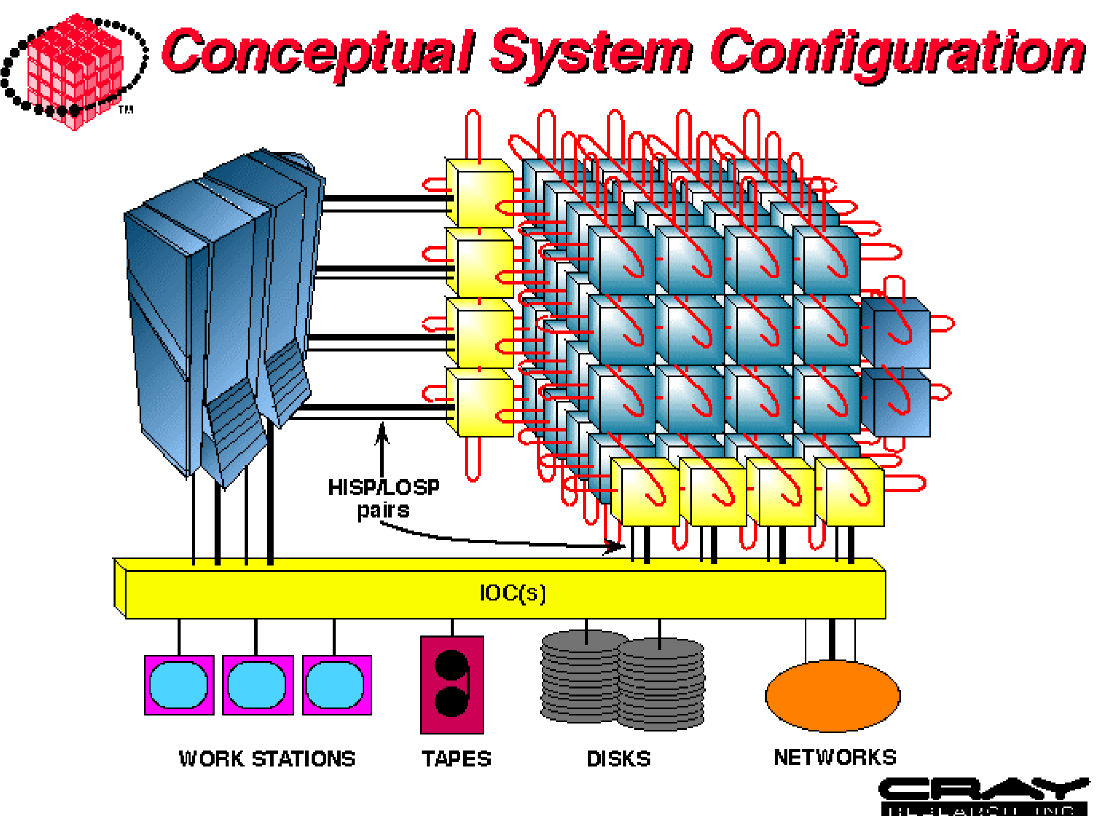
Paralleldatenverarbeitung



Parallele Rechnerarchitekturen
und parallele Programmierung

Günther Fröhlich

1 EINLEITUNG 5

2 AKTUELLE PRESSEMELDUNGEN 9

3 AUS TOP500 VON NOVEMBER 2001 20

4 STRUKTUREN VON PARALLELRECHNER-ARCHITEKTUREN 22

4.1 Einordnung von Parallelrechnern - Flynnsche Klassifikation 22

4.2 Speichergekoppelte Multiprozessoren 25

4.3 Nachrichtengekoppelte Multiprozessoren 26

4.4 Ebenen der Parallelität 27

4.5 Techniken der Parallelarbeit 31

4.6 Beschleunigung und Effizienz von parallelen Systemen 32

5 KONZEPTE DER PARALLELARBEIT 38

5.1 Parallelität und die sie einschränkende Abhängigkeiten 38

5.1.1 Implizite Programmparallelität 39

5.1.2 Explizite Datenparallelität 42

5.2 Die Anweisungsebene 43

5.2.1 Register-Optimierung 43

5.2.2 Schleifen-Aufrollen (loop unrolling) 43

5.2.3 Loop Jamming 44

5.2.4 Erkennen der Parallelität auf Anweisungsebene über Basisblöcke hinweg 45

5.3 Die Prozeßebene 46

5.4 Nutzung der Anweisungs-Parallelität 46

5.4.1 Schleifen-Parallelisierung 46

5.5 Synchronisation von Parallelarbeit 49

5.5.1 Fork/Join-Modell nach Pancake: 49

5.5.2 Reusable-Thread-Pool-Modell nach Pancake: 50

5.5.3 Lock-Step-Betrieb und Barrieren-Synchronisation 50

5.5.4 Synchronisation kooperierender Prozesse 51

6 SUPERSKALARE PROZESSOREN UND VLIW MASCHINEN 53

6.1 Superskalare Prozessoren 53

6.1.1 Alpha 21064 54

6.1.2 Grenze der Superskalarität 56

6.2 VLIW-Maschinen 57

6.2.1 Mängel der Architektur 58

7 VEKTORMASCHINEN UND ANORDNUNGEN VON RECHENELEMENTEN 60

8 DATENFLUßARCHITEKTUREN 62

8.1 Überblick 62

8.2 Grundlagen der Datenflußarchitekturen 63

9 GRUNDLAGEN DER MIMD-PARALLELRECHNER 66

9.1 Die Hauptformen von MIMD-Architekturen 66

9.2 Steigerung der Effizienz von Multiprozessorsystemen mit gemeinsamem Speicher: Snoopy Logic und Mesi-Protokoll 74

9.3 Programmiermodell und abstrakte Maschine 75

9.4 Das Kommunikationssystem 77

9.5 Kommunikationslatenz 78

10 VERBINDUNGSNETZ-TOPOLOGIEN 81

10.1 Verbindungsart 82

10.2 Wegsteuerung 83

10.3 Arbeitsweise 83

10.4 Art der Topologie 83

10.4.1 Realisierungskriterien 83

10.4.2 Übertragunsstrategie bei Paketvermittlung 84

10.4.3 Busverbindungen 86

10.4.4 Ringstrukturen 86

10.4.5 Gitterverbindungen 88

10.4.6 Mehrstufige Verbindungsnetze 88

10.4.7 Hyperwürfel 92

10.4.8 Hierarchien von Crossbars 92

10.4.9 Eins-zu-N-Kommunikation in Verbindungsnetzen 94

11 GRUNDLAGEN PARALLELER SOFTWARE 95

11.1 Begriffsbestimmungen 95

11.2 Sprachansätze zur Programmierung von Parallelrechnern 97

11.3 Prozesse und Threads 99

11.4 Synchronisation und Kommunikation über gemeinsame Variable 101

11.4.1 Grundlagen 101

11.4.2 Schloßvariable 103

11.4.3 Semaphor 104

11.4.4 Bedingter kritischer Bereich 108

11.4.5 Bedingungsvariable 109

11.5 Posix-Thread Beispiel in C 111

12 NACHRICHTENORIENTIERTE PROGRAMMIERUNG 120

12.1 Programmbeispiel Matrixmultiplikation 120

12.2 Nachrichtengekoppelte Programmierschnittstellen 122

12.3 Implementierung nachrichtengekoppelter Programme 123

12.4 MPI Der Message Passing Interface Standard 123

12.4.1 MPI Beispielprogramm : Berechnung von Pi 123

12.5 Datenparalle Programmierung: Open MP 124

12.5.1 OpenMP Beispielprogramm 124

13 BIBLIOGRAPHIE 127

14 GLOSSAR 128

15 INDEX 131

1 Einleitung

Um die Bedeutung der Vorlesung *Paralleldatenverarbeitung - Parallele Rechnerarchitekturen und parallele Programmierung* zu unterstreichen, wäre es sehr schön, könnte man auf einen sprunghaften Anstieg beim Einsatz von Supercomputern mit massiv parallelen Architekturen in den letzten Jahren verweisen. Das Thema wäre dann offensichtlich aktuell und, da neue Programmiermodelle dafür entwickelt werden, für die Qualifikation von Informatikern von besonderem Interesse. In der Realität ist es nach wie vor so, daß massiv parallele Rechner in erster Linie nur in Forschungsinstituten und im privatwirtschaftlichen Bereich nur in der Luft- und Raumfahrtindustrie sowie in der Automobil- und Rüstungsindustrie eingesetzt werden. Supercomputer nehmen also heute immer noch eine relativ exotische Stellung ein.

Dennoch wird in allen Klassen von Computern die Parallelisierung der Programmausführung als Mittel der Leistungssteigerung vorangetrieben. Deshalb ist die Vorlesung - wie im Titel bereits deutlich wird - möglichst allgemein aufgebaut.[1]

Andererseits ist es richtig, daß im Bereich der Forschung der Bedarf nach immer leistungsfähigeren Computern in den letzten Jahren stetig gestiegen ist. Simulationsaufgaben auf den Gebieten der Strömungsmechanik, der Wettervorhersage und des Klimageschehens, der Meeresströmungen, der Entwicklung chemischer Verbindungen, der biologischen und medizinischen Forschung einschließlich der Gentechnik verlangen mittlerweile Rechenleistungen im Teraflop-Bereich. Und diese Rechenleistungen sind nur durch massiv-parallele Rechner zu erzielen. Unter dem Titel *Grand Challenges* (große Herausforderungen) wurde Anfang der neunziger Jahre von der Regierung der Vereinigten Staaten ein groß angelegtes, sich über mehrere Jahre erstreckendes und einige Millirdaen Dollar umfassendes Forschungsförderungsprogramm begonnen. Ziel des Programms ist die Weiterentwicklung des Supercomputing, die als notwendig erachtet wird, um die großen Probleme unserer Zeit – z.B. das Überle-

---

[1] Der typische Einsatzbereich von Supercomputern ist die Simulation komplexer mathematischer und physikalischer Probleme. Ein aktuelles Beispiel hierfür bilden die Bestrebungen, die Simulation von Kernreaktionen zwecks Überprüfung der Zuverlässigkeit und Entwicklung von neuen Atomwaffen zu perfektionieren. („Computersimulation wird ein wesentliches Mittel sein, um die Sicherheit , Zuverlässigkeit und Effektivität der nuklearen Abschreckung der USA zu gewährleisten.“. Victor Reis vom US-Energieministerium, zitiert in c´t, Februar 1996, S. 71) Das wesentliche Ziel der hierfür in Amerika ins Leben gerufenen Accelerated Strategic Computing Initiative *ASCI* (Beschleunigte Strategische Computerinitiative) ist die Fähigkeit, zuverlässige Voraussagen über das gesamte System einer Atomwaffe machen zu können. Verbunden wird dieses Ziel mit dem Ansinnen, die Leistungfähigkeit herkömmlicher Computer um den Faktor 1000 zu erhöhen. Die US-Regierung erteilte nach einer Ausschreibung im September 1995 der Firma Intel den Auftrag zum Bau eines Computers, der nach Intel-Angaben die gleiche Leistung wie alle rund 50 000 weltweit installierten großen Mainframes zusammengenommen haben soll. Dafür sollen 9000 Pentium Pro Prozessoren parallel arbeiten. Dieser Computer wird eine Spitzenleistung von 1,8 Teraflops (=1,8 Billionen Fließkommaoperationen pro Sekunde) erreichen und einen Arbeitsspeicher von 262 Gigabytes haben. Der neueste Cray-Rechner T3E soll 1,2 Teraflops leisten und wird unter anderem an die französische Atomenergiekommision, Abteilung Militärische Forschung, ausgeliefert. Trotzdem: ein vom amerikanischen Energieministerium eingesetztes Gremium kam zu dem Ergebnis, daß die Zukunft solcher millionenschwerer Supercomputer fraglich ist. Es wird angezweifelt, daß ein kommerzieller Markt dafür entstehen kann, ohne daß massive finanzielle Unterstützung durch die Regierung gewährt wird. Die Zukunft könnte eher bei Standard-Workstations liegen, die in lokalen Netzen verbunden sind (Vgl. Martin B. Kalinowski, Bombengeschäft, in c´t Februar 1996, S.70) Auch diese Ausführungen belegen, daß man nicht unbedingt auf der Höhe der Zeit ist, wenn man die Betrachtung paralleler Architekturen auf die massiv parallelen Supercomputer beschränkt..

ben der Menschheit in einer immer stärker beschädigten Umwelt zu sichern – angehen zu können..

Das Kapitel *aktuelle Pressemeldungen* liefert Material zur Veranschaulichung, welche Firmen Supercomputer entwickeln – siehe hierzu auch den Auszug aus den TOP 500 im nächsten Kapitel – in welchen Bereichen und von welchen Auftraggebern Supercomputer eingesetzt werden.

Das folgende Kapitel *Strukturen von Parallelrechner-Architekturen* beschreibt wesentliche Eigenschaften der Parallerechner vom Standpunkt der Hardware. Dabei wird herausgestellt, wie die auf einer Ebene (siehe Ebenen der Parallelarbeit) realisierte Parallelität mit den jeweiligen Rechnerarchitekturen korreliert.

Im Kapitel *Konzepte der Parallelarbeit* werden die theoretischen Grundlagen der Parallelarbeit behandelt. Wichtiges Lernziel ist es zu verstehen, daß Parallelarbeit, also die gleichzeitige Ausführung logisch voneinander trennbarer Operationen innerhalb eines Computersystems, auf unterschiedlichen Ebenen realisiert werden kann. So gibt es Parallelarbeit auf *Anweisungsebene* und Parallelarbeit auf *Prozeßebene*. Es wird ausgeführt, mit welchen Mitteln Anweisungen parallelisiert werden können (explizite Anweisungen des Programmierers, selbständige Erkennung des Compilers), welche Schwierigkeiten beim Erkennen von Parallelität auf Anweisungsebene auftreten (Datenabhängigkeitsanalyse), welche Bedeutung den Compilern in diesem Zusammenhang zukommt (Optimierung für superskalare Befehlsausführung), wie Parallelarbeit auf Prozeßebene organisiert werden kann und wie die verschiedenen Prozesse wiederum synchronisiert werden können.

Im Kapitel *Superskalare Prozessoren und Very Long Instruction Word (VLIW)-Maschinen* werden zwei aktuelle Konzepte der Parallelarbeit auf Anweisungsebene vorgestellt und diskutiert. Beiden gemeinsam ist der Einsatz mehrerer Funktionseinheiten wie zum Beispiel Integer-Unit, Floatingpoint-Unit, Branch-Unit, Load/Store-Unit u.ä. Obwohl die VLIW-Maschinen, wie der Name schon sagt, mehrere Instruktionen auf einmal in den Prozessor laden und verarbeiten können, wird gezeigt, weshalb die Performance von superskalare Prozessoren zur Zeit die der VLIW-Maschinen übertrifft und weshalb beide Architekturen mehr als alle anderen so gut oder so schlecht sind wie ihre Compiler!

Auf dem Gebiet der Superrechner dominierten bis Mitte der neunziger Jahre die *Vektorrechner*. Diese Architektur setzt auf der Mikroarchitekturebene - neben mehreren Funktionseinheiten mit arithmetischen Pipelines - spezielle Vektorregisterbänke ein. Damit läßt sich die Verarbeitung von Aufgabentypen mit feldartigen Datenstrukturen beschleunigen. Um die parallelen Eigenschaften von Vektorrechnern durch Software nutzen zu können, wird die Befehlssatzarchitektur durch spezielle Vektorbefehle ergänzt. Dies wird eingehend im Kapitel *Vektormaschinen* erläutert. Dabei wird auch der Unterschied zu den *RE-Arrays* herausgearbeitet, die auf der Software-Seite wie Vektormaschinen erscheinen, sich aber auf der Mikroarchitekturebene durch den Einsatz einer Vielzahl von ALUs unterscheiden und auch daraus ihre besondere Rechenleistung beziehen.

Die Verwendung von Variablen in prozeduralen, höheren Programmiersprachen ist den Studenten derart vertraut und selbstverständlich, daß eine Kritik der Variable als Container für Operanden schon beinahe als Verletzung eines Naturgesetzes erscheint. Im ersten Kapitel wurde die Variable bereits dadurch kritisiert, daß beim Versuch der Parallelisierung eines Algorithmus Datenkonflikte erkannt und sehr aufwendig vermieden werden müssen. Einen ganz anderen Weg beschreiten hier die *Datenflußarchitekturen*, die mit der Zündregel-Semantik in Verbindung mit dem Prinzip der einmaligen Zuweisung in der Lage sind, jede Möglichkeit der Parallelarbeit selbst zu erkennen und mit dem Ziel der bestmöglichen Ausnutzung der Betriebsmittel auszuführen. Obwohl Datenflußarchitekturen kaum über das Stadium von Prototypen hinaus gelangten, sollen sie wegen des interessanten parallelen Ansatzes

und der konsequentesten Kritik der von Neumann Variable in einem gesonderten Kapitel gewürdigt werden.

Das letzte Kapitel gilt den *massiv parallelen Systemen*. Wo die hohe Leistung der heutigen superskalaren Prozessoren immer noch nicht ausreicht, geht man zu Multiprozessor-Architekturen über. Auf dem Gebiet der Supercomputer hat der bislang dominierende Vektorrechner in dem massiv-parallelen Rechner mit Tausenden oder Zehntausenden von Prozessoren eine starke Konkurrenz erhalten. Die zukünftigen Supercomputer im Teraflops-Leistungsbereich können nur als massiv-parallele Rechner realisiert werden. Die unterschiedlichen Ansätze für massiv-parallele Systeme (speichergekoppelte Systeme und Systeme mit verteiltem Speicher) werden in diesem Kapitel vorgestellt und die jeweiligen Vor- und Nachteile bezüglich Programmierung, Skalierbarkeit, Synchronisation usw. herausgearbeitet.

Da die Systeme mit verteiltem Speicher den großen Vorteil der freien Skalierbarkeit besitzen, gehört die Zukunft wohl dieser Architekturform. Systeme mit mehreren tausend Prozessoren lassen sich nicht mit gemeinsamem Speicher realisieren. Deshalb und aus Zeitgründen werden die Systeme mit gemeinsamem Speicher nur sehr kurz besprochen.

Die Programmierung von Parallelrechnern war lange Zeit nicht nur sehr aufwendig, sondern auch extrem hardwareabhängig und deshalb nicht portierbar. Sie mußte häufig bei Rechnerwechsel wieder neu erstellt werden, so daß die Entwicklungskosten oft in ungünstigem Verhältnis zur Verwendungszeit des Parallelrechners standen. In Anbetracht der Tatsache, daß die Rechenleistung von Standard-Mikroprozessoren sich jedes Jahr verdoppelt (Moore's Law), war die Alternative - sequentielle portierbare Programmierung in einer Standard-Programmiersprache für eine Standard-Prozessorfamilie (Intel x86, Alpha 21x64, PowerPC etc.) - oft die langfristig günstigere Lösung. Es gibt verschiedene Ansätze zur Lösung dieses Problems: Davon werden zwei exemplarisch vorgestellt:

- ***Threads*** zur Programmierung von parallelen Systemen mit gemeinsamem Speicher. Die standardisierten POSIX-Threads eignen sich unter verschiedenen Betriebssystemen sowohl für Systeme mit gemeinsamem Speicher (symetrische Multiprozessoren, SMP) als auch für Systeme mit gemeinsamem verteiltem Speicher (distributed shared memory multiprozessoren , DSM). Aktuelle Supercomputer wie der Alphaserver SC von Compaq verwenden Alpha 21364 Prozessoren unter True64 Unix und POSIX-Threads. Compaq setzt beim Alphaserver SC auf die DSM Architektur, weil dass die Systeme so frei skalierbar wie nachrichtenorientierte Systeme und so einfach programmierbar wie Systeme mit gemeinsamem Speicher sind
- und ***MPI*** (message passing interface, 1994), zur Programmierung von parallelen Systemen mit verteiltem Speicher. MPI ist standardisierte Bibliothek, die Funktionen zur Inter-Prozessommunikation und Verteilung der Prozesse auf die Prozessoren beinhaltet. Ziel von MPI ist es, für Programme und Bibliotheken Portabilität zu gewährleisten. MPI soll langfristig Kontinuität der parallelen Programmierschnittstelle gewährleisten, damit Programmentwickler nicht in wenigen Jahren auf Grund eines neuen Standards oder einer neuen Rechnergeneration noch einmal von vorne anfangen müssen.

Mit dieser Vorlesung wird das Angebot im Bereich Rechnerarchitektur vervollständigt. Die Studenten kennen bereits das Operationsprinzip traditioneller sequentieller Rechner aus der Vorlesung Rechnergrundlagen. In den Vorlesungen *Designprinzipien moderner Prozessoren* und *Paralleldatenverarbeitung – parallele Rechnerarchitekturen und parallele Programmierung* werden innovative Architekturen behandelt, wobei bei ersterer der Schwerpunkt auf optimales Phasenpipelining *(RISC-Prozessoren*) gelegt wird. Im vorliegenden Thema steht das Funktionspipelining, die Nebenläufigkeit im Mittelpunkt. Es wird hier der Bogen gespannt von Superskalaren Prozessoren über Vektormaschinen, Datenfluß-Architekturen bis zu

massiv parallelen Systemen mit gemeinsamen oder verteiltem Speicher. Dabei sollen die unterschiedlichen Operationsprinzipien herausgearbeitet werden.[2]

Die Vorlesung wird als ein allgemeines Wahlpflichtfach angeboten, bietet sich aber auch als Ergänzung zum Fach Prozeßdatenverarbeitung an, insbesondere zur Vorlesung *Programmierung paralleler Prozesse*..

[2] „Das sehr vielfältig gewordenen Gebiet der Parallelrechner-Architekturen hat gerade in jüngster Zeit eine gewisse Abklärung erfahren. Dies setzt den Autor eines Buchs über Rechnerarchitektur, das heutzutage natürlich vorwiegend ein Buch über Parallelrechner-Architekturen sein muß, in die glückliche Lage, recht klar die Unterschiede zwischen den einzelnen Operationsprinzipien herausstellen zu können." [Gil93, S.XII]

2 Aktuelle Pressemeldungen

Kasparow vs. Deep Blue: Mensch unterliegt Maschine

Schachweltmeister Garry Kasparow hat das Duell gegen IBMs Parallelrechner Deep Blue (c't 11/96, S. 334) mit 2,5 gegen 3,5 verloren. Zum ersten Mal triumphierte Silizium über einen Großmeister dieser Spielstärke. Noch vor einem Jahr hatte Kasparow mit 4:2 gesiegt und mutig behauptet, vor der Jahrtausendwende werde ihn kein Computer schlagen.

Auf die Revanche, die gerade in New York zu Ende ging, bereiteten sich die IBM-Entwickler besser vor denn je: Deep Blue bekam weitere Parallel-Prozessoren implantiert und wurde von 67 auf 130 MHz hochgetaktet - bis zu 200 Millionen Positionen pro Sekunde kann das System auf Grundlage einer IBM RS/6000 SP nun durchrechnen. Der US-Großmeister und Kasparow-Kenner Joel Benjamin trainierte über Monate mit der Maschine; die Programmierer stockten die Spieldatenbank um alle wichtigen Partien auf, die jemals gespielt wurden. Kasparow schimpfte auf der anschließenden Pressekonferenz: "Mit Wissenschaft hat das nichts zu tun, diese Maschine wurde ausschließlich mit der Motivation weiterentwickelt, mich zu schlagen." Doch auch der Weltmeister kommt längst nicht mehr ohne Computer aus; er bereitet sich - nicht nur auf Partien gegen Deep Blue - mit dem Hamburger Schachprogramm "Fritz" vor.

Details des imagefördernden Schaukampfes veröffentlicht IBM unter http://www.chess.ibm.com/. Der Server war bisweilen hoffnungslos überlastet; IBM will alleine am Dienstag während des Matches 22 Millionen Hits gezählt haben. Viele schachbegeisterte Anwender blieben außen vor; andere beschwerten sich über das anfangs mangelhaft programmierte Java-Applet, das die Züge der einzelnen Partien grafisch nachvollzieht.

URL dieses Artikels:

http://www.heise.de/newsticker/data/se-12.05.97-000/

Potsdamer Klima-Institut erhält IBM-Supercomputer

Teraflop-System im Oktober / Erweiterung durch Power4-Module 2002

21. Juni 2000 .

Das Potsdam Institute for Climate Impact Research (PIK) schafft einen RS/6000 SP-Supercomputer von IBM (Börse Frankfurt: IBM) an, der mit 200 Prozessoren bestückt ist. Das System solle ab Oktober 2000 installiert und für Klimaberechnungen eingesetzt werden.

Der neue Superrechner löst das seit 1994 bestehende IBM-System ab. Im Jahr 2002 soll er durch Power4-Multichip-Module von IBM aufgerüstet werden. Aktuell kommen 64-Bit, 375 MHz POWER3-II-Prozessoren auf Kupferbasis zum Einsatz.

Need for Speed: Unis kriegen Supercomputer

Compaq setzt Marke in IBMs Revier

22. August 2000 (sri)

Compaq (Börse Frankfurt: CPQ) hat einen zivilen Supercomputer an die National Science Foundation in Arlington, Virginia, verkauft. Im November sollen die ersten Systeme geliefert werden. Der Supercomputer basiert auf Compaqs Alphaserver SC-Architektur und besteht aus 2728 Alphaprozessoren mit 2728 GByte RAM und 50 TByte Massenspeicher. Die Spitzen-

leistung der Maschine liegt bei sechs Teraflops (Fließkommaoperation pro Sekunde). Betriebssystem ist Tru64 Unix von Compaq. Die Portierung, das Tuning und die Entwicklung von Parallelanwendungen wird das Pittsburgh Supercomputing Center übernehmen, den Rest erledigt Compaq.

Der Rechner wird bei der Proteinforschung, der Wirbelsturm- und Erdbebensimulation sowie der Berechnung der globalen Klimaveränderung eingesetzt. Erklärtes Ziel von Compaq ist es, im Bereich Supercomputer die Marktführerschaft zu erlangen.

Auch IBM (Börse Frankfurt: IBM) hält da mit seiner Maschine nicht zurück. Big Blue hat erneut eine zivile Version des militärischen Supercomputers ASCI White verkauft, diesmal an die Universität von Boston. Mit dem neuen RS/6000 SP soll die Quantenphysik, die Genom- sowie die Alzheimer-Forschung vorangetrieben werden.

ASCI White schafft in der militärischen Version 12,3 Billionen Rechenschritte pro Sekunde, ist damit mehr als dreimal so schnell wie der bisher schnellste Computer und bedeckt die Fläche von zwei Basketballfeldern. Das neue RS/6000 SP-System benutzt Power3-II-Kupfer-CPUs, Silizium-Switch-Technologie und eine neueartige Software. 2002 soll das System mit Power4-Prozessoren von IBM nachgerüstet werden.

Alpha-Chip: Konzentration auf Linux-Supercomputer

Compaq, Delta und Partec bieten Cluster für den technisch-wissenschaftlichen Bereich

25. Mai 2000 (dmu)

Compaq (Börse Frankfurt: CPQ) will den von DEC übernommenen Alpha-Chip künftig verstärkt im Bereich der Supercomputer einsetzen. Zusammen mit dem Karlsruher Universitäts Spin-off Partec und dem Vertiebspartner Delta werde man Cluster-Lösungen von acht bis hundert Alpha-Prozessoren für HPTC (High Performance Technical Computing) anbieten.

Die Hardwaregrundlage für den Alpha-Verbund liefert Myrinet, als Betriebssystem kommt Linux zum Einsatz. "Egal wo sie hingehen, jede Firma zeigt sich von Linux begeistert, erklärte der Partec-Manager Bernhard Frohwitter. "Bislang können wir schon mehrere hundert Aufträge vorweisen", berichtete der Compaq-Manager Harald Meier-Fritsch. Bei einem Preis von 250.000 Dollar für eine einzige auch Beowulf genannte Lösung kein Pappenstil.

An der Bergischen Universität in Wuppertal etwa existiert bereits ein Cluster mit 64 Alpha-Chips, die mittels einer sogenannten Fat Tree Topologie von Myrinet vernetzt wurden. Das Netzwerk besitzt eine Peakrate von 1,28 GBit/s. Dem Benutzer steht nun eine Netto-Datentransferrate von 150 MByte/s zur Verfügung. In den kommenden Wochen soll die Leistung verdoppelt werden - was das stärkste Linux-Cluster in Deutschland ergeben würde.

Digitals Alpha-Chip ist ein schneller RISC-Prozessor (Reduced Instruction Set Computer). Er arbeitet mit einem kleinen Befehlssatz und rechnet deshalb schneller als Prozessoren mit CISC-Architektur (Complex Instruction Set Computer). Aktuelles und Grundlegendes zu Prozessoren und Mainboards von AMD über Intel bis Cyrix bietet ein ZDNet-Spezial.

Kontakt:
Compaq Infoline, Tel.: 01803/3221221

Supercomputer: Dauerspitzenreiter ASCI Red verliert Führung

In der neuen, 16ten Top500-Liste[1] der leistungsfähigsten Supercomputer musste Intels ASCI Red erstmals seit nunmehr dreieinhalb Jahren den Spitzenplatz[2] räumen. Die mit rund 10.000 Pentium-Pro-Overdrives (333MHz Pentium II Deschutes) bestückte Maschine wurde

in der Linpack-Leistung nun von IBMs ASCI White SP Power3 (8192 Power3 Prozessoren mit 375 MHz) deutlich übertroffen, die in diesem Benchmark mit knapp 5 Tera-Flops mehr als die doppelte Leistung erzielen kann. Dahinter folgen ebenfalls zwei Maschinen aus dem vom amerikanischen Staat mitfinanzierten Supercomputer-Programm ASCI (American Strategic Computing Initiative) von SGI und Intel. IBM stellt damit nicht nur erstmals den Spitzenreiter, sondern konnte gleich neun Systeme unter den ersten 20 platzieren. Insgesamt kommen nun 43 Prozent aller in der Top500-Liste verzeichneten Systeme von IBM: ein rasanter Anstieg von 28 Prozent gegenüber der Liste vom Sommer 2000. Dahinter folgt Sun mit 18 Prozent und SGI mit 13 Prozent. Insgesamt stieg die Rechenleistung aller Top500-Systeme gegenüber der Juni-Liste um fast 40 Prozent von 64,3 TFlops auf 88,1 TFlops, was mehr als "Mooreschem Tempo" entspricht. (Das nach Intel-Gründer Gordon Moore benannte Gesetz geht von einer Verdoppelung der Chip-Komplexität alle 18 Monate aus und wird auch gerne auf die Steigerung der Performance übertragen.) Auffällig ist, dass sich die Clustersysteme herkömmlicher Server immer stärker in Szene setzten. Ihr Anteil stieg von 11 Systemen im Sommer auf nunmehr 28. Cluster aus SMP-Maschinen (Constellations) sind ebenfalls auf dem Vormarsch, während die Anzahl reiner SMP-Computer von 120 auf 17 sank. Interessant ist auch die geografische Verteilung der Einsatzorte (mit leicht wachsender Tendenz für Europa): 241 Systeme stehen in Nordamerika, 177 Systeme in Europa, 63 Systeme in Japan und 19 Systeme in der restlichen Welt. (Christian Birkle) / (as[3]/c't) URL dieses Artikels: http://www.heise.de/newsticker/data/as-03.11.00-000/ Links in diesem Artikel: [1] http://www.top500.org/list/2000/11/ [2] http://www.heise.de/newsticker/data/as-08.06.00-000/ [3] mailto:as@ct.heise.de

IBMs neuer Supercomputer bricht alle Rekorde

Den Titel des schnellsten Rechners der Welt trägt seit heute der IBM RS/6000 ASCI White. Der von IBM im Auftrag der amerikanischen Regierung entwickelte Supercomputer vereint insgesamt 8192 Prozessoren in 512

Rechnereinheiten mit jeweils 16 IBM Power3-III CPUs über Hochleistungsswitches zu einem System, das 12,3 Teraflops leisten soll - zwölf Mal mehr, als der gerade erst im Münchener Leibniz-Institut eingeweihte[1] Hitachi SR8000 F1, der als Europas schnellster Rechner gilt. Der gigantischen Rechenleistung angemessen gehören 6,2 Terabytes Arbeitsspeicher und 160 Terabytes Festplattenkapazität zur Peripherie des 110 Millionen US-Dollar teuren Systems. Der ASCI White ist damit bereits 1000 Mal leistungsfähiger als IBMs Deep Blue, der 1997 Schachgenie Kasparov schlagen[2] konnte.

Für die Endmontage des Rechners dürfte man im Lawrence Livermore National Laboratory[3] (LLNL) in Livermore, Kalifornien, einige bauliche Erweiterungen vorgenommen haben. Schließlich benötigt die Anlage die Fläche von zwei Basketballfeldern, wiegt mit 106 Tonnen ungefähr so viel wie 17 ausgewachsene Elefanten und beansprucht 1,2 Megawatt elektrischer Leistung – so viel wie rund 1000 Haushalte. Über 3000 Kilometer Kabel sind in ASCI White verdrahtet und gigantische Kühlanlagen erforderlich, um diese Leistungsaufnahme zu kontrollieren.

Seit 1999 gilt IBM als Führer auf dem Markt der Supercomputer, auf dem jährlich lediglich rund 250 Rechner mit Preisen zwischen 2 und 100 Millionen US-Dollar verkauft werden. Die

Entwicklung von ASCI White dürfte deshalb auch Fortschritte bei kleineren Systemen bewirken, die typischerweise in der Klimaforschung, der Datenverschlüsselung oder beim Design komplexer technischer Systeme eingesetzt werden. Der Rekordrechner wurde im Rahmen des US-ASCI-Programms zum Aufbau von Supercomputern aus vielen vergleichsweise gewöhnlichen Einheiten entworfen, in das auch andere Firmen wie SGI und Intel involviert sind.

Das US-Department of Energy wird die geballte Rechenleistung des ASCI White nutzen, um wesentliche Faktoren der Wirkung von Atomwaffen, wie beispielsweise Alter und Design des Sprengkopfs, zu simulieren. Nach Ansicht von David Cooper, Chief Information Officer des LLNL, reicht die Rechenleistung des Systems aber noch nicht aus, um vollständig auf Nuklearwaffentests verzichten zu können. Dazu wären 100 Teraflops erforderlich, die nicht vor 2004 zu erwarten seien. Demnach wird es leider noch einige Zeit dauern, bis auch die USA den Vertrag über den Stopp von Atomwaffentests unterzeichnen und endlich auf unterirdische Explosionen verzichten. (law[4]/c't)

URL dieses Artikels:

http://www.heise.de/newsticker/data/law-29.06.00-000/

Viel hilft viel: Die neuen Supercomputer haben Billigprozessoren wie der PC nebenan - aber zu Tausenden

von Wolfgang Blum

Eine magische Grenze ist gefallen: Aus 7264 Prozessoren des Typs Pentium Pro montierte vor ein paar Wochen der amerikanische Prozessorhersteller Intel den schnellsten Computer der Welt zusammen. Die Maschine schafft mehr als eine Billion Rechenoperationen pro Sekunde (im Fachjargon: ein Teraflops). Gebaut wurde sie im Auftrag des Washingtoner Energieministeriums; mit ihr sollen Atomwaffenversuche simuliert werden.

Auf der Liste der 500 schnellsten Rechner, die Mitarbeiter der Universitäten Mannheim und Tennessee zweimal jährlich erstellen, taucht der Gigant von Intel zwar noch nicht auf. Doch auch hier dominieren die sogenannten Parallelrechner, bei denen sich mehrere Rechenwerke die Arbeit teilen. Unter den ersten zehn finden sich sechs Maschinen mit jeweils mehr als tausend Prozessoren.

Vor wenigen Jahren waren sie noch Exoten. Da gab es fast nur Computer, in denen ein einzelner Prozessor seine Arbeit hübsch der Reihe nach erledigte, einen Rechenschritt nach dem anderen. Ein zeitraubendes Verfahren, das im richtigen Leben absurd anmuten würde. Man stelle sich vor, Michael Schumacher brauste zum Reifenwechsel an die Boxen und nur ein einziger Mechaniker dürfte sich ans Werk machen.

Die schnellsten Computer der Welt benötigten bei dieser Bauweise eben auch die schnellsten Prozessoren. Das waren ungemein teure Spezialkonstruktionen, die aufwendig mit Kühlmitteln vor dem Verglühen bewahrt werden mußten. Heute sind die meisten Supercomputer Parallelrechner, und ihre Spitzenleistungen beruhen auf der Zusammenarbeit vieler billiger Mikroprozessoren, wie man sie auch in einem PC oder in einer Workstation findet.

Der Trend gehe zu Maschinen mit solchen Standardprozessoren, die sich nach dem Baukastensystem erweitern ließen, sagt Ulrich Trottenberg vom GMD-Forschungszentrum Informationstechnik in St. Augustin bei Bonn. Der Käufer könne mit wenigen Recheneinheiten anfangen und später weitere nachrüsten. "Alle Firmen, die eigene Prozessoren für Parallelrechner entwickelt haben, sind daran zugrunde gegangen."

Der schnellste in Europa installierte Computer dieser Gattung nimmt Platz zehn der Rangliste ein. Er stammt von dem japanischen Hersteller Fujitsu und steht seit September vergangenen Jahres im Europäischen Zentrum für mittelfristige Wettervorhersage im englischen Reading. Auch der **Deutsche Wetterdienst** in Offenbach will noch in diesem Jahr auf solch einen Rechner umsteigen.

In der **Meteorologie** ist der Bedarf an Rechenleistung besonders hoch, ja er läßt sich fast beliebig ausdehnen. Für eine Prognose legen die Wetterforscher ein Gitter in die Lufthülle um

unseren Planeten und berechnen dann für jeden Gitterpunkt, wie sich mit der Zeit Temperatur, Wind, Druck und Feuchte ändern. Je enger das Netz geknüpft ist, desto zuverlässiger sind die Ergebnisse. Für eine Maschenbreite von dreißig Kilometern hätte der alte Computer des Zentrums in Reading - eine Cray C916 - dreißig Stunden gebraucht, zu lang für aktuelle Vorhersagen. Der neue Parallelrechner mit seinen 46 Prozessoren dagegen gibt das Resultat bereits nach sechs Stunden aus.

In der **Industrie** konnten sich, anders als in der Wissenschaft, Parallelrechner bislang kaum durchsetzen. Die Unternehmen scheuen davor zurück, ihre Software, die sie oft in jahrelanger Arbeit entwickelt haben, umarbeiten zu lassen. Da es kaum passende Software gibt, investiert umgekehrt auch niemand gerne in neue Computer.

Allerdings zeichnet sich ein einfacher Ausweg aus dieser Sackgasse ab: Neue Technik ermöglicht es, die Rechner, die man schon hat, seien es PCs oder Workstations, über das firmeneigene Netz zusammenzuschalten. So entsteht gewissermaßen ein **verteilter Parallelrechner**, der zu ungeahnten Leistungen imstande ist.

Bislang liegt in den Büros ein Großteil der Rechenkapazität brach - nachts, wenn die Computer ausgeschaltet sind, aber auch tagsüber. Die meiste Zeit warten die Rechner schlicht auf die nächste Eingabe. Sie besser zu nutzen kostet die Unternehmen nur wenig. Man bräuchte nur Software, die die Arbeit auf die einzelnen Rechner verteilt und am Ende die Ergebnisse einsammelt.

Die Entwickler, die daran arbeiten, haben allerdings heikle Probleme zu meistern. In den meisten Unternehmen haben sich im Lauf der Zeit ganz verschiedenartige Computer angesammelt. Sie zur Zusammenarbeit zu bewegen ist nicht leicht. Hinzu kommt, daß die beteiligten Recheneinheiten schon während der Arbeit ständig in Kontakt bleiben müssen, um Zwischenresultate auszutauschen. Heutige Datennetze sind da schnell überfordert, selbst wenn höhere Programmierkunst das Hin und Her minimiert. Weil **Staus im Netz** den Zeitgewinn durch die parallele Verarbeitung mindern, lohnt sich der Zusammenschluß von mehr als zehn Computern bislang nur selten.

Bei **normalen Parallelrechnern** sitzen hingegen alle Prozessoren **im selben Gehäuse** und sind über **Hochleistungskanäle** zusammengeschaltet. Zu Engpässen kommt es da nicht so schnell. Nur mangelt es noch an Programmen, wie sie die Unternehmen brauchen.

In einem von der Europäischen Union finanziell unterstützten **Projekt namens "Europort**" wurden aber bereits 38 industrielle Programme an Parallelrechner angepaßt ("portiert"). Der Großteil davon sind **Simulationen**, etwa von **Feuerausbrüchen**, der **Aerodynamik** von Flugzeugen, dem **Schmieden von Maschinenteilen**, **dem Design von Turbinen und Polymeren, dem Luftzug in und um Autos, der Wirkung neuer Arzneimittel oder Crash-Versuchen für Autos.**

Ein Fahrzeug aus Testzwecken gegen die Wand zu setzen kostet mehrere hunderttausend Mark. Soweit wie möglich wollen die Automobilunternehmen daher ihre Versuche in den Computer verlegen. Das erfordert aber viel Rechenleistung. "**Crash-Simulationen sind das Sesam-öffne-dich-Wort für Parallelrechner in der Industrie**", sagt Wolfgang Gentzsch, Professor an der Fachhochschule Regensburg und Chef von Genias, einem der beiden Softwarehäuser in Europa (neben Pallas in Brühl), die sich auf parallele Systeme spezialsiert haben.

Die Firma Pars in Altenau stellt **Lenkräder für Automobilfirmen** her. Um neue Modelle in Crash-Simulationen zu testen, rechnet eine Workstation mehrere Wochen. Weil in dieser Branche die Zeit immer drängt, muß Pars die Gußformen bestellen, bevor die Resultate vorliegen. Sind Änderungen nötig, müssen die Formen kostspielig umgebaut werden. Nun will sich das Unternehmen zwar nicht gleich einen teuren Parallelrechner zulegen. Aber künftig

sollen dank Europort gleichzeitig alle verfügbaren Workstations an einer Simulation arbeiten und sie so um ein Vielfaches schneller machen. Jeder Prozessor ist dabei für ein Teilstück des Lenkrads zuständig. Einsparungen von drei Millionen Mark jährlich verspricht sich Pars dadurch.

Ein anderes Beispiel: Die britische Firma Animo produziert **Cartoons** für die Filmindustrie, unter anderem für Warner und Spielberg. Europort parallelisierte die Programme für die Nachbearbeitung der Bilder im Computer. Hunderte von PCs können nun gleichzeitig an einer Szene feilen.

Auf den Plätzen 230 und 231 der schnellsten 500 sind die ersten Computer zu finden, die nicht von einem japanischen oder amerikanischen Hersteller stammen. Die beiden GC PowerPlus der Aachener Firma Parsytec mit je 192 Prozessoren laufen an den Universitäten Heidelberg und Paderborn.

Parsytec, weltweit einer der ersten Hersteller, die ausschließlich auf paralleles Rechnen setzten, hat sich inzwischen auf **Mustererkennung** spezialisiert. So kontrolliert zum Beispiel ein Computer aus Aachen die Qualität der Bleche des zweitgrößten Stahlwerks der Welt, das die Pohang Iron & Steel Company in Südkorea betreibt. 900 Meter Blech sausen pro Minute an 80 Kameras vorbei. Auf deren Bildern erkennt der Rechner noch Defekte, die kleiner als ein Millimeter sind. Andere Parsytec-Computer entziffern Handschriften oder spüren in Durchleuchtungsgeräten für Fluggepäck Sprengstoff auf.

Der Markt für die neue Computertechnik rührt aber nur zu etwa einem Drittel vom klassischen "Number-Crunching", der Zahlenfresserei, her. Den Rest macht die Datenverwaltung aus. Wollen Unternehmen große Datenbestände durchforsten, teilen sich meist viele Prozessoren die Mühe. Beim führenden US-Paketdienst Federal Express gewährleisten Parallelrechner, daß jeder Kunde via Internet abrufen kann, wo sich das Paket, das er erwartet, gerade befindet. Liegt es noch in New York oder bereits beim Frankfurter Zollamt?

Jörg Thielges, Spezialist für Großrechner bei IBM , berichtet von einer amerikanischen Kaufhauskette, die innerhalb weniger Minuten herauskriegen wollte, welcher Badeanzug sich am besten verkaufte. Paralleler Datenverarbeitung zufolge ging ein Modell, das ausrangiert werden sollte, in Kalifornien gar nicht. In Texas hingegen war es der Renner. Also ließ das Management die restlichen Bestände nach Austin, Houston und Dallas bringen.

Viele Firmen, zum Beispiel die Lufthansa , haben sich Parallelrechner zugelegt, um das Verhalten ihrer Kunden zu analysieren. Kundenkarteien auszuwerten halten Datenschützer zwar zuweilen für bedenklich, Werbeagenten aber für unentbehrlich.

Auch als Internet-Server eignen sich Parallelrechner. So lassen sich Staus vermeiden und Ausfälle verhindern. Sollte eine Recheneinheit einmal nicht funktionieren, übernehmen die anderen ihre Aufgaben. Selbst PCs bleiben von Parallelität wohl nicht verschont. Wolfgang Gentzsch von Genias rechnet damit, daß demnächst PCs mit vier oder acht Prozessoren auf den Markt kommen. Andere Experten wie Thielges von IBM glauben hingegen, daß künftig nur noch Terminals auf den Schreibtischen stehen, die sich alles Nötige aus dem Netz holen. Auch dann werden aber meist Parallelcomputer die Daten bereitstellen.

