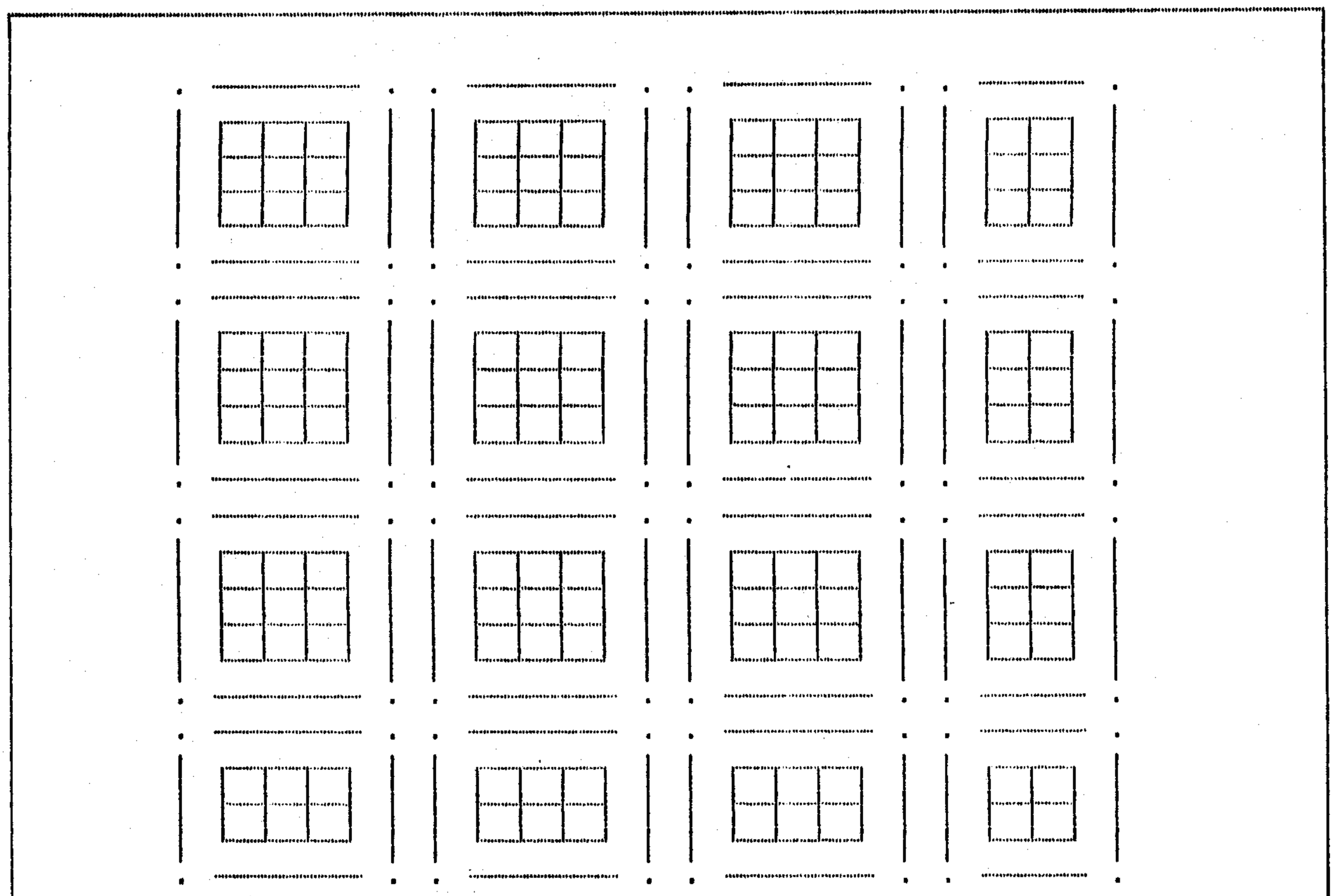


Bild 9: Aufteilung der Prozessoren

Eine eventuell benötigte globale Fehlerbestimmung kann wie folgt unter Verwendung der Baum-Sub-Struktur bestimmt werden:

1. Berechne Fehlersumme der eigenen Gitterpunkte
2. Lese Fehlersumme aller Söhne und addiere sie hinzu
3. Schicke Summe an Vater-Knoten falls Vater existiert
4. Lese Summe von Vater-Knoten falls Vater existiert
5. Schicke Summe an alle Söhne

Nach $O((\log P)^2)$ Zeit enthält 'Summe' die globale Fehlersumme.

Da der Multigridalgorithmus nur eine mehmalige Anwendung von Vergröberung, Verfeinerung, Relaxation, Transfer und ev. Fehlerbestimmung ist, ist damit auch dieser parallelisiert.

7. Literaturverzeichnis

[1] Stoer Bulirsch: Numerische Mathematik 2 Kap. 8 3.Aufl.: Springer-Verlag,1990

[2] Hackbush, W., Trottenberg, U. (eds.): Multigrid Methods. Lecture Notes in Mathematics. Vol. 960. Berlin-Heidelberg-New York: Springer-Verlag 1982

[3] O.McBryan und E.F. Van de Velde: The Multigrid Method on parallel Processors, in Lecture Notes in Mathematics 1228: Multigrid Methods II, Proceedings of the Conference Held at Cologne, October 1-4,1985

[4] D.J.Evans: Parallel S.O.R. Iterative Methods, Parallel Computing, vol. 1,pp.3-18,1984

1. Einleitung

o- Numerische Algorithmen zur parrallelierung geeignet

2. Parallele Rechnerarchitekturen

o- SISD,SIMD,MISD,MIMD-Rechner
o- Vektorrechner: Skalarprodukt,2 Ops parallel, sonst seriell
o- Synchrone / Asynchrone Parallelrechner
o- Echte Parallelrechner
o- Pipeline: Bedienzeit, Durchsatz, Assembler schwierig.
o- Kommunikation: speichergekoppelt / Botschaftsorientiert
o- Verbindungsstrukturen: BUS,gem Speicher, ser. Schnittstelle
o- Hypercube -> Netze (Gray-Code), Baum,
o- Speedup, Effizienz, Stischer/dyn. Overhead,Wasted Time, Thystone, Whetstone,
o- Task- /Loop-/ Instructionlevel-Parallelisierung
o- Totale Parallelisierung, dann Verteilen aus Rechnerknoten
- Semaphore Prozesse, Linpack, Eispack,

3. Existierende & geplante Parallelrechner

o- Cray
o- Alliant
o- Suprenum
o- Transputer

4. Der Multigrid-Algorithmus

o- Modellproblem, Herleitung
o- Jakobi-Verfahren, Gauß-Seidel
o- Restriktion, Interpolation, Datentransfer

5. Parallelisierung des Multigrid-Algorithmus

o- parallele Varianten von Gauß-Seidel
 - Red-Black- Diagonal-Färbung
o- parallelisierung von Interpol., Restr.,Gauß-Seidel

6. Aufteilung des Gitters auf die Prozessoren

o- Minimale Grenzen, Gleiche Rechenlast
- $(2^n+1)^2$ Gitterpunkte
o- Verdoppelung der Grenzen
o- Berechnung des globalen Fehlers
 - gröbstes Gitter = Prozessorgitter
o- Lowlevel Blocking
o- Simulationsergebnisse aus Artikel