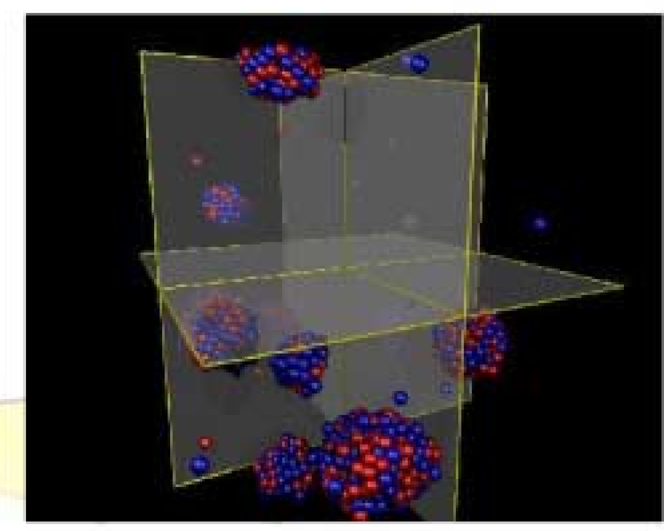
Visionäre träumen bereits von der **Ära des Meta-Computing**: Alle Rechner sind über ein Netz zusammengeschlossen. Einzelne Demonstrationsprojekte haben schon gezeigt, wie etwa Maschinen in den USA und Deutschland, verbunden über ein Breitbandnetz, gemeinsam Aufgaben lösten. Im Auftrag des deutschen Klimarechenzentrums in Hamburg sollen demnächst Computer der GMD in St. Augustin und des Forschungszentrums Jülich zusammengeschaltet werden: Die eine Großforschungseinrichtung will das Geschehen in der Atmosphäre simulieren, die andere das in den Ozeanen.

Bei allen Versprechungen des Netzes kommt in Jülich das Streben nach hauseigener Spitzenleistung nicht zu kurz. Das Forschungszentrum will in diesen Tagen 512 Prozessoren einer Cray T3E zusammenschalten und damit dann in der Liste der schnellsten 500 Rechner ganz vorne mitmischen.

(C) DIE ZEIT Nr.05 vom 24.01.1997

Molek**u**lardynamik

Simulation eines Edelgases (Argon) mit Partitionierung für 8 Prozessoren: 2048 Moleküle in einem Würfel von 16 nm Kantenlänge werden in Zeitschritten von $2x10^{-14}$ sec simuliert.



Simulation auf dem Vormarsch

Bei der Entwicklung und Fertigung industrieller Produkte wird heute praktisch jeder Schritt am Computer durchgespielt. Simulations-Software ergänzt dabei vielfach traditionelle CAD-Werkzeuge, ob bei Festigkeitsberechnungen oder der Konstruktion neuer Bauteile.

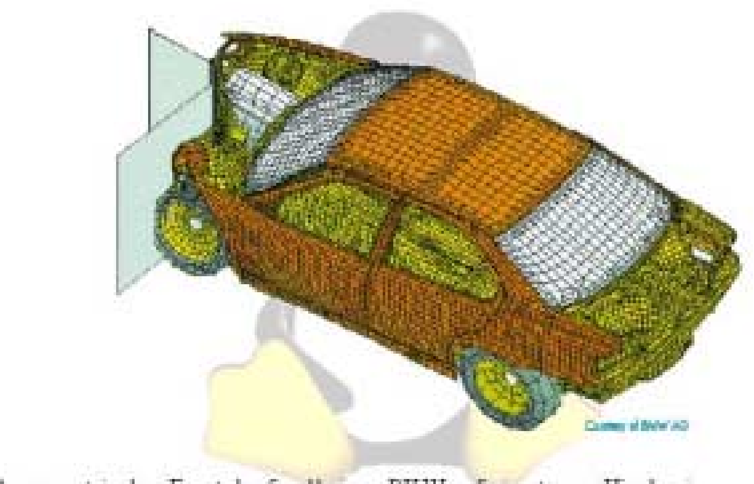
Die Einsatzmöglichkeiten von Simulationsverfahren sind vielfältig. Sie reichen von der Klimaforschung über die Logistik bis zur Chirurgie. In der Produktion spart Simulation Zeit, Geld und Material. "Wir können schneller, besser und effizienter entwickeln, weil wir im Entwicklungsprozess noch früher Simulationsverfahren einsetzen und Entwicklungsergebnisse dadurch früher überprüfen können", stellt Peter Zimmermann fest, Leiter Virtual Reality beim Automobilhersteller VW. Wenn von Designern über Crash-Experten und Motoren-Technikern bis zu Werkzeugherstellern und Zulieferern alle zu jeder Zeit auf dem aktuellsten Stand sind, lassen sich Wege abkürzen und Umwege vermeiden. "Schon vor dem ersten Prototypen wird am Rechner gecrasht, und wir wissen über das Dynamikverhalten der neuen Hinterachse Bescheid", erklärt Zimmermann.

Produkte vom Computer

Die Vision der Automobilhersteller ist das virtuelle Auto aus dem Rechner. Von der ersten Skizze, über die Konstruktion der Karosserie, den Einbau von Komponenten, Crashtests bis hin zur Montage des Wagens sollen alle Schritte im Computer erfolgen. Das spart Zeit und teure Prototypen. Mit einer gewissen Anzahl simulierter Crashs wird die Gefahr vermindert, dass ein realer Crashtest nicht die vorberechneten Ergebnisse bringt und mit einem weiteren, teuren Prototypen wiederholt werden muss.

Die Computerprogramme, mit denen alle Parameter für unterschiedliche Aufprallgeschwindigkeiten oder -winkel abgebildet werden können, basieren auf einem algorithmischen Modell, das unter dem **Namen Finite-Elemente-Methode (FEM)** bekannt ist.

Mit diesem Ver-fahren wird um die Oberfläche eines Bauteils ein Netz gelegt und jedes Teil-stück der Struktur als Gruppe von finiten Ele-men-ten beschrieben. Jedes einzelne Ele-ment ist ein Poly-gon, das mit einer mathematischen Be-schreibung der phy-sikalischen und werk-stofflichen Eigenschaf-ten verknüpft ist. Das Gesamtmodell für eine Crashsimulation besteht in der Regel aus mehreren Komponenten: der Fahrzeugkarosserie, den Sitzen, dem Motorblock und den Passagieren. Diese Komponenten werden anschließend weiter untergliedert, das Fahrzeug zum Beispiel in Türen, Verglasung und Säulen. Je mehr finite Elemente in dem Modell abgebildet werden, desto realitätsnaher ist die Simulation. Ein Fahrzeugmodell kann aus 200000 bis 300000 solcher Elemente bestehen. Durch die Abbildung der Sitze, des Motorblocks und der Passagiere können weitere 100000 bis 200000 Polygone dazukommen.Um die vielen Tausend finiten Elemente eines Modells zu erstellen, nutzt man die schon erarbeiteten CAD-Daten. Dann wird jedes Element mit seinen physikalischen Eigenschaften (Masse, spezifisches Gewicht, Steifigkeit) und den Beziehungen zu den Nachbarelementen verknüpft. Unter Berücksichtigung der jeweils aktuellen Position und der Spannungsverhältnisse jedes einzelnen Elements wird dann das Gleichungssystem immer wieder neu berechnet. Jede Iterationsschleife nutzt dabei als Ausgangspunkt das Endergebnis der vorausgegangenen Berechnung. FEM ist freilich nicht auf den Automobilbau beschränkt, sondern bei allen Festigkeitsberechnungen anwendbar. So ist FEM auch im Flugzeugbau ein sehr nützliches Hilfsmittel, etwa bei der Minimierung von Luftturbulenzen an der Außenhaut. Denn auch wenn ein Flugzeug ruhig gleitet, geht es außen stürmisch zu. Die Luft umströmt die Außenhaut an vielen stellen nicht glatt, sondern löst sich in eine Unzahl kleiner Wirbel auf, die den Kraftstoffverbrauch deutlich erhöhen. Diese wenige Zentimeter dünne, turbulente Luftschicht minimieren die Konstrukteure heute per Computersimulationen. Der komplette CAD-Datensatz der Maschine wird auch für Simulationsrechnungen zum Schwingungsverhalten und zur Festigkeit der Maschine unter verschiedenen Belastungen per FEM benutzt.


Asymmetrischer Frontalaufprall eines PKW auf ein starres Hindernis

Rapid Prototyping

Noch zu Beginn der 90er-Jahre bauten alle Automobilhersteller Motormodelle im Maßstab 1:1 aus Holz. Einerseits dienten sie als plastische Anschauungsmodelle, andererseits wurden sie herangezogen, um zu prüfen, wie gut sich die Einzelteile zusammensetzen ließen und wie gut man den Antriebsblock in den Motorraum ein- und ausbauen konnte. An einem solchen "Holzmotor" arbeiteten mehrere Schreiner gut acht Monate. Kostenpunkt: einige hunderttausend Euro. Mit Hilfe der CAD-Daten werden mit unterschiedlichen Techniken naturgetreue Modelle, Bauteile und auch Werkzeuge schichtweise aufgebaut. "Der mit den Rapid-Technologien verbundene Zeitgewinn ist unerlässlich, um die Entwicklungszyklen für neue Fahrzeuge zu verkürzen", erläutert Bernhard Wiedemann, Leiter des Fachgebiets Rapid Prototyping im DaimlerChrysler-Forschungszentrum in Ulm. Funktionale Prototypen ähneln dem späteren Serienteil in Form und Funktion sehr stark, bestehen aber nicht zwingend aus demOriginalwerkstoff. Technische Prototypen werden mit besonderen Werkzeugen hergestellt, die

mittels Rapid Tooling entstanden sind. Sie entsprechen dem späteren Serienbauteil weitgehend in Form, Funktion und Material.

Typische Beispiele sind Kunststoffspritzguss-Werkzeuge, mit denen etwa Pedalhebel hergestellt werden. Technische Prototypen bilden den Übergang zum Rapid Tooling oder Rapid Manufacturing und können sogar für Vor- und Kleinserien eingesetzt werden. Einen der größten Vorzüge im Rahmen des gesamten Produktentwicklungsprozesses erwartet Wiedemann in Zukunft bei der schnellen Herstellung von Vorserien-werkzeugen. Für die Entwicklung eines neuen Autos zum Beispiel können mit Rapid Prototyping rasch erste Anschauungsmodelle für Konstrukteure und Designer erzeugt werden. In der anschließenden Konstruktionsphase entstehen funktionale Prototypen für erste Tests. In der Festlegungsphase schließlich lassen sich die notwendigen Produktions-mittel entwickeln und erproben. Im Vergleich zum Holzmotor ist heute innerhalb von sieben Wochen eine Motorattrappe aus Epoxidharz und Polyamid fertig,die nur rund 60000 Euro kostet. Gegenüber der traditionellen Fertigung reduzieren sich dieKosten um 85 und der Zeitaufwand um 80 Prozent.

Auf dem Weg von der Idee zum Produkt werden sich in Zukunft Digitaltechniken wie Digital Mockup oder Virtual Reality und die Rapid-Technologien ergänzen. Mit Digital Mockup (DMU) ist der digitale Zusammenbau von komplexen Produkten wie Automobilen oder Flugzeugen am Rechner simulierbar, um mögliche Kollisionen von Detailkonstruktionen zu entdecken.

"DMU ist die digitale Integrationsplattform und benötigt eine adäquate visuelle Repräsentation der Produktdaten", meint Oliver Riedel, VR-Bereichsleiter beim Stuttgarter Systemhaus Cenit. Virtual Reality sei am besten zur Darstellung von DMU geeignet, weil diese Technologie sowohl das Eintauchen des Benutzers in eine virtuelle Umgebung als auch auf natürliche Weise in die Daten selbst erlaube und dabei die Teamarbeit unterstützt. In naher Zukunft würden CAD-Standardsoftware-Pakete VR-Technologien als integralen Bestandteil anbieten. Damit erfülle VR den eigentlichen Zweck, eine bessere Schnittstelle zwischen dem Menschen und den im Computer modellierten Daten zu bieten.

Datenhandschuh und Brille

Die deutsche Automobilindustrie ist bei der Anwendung von Virtual Reality im Automobilbau vorn dran. Virtuelle Montagetechniken erlauben eine Überprüfung der Konstruktion ohne Prototypen. "Das Sichtbarmachen von Produkten und Prozessen bei der Entwicklung eines neuen Modells - das ist der entscheidende Vorzug der virtuellen Realität", stellt Klaus Vöhringer, Forschungschef von Daimler Chrysler, fest. In der Cave, einem auf einer Seite offenen Acrylglas-Würfel mit einer Kantenlänge von je 2,5 Metern sind Projektionsflächen als quasi begehbares 3D-Kino aufgebaut. Der Monteur streift sich einen Datenhandschuh über und setzt sich eine "getrackte" Brille auf. Per Kabel ist dieStereo-Sichtbrille mit dem Grafikcomputer verbunden. Jede Kopfbewegung des Monteurs melden Lagesensoren, so genannte Tracker, dem Grafikrechner, und der berechnet zwölf Mal pro Sekunde für alle fünf Projektionsseiten, wie sich die Szenerie für den Monteur durch den neuen Blickwinkel verändert.Der Monteur soll prüfen, ob er an das Getriebe des neuen Modells herankommt oder ob eine darunter verlaufende Querstrebe eventuell fällige Servicearbeiten behindert. Aus den Konstruktionsdaten berechnet der Grafikcomputer ein räumliches Modell von der Getriebeumgebung und wirft es als Bild auf die Projektionsflächen. Der Monteur nimmt diese Szenerie so wahr, als befände er sich in einer virtuellen Werkstatt. Mit dem Datenhandschuh sucht er den Zugang zum Getriebe: Lassen sich alle Schrauben lösen und wieder festziehen? Ist genügend Platz, um mit dem Werkzeug vernünftig arbeiten zu können? Lässt sich das Getriebe an der Querstrebe vorbei aus- und wieder einbauen? Solche Fragen kann er virtuell klären. Eckt er etwa mit der Getriebeabdeckung irgendwo an, färbt sich der Kollisionsbereich rot. Die VR-Technologie bietet so nicht nur neuen kreativen Spielraum, auch Probleme sind auf diese

Weise frühzeitig erkennbar und damit vermeidbar**. Virtuell lassen sich mehr Produktvarianten in kürzerer Zeit testen, als wenn man dafür jedes Mal ein Modell in langwieriger Handarbeit fertigen müsste**. **"Insgesamt bietet VR die Möglichkeit, die Qualität unserer Produkte zu steigern, die Entwicklungszeiten zu verkürzen und Kosten zu sparen.** Allein im Modell- und Attrappenbau liegt das Einsparpotenzial bei ungefähr 20 Prozent", sagt Vöhringer. Während früher ein neues Modell in separat agierenden Abteilungen nacheinander entstand, wird es nun von kooperierenden und parallel arbeitenden Spezialisten aller beteiligten Disziplinen Schritt für Schritt zur Produktionsreife gebracht. Daimler Chrysler hat mit DB View eine eigene VR-Software entwickelt, setzt aber wie auch VW auf das Softwarepaket Covise der Vircinity IT-Consulting."Um im harten Wettbewerb bestehen zu können, ist die Industrie gefordert, neben der reinen Produktinnovation und der Abdeckung einer möglichst breiten Produktpalette, auch die Erforschung und Nutzung fortschrittlicher Methoden des Produktentstehungsprozesses voranzutreiben", sagt VW-Mitarbeiter Zimmermann. CAx, Rapid Prototyping, Digital-Mock-Up, Virtual Reality und seit kurzem Augmented Reality sind nur einige Schlagworte für Technologien, an deren sinnvollem Einsatz in durchgängigen Prozessketten Forscher und Anwender arbeiten. "**Während die flächendeckende Nutzung von CAD und CAE heute schon Realität ist, steht der Einsatz von DMU-Datenbanken und -Funktionen sowie der durchgängige Einsatz von VR- und erst recht AR-Methoden noch am Anfang**", räumt der VR-Spezialist ein. Viel versprechend und technisch anspruchsvoll sind virtuelle Montagen. "Sie werden jetzt vermehrt von den Entwicklungsbereichen gefordert, obwohl noch einige Komponenten fehlen", bekräftigt Zimmermann. In zwei großen öffentlich geförderten Projekten wie IVIP (Integrierte Virtuelle Produktentstehung) und ARVIKA (Augmented Reality in Entwicklung, Konstruktion und Produktion), steht gerade das Thema Montage im Vordergrund. Industrie und Forschungsinstitute haben sich hier zum weltweit größten Konsortium zusammengeschlossen. Sie wollen VR/AR-Techniken in marktreife Anwendungen umsetzen, insbesondere für Bau und Wartung von Automobilen, Flugzeugen, Maschinen und Anlagen. Die technische Koordination liegt beim Fraunhofer-Institut für Graphische Datenverarbeitung (IGD) in Darmstadt, die Leitung hat Siemens und gefördert wird das Projekt vom Bundesforschungsministerium.

Erweiterte Einsichten

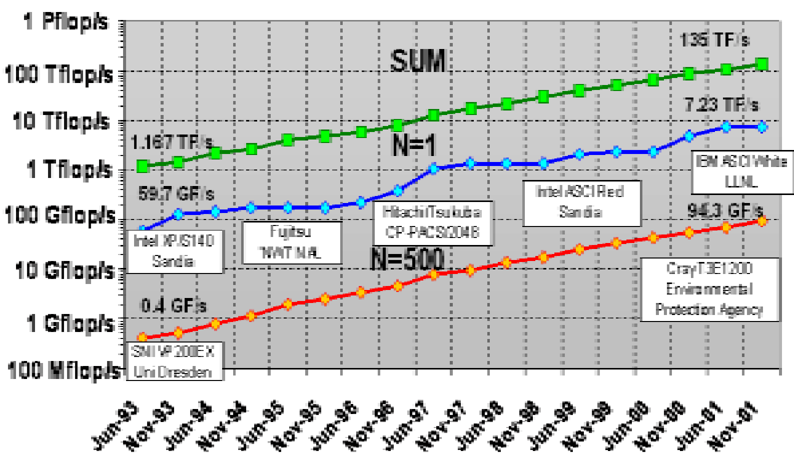
Realität und vom Computer generierte Bilder überlagern sich für den Betrachter bei der Augmented Reality (AR), indem über eine Datenbrille oder ein semitransparentes Display Informationen eingeblendet werden. "Ein am Gürtel befestigter Minirechner oder ein Laptop erzeugen lagerichtig zum Objekt die akustischen und visuellen Informationen, eine bleistiftdünne Videokamera an der Brille dient dazu, die Position und die Blickrichtung des Benutzers exakt zu ermitteln", erläutert Stefan Müller, VR-Abteilungsleiter am Fraunhofer IGD. Prädestiniert sei AR nicht nur für den Bau hoch komplexer Produkte wie Flugzeuge, Kraftwerke oder Maschinen. Mit mobilen AR-Komplettlösungen würden sich enorme Potenziale für neue Anwendungen eröffnen. Montage- und Servicetechniker erhalten digitale Handbücher, Mediziner können sich bei Diagnose und Operation Röntgenbilder oder Ultraschalldaten einblenden lassen und müssen dabei nicht den Blick von der Eingriffsstelle abwenden. Architekten können mit Hilfe von AR prüfen, ob geplante Brücken oder Gebäude sich in das landschaftliche Umfeld einfügen.

Hersteller könnten weltweit den Technikern vor Ort alle digitalen Daten für die Reparatur oder Wartung via Internet bereit stellen und einfach aktualisieren. Auch könnte ein Experte am Monitor die Arbeiten des Technikers verfolgen und Arbeitsanweisungen geben. Hier setzt der Prototyp eines VR-basierten Überwachungs- und Konfigurationssystems an, das am Stuttgarter Fraunhofer-Institut für Produktionstechnik und Automatisierung (IPA) entwickelt wurde. "Unser System geht einen Schritt weiter als konventionelle Teleservice-Technologien", stellt Projektleiter Markus Mersinger fest. Während Teleservice zum Ziel hat,

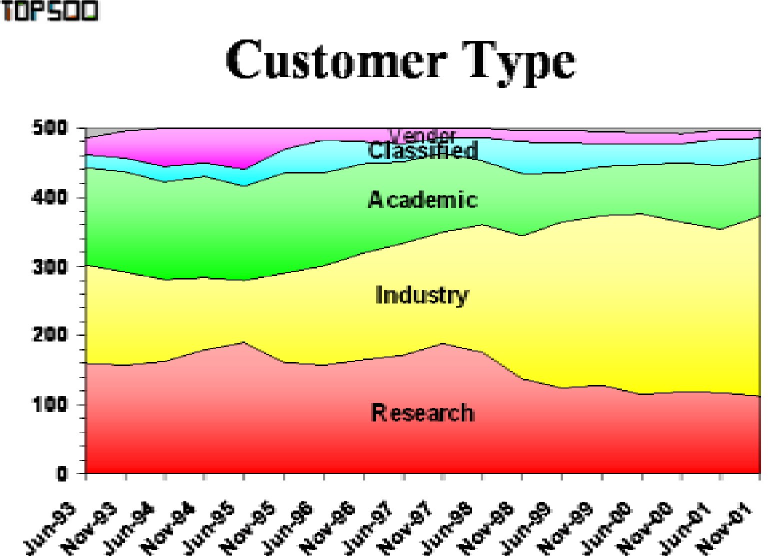
entstandene Defekte bei einer Maschine oder Anlage über Fernwartung und Ferndiagnose möglichst schnell zu beheben, geht es bei Teleoperations darum, Fehler vorherzusehen und zu verhindern. "Wenn nötig, auch mittels Ferneingriff in den laufenden Prozess", sagt der Fraunhofer-Forscher.Das Produktionssystem mit all seinen Spezifika existiert als Modell in einer virtuellen Welt. Dabei hat es jederzeit direkten Bezug zur realen Anlage; nicht nur im Aufbau, sondern auch in seinen jeweils aktuellen Prozesszuständen. Die nötigen Sensordaten liefert das reale Steuerungssystem. "In der VR-Welt ist jeder Punkt erreichbar, selbst Komponenten, die im laufenden Betrieb real nicht zugänglich sind wie der Bearbeitungsraum einer Werkzeugmaschine oder der Innenraum einer Roboterzelle", erklärt Mersinger. Diese plastische und wirklichkeitsgetreue Visualisierung helfe nicht nur, Fehler frühzeitig zu erkennen, sondern bereits vor dem Produktionsbeginn lassen sich alle Prozessschritte der Anlage simulieren und optimieren sowie das Bedienpersonal schulen.

http://www.informationweek.de/index.php3?/channels/channel06/011328.htm

3 Aus TOP500 von November 2001



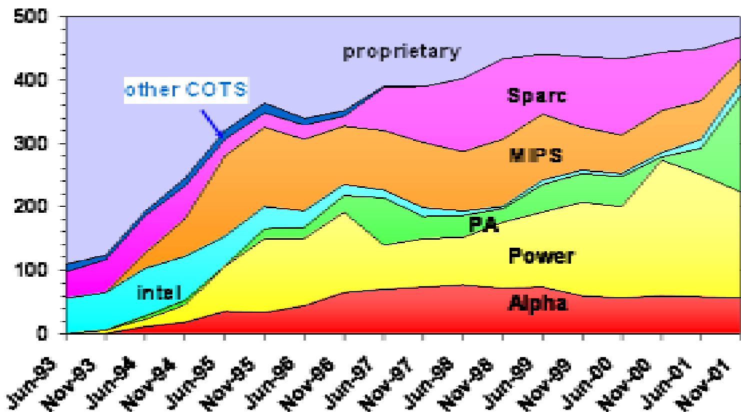
Die gesamte Rechenleistung der 500 leistungsfähigsten Rechner ist in weniger als 10 Jahren mit einem Faktor > 100 gestiegen.



Industrielle Anwendungen des Hochleistungsrechnen nehmen deutlich zu: Computersimulationen bei der Entwicklung industrieller Produkte.

TOP500

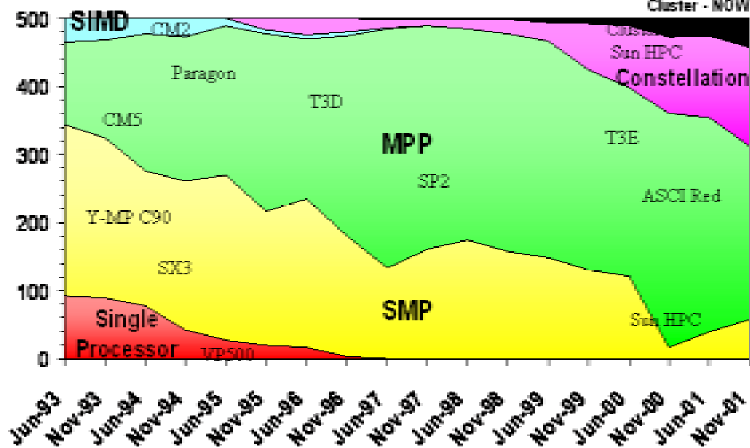
Chip Technology



Das Verhältnis von proprietären zu Standardprozessoren beim Einsatz in Hochleistungsrechnern hat sich in den letzten acht Jahren umgekehrt.

TOP500

Architectures



Vektorrechner (SIMD) spielen praktisch keine Rolle mehr, Workstation- und PC-Cluster sind auf dem Vormarsch. MPP: Message Passing Processing, SMP: Symetric Multiprocessing

4 Strukturen von Parallelrechner-Architekturen

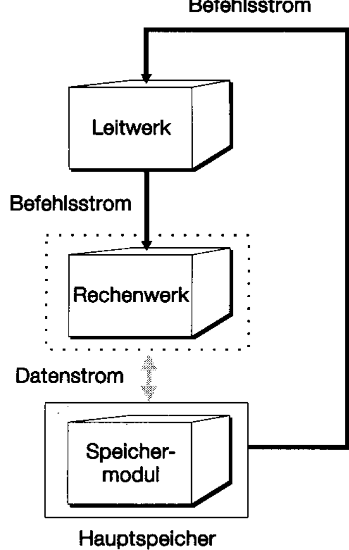
Da bei der Entwicklung von Parallelrechnern das Ziel in der Leistungssteigerung gegenüber sequentiellen Rechnern liegt, denkt man einerseits zu recht, andererseits vorschnell an Multiprozessorsysteme oder Supercomputer wie die Cray. Denn die Organisation von Parallelarbeit kann mit unterschiedlich hohem Einsatz von Hardware-Betriebsmitteln realisiert werden und damit mit entsprechend unterschiedlichen Architekturen. Nach der Einordnung der Parallelrechner mit der Flynnschen Klassifikation gibt die folgende Tabelle einen Überblick und zeigt, wie man Parallelrechner-Architekturen nach dem Einsatz der Hardware-Betriebsmittel unterscheiden kann.

4.1 Einordnung von Parallelrechnern - Flynnsche Klassifikation

Flynn charakterisiert Rechner als Operatoren auf zwei verschiedenartigen Informationsströmen: dem Befehlsstrom (bzw. den Befehlsströmen) und dem Datenstrom (bzw. den Datenströmen)

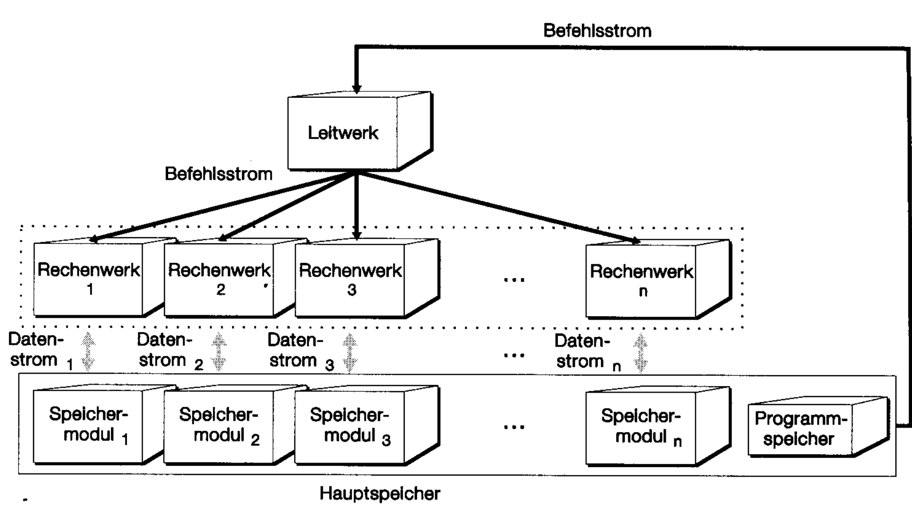
SISD Single instruction stream over a single data stream

von-Neumann-Architekturen (Einprozessor-rechner)



SIMD Single instruction stream over multiple data streams

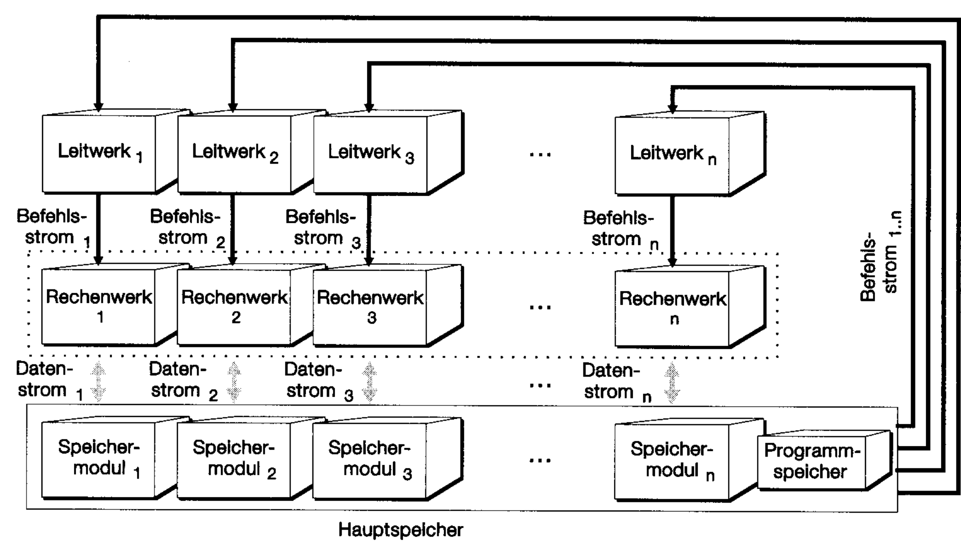
Feldrechner und Vektorrechner



MISD Multiple instruction streams over a single data stream

MIMD Multiple instruction streams over multiple data streams

die Multiprozes-sorsysteme



- Einprozessor-Systeme

  Klassisches System der Kategorie SISD mit einer zentralen Prozessor (CPU). Spezielle Prozessoren für die Ein/Ausgabe werden dabei nicht gezählt.

- Arrays von gleichartigen, *universellen* Recheneinheiten (RE), Feldrechner

  Rechner mit einem Feld von regelmäßig verbundenen Verarbeitungselementen, die unter Aufsicht einer zentralen Steuereinheit immer gleichzeitig dieselbe Maschinenoperation auf verschiedenen Daten ausführen

- Pipelines aus verschiedenartigen, *spezialisierten* Recheneinheiten

  sind von der Kategorie SIMD und sind in der Regel eindimensionale Anordnungen aus Recheneinheiten, die in der Regel stärker spezialisiert sind als die Recheneinheiten eines RE-Arrays.

  Vektorrechner: Rechner, dessen Rechenwerk aus mindestens einer pipelineartig aufgebauten Funktionseinheit zur Verarbeitung von Arrays (Vektoren) von Gleitpunktzahlen besteht.
  Eine solche Funktionseinheit nennt man **Vektorpipeline**

- Systolische Arrays

  sind von der Kategorie SIMD. Man kann sie als mehrdimensionale Erweiterung des Pipeline-Prinzips betrachten.

- Multiprozessor-Systeme

  gehören in die Kategorie MIMD. Sie bestehen aus mehr als einem Prozessor. Sie sind *homogen*, wenn alle Prozessoren hardwaremäßig gleich sind (bezüglich ihrer Rolle im System können sie sich unterscheiden), andernfalls sind sie heterogen. Spielen die Prozessoren eines Multiprozessor-Systems verschiedenartige Rollen, so nennt man das System *asymmetrisch*. Im Gegensatz dazu sind die Prozessoren eines *symmetrischen* Multiprozessor-Systems bezüglich ihrer Rolle im System austauschbar.

  - speichergekoppelte Systeme
    .

    - Es gibt einen zentralen Speicher für alle Prozessoren oder zumindest einen allen zugänglichen Kommunikationsspeicher. Man nennt sie deshalb auch *Systeme mit gemeinsamem Speicher.* Alle Prozessoren besitzen einen gemeinsamen Adreßraum. Kommunikation und Synchronisation geschehen über gemeinsame Variablen.. Man unterscheidet:

      1. Symmetrischer Multiprozessor ***SMP***: ein globalen Speicher
      2. Distributed-shared-memory-System ***DSM:*** gemeinsamer Adreßraum trotz physikalisch verteilter Speichermodule

  - nachrichtenorientierte Systeme

    Hier gibt es nur private Speicher . Die Prozessoren können daher nicht über den Speicher kommunizieren, sondern müssen sich explizit Nachrichten (*messages*) über eine Verbindungsstruktur zusenden. Man spricht deshalb auch *von Systemen mit verteiltem Speicher.* Jeder Prozessore verfügt über

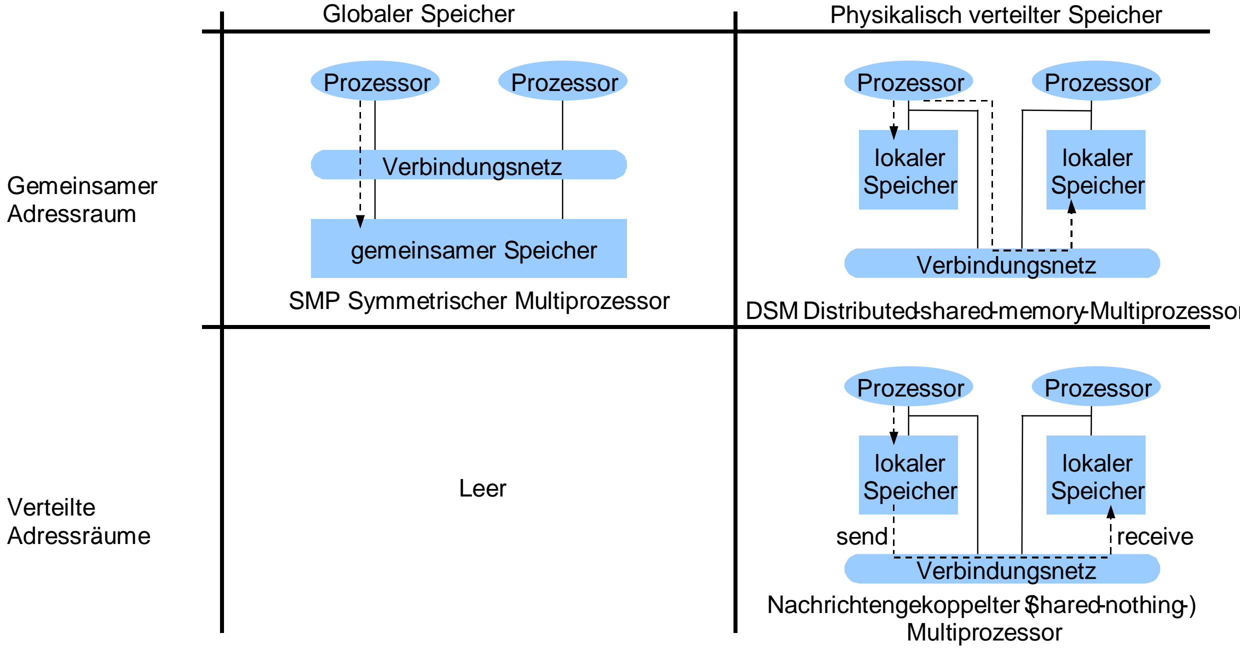
seinen eigenen Adressraum.

• Verteiltes Rechnen in einem **Workstation-Cluster**	• Kopplungsgrad nimmt ab, Programme müssen immer grobkörniger sein
• **Metacomputer**: Zusammenschluß weit entfernter Rechner	• Skalierbarkeit der Hardware nimmt zu.

Erläutern Sie die Flynnsche Klassifizierung von Rechnern (SISD, SIMD, MISD, MIMD)

In der Einleitung heißt es, die Vektorrechner seien 15 Jahre führend auf dem Gebiet der Supercomputer gewesen. Warum tauchen diese Vektorrechner nicht als eigene Kategorie in der obigen Tabelle auf?

Konfigurationen von Multiprozessoren



4.2 Speichergekoppelte Multiprozessoren

Alle Prozessoren besitzen einen gemeinsamen Adressraum; Kommunikation und Synchronisation geschieht über gemeinsame Variablen.

Je nach Speicheranordnung werden bei speichergekoppelten Multiprozessoren bezüglich der Zugriffszeiten folgende Modelle unterschieden.:

Uniform-memory-access-Modell (UMA):

Alle Prozessoren greifen gleichermaßen auf einen gemeinsamen Speicher zu. Insbesondere ist die Zugriffszeit aller Prozessoren auf den gemeinsamen Speicher gleich. Jeder Prozessor kann zusätzlich einen lokalen Cache-Speicher besitzen. Typische Beispiele: die symmetrischen Multiprozessoren (SMP)

Nonuniform-memory-access-Modell (NUMA):

Die Zugriffszeiten auf Speicherzellen des gemeinsamen Speichers variieren je nach dem Ort, an dem sich die Speicherzelle befindet. Die Speichermodule des gemeinsamen Speichers sind physikalisch auf die Prozessoren aufgeteilt. Typische Beispiele: Distributed-Shared-Memory-Systeme. (DSM)

4.3 Nachrichtengekoppelte Multiprozessoren

Bei nachrichtenorientierten Systemen spricht man in Analogie zu UMA und NUMA von **No-remote-memory-access-Modell (NORMA**),. Kein Prozessor hat selbst Zugriff auf den Speicher eines anderen Prozessors, sondern muss mittels Nachrichtenaustausch mit dem Prozessor, der den Speicher besitzt, die Speicherzellen anfordern.

Bezüglich der Zugriffszeiten auf Daten eines anderen Prozessors unterscheidet man zwischen

Uniform-communication-architecture-Modell (UCA): Zwischen allen Prozessoren können gleich lange Nachrichten mit der einheitlicher Übertragungszeit geschickt werden.

Non-uniform-communication-architecture-Modell (NUCA): Die Übertragungszeit des nachrichtentransfers zwischen den Prozessoren ist je nach Sender- und Empfänger-Prozessor verschieden lang.

Das NUCA-Modell kommt bei den nachrichtengekoppelten Multiprozessoren natürlich am häufigsten vor, und zwar immer dann, wenn eine Nachricht vom Empfänger-bis zum Ziel-Knoten über Zwischenknoten geleitet werden muß *(multiple hops).*

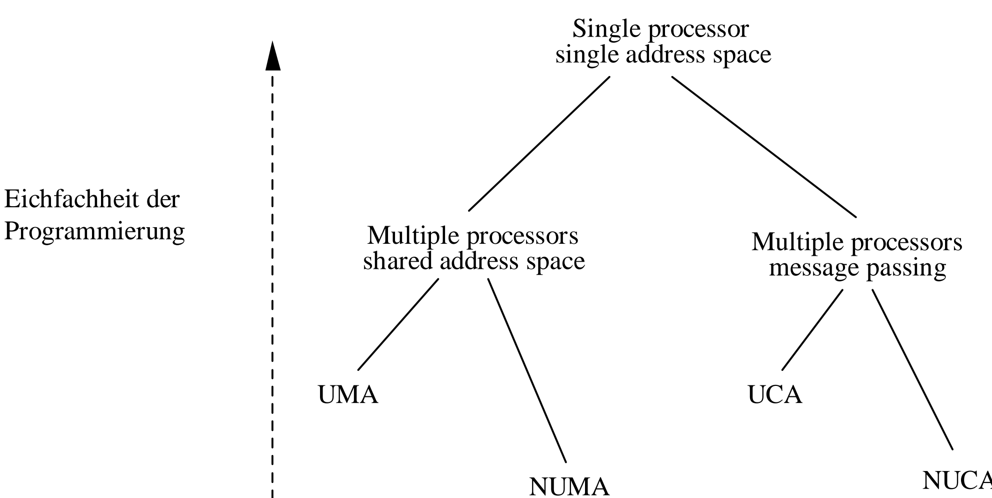
Falls das Multiprozessorsystem aus einer Menge hardwaremäßig gleichartiger Prozessoren besteht, spricht man von einem **homogenen,** ansonsten von einem **inhomogenen Multiprozessorsystem.**

Multiprozessoren, die aus mehreren autonomen Rechnern bestehen, werden auch als Multicomputer bezeichnet. Dazu können speicher- und nachrichtengekoppelte Systeme zählen. Insbesondere ist dabei an diejenigen Rechner gedacht, die aus einer Vervielfachung vollständiger Zentraleinheiten von Großcomputern oder Vektorrechnern aufgebaut sind. Beispiele dafür sind die älteren IBM-Großrechner IBM 3090-200E mit zwei Zentraleinheiten bis hin zum IBM 3090-600E mit sechs Zentralemheiten. Diese Systeme besitzen lokale Speicher und sind über gemeinsame Speicher gekoppelt. Auch die Vektor-Supercomputer Cray-XM-P, Cray-Y-MP, Cray-2 und Cray-C90 können als Multicomputer oder Multiprozessoren eingeordnet werden. Gelegentlich wird der Begriff „Multicomputer" auch synonym für nachrichtengekoppelte Multiprozessoren verwendet.

Workstation- oder PC-Cluster unterscheiden sich von Multiprozessoren durch die größere räumliche Entfernung der Verarbeitungseinheiten, durch die im Regelfall wesentlich langsamere Übertragungsgeschwindigkeit der Verbindungseinrichtungen und durch ihre Verwendung als Arbeitsplatzrechner im Normalbetrieb. Workstation- und PC-Cluster werden deshalb

im folgenden unter dem Stichwort verteilte Systeme behandelt und als **virtuelle Multiprozessoren** oder virtuelle **Parallelrechner** sowie beim Zusammenschluß räumlich weit entfernter Rechner als **Hyper-** oder **Meta**computer bezeichnet. Programmiert werden verteilte Systeme heute meist mittels der nachrichtenbasierten Programmierschnittstellen PVM oder MPI, die genauso auf nachrichtengekoppelten Multiprozessoren eingesetzt werden. Deshalb und wegen der steigenden Übertragungsgeschwindigkeiten bei lokalen Netzen wie auch bei Weitverkehrsnetzen wird in Zukunft keine klare Trennung zwischen Multiprozessoren und verteilten Systemen mehr möglich sein [Unge97].

Zugriffszeit-/Übertragungszeit-Modelle



4.4 Ebenen der Parallelität

Bei den nun folgenden Betrachtungen der **Ebenen der Parallelität** wird von einem parallelen Programm ausgegangen, bei dem die Parallelität explizit vorliegt. Ob diese Betrachtungen in der höheren Programmiersprache, der Zwischensprache oder der Maschinensprache durchgeführt werden, ist für die Unterscheidung der Parallelitätsebenen unerheblich. Bei der Übersetzung aus einer höheren Programmiersprache in eine Zwischen- oder Maschinensprache kann Parallelität von einer Ebene zu einer anderen transformiert werden.

Ein paralleles Programm läßt sich als halbgeordnete Menge von Befehlen darstellen, wobei die Ordnung durch die Abhängigkeiten der Befehle untereinander gegeben ist. Befehle, die nicht voneinander abhängig sind, können parallel ausgeführt werden. Eine total geordnete Teilmenge von Befehlen eines parallelen Programms bildet eine sequentielle Befehlsfolge. Dabei können verschiedene sequentielle Befehlsfolgen voneinander unabhängig sein.

Die **Körnigkeit** oder **Granularität** *(grain size)* ergibt sich aus dem Verhältnis von Rechenaufwand zu Kommunikations- oder Synchronisationsaufwand. Sie wird im folgenden nach

der Anzahl der auszuführenden Befehle in einer sequentiellen Befehlsfolge definiert, d.h. nach der Anzahl der auszuführenden Befehle, bevor eine Kommunikation oder Synchronisation mit einer anderen Befehisfolge notwendig wird.

Hinsichtlich der Kömigkeit paralleler Programme (in Quell-, Zwischen- oder Maschinensprache) können fünf **Ebenen der Parallelität** unterschieden werden:

- **Programmebene (oder Jobebene):** Diese Ebene wird durch die parallele Verarbeitung verschiedener Programme charakterisiert, die vollständig voneinander unabhängige Einheiten ohne gemeinsame Daten und mit wenig oder keinerlei Kommunikations- und Synchronisationsbedarf sind. Parallelverarbeitung auf dieser Ebene wird vom Betriebssystem organisiert.
- **Prozeßebene** (oder Taskebene): Parallelität auf dieser Ebene tritt auf, wenn ein Programm in eine Anzahl parallel auszuführender Prozesse (hier im Sinne **schwergewichtiger Prozesse -** *heavy-weighted processes, coarse-grain tasks)* zerlegt wird. Jeder Prozeß besteht aus vielen tausend sequentiell geordneten Befehlen und umfaßt eigene Datenbereiche. Typische Beispiele sind UNIX-Prozesse und Tasks in Ada. Da die einzelnen Prozesse innerhalb eines Programms ablaufen, müssen sie synchronisiert werden, und in der Regel kommunizieren sie miteinander. Von Betriebssystemseite findet eine Unterstützung dieser Parallelitätsebene durch Betriebssystem-Primitive zur Prozeßverwaltung, Prozeßsynchronisation und Prozeßkommunikation statt.
- **Blockebene**: Diese Ebene betrifft Anweisungsblöcke oder leichtgewichtige **Prozesse** *(threads, light-weighted processes).* Die leichtgewichtigen Prozesse unterscheiden sich von den schwergewichtigen darin, daß sie aus weniger Befehlen bestehen und mit anderen leichtgewichtigen Prozessen einen gemeinsamen Adreßbereich teilen. Dieser gemeinsame Adreßraum ist in der Regel der Adreßraum eines umfassenden, schwergewichtigen (U-NIX-)Prozesses. Typische Beispiele für leichtgewichtige Prozesse sind die Threads gemäß POSIX-1003.lc-1995-Standard, wie sie in mehrtädigen Betriebssystemen Verwendung finden. Die Kommunikation geschieht über die gemeinsamen Daten, die Synchronisation über Schloßvariablen *(mutexes)* und Bedingungsvariablen *(condition variables)* oder darauf aufbauenden Synchronisationsmechanismen. Da für die Ausführung eines solchen leichtgewichtigen Prozesses kein eigener Adreßbereich geschaffen wird, ist im Vergleich zu einem schwergewichtigen Prozeß der Aufwand für die Prozeßerzeugung, -beendigung oder einen Prozeßwechsel gering. Zur Blockebene der Parallelität gehören außerdem innere oder äußere parallele Schleifen in FORTRAN-Dialekten, die Verwendung von *Microtasking* und die Anweisungsblöcke bei der grobkörnigen Datenfiußtechnik[3]. Für viele, meist numerische Programme liegt auf der Blockebene durch parallel ausführbare Schleifeniterationen die potentiell größte Parallelität vor.
- **Anweisungsebene** (oder **Befehlsebene):** Auf dieser Ebene können einzelne Maschinenbefehle oder elementare Anweisungen (in der Sprache nicht weiter zerlegbare Datenoperationen) parallel zueinander ausgeführt werden.Optimierende Compiler für superskalare Prozessoren und für VLIW-Prozessoren sind in der Lage, Parallelität auf der Befehl sebene durch die Analyse der sequentiellen Befehlsfolge, die durch Übersetzung eines imperativen, sequentiellen Programms entsteht, zu bestimmen und durch eventuelle Umordnungen der Befehle für den Prozessor nutzbar zu machen. In Datenflußsprachen

[3] Bei der grobkörnigen Datenflußtechnik besteht das parallele Programm aus einer Anzahl von sequentiellen Anweisungsblöcken und einer Netzliste, welche die Zusammenhänge zwischen den Blöcken wiedergibt. Ein Anweisungsblock kann ausgeführt werden, sobald alle Eingabedaten vorhanden sind

und in manchen funktionalen Programmiersprachen wird Parallelität auf der Anweisungsebene sogar explizit ausgedrückt.

- **Suboperationsebene:** Eine elementare Anweisung wird durch den Compiler oder in der Maschine in Suboperationen aufgebrochen, die parallel ausgeführt werden. Typische Beispiele dafür sind Vektor- oder Feldoperationen, die von einem Vektorrechner mittels einer Vektor-Pipeline „überlappt parallel" oder von einem Feld-rechner durch ein Feld von Verarbeitungseinheiten „datenparallel" ausgeführt werden. Um Parallelität auf der Suboperationsebene auszudrücken, müssen komplexe Datenstrukturen und Datenoperationen entweder in der höheren Programmiersprache verfügbar sein oder von einem vektorisierenden oder parallelisierenden Compiler aus einer sequentiellen Programmiersprache für die Maschinensprache erzeugt werden. Beispiele für Programmiersprachen mit komplexen Datenstrukturen sind APL, PASCAL-SC, FORTRAN-90, HPF, C*, Parallaxis und Modula2*. Suboperationsparallelität bietet in Verbindung mit komplexen Operationen wie beispielsweise Vektor- oder Matrixoperationen die Möglichkeit, einen hohen Parallehtätsgrad zu erreichen. Was bei konventionellen Sprachen auf der Blockebene durch mehrere geschachtelte Schleifen programmiert werden muß, kann dann oft durch eine einzige elementare Anweisung ausgedrückt und auf der Maschine parallel ausgeführt werden. Falls ein Parallelrechner so entworfen ist, daß Parallelarbeit auf der Suboperationsebene für einen Satz komplexer Maschinenbefehle ausgeführt wird, entfällt ein Großteil der Schleifen und damit ein wesentlicher Anteil der sonst hohen Parallelarbeit auf der Blockebene.

Programm-, Prozeß- und Blockebene werden häufig auch als **grobkörnige** *(large grained)* Parallelität, die Anweisungsebene als **feinkörnige** *finely grained)* Parallelität bezeichnet. Seltener wird auch von mittelkörniger *(medium grained)* Parallelität gesprochen, dann ist meist die Blockebene gemeint.

Programm-ebene	Kennzeichnung	Kommentar, Beispiele	Paralleli-tätsgrad
Benutzer-programme (Jobs)	Mehrere Jobs werden gleichzeitig ausgeführt	Der Parallelitätsgrad auf der Job-Ebene kann nicht sehr hoch sein. Die Job-Ebene spielt in der Parallelrechner-Architektur keine große Rolle und wird deshalb nicht weiter betrachtet.	niedrig
Prozesse (Tasks)	Mehrere Prozesse laufen konkurrent ab	Parallelarbeit muß zur Zeit noch vom Programmierer festgelegt werden. Compiler dafür sind in Arbeit.	hoch
Anweisungen	Mehrere Anweisungen eines Programms werden gleichzeitig ausgeführt.	Beschränkt man sich auf *Basisblöcke*[4] ist der Parallelitätsgrad gering, da im Schnitt jede 5. Anweisung eine Verzweigungsanweisung ist (einfach) Besser: Man betrachtet größere, über Schleifen hinausgehende Programmteile (aufwendiger). Compiler können die Abhängigkeitsanalysen und die Parallelisierung durchführen.	niedrig bis hoch[5]
Elementaroperationen	Mehrere Operationen eines zusammengesetzten Ausdrucks werden gleichzeitig ausgeführt	Beim Ausdruck: Y=((X-X1)*(Y2-Y1)/(x2-X1))+Y1 könnten X-X1, Y2-Y1 und X2-X1 parallel ausgeführt werden. x x1 y2 y1 x2 x1 y1 (Verarbeitungsbaum des Ausdrucks) Parallelität ist für den Compiler leicht zu erkennen.	niedrig

Ebenen der Parallelarbeit in von-Neumann-Sprachen

4 Ein Basisblock ist die Anweisungsfolge zwischen zwei Verzweigungsbefehlen.

5 Je nach dem, über welche Programmiereinheiten die parallel ausführbaren Anweisungen ermittelt werden (niedrig bei Basisblocks, hoch für Schleifen)

Parallelitätsebene	Architekturen
Prozesse	*Multiprozessorsysteme* mit verteiltem Speicher *Nachrichtenorientierte MIMD-Systeme*
Datenstrukturen	*Vektorrechner* *Rechenelemente-Arrays* *Multiprozessorsysteme* mit verteiltem Speicher
Anweisungen der höheren Programmiersprache	*Multiprozessesorsysteme* mit gemeinsamem Speicher und Schleifen-Parallelisierierung *Vielfädige Architekturen* feiner Granularität
Maschinenbefehle (Anweisungen inklusive Elementaroperationen)	(*RISC* Prozessoren mit Phasenpipelining) *Superskalare Prozessoren* mit parallel arbeitenden Funktionseinheiten *VLIW* Maschinen mit parallel arbeitenden Funktionseinheiten *feinkörnige Datenflußrechner* mit parallel arbeitenden Funktionseinheiten

Parallelitätsebenen und Architekturformen [GIL93, S.150]

Die Tabelle gibt einen Überblick, welcher Parallelitätsebene die verschiedenen parallelen Architekturen, von denen in den weiteren Kapiteln die Rede sein wird, zugeordnet werden.

Die Nutzung der Parallelität auf der Anweisungsebene ist das wesentliche Merkmal der superskalaren Prozessoren. Sie bildet aber auch die Grundlage in einer Reihe anderer Architekturen (siehe Tabelle). Die Besonderheiten der Parallelität auf der Anweisungsebene werden deshalb im nächsten Abschnitt vorgestellt.

4.5 Techniken der Parallelarbeit

Den **Ebenen der Parallelität** in einem Programm stehen **Techniken der Parallelarbeit** in der Hardware, unterstützt durch das Betriebssystem, gegenüber. Diese Techniken korrespondieren grob mit den Ebenen der Parallelität in der Programmier-, Zwischen- oder Maschinensprache. Die Korrespondenz ist eine Frage der effizienten Implementierbarkeit einer Ebene der Parallelität durch eine spezifische Technik der Parallelarbeit. Anweisungsparallelität läßt sich beispielsweise nicht effizient auf einem nachrichtengekoppelten Multiprozessorsystem ausnutzen, wohl aber durch die Superskalartechnik eines Mikroprozessors.

Einen Überblick gibt die folgende Tabelle, bei der in der linken Spalte Parallelarbeitstechniken eingetragen sind und ein X in einer der Spalten eine effiziente Nutzung der Parallelität auf der im Kopf der Tabelle angetragenen Parallelitätsebene bedeutet. Techniken zur Implementierung von Parallelität auf Programm- und Taskebene können in der Praxis kaum unterschieden werden.

Die Techniken der Parallelarbeit sind in der Tabelle in vier Gruppen eingeteilt: Techniken der Parallelarbeit durch Rechnerkopplung, durch Prozessorkopplung, in der Prozessorarchitektur und SIMD-Techniken.

Häufig kommt auch die Kombination mehrerer Techniken der Parallelarbeit bei Mehr-Ebenen-parallelen Rechnern vor, d.h. Rechnern, die mehrere Ebenen der Parallelität unterstützen. Dazu zählen mittlerweile praktisch alle Multiprozessoren, die grobkörnige Parallelität (Programm-, Prozeß- oder Blockebene) durch ihre mehrfach vorhandenen Prozessoren und feinkörnige Parallelität (Befehlsebene) durch die Superskalartechnik ihrer Mikroprozessoren nutzen.

In dieser Aufstellung der Parallelarbeitstechniken wird mit absteigender Reihenfolge die Kopplung zwischen den parallel arbeitenden Verarbeitungselementen oder Verarbeitungsein-

heiten enger, d.h., die Kommunikationsgeschwindigkeit steigt und die Synchronisationskosten sinken. Weiterhin zeigt sich in der absteigenden Reihung ein Trend von asynchronen hin zu (takt)synchronen Techniken der Parallelarbeit.

Techniken der Parallelarbeit vs. Parallelitätsebenen

Parallelarbeitstechniken	Programmebene	Prozeßebene		Anweisungsebene	Suboperationsebene
Techniken der Parallelarbeit durch Rechnerkopplung					
Hyper- und Metacomputer	X	X			
Workstation-Cluster	X	X			
Techniken der Parallelarbeit durch Prozessorkopplung					
Nachrichtenkopplung	X	X			
Speicherkopplung (SMP)	X	X	X		
Speicherkopplung (DSM)	X	X	X		
Grobkörniges Datenflußprinzip		X	X		
Techniken der Parallelarbeit in der Prozessorarchitektur					
Befehlspipelining				X	
Superskalar				X	
VLIW				X	
Überlappung von E/A- mit CPU-Operationen				X	
Feinkörniges Datenflußprinzip				X	
SIMD-Techniken					
Vektorrechnerprinzip					X
Feldrechnerprinzip					X

23

4.6 Beschleunigung und Effizienz von parallelen Systemen

P(1): Anzahl der auszuführenden (Einheits-)Operationen des Programms auf einem Einprozessorsystem.

P(n): Anzahl der auszuführenden (Einheits-)Operationen des Programms auf einem Multiprozessorsystem mit *n* Prozessoren.

T(1): Ausführungszeit auf einem Finprozessorsystem in Schritten (oder Takten).

T(n): Ausführungszeit auf einem Multiprozessorsystem mit *n* Prozessoren in Schritten (oder Takten).

Es wird von folgenden vereinfachenden Voraussetzungen ausgegangen:

T(l) P(1), da in einem Einprozessorsystem jede (Einheits-)Operation in genau einem Schritt ausgeführt werden kann.

$T(n) < P(n)$, da in einem Multiprozessorsystem mit *n* Prozessoren ($n > 2$) in einem Schritt mehr als eine (Einheits-)Operation ausgeführt werden kann.

Die Beschleunigung (Leistungssteigerung, *Speed-up) S(n)* ist definiert als

$$S(n) = \frac{T(1)}{T(n)}$$

und gibt die Verbesserung in der Verarbeitungsgeschwindigkeit an, wobei sich der Wert natürlich nur auf das jeweils bearbeitete Programm bezieht bzw. als Mittelwert für eine Reihe von Programmen angesehen werden kann.

Üblicherweise gilt

$$1 \leq S(n) \leq n$$

Danach arbeitet also im schlechtesten Fall das Multiprozessorsystem gerade so schnell wie das Einprozessorsystem und im besten Fall ist die Leistungssteigerung linear zur Anzahl der eingesetzten Prozessoren[7]. Diese Ungleichung kann als Grundlage für alle weiteren Überlegungen angesehen werden.

Die Effizienz *E(n) (efflciency)* ist definiert als

$$E(n) = \frac{S(n)}{n}$$

und gibt die relative Verbesserung in der Verarbeitungsgeschwindigkeit an, da die Leistungssteigerung mit der Anzahl der Prozessoren *n* normiert wird. Unter Voraussetzung der obigen Ungleichung gilt:

$$\frac{1}{n} \leq E(n) \leq 1$$

Man kann die Begriffe Beschleunigung und Effizienz „algorithmenunabhängig" oder „algorithmenabhängig" definieren [Fost95l. Im ersten Fall setzt man den besten bekannten sequentiellen Algorithmus für das Einprozessorsystem in Beziehung zum vergleichenden parallelen Algorithmus für das Multiprozessorsystem. Man erhält dann die sogenannte **absolute Beschleunigung bzw. die absolute Effizienz.** Im zweiten Fall benutzt man den parallelen Algorithmus so, als sei er sequentiell, und mißt dessen Laufzeit auf einem Einprozessorsystem. Man spricht dann von **relativer Beschleunigung und relativer Effizienz.** In diesem Fall kommt der für die Parallelisierung erforderliche Zusatzaufwand an Kommunikation und Synchronisation „ungerechterweise" auch für den sequentiellen Algorithmus zum Tragen.

Von Skalierbarkeit eines Parallelrechners spricht man, wenn das Hinzufügen von weiteren Verarbeitungselementen zu einer kürzeren Gesamtausführungszeit führt, ohne daß das Programm geändert werden muß. Insbesondere meint man damit eine lineare Steigerung der Beschleunigung mit einer Effizienz nahe bei Eins.

Wichtig für die Skalierbarkeit ist eine angemessene Problemgröße. Bei fester Problemgröße und steigender Prozessorzahl wird ab einer bestimmten Prozessorzahl eine Sättigung eintreten. Die Skalierbarkeit ist in jedem Fall beschränkt.

Skaliert man mit der Anzahl der Prozessoren auch die Problemgröße (scaled problem analysis, so tritt dieser Effekt bei gut skalierenden Hardware-oder Software-Systemen nicht auf. Die Belastung eines Systems muß der jeweiligen Leistungsfähigkeit entsprechen. Die folgen-

de Analogie demonstriert diese Zusammenhänge: Ein Gärtner pflanzt einen Baum in einer Minute. Dann ist nicht zu erwarten, daß 60 Gärtner einen Baum in einer Sekunde pflanzen. Auch hier ist klar, daß sich eine sinnvolle Leistungssteigerung erst beirn Pflanzen von tausend oder zehntausend Bäumen einstellen kann. Der Mehraufwand für die Parallelisierung **R(n)** ist definiert als

$$R(n) = \frac{P(n)}{P(1)}$$

und beschreibt den bei einem Multiprozessorsystem erforderlichen Mehraufwand für die Organisation, Synchronisation und Kommunikation der Prozessoren. Es gilt:

$$1 \leq R(n)$$

Das bedeutet, daß die Anzahl der auszuführenden Operationen eines parallelen Programms größer als diejenige des vergleichbaren sequentiellen Programms ist.

Der **Parallelindex *I(n)*** *(parallel index)* ist definiert als

$$I(n) = \frac{P(n)}{T(n)}$$

und gibt den mittleren Grad an Parallelität bzw. die Anzahl der parallelen Operationen pro Zeiteinheit an.

Die **Auslastung** *U(n) (utilization)* ist definiert als

$$U(n) = \frac{I(n)}{n}$$

$$U(n) = R(n) \bullet E(n)$$

$$U(n) = \frac{P(n)}{n \bullet T(n)}$$

und entspricht dem normierten Parallelindex. Sie gibt also an, wieviele Operationen jeder Prozessor im Durchschnitt pro Zeiteinheit ausgeführt hat. Aus diesen Definitionen lassen sich einige Folgerungen ableiten:

- Alle definierten Ausdrücke haben für $n = 1$ den Wert 1.
- Der Parallelindex gibt eine obere Schranke für die Leistungssteigerung:

Die Auslastung ist eine obere Schranke für die Effizienz:

Die Bedeutung dieser Definitionen soll mit einem Zahlenbeispiel verdeutlicht werden. Ein Einprozessorsystem benötige für die Ausführung von 1000 Operationen 1000 Schritte. Ein Multiprozessorsystem mit 4 Prozessoren benötige dafür 1200 Operationen, die aber in 400 Schritten ausgeführt werden können. Damit gilt also

$P(1) = T(1) = 1000$, $P(4) = 1200$ und $T(4) = 400$

Daraus ergibt sich:

$S(4) = 2.5$ und $E(4) = 0.625$

Die Leistungssteigerung verteilt sich also zu 62,5% auf jeden Prozessor.

$I(4) = 3$ und $U(4) = 0.75$

Es sind im Mittel also drei Prozessoren gleichzeitig tätig, d.h., jeder Prozessor ist nur zu 75% der Zeit aktiv.

R(4) = 1.2

Bei der Ausführung auf einem Multiprozessorsy stern sind 20% mehr Operationen als bei der Ausführung auf einem Einprozessorsystem notwendig.

Bei diesen Berechnungen muß man sich darüber im klaren sein, daß die Ergebnisse lediglich Mittelwerte für die Programme darstellen, für die Messungen durchgeführt werden. Um fundierte Aussagen für ein bestimmtes System zu erhalten, ist darauf zu achten, daß das ganze Anwendungsgebiet abgedeckt wird und entsprechend viele Algorithmen ausgewertet werden.

Die wichtigste Abschätzung für die Leistungssteigerung *S(n)* durch Nutzung eines Multiprozessorsy sterns stammt von Amdahl. Dieser betrachtet einen weiteren Parameter a (0< a < 1), der den Bruchteil eines Programms darstellt und der nur sequentiell bearbeitet werden kann. Die entstehende Formel ist als **Amdahis Gesetz** bekannt geworden.

Die Gesamtausführungszeit errechnet sich aus der Summe der Ausführungszeit des nur sequentiell ausführbaren Programmteils a und der Ausführungszeit des parallel ausführbaren Programmteils 1 - a, die letztere dividiert durch die Anzahl *n* der parallel arbeitenden Prozessoren. Es gilt somit unter der Vernachlässigung von Synchronisations- und Kommunikationszeiten:

Amdahls Gesetz

$$T(n) = T(1) \cdot \frac{1-a}{n} + T(1) \cdot a$$

a Summe der Ausführungszeit des nur sequentiell ausführbaren Programmteils

$$S(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{T(1) \cdot \frac{1-a}{n} + T(1) \cdot a}$$

$$= \frac{1}{\frac{1-a}{n} + a} = \frac{n}{(1-a) + n \cdot a}$$

$$S(n) \leq \frac{1}{a}$$

Amdahls Gesetz zufolge kann eine kleine Anzahl von sequentiellen Operationen die mit einem Parallelrechner erreichbare Beschleunigung signifikant begrenzen. Falls z.B. a = 1/10 des parallelen Programms nur sequentiell ausgeführt werden kann, so kann das gesamte Programm maximal zehnmal schneller als ein vergleichbares, rein sequentielles Programm verarbeitet werden. Amdahls Gesetz wurde auch als eines der stärksten Argumente gegen die weitere Entwicklung der Multiprozessorsysteme bezeichnet. Wenn nämlich Multiprozessorsysteme niemals mehr als vielleicht zehnbis zwanzigmal schneller als Einprozessorsysteme sein könnten, so würden sich die Entwicklungskosten für Multiprozessorsysteme kaum lohnen.

Es hat sich jedoch gezeigt, daß es viele parallele Programme gibt, die einen sehr geringen sequentiellen Anteil (a « 1) besitzen. Viele Anwendungen verlangen nach Rechnern, die um mehrere Größenordnungen schneller als die existierenden Einprozessorrechner sind. Als Erfahrungstatsache gilt, daß ein Problem für Multiprozessoren dann interessant ist, wenn der parallel bearbeitbare Anteil die Anzahl der zur Verfügung stehenden Prozessoren weit übersteigt. Heutige Erfahrungen zeigen, daß bei derartigen Problemen typischerweise eine fast lineare Beschleunigung erreicht werden kann.

In Multiprozessorsystemen treten auch noch andere Effekte auf, die ähnlich wie die synergetischen Effekte schwer abzuschätzen sind, im Gegensatz zu diesen jedoch zu einer Verschlechterung des Wirkungsgrades führen. Dazu gehören z.B.

- der Verwaltungsaufwand *(overhead),* der mit der Zahl der zu verwaltenden Prozessoren ansteigt,
- die Möglichkeit von Systemverklemmungen *(deadlocks)* und
- die Möglichkeit von Sättigungserscheinungen, die durch Systemengpässe *(bottlenecks)* verursacht werden.

Zusammenfassend läßt sich sagen, daß durchaus für ein vorgegebenes Multiprozessorsystem in einer bestimmten Konfiguration sowie für eine vorgegebene Menge von Programmen mit einer vorgegebenen Menge von Eingabedaten die oben angegebenen Maßzahlen wie Leistungssteigerung, Effizienz, Mehraufwand, Parallelindex und Auslastung gemessen werden können. Detaillierte, allgemeingültige Aussagen sind jedoch wegen der vielen nichtdeterministischen Effekte, die in der Praxis auftreten, schwer zu treffen.

Zusammenfassung: Speed up und Effizienz

- Begriffe Beschleunigung und Effizienz lassen sich „algorithmenunabhängig " oder „algorithmenabhängig" definieren.

- **absolute Beschleunigung** (absolute Effizienz):
  Man setzt den besten bekannten sequentiellen Algorithmus für das Einprozessorsystem in Beziehung zum vergleichenden parallelen Algorithmus für das Multiprozessorsystem.

- **relative Beschleunigung** (relative Effizienz):
  Man benutzt den parallelen Algorithmus so, als sei er sequentiell, und mißt dessen Laufzeit auf einem Einprozessorsystem.

  Der für die Parallelisierung erforderliche Zusatzaufwand an Kommunikation und Synchronisation kommt „ungerechterweise" auch für den sequentiellen Algorithmus zum Tragen.

Zusammenfassung:Skalierbarkeit eines Parallelrechners

- **Skalierbarkeit** eines Parallelrechners: das Hinzufügen von weiteren Verarbeitungselementen führt zu einer kürzeren Gesamtausführungszeit, ohne daß das Programm geändert werden muß.
- Insbesondere meint man damit eine lineare Steigerung der Beschleunigung mit einer Effizienz nahe bei Eins.
- Wichtig für die Skalierbarkeit ist eine angemessene Problemgröße.
- Bei fester Problemgröße und steigender Prozessorzahl wird ab einer bestimmten Prozessorzahl eine Sättigung eintreten. Die Skalierbarkeit ist in jedem Fall beschränkt.
- Skaliert man mit der Anzahl der Prozessoren auch die Problemgröße (*scaled problem analysis*), so tritt dieser Effekt bei gut skalierenden Hardware- oder Software-Systemen nicht auf.

5 Konzepte der Parallelarbeit

Die wichtigste *organisatorische* Maßnahme zur Leistungssteigerung in einer Rechnerarchitektur ist die Einführung eines möglichst hohen Grades an Parallelarbeit.[6] Erinnern Sie sich an die von Neumann Maschine: Alle Befehle und Operanden eines Programmes stehen in einem von der CPU getrennten Speicher und werden nach dem *Befehlszählerprinzip* einer nach dem anderen aus dem Speicher gelesen und von der CPU ausgeführt. Die *sequentielle* Bearbeitung eines Programms ist so gesehen die selbstverständlichste Sache der Welt. Schaut man sich die Mikrostruktur eines von Neumann Rechners an, stellt man fest, daß es auch hier bereits einen gewissen Grad an Parallelarbeit gibt: für den Transport von Daten werden diese parallel zwischen CPU und Peripherie (8,16,32 oder 64 Bit, je nach Prozessorgeneration) ausgetauscht, auch die Verbindungswege zwischen den Registern auf der CPU sind so breit wie die Register selbst. Auch die Verarbeitung der Daten geschieht bereits parallel: Das Rechenwerk führt arithmetische Operationen mit mehreren Bits gleichzeitig aus. In der Regel werden so viele Bits von der ALU parallel bearbeitet, wie die Register breit sind (das ist auch hier nur eine Frage der Prozessorgeneration). Das Rechenwerk eines 8 Bit Paralleladdierwerkes besteht dementsprechend aus 8 *Fulladder*. Es wäre durchaus denkbar, alle Bits nacheinander mit einem einzigen Fulladder zu addieren; man verwendet dann ein Serienaddierwerk. Man sieht hier unmittelbar den Vor- und Nachteil der Parallelarbeit. Je mehr Funktionen parallel ausgeführt werden, desto schneller ist die Aufgabe erledigt aber desto höher ist der Hardwareaufwand.

5.1 Parallelität und die sie einschränkende Abhängigkeiten

Der zentrale Begriff des parallelen Rechnens ist die in den auszuführenden Programmen zu findende Parallelität. Der Grad an Parallelität gibt an, wieviele Operationen parallel ausführbar sind. Dieser Parallelitätgrad ist im allgemeinen keine konstante Größe, sondern er hängt vom jeweiligen Stand der Ausführung, (d.h. vom Zustand des zugehörigen Programmlaufs) ab.

Man unterscheidet dabei zwei Arten von Parallelität :

- die explizite Datenparallelität
- die implizite Programmparallelität (welche die Datenparallelität mit einschließt).

Die explizite Datenparallelität ist dann gegeben, wenn strukturierte Datenmengen, wie z.B. Arrays oder Vektoren zu verarbeiten sind, wobei von den Operationen solcher Datenstrukturtypen von vornherein bekannt ist, wieweit sie parallel ausgeführt werden können. Ist z.B. das äußere Produkt (die komponentenweise Verknüpfung) zweier Vektoren der Länge N zu bilden, so ist damit explizit die Möglichkeit der gleichzeitigen Durchführung dieser Verknüpfungen mit dem Parallelitätgrad N vorgegeben, da bekanntlich keine Datenabhängigkeiten zwischen den Teiloperationen bestehen:

[6] Fortschritte in der Halbleitertechnik wie höhere Integrationsdichte, kürzerer Schaltzeiten, höhere Taktfrequenz sollen hier explizit von organisatorischen Maßnahmen abgesetzt werden.

$A = [a_0,a_1,a_2...a_{n-1}]$

$B = [b_0, b_1,b_2...b_{n-1}]$

$C = [a_0 \bullet b_0,\ a_1 \bullet b_1,\ a_2 \bullet b_2 \ldots.\ a_{n-1} \bullet b_{n-1}]$

5.1.1 Implizite Programmparallelität

Eine implizite Programmparallelität ist immer dann gegeben, wenn es in dem Algorithmus eine Menge von Operationen gibt, innerhalb derer es auf die Reihenfolge der Ausführung nicht ankommt. Dazu folgendes FORTRAN-Beispiel:

```
1       DO WHILE (X  .NE.  Stop)
2               Y := S(X)
3               IF  (X+L(X)  .EQ.  Y)  THEN
4                       L(X) = L(X) + L(Y)              Diese beiden Befehle
5                       S(X) = S(Y)                     sind parallel ausführbar
6               ELSE
7                       X = Y
8               ENDIF
9       ENDDO
```

Wie an dem Beispiel leicht zu sehen ist, ist die Reihenfolge der Anweisungen 4 und 5 gleichgültig und könnten deshalb auch gleichzeitig ausgeführt werden. Bei allen anderen Befehlen muß die Reihenfolge eingehalten werden und das Ergebnis der n-1ten Anweisung muß bekannt sein bei der n-ten Anweisung. Eine sequentielle Ausführung ist für diese Befehle zwingend.

Diese Analyse des Programmteils nennt man *Datenabhängigkeitsanalyse* und kann vom Compiler durchgeführt werden. Diese Methode hat inzwischen weite Verbreitung gefunden.

5.1.1.1 Datenabhängigkeiten und Datenabhängigkeitsanalyse

Am folgenden Programmbeispiel sollen die Datenabhängigkeiten mit einem Berechnungsgraphen und Datenabhängigkeitsgraphen anschaulich gemacht werden.

```
o0      DO I = Anfang, Ende
o1              c = a + b
o2              b = b²
o3              d = 5 • c
o4              e = c • d
o5              f = e -1√ABS(e)
o6      END DO
```

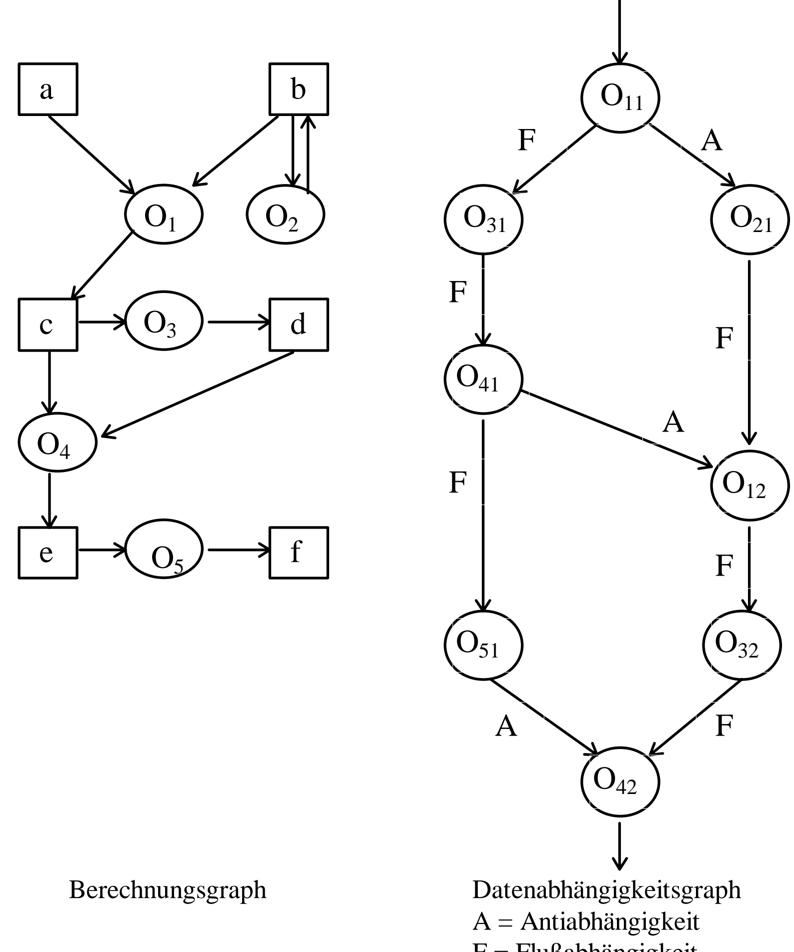
Der *Berechnungsgraph* gibt an, durch welche Operationen O_n welche Speicherzellen verändert werden können.

Der *Datenabhängigkeitsgraph* schreibt die Reihenfolge vor, in der die Operationen aufgrund der bestehenden Datenabhängigkeiten auszuführen sind. Dabei bedeutet die Operation O_{ji} die i-te Ausführung der Operation O_j. Man kann so die schleifenbedingten Abhängigkeiten leicht zeigen.

Am Berechnungsgraphen sieht man, daß O_1 und O_2 genau in dieser Reihenfolge nacheinander ausgeführt werden müssen, weil O_1 die Variable b mit seinem ursprünglichen Wert benötigt, bevor O_2 den Wert von b verändert. Man spricht deshalb von einer *Anti-Datenabhängigkeit* oder *Anti Dependence* oder Write after Read = *WAR Dependence.* Beim Datenabhängigkeitsgraph wird dies deutlich durch den Pfeil mit der Kennung *A* zwischen O_{11} und O_{21}. Weiter sieht man am Berechnungsgraphen, daß die Operation O_3 abgeschlossen sein muß bevor O_4

starten kann, weil O_3 einen Operanden für O_4 liefert. Die nennt man eine *Fluß-Abhängigkeit*, oder Datenfluß-Abhängigkeit, *Flow Dependence* oder Read after Write = *RAW dependence.*

Der Datenabhängigkeitsgraph zeigt an, welche Operationen gleichzeitig ausgeführt werden können: alle Operationen, die auf einer Ebene des Graphen liegen. [GIL93,S.42]



Berechnungsgraph

Datenabhängigkeitsgraph
A = Antiabhängigkeit
F = Flußabhängigkeit

Warum existiert eine Anti-Abhängigkeit zwischen der Operation vier im ersten Durchlauf und der Operation eins im zweiten Durchgang der Schleife ?

Warum existiert eine Datenfluß-Abhängigkeit zwischen der Operation zwei im ersten Durchlauf und der Operation eins im zweiten Durchgang der Schleife ?

Warum existiert eine Anti-Abhängigkeit zwischen der Operation fünf im ersten Durchlauf und der Operation vier im zweiten Durchgang der Schleife ?

Datenabhängigkeiten schränken also die Parallelisierung eines Algorithmus ein. Fassen wir zusammen und ergänzen die Abhängigkeiten um eine weitere, die am Wesen der von Neumann Variablen selbst liegt:

Definition: Benutzung:	X = A + B Y = X	Datenfluß-Abhängigkeit - RAW (eine Variable darf nicht benutzt werden, wenn sie noch nicht definiert ist)
Definition: Benutzung:	Y = X X = A+B	Anti-Datenabhängigkeit - WAR (eine Variable darf noch nicht geändert werden, wenn eine Operation noch den vorherigen Wert verlangt)
Definition: Benutzung:	X = A+B ... X = C+D	Ausgangs-Datenabhängigkeit - WAW (vor dem Wiederbeschreiben einer Variablen muß sicher gestellt werden, daß nicht eine andere Anweisung noch nicht abgeschlossen ist und den vorherigen Wert verlangt.)

Bei WAW Dependence sieht man sehr leicht, daß das an sich so nützliche Prinzip der von-Neumann-Sprachen, Variable als Behälter von Werten immer wieder zu verwenden, nur zu dem Preis zu haben ist, daß die im Algorithmus vorhandenen Möglichkeiten zur Parallelarbeit unnötig, d.h. über die durch die echten Datenflußabhängigkeiten gegebenen Einschränkungen hinaus beschränkt werden. Nun gibt es die Möglichkeit, die Anti Dependence durch Umbenennen von Variablen bzw. die Output Dependence durch Einführung von Zwischengrößen zu beseitigen. Die durch die Verwendung von von-Neumann-Variablen entstehenden Pseudo-Datenabhängigkeiten lassen sich durch das Abgehen von den von-Neumann-Variablen auch ganz vermeiden. Ein Ansatz hierfür sind die *Datenflußarchitekturen*, die in einem gesonderten Kapitel behandelt werden

5.1.1.2 Prozedurale Abhängigkeiten

Die implizite Parallelität eines Algorithmus wird aber nicht nur durch die Datenabhängigkeiten begrenzt, sondern auch durch Iterationen und Fallunterscheidungen. Das bedeutet, daß die Anweisungen, die auf eine Programmverzweigung folgen, erst dann ausgeführt werden können, wenn entschieden ist, wohin verzweigt werden soll.

Die Freiheit von prozeduralen Abhängigkeiten läßt sich nur bei Algorithmen erreichen, die ohne Fallunterscheidung und Iteration darstellbar sind. Es ist genau dies der Vorteil der expliziten Parallelität bei strukturierten Datenmengen, daß keine induktiven Schleifen für die Bearbeitung solcher Datenmengen benötigt werden.

5.1.1.3 Operationale Abhängigkeiten

Die tatsächlich durchführbare Parallelarbeit hängt natürlich auch davon ab, welche Betriebsmittel (Integer Units, Floatingpoint Units, Branch Unit etc.) dafür zur Verfügung stehen. Unter Umständen muß eine ausführbereite Aktivität mangels des dafür vorhandenen Betriebsmittels auf die Ausführung warten. Man spricht in diesem Fall von einer operationalen Abhängigkeit.

5.1.1.4 Realisierung der impliziten Parallelität

Um die gegeben implizite Parallelität eines Algorithmus zu r Parallelarbeit nutzen zu können, muß eine Maschine diese kennen. Grundsätzlich gibt es hierfür drei Möglichkeiten:

- das Programm enthält bereits explizite Anweisungen darüber, welche Aktivitäten parallel ausgeführt werden können;

- der Compiler erkennt die Möglichkeiten zur Parallelarbeit und parallelisiert das Programm entsprechend;
- die Maschine entscheidet selbsttätig aus dem vorgegebenen Datenfluß, welche Operationen sie parallel ausführen kann.

5.1.2 Explizite Datenparallelität

In von-Neumann-Maschinen existieren keine geordneten Datenmengen. Ist die Programmiersprache eine von-Neumann-Sprache, so gibt es auch in ihr keine vollwertigen Datenstrukturtypen. Der übliche Typ **array** zählt hier nicht, da Operationen nie auf das Array, sondern nur auf ein Element des Arrays, ein skalares Datum, ausgeführt werden kann. Es gibt aber auch Programmiersprachen, in denen strukturierte Datenmengen echt als solche behandelt werden können. Musterbeispiele sind die Sprachen *APL* und *FORTRAN90*. In APL gibt es Strukturoperationen auf n-dimensionalen Arrays, innere und äußere Produkte und vieles mehr als elementare Operationen der Sprache. Wesentliche Züge von APL wurden in FORTRAN90 übernommen.

Das folgende Beispiel zeigt die Berechnung des inneren Produkts zweier Matrizen in ALGOL:

```
procedure MMULT(I,J,K, A,B,C);
       value I,J,K;
       integer I,J,K;
       real array A,B,C;
       begin
              integer L,M,N;
              for L := 1 step 1 until K do
                     for  N := 1 step 1 until k do
                            begin
                                   C[L,N] :=  0;
                                  for M := 1 step 1 until J do
                                   C[L,N] := C[L,N] + A[L,M] * B[M,N]
                            end
       end
```

Wieviel einfacher wäre es dagegen, für die als Matrizen deklarierten Variablen A,B und C zu schreiben

```
       C := A*B
```

und damit die Matrizenmultiplikation zum Ausdruck zu bringen. Dieses Thema wird im Kapitel *Datenstruktur-Architekturen* genauer besprochen. Aber hier schon so viel: Es handelt sich dabei um Rechnerarchitekturen, deren Operationsprinzip darin besteht, daß Datenstrukturtypen als elementare Einheiten der Maschine definiert werden und als solche unmittelbar verarbeitet werden können. Besitzt die Programmiersprache dieselben Typen, so führt die Maschine die Operationen dieser Typen unmittelbar parallel aus, und es bedarf dafür keiner Programmstrukturanalyse. Ein Merkmal von Datenstruktur-Architekturen ist die Existenz eines speziellen Strukturprozessors. In der einfachsten Form ist dies ein Adreßgenerator. Die einfachste Form einer Datenstruktur-Architektur ist die *Vektormaschine*.

Parallelarbeit in einer Datenstruktur-Architektur bedeutet, daß die in den einzelnen Schritten der komplexen Rechenoperationen der Strukturdatentypen auftretenden Elementaroperationen gleichzeitig ausgeführt werden. Um dies zu ermöglichen, wird man entweder ein *Array von Rechenelementen* oder aber einen *Pipeline-Prozessor* vorsehen.

5.2 Die Anweisungsebene

Zusammengesetzte Ausdrücke in den Programmen einer höheren Programmiersprache werden bei der Übersetzung durch eine Folge von Maschinenbefehlen, die für elementare Operationen stehen, ersetzt. Gegenüber der höheren Programmiersprache ergeben sich damit längere Befehlsfolgen, wobei die Befehle datenabhängig sind, wenn die Operationen verschiedenen Ebenen des Ausdrucksbaums angehören, und nicht datenabhängig, wenn die Operationen auf der gleichen Ebene des Ausdrucksbaums liegen.

Die in den oben aufgeführten Architekturformen haben gemeinsam, daß sie mehrere Funktionseinheiten zur Ausführung elementarer Operationen besitzen. Sie alle bedürfen zur Leistungsoptimierung in starkem Maße Unterstützung durch den Compiler. Denn der Compiler muß dafür sorgen, daß bereits beim Übersetzen möglichst viele datenunabhängige Befehle erzeugt werden. Dazu führt der Compiler unter anderem folgende Optimierungen durch:

5.2.1 Register-Optimierung

Die Register-Optimierung hat mit einer etwaigen Parallelisierung noch nichts zu tun. Sie wird von jedem guten Compiler unabhängig von den weiteren, spezielleren Anforderungen der gegebenen Architekturform ausgeführt. Vergleichen Sie hierzu auch die allgemeinen Ausführungen im Skript zur Vorlesung *RISC-Architekturen* . Aus den auftretenden, bereits ausgeführten Datenabhängigkeiten ergibt es sich z.B., daß Ausgangswerte einer Operation etwas später als Eingangswerte einer anderen Operation verwendet werden. In diesem Falle ist es zweckmäßig, die Zwischenergebnisse im Registerfile so lange zu halten, wie sie benötigt werden um langsame Hauptspeicherzugriffe zu vermeiden. Diese Registeroptimierung wird vom Compiler in einem eigenen Durchlauf vorgenommen und basiert auf dem Verfahren des *Graphen-Einfärbens*, [Mül91,S.172] das hier nicht weiter beschrieben werden soll.

5.2.2 Schleifen-Aufrollen (loop unrolling)

Unter Schleifen-Aufrollen versteht man eine Programmtransformation des Compilers, bei der mehrere Schleifeniterationen als lineares Programmstück hintereinander geschrieben werden. Der Nutzen dieses Vorgehens liegt darin, daß Operationen eingespart werden können. Dieses Verfahren wird der Einfachheit halber auf der Ebene einer höheren Programmiersprache gezeigt, auch wenn der Compiler es natürlich mit dem etwas unübersichtlicheren und maschinenabhängigen Code zu tun hat. Als Beispiel dient nochmals obiger kleiner, leicht modifizierter Programmausschnitt:

```
DO I = 1, 10
       c = a/I + b/I
       b = I/b^2
       d = I • c
       e = c • d
       f = e - 1√ABS(e)
       print *, c,b,d,e,f
END DO
```

Durch vierfaches Aufrollen der Schleife wird der Schleifenkörper fünfmal aneinandergefügt:

```
Var1    DO I = 1, 10,5
                 c = a/I + b/I
                 b = I/b²
                 d = I • c
                 e = c • d
                 f = e - 1√ABS(e)
                 print *, c,b,d,e,f
                 I = I+1
                 c = a/I + b/I
                 b = I/b²
                 d = I • c
                 e = c • d
                 f = e - 1√ABS(e)
                 I = I+1
                 print *, c,b,d,e,f
                 c = a/I + b/I
                 b = I/b²
                 d = I • c
                 e = c • d
                 f = e - 1√ABS(e)
                 print *, c,b,d,e,f
                 I = I+1
                 c = a/I + b/I
                 b = I/b²
                 d = I • c
                 e = c • d
                 f = e - 1√ABS(e)
                 print *, c,b,d,e,f
                 I = I+1
                 c = a/I + b/I
                 b = I/b²
                 d = I • c
                 e = c • d
                 f = e - 1√ABS(e)
                 print *, c,b,d,e,f
        END DO
```

```
Var2    DO I = 1, 10,5
                 c = a/I + b/I
                 b = I/b²
                 d = I • c
                 e = c • d
                 f = e - 1√ABS(e)
                 print *, c,b,d,e,f
                 I = I+1
                 c = a/I + b/I
                 b = I/b²
                 d1= I • c
                 e1= c • d1
                 f = e1 - 1√ABS(e)
                 I = I+1
                 print *, c,b,d1,e1,f
                 c = a/I + b/I
                 b = I/b²
                 d2 = I • c
                 e2 = c • d2
                 f = e2 - 1√ABS(e)
                 print *, c,b,d2,e2,f
                 I = I+1
                 c = a/I + b/I
                 b = I/b²
                 d3= I • c
                 e3= c • d3
                 f = e3 - 1√ABS(e)
                 print *, c,b,d3,e3,f
                 I = I+1
                 c = a/I + b/I
                 b = I/b²
                 d4= I • c
                 e4= c • d4
                 f = e - 1√ABS(e)
                 print *, c,b,d4,e4,f
        END DO
```

Der Vorteil besteht darin, daß insgesamt weniger Code ausgeführt wird, weil das Testen der Schleifenenbebedingung und das Springen zum Schleifenbeginn nur noch einmal statt neunmal ausgeführt werden müssen. Die Datenabhängigkeiten können vom Compiler leichter aufgelöst werden, wie es in Variante zwei durch Kopien der Variablen d und e zu sehen ist (Im Maschinencode werden dafür zusätzliche Register eingesetzt). Damit läßt sich der Code leichter parallelisieren ! Natürlich gibt es auch Grenzen für das Aufrollen. Es verbietet sich, wenn durch das Aufrollen (und damit der Vergrößerung des Programms) eine Schleife nicht mehr in einen schnellen Zwischenspeicher gelegt werden kann und deshalb zu viele langsame Hauptspeicherzugriffe erfolgen müssen.

Eine weitere Standard-Optimierung des Compilers ist das

5.2.3 Loop Jamming

Dabei werden Schleifen mit gleicher Struktur wenn möglich in einer Struktur zusammengefaßt.

Zum Beispiel wird aus

```
DO N = K, J
       S(N) = S(K+N) * S(J-N)
ENDDO
X = X**2
DO N = K, J
        T(N) = K * J / N
ENDDO
```

folgende Sequenz

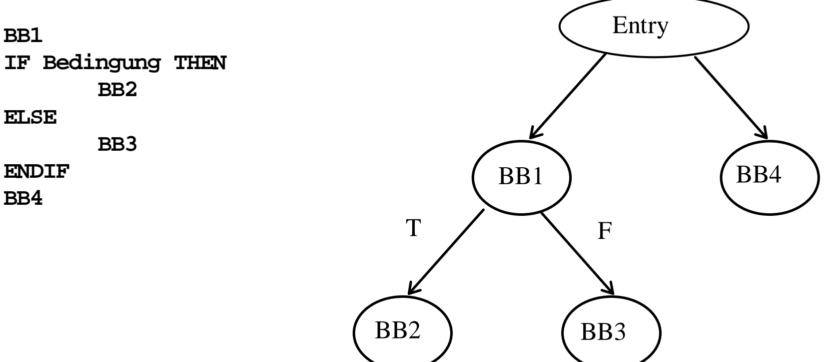
```
DO N = K, J
       S(N) = S(K+N) * S(J-N)
       T(N) = K * J / N
ENDDO
X = X**2
```

Die Vorteile zur Parallelisierung der Anweisungen liegen auf der Hand.

5.2.4 Erkennen der Parallelität auf Anweisungsebene über Basisblöcke hinweg

Wie im Abschnitt *Die Ebenen der Parallelarbeit* bereits angesprochen, ist das Erkennen von parallelisierbaren Instruktionen innerhalb von Basisblöcken relativ einfach, der erreichbare Parallelitätsgrad jedoch ziemlich niedrig. Eine bedeutende Verbesserung des Parallelitätsgrades erreicht man, wenn die Parallelarbeit auch über Verzweigungen hinweg verfolgt wird. Dies erfordert zusätzlich zum Datenabhängigkeitsgraphen auch die Konstruktion eines *Kontrollabhängigkeitsgraphen*. Eine Kontrollabhängigkeit entsteht dadurch, daß die Verzweigungsbedingungen eines Basisblocks in einem anderen Basisblock errechnet wird. Im folgenden Beispiel steht BB für Basisblock:

```
BB1
IF Bedingung THEN
       BB2
ELSE
       BB3
ENDIF
BB4
```



Kontrollabhängigkeitsgraph des Beispiels

Der Basisblock BB_1 wird ebenso wie Basisblock BB_4 ausgeführt, unabhängig davon, ob die Bedingung wahr oder falsch ist. BB_1 kann demnach parallel zu BB_4 ausgeführt werden. Dies gilt aber nur, wenn es zwischen BB_4 und den anderen Basisblöcken keine Datenabhängigkeiten gibt! Der Kontrollabhängigkeitsgraph berücksichtigt aber keine Datenabhängigkeiten. Diese müssen deshalb in einem getrennten Datenabhängigkeitsgraphen ermittelt werden. Die Gesamtheit von Datenabhängigkeitsgraph und Kontrollabhängigkeitsgraph wird auch *Programmgraph* genannt.

Ein weiterer Schritt zur Erhöhung der Parallelität könnte darin bestehen, die Datenabhängigkeitsanalyse über Prozedurgrenzen hinweg durchzuführen. Der notwendige Aufwand für diese *interprozedurale Datenabhängigkeitsanalyse* steht bisher aber in keinem günstigen Verhältnis zum erzielbaren Gewinn an Parallelität.

Auf Grundlage der vom Compiler erkannten Parallelität können so viele Operationen gleichzeitig ausgeführt werden, wie Betriebsmittel zur Verfügung stehen (maschinenabhängig). Da unterschiedliche Operationen unterschiedliche Bearbeitungszeiten (Taktzyklen) benötigen, ergibt sich aus der parallelen Ausführung das *Problem der Synchronisation*, das später noch betrachtet wird.

5.3 Die Prozeßebene

Die über der Anweisungsebene liegende Form der Parallelarbeit ist die Prozeßebene. Darunter versteht man, daß mehrere Prozesse (Programme) getrennt aber nicht unabhängig voneinander verschiedene Aspekte einer Aufgabe erledigen. Daraus ergibt sich sofort die Frage, wie diese Ausführung von Teilaufgaben im Sinne einer geordneten Ausführung der Gesamtaufgabe synchronisiert wird. Bei *nachrichtenorientierten* MIMD-Architekturen mit verteiltem Speicher müssen die Prozesse über sogenannte *messages* kommunizieren. Dabei entsteht das Problem der *Kommunikationslatenz*. Das ist die Zeit, während der ein Rechenknoten nicht arbeiten kann, weil er auf die Ausführung einer Kommunikation wartet. Dies verursacht Effizienzprobleme. Was nützt ein Multiprozessor-System, wenn wenige Prozessoren arbeiten und die meisten nur auf deren Ergebnisse (eine Nachricht) warten, bevor sie weiterarbeiten können. Klar ist, daß die Kommunikationslatenz sinkt, wenn die kooperierenden Prozesse möglichst komplexe Teilaufgaben übernehmen und damit die Kommunikationshäufigkeit und der Kommunkations-Overhead sinkt.

Einen Compiler, der ein Anwendungsprogramm in eine Vielzahl von kommunizierenden Prozessen zerlegt, gibt es zur Zeit noch nicht. Die Realisierbarkeit steht noch in Frage. Nach wie vor ist es Aufgabe des Programmierers, diese Zerlegung durchzuführen und die Kommunikation sowie die Synchronisation zu organisieren. Die *Synchronisationsstrategien* werden in einem gesonderten Abschnitt besprochen.

5.4 Nutzung der Anweisungs-Parallelität

Die Möglichkeiten des Compilers zur Generierung von parallelem Code bei mehreren datenunabhängigen Anweisungen wurden bereits besprochen. Beispiele dafür sind die Compiler für superskalare Prozessoren und VLIW-Maschinen. Diese Art der Parallelarbeit muß somit nicht programmiert werden. Konstrukte (Compiler-Direktiven), die Programmiersprachen wie ALGOL bereits in den sechziger Jahren anboten, damit bei vom Programmierer erkannter Datenunabhängigkeit der Compiler parallelen Code erzeugen konnte

(z.B. PARBEGIN A1;A2;A3 END)

sind nur noch von historischem Interesse und wird hier nicht weiter verfolgt. Aber bei der Nutzung der Schleifen-Parallelität findet man sowohl die explizite Programmierung in einer Sprache, die Anweisungen für die parallele Schleifenausführung hat (z.B. FORTRAN 90) als auch die automatische Erkennung durch den Compiler. Die Compiler für *Vektormaschinen* haben heute immer auch einen automatischen Vektorisierer, der Schleifen in skalaren Programmiersprachen (z.B. FORTRAN 77) erkennen und automatisch für die Verarbeitung aufbereiten kann. An der *Schleifen-Parallelisierung* kann man prinzipielle Unterschiede zwischen verschiedenen parallelen Architekturen deutlich machen.

5.4.1 Schleifen-Parallelisierung

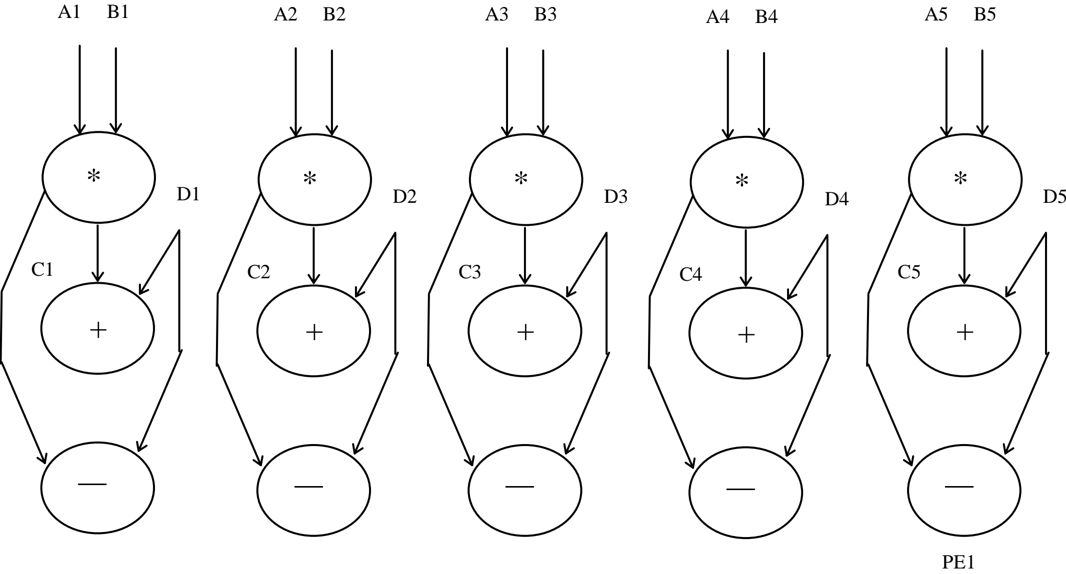
Es gibt zwei Methoden der Schleifen-Parallelisierung:

- Vektorisierung
- Schleifeniteration

Dazu wieder ein einfaches Beispiel:

```
DO I = 1,8
       C(I) = A(I) * B(I)
       E(I) = C(I) + D(I)
       F(I) = C(I) - D(I)
ENDDO
```

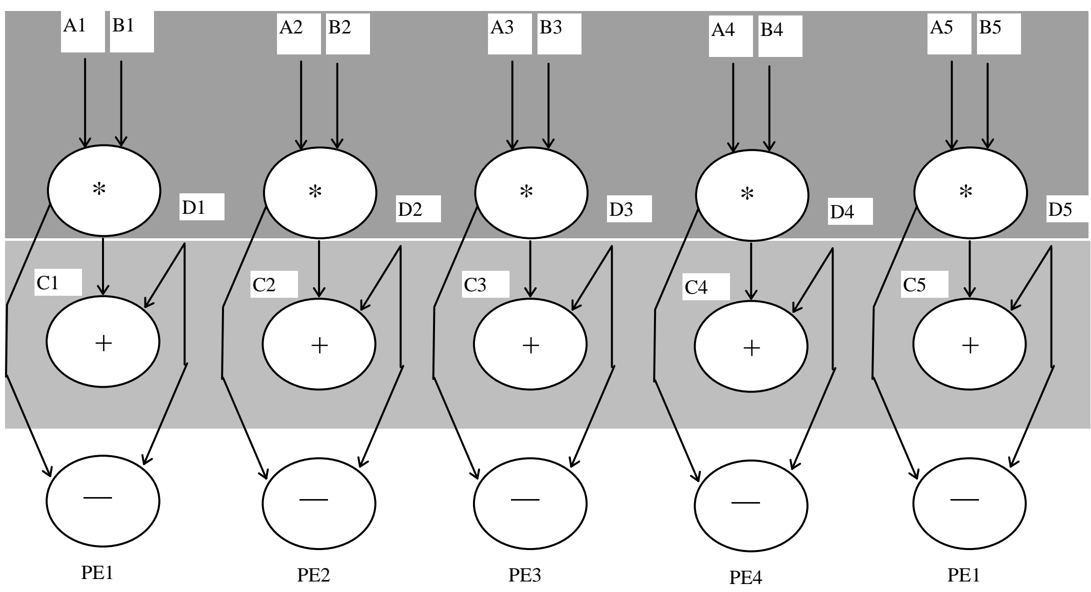
Rollt man diese Schleife auf, ergibt sich folgender Datenflußgraph.



Datenflußgraph nach Aufrollen der Schleife

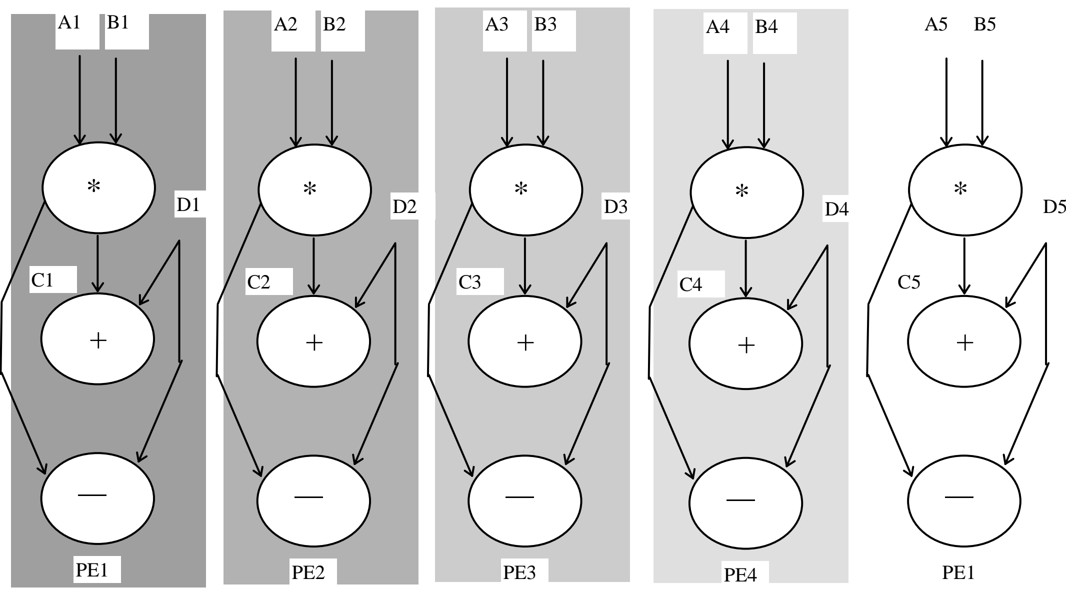
Der Datenflußgraph ist aus Platzgründen nicht vollständig gezeichnet. Er zeigt nur die Operationen für I = 1,5.

Man sieht sehr leicht, daß es verschiedene Möglichkeiten gibt, wie diese Rechenschritte parallel ausgeführt werden können. Entweder horizontal oder vertikal. Zur Veranschaulichung wird der obige Graph mit unterschiedlichen Balken hinterlegt,



horizontale Parallelisierung - Vektorisierung.

Diese Art der Parallelisierung wird von den Compilern der Vektormaschinen angewandt, aber auch von manchen Compilern für superskalare Prozessoren.[7]



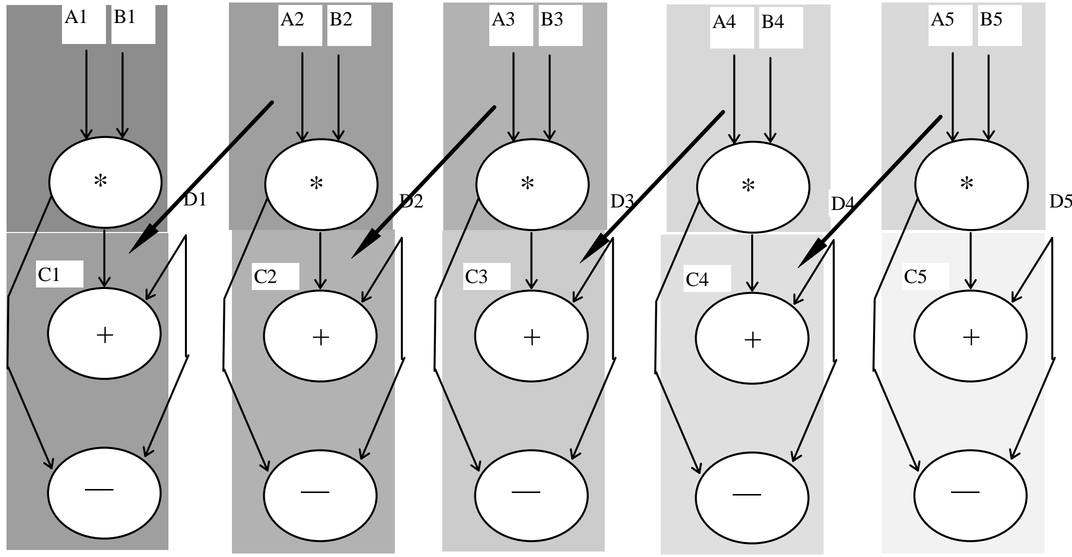
vertikale Parallelisierung - Iteration

Bei diesem Beispiel ist unterstellt, daß vier Prozessoren oder PEs (processing elements) zu Verfügung stehen. Die Prozessoren teilen sich die Arbeit nach unterschiedlichen Iterationschritten auf. Wegen der vier Prozessoren muß die Schleife dann nur zweimal statt achtmal durchlaufen werden. Das geht aber nur deshalb, weil in diesem Beispiel keine Datenabhängigkeiten zwischen den einzelnen Iterationen besteht.

Die letzte Variante, die hier gezeigt werden kann, ist die Arbeitsweise des *Pipeline-Vektorprozessors*

---

[7] Auch wenn die Vektormaschine noch nicht besprochen wurde, ist hier ein Hinweis notwendig: Auf Vektorrechnern sinnvollerweise eingesetzte Programmiersprachen kennen den *Datentyp Vektor*. Der Programmierer formuliert den Algorithmus demnach nicht als Schleife, sondern als drei *Vektoroperationen*. Der Compiler braucht dann diese Transformation nicht mehr zu leisten.. Anders verhält es sich, wenn ein Programm in einer Standard-Programmiersprache unabhängig von der Hardware formuliert wird. Dann entscheidet der Compiler, ob er sequentiellen Code oder wie im Falle einer Vektormaschine vektorisierten Code erzeugt.

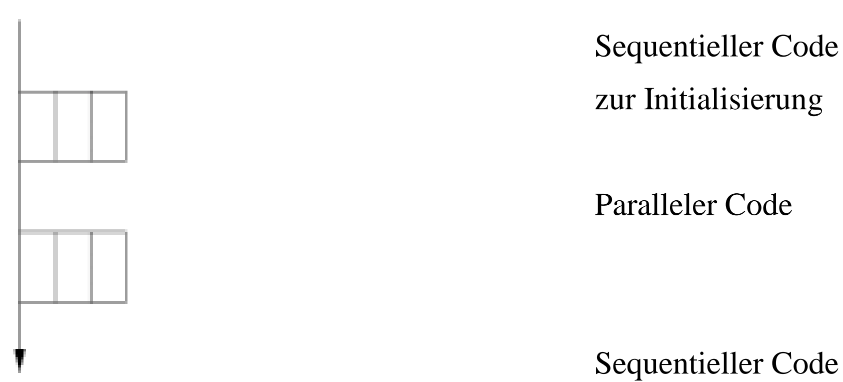


Dieses Verfahren ist dadurch gekennzeichnet, daß ein Multiplizierwerk und 2 Addierer /Subtrahierer überlappend (pipelined) eingesetzt werden (kostengünstig). Mit den verschiedenen Hintergrundfarben und den Pfeilen wird die zeitliche Abfolge und Überlappung gezeigt.: Wird C_1 berechnet, bleibt das Addierwerk ungenutzt, die Pipeline wird gefüllt. Wird C_2 vom Multiplizierwerk berechnet, kann das Addierwerk E_1 und F_1 berechnen usw. Im letzten Arbeitsschritt bleibt das Multiplizierwerk ungenutzt und E_8 und F_8 werden vom Addierwerk berechnet.

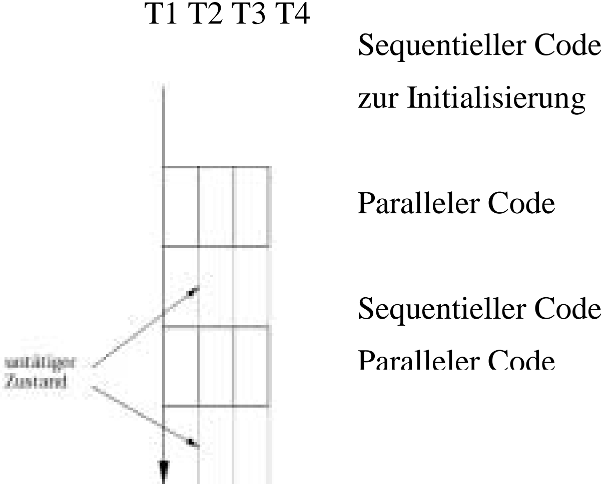
Zeigen Sie, warum das Addierwerk eine Pipeline-Stufe nach dem Multiplizierwerk im obigen Beispiel eingesetzt wird.

5.5 Synchronisation von Parallelarbeit

5.5.1 Fork/Join-Modell nach Pancake:



5.5.2 Reusable-Thread-Pool-Modell nach Pancake:



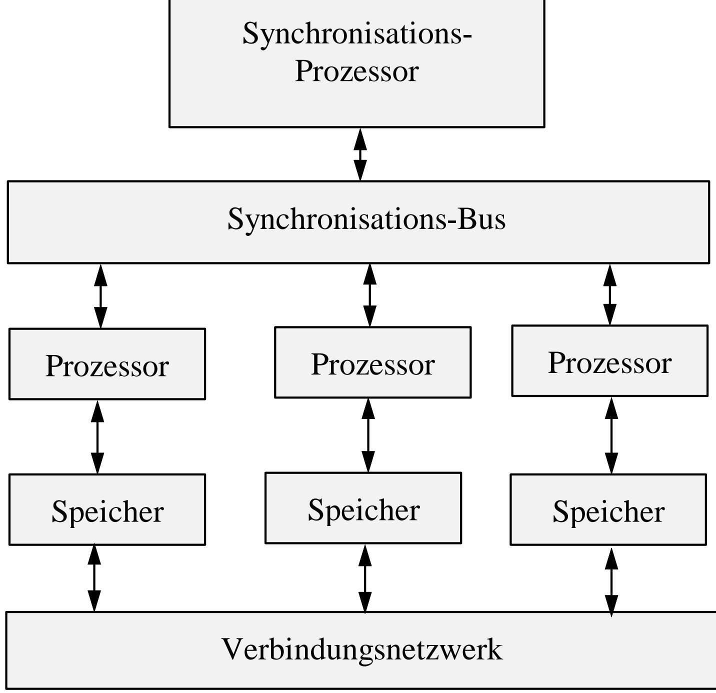
5.5.3 Lock-Step-Betrieb und Barrieren-Synchronisation

Eine sehr einfache und übersichtliche Organisationsform von Parallelarbeit ist *der Lock-Step-Betrieb*. Die Programmausführung wird in zwei alternierende Phasen eingeteilt:

- STEP umfaßt alle Operationen eines Rechenschritts
- LOCK umfaßt den Datenaustausch nach einem Rechenschritt.

Während der STEP-Phase wird nicht kommuniziert und während der LOCK-Phase wird nicht gerechnet. Während der STEP-Phase kann eine Vielzahl von Kontrollfäden (Programmteilen) auszuführen sein. Die Synchronisation erfolgt dabei nach dem Prinzip der *Barrieren-Synchronisation*. Diese synchronisiert eine Vielzahl von Kontrollfäden so, daß beim Erreichen einer Barriere alle Kontrollfäden abgearbeitet sein müssen, bevor das Programm weiter fortschreitet. Dazu führt jeder Kontrollfaden am Ende eine WAIT-Instruktion aus. Kontrollfäden, die zuerst fertig sind, verbleiben solange in dem Wartezustand, bis der zeitlich letzte Kontrollfaden ebenfalls den Wartezustand erreicht hat.

Die Barrieren-Synchronisation kann sowohl durch Software (zählende Semaphoren) realisiert als auch in Hardware implementiert werden. Bei der Hardwarelösung gibt es im System einen besonderen Synchronisationsbus, auf dem die Knoten melden, wann sie den Wartezustand erreicht haben. Diese Meldungen können von dem Synchronisationsprozessor zur Steuerung des LOCK-STEP-Programmablaufs ausgewertet werden.



5.5.4 Synchronisation kooperierender Prozesse

Parallelarbeit in Multiprozessor-Systemen mit verteiltem Speicher spielt sich in der Regel auf der Ebene kooperierender Prozesse ab. Dazu werden geeignete Protokolle (IPC-Protokolle, inter process communication) benötigt.

Zur allgemeinen Betrachtung der Synchronisation ist es günstig, ein *allgemeines Modell der Programmausführung* kooperierender Prozesse zugrunde zu legen. Dieses Modell läßt sich gut als ein Netzwerk von Bearbeitungsknoten oder *Knoten*, die durch Kommunikationskanäle verbunden sind, beschreiben. Für die Synchronisation ist es dann unerheblich, ob das Modell in Form von vernetzten Einprozessor-Systemen, als Multiprozessor-System oder als Multicomputer-System mit Multiprozessoren realisiert wird. Ein Knoten führt Operationen auf Daten aus, die er über seine *Eingabekanäle* empfängt, und produziert Ergebnisse, die er durch *Ausgabekanäle* an andere Knoten weiterleitet. Dabei wird allgemein die Gültigkeit der folgenden Regeln angenommen:

- Die Kommunikationskanäle sind die einzigen Wege, über die die Knoten kommunizieren können.
- Ein Kommunikationskanal überträgt Informationen innerhalb einer endlichen Zeit, im übrigen aber nicht beschränkten Zeit. Diese Zeit heißt *Kommunikationslatenz.*
- Ein Knoten ist zu jeder Zeit entweder mit der Ausführung einer Operation beschäftigt, oder er wartet auf eine Eingabe über einen seiner Eingabekanäle.
- Jeder Knoten verfügt über einen gewissen Speicherbereich.
- Jeder Knoten führt einen bestimmten Prozeß (single processing) oder eine Anzahl von Prozessen (multi processing) aus.

Physikalisch gesehen muß ein Knoten mindestens über einen Speicher (zur Aufnahme eines Programms und seiner Daten) und einen Prozessor (zur Ausführung eines Programms) verfügen. Ein Kommunikationskanal besteht bei speichergekoppelten Systemen in einem den kommunizierenden Knoten gemeinsam zugänglichen Speicherbereich, bei den Architekturen

mit verteiltem Speicher (nachrichtenorientierte Systeme) hingegen tatsächlich auch physikalisch um einen Übertragungskanal.

Folgende Mechanismen für die Kommunikation zwischen Prozessen haben sich etabliert:

- Remote Process Invocation: Ein Prozeß startet einen anderen Prozeß und fährt dann mit seiner Arbeit fort, ohne daß eine weitere Kommunikation mit dem initiierten Prozeß besteht.

  Kommunikationsart: Strenggenommen *keine Kommunikation*

- Remote Procedure Call: Der sendende Prozeß geht nach dem Aussenden einer Nachricht in den Wartezustand, bis der Empfänger den Vollzug des in der Botschaft enthaltenen Auftrags meldet.

  Kommunikationsart: *synchron*

- Rendezvous: Der Senderprozeß stellt an den Empfängerprozeß den Antrag auf ein Rendezvous und geht unmittelbar danach in den Wartezustand, in dem er verbleibt, bis der Empfänger seine Bereitschaft zur Kommunikation durch eine entsprechende Rückantwort gezeigt hat. Der Antrag enthält bereits eine Beschreibung der Objekte, die der Senderprozeß dem Empfänger übermitteln möchte. Dies setzt den Empfängerprozeß in die Lage, zu prüfen, ob er zum Empfang der Nachricht bereit ist. Erst dann meldet er sich zurück. Nachdem auf diese Weise eine Synchronisation zwischen Sender und Empfänger hergestellt worden ist (in ADA Rendezvous genannt), kann die eigentliche Kommunikation zwischen den Prozessen beginnen. Z.B. können Datenobjekte vom Speicherraum des Senders in den Speicherraum des Empfängers kopiert werden.

  Kommunikationsart: *synchron*

- No-Wait-Send: Der sendende Prozeß fährt nach dem Aussenden der Botschaft in seiner Arbeit fort, bis er an die Stelle kommt, wo er ohne Erhalt einer Rückantwort nicht weiterarbeiten kann und gegebenenfalls darauf warten muß.

  Kommunikationsart: *asynchron*

Finden Sie Beispiele für die vier Kommunikationsmodelle.

6 Superskalare Prozessoren und VLIW Maschinen

Der *superskalare Prozessor* und *Very Long Instruction Word Maschine* (VLIW-Maschine) beziehen beide ihre Leistung aus der Parallelarbeit mehrerer Funktionseinheiten. Dabei kann jede Funktionseinheit einen Maschinenbefehl ausführen. Das heißt, beide Prozessoren benutzen die *potentielle Programmparallelität* auf der *Anweisungsebene*. Die Funktionseinheiten sind Datenprozessoren, die alle auf einem gemeinsamen Registerfile arbeiten und für eine bestimmte Aufgabe spezialisiert sind. So gibt es je nach Prozessor z.B. Funktionseinheiten für die Ausführung von

- Ganzzahl-Operationen
- Gleitpunkt-Operationen
- Ausführung von graphischen-Operationen
- Sprunganweisungen
- Lade/Speicher-Operationen

Jeder Funktionseinheiten Typ kann mehrfach vorhanden sein. Caches und Speicherverwaltung (MMU) vervollständigen die Architektur des superskalaren Prozessors, VLIW Maschinen arbeiten in der Regel ohne Caches.

6.1 Superskalare Prozessoren

Superskalare Rechnerarchitekturen kombinieren effizientes Phasenpipeling der Befehlsabarbeitung mit Funktionspipelining in der Ausführungsstufe der Prozessor-Pipeline. Mehrere Funktionseinheiten nutzen die Parallelität des Befehlsstroms. Durch gleichzeitige Ausführung mehrerer voneinander unabhängiger Operationen pro Taktzyklus können sich bei ideal beschaffenem Befehlsstrom CPI-Werte unter 1.0 ergeben. Die Funktionseinheiten sind dabei meist auf bestimmte Befehlstypen spezialisierte und keine vollständigen universellen Rechenwerke. So hat zum Beispiel der Alpha-Prozessor nicht zwei Rechenwerke, sondern eine Funktionseinheit für Integer- und eine für Floating-Point-Arithmetik. Liegen in der Befehlspipeline zwei Integer-Befehle an, können diese also nicht auf zwei Rechenwerke verteilt werden. Der Alpha-Chip ist so gesehen nur bedingt superskalar. Demgegenüber ist der Pentium-Prozessor mit zwei Integer-Einheiten und einer Floating-Point-Einheit ausgestattet und gehört deshalb zu den Superskalar-Architekturen.

Das Superpipelining erfordert durch sein Konzept hohe interne Taktfrequenzen, mit denen das Design schneller als bei einfachen Pipelines an die Grenzen der Technologie stößt. Um dieses zu umgehen und die Taktfrequenz in der Größenordnung wie bei einfachen Pipelines zu halten, sieht die Superskalar-Architektur die parallele Abarbeitung mehrerer Befehle vor.

Im folgenden Bild ist eine Superskalar-Architektur skizziert: Es werden bei jedem Zugriff 4 Befehle eingelesen, die dann soweit wie möglich parallel ausgeführt werden. Die richtige Zuweisung der Befehle an die einzelnen Ausführungseinheiten übernimmt ein Scheduler. Die Berücksichtigung sämtlicher Konsistenzkonflikte wird dabei durch ein Scoreboard unterstützt. Bei Bedarf wird dann wieder ein Satz von Befehlen gelesen, um den Grad der Parallelität so groß wie möglich zu halten.

n	IF	DR	EX_1	MA	WB				2 Befehle parallel
n+1	IF	DR	EX_2	MA	WB				
n+2	IF	DR		EX_2 [1]	MA	WB			
n+3	IF	DR			EX_3 [2]	MA	WB		
n+4			IF	DR	EX_1	MA	WB		2 Befehle parallel
n+5			IF	DR		EX_1 [1]	MA	WB	3 Befehle parallel
n+6			IF	DR		EX_2	MA	WB	
n+7			IF	DR		EX_3	MA	WB	

[1] Resourcenkonflikt

[2] Datenkonsistenzkonflikt

Ablaufdiagramm einer Superskalar-Architektur mit drei Ausführungseinheiten

Die Idee des superskalaren Prozessors ist ebenso aktuell wie altbekannt. Bereits in den 60er Jahren kam eine Maschine auf den Markt, die alle Kriterien eines superskalaren Prozessors erfüllt. Es war dies der CDC 6600 der Firma Control Data, der von Seymour Cray entwickelt wurde. Er enthält 10 Datenprozessoren, die für bestimmte Aufgaben spezialisiert sind. Es gibt Funktionseinheiten für

- die Ausführung der Sprungbefehle
- die Festpunkt-ALU-Operationen
- die logischen Operationen
- den Shift
- die Indexrechnung (zweifach vorhanden)
- die Gleitpunkt - ALU-Operationen (Addition und Subtraktion)
- die Gleitpunkt - und Festpunkt-Multiplikation (zweifach vorhanden)
- die Gleitpunkt - und Festpunkt-Division

Man bezeichnet die 6600 oft als Vorbild für alle nachfolgenden Supercomputer -, RISC- und Superskalar-Architekturen. Für die weiteren exemplarischen Ausführungen wurde aber ein Prozessor ausgewählt, der sowohl aktuell als auch besonders Leistungsfähig ist. Der ALPHA Prozessor mit seinen Varianten (21064, 21066, 21164 etc.) der Firma Digital Equipment gilt seit 1992 immer wieder als der schnellste Mikroprozessor.

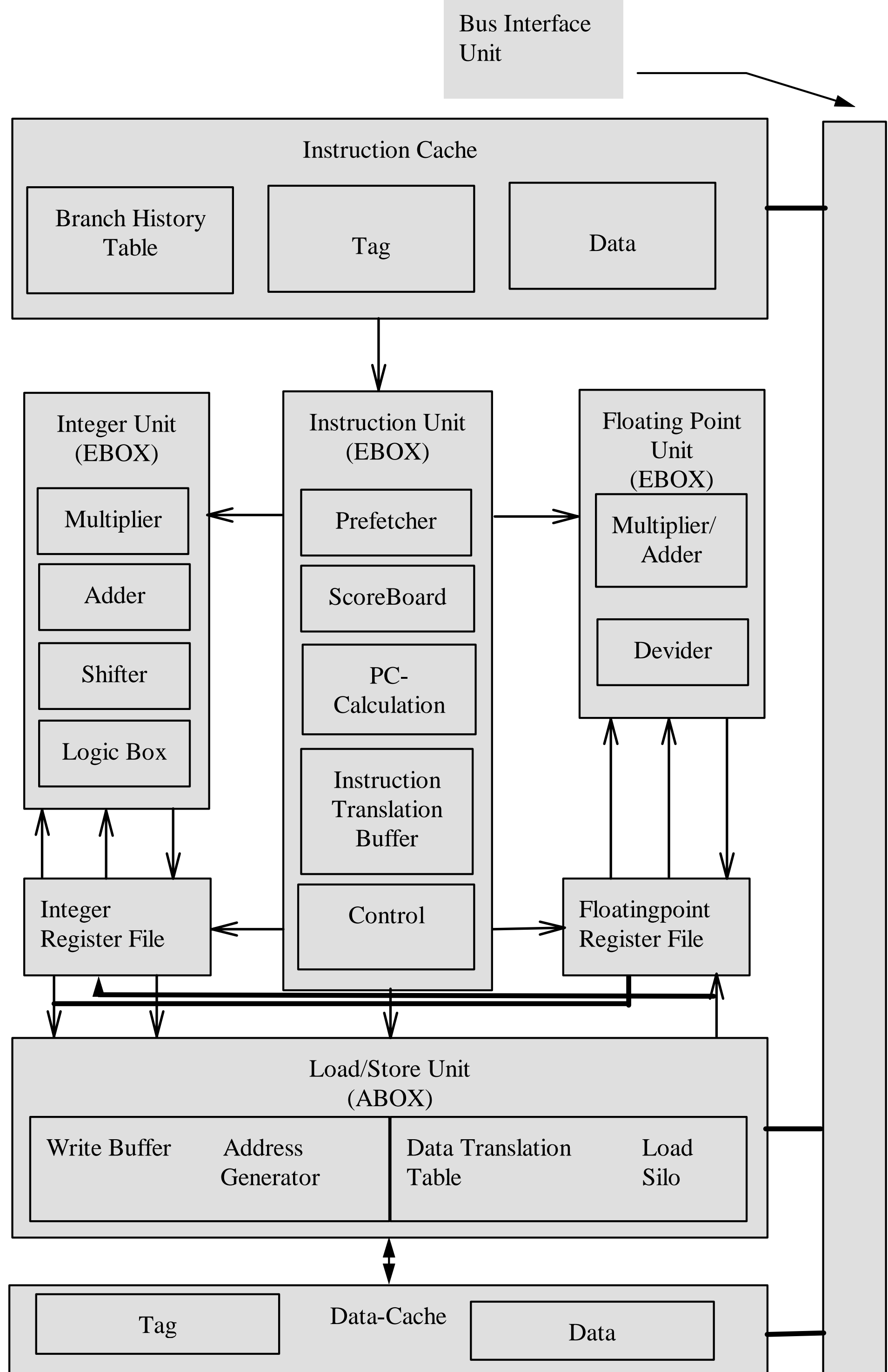
6.1.1 Alpha 21064

Die Darstellung beschränkt sich vor allem auf die superskalaren Eigenschaften des Alpha, eine genauere Betrachtung ist Gegenstand der Vorlesung *RISC-Architekturen*.

Wie dem folgenden Blockschaltbild zu entnehmen ist, besitzt der Prozessor folgende Funktionseinheiten:

- Befehls-Einheit (Instruction Unit - IBOX)
- Integer-Einheit (Integer Unit - EBOX)
- Gleitpunkt-Einheit (Floatingpoint Unit - FBOX)

- Laden/Speichern-Einheit (Load/Store Unit -ABOX)



Blockschaltbild des Alpha-Prozessors

Die Befehlseinheit kann gleichzeitig zwei Befehle ausgeben. Damit könne eine Ganzzahl-Operation mit einer Gleitpunkt Operation kombiniert werden oder eine der arithmetischen Operationen mit einem Sprungbefehl oder Speicherzugriff. Die Superskalarität ist also lange nicht so hoch getrieben wie bei anderen Prozessoren (MC88110,Pentium, Pentium Pro ...) Was den Alpha besonders bemerkenswert mach, ist seine Taktfrequenz von über 200 MHz..

Von besonderer Bedeutung an dieser Stelle ist, daß die Alpha-Architektur die Konfiguration mehrerer Alpha-Chips zu *enggekoppelten Multiprotessor-Systemen* (mit gemeinsamem Speicher) unterstützt. Die atomare Aktualisierung eines Speicherinhaltes, mit der eine Softwaresperre (z.B. Semaphor) für einen kritischen Bereich implementiert werden kann, ist mit einem Alpha-Prozessor folgendermaßen zu erreichen:

- *LDQ_L*: Load Quadword Locked. Laden eines 64-Bit Datenwertes in ein Register Rx des anfordernden Prozessors A. Bei erfolgreichem Laden wird ein Lock-Flag im Prozessor gesetzt. Gleichzeitig wird die physikalische Speicheradresse des geholten Operanden im *Locked_Physical_Address-Register* des Prozessors zwischengespeichert. Wenn ein anderer Prozessor zwischenzeitlich auf die gesperrte Speicheradresse oder deren durch Alignment festgelegte Nachbarschaft zugreift, wird das Lock-Flag von Prozessor A automatisch auf null zurückgesetzt.
- Aktualisierung des Datenwertes im Register Rx durch einen oder mehrere Alpha-Befehle.
- *STQ_C:* Store Quadword Conditional. Nur wenn das Lock-Flag von Prozessor A noch gesetzt ist, wird der Inhalt von Rx an die vom Befehl spezifizierte Speicheradresse zurückgeschrieben. Das Lock-Flag wird zuerst nach Rx kopiert und anschließend auf null zurückgesetzt..
- Ein arithmetischer Vergleich oder ein bedingter Verzweigungsbefehl stellt fest, ob der Inhalt von Rx größer null ist und der atomare Aktualisierungszyklus somit erfolgreich war. Andernfalls muß der Zyklus wiederholt werden.

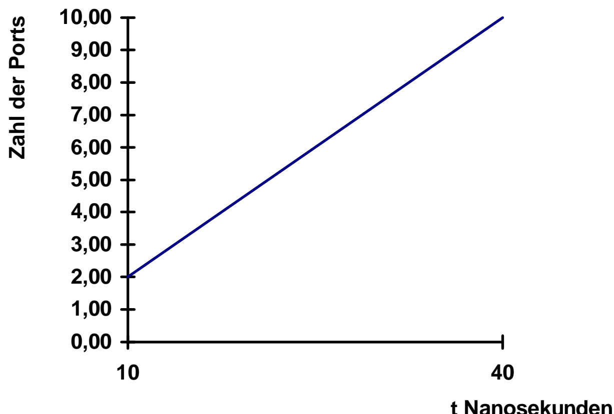
Solange von anderen Prozessoren keine Write-Befehle auf eine gesperrte Adresse durchgeführt werden, können die Aktualisierungen ohne Interaktion mit anderen CPUs in einem privaten CPU-Cache durchgeführt werden. Dadurch sind hohe Multiprozessor-Performanceraten zu erzielen, wenn ein gemeinsamer Speicher verwendet wird, aber nur selten gleichzeitig von unterschiedlichen CPUs auf die gleichen oder benachbarten Datenstrukturen zugegriffen wird. [Mär 94,S.214f]

6.1.2 Grenze der Superskalarität

Moderne Prozessoren haben bereits bis zu acht Funktionseinheiten, die parallel arbeiten können. Wie hoch wird man die Zahl der Funktionseinheiten noch treiben können? Vom Standpunkt der verfügbaren Chipfläche könnte man bei den zukünftigen CMOS-Techniken mit bis zu 10^8 Transistoren pro Chip noch sehr viel mehr Funktionseinheiten unterbringen. Es gibt aber zwei begrenzende Faktoren:

6.1.2.1 Multiport-Registerfile

Die Funktionseinheiten arbeiten parallel auf einem Registerfile, das *pro Funktionseinheit drei Ports*, zwei Lese- und ein Schreibport benötigt. Jedes Bit eines Registers benötigt sechs Transistoren; vier zum Speichern der Information und jeweils eins für den Lese- und Schreibport. Jeder weitere Port führt bei dieser Schaltung zu einer zusätzlich benötigten Chip-Fläche von etwa 25-30%. Da die Länge der Bitleitung um den selben Prozentsatz anwächst, und da die Verzögerungszeit wiederum proportional zur Länge der Bitleitung ist, ergibt sich die im folgenden Bild gezeigte Gesetzmäßigkeit für die Zugriffszeit eines Multiport-Registerfiles als Funktion der Portzahl. Andererseits muß die Zugriffszeit auf den Registerfile genügend klein gehalten werden, um die benötigten Zugriffe in einem Takt zu ermöglichen.



Je mehr Funktionseinheiten eingesetzt werden, desto langsamer wird der Zugriff auf das Registerfile.

Superskalare Prozessoren tragen dieser Tatsache Rechnung, in dem sie zwei Multiport Registerfiles einsetzen. Ein Integer- und ein Floatingpoint-Registerfile. Durch diese Trennung wird die Anzahl Ports gesenkt.

6.1.2.2 Compiler

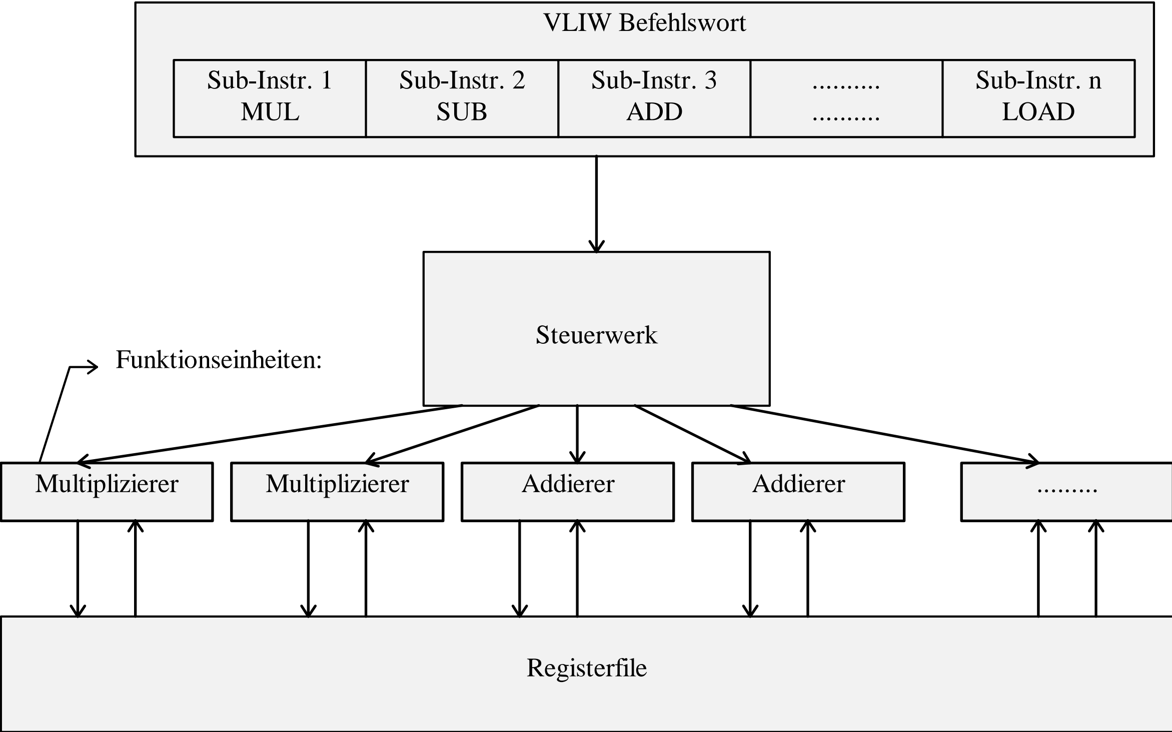
Je mehr Funktionseinheiten eingesetzt werden, desto wirkungsvollere Compiler-Optimierungen sind notwendig, um parallel ausführbaren Code zu erzeugen. Die Parallelisierbarkeit steigt aber nicht proportional mit der Anzahl von Funktionseinheiten.

Ein superskalarer Prozessor ist nur so gut wie sein Compiler.

6.2 VLIW-Maschinen

Die *Very Long Instruction Word* Maschinen ähneln im Wesentlichen den superskalaren Prozessoren. Das Konzept geht jedoch bezüglich der Parallelarbeit einen Schritt weiter: wenn man schon n Funktionseinheiten parallel arbeiten läßt, dann sollte man den Prozessor auch mit n Befehlen gleichzeitig versorgen.

Deshalb gibt es im Unterschied zum superskalaren Prozessor nicht nur einen Befehlsstrom, der im Pipeline-Betrieb abgearbeitet wird, sondern so viele Befehlsströme, wie Funktionseinheiten vorhanden sind. Die Funktionseinheiten arbeiten parallel. Die Befehlsströme werden dadurch erzeugt, daß das Befehlswort der Maschine so viele Operationsfelder umfaßt, wie Funktionseinheiten vorhanden sind. Der Compiler packt jeweils n unabhängige Befehle in ein *Very Long Instruction Word*, ein Befehlswort mit 512 oder mehr Bit. Ein Wort ist in n Felder für n gleichzeitig abzuarbeitende Operationen unterteilt.



Struktur einer VLIW-Maschine

Die Planung der einzelnen Operationen, das sogenannte *scheduling*, erfolgt bereits durch den Compiler. Dazu müssen dem Compiler die Quell- und Zielregister der Daten und die Zeiten für die einzelnen Datentransporte bekannt sein. Es wird vorausgesetzt, daß alle Operationen die gleiche Ausführungszeit haben. Das bedeutet, es wird immer *ein* VLIW zu einem Zeitpunkt bearbeitet (Synchronisation).

6.2.1 Mängel der Architektur

Im Gegensatz zu superskalaren Prozessoren werden bei VLIW-Maschinen die Operanden aus einem allen Funktionseinheiten zugänglichen Registerfile entnommen und die Ergebnisse dort abgelegt. Das *Problem der Multiport-Registerfiles* macht sich deshalb mit größerer Schärfe geltend und ist ein starker leistungsbegrenzender Faktor dieser Architektur (relativ *niedrige Taktfrequenz*).

Im Idealfall werden bei superskalaren wie bei den VLIW-Maschinen n Operationen gleichzeitig ausgeführt, vorausgesetzt, der Compiler kann diesen n-fach parallelen Code erzeugen. Sind Programmsegmente nicht derart parallelisierbar, büßt die VLIW-Maschine drastisch an Performance ein. Im *schlimmsten Fall wäre in einem VLIW nur eine einzige Operation* enthalten. Demgegenüber wird bei superskalaren Prozessoren ein solcher Umstand durch das Phasenpipelining teilweise kompensiert.

Die VLIW-Maschine arbeitet normalerweise *nicht mit Cache-Speichern*. Das liegt daran, daß man beim parallelen Zugriff von n Funktionseinheiten auch n Datencaches benötigen würde, was erhebliche Probleme bezüglich der Cache-Kohärenz zur Folge hätte. VLIW-Maschinen müssen deshalb höhere Anforderungen an das Speichersystem (Hauptspeicher) stellen.

VLIW-Maschinen konnten sich aus diesen Gründen nicht auf dem Markt durchsetzen. Inzwischen gibt es extrem leistungsfähige superskalare Single Chip Prozessoren, aber keine Single Chip VLIW Maschinen.

Was somit von den intensiven Forschungsarbeiten, die zur Entwicklung der VLIW-Maschine führten, geblieben ist, ist ein Durchbruch auf dem Gebiet der Theorie und Praxis des optimalen Befehls-Scheduling durch den Compiler.

Die enormen Fortschritte, die hier bei der Entwicklung immer besserer Compiler für die VLIW-Maschine erzielt worden sind, kommen heute dem superskalaren Prozessor zugute.

7 Vektormaschinen und Anordnungen von Rechenelementen

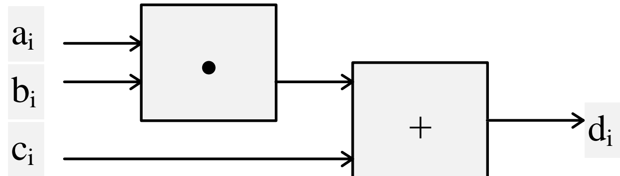
Von *Vektorrechnern* war bisher schon mehrmals die Rede. Im Kapitel *Konzepte der Parallelarbeit*, Abschnitt *Explizite Datenparallelität* wurde gezeigt, daß Vektorrechner den Datentyp Vektor verwenden und damit Operationen auf alle Elemente von Arrays ausführen können.

```
D := A*B+C
```

wobei A,B,C und D Vektoren sind, kann von einer solchen Maschine ausgeführt werden (und eine umständliche Programmierung mit mehreren verschachtelten Schleifen auf alle Feldelemente kann unterbleiben).

Aber wie führt ein Rechner Vektoroperationen aus? Es gibt zwei prinzipiell unterschiedliche Varianten:

Arbeitsweise eines Pipeline-Vektorprozessor



Alle Ausdrücke $a_i \bullet b_i + c_i$ werden nach dem Pipeline-Prinzip berechnet. Das Pipeline-Prinzip besteht darin, daß $a_{i+1} \bullet b_{i+1}$ bereits berechnet wird, wenn $c_i + d_i$ addiert wird. An Hardware benötigt man dabei lediglich *einen Multiplizierer* und *einen Addierer*.

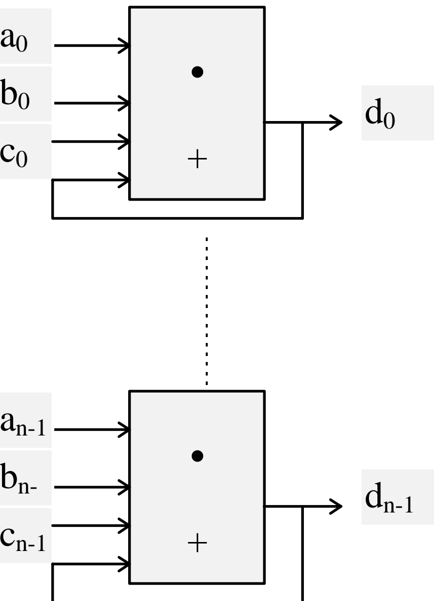
Für die Vektoroperation müssen hier offensichtlich alle Iterationsschritte für i=1... n ausgeführt werden. Die gesamte Vektoroperation benötigt n+1 Taktzyklen .

Worin unterscheidet sich jetzt dieser Vektorprozessor von einem superskalaren Pipeline-Prozessor? An den benötigten Hardwaremitteln kann man keinen entdecken.

Ein Pipeline-Vektorprozessor kennt den Maschinenbefehl `D:= A•B+C`. Bei Ausführung des Befehls wird ein *Mikroprogramm* angestoßen, das die obigen Iterationsschritte ausführt. Das Mikroprogramm wird üblicherweise eher der Hardware als der Software einer Architektur zugeordnet, da es selbst für den Assembler-Programmierer transparent ist.

Auf der Prozessorebene betrachtet, kann ein Pipeline-Vektorprozessor die Vektorelemente nur zeitlich nacheinander berechnen, auf der Softwareebene aber gleichzeitig.

Arbeitsweise eines Rechenelemente-Arrays (RE-Array)



Hat man n Rechenelemente (Rechenwerke, ALUs) kann man im ersten Schritt alle Multiplikationen und in einem zweiten Schritt alle Additionen durchführen. Rechenelemente können die üblichen arithmetischen Operationen ausführen. Zu jedem Zeitpunkt führen sie aber alle dieselbe Operation aus.

Die Vektoroperation benötigt nur zwei Taktzyklen, wenn genügend Rechenelemente vorhanden sind.

Beide Varianten gehören zu der Klasse der SIMD-Rechner, da mit einem Vektorbefehl n Datenelemente modifiziert werden. Für eine effiziente Bearbeitung von Vektoren werden spezielle *Vektorregister* eingesetzt. Damit läßt sich die Zugriffsrate auf den langsamen Hauptspeicher verringern. Anzahl und Tiefe der Vektorregister ist natürlich applikationsabhängig. Die CRAY-1 von 1976 z.B. hatte neben acht Skalar- und acht Adreßregistern noch acht Vektorregister mit je 64 Komponenten.

8 Datenflußarchitekturen

8.1 Überblick

Datenflußarchitekturen werden im Rahmen dieser Vorlesung behandelt, weil sie eine Architektur verkörpern, bei der die Maschine selbst jede Möglichkeit der Parallelarbeit - Datenparallelität wie Programmparallelität - erkennen kann.

Die Entwicklung der Datenflußarchitekturen begann bereits Mitte der sechziger Jahre und wurde immer wieder aufgegriffen. Die Grundidee besteht darin, die Ausführung eines Algorithmus im Compiler nicht in der üblichen Weise durch einen vorgegebenen Kontrollfluß zu steuern, sondern die Maschine selbst zur Laufzeit entscheiden zu lassen, welche Operationen in welchen Zeitpunkten ausgeführt werden können. Die Grundlage hierfür ist der Datenflußgraph, der Operationen des Algorithmus und die zwischen ihnen bestehenden Datenabhängigkeiten angibt. Jeder Knoten des Datenflußgraphen repräsentiert eine Operation. Die Operanden der Operationen werden durch Marken (token) repräsentiert, mit denen die Eingänge des Knotens belegt werden können. Erst wenn jeder Eingang mit einer Marke belegt ist, d.h. wenn alle Operanden vorhanden sind, kann die Operation ausgeführt werden. Man sagt statt dessen auch, daß der Knoten *zünden* kann und nennt diese Bedingung die *Zündregel-Semantik* des Datenflußprinzips.

Das Datenflußprinzip hat in Bezug auf Parallelarbeit den Vorteil der Nähe zum funktionalen Programmierstiel, da Variablen nur einmal beschrieben werden. Dadurch könnte das größtmögliche Maß an Parallelarbeit realisiert werden. Allerdings, um schon vorwegzugreifen, können Datenflußmaschinen dieses hohe Maß unter Umständen nicht handhaben.

Die Zündregel-Semantik in Verbindung mit dem Prinzip der *einmaligen Zuweisung* führt dazu, daß die Maschine jede Möglichkeit der Parallelität erkennen und mit dem Ziel der optimalen Ausnutzung der vorhandenen Betriebsmittel nutzen kann. Diese faszinierende potentielle Eigenschaft hat im Laufe der Zeit immer wieder zu Forschungsprojekten geführt, wie man das Prinzip kosteneffektiv realisieren kann. Allerdings gibt es bis heute keine Produkte dieser Art auf dem Markt.

Man mußte bald erkennen, daß das Prinzip der einmaligen Zuweisung bei der Array-Verarbeitung *extrem speicheraufwendig* und wegen des sehr hohen Aufwands für das Kopieren von Datenobjekten auch *sehr ineffizient* ist. Dieser gravierende Nachteil des reinen Datenflußprinzips wurde dadurch gemildert, daß durch spezielle Mechanismen eine Rückkehr zum Prinzip des *wiederbeschreibbaren Speichers* des von-Neumann-Rechners stattfand, allerdings nicht in völliger Freiheit wie bei diesem, sondern mit durch Hardware überwachten Einschränkungen, die zur Gewährleistung der Programmkorrektheit notwendig ist.

Weiter hatte es sich gezeigt, daß die Datenflußmaschine während der Ausführung eines Programms sehr häufig mit einer sehr großen Zahl von parallel ausführbaren Aktivitäten „überflutet“ wird, die sie alle auszuführen versucht, wegen der begrenzten Zahl der zur Verfügung stehenden Hardware-Betriebsmittel jedoch nicht parallel ausführen kann. Der hohe Grad an Parallelität führt damit nicht zu einer höheren Leistung, sondern nur zu einem starken Anstieg des organisatorischen Aufwands und damit zu einer Leistungsminderung. Daher wird neuerdings durch eine spezielle Hardwareeinrichtung oder schon beim Übersetzen der Programme die Parallelität auf ein handhabbares Maß begrenzt.

Nachdem die reinen Datenflußarchitekturen leistungsmäßig von den RISC Prozessoren völlig überrundet wurden, versuchte man es mit einer Symbiose von Datenflußmechanismen mit dem Operationsprinzip des superskalaren Prozessors. Dabei entstanden sogenannte *hybride Datenflußarchitekturen* und *vielfädige Architekturen*. Die Forschung der vielfädigen Architektur tritt zunehmend an die Stelle der Datenflußarchitektur-Forschung.

8.2 Grundlagen der Datenflußarchitekturen

Im Datenflußrechner wird von der sequentiellen Programmausführung völlig abgegangen. Die Operationen des Programms werden nach dem Datenflußprinzip grundsätzlich in dem Augenblick ausgeführt, indem

- sie mit allen ihren Operandenwerten versehen sind und
- ein Prozessor zu ihrer Ausführung verfügbar ist.

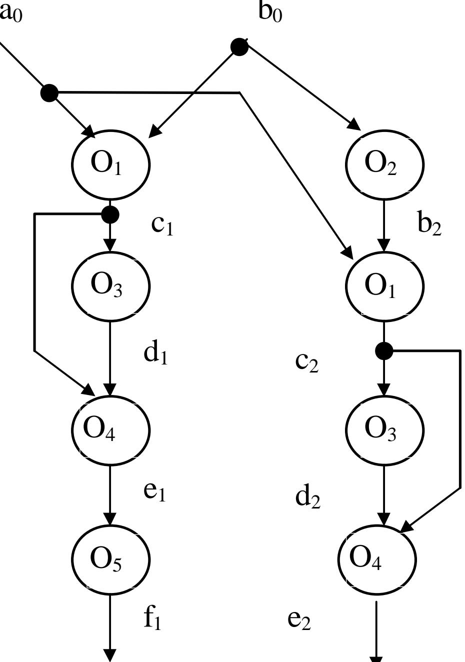
Einen Befehlszähler wie in der von-Neumann-Maschine gibt es damit nicht.

Im Idealfall sind immer so viele ausführbare Aktionen vorhanden, wie Prozessoren zur Verfügung stehen, so daß alle Prozessoren beschäftigt werden können. Durch die Anwendung des Prinzips der einmaligen Zuweisungen wirken sich bei den die Parallelität bestimmenden Datenabhängigkeiten nur die echte *Flußabhängigkeit* aus, (Vergleiche dazu Kapitel *Konzepte der Parallelarbeit*, Abschnitt *Parallelität und die sie einschränkende Faktoren*) weil es die im von-Neumann-Rechner übliche Mehrfachbelegung von Speicherplätzen nicht gibt.

Die Synchronisation zwischen den datenabhängigen Operationen wird von der Maschine selbst, d.h. zur Laufzeit auf Grundlage einer Zündregel-Semantik vorgenommen. Deshalb sagt man auch, daß die Ausführung eines Datenflußprogramms *datengetrieben* ist und nicht, wie sonst allgemein üblich, durch einen Kontrollfluß gesteuert wird.

Das *Datenflußprinzip* ist das *Operationsprinzip einer Rechnerarchitektur*, bei der die Steuerung des Ablaufs eines Algorithmus nicht explizit durch einen Kontrollfluß bestimmt wird, sondern implizit aus dem Datenfluß abgeleitet wird, der mit der Berechnung verbunden ist.

Eine einfache Darstellung des Datenflusses eines Algorithmus bietet der Datenflußgraph:



Beispiel für einen Datenflußgraphen

Die Knoten sind die Operationen, auch *Aktivitäten* oder *Akteure* genannt. Die Verbindung zweier Knoten durch eine Kante bedeutet, daß der den Kanten zugeordnete Wert Ausgabe der Operationsausführung des Quellknotens und Eingabe für die Operationsausführung des Zielknotens ist. Damit geben die Kanten des Datenflußgraphen die *Datenabhängigkeiten* an. Die Tatsache, daß es keine Variablen, sondern nur Werte gibt, läßt sich auch so interpretieren, daß die Kanten Speicherplätzen entsprechen, die während der Laufzeit des Algorithmus nur ein einziges Mal mit einem Wert belegt werden.

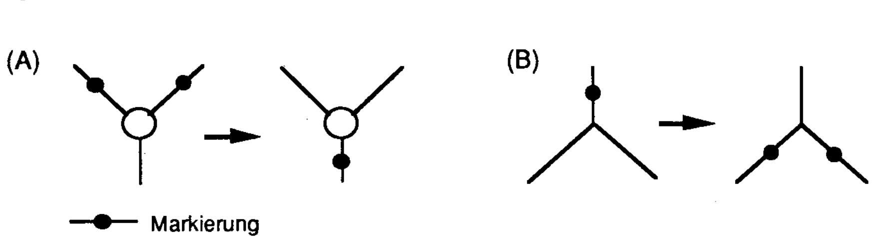
Eine Aktivität kann dann gezündet werden, wenn alle Eingabewerte verfügbar geworden sind. Dies erklärt, warum im Datenflußschema Daten mehr sind als die zu manipulierenden Objekte. Sie sind vielmehr die treibende Kraft bei der Programmausführung (*datengetrieben*). Dies erfordert, daß ein Akteur seine Eingabedaten *konsumieren* muß, damit diese nicht einen weiteren Akteur zünden können. Gleichzeitig produziert die Ausführung des Akteurs neue Werte, die Ausgabewerte, die ihrerseits Eingabewerte nachfolgender Akteure sein können. Damit werden *alte Werte* konsumiert und *neue Werte* erzeugt, aber es werden keine bestehenden Werte verändert. Dies ist eine andere Erklärung der Tatsache, daß es im Datenflußprinzip den üblichen Variablenbegriff nicht gibt. Auf der Hardwareebene bedeutet dies, daß es keine veränderlichen Speicherzustände gibt; der Speicher dient vielmehr dazu, Zwischenresultate aufzunehmen.

☞ Damit kann es auch keine konkurrierenden Datenzugriffe geben, und es entfällt die Notwendigkeit der Synchronisation der Datenzugriffe, wie sie im Multiprozessorbetrieb typisch ist.[GIL93, S.295].

Man hat hiermit eine völlige Abkehr von dem üblichen Programmbegriff als einer Folge von Instruktionen, deren Ausführung den Zustand der Maschine verändert, der wiederum durch die aktuelle Speicherbelegung gegeben ist. Statt dessen besteht ein Datenflußprogramm aus einer Menge von Akteuren, die durch den gemeinsamen Zugriff auf Datenwerte zueinander in Relation stehen, wobei die Datenwerte ihrerseits während der Programmausführung dynamisch generiert und konsumiert werden.

Das Prinzip der einmaligen Zuweisung bringt es mit sich, daß bei mehrfacher Verwendung eines Datenwertes entsprechend *viele Kopien* erzeugt werden. Dem Vorteil, daß damit der Zugriff eines Akteurs auf einen Datenwert völlig vom Zugriff anderer Akteure auf denselben Datenwert entkoppelt ist, ist andererseits mit dem Nachteil verbunden, daß der Speicherbedarf entsprechend vervielfacht wird. Daher gehen praktisch alle bisher realisierten Datenflußmaschinen hier vom reinen Datenflußprinzip ab.

✍ Erstellen Sie den Datenflußgraphen für das Beispiel aus dem *Abschnitt Datenabhängigkeit und Datenabhängigkeitsanalyse*




```
input u, v, w;
x = u + v*w;
y = u — v*w;
output x, y;
```

Bild 9-3 zeigt das zugehörige DFS. Dieses besteht aus den 8 Akteuren S1 bis S8 und den 6 Verbindungsknoten L1 bis L6. Die Akteure Sl, S2, S3 bzw. S7, S8 übernehmen die Eingabe bzw. die Ausgabe.

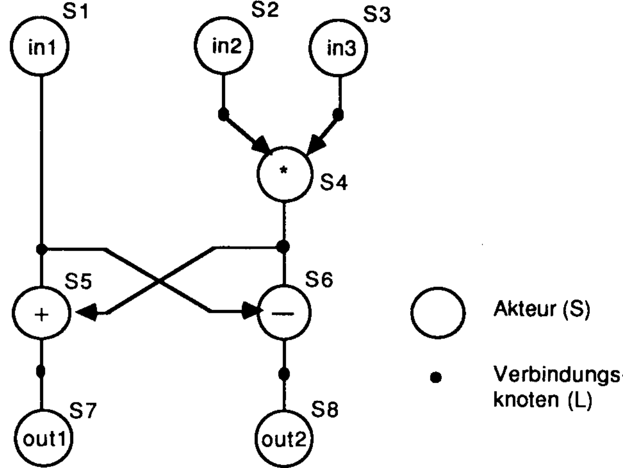


Bild 9-3 Datenflußschema für eine *FFT-Butterfly*-Operation (ohne Markierungen)

9 Grundlagen der MIMD-Parallelrechner

Bei Multiple Instruction Multiple Data-Systemen (MIMD) sind vollständige Prozessorelemente mehrfach vorhanden. Ein Prozessor umfaßt jeweils ein Leitwerk und ein oder mehrere Rechenwerke zuzüglich aller internen Register- und Speicherelemente. Die einzelnen Prozessoren können auch als Pipeline-, Vektor- oder Array-Prozessoren ausgelegt sein. Häufig werden heute superskalare RISC-Prozessoren in MIMD Systemen eingesetzt. MIMD Systeme können mehrere Programme, mehrere Teile eines parallelen Programms oder mehrere Prozesse gleichzeitig bearbeiten.. Diese Flexibilität ist ein entscheidender Vorteil gegenüber SIMD-Systemen. Deshalb besitzen fast alle modernen Parallelrechner eine MIMD-Struktur.

Es gibt viele Möglichkeiten, Prozessoren untereinander und mit dem Hauptspeicher des Systems zu koppeln. Die Topologie ist aber abhängig von der Prozessorzahl und der logischen Organisation des Systems. Auch die Speicherorganisation beeinflußt die Topologie von MIMD-Rechnern. Je nach Struktur, Organisation und zugrundeliegender Verarbeitungsphilosophie (Parallelität auf Befehls,- Aufgaben-, Prozeßebene) müssen unterschiedliche Methoden zur Programmierung, Steuerung und Synchronisation eines MIMD-Systems sowie zur Inter-Prozeß-Kommunikation bereitgestellt werden.

9.1 Die Hauptformen von MIMD-Architekturen

Abgrenzung von anderen Architekturen

Multiple Instruction and Multiple Data (MIMD) Architekturen verfügen über eine Anzahl von Prozessoren, die unabhängig voneinander parallel arbeiten können. Dabei handelt es sich um *Universalprozessoren*, die auch in einem Einprozessorsystem als zentrale Recheneinheit arbeiten könnten. Die Eigenschaft unterscheidet MIMD-Architekturen einerseits von Mehrprozessorsystemen, in denen eine zentrale Recheneinheit von einer Anzahl von *Coprozessoren* zur Durchführung spezieller Aufgaben unterstützt wird (z.B. I/O-Prozessoren, Sound- und Videoprozessoren), und andererseits von den *Anordnungen von Rechenelementen* (RE-Arrays), in denen die REs keine autonomen Prozessoren sind, sondern arithmetisch-logische Funktionseinheiten, die von außen gesteuert werden.

Symmetrische und asymmetrische Architektur

Man nennt bei den MIMD Architekturen einen Prozessor mit seinem lokalen Speicher einen *Rechenknoten* oder kurz Knoten. Man spricht von einer *symmetrischen* MIMD-Architektur, wenn alle Rechenknoten des Systems bezüglich ihrer Rolle im System vergleichbar sind. Üben die Knoten verschiedene, spezialisierte Funktionen aus, so spricht man von einer *asymmetrischen* Architektur.

Master-Slave oder Prinzip der kooperativen Autonomie

MIMD-Architekturen kann man weiter danach unterscheiden, ob die Rechenknoten von einer zentralen Systemaufsicht überwacht werden (*Master-Slave Systeme)* oder ob die Systemaufsicht nach dem Prinzip der *kooperativen Autonomie* im System verteilt ist

Speichergekoppelte Systeme

Systeme mit verteiltem Speicher

MIMD Architekturen lassen sich bezüglich ihrer *physikalischen* Struktur in zwei Hauptkategorien einteilen, die im Bild unten skizziert sind:

- Die *speichergekoppelten Systeme* mit einem zentralen Speicher für alle Knoten (shared memory architecture)
- Die *Systeme mit verteiltem Speicher* bei denen es nur die lokalen Knotenspeicher gibt (distributed memory architecture).

Nachrichtenorientierte Systeme

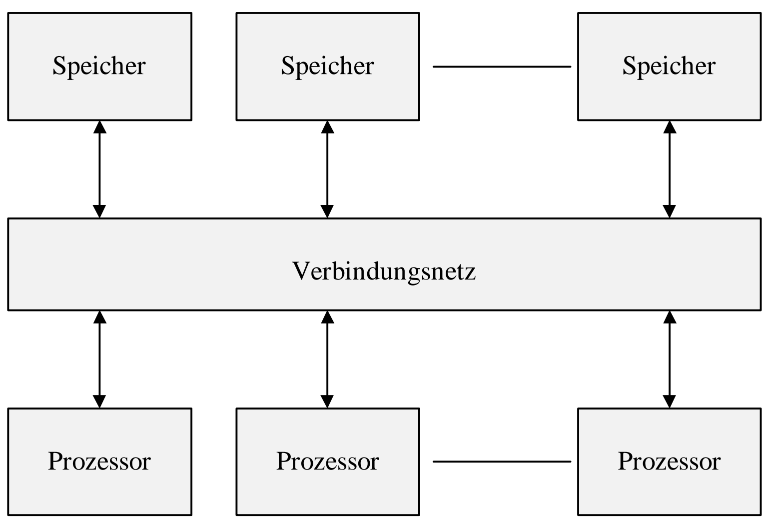
Systemen mit verteiltem gemeinsamen Speicher

Bei den Systemen mit verteiltem Speicher ist zu unterscheiden, ob ein Knotenprozessor nur auf seinen eigenen Knotenspeicher zugreifen kann oder ob auch Zugriff auf die Speicher in den anderen Knoten hat. Im ersten Fall kann die Kommunikation zwischen den Knoten nur durch das Austauschen von Nachrichten geschehen. Man spricht in diesem Fall von *nachrichtenorientierten Systemen.* Im zweiten Fall spricht man *von Systemen mit verteiltem gemeinsamen Speicher.* Bei diesen Architekturen besteht somit der gemeinsame Speicher aus der Gesamtheit aller Knotenspeicher.

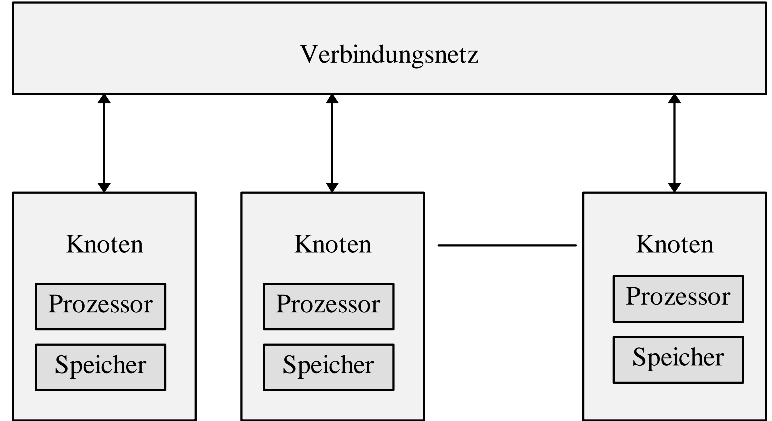
Speicherverschränkung

Gemeinsamer Speicher + lokaler Speicher

Zur Erhöhung der *Zugriffsbandbreite* kann der zentrale Speicher der *speichergekoppelten* Architektur aus einer Anzahl autonomer Speicherbänke bestehen. Die zu bearbeitenden Programme mit ihren Daten befinden sich in dem zentralen Speicher, und die Kommunikation zwischen den Prozessoren findet ebenfalls über diesen statt. Man spricht auch dann von einem System mit gemeinsamem Speicher, wenn die Prozessoren zusätzlich zu dem allen gemeinsamen Speicher noch jeweils einen *lokalen Speicher* haben.



System mit gemeinsamem Speicher



System mit verteiltem Speicher (Nachrichtenorientiertes System)

Aufteilung der Programme in die Speicher

Bei den Systemen mit verteiltem Speicher müssen die auszuführenden Programme mit ihren Daten in geeigneter Weise in Teilprogramme zerlegt werden und dann so über die Speicher verteilt werden, daß jeder Prozessor möglichst denjenigen Teil, den er bearbeiten soll, in seinem lokalen Speicher vorfindet.

Skalierbarkeit des Systems

MIMD-Architekturen unterscheiden sich auch in der *Skalierbarkeit des Systems*. Eine Rechnerarchitektur heißt skalierbar, wenn sie es ermöglicht, mit den gleichen Hardware- und Softwarekomponenten Konfigurationen beliebiger Größe zu erstellen. Eine vollständig skalierbare Architektur hat die Eigenschaft, daß Anwendungsprogramme für sie unabhängig von der aktuellen Konfiguration geschrieben werden können.

Entscheiden Sie an Hand der obigen Schaubilder, welches der MIMD-Systeme besser skalierbar ist: Systeme mit gemeinsamem Speicher oder Systeme mit verteiltem Speicher ?

Problem der Speicherbandbreite

Architekturen mit zentralem gemeinsamen Speicher sind nicht skalierbar. Der Grund liegt darin, daß die Speicherbandbreite mit wachsender Prozessorzahl nicht mit wächst, sondern konstant bleibt. Dadurch wird der gemeinsame Speicher sehr schnell zum *Systemflaschenhals* Diese Gesetzmäßigkeit zwingt dazu, die Zahl der Prozessoren im System hinreichend klein zu halten.

☞ Die Zahl der möglichen Prozessoren ist um so geringer, je höher die Verarbeitungsbandbreite des einzelnen Prozessors ist.

Gegenwärtig liegt die Maximalzahl der Prozessoren bei maximal 30. Natürlich kann ein zentraler Speicher - auch wenn er aus entsprechend vielen Bänken besteht - nur dann diese verhältnismäßig große Zahl von Prozessoren unterstützen, wenn der Speicher durch eine leistungsfähige Cache-Hierarchie unterstützt wird..

Falls in einem Multiprozessorsystem mehrere Prozessoren mit jeweils eigenen Cache-Speichern unabhängig voneinander auf Speicherwörter des Hauptspeichers zugreifen können, entstehen Gültigkeitsprobleme. Mehrere Kopien des gleichen Speicherworts müssen miteinander in Einklang gebracht werden.

Cache-Kohärenz und Speicherkonsistenz

Eine Cache-Speicherverwaltung heißt **(cache-)kohärent,** falls ein Lesezugriff immer den Wert des zeitlich letzten Schreibzugriff auf das entsprechende Speicherwort liefert.

Kohärenz bedeutet das korrekte Voranschreiten des Systemzustands durch ein abgestimmtes Zusammenwirken der Einzelzustände. Im Zusammenhang mit einem Cache-Speicher muß das System dafür sorgen, daß immer die aktuellen und nicht die veralteten Daten gelesen werden.

Ein System ist konsistent, wenn alle Kopien eines Speicherworts im Hauptspeicher und den verschiedenen Cache-Speichern *identisch* sind. Dadurch ist auch die Kohärenz sichergestellt.

Eine Inkonsistenz zwischen Cache-Speicher und Hauptspeicher entsteht dann, wenn ein Speicherwort nur im Cache-Speicher und nicht gleichzeitig im Hauptspeicher verändert wird. Dieses Verfahren nennt man **Rückschreibeverfahren** *(copyback* oder *write-back cache policy)* im Gegensatz zum Durchsehreibeverfahren *(write-through cache policy).*

Um alle Kopien eines Speicherworts immer konsistent zu halten, müßte ein hoher Aufwand getrieben werden, der zu einer Leistungseinbuße führen würde. Man kann nun im begrenzten Umfang die Inkonsistenz der Daten zulassen, wenn man durch ein geeignetes **Cache-Kohärenzprotokoll** dafür sorgt, daß die Cache-Kohärenz gewährleistet ist. Das Protokoll muß sicherstellen, daß immer die aktuellen und nicht die veralteten Daten gelesen werden. Dabei gibt es zwei prinzipielle Ansätze für Cache-Kohärenzprotokolle:

- **Write-update-Protokoll***:* Beim Verändern einer Kopie in einem Cache-Speicher müssen alle Kopien in anderen Cache-Speichem ebenfalls verändert werden, wobei die Aktualisierung auch verzögert (spätestens beim Zugriff) erfolgen kann.

- **Write-invalidate-Protokoll:** Vor dem Verändern einer Kopie in einem CacheSpeicher müssen alle Kopien in anderen Cache-Speichern für „ungültig" erklärt werden.

Üblicherweise wird bei symmetrischen Multiprozessoren ein *Write-invalidate-Cache*Kohärenzprotokoll mit Rückschreibeverfahren an-

gewandt.

In symmetrischen Multiprozessoren, bei denen mehrere Prozessoren mit lokalen Cache-Speichern über einen Systembus an einen gemeinsamen Hauptspeicher angeschlossen sind, verwendet man das sogenannte **Bus-Schnüffeln** *(bus-snooping).* Die Schnüffel-Logik jedes Prozessors hört am Bus die Adressen mit, die die anderen Prozessoren auf den Bus legen. Die Adressen auf dem Bus werden mit den Adressen der Cache-Blöcke im Cache-Speicher verglichen. Bei Übereinstimmung der auf dem Systembus erschnüffelten Adresse mit einer der Adressen der Cache-Blöcke im Cache-Speicher geschieht folgendes:

Im Fall eines erschnüffelten Schreibzugriffs wird der im Cache gespeicherte Cache-Block für „ungültig" erklärt, sofern auf den Cache-Block nur lesend zugegriffen wurde.

Wenn ein Lese- oder Schreibzugriff erschnüffelt wird und die Datenkopie im Cache-Speicher verändert wurde, dann unterbricht die Schnüffel-Logik die Bustransaktion. Die „schnüffelnde" Hardware-Logik übernimmt den Bus und schreibt den betreffenden Cache-Block in den Hauptspeicher. Dann wird die ursprüngliche Bustransaktion erneut durchgeführt. Alternativ könnte auch ein direkter Cache-zuCache-Transfer durchgeführt werden *(Snarfing* genannt, dieser ist jedoch derzeit noch selten implementiert.

Als Cache-Kohärenzprotokoll in Zusammenarbeit mit dem Bus-Schnüffeln hat sich das sogenannte MESI-Protokoll durchgesetzt. Dieses ist als Write-invalidate-Kohärenzprotokoll einzuordnen.

Das MESI-Protokoll ordnet jedem Cache-Block einen der folgenden vier Zustände zu:

Exclusive modijjed: Der Cache-Block wurde durch einen Schreibzugriff geändert und befindet sich ausschließlich in diesem Cache.

Exclusive unmodified: Der Cache-Block wurde für einen Lesezugriff übertragen und befindet sich nur in diesem Cache.

• *Shared unmodifled:* Kopien des Cache-Blocks befinden sich für Lesezugriffe in mehr als einem Cache.

• *Invalid:* Der Cache-Block ist ungültig.

Bei der Erläuterung der Funktionsweise des MESI-Protokolls beginnen wir mit einem Lesezugriff auf ein Datenwort in einem Cache-Block, der sich im Zustand *Jnvalid* befindet. Bei einem solchen Lesezugriff wird ein Cache-Fehlzugriff ausgelöst, und der Cache-Block, der das Datenwort enthält, wird aus dem Hauptspeicher in den Cache-Speicher des lesenden Prozessors übertragen.

Je nachdem, ob der Cache-Block bereits in einem anderen Cache-Speicher steht, muß man folgende Fälle unterscheiden:

• Der Cache-Block stand in keinem Cache-Speicher eines anderen Prozessors: Dann wird ein RME *(Read miss exclusive)* durchgeführt und der übertragene CacheBlock erhält das Attribut *Exciusive unmodifled.* Dieses Attribut bleibt so lange bestehen, wie sich der Cache-

Block in keinem anderen Cache-Speicher befindet und nur Lesezugriffe des einen Prozessors stattfinden (RH, *Read hit).*

Falls beim Lesezugriff erkannt wird, daß der Cache-Block bereits in einem (oder mehreren) anderen Cache-Speichern steht und dort den Zustand *Exclusive unmodified* (bzw. *Shared unmodijied)* besitzt, wird das Attribut auf *Shared unmodified* gesetzt (RMS, *Read miss shared)* und der andere Cache-Speicher ändert sein Attribut ebenfalls auf *Shared unmodifled.* Das geschieht durch die Schnüffelaktion SHR *(Snoop hit on a read).*

Falls beim Lesezugriff ein anderer Cache-Speicher den Cache-Block als *Exclusive modified* besitzt, so erschnüffelt dieser die Adresse auf dem Bus (5 HR, *Snoop hit on a read),* er unterbricht die Bustransaktion, schreibt den Cache-Block in den Hauptspeicher zurück *(Dirtv line copyback)* und setzt in seinem Cache-Speicher den Zustand auf *Shared unmodified.* Danach wird die Leseaktion auf dem Bus wiederholt.

Falls ein Prozessor ein Datenwort in seinen Cache-Speicher schreibt, so kann dies nur im Zustand *Exclusive modified* des betreffenden Cache-Blocks ohne Zusatzaktion geschehen (WH, *Write hit).* Der geänderte Cache-Block wird wegen des Rückschreibeverfahrens nicht sofort in den Hauptspeicher zurückgeschrieben.

Ist der vorn Schreibzugriff betroffene Cache-Block nicht im Zustand *Exelusive modified,* so wird die Adresse des Code-Blocks auf den Bus gegeben. Dabei muß man folgende Fälle unterscheiden:

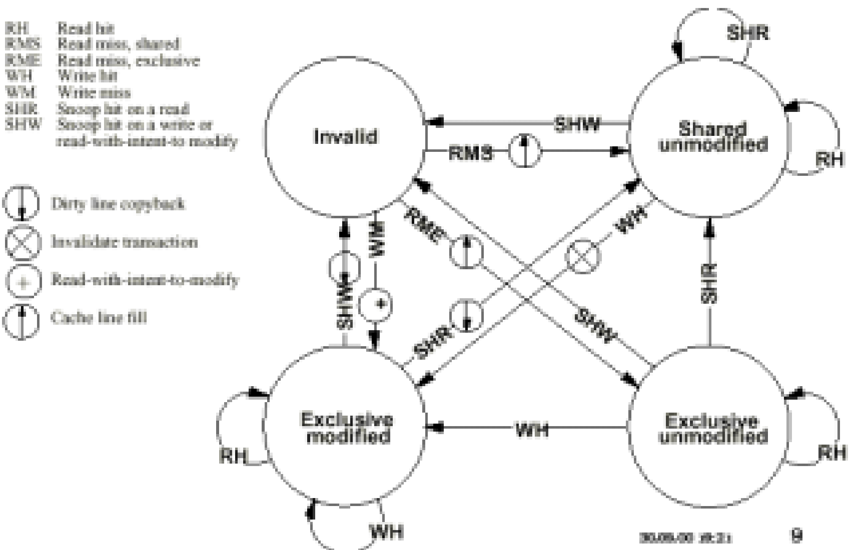
Der Cache-Block war noch nicht im Cache vorhanden (Cache-Fehlzugriff, Zustand *Jnvalid):* Es wird ein *Read-with-intent-to-modtß~* auf den Bus gegeben, alle anderen Cache-Speicher erschnüffeln die Transaktion (SHW, *Snoop hit on a write)* und setzen ihren Cache-Blockzustand auf *Jnvalid,* falls dieser vorher *Shared unmodijfled* oder *Exclusive unmodifled* war. In diesen Cache-Speichern kann nun nicht mehr lesend auf den Cache-Block zugegriffen werden, ohne daß ebenfalls ein Cache-Fehlzugriff ausgelöst wird. Der Cache-Block wird aus dem Hauptspeicher übertragen und erhält das Attribut *Exclusive modified.* War der Cache-Block jedoch in einem anderen Cache-Speicher mit dem Attribut *Exelusive modifled,* so geschieht (wie im obigen Fall eines Lesezugriffs) ein Unterbrechen der Bustransaktion und Rückschreiben des Cache-Blocks in den Hauptspeicher.

Der Cache-Block ist im Cache-Speicher vorhanden, aber im Zustand *Exclusive unmodified:* Hier genügt eine Änderung des Attributs auf *Exclusive modifled.*

• Der Cache-Block ist im Cache-Speicher vorhanden, aber im Zustand *Shared Unmodifled:* Hier müssen zunächst wieder über eine *Invalidate-Transaktion* der oder die anderen betroffenen Cache-Speicher benachrichtigt werden, daß ein Schreibzugriff durchgeführt wird, so daß diese ihre Kopien des Cache-Blocks invalidieren können.

Ein Cache-Block wird erst dann in den Hauptspeicher zurückge-

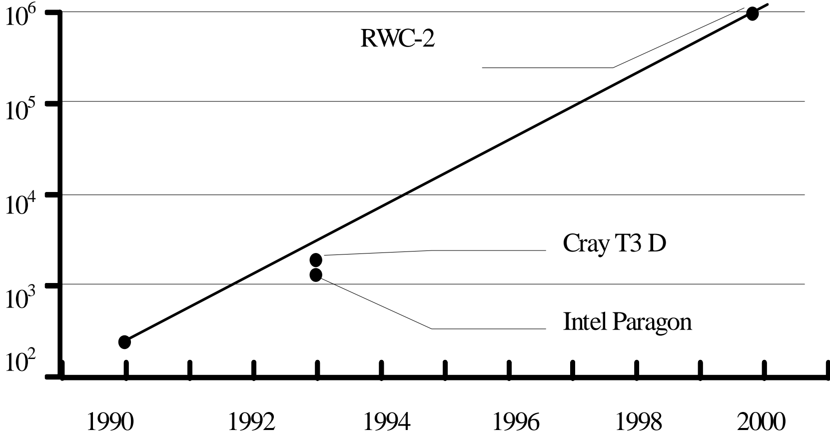
schrieben, wenn er gemäß der Verdrängungsstrategie ersetzt wird oder einer der anderen Prozessoren darauf zugreifen will. Weitere Beschreibungen des komplexen Protokollablaufs geben, [Gi193] oder [Märt94].



Massiv-parallele Systeme

Systeme mit verteilten Knotenspeicher können hingegen fast jede beliebige Größe annehmen, da bei ihnen das Verhältnis von *Verarbeitungsbandbreite* des Prozessors zur *Zugriffsbandbreite* des Knotenspeichers konstant ist. Ab einer gewissen Anzahl von Knoten spricht man von *massiv-parallelen-Systemen*, wobei die Grenze aber nicht definiert ist, sondern eine Funktion der zur Verfügung stehenden Technik höchstintegrierter Schaltungen ist. Das folgende Bild gibt eine Vorhersage für das Anwachsen der maximalen Knotenzahl in massiv-parallelen Systemen bis zum Jahr 2000.

Entwicklung der Knotenanzahl



Diese Vorhersage gründet sich auf das japanische *Real World Computing Program (RWC).*

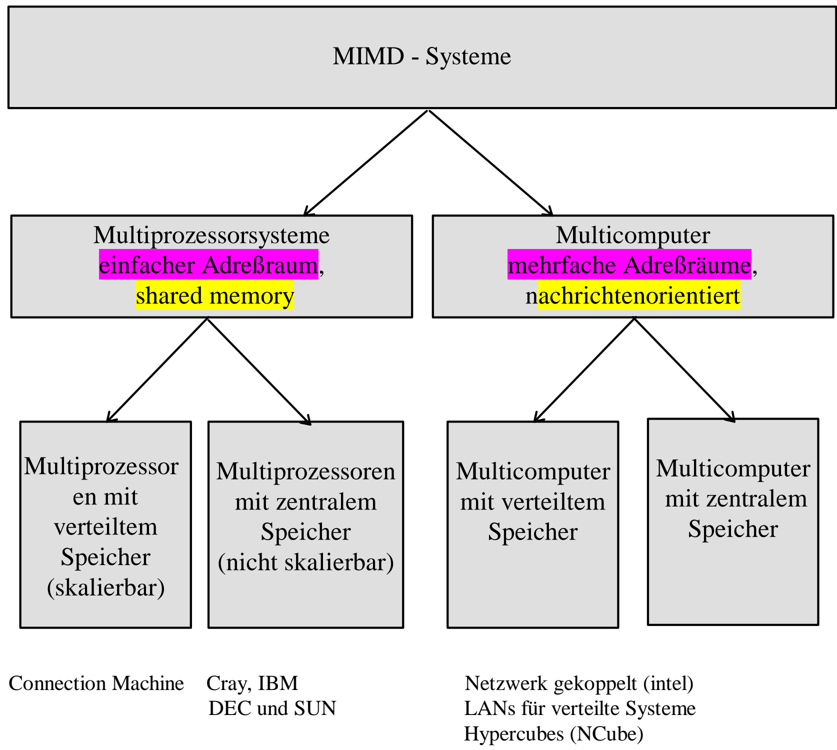
Vorteil des globalen Adreßraums versus Skalierbarkeit.

Systeme mit zentralem Speicher sind nicht wie die Systeme mit verteiltem Speicher skalierbar, dafür bieten sie aber den Vorteil des globalen Adreßraums und damit die Möglichkeit, im *herkömmlichen Programmierstiel* und in den üblichen Programmiersprachen pro-

grammieren zu können. Konventionelle Sprachen wie Fortran, Lisp, Modula 2, C usw. setzen einen globalen Adreßraum voraus, der bei den Systemen mit gemeinsamem Speicher von Natur aus vorhanden ist. Dagegen fordern Systeme mit verteiltem Speicher vom Benutzer oder Compiler die Partitionierung der Anwendungsprogramme in eine Vielzahl kooperierender Prozesse, wobei die Programmierung explizit die Kommunikation zwischen den Prozessen und damit deren Synchronisation einschließen muß. Die obengenannten Sprachen bieten dafür keine Mechanismen an. Für das Programmiermodell der nachrichtenorientierten Systeme benötigt man daher im allgemeinen Falle entsprechende Erweiterungen der verwendeten Programmiersprache.

Tera-Taxonomie

Das folgende Klassifikationsschema für MIMD-Systeme veranschaulicht teilweise die obigen Ausführungen. [Mär 94] Dabei werden Systeme mit gemeinsamem Speicher als eigentliche *Multiprozessorsysteme* und nachrichtenorientierte Systeme mit verteiltem Speicher als *Multicomputer* bezeichnet.



9.2 Steigerung der Effizienz von Multiprozessorsystemen mit gemeinsamem Speicher: Snoopy Logic und Mesi-Protokoll

Schranke der Effizienz	**Speicherbandbreite -> max Knotenzahl < 10 wenn keine zusätzlichen Maβnahmen ergriffen werden**
Maβnahmen	• Speicherzugriffe minimieren -> je kleiner die Programme, desto besser (CISC Philosophie) Jedoch: Aktuelle RISC Prozessoren benötigen bis zu 4 mal mehr Speicherzugriffe (Befehle) weil die Programme rund 4 mal länger sind. • Speicher auf mehrere Speicherbänke verteilen • Speicherzugriffe überlappt ausführen (pipelined) • Burst Mode (benachbarte Daten des angeforderten Datums mitliefern • Kombination von allem
Architekturmaβnahme **lokale Caches**	Der gemeinsame Hauptspeicher wird ergänzt mit lokalen Knotenspeichern (Annäherung an System mit verteiltem Speicher) jedoch der gemeinsame Adressraum bleibt erhalten und die lokalen Speicher dienen 'nur' als Cache-Speicher. Lokale Speicher bilden somit keine Speichererweiterng, sondern präsentieren einen Ausschnitt des Hauptspeichers. Je weniger Überschneidungen im Datenbereich der einzelnen Knoten existieren, desto weniger bildet die Speicherbandbreite eine Schranke für die Effizienz -> Systeme mit gemeinsamem Speicher können bis zu 32 Knoten effizient eingesetzt werden.
Problem **Speicherkonsistenz**	Bei Überschneidungen der Datenbereiche (mehrere Knoten greifen auf dieselben Speicherzellen zu und halten sie in ihren lokalen Caches) muss sichergestellt sein, dass jeder Knoten das zuletzt geschriebene Datum einer Speicherzelle liest. Das bedeuted: Ein Datum verändert in einem Cache führt zum update aller Caches und des Hauptspeichers. Das ist strenge Speicherkonsistenz.(Folie 2) Das bedeutet hohen technischen Aufwand, der sich nicht lohnt, wenn ein verändertes Datum nie mehr von einem anderen Knoten verwendet wird -> strenge Speicherkonsistenz ist nicht unbedingt nötig
Effizienter: **Cache-Koheräenz Protokoll**	Protokoll sorgt dafür, dass immer die aktuellen, nicht veraltete Daten verwendet werden.
Write Update Protokoll	Änderung eines Datum -> Änderung aller Kopien, wenn auch verzögert
Write Invalidate Protokoll	Änderung eines Datum -> alle Kopien gelten als ungültig

9.3 Programmiermodell und abstrakte Maschine

Programmiermodell

Ein Programmiermodell legt fest:

- welche Datentypen dem Programmierer zur Verfügung stehen
- wie ein Programm in Programmeinheiten strukturiert wird
- wie diese Einheiten miteinander Daten austauschen
- wie die Ablaufkontrolle eines Programms festzulegen ist..

Speziell bei den Parallelrechnern sind diese Festlegungen noch weiter zu konkretisieren in Hinsicht auf folgende Feststellungen:

- welche Art von Parallelität genutzt werden soll (Programm- oder Datenparallelität)
- welche Programmiereinheiten parallel ausgeführt werden sollen (Granularität)
- wie die Kommunikation zwischen den parallelen Programmeinheiten organisiert werden soll (Kommunikationsprotokolle)
- wie die Korrektheit des parallelen Ablaufs gewährleistet wird. (Koordination)

Als Vehikel für die Spezifikation eines Programmiermodells kann die Festlegung einer abstrakten Maschine dienen, die wie folgt definiert wird:

abstrakte Maschine

Eine abstrakte Maschine zeigt die wesentlichen Eigenschaften und Mechanismen eines Programmiermodells auf, ohne auf Realisierungsdetails einzugehen. Insbesondere abstrahiert sie den Kontrollfluß, die Operationen und die Daten von der realen Maschine.

- **Kontrollfluß**

Die Abstraktion des Kontrollflusses erlaubt es dem Programmierer, die Folge von Aktionen anzugeben, die zur Ausführung eines Lösungsalgorithmus notwendig sind, ohne jeden einzelnen Schritt des physikalischen Rechners vorgeben zu müssen.

- **Operationen**

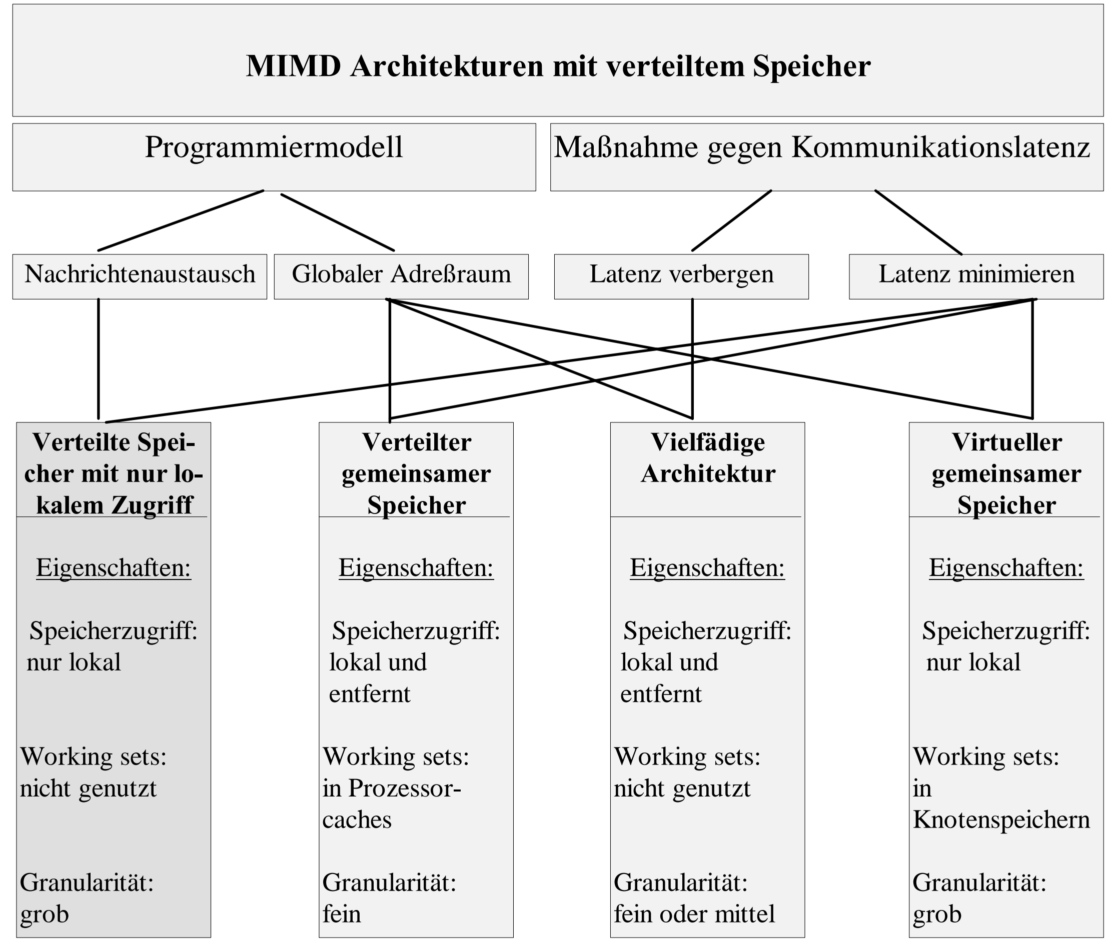
Die Abstraktion der Operation gibt dem Programmierer die Operatoren an die Hand, die zur Ausführung des Lösungsalgorithmus benötigt werden und die wesentlich mächtiger als die realen Maschinenbefehle sein können.

- **Datenstrukturen**

Die Abstraktion der Daten gibt ihm die Sicht der Datenstrukturen, die er für seinen Lösungsalgorithmus benötigt.

Die abstrakte Maschine muß nicht ein isomorphes (umkehrbar eindeutiges) Abbild der physikalischen Maschine sein. Insbesondere kann man auf physikalische Architekturen mit verteiltem Speicher eine abstrakte Maschine mit gemeinsamem Speicher, d.h. mit zentralem Adreßraum, softwaremäßig implementieren und architektonisch unterstützen Dies vereinfacht die Programmierung, da jetzt die Kommunikation zwischen den verteilten Programmeinheiten wie bei der Architektur mit gemeinsamem Speicher über gemeinsam genutzte Variablen erfolgen kann. Hierfür gibt es drei Architekturformen, die im folgenden Bild mit ihren wichtigsten Eigenschaften zusammenge-

stellt sind und mit der nachrichtenorientierten Architektur verglichen werden.



Drei Architekturen, die eine abstrakte speichergekoppelte Maschine verwirklichen, im Vergleich zur vollständig nachrichtenorientierten Architektur (erste Position, dunkel schattiert)[GMD 3/93]

Vergleich

Die erste Lösung erfordert eine spezielle Prozessorarchitektur, was dem Hersteller die Wahlmöglichkeit bezüglich des Knotenprozessors nimmt. Die zweite Lösung ist nur beschränkt skalierbar (bis zu 256 Knoten); sie ist daher ungeeignet zur Realisierung der zukünftigen massiv-parallelen Supercomputer mit zehntausenden und mehr Knoten. Nur die dritte Lösung mit virtuellem gemeinsamem Speicher bietet eine unbegrenzte Skalierbarkeit. Ihre unbefriedigende Leistung stand bisher der Verbreitung dieser Architekturform entgegen.

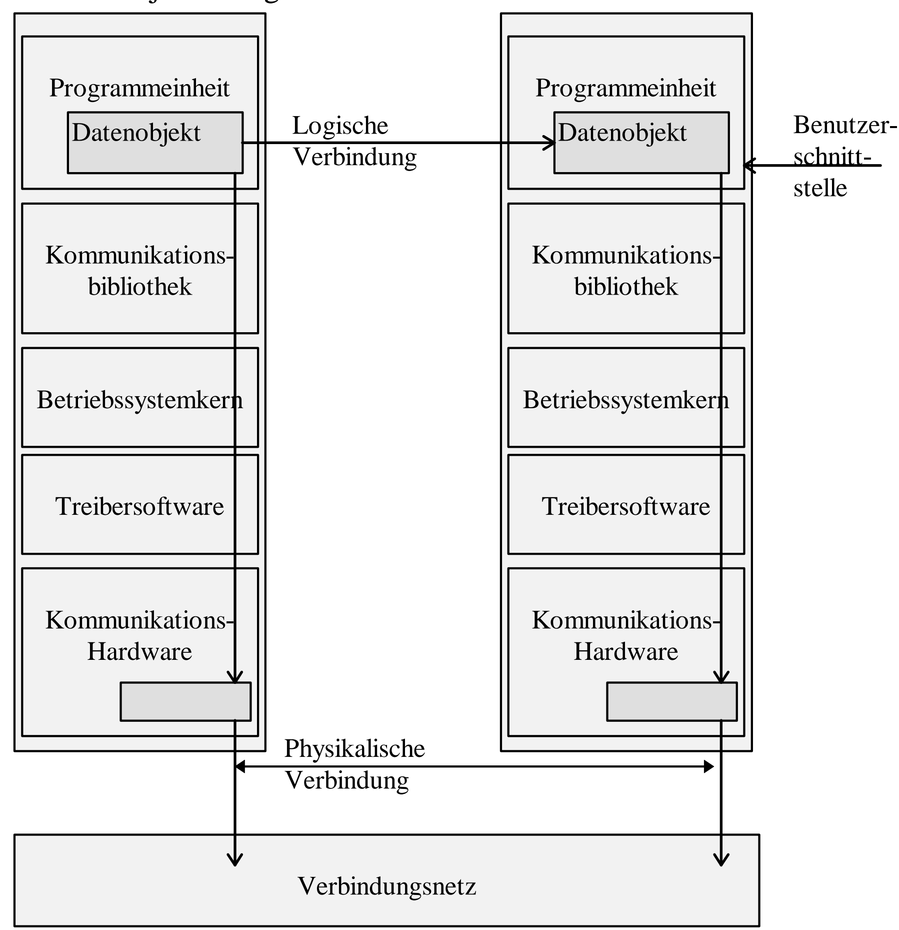
MANNA

Der in dreijähriger Entwicklungszeit beim GMD-Institut für Rechnerarchitektur und Softwaretechnik entstandene MANNA-Rechner ist der Prototyp einer leistungsfähigen *Architektur mit virtuellem gemeinsamem Speicher.* Auf dieses Projekt wird später noch genauer eingegangen.

9.4 Das Kommunikationssystem

Das Kommunikationsprotokoll

Die Kommunikation zwischen den Knoten einer MIMD-Architektur erfordert mehr als nur die Hardware-Einrichtung, über die die Kommunikation stattfindet, d.h. das Verbindungsnetz. Eine ordnungsgemäße Kommunikation muß nach festen Regeln stattfinden, welche die korrekte Durchführung gewährleisten. Diese Regeln nenn man das *Kommunikationsprotokoll*. Da eine Kommunikation in der Regel mit der Synchronisation des Ablaufs parallel arbeitender Programmeinheiten verbunden ist, ist ein Kommunikation nur dann korrekt, wenn sie zu einer korrekten Synchronisation dieser Abläufe führt. Das folgende Bild zeigt die Stufen, über die das Kommunikationssystem für eine MIMD-Architektur mit verteiltem Speicher realisiert wird. Dabei soll eine Verbindung hergestellt werden, über die Datenobjekte ausgetauscht werden können.



Die Stufen des Kommunikationssystems

An der Durchführung der Kommunikation können also folgende Instanzen beteiligt sein:

- **Kommunikations-bibliothek** eine Kommunikationsbibliothek, die mit Hilfe vorgefertigter Kommunikationsroutinen für eine sinnfällige, auf einer entsprechend hohen Abstraktionsebene realisierten Benutzerschnittstelle sorgt.
- **Betriebssystem-kern** der Betriebssystemkern, der für die höheren Ebenen des Kommunikationsprotokolls (die Inter-Prozeß-Kommunikation) zuständig ist

- **Treibersoftware** die Treibersoftware, welche die niedrigen (hardwarenahen) Routinen des Kommunikationsprotokolls ausführt
- **Verbindungsnetz** das Verbindungsnetz, über das die Nachricht mit Hilfe der Weginformation zum Empfängerknoten geleitet wird..

Latenzzeit

Die gesamte Zeit, die zwischen der Einleitung eines Kommunikationsvorganges und dessen Abschluß vergeht, nennt man *Latenzzeit* oder *Kommunikationslatenz.* Wie im Kapitel *Konzepte der Parallelarbeit - Die Prozeßebene* bereits ausgeführt, verursacht die Latenzzeit bei MIMD-Systemen Effizienzprobleme. Die im obigen Schaubild zu sehende Mehrstufigkeit verdeutlicht das Problem, da jede Software- und Hardwareebene einen Beitrag zur Erhöhung der Latenzzeit liefert. Deshalb dazu ein paar tiefergehende Betrachtungen

9.5 Kommunikationslatenz

$T_K = T_S + T_Ü$

Die Latenzzeit der Kommunikation besteht aus zwei additiven Komponenten:

- Startzeit der Kommunikation T_S und
- Übertragungszeit $T_Ü$

Übertragungszeit

$T_Ü = T_V + L_N / R_T$

Die *Übertragungszeit* der Nachricht wird bestimmt durch die

- Zeit für den Wegaufbau (routing) im Verbindungsnetz T_V
- Nachrichtenlänge (in Byte) L_N
- Übertragungsrate (transfer rate) in MByte/s R_T

Für eine möglichst kurze Übertragungszeit gilt demnach:

- Minimierung der Zeit für den Wegaufbau
- Maximale Übertragungsrate

Die Zeit für den Wegaufbau hängt wesentlich von der *Übertragungsstrategie* ab, auf de später noch eingegangen wird. Bezüglich der Übertragungsrate sind heute etwa 50 bis 100 MBytes/s für die einzelne Verbindung (link) ein guter technischer Stand.. Zum Vergleich: Die Übertragungsrate bei *Ethernet* beträgt gut 1 MByte/s.

Startzeit

Für die Startzeit der Kommunikation gibt es keine so einfache Formel wie für die Übertragungszeit. Die Startzeit hängt wesentlich davon ab,

- wer die Kommunikation durchführt und
- welche Anforderungen an das Kommunikationsprotokoll gestellt werden.

gesicherte Kommunikation

positive acknowledgement

Prüfsummeninfor-

In nachrichtenorientierten Systemen besteht die „klassische“ Lösung darin, daß der Betriebssystemkern für die Kommunikation zwischen den kommunizierenden Prozessen zuständig ist. Eine Hauptanforderung an das Kommunikationsprotokoll ist in der Regel die Forderung nach einer *gesicherten Kommunikation* . Diese Absicherung kann

mation

durch eine Rückantwort des Empfängers geschehen, durch welcher dieser anzeigt, daß er die Nachricht ordnungsgemäß erhalten hat. Dazu wird die Nachricht so *kodiert,* daß ein Übertragungsfehler erkannt werden kann, z.B. durch eine zusätzliche Prüfsummen-Information. Noch weiter geht das Rendezvous-Protokoll (siehe Konzepte der Parallelarbeit - Synchronisation kooperierender Prozesse) bei dem der Sender erst dann sendet, nachdem ein Rendezvous mit dem Empfänger hergestellt wurde. Danach wird sichergestellt, daß der Empfänger auch wirklich bereit ist, die Nachricht entgegenzunehmen.

Multi-Tasking-Beriebssystem - Umgebungswechsel ➔ dreifache Startzeit

Die Mechanismen für eine gesicherte Kommunikation sind also verhältnismäßig zeitaufwendig. Werden sie durch den Betriebssystemkern durchgeführt, dann spielt es weiterhin eine entscheidende Rolle, ob es sich dabei um ein Multi-Tasking-Betriebssystem handelt. In diesem Falle erfordert jeder Kommunikationsvorgang auch noch einen Umgebungswechsel: Jede Kommunikationsanforderung (send,receive) führt zu einem *Trap* in die entsprechende Routine des Betriebssystemkerns. Messungen haben gezeigt, daß die damit verbundenen Umgebungswechsel gegenüber einer Kommunikation ohne Umgebungswechsel die Startzeit etwa verdreifachen

paralleles Betriebssystem

Um unter diesen Bedingungen die Startzeit so klein wie möglich zu halten, ist ein *paralleles Betriebssystem* erforderlich. Damit lassen sich bei Verwendung eines superskalaren Prozessors mit hoher MIPS-Rate als optimale Werte erzielen:

$T_S = 70 ... 300\ \mu s$

Der Wert am unteren Ende gilt dabei für Single-Tasking, und der Wert am oberen Ende gilt für Multi-Tasking. Verwendet man ein konventionelles Mehrbenutzer-Betriebssystem wie UNIX, so kann dieser Wert im Bereich von bis zu einigen 10 Millisekunden betragen. Ähnlich ungünstige Verhältnisse ergeben sich bei Betriebssystemen, welche zwar als verteilte Betriebssysteme konzipiert wurden, aber immer noch die Mechanismen von UNIX benutzen, wie z.B. MACH.

UNIX und MACH

Der Flaschenhals Latenzzeit

Man sieht aus den genannten Zahlen, daß die Latenzzeit der Kommunikation in nachrichtenorientierten Systemen zum gravierenden Flaschenhals werden kann. Aber auch bei Systemen mit gemeinsamem Speicher stellt sich das Latenzzeit-Problem, da hier in der Regel der Zugriff zum zentralen Speicher ein Mehrfaches der Zugriffszeit zum lokalen Speicher beträgt. Diesem Nachteil kann man durch geeignete Cache-Speicher in den Knoten entgegenwirken.

Das Startzeit-Problem ist deshalb so gravierend, da es zu erheblichen Verlustzeiten führen kann, d.h. Zeiten, während derer ein Knoten nicht arbeitet, da er auf die Kommunikation wartet. Generell gibt es zwei mögliche Wege zur Überwindung dieses Problems:

Latenz Minimieren

Durch geeignete Organisation der Kommunikation, geeignete Granularität der zu übertragenden Objekte und die geeignete architektonische Unterstützung Hardwareaufwand) werden die Verlustzeiten auf tragbare Werte begrenzt.

Latenz Verbergen

Durch die geeignete Organisation der Parallelarbeit werden die un-

vermeidlichen Verlustzeiten durch sinnvolle Arbeit überbrückt. Dazu muß man den Knoten in die Lage versetzen, während des Kommunikationsvorganges eine andere Aufgabe, die nicht auf diese Kommunikation wartet, zu bearbeiten.

MANNA

Das Manna-Projekt zeichnet sich unter anderem dadurch aus, daß es beide Wege beschreitet. Durch den Einsatz von zwei Prozessoren auf einem Knoten, wobei ein Prozessor nur für die Kommunikation eingesetzt wird, kann die Latenz zum großen Teil verborgen werden.

Latenzminimierung bei Systemen mit gemeinsamem Speicher

Speziell in Architekturen mit verteiltem gemeinsamem Speicher kann man folgende Maßnahmen zur Latenzminimierung treffen

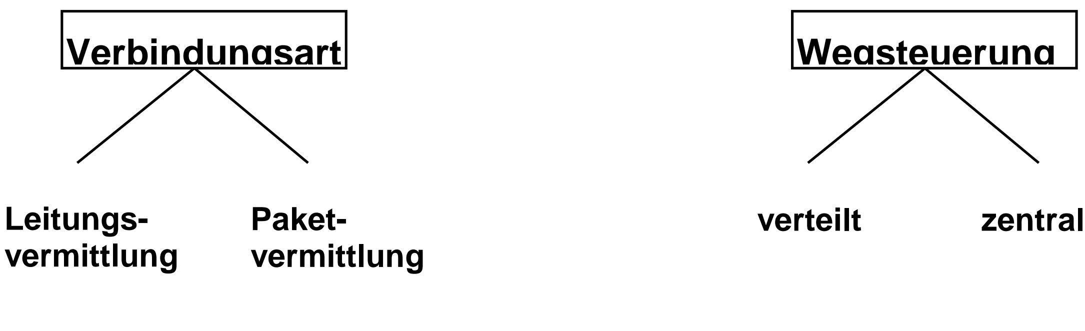
- jeder Prozessor des Systems wird mit einem zusätzlichen, hinreichend großen Cache ausgerüstet
- Verwendung schwacher Konsistenzmodelle, bei denen zugunsten der Leistung auf die strenge (sequentielle) Konsistenz der Speicherzugriffe verzichtet wird.
- Objekte werden bereits vorsorglich von anderen Knoten geholt, bevor sie gebraucht werden (*prefetch*) . Dies ist eine reine Software-Lösung, die nicht weiter betrachtet werden soll.

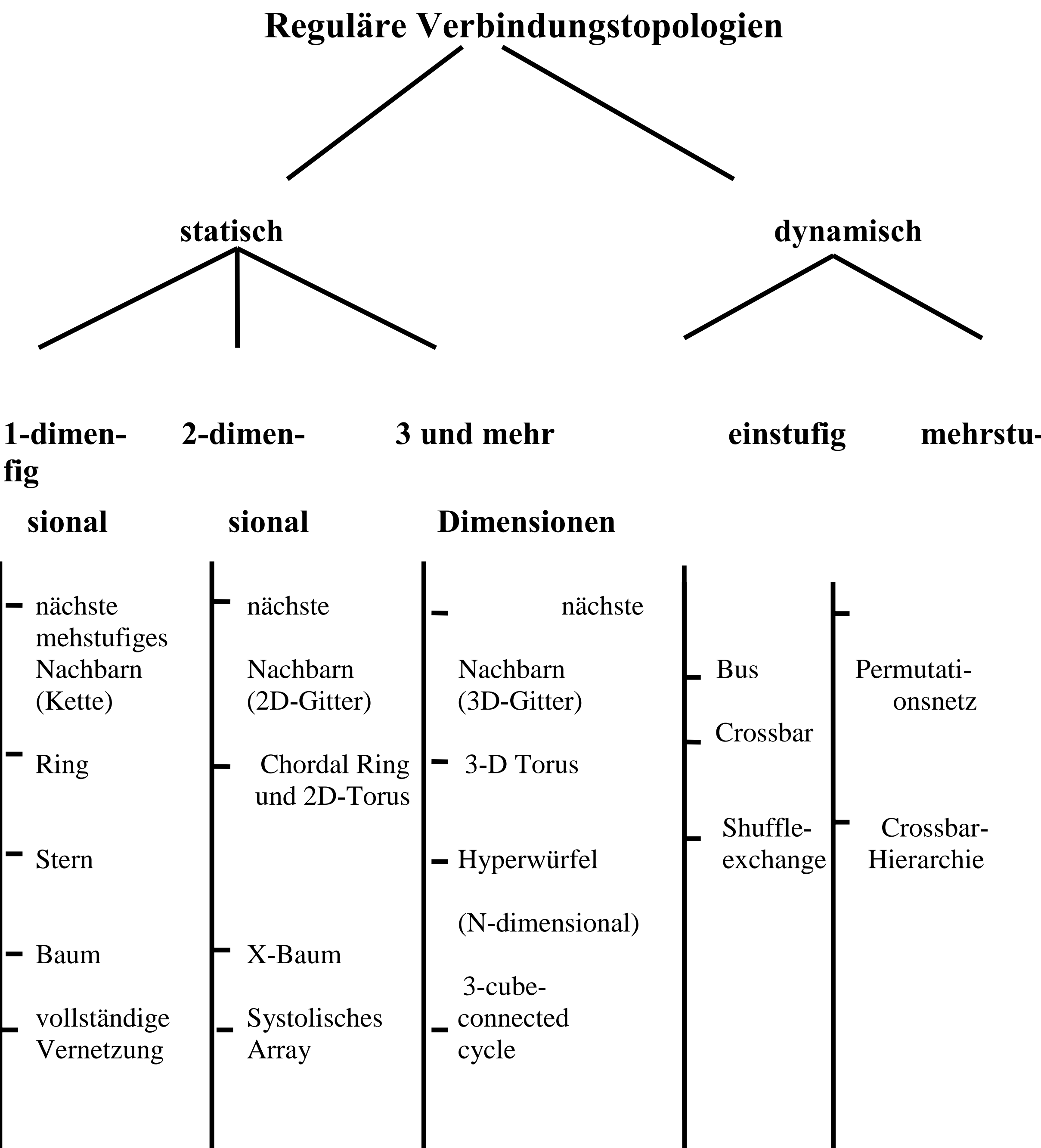
10 Verbindungsnetz-Topologien

Was versteht man unter dem Begriff „Verbindungsnetz-Topologie“?

Unter der Verbindungsnetz-Topologie versteht man die Art und Weise, mit der die einzelnen Komponenten in einem MIMD-System miteinander verbunden sind. Diese einzelnen Bestandteile müssen so miteinander verbunden sein, so daß sich die einzelnen Komponenten nicht gegenseitig behindern und die Datenkonsistenz gewährleistet ist. Dabei muß auch darauf geachtet werden, das in einem solchen System verschieden schnelle Komponenten verwendet werden können.

Verbindungsnetze kann man zunächst nach vier Hauptmerkmalen unterscheiden:





Hauptmerkmale von Verbindungsnetzen

10.1 Verbindungsart

Es sind zwei Verbindungsarten möglich: **Leitungsvermittlung** und **Paketvermittlung**.

Bei der Leitungsvermittlung muß eine feste Verbindung zwischen Sender und Empfänger für die gesamte Übertragungsdauer bestehen. Die Verbindung ist in der Regel exklusiv für die Dauer der Übertragung, d.h. eine Verbindung kann ausschließlich von den Komponenten verwendet werden, die stets gleichzeitig die gleichen Daten erhalten, genutzt werden.

Bei der Paketvermittlung können alle Komponenten an der gleichen Leitung angeschlossen sein. In dieser Leitung werden die Daten in Paketen gesendet, welche mit einer Art Adresse versehen sind und somit der Empfänger sich selbst seine Daten heraussucht.

10.2 Wegsteuerung

Bei der Wegsteuerung unterscheidet man zwischen **verteilt** und **zentral.** Bei einem Netz mit Paketvermittlung muß der Weg vom Sender bis zum Empfänger bekannt sein.

Bei der zentralen Wegsteuerung sind alle möglichen Wege dem System bekannt, d.h. das Routing wird von der Hardware übernommen und ein Datenpaket muß lediglich die Information über den Zielort mitbringen.

Bei der verteilten Wegsteuerung muß jedes Datenpaket den Weg zum Empfänger im Kopf (Header) vermerkt haben. Dem Absender muß also der Weg zum Empfänger bekannt sein.

10.3 Arbeitsweise

Unter **synchroner** Arbeitsweise versteht man, daß alle Daten zu einem festgelegten Zeitpunkt und vor allem gleichzeitig abgesendet werden - sozusagen in einem bestimmten Takt. Die synchrone Arbeitsweise ist im Prinzip einfacher zu realisieren. Bei sehr großen Systemen kann es jedoch vorkommen, daß die Laufzeiten der einzelnen Pakete unterschiedlich lang sind und somit eine synchrone Arbeitsweise nicht mehr möglich ist.

Im Gegensatz dazu kann bei **asynchroner** Arbeitsweise jeder Sender zu jedem Zeitpunkt eine Übertragungsanforderung an das Netz stellen. Auf diese Weise können zum einen verschieden schnelle Bausteine miteinander verwendet werden. Es muß jedoch gewährleistet sein, daß vor einem langsameren Baustein, der von einem schnelleren mit Daten versorgt wird, kein Datenstau entsteht.

Bei Systemen mit asynchroner Arbeitsweise kommt nur die dezentrale Wegsteuerung in Frage.

10.4 Art der Topologie

Irreguläre Topologien sind im wesentlichen nur von theoretischem Interesse, auf sie soll hier nicht eingegangen werden.

10.4.1 Realisierungskriterien

Wegen der großen Vielfalt von regulären Netztopologien gibt es hier eine weitere Klassifizierung aufgrund zusätzlicher Realisierungskriterien und Topologiemerkmale.

- statische oder dynamische Verbindungsnetze
- Dimensionalität bei statischen Verbindungsnetzen
- Stufenzahl bei dynamischen Verbindungsnetzen

Statisches Verbindungsnetz besteht aus festen Punkt-zu-Punkt-Leitungsverbindungen.

Schalteinrichtungen nur am Angang und am Ende

Dynamisches Verbindungsnetz enthält eine Vielzahl von Schaltknoten, die durch die Wegsteuerung so gesetzt werden, daß ein bestimmter Verbindungsweg entsteht.

	Dynamische Verbindungsnetze sind die Lösung beim heutigen Stand der Technik
Dimensionalität	gibt an, in wieviel Dimensionen Verbindungen von jedem Schaltknoten des Netzes ausgehen.
Stufenzahl	Anzahl der Stufen, über die der Verbindungsweg aufgebaut wird

Auch die **Übertragungsstrategie** hat einen wesentlichen Einfluß auf die Leistung des Netzes, nicht zuletzt deshalb, weil sich diese direkt in der technischen Realisierung widerspiegelt. Zudem muß ein solches Netz konfigurierbar sein.

10.4.2 Übertragunsstrategie bei Paketvermittlung

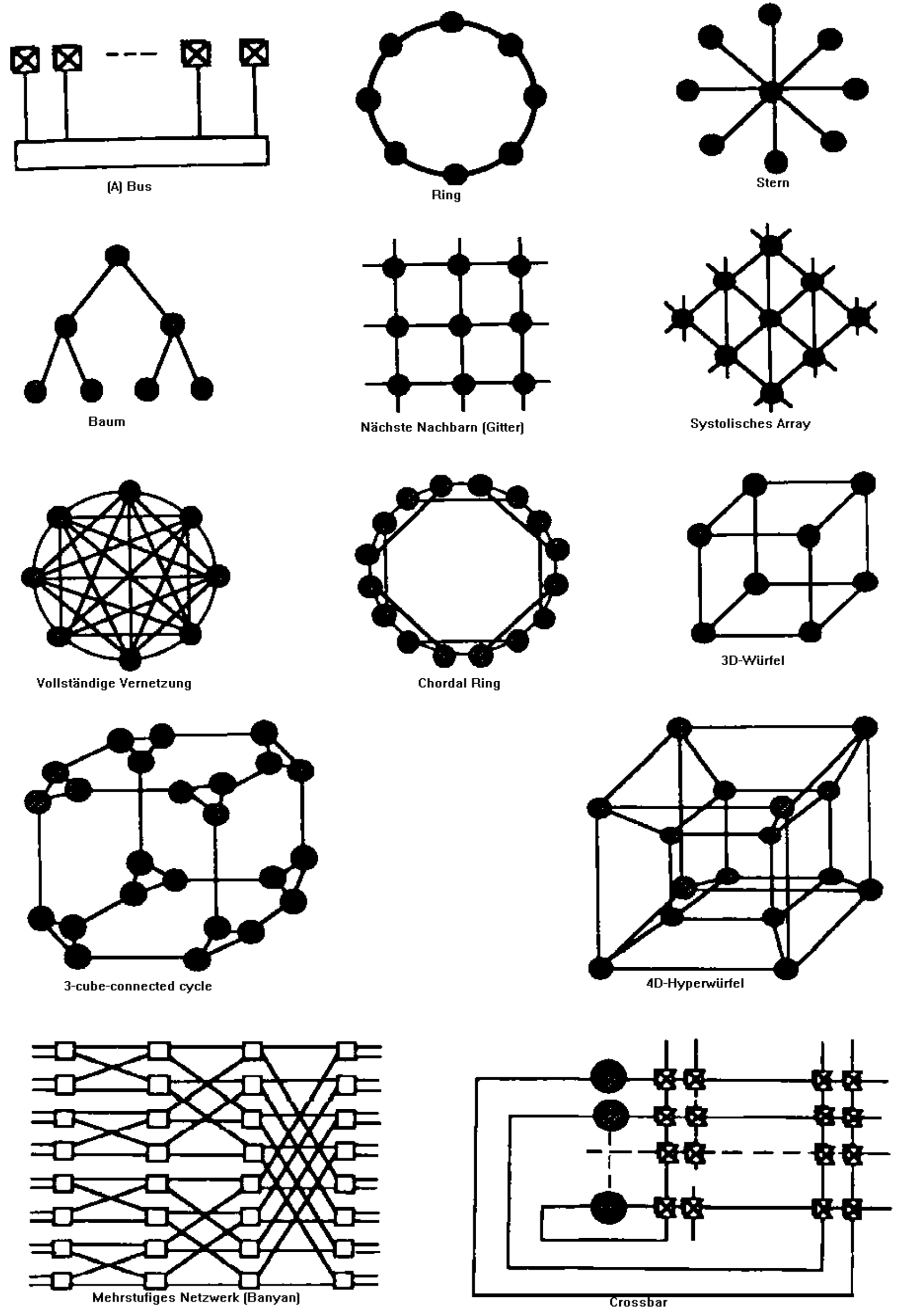
Es gibt 2 wesentliche Übertragungsstrategien, *Store-and-forward* Routing und *Worm Hole Routing*.

Bei *Store-and-forward* Strategie werden vom Sender vollständige Nachrichtenpakete in das Netz gesandt. Die Pakete bestehen aus Kopfteil mit Wegangabe und der eigentlichen Nachricht. Die Pakete werden im Netz von Station zu Station weitergereicht.

In jedem Schaltknoten wird das Paket zwischengespeichert und zwar solange, bis der Weg zum nächsten Schaltknoten frei ist. Die im Header enthaltene Wegangabe wird in jedem Knoten neu interpretiert.

Bei *Warm Hole-Strategie* wird zuerst der Header mit Wegangabe abgesandt und der Verbindungsaufbau durch das Netz bewerkstelligt. Dem Header folgen dann die weiteren Datenpakete. Jede Nachricht ist durch ein Schwanzpaket abgeschlossen. Im Netz ist eine Flußkontrolle erforderlich, da die Blockierung eines Schaltknotens dem Sender gemeldet werden muß. Bei Blockierungsmeldung unterbricht der Sender das Aussenden, bis die Blockierung wieder aufgehoben ist. In Phasen des Stillstandes verharren so viele Datenpakete der Nachricht im Netz, wie die freie Speicherkapazität der auf dem Wege liegenden Schaltknoten beträgt. Das Durchschieben von Nachrichten durch das Netz kann im Pipelining-Verfahren erfolgen. Dieses Verfahren bietet den Vorteil, daß es im blockierungsfreien Fall die Nachricht mit minimaler Verzögerung durch das Netz schleust.

☞ Das Store-and Forward Verfahren hat wegen der zeitaufwendigen Zwischenspeicherung der gesamten Nachricht in jedem Schaltknoten an Bedeutung verloren und wird praktisch nicht mehr angewandt.



Taxonomie regulärer Verbindungstopologien

Große Crossbar-Hierarchien sind stark im Kommen, da sie etwa gleich gute Verbindungseigenschaften wie die Hyperwürfel aufweisen, in der Realisierung jedoch kostengünstiger sind.

Technische Ausführung dynamischer Verbindungstopologien

- Crossbar ist ein Vielfachbussystem
- die Verbindungsleitungen werden auf den Bus durch eine spezielle Tri-State-Logik aufgeschaltet

10.4.3 Busverbindungen

Sie dienen für die verschiedensten Aufgaben in einem Parallelrechner

Beispiele:

- Datentransport
- Nachrichtenverkehr
- Synchronisation
- Interrupts
- Fehlerdiagnose

Die Ausführungsform der Busse sowie die Anforderungen an ihre Übertragungsgeschwindigkeit (Datenrate) kann entsprechend der Anwendung stark variieren.

Außer der Datenrate gibt es folgende Busparameter:

- Wortbreite: seriell bis parallel mit bis zu 128 Bits
- Übertragungsverfahren (synchron, asynchron)
- Hierarchiestufen: lokale Busse, globale Busse

Die Geschwindigkeit eines Bussystems wird durch eine Reihe von Faktoren begrenzt.

- Laufzeit der Leitungen (erfordert kurze Leitungen)
- Kapazitive Last der Empfängerkanäle
- Verzögerungszeit der Port-Ankopplung
- Arbitrierungszeit
- Taktversetzungen (clock skew)

Aus diesen Parametern sowie der Treiberleistung der Sender ergibt sich die erzielbare Zeit für einen Buszyklus und damit die Taktrate.

Busverbindungen stellen aus diesen Gründen nur solange eine schnelle Verbindungsstruktur dar, wie der Bus kurz gehalten werden kann. Ein Parallelbus, der nicht länger als eine Einschubbreite (ca. 40 cm) ist, kann beim heutigen Stand der Bauteiltechnik mit einem Bustakt von bis zu etwa 50 MHz arbeiten.

Der Parallelbus als Knotenverbindungsnetz ist nur für kleine Systeme mit einer geringen Knotenzahl nutzbar, z.B. innerhalb eines Clusters von 16 Knoten.

Beispiel einer Lösung im SUPRENUM-Rechner

- 16 Arbeitsknoten
- 2 Kommunikationsknoten
- Ein/Ausgabeknoten für die in jedem Cluster vorhandenen Plattenspeicher
- 1 Diagnoseknoten für alle Einrichtungen des Clusters

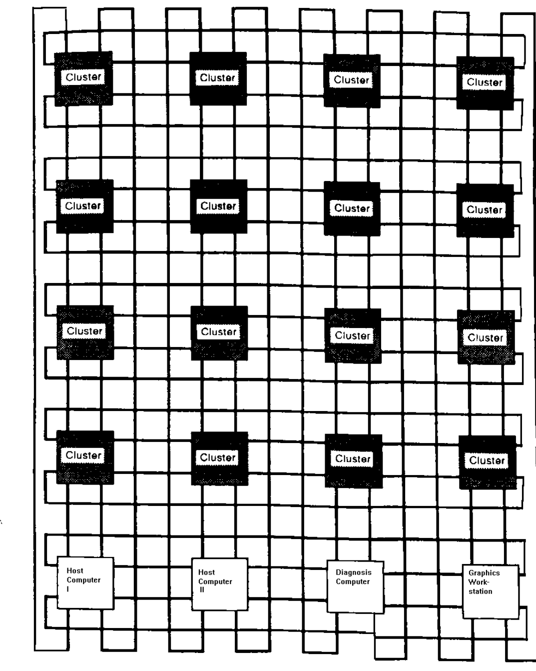
Diese 20 Knoten sind über 2 Parallelbusse mit einer Datenrate von je 160 MByte/s verbunden

10.4.4 Ringstrukturen

Ringstrukturen werden in der Regel als Tokenringe ausgeführt.

10.4.4.1 Übertragungsprotokoll:

- auf dem Ring zirkuliert ständig ein Token
- wenn niemand sendet, ist das Token als frei gekennzeichnet
- ein Knoten der Senden will, muß auf das Frei-Token warten
- sobald dieses zu ihm durch kommt, belegt er es (Tokenkennung ändert sich von frei in belegt)
- in den Header schreibt er die Empfänger- und seine eigene Adresse. Auch Multicast ist möglich
- an das Kopfpaket werden die Nachrichtenpakete angehängt.
- es können max. soviele Datenpakete im Ring kreisen, wie dessen Speicherkapazität beträgt.
- vom Senderknoten zirkuliert die Nachricht durch den Ring
- jeder Zwischenknoten liest den Header, ist die Empfängeradresse die eigene, so wird die Nachricht in einen Puffer kopiert, jedoch nicht aus dem Ring genommen.
- der Empfängerknoten quittiert die Nachricht oder gibt Fehlermeldung im Tail der Nachricht (mittels Prüfsumme im Endpaket)
- bei Erkennung der Sendeadresse als eigene Adresse weiß ein Knoten, daß dies seine Nachricht war. Die Nachricht wird jetzt aus dem Netz entfernt (Token wird wieder freigegeben) Eine sofortige Neubelegung des Tokens ist jetzt nicht möglich, sondern erst wenn der Token erneut durch den Ring gelaufen ist.



Ringbus-Verbindungssystem der SUPRENUM-Cluster

Ein Beispiel eines Ringbusverbindungsnetzes, welches im Suprenum-Rechner die Cluster verbindet.

Jede der Ringverbindungen wird durch 2 Token-Ring-Busse realisiert.

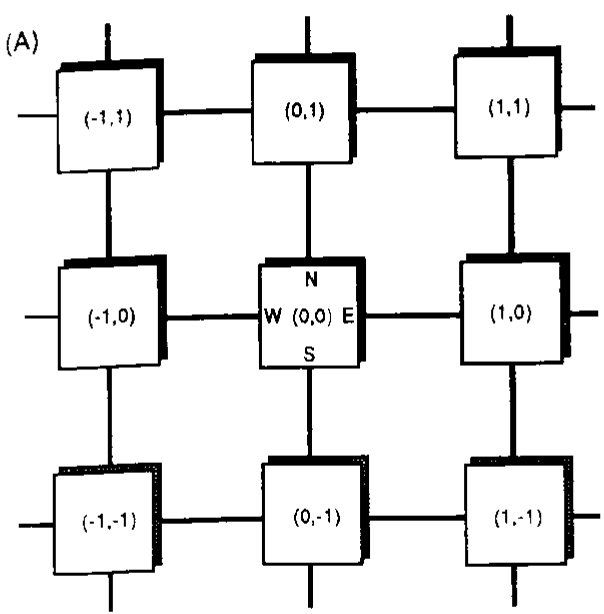
10.4.5 Gitterverbindungen

Vorteile:

Beliebige Skalierbarkeit:	es werden für jeden Knoten im 2D-Gitter genau vier Verbindungskanäle benötigt.
Wirtschaftlichkeit:	Kostengünstig wegen der nur 4 Verbindungskanälen
Günstiger Aufbau:	Innerhalb eines Einschubs kann die zweidimensionale Gitterverbindung vollständig durch eine Verbindungsplatine und damit ohne Kabel hergestellt werden.

Nachteil:

Ungünstiges Blockierungsverhalten	Einsatz in massiv-parallelen Rechnern hauptsächlich bei Anwendungen, deren Kommunikation eine starke Lokalität aufweist.



Realisierung einer 2-D-Gitterverbindung

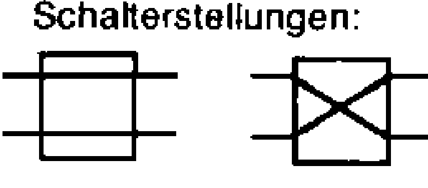
Die Kommunikation findet über spezielle Hardwareeinrichtung des Knotens (die Link Unit) statt.

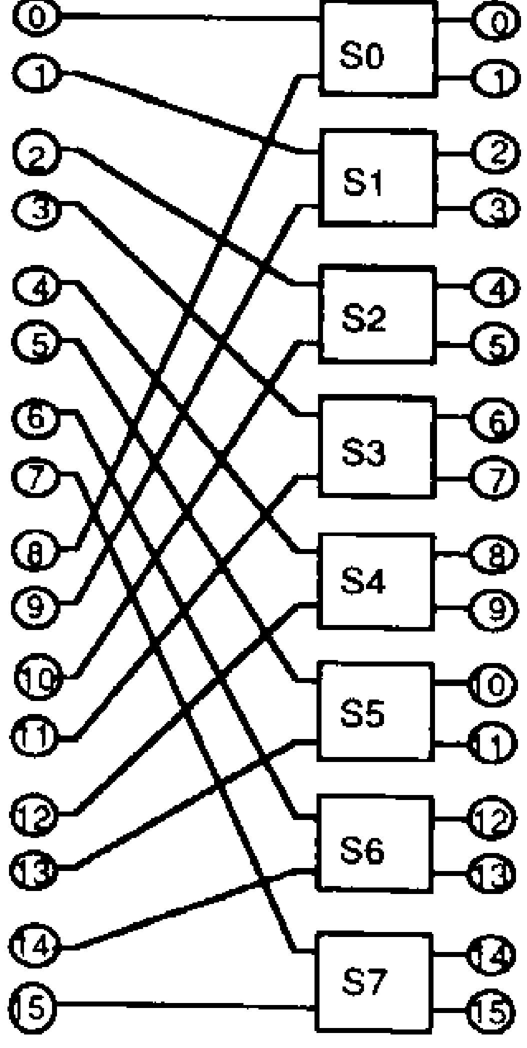
10.4.6 Mehrstufige Verbindungsnetze

- Sie verbinden N Sender mit N Empfängern bzw. N Knoten untereinander
- Die Knoten können Senden und Empfangen
- Verbindungsaufbau über mehrere Schaltstufen
- Jede Schaltstufe besteht aus derselben Zahl von Schaltzellen (switch boxes)

Skizze Shuffle-Exchange-Netzwerk

eine Grundstruktur zeigt ein Shuffle-Exchange-Netz





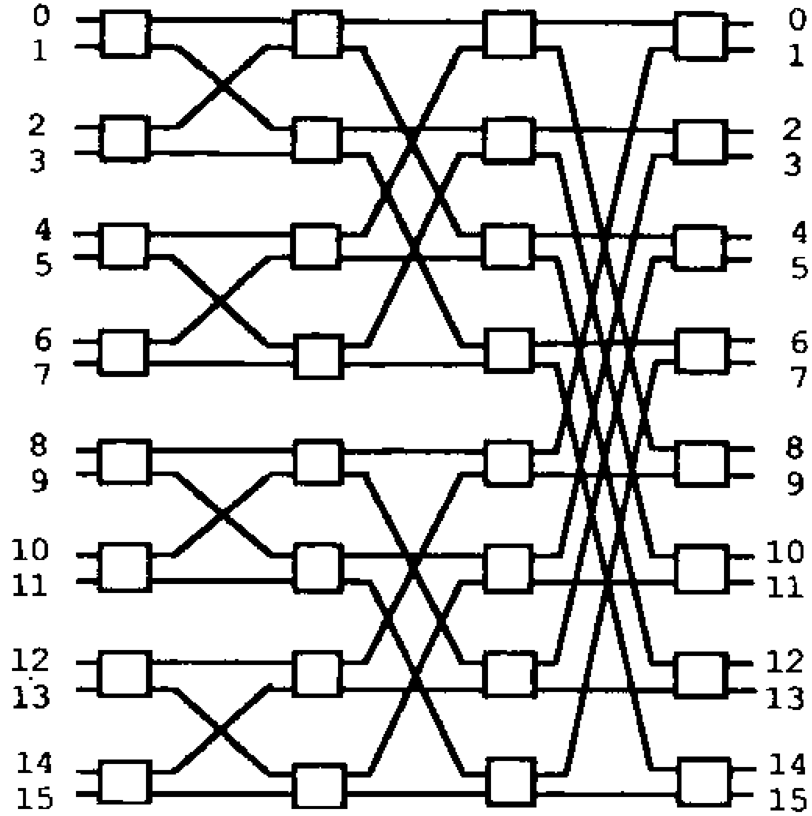
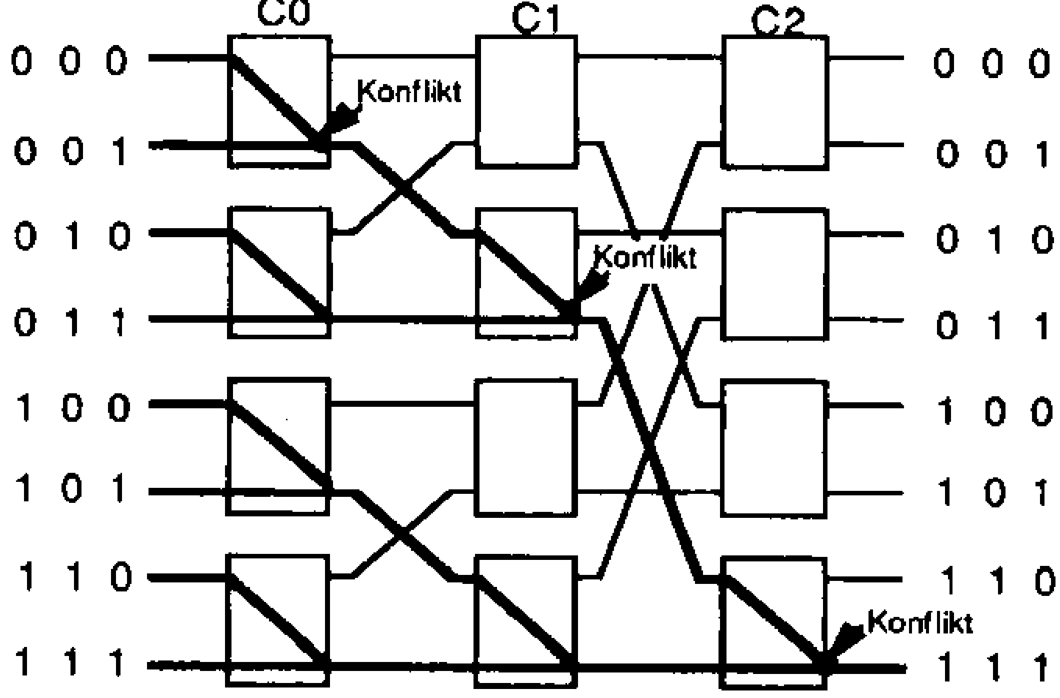


Bild 10-15 Banyan-Netzwerk für N=16



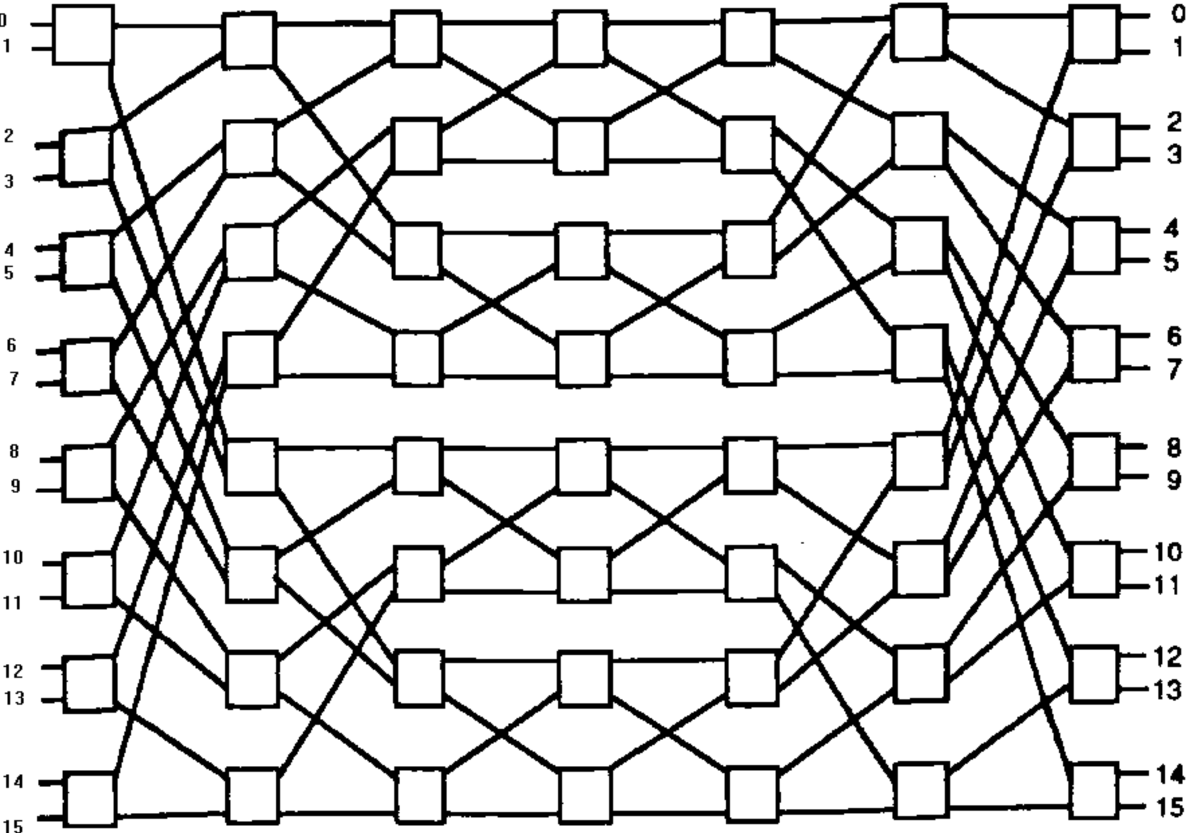
Der Weg von 000 nach 111 druch das Banyan-Netz blockiert das halbe Netz

Anstatt das Shuffle-Exchange-Netz mehrmals zu durchlaufen, kann man auch entsprechend viele Netzwerke dieser oder ähnlicher Art hintereinanderschalten

Beispielhaft das Banyan-Netzwerk

Alle diese Netzwerke haben die Eigenschaft, daß zu ihrer Realisierung genau N(log2N) binäre Schaltzellen benötigt werden.

- Aber wie in Skizze gezeigt ungünstiges Blockierungsverhalten
- Ein einziger Weg durch das Netz kann das halbe Netz blockieren.



- Solche Verbindungsnetze sind für eine Leitungsvermittlung ungeeignet.

Skizze Benes-Netzwerk für N = 16

10.4.7 Hyperwürfel

- hat dieselbe Komplexität wie das mehrstufige Netzwerk

- wesentlich besseres Verbindungsverhalten

- ein viel besseres Blockierungsverhalten

- Schaltzellen fallen mit den Knoten zusammen, d.h. auf dem Wege von einem Knoten zu einem anderen gibt es nicht wie beim mehrstufigen Netzwerk noch Schaltzellen als Zwischenstufen

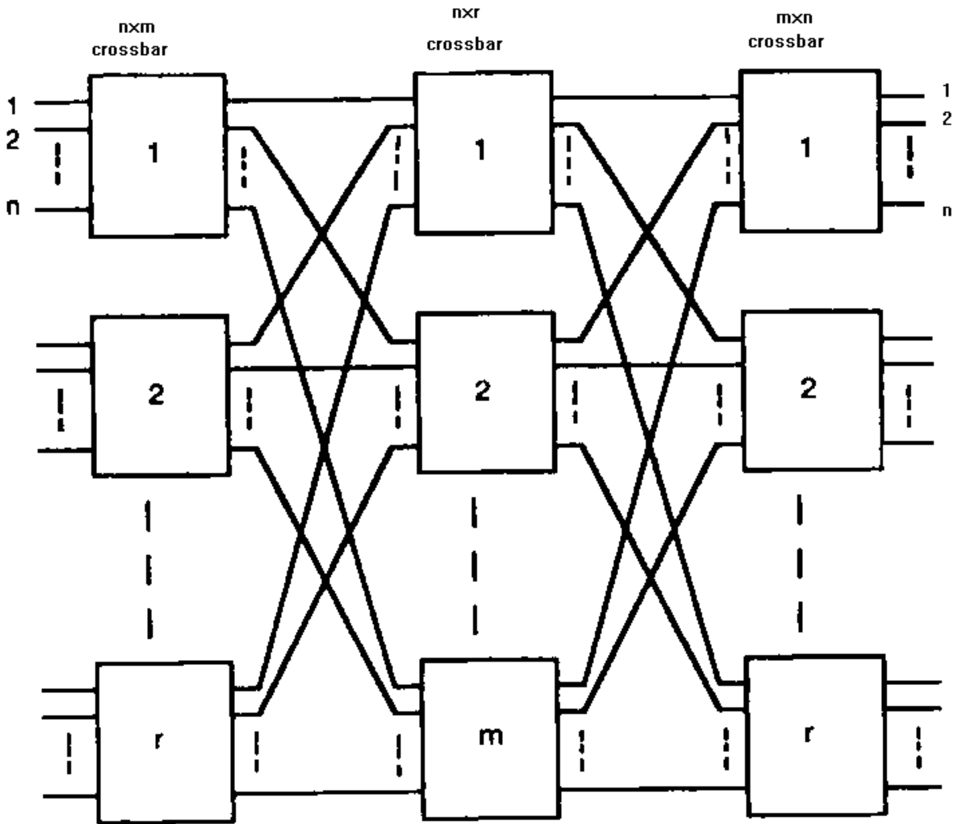
- kommt aus der Mode wegen hoher Kosten, die durch die vielen benötigten Verbindungskanäle entstehen.

10.4.8 Hierarchien von Crossbars

- einstufige Crossbar-Verbindung stellt die ideale Verbindungsform da, da jeder Sender jeden Empfänger ohne Zwischenstufen erreichen kann ⇒ keine Blockierungen
- bei integrierter Ausführung des Crossbars auf einem VLSI-Chip kosten die Schalter lediglich Chipfläche

Die Tabelle zeigt die beiden heutigen Ausführungsformen von Crossbar-Bausteinen

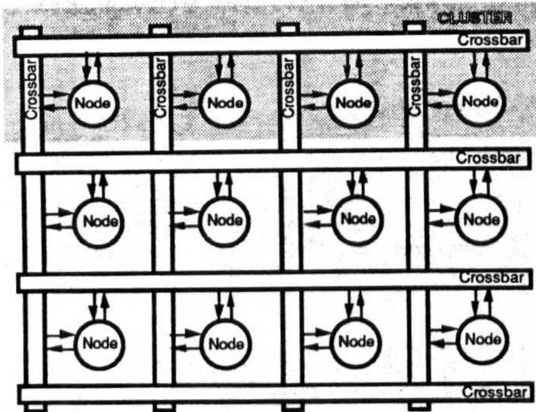
Anzahl der Datenbits pro Kanal	Crossbargröße	Takt	Übertragungsrate	Technologie
1	16X16 bis 64X64	0,4...2,4 Ghz	50...300 Mbyte/s	GaAs
8	16X16	50...70 Mhz	50 Mbyte/s	CMOS



Größere Systeme werden als Hierarchien von Crossbars aufgebaut.

Dazu werden auf der untersten Ebene jeweils eine Gruppe von Knoten (je nach Größe des zur Verfügung stehenden Crossbar-Bausteins) zu einem Cluster zusammengefasst. Auf der folgenden Ebene werden diese Cluster wieder durch Crossbars miteinander verbunden. (Vorgehen läßt sich bei Bedarf rekursiv wiederholen) Die Verbindung von zwei Knoten, die verschiedenen Clustern angehören, führen über maximal (2h+1) Zwischenstufen (h ist die Zahl der Hierarchieebene). Jetzt ist aber im Gegensatz zur einstufigen Crossbar-Verbindung Entstehung von Blockierung möglich.

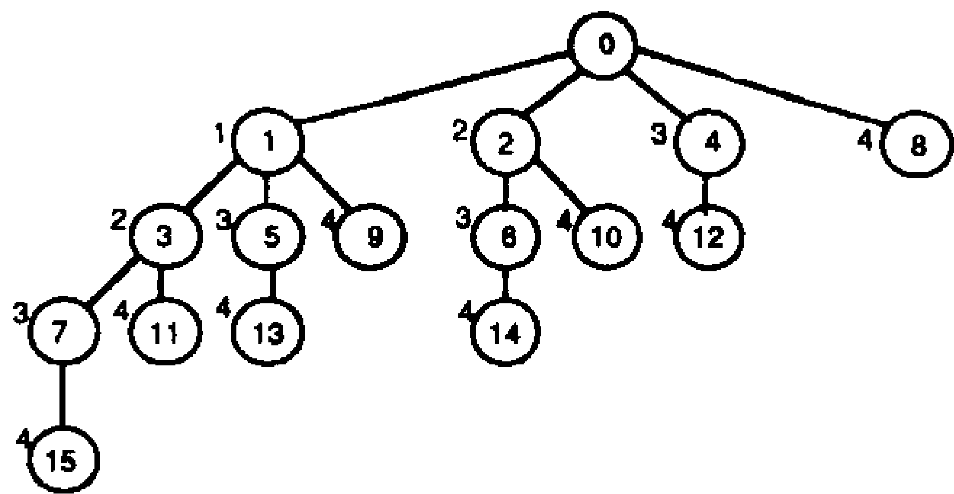
Durch die Wahl, wieviele Kanäle des Crossbars für die Verbindung der Cluster untereinander benutzt werden, läßt sich die Blockierungswahrscheinlichkeit bestimmen.



Skizze Beispiel einer zweistufigen Crossbar-Hierarchie

10.4.9 Eins-zu-N-Kommunikation in Verbindungsnetzen

- **multicast** zur Versendung von Nachrichten von einem Knoten des Systems zu N anderen Knoten (zum Beispiel für die Herstellung eines virtuellen gemeinsamen Speichers)
- broadcast zur Versendung von einem Sender an alle anderen Knoten
- nach der Theorie paralleler Algorithmen führt ein multicast auf baumförmigen Verbindungswegen durch das Verbindungsnetz



Skizze Beispiel eines logarithmischen Baums (Logtree)

- Die Zahlen neben den Knoten geben den jeweiligen Übertragungsschritt an
- Die Zahlen im Knoten sind die Knotenadresse
- Diese wird in der Reihenfolge der Nachrichtenübertragung (von oben nach unten und von rechts nach links) gebildet

Bildungsgesetz:

1. Anfangs gibt es einen Knoten

2. In jedem Dimensionsschritt bekommt jeder Knoten eine Korrespondenten in der neuen Dimension und wird mit ihm verbunden.

3. Die neuen Knoten werden untereinander genauso vernetzt wie die bisherigen

11 Grundlagen paralleler Software

11.1 Begriffsbestimmungen

Parallele Algorithmen definieren eine Halbordnung zwischen Anweisungen oder zwischen Anweisungsblöcken. Die Ausführung eines parallelen Programms kann dann auf folgende Weise geschehen:

- konkurrent auf einem Einprozessorsystem mit *Multiprogramming-Betriebssy*stern, wobei die parallelen Programmteile - hier meist Prozesse oder Threads - im Zeitscheibenverfahren ausgeführt werden,
- parallel auf einem nachrichten- oder speichergekoppelten Multiprozessor,
- verteilt auf einem Workstation-Cluster oder einem Metacomputer.

Paralleles und verteiltes Rechnen unterscheiden sich insofern, als in einem verteilten System Probleme wie Zuverlässigkeit, Sicherheit und Heterogenität beachtet werden müssen, welche in einem Multiprozessor von geringerer Relevanz sind. Auch sind die Kommunikationsgeschwindigkeiten über das lokale oder entfernte Netz geringer als über das Verbindungsnetz oder den Bus in einem Multiprozessor. Üblicherweise können deshalb nur sehr grobkörnig parallele Programme in effizienter Weise verteilt ausgeführt werden, während auf einem Multiprozessor mit schneller Kommunikation häufig auch relativ feinkömige parallele Programme eine hohe Beschleunigung erreichen können. Häufig werden jedoch die gleichen parallelen Programme sowohl auf Parallelrechnern als auch auf verteilten Systemen genutzt. Das wird dadurch ermöglicht, daß parallele Programmierumgebungen wie PVM und MPI gleichermaßen auf Parallelrechnern als auch auf verteilten Systemen zur Verfügung stehen. Grundlegende Anforderungen an parallele Software sind dabei neben der Parallelität die Skalierbarkeit des Algorithmus, die Lokalität der Datenzugriffe und die Modularität der Teilprogramme.

Parallele Programmiermodelle stellen parallele Abläufe und die Bindung von Daten an einen parallelen Ablauf explizit dar. Dazu definieren sie Abstraktionen für sequentielle Abläufe (z.B. Prozeß, Task, Thread) und für Interdependenzen zwischen den sequentiellen Abläufen (Nachrichten, entfernte Speicherzugriffe, Synchronisationen). Ein Programmiermodell sollte nicht die Maschinenkonfiguration (Anzahl der Verarbeitungselemente, Verbindungsnetzwerk) widerspiegeln. Neben konkreten Programmiermodellen gibt es auch **abstrakte parallele Programmiermodelle,** von denen im folgenden vier vorgestellt werden:

- **Speicherkopplung** (gemeinsamer Adreßraum): Der Zugriff auf gemeinsame Variablen geschieht entweder wie auf lokale Variablen oder durch spezialisierte Befehle für entfernte Speicherzugriffe (shmem_get, shmem_put). Weiterhin wird ein Synchronisationsmechanismus benötigt, wie z.B. Schloßvariablen, Semaphore oder Barrieren.
- Als Hardware-Realisierungen kommen speichergekoppelte Multiprozessoren in den Ausprägungen der symmetrischen Multiprozessoren und der Distributed shared-memory-Systeme in Betracht. Außerdem kann ein gemeinsamer Adreßraum auf einem nachrichtengekoppelten Multiprozessor oder auf einem verteilten System in Software realisiert werden.

- **Nachrichtenkopplung mit Prozeß-Kanal-Modell:** Der Austausch von Daten geschieht über Nachrichten. Prozesse kapseln sequentielle Programme und lokale Daten. Zu jedem Prozeß wird eine Schnittstelle zu anderen Prozessen als Menge von Pforten *(ports)* definiert. Meist werden Eingangs- *(inports)* und Ausgangspforten *(outports)* unterschieden. Ein Prozeß kann Nachrichten auf seine Ausgangspforte schicken und Nachrichten von seiner Eingangspforte empfangen. Nur die entsprechende Pfortennummer ist in den Sende- oder Empfangsoperationen als Adresse nötig. Die Nachrichtenreihenfolge bleibt auf jeder Pforte erhalten. Ausgangspforten und Eingangspforten (verschiedener Prozesse) werden über sogenannte Kanäle *(channels)* miteinander verbunden. Das geschieht häufig in einer speziellen Konfigurationssprache oder einer Verbindungsdefinition *(channel conflguration)* im Programm.
- **Nachrichtenkopplung mit Prozeß-Nachrichten-Modell:** Ähnlich wie im vorherigen Modell werden Daten über Nachrichten ausgetauscht. Jeder Prozeß wird durch eine Prozeßnummer identifiziert. Diese Identifikation geschieht entweder über eine systemeindeutige Prozeßnummer oder durch eine virtuelle Knotennummer und eine knotenrelative Prozeßnummer. Eine Nachricht enthält im letzten Fall Zielprozeß und Zielknoten als identifizierende Adresse. Die Einhaltung der Reihenfolge der verschickten Nachrichten ist von praktisch allen Systemen garantiert.
- **Datenparallelität:** Dabei wird die gleiche Operation auf allen Elementen eines Datenfeldes parallel ausgeführt. Der Programmierer muß die Datenverteilung auf ein virtuelles Prozessorfeld vornehmen. Innerhalb des virtuellen Prozessorfeldes können Daten zwischen benachbarten virtuellen Prozessoren in für das gesamte Prozessorfeld synchroner Weise ausgetauscht werden. Zum Beispiel können alle virtuellen Prozessoren gleichzeitig den Inhalt einer (lokalen) Speicherzelle an ihre nördlichen Nachbarprozessoren schicken. Als Hardware-Realisierung ist hier idealerweise ein Feldrechner vorgesehen. Jedoch können datenparallele Programme auch in SPMD-Programme *(single program multiple data)* überführt und dann auf Multiprozessoren ausgeführt werden.

Zur Formulierung eines Programms nach einem Programmiermodell kann eine parallele Programmiersprache oder eine parallele Programmierschnittstelle eingesetzt werden. Eine **parallele Programmierschnittstelle** wie z.B. PVM, MPI und Linda zeichnet sich dadurch aus, daß eine sequentielle Sprache wie C, FORTRAN etc. um zusätzliche parallele Sprachkonstrukte erweitert wird. Diese Konstrukte werden jedoch nicht in die Sprache integriert (und damit vom Compiler umgesetzt), sondern durch Aufruf von Standardfunktionen in Verbindung mit einem Laufzeitsystem implementiert. Die Standardfunktionen selbst sind wieder in einer sprachspezifischen Funktionsbibliothek zusammengefaßt.

Unter einer **parallelen Programmierumgebung** wie PVM 3.3.x, MPICH etc. versteht man eine parallele Programmierschnittstelle oder Sprache mit zusätzlichen Werkzeugen. Diese Werkzeuge sind im Falle einer parallelen Programmiersprache zumindest der zugehörige Compiler und im Falle einer parallelen Programmierschnittstelle das Laufzeitsystem und die Funktionsbibliothek, welche die parallelen Sprachkonstrukte implementieren. Hinzu kommen eventuell nützliche Werkzeuge zum Debuggen, zur Leistungseinschätzung und zur Visualisierung. Im Falle von PVM oder MPI sind die Programmierumgebungen weitgehend maschinenunabhängig. Häufig gehört jedoch eine eigenständige, maschinenspezifische Programmierumgebung zu einem Parallelrechner dazu.

Parallele und **verteilte Betriebssysteme** gehen über eine parallele Programmierumgebung durch zusätzliche Leistungen wie Prozeßmanagement, Ressourcenmanagement oder Dateiverwaltung hinaus.

11.2 Sprachansätze zur Programmierung von Parallelrechnern

Die wichtigsten derzeit verwendeten Ansätze zur Programmierung von Parallelrechnern sind:

- Nutzung implizit vorhandener Parallelität durch parallelisierende oder vektorisierende Compiler für sequentielle Sprachen,
- Erweiterung von sequentiellen Sprachen um eine parallele Programmierschnittstelle durch Betriebssystemprimitive oder Funktionsbibliothek und Laufzeitsystem,
- imperative Sprachen mit Konstrukten zur Spezifikation expliziter Parallelität,
- objektorientierte parallele Sprachen und
- Datenflußsprachen sowie parallele funktionale und logische Sprachen.

Zur Nutzung implizit vorhandener Parallelität werden Programme, die in einer schon bestehenden sequentiellen Sprache, meist FORTRAN, geschrieben sind, durch parallelisierende oder vektorisierende Compiler auf die Maschinensprache des Parallelrechners abgebildet. Unterstützt wird diese Abbildung durch Compilerdirektiven des Programmierers. Parallelisiert werden nur Schleifen, wobei meist zuvor umfangreiche Schleifenumordnungen durchgeführt werden. Damit lassen sich die schon existierenden, umfangreichen FORTRAN-Programme mit nur geringen Änderungen weiterverwenden. Die genutzte Parallelitätsebene ist die Blockebene bei parallelisierenden Compilern und die Suboperationsebene bei vektorisierenden Compilern. Diese Vorgehensweise wird heute bei speichergekoppelten Multiprozessorsystemen und mit vektorisierenden Compilern bei Vektorrechnern eingesetzt. Parallelisierende Compiler für nachrichtengekoppelte Multiprozessoren gibt es bereits (DEC F90, Portland Group-Compiler), sind jedoch noch nicht so weit entwickelt wie für speichergekoppelte Systeme. Es läßt sich dagegenhalten, daß sich Parallelalgorithmen in ihrer Konstruktion von sequentiellen Algorithmen meist signifikant unterscheiden. Das Wissen des Programmierers über die im Problem vorhandene Parallelität kann von einem parallelisierenden Compiler nicht genutzt werden.

Eine weitere Methode besteht darin, eine sequentielle Sprache (FORTRAN, Modula-2, C oder C++) um Bibliotheksfunktionen für parallele Kontroll- und Synchronisationskonstrukte zu erweitern. Die explizite Programmierung paralleler Abläufe wird somit in Bibliotheksfunktionen versteckt, die vor der Ausführung hinzugebunden werden. Das erlaubt es, schon existierende sequentielle Compiler relativ leicht an einen Parallelrechner anzupassen. Auch können einmal entwickelte Bibliotheksfunktionen für mehrere Sprachen verwendet werden.

Durch Erweiterung imperativer Sprachen um Betriebssystemfunktionen spezifischer Parallelrechner kann Parallelität auf niedriger Abstraktionsebene spezifiziert werden. Dies kann zu hoher Effizienz führen und ermöglicht das Plazieren von Prozessen auf Prozessoren durch den Programmierer. Typischerweise liegt die Parallelität auf Task- und Blockebene. Andererseits führt die niedrige Abstraktionsebene der Betriebssystemprimitive oft zu mühsamer Programmierung. Durch die Verwendung der Betriebssystemfunktionen ist die Programmentwicklung meist maschinenabhängig. Einmal entwickelte Programmsysteme sind nicht leicht auf andere Parallelrechner übertragbar.

Um diese Übertragbarkeit zu vereinfachen, werden zur Programmierung nachrichtengekoppelter Multiprozessoren in der letzten Zeit zunehmend die maschinenunabhängigen Programmierumgebungen PVM und MPI verwendet und von den verschiedenen Multiprozessorherstellern auf ihren Systemen zur Verfügung gestellt. Bei speichergekoppelten Multipro-

zessoren beginnt sich der POSIX-Standard für Threads als maschinenunabhängige Programmierschnittstelle durchzusetzen.

Als weiterer Ansatz kann auch eine neue parallele Sprache geschaffen werden, die explizit parallele Kontroll- und Synchronisationskonstrukte enthält. Oft entsteht eine derartige Sprache aus Erweiterungen bekannter sequentieller Sprachen. Die Sprache kann maschinenabhängig oder maschinenunabhängig definiert sein. Im ersten Fall versucht man, die Hardware-Strukturen einer speziellen Maschine optimal auszunutzen. Im zweiten Fall wird eine hardwareunabhängige Sprache geschaffen, die parallele Sprachkonstrukte für verschiedene Parallelitätsebenen enthält. Durch Compilation können verschiedene Typen von Parallelrechnern so unterstützt werden, daß nur die umsetzbare Parallelität auch wirklich genutzt wird, während die auf der speziellen Maschine nicht nutzbare Parallelität in eine sequentielle Verarbeitungsfolge übersetzt wird.

Für FORTRAN-Erweiterungen wird oft ein PARALLEL-DO-Konstrukt verwendet, das eine parallele Version der FOR-Schleife darstellt, wobei abgesehen von der Laufvariablen keine Datenabhängigkeiten zwischen den einzelnen Iterationen erlaubt sind.

Ein parallelisierender Compiler kann zwei verschiedene Strategien verwenden, um die mittels eines PARALLEL-DO-Konstruktes definierte Parallelität in Parallelarbeit umzusetzen. Zum einen kann die erste Iteration dem ersten Prozessor, die zweite dem zweiten Prozessor etc. zugeteilt werden. Ein Prozessor, der mit seiner Iteration fertig ist, nimmt sich automatisch die nächste verfügbare Iteration vor, solange noch Iterationen übrig sind. Diese Strategie wird als *Selfscheduled PARALLEL DO* bezeichnet. Sie führt automatisch zu einer guten Ausnutzung der Prozessoren, benötigt jedoch zusätzliche Synchronisationsbefehle, um sicherzustellen, daß kein Prozessor dieselbe Iteration zur Bearbeitung erhält. Die FORTRAN-Dialekte VMIEPEX und FORCE enthalten beide *Self_scheduled PARALLEL* DO-Konstrukte.

Im anderen Fall weist der Compiler jedem Prozessor eine bestimmte Anzahl von Iterationen der Schleife zu. Diese Strategie wird als *Prescheduled PARALLEL DO* bezeichnet. Sie führt zu einer schlechteren Prozessorauslastung, falls die einzelnen Iterationen verschieden lange Ausführungszeiten benötigen.

PARALLEL-DO-Konstrukte werden sowohl in Sprachen für Vektorrechner als auch in solchen für speichergekoppelte Multiprozessoren eingesetzt. Bei geschachtelten Schleifen wird im ersten Fall in der Regel über die innerste Schleife parallelisiert, während im zweiten Fall die Parallelisierung auf die äußerste Schleife angewandt wird. Bei Sprachen für Feldrechner werden PARALLEL-DO-Konstrukte mit spezifisch feldrechnertypischer Semantik eingesetzt.

Die Komplexität paralleler Programme stellt besonders hohe Anforderungen an Software-Entwicklung und -pflege. Objektorientierte parallele Programmiersprachen kombinieren die softwaretechnischen Eigenschaften objektorientierter Sprachen mit dem Konzept der aktiven, miteinander kommunizierenden Objekte. Dieses Konzept stellt eine abstrakte Form der Parallelprogrammierung dar, ist jedoch auf die Ebene der Task-Parallelität beschränkt. Nachteilig ist, daß diese Sprachen sich allesamt in einem Experimentierstadium befinden, nur auf wenigen speziellen Rechnern verfügbar sind und noch wenig Erfahrungen mit dieser Form des Programmierstils vorliegen. Einzige Ausnahme ist hier das Thread-Konzept der mittlerweile weit verbreiteten objektorientierten Programmiersprache Java.

Für funktionale und logische Programmiersprachen ist die Umsetzung der implizit vorhandenen Parallelität durch den Compiler noch nicht zufriedenstellend gelöst. Datenflußsprachen sind durch ihre Parallelität auf der Anweisungsebene auf Datenflußrechner zugeschnitten. Allerdings werden Datenflußsprachen wie SISAL und Id mittlerweile auch erfolgreich auf Multiprozessoren implementiert.

11.3Prozesse und Threads

Bei einem **sequentiellen** Programm werden die Anweisungen des Programms und die einzelnen Schritte des Algorithmus nacheinander in einer vorbestimmten Folge durchlaufen. Ist Sequentialität bei einer Problemlösung nicht zwingend erforderlich, dann kann der Verzicht auf die Forderung nach sequentieller Ausführung von Anweisungen zweierlei bedeuten:

- Die Anweisungen können parallel (im Sinne von zeitgleich) von mehreren Prozessoren ausgeführt werden.
- Die Anweisungen können in einer beliebigen Folge sequentiell von einem Prozessor ausgeführt werden.

Genau diese Eigenschaften sind gemeint, wenn man die Anweisungen als **nebenläufig** *(concurrent)* bezeichnet. Parallelität ist damit eine spezielle Form der Nebenläufigkeit: Nebenläufige Anweisungen sind nur dann parallel, wenn für ihre Ausführung tatsächlich mehrere Prozessoren zur Verfügung stehen.

Dieser strengen Unterscheidung von „nebenläufig" und „parallel" steht der meist laxere Sprachgebrauch entgegen, wonach „nebenläufig" und „parallel" oft gleichgesetzt werden. **„Parallelität"** ist hierbei eine Kurzform für „die Möglichkeit der parallelen (gleichzeitigen) Ausführung mehrerer Operationen" [Gil93]. Wenn nicht direkt unterschieden werden muß, wird auch im folgenden meist der Begriff „Parallelität" statt „Nebenläufigkeit" verwendet, d.h., der Begriff „Parallelität" wird im Sinne der Möglichkeit zur parallelen (gleichzeitigen) Ausführung mehrerer Operationen angewandt.

Die Leistung eines Programms wird durch das Ablaufen vieler Einzelschritte erbracht, die durch bestimmte Anweisungen und Eingaben gesteuert werden. Jeder dieser Schritte kann als Aktivität bezeichnet werden. Aktivitäten existieren auf verschiedenen Abstraktionsebenen z.B. auf der Ebene der Maschinenbefehle oder der Ebene der Anweisungen in einer höheren Programmiersprache. Eine Aktivität auf einer höheren Abstraktionsebene kann sich aus mehreren Aktivitäten der nächstniedrigeren Ebene zusammensetzen. Unabhängig vom Charakter der zusammengesetzten Aktivität können ihre Teilaktivitäten sequentiell oder parallel sein. Auf jeder Abstraktionsebene lassen sich Aktivitäten strukturieren, indem man „Prozesse" bildet. In diesem Sinne ist ein **Prozeß** eine sequentielle Folge von Aktivitäten, durch die eine in sich abgeschlossene Aufgabe bearbeitet wird.. Programme, aus denen mehrere Prozesse resultieren, werden auch als parallele Programmsysteme bezeichnet.

Ein stärkerer Bezug auf die „dynamische" Natur eines Prozesses ergibt sich daraus, daß ein Prozeß oft mit einem „in Bearbeitung befindlichen Programm" gleichgesetzt wird. Die Definition nach [Gil93] betont diesen dynamischen Aspekt des Prozeßbegriffs.

Ein **Prozeß** wird als eine funktionelle Einheit definiert, die aus

- einem zeitlich invarianten Programm,
- einem Satz von Daten, mit denen der Prozeß initialisiert wird, und einem zeitlich varianten Zustand besteht.

Hilfreich ist die Unterscheidung der Umgebung und des Kontextes eines Prozesses. Prozesse sind geschützte Programmeinheiten: Jeder Prozeß hat seine eigenen Code- und Datenbereiche im Speicher, deren Zugang allen anderen Prozessen verwehrt ist. Dazu kommen die geöffneten Dateien und Ressourcenverweise. Die geschützten Adreßbereiche eines Prozesses werden als **Umgebung** eines Prozesses bezeichnet. Ein **Prozeß** besteht aus einem Speicherbereich, der den Code, die Daten und Datei- bzw. Ressourcen-Kontrollblöcke enthält, und einem Befehlszähler als Zeiger in die Befehlsfolge. Die Adreßbereiche sind in Seiten- oder Segmentta-

bellen niedergelegt, deren Inhalte bei einem Umgebungswechsel ausgetauscht werden müssen. Ein Prozeßwechsel bedingt somit einen Umgebungswechsel.

Unter dem **Kontext** eines Prozesses versteht man die Registerwerte des mit der Prozeßausführung beschäftigten Prozessors. Zum Kontext gehören der Befehlszähler, der Zeiger auf den Laufzeitstapel oder den Aktivierungssatz sowie weitere Zustandsinformationen und Datenwerte, die bei der Wiederaufnahme einer Prozeßausführung gebraucht werden. Ein Prozeßwechsel bedingt folglich auch immer einen Kontextwechsel.

Aus Betriebssystemsicht führt das Prozeßkonzept zu **erheblichem Laufzeitaufwand**:

- Das Erzeugen eines Prozesses erfordert zeitaufwendige Aufrufe des Betriebssystems.
- Falls Prozesse von mehreren Benutzern gleichzeitig ablaufen sollen, sind Zugriffsschutzmechanismen nötig.
- Ein Prozeßwechsel ist oft - bedingt durch den Kontext- und Umgebungswechsel - eine relativ aufwendige Operation.
- Falls Prozesse miteinander kommunizieren, müssen Kommunikationswege eingerichtet und Synchronisationsmechanismen bereitgestellt werden.
- Im klassischen UNIX-Gebrauch besitzt ein Prozeß nur *einen* Kontrollfaden.

Aus diesen Gründen werden zunehmend sogenannte mehrfädige *(multithreaded)* Betriebssysteme eingesetzt, bei denen zwischen sogenannten schwer- und leichtgewichtigen Prozessen unterschieden wird. Schwergewichtige Prozesse sind die oben beschriebenen Prozesse im klassischen Sinne.

Leichtgewichtige Prozesse dagegen werden auch als Threads (threads of control) oder in wörtlicher Ubersetzung als Kontrollfäden bezeichnet. Eine Anzahl von Threads teilt sich eine gemeinsame (Prozeß-)Umgebung, d.h. den Adreßraum des umgebenden Prozesses, geöffnete Dateien und andere Betriebsmittel. Geschützte Speicherbereiche gibt es zwar noch für den umschließenden Prozeß, nicht jedoch für den einzelnen Thread. Mit einem Thread-Wechsel ist somit kein Umgebungswechsel, sondern nur ein Kontextwechsel verbunden. Sinn des Thread-Konzepts ist es, eine Programmeinheit zu bilden, die parallel zu anderen Programmeinheiten arbeiten kann. Da sich die Threads den Adreßraum des umgebenden Prozesses teilen, muß ein unsynchronisierter, lesender und schreibender Zugriff der Threads aufgemeinsame Variablen oder auf gemeinsame Betriebsmittel durch Synchronisationen verhindert werden.

Praktisch alle Betriebssysteme von Arbeitsplatzrechnern sind heute mehrfädige Betriebssysteme, die auf UNIX aufsetzen. Auch die PC-Betriebssysteme Windows95, Windows NT und LINUX 2.x sind mehrfädige Betriebssysteme. Dabei können innerhalb der UNIX-Prozesse Threads als parallele Kontrollfäden vom Programmierer genutzt werden.

Bislang definiert jedes dieser Betriebssysteme seine eigene Thread-Bibliothek mit untereinander leicht variierender Syntax und Semantik. (z.B. DEC-Threads, Solaris-Threads, DCE-Threads).

Mit der endgültigen Standardisierung **POSIX 1003.lc-1995** wurde 1995 vom POSIX-Komitee eine Thread-Schnittstelle geschaffen, die im folgenden als **Pthread (POSIX-Thread)** bezeichnet wird Die Pthread-Schnittstelle stellt zur Synchronisation Schloßvariablen *(mutex)* und Bedingungsvariablen *(condition variable)* zur Verfügung Pthread-Bibliotheken werden seit der Standardisierung zunehmend auf Arbeitsplatzrechnern neben den herstellerabhängigen Thread-Bibliotheken bereitgestellt. Die Pthread-Schnittstelle hat die Chance, in den nächsten Jahren eine ähnlich weit verbreitete Standardschnittstelle für Arbeitsplatzrechner und

und speichergekoppelte Multiprozessoren zu werden, wie sie der MPI-Standard und der De-facto-Standard PVM für nachrichtengekoppelte Systeme darstellen.

11.4 Synchronisation und Kommunikation über gemeinsame Variable

11.4.1 Grundlagen

Prozesse [8] sind voneinander **abhängig,** wenn sie in einer bestimmten Reihenfolge ausgeführt werden müssen. Es gibt zwei mögliche Gründe für die Abhängigkeit von Prozessen und damit die Notwendigkeit ihrer Koordination:

- Eine **Kooperation** entsteht dadurch, daß die Prozesse jeweils bestimmte Teilaufgaben im Rahmen der vorgegebenen Gesamtaufgabe erfüllen. Es gibt unterschiedliche Formen der Kooperation. Bei einem Produzenten/Konsumenten-System oder Erzeuger-/Verbraucher-System *i(nroducer/consumer System)* nimmt ein Prozeß Daten auf, die ein anderer erzeugt hat. Bei einem **Auftraggeber-/Auftrag-nehmer-System** *(client/server System)* nimmt ein Prozeß nicht nur Daten von einem anderen Prozeß auf, sondern liefert diesem auch später Daten zurück Weitere kompliziertere Kooperationsformen entstehen durch Kombination dieser einfacheren Formen.
- Eine **Konkurrenz** *(competition)* entsteht dadurch, daß die Aktivitäten eines Prozesses die eines anderen zu behindem drohen.

Kooperation und Konkurrenz schließen einander nicht aus, sondern gehen häufig Hand in Hand. Vor allem kooperierende Prozesse stehen bei der Lösung von Teilaufgaben oft zugleich miteinander in einer Konkurrenzsituation.

Die Koordination der Kooperation und Konkurrenz zwischen Prozessen wird Synchronisation genannt. Eine Synchronisation bringt die Aktivitäten verschiedener Prozesse in eine Reihenfolge. Durch die Synchronisation von Prozessen wird die Unabhängigkeit der Abfolge von Aktivitäten verschiedener Prozesse eingeschränkt. Bestimmte Aktivitäten eines Prozesses können erst nach - oder nicht gleichzeitig mit - bestimmten Aktivitäten anderer Prozesse ablaufen. Der Beginn ihrer Durchführung muß somit verzögert werden.

Prozesse können nicht nur ihre Aktivitäten aufeinander abstimmen, sondern darüber hinaus einander mit Informationen versorgen. Man nennt diesen Datenaustausch über Prozeßgrenzen hinweg **Kommunikation.** Kommunikation und Synchronisation bedingen sich gegenseitig: Zum Abstimmen zweier Aktivitäten können Prozesse Informationen austauschen, und zum Zweck des Austauschs müssen sie die Aktivitäten der Informationsabgabe und -aufnahme aufeinander abstimmen.

Um ihre Aktivitäten untereinander abzustimmen, müssen sich Prozesse gegenseitig mit Informationen versorgen. Eine Möglichkeit dieser Kommunikation bieten gemeinsame **Variablen** *(shared variables),* auf die Prozesse schreibend (modifizierend) und lesend (nichtmodifizierend) zugreifen können.

Gemeinsame Variablen sind nicht nur zu Synchronisationszwecken notwendig, sondern können auch aufgrund einer gegebenen Problemstellung erforderlich sein, wenn mehrere Prozesse für ihre jeweilige Arbeit die gleichen Ausgangsdaten benötigen.

---

[8] Für den Rest des Abschnitts wird zwischen schwergewichtigen und leichtgewichtigen Prozessen nicht unterschieden.

Die Folge der einzelnen Lese- oder Schreibzugriffe wird dadurch bestimmt, wie schnell es den Prozessoren gelingt, ihre Zugriffsoperationen durchzuführen. Man sagt, die Prozessoren liefem sich ein Wettrennen *(race).* Programmsysteme sollen im allgemeinen determinierte Ergebnisse liefem, egal welche Zugriffsfolge durch das Wettrennen entsteht. Ihre Ausgaben sollen unabhängig von den Umständen des **Wettrennens** *(race conditions)* sein. Eine Synchronisation schränkt das Wettrennen unter den Prozessen ein, um dadurch entstehende Fehler zu vermeiden. Man unterscheidet zwei verschiedenen Arten der Synchronisation:

Bei der einseitigen Synchronisation sind zwei Aktivitäten AI und A2 voneinander abhängig, und zwar in dem Sinn, daß Al die Voraussetzung für das richtige Ergebnis von A2 bildet. Al muß also vor A2 ausgeführt werden. A2 wird so lange verzögert, bis Al zum Abschluß gekommen ist. Die Synchronisation wirkt sich auf keinen Fall auf Al aus, sie wird deshalb als einseitig (unilateral) bezeichnet.

Bei der mehrseitigen Synchronisation ist die zeitliche Abfolge zwischen zwei Aktivitäten AI und A2 gleichgültig. Aufgrund von Schreib/Schreib- oder Schreib-,Lese-Konflikten besteht jedoch eine Abhängigkeit zwischen den Aktivitäten. A 1 darf nicht zusammen mit A2 ausgeführt werden. Soll A2 begonnen werden, während Al durchgeführt wird, verzögert sich ihr Start so lange, bis Al zum Abschluß gekommen ist. Entsprechendes gilt für AI, sollte A2 bereits begonnen worden sein. Dies läßt sich auf mehrere Aktivitäten erweitern. Die Synchronisation wird deshalb als mehrseitig (multilateral) bezeichnet.

Kennzeichnend für eine mehrseitige Synchronisation ist, daß Aktivitäten, die von ihr betroffen sind, in einem gegenseitigen **Ausschluß** *(mutual exclusion)* zueinander stehen. Sich gegenseitig ausschließende Aktivitäten werden nie parallel durchgeführt und verhalten sich zueinander, als seien sie unteilbar, weil keine Aktivität eine andere Aktivität unterbrechen kann. Anweisungen, deren Ausführung einen gegenseitigen Ausschluß erfordert, heißen kritische Abschnitte oder kritische **Bereiche** *(critica/ sections).*

Durch Synchronisationsmaßnahmen werden Prozesse verzögert. Irgendwann sollte der Grund für die Verzögerung wegfallen, damit die Prozesse weiterarbeiten können. Leider können Prozesse selbst verhindern, daß sie fortgesetzt werden.Eine solche Situation, in der Prozesse auf Ereignisse warten, die nicht mehr eintreten können, wird als **Verklemmung** *(deadlock)* bezeichnet.

Eine **Verklemmung** bedeutet, daß die Dienstleistung nicht mehr erbracht werden kann, weil sich mehrere Prozesse gegenseitig blockieren.

Eine **Aussperrung** tritt auf, wenn ein Prozeß undefinierbar lang verzögert wird.

Es läßt sich zeigen, daß Verklemmungen beim Streit um Betriebsmittel nur dann auftreten können, wenn die folgenden **vier notwendigen und hinreichenden Bedingungen** erfüllt sind

- Die umstrittenen Betriebsmittel sind nur exklusiv nutzbar.
- Die umstrittenen Betriebsmittel können nicht entzogen werden.
- Die Prozesse belegen die schon zugewiesenen Betriebsmittel auch dann, wenn sie auf die Zuweisung weiterer Betriebsmittel warten.
- Es gibt eine zyklische Kette von Prozessoren, von denen jeder mindestens ein Betriebsmittel besitzt, das der nächste Prozeß der Kette benötigt.

Dem Verklemmungsproblem kann man auf drei Arten begegnen.

- Man kann versuchen, die Verklemmung zu vermeiden, indem man darauf achtet, daß immer mindestens eine der obigen vier Bedingungen nicht erfüllt ist (Verklemmungsvermeidung durch Regeln).

- Man kann versuchen, die Verklemmung zu vermeiden, indem man die zukünftigen Betriebsmittelanforderungen der Prozesse analysiert und Zustände verbietet, die zu Verklemmungen führen können (Verklemmungsvermeidung durch Bedarfsanalyse).
- Man kann versuchen, eine Verklemmung festzustellen und sie, sollte sie eingetreten sein, dann zu beseitigen (Verklemmungserkennung).

Im folgenden soll nun untersucht werden, auf welche Weise **Synchronisationsmaßnahmen** reaisiert werden *können.*

11.4.2 Schloßvariable

Um den Zugang zu kritischen Abschnitten zu regeln, versieht man diese mit einem „Schloß", das von Prozessen aufgesperrt und verriegelt werden kann. Programmtechnisch wird ein solches Schloß durch eine **Schloßvariable** *(lock variable),* auch **Mutex** (von: *mutual exclusion)* genannt, realisiert.

Beim Betreten eines kritischen Abschnitts geht ein Prozeß folgendermaßen vor: Er wartet so lange, bis das zugehörige Schloß offen ist, dann betritt er den Abschnitt und verschließt das Schloß (quasi von innen), so daß ihm kein anderer Prozeß folgen kann. Diese Operation nennt man *lock* (verschließen). Hat er den kritischen Abschnitt beendet, schließt der Prozeß das Schloß wieder auf. Diese Operation wird *unlock* (aufschließen) genannt. Eine unlock-Operation muß von demselben Prozeß ausgeführt werden, der das Schloß vorher geschlossen hat.

Eine Schloßvariable ist ein abstrakter Datentyp, der aus einer booleschen Variablen vom Typ *mutex* und mindestens aus den zwei Operationen *lock* und *unlock* besteht.

In der Variante der POSIX-Threads besitzt eine Schloßvariable den Typ pthread mutex~t, und es sind folgende Funktionen definiert:

```
int pthread~mutex~init(pthread_mutex_t *mutex,
                        const pthread~mutexattr_t *attr);
```

Die Funktion initialisiert die Schloßvariable mutex mit dem Attribut eines spezifischen Mutex-Attributobjekts. Falls attr als NULL gewählt wird, so werden die Default-Attribute eingesetzt. Eine Schloßvariable muß vor ihrem ersten Gebrauch initialisiert werden.

```
int pthread mutex~destroy(pthreadmutex_t *mutex);
```

Diese Funktion löscht die Schloßvariable mutex.

```
int pthread_mutex~lock(pthread_mutex_t *mutex);
```

Diese Funktion schließt die Schloßvariable mutex. Falls der Mutex bereits geschlossen ist, wird der aufrufende Thread blockiert, bis der Mutex wieder freigegeben wird.

```
int pthread_mutex~trylock(pthread~mutex_t *mutex);
```

Diese Funktion versucht, die Schloßvariable mutex zu schließen. Falls der Mutex bereits geschlossen ist, wird der aufrufende Thread nicht blockiert, sondern fährt fort ohne einzutreten. Falls der Mutex frei ist, wird er gesperrt.

```
int pthread~mutez_unlock(pthread_mutex_t *mutex).;
```

Diese Funktion öffnet die Schloßvariable mutex. Es hängt von der Thread-Scheduling-Strategie des Betriebssystems ab, welcher der blockierten Threads als erster wiederaufgenommen wird. Der wieder aufgenommene Thread wiederholt die *lock-Op*eration,

doch ist nicht sicher, daß er als nächster durchkommt und in den kritischen Bereich eintreten darf. Denn in der Zeit zwischen der Wiederaufnahme und der erneuten Ausführung der lock-Operation kann ihm ein anderer (noch nicht blockierter) Thread mit seiner lock-Operation zuvorkommen.

Die lock-Operation besteht aus den zwei Operationen „Prüfen" und „Setzen" der Schloßvariablen. Die Implementierung eines lock-Befehls muß selbst wieder eine nicht unterbrechbare Aktivität sein, welche die Prüfen- und die Setzen-Operationen umfaßt. Bei der Implementierung benötigt man dafür spezielle Maschinenbefehle, die nicht unterbrochen werden können. Der auf vielen Rechnern vorhandene Maschinenbefehl test_and_Set (ab) liest den Wert der booleschen Variablen a, kopiert ihn nach b und setzt a auf *true* in einer unteilbaren Aktivität. Eine Alternative für die Implementierung eines lock-Befehls ist der swap-Befehl. Dieser tauscht den Inhalt eines Registers und eines Speicherplatzes, ohne daß er während seiner Ausführung unterbrochen werden kann. Ein lock-Befehl kann mit Hilfe des swap-Befehls folgendermaßen implementiert werden (symbolische Assemblernotation):

```
; Speicherzelle 4712 enthält die Mutex-Variable, die mit 1 (Mutex offen) initialisiert sein
muß
...     vor Eintritt in den   kritischen Bereich (lock-Operation)
        ld Ri,O             ; lade 0 ins Register Ri
m:      swap Ri, [4712]      ; vertausche Inhalt Ri mit Inhalt der Speicherzelle 4712
                            ; die die MutexVariable enthält
        bz Ri,m              ; falls die Speicherzelle mit 0 belegt war (Mutex geschlossen
                             ; dann springe nach Marke m und wiederhole die Aktion

;     Speicherzelle war mit 1 belegt (Mutex offen)
...     Eintritt in den kritischen Bereich

...     am Ende des kritischen Bereichs <unlock-Operation)
        ld Ri,l             ; lade 1 ins Register Ri
        st Ri, [4712]       ; speichere Inhalt Ri in die Speicherzelle 4712,
                            ; d.h. der Mutex ist wieder offen
```

Falls der kritische Bereich durch einen anderen Prozeß oder Thread belegt ist, wird der swap-Befehl solange wiederholt, bis der kritische Bereich freigegeben wird. Mutex-Implementierungen, die mittels eines solchen aktiven Wartens implementiert sind, werden auch als *Spin locks* bezeichnet. Die Implementierung ist besonders schnell, kann aber durch das aktive Warten Systemressourcen, d.h. insbesondere den Prozessor und den Systembus, unnötig stark belasten. Es ist von der Implementierung her klar, daß im Falle mehrerer wartender Prozesse (oder Threads), bei der Freigabe des kritischen Bereichs ein zufälliger Prozeß (oder Thread) den Zuschlag erhält.

Eine Mutex-Implementierung wird als *Suspend lock* bezeichnet, wenn das Betriebssystem die wartenden Prozesse oder Threads verwaltet und erst dann wieder einen der wartenden Prozesse oder Threads aktiviert, wenn der kritische Bereich freigegeben wird.

11.4.3 Semaphor

Ein **Semaphor** (engl. *semaphore)* ist der Bedeutung des Wortes nach ein Signal, das vor der Einführung des Telegrafendienstes vor allem in der Seefahrt zur optischen Übermittlung von Nachrichten über große Entfernungen verwendet wurde. Dijkstra hat den Begriff des Semaphors gewählt, um ein Hilfsmittel zur Synchronisation paralleler Prozesse zu bezeichnen.

Ein Semaphor S ist ein abstrakter Datentyp, der aus einer nichtnegativen Integer-Variablen (dem Semaphorzähler) und zwei Operationen besteht, die traditionell mit P und V bezeichnet werden. Diese Kürzel stammen aus dem Holländischen und stehen für „passeeren" (passieren) und „vrijgeven" (freigeben, im Deutschen auch „verlassen", um die Mnemonik nachzubilden). Nachdem bei der Initialisierung des Semaphors S dem Semaphorzähler einmal ein Wert

zugewiesen worden ist, kann er nur noch mit den zwei primitiven Operationen P(S) und V(S) manipuliert werden. Diese Operationen sind folgendermaßen definiert:

```
P(S):  Wenn S>0,
       dann S:=S-1,
       sonst wird der Prozeß, der P(S) ausführt, suspendiert (in den
       Wartezustand versetzt).
V(S):  S:=S+1.
```

P und V sind wieder unteilbare Operationen, d.h., bei ihrer Implementierung auf einem Rechner darf die Ausführung einer dieser Operationen nicht unterbrochen werden. Auf der Hardware-Ebene wird der Semaphormechanismus - ähnlich wie die Synchronisation mittels einer Schloßvariablen - durch spezielle Maschinenbefehle unterstützt, die dazu benutzt werden, die P- und V-Operationen zu implementieren.

Falls ein wartender Prozeß existiert, wird er nach dem Ausführen einer V-Operation erweckt. Die Definition von V gibt nicht an, welcher Prozeß erweckt wird, falls mehrere Prozesse auf das gleiche Semaphor warten. Dies muß durch das Betriebssystem gesteuert werden und kann beispielsweise bedeuten, daß ein beliebiger Prozeß erweckt wird (wie z.B. im Fall einer Busy-waiting-Implementierung der suspendierten Prozesse), daß immer ein bestimmter Prozeß erweckt wird, oder daß die Prozesse gemäß einer Warteschlange verwaltet werden. Nach dem Erwecken führt ein Prozeß die (unteilbare) P-Operation erneut aus.

P und V sind - abgesehen von der Initialisierung - wirklich die einzigen erlaubten Operationen bezüglich der Semaphorvariablen S. Insbesondere sind Zuweisungen an das Semaphor S verboten.

Semaphore, die nur die Werte Null und Eins annehmen können, heißen **binäre** Semaphore. Sie ähneln in ihrer Funktionsweise den Schloßvariablen mit dem Unterschied, daß bei Schloßvariablen eine *unlock-Operation* nur von dem Prozeß ausgeführt werden darf, der zuvor die lock-Operation durchgeführt hat. Diese Zusatzbedingung ist für die P- und V-Operationen eines Semaphors nicht gegeben, wie das Erzeuger-Verbraucher-Beispielprogramm (siehe später in diesem Abschnitt) zeigt.

Wenn Semaphore beliebige nichtnegative, ganzzahlige Werte annehmen können, heißen sie allgemeine Semaphoren.

Im Falle der mehrseitigen Synchronisation gibt der Initialwert des Semaphorzählers die maximale Anzahl der Prozesse an, die gleichzeitig im kritischen Abschnitt arbeiten können. Wenn ein kritischer Abschnitt nur exklusiv betreten werden darf, hat der Semaphorzähler den Initialwert Eins. P und V umschließen den kritischen Abschnitt genau wie *lock* und *unlock* bei der Synchronisation durch Schloßvariable.

In einer Prozedur, die einen kritischen Bereich enthält, wird vor dem Eintritt in diesen eine P(S)-Operation und nach dessen Verlassen eine V(S)-Operation gesetzt. Dies wird mit dem folgenden Programm demonstriert, das eine Lösung des Problems des gegenseitigen Ausschlusses mit Hilfe eines Semaphors zeigt

```
program gegenseitigerausachluß;
var s: (* binäres *) semaphore;

procedure pl;
begin
    repeat
        P(s);
        kritischerBereichl;
        V(s);
```

```
        Rest1
    forever
end;

procedure p2;
begin
    repeat
        P(s)
        kritischerBereich2;
        V(s);
        Rest2
    forever
end;


begin (* Hauptprogramm *)
    S: =1;
    cobegin  (* Start einer AnZahl paralleler Aktivitäten *)
        pl; p2
    coend    (* Beendigung der parallelen Aktivitäten *)
end.
```

Dieses Programm läßt sich leicht auf *n* Prozesse erweitern:

```
program gegenseitigerausschluß;
const n = ....;           (* Anzahl Prozesse ~)
var s: semaphore;

procedure process(i: integer);
begin
repeat
    P(s);
    kritischerBereich;
    V(s);
    Rest
    forever
end;

begin (* Hauptprogramm *)
s: =1;
    cobegin
        process(1);
        process(2);
        ....
        process (n);
    coend
end.
```

In diesem Fall ist jedoch eine Aussperrung möglich. Man nehme drei Prozesse und die folgende willkürliche Wahl unter den suspendierten Prozessen an: Es wird bei einer V-Operation immer der Prozeß mit dem kleinsten Index ausgewählt. Wenn sich dann pl und p2 jeweils wechselseitig erwecken, so könnte p3 unendlich lange aufgehalten werden.

Das folgende Programm gibt als Beispiel für ein Erzeuger-JVerbraucher-Problem die Lösung einer Pufferverwaltung mittels eines allgemeinen Semaphors wieder. Es gelte die Annahme, der Puffer ist beliebig groß und die Operationen legeab (im Puffer) sowie entnehme (aus dem Puffer) sind nicht unterbrechbare Operationen.

```
program erzeugerverbraucher;
var n: semaphore;

procedure erzeuger;
begin
    repeat
        erzeuge;
        legeah;
        V(n)
    forever
end;
```

```
procedure verbraucher;
begin
    repeat
      P(n)
      entnehme;
      verbrauche
    forever
end;

begin (~ Hauptprograrnrn *)
n: =0;
cobegin
erzeuger; verbraucher coend
end.
```

Das Semaphor kann als Differenz zwischen der Anzahl der von V erzeugten Signale und der Anzahl der von P verbrauchten Signale aufgefaßt werden.

Falls die Befehle `legeab` und `entnehme` selbst wieder kritische Bereiche sind, die sich nicht überlappen dürfen, kann das obige Programm leicht durch Hinzunahme eines weiteren binären Semaphors erweitert werden:

```
program erzeugerverbraucher;
var n: semaphore;
s: (* binäres *) semaphore;

procedure erzeuger;
begin
    repeat
        erzeuge;
        P(s);
        legeäb;
        V(s)
        V(n)
    forever
end;

procedure verbraucher;
begin
    repeat
        P(n);
        P(s);
        entnehme;
        V(s);
        Verbrauche;
    forever
end;

begin (* Hauptpropranun *)
    n: =0;
    s: =1;
    cobegin
        erzeuger; verbraucher
    coend
end.
```

Man beachte, daß sich ein Vertauschen der P-Operationen im Verbraucher-Programm fatal auswirken würde. Beispiel: Der Verbraucher führt erfolgreich P Is) aus (da ja anfänglich s=l gilt) und wird dann durch P (n) blockiert (da anfänglich n=0 gilt). Aber nun gilt s=l, und damit ist der Erzeuger nicht mehr in der Lage, etwas im Puffer abzulegen. Somit ist das System verklemmt.

Semaphore lösen Synchronisationsprobleme auf einer sehr niedrigen Ebene. Der Rauptnachteil des Semaphormechanismus ist, daß es leicht zu einem totalen Zusammenbruch des ge-

samten Systems kommen kann, wenn nur eine einzige SemaphorOperation falsch programmiert wird. Das läßt sich verhindern, wenn man Synchronisationswerkzeuge höherer Stufe wie den Monitor verwendet.

11.4.4 Bedingter kritischer Bereich

Bei der Methode des **bedingten kritischen Bereichs** oder des **bedingten kritischen Abschnitts** (region r) wird der Eintritt eines Prozesses in den Bereich (eine Anweisungsfolge s1, ... sn) davon abhängig gemacht, daß kein weiterer Prozeß bereits eingetreten ist. Weiterhin kann der Eintritt auch noch von der Erfüllung einer Bedingung b abhängig gemacht werden.

```
await b
region r
begin
    s1,....,sn
end
```

Wenn mehrere Prozesse versuchen, gleichzeitig in einen kritischen Bereich einzutreten, so kann nur einer erfolgreich eintreten, und die anderen müssen warten. Verläßt ein Prozeß einen kritischen Bereich, so kann ein anderer Prozeß eintreten.

Ist eine Bedingung b vorhanden, so wird diese zuerst berechnet. Wenn b „wahr" ist, dann kann der Prozeß in den Wettbewerb um den Eintritt in den kritischen Bereich einsteigen (und eintreten, falls der Bereich frei ist). Ist b „falsch", so muß sich der Prozeß zu den wartenden Prozessen einreihen. Sobald ein Prozeß den kritischen Bereich freigibt, werden alle wartenden Prozesse wieder aufgenommen. Die Prozesse mit einer Eintrittsbedingung müssen diese neu berechnen. Prozesse mit zu „wahr" ausgewerteten Bedingungen sowie Prozesse ohne Bedingung bewerben sich dann um den Eintritt in den kritischen Bereich. Die Implementierung sollte sicherstellen, daß dieser Wettbewerb fair verläuft.

Als Beispielprogramm wird im folgenden die Lösung des Problems des gegenseitigen Ausschlusses zweier Prozesse angegeben:

```
progratn gegenseit igerAussahlul,;
procedure pl;
begin
    repeat
        Restl;
        region r
        begin
            kritischerBereichl;
        end
    forever
end;

procedura p2;
begin
    repeat
       Rest2;
       region r
       begin
           kritischerBereich2;
       end
   forever
end;

begin (* Hauptprograrnrn *)
cobegin
   pl; p2
coend
end.
```

11.4.5 Bedingungsvariable

Bedingungsvariablen werden bei allen Thread-Konzepten angewandt. Die Thread-Bedingungsvariablen werden immer in Verbindung mit Mutex-Variablen eingesetzt. Das Sperren der Mutex-Variablen sichert den exklusiven Zugriff auf die Bedingungsvariable und auf andere gemeinsame Daten, die für die Auswertung einer Bedingung notwendig sind. Die wichtigsten Operationen mit Bedingungsvariablen sind ***Signal, Broadcast*** und ***Wait***

In der Variante der POSIX-Threads nach POSIX lOO3.lc-1995-Standard besitzt eine Bedingungsvariable den Typ pthread_cond_t, und es sind folgende Funktionen definiert:

11.4.5.1 pthread_cond_init

Die Funktion initialisiert die Bedingungsvariable `cond` mit dem Attribut eines spezifischen Bedingungsvariablen-Attributobjekts. Falls `attr` als NULL gewählt wird, so werden die Default-Attribute eingesetzt. Eine Bedingungsvariable muß vor ihrem ersten Gebrauch initialisiert werden.

```
int pthread_cond_init  (pthread_cond_t *cond,
                         const pthread_condattr_t *attr);
```

11.4.5.2 pthread_cond_destroy

Diese Funktion löscht die Bedingungsvariable `cond`

```
int pthread_cond_destroy(pthread_cond~t *cond);
```

11.4.5.3 pthread_cond_wait

.Diese Funktion öffnet die Schloßvariable mutex und setzt den aufrufenden Thread in einen Wartezustand für die Bedingungsvariable `cond`. Beide Operationen geschehen in einer unteilbaren Aktion. Durch das implizite Freigeben der zugeordneten MutexVariablen wird anderen Threads ermöglicht, in den kritischen Bereich einzutreten. Sobald der wartende Thread durch eine ***signal-*** oder ***broadcast-Operation*** wieder aufgenommen wird, muß erst der Mutex wieder geschlossen werden, bevor der Thread weiterfahren kann.

```
int pthread_cond_wait(  pthread_cond_t *cond,
                        pthread_mutex_t *mutex);
```

11.4.5.4 pthread_cond_timedwait

Wie bei pthread_cond_wait öffnet diese Funktion die Schloßvariable mutex und setzt den aufrufenden Thread in einen Wartezustand für die Bedingungsvariable `cond`. Beide Operationen geschehen in einer unteilbaren Aktion. Der wartende Thread wird aufgrund einer ***signal-*** oder ***broadcast-Operation*** wieder aufgenommen oder dann, wenn die Systemzeit größer oder gleich `abstime` ist. Er muß sich dann wieder um den Mutex bemühen. Alles weitere geschieht wie beim Aufruf von pthread_ cond_wait.

```
int pthread cond_timedwait( pthread~cond_t *cond,
                            pthread~mutex_t *mutex,
                            const struct timespec *abstime);
```

11.4.5.5 pthread_cond_signal

```
int pthread_cond_signal(pthread_cand_t *cond>;
```

Die Signal-Operation weckt einen der Threads auf, die für die Bedingungsvariable in den Wartezustand gesetzt wurden. Die Betriebssystem-Scheduling-Strategie bestimmt, welcher Thread aufgeweckt wird. Ein aufgeweckter Thread bemüht sich automatisch erst wieder um den Erhalt der Mutex-Variable, bevor er in den kritischen Bereich eintritt.

11.4.5.6 pthread_cond_broadcast

Die broadcast-Operation weckt *alle* Threads auf, die für die Bedingungsvariable in den Wartezustand gesetzt wurden. Jeder der aufgeweckten Threads bemüht sich automatisch erst wieder um den Erhalt der Mutex-Variable, bevor er in den kritischen Bereich eintreten kann.

```
int pthread_cond_broadcast(pthread_cand_t *cond);
```

Die POSIX-Bcdingungsvariablen ermöglichen in Verbindung mit Mutex-Variablen die Realisierung von bedingten kritischen Abschnitten. Dabei wird der kritische Abschnitt wie bisher mit einer Mutex-Variablen geschützt. Zusätzlich kann jedoch im kritischen Abschnitt noch eine Bedingung ausgewertet werden. Ist die Bedingung erfüllt, dann wird der kritische Abschnitt weiter ausgeführt. Ist die Bedingung jedoch nicht erfüllt, dann wird der Thread durch Aufruf von pthread_cond_wait (*cond, *mutex) blockiert und die Mutex-Variable automatisch wieder freigegeben (siehe das nachfolgende Codestück für das Zusammenspiel von Mutex- und Bedingungsvanablen).

```
pthread~mutex_lock(&mutex);          /* exklusiven Zugriff sichern */
while (bedingung == FALSE)            /* hier kann die spezifische  */
                                     /* Bedingung eingesetzt werden */
    pthread_cond_wait(&cond,&mutex);
pthread~mutex_unlock(&mutex);        /* exklusiven Zugriff freigeben */
```

Blockierte Threads müssen so lange warten, bis von einem anderen Thread signalisiert wird, daß sich die Bedingung geändert hat. Dieses Signalisieren kann, ähnlich wie beim Monitorkonzept, durch eine signal-Operation auf die Bedingungsvariable geschehen, die nur einen der blockierten Threads aufweckt, oder durch eine *broad*cast-Operation, die alle (in Zusammenhang mit dieser Bedingungsvariablen) blockierten Threads aufweckt. Das folgende Programmstück demonstriert die Verwendung einer signal-Operation *(broadcast* analog).

```
pthread~mutex_lock(&mutex);           /* exklusiven Zugriff sichern */
  .....
      modify bedingung;                /* hier kann die spezifische      */
                                      /* Bedingung geaendert werden     *1
      pthread~cond_signal (&cond);
     .....
pthread~mutex_unlock(&rnutex);/* exklusiven Zugriff freigeben */
```

Die Threads werden beim Konzept der POSIX-Bedingungsvariablen in beliebiger Reihenfolge aufgeweckt - im Gegensatz zur FIFO-Reihenfolge bei den Monitor-Bedingungsvariablen. Auch muß die ***signal-*** oder ***broadeast-Operation*** nicht direkt vor dem Verlassen der Programmeinheit stehen, wie im Falle eines Monitors gefordert ist. Das Verlassen des kritischen

Bereichs muß ein Pthread immer explizit z.B. mit pthread_mutex_unlock (&mutex) anzeigen. Erst dann kann einer der aufgeweckten Threads in den kritischen Bereich eintreten.

11.5Posix-Thread Beispiel in C

```
#include <iostream.h>
#include <pthread.h>
#include <pthread_exception.h>
#include <lib$routines.h>
#include <errno.h>
#include <time.h>
#include <string.h>
#include <math.h>

extern "C" int mysource(int);
extern "C" int mysource2(int);
extern "C" int RQX_InitXInfo(int);

const  int workers = 3;
const  int maxThreads = 16;
int    thrcount    = 0;
int    thread_hold = 1;
pthread_mutex_t   SlowMutex;
pthread_mutex_t   WeckMutex;
pthread_mutex_t   CountMutex;
pthread_cond_t WeckCond;
pthread_cond_t CountCond;

static void
unlock_cond (void * arg)
{
    int status;
    cout<<int(arg)<<" AnzOwner vor Unlock "<<WeckMutex.owner<<endl;
    status = pthread_mutex_unlock (&WeckMutex);
    cout<<int(arg)<<" AnzOwner nach Unlock "<<WeckMutex.owner<<endl;
    if (status !=0)
        cout<<int(arg)<<" "<<status<<"Mutex unLock failure"<<endl;
    else
       cout<<int(arg)<<" Mutex ist freigegeben für"<<int(arg)<<endl;
}

void *req_thread1(void *thrnum)

{
    int a =(int) thrnum;
    time_t timesek = time(NULL);
    tm *zeitp;
    int status;

    time (&timesek);
    zeitp     = localtime(&timesek);
    cout<<a<<" thread "<<a<<" um:"<<zeitp->tm_hour<<":"<<zeitp->
             tm_min<<":"<<zeitp->tm_sec<<endl;

    status = mysource (a);
    cout<<a<<" Fertig!..thread "<<a<<endl;
    return thrnum;
}
void *req_thread2(void *thrnum)

{
    time_t timesek = time(NULL);
    tm *zeitp;

    int a =(int) thrnum;
```

```
    intstatus;

    time (&timesek);
    zeitp     = localtime(&timesek);
    cout<<a<<" thread "<<a<<" um:"<<zeitp->tm_hour<<":"<<zeitp->tm_min<<":"<<zeitp-
>tm_sec<<endl;

    status = mysource2(a);
    cout<<a<<" Fertig!..thread "<<a<<endl;
    return thrnum;
}

void *test_thread3(void *thrnum)

/* Thread wird in MAIN gestartet und führt Berechnungen mit */
/* vielen Iterationsschritten durch. Getestet wird die        */
/* Prioriätssteuerung (Scheduling) dieses Threads            */
/* im Verhältnis zu den konkurrierenden Threads.                */
{
    double  x;
    double  y = 1.0;
    int     count1 = 1;
    int     count= 3;
    int     i;
    time_t timesek = time(NULL);
    tm *zeitp;

    int a =(int) thrnum;
    intstatus;

    time (&timesek);
    zeitp     = localtime(&timesek);
    cout<<a<<" thread "<<a<<" um:"<<zeitp->tm_hour<<":"<<zeitp->tm_min<<":"<<zeitp-
>tm_sec<<endl;

    for (count1 = 1; count1 <=2; count1++)
    {
       for (count = 1; count <=999999; count++)
       {
           y = 1.0;
           x = sin(float(count))    ;
           for (i =3; i<=5; i++)
           {
              y = y+(y*(i+1)/(i));
           }
           if(count % 10000 == 0)
              {
              cout<<a<<" x " <<x<<" count "<<count<<endl;
              cout<<a<<" y in Thread Typ 3 "<<y<<endl;
              }
       }
    }
    cout<<a<<" y in Thread Typ 3 "<<y<<endl;
    cout<<a<<" Fertig!..thread "<<a<<endl;
    return thrnum;
}
void *test_thread2(void *thrnum)

/* Endlos-Thread sitzt in ConditionWait und wird von anderen Threads        */
/* via SIGNAL geweckt um dann irgendeinen Unsinn zu machen                  */
/* Zusätzlich:Sperren und Freigeben einer globalen Variablen*/
/* mittels    Mutex Lock und Unlock                             */
/* Beenden des Endlos-Threads via PTHREAD_CANCEL in MAIN   */
/*                                                                             */
/* Verwendete Prozeduren:                                              */
/*                                                                             */
/*      pthread_mutex_init   Initialisierung der Mutexvariablen   */
/*                                 für pthread_cond_wait              */
/*      pthread_cond_wait    Aufwachen des Threads mit Mutex      */
```

```
/*      pthread_mutex_lock   WeckMtx locken für pthread_cond_wait*/
/*      pthread_mutex_unlock WeckMtx freigeben nach                    */
/*                                  pthread_cond_wait                          */
{
    double count= 3.0;
    double x;
    double y;

    int     a =(int) thrnum;
    int     i;
    int     status;

    char    *mytimep;
    pthread_mutex_t   WaitMutex;
    time_t timesek  = time(NULL);
    tm      *zeitp;

    pthread_mutex_init(&WaitMutex, NULL);

    status = 0;
    while (status)
    {
       x = sin(count)        ;
       count++;
       cout<<a<<" x " <<x<<" count "<<count<<endl;

       time (&timesek);
       zeitp  = localtime(&timesek);
       cout<<a<<" thread "<<a<<" um:"<<zeitp->tm_hour<<":"<<zeitp->tm_min<<":"<<zeitp-
>tm_sec<<endl;

        cout<<a<<" Warte auf Condition für thread "<<a<<endl;

       cout<<a<<" Locke WaitMutex für thread "<<a<<endl;
       status = pthread_mutex_lock(&WaitMutex);
       if (status !=0)
            cout<<a<<" "<< status<<"WaitMutex Lock failure"<<endl;
       else
           cout<<a<<" WaitMutex ist gelockt für thread "<<a<<endl;

       status = pthread_cond_wait(&CountCond, &WaitMutex);
       if (status !=0)
            cout<<a<<" "<<status<<"Cond_Wait failure"<<endl;
       else
           cout<<a<<" Warte auf Condition für thread "<<a<<endl;
       status = pthread_mutex_unlock(&WaitMutex);
       if (status !=0)
            cout<< status<<" WaitMutex UnLock Failre im Thread Typ 2"<<endl;
       else
           cout<<a<<" WaitMutex unlocked in Thread Typ 2"<<endl;


       time (&timesek);
       zeitp  = localtime(&timesek);
       cout<<a<<" abgelaufen um:"<<zeitp->tm_hour<<":"<<zeitp->tm_min<<":"<<zeitp-
>tm_sec<<endl;


/* Bereich der globalen Variablen thrcount schuetzen                         */

       status = pthread_mutex_lock(&CountMutex);
       if (status !=0)
            cout<< status<<"Count Mutex Lock Failure"<<endl;
       else
           cout<<a<<" CountMutex ist gelockt für thread Typ 2 "<<a<<endl;

       thrcount++;

       for (i =2; i<=3; i++)
```

```
            y = y+(y*(i+1)/(i-1));
        cout<<a<<" y in Thread Typ 2"<<y<<endl;
        cout<<a<<" Absolute Anzahl in Thread Typ 2 "<<thrcount<<endl;
        status = pthread_mutex_unlock(&CountMutex);
        if (status !=0)
             cout<< status<<" CountMutex UnLock Failre im Thread Typ 2"<<endl;
        else
            cout<<a<<" CountMutex unlocked in Thread Typ 2"<<endl;


    }
    cout<<a<<" Fertig!..thread"<<a<<endl;
    return thrnum;
}
void *test_thread1(void *thrnumadr)

/* Thread wird mehrfach (Anzahl= workers), konkurrierend in MAIN erzeugt    */
/* und mittels Broadcast geweckt.                                          */
/* Verwendete Prozeduren:                                          */
/*      pthread_cond_wait    Warten auf Ereignis und          */
/*                    Aufwachen des Threads mit Mutex         */
/*      pthread_cond_timedwait      Warten auf Ereignis mit Timeout*/
/*                            (verwendbar als Timer bei Threads)      */
/*      pthread_cleanup_push Aufsetzen eines Exithandlers für */
/*                                   einen Thread (Cleanuphandler) */
/*    pthread_cleanup_pop Ausführen eines Exithandlers      */
/*      pthread_get_expiration_np Bestimmung der Ablaufzeit des*/
/*                              Timers      indem zur Systemzeit*/ /*                    ein
Delta in Sekkunden addiert wird */
/*      pthread_mutex_lock   WeckMtx locken für pthreadcond_wait*/

{
    char       *mytimep;
    EXCEPTION  except;

    int*            ap = (int*) thrnumadr;
    int            a= *ap;
    int         error;
    int         index = 0;
    int         status;
    tm          zeit;
    tm          *zeitp;
    time_t      timesek = time(NULL);
    timespec    waittime;
    timespec    reswaittime;
    unsigned long excode;

    cout<<a<<&a<<&error<<" Thread startet zum "<<a<<".mal"<<endl;

    waittime.tv_sec  = 1;
    waittime.tv_nsec = 0;

/* Synchronisierung der Threads mit dem Hauptprogramm mittels einer          */
/* Condition Variablen                                                       */

    cout<<a<<" Locke Mutex für thread "<<a<<endl;
    cout<<a<<" AnzOwner vor Lock "<<WeckMutex.owner<<endl;
    status = pthread_mutex_lock(&WeckMutex);
    cout<<a<<" AnzOwner nach Lock "<<WeckMutex.owner<<endl;
    if (status !=0)
        cout<< status<<"Mutex Lock"<<endl;
    else
       cout<<a<<" Mutex ist gelockt für thread "<<a<<endl;

    cout<<a<<" Mutex freigeben (Cleanup) für thread "<<a<<endl;
    pthread_cleanup_push(unlock_cond, (void*)a);
    cout<<a<<" Mutex freigegeben (Cleanup) für thread "<<a<<endl;

    cout<<a<<" Thread_hold = "<<thread_hold<<endl;
```

```
    while (thread_hold)
    {
         cout<<a<<" Warte auf Condition für thread "<<a<<endl;
         cout<<a<<" Thread_hold = "<<thread_hold<<endl;
        cout<<a<<" AnzOwner vor CondWait "<<WeckMutex.owner<<endl;
        status = pthread_cond_wait(&WeckCond, &WeckMutex);
        cout<<a<<" AnzOwner nach CondWait "<<WeckMutex.owner<<endl;
        if (status !=0)
             cout<< status<<"Cond_Wait"<<endl;
        else
            cout<<a<<" Warte auf Condition für thread "<<a<<endl;

        pthread_cleanup_pop (1);

    }
/* Timer definieren  */
    status=pthread_get_expiration_np (&waittime,&reswaittime);
    if (status !=0)
        {
         cout<< status<<"get_expiration_np failure "<<endl;
        }
    for (index = 0; index <= 3; index ++)
    {
        status = pthread_mutex_lock(&CountMutex);
        if (status !=0)
             cout<<a<< status<<"CountMutex Lock failure für thread "<<a<<endl;
        else
            cout<<a<<" CountMutex ist gelockt für thread "<<a<<endl;

        thrcount++;
        status = pthread_mutex_unlock(&CountMutex);
        if (status !=0)
            cout<<a<< status<<"CountMutex UnLock im Thread failure"<<a<<endl;
        else
            cout<<a<<" CountMutex UnLock im Thread "<<a<<endl;

/* Für sauberes Arbeiten des Condition Wait muss der                          */
/* Mutex vorher gelockt werden.                                               */

        cout<<a<<" AnzOwner vor Lock "<<WeckMutex.owner<<endl;
        status = pthread_mutex_lock (&WeckMutex);
        cout<<a<<" AnzOwner nach Lock "<<WeckMutex.owner<<endl;
        if (status !=0)
             cout<<a<< status<<"Mutex Lock vor Wait "<<a<<endl;
        else
            cout<<a<<" WeckMutex Lock vor WAIT im Thread "<<a<<endl;


        time (&timesek);
        cout<<a<<" "<<"Absolute Anzahl "<<thrcount<<endl;
        cout<<a<<" Timer aufgesetzt für thread "<<a<<" Lauf "<<index<<endl;
        zeitp  = localtime(&timesek);
        cout<<a<<" aufgesetzt um:"<<zeitp->tm_hour<<":"<<zeitp->tm_min<<":"<<zeitp-
>tm_sec<<endl;
        cout<<a<<" Ende erwartet um "<<reswaittime.tv_sec<<":"<<reswaittime.tv_nsec<<endl;

        cout<<a<<" AnzOwner vor CondTimewait "<<WeckMutex.owner<<endl;
        status = pthread_cond_timedwait(&WeckCond, &WeckMutex,
        &reswaittime);
        cout<<a<<" AnzOwner nach CondTimewait/vor Unlock "<<WeckMutex.owner<<endl;

        error = pthread_mutex_unlock (&WeckMutex);
        cout<<a<<" AnzOwner nach Unlock "<<WeckMutex.owner<<endl;
        if (error !=0)
             cout<<a<< error<<"Mutex UnLock nach Wait "<<a<<endl;
        else
            cout<<a<<" WeckMutex Unlock nach WAIT im Thread "<<a<<endl;

        time (&timesek);
```

```
        zeitp  = localtime(&timesek);
        cout<<a<<" abgelaufen um:"<<zeitp->tm_hour<<":"<<zeitp->tm_min<<":"<<zeitp-
>tm_sec<<endl;
        switch (status)
            {
            case (0) :
               cout<<a<<" "<< status<<" über Mutex/Condition rausgekommen"<<endl;
               break;
            case (ETIMEDOUT):
               mytimep = ctime(&timesek);
               cout<<a<<" "<<mytimep;
               cout<<a<<" "<<status<<" abgelaufen thread "<<a<<" Lauf "<<index<<endl;
               break;
            default :
               EXCEPTION except;
               EXCEPTION_INIT (except);
               excode = (unsigned long) status;
               pthread_exc_set_status_np (&except, excode);
               pthread_exc_report_np (&except);
            }
        status = pthread_get_expiration_np (&waittime,&reswaittime);
        status = pthread_cond_signal(&CountCond);
        if (status !=0)
             cout<<" signal failure"<<status<<endl;
        else
            cout<<a<<" Signal für Sinus Berechnung von thread "<<a<<endl;

    }
    cout<<a<<" Fertig....thread"<<a<<endl;
    return (void*)a;
}

void main()
/* Thread Testprogramm                                                   */
/* Programm kreiert und startet mehrere Threads:                         */
/* - einThread  wird mehrmals kreiert (Master-Worker) und gestartet.     */
/*   Diese Workerthreads arbeiten periodisch mit Hilfe eines Timers.     */
/*   Sie stoßen wiederum alle den selben Thread an                       */
/* - Thread2 ist der mittels pthread_cond_signal von einThread ange-     */
/*   stossene Thread, der in einer Endlosschleife mit pthread_cond_wait  */
/*   auf Aufträge wartet. Er muss vom Hauptprogramm mit pthread_cancel   */
/*   explizit beendet werden                                             */
/* - Thread3 läuft als unabhängiger Thread und führt eine Anzahl Ite-    */
/*   rationen aus. Für ihn werden Scheduling-Parameter (Priorität)       */
/*   gesetzt.                                                            */
/* - Das Programm endet erst, wenn alle Threads abgearbeitet sind.       */
/*   Geprüft wird dies mit pthread_join.                                 */
/*                                                                       */
/* Verwendete Prozeduren:                                                */
/*                                                                       */
/*      pthread_cond_init    Conditionvariable erzeugen und mit          */
/*                           Defaultwert initialisieren                  */
/*      pthread_cond_broadcast      Alle Thread, die auf eine bestimmte  */
/*                           Condition-Variable hören, benachrichtigen   */
/*      pthread_cond_signal  Einen Thread, der auf eine bestimmte        */
/*                           Condition-Variable hört, benachrichtigen    */
/*      pthread_create              Einen Thread erzeugen                */
/*      pthread_join         Synchronisieren mehrerer Threads            */
/*      pthread_mutex_init   Mutexvariable erzeugen und mit              */
/*                           Defaultwert initialisieren                  */
/*      pthread_mutex_lock   Mutex locken vor broadcast                  */
/*      pthread_mutex_unlock Mutex unlocken nach broadcast               */
/*      pthread_setschedparam       Scheduling Verfahren und Priorität   */
/*                           für einen Thread festlegen                  */
{
    int        index = 0;
    int        status =0;
    int        thrnum = 1;
    int          thrNumArr [maxThreads];
```

```
    int         x=1;
    size_t      ThreadStackSize = (100*8192);
//    size_t    ThreadStackSize = 2147500000;
    void        *exitval;
    float       time ;


    pthread_t einThread [workers+1];
    pthread_t Thread2;
    pthread_t Thread3;
    pthread_t ReqThread1;
    pthread_t ReqThread2;
    pthread_attr_t s_gl_pthread_attr;       // Address struct for init. thread
    pthread_attr_t s_gl_pthread_attr2;      // Address struct for init. thread
    sched_param       schedparam;


    RQX_InitXInfo (thrnum);

    schedparam.sched_priority = PRI_FG_MIN_NP;     /* 16 -> 31*/
    cout<< "schedparam.sched_priority"<<schedparam.sched_priority;

/*    schedparam.sched_priority = sched_get_priority_min (SCHED_OTHER);*/
    status = sched_get_priority_min(8);
    status = sched_get_priority_min(SCHED_OTHER);

    cout<< "schedparam.sched_priority"<<schedparam.sched_priority;

    status = pthread_mutex_init(&CountMutex, PTHREAD_MUTEX_DEFAULT);
    status = pthread_mutex_init(&WeckMutex, PTHREAD_MUTEX_DEFAULT);
    cout<<" Main AnzOwner nach Init "<<WeckMutex.owner<<endl;
    if (status !=0)
        cout<< status<<"mutex_init failure "<<endl;

/* Thread Condition Varibale initialisieren*/

    status = pthread_cond_init(&WeckCond, NULL);
    if (status !=0)
        cout<< status<<"cond_init failure "<<endl;
    status = pthread_cond_init(&CountCond, NULL);
    if (status !=0)
        cout<< status<<"Count_cond_init failure "<<endl;


    time = 1.0;
    status = pthread_attr_init(&s_gl_pthread_attr);
    status = pthread_attr_init(&s_gl_pthread_attr2);
    if (status != 0 )
    {
       perror("init thread FAILURE ");
       exit(EXIT_FAILURE);
    }

    status = pthread_attr_setstacksize(&s_gl_pthread_attr, ThreadStackSize);
    if (status != 0 )
    {
       perror("set stack size of thread FAILURE ");
       exit(EXIT_FAILURE);
    }
    status = pthread_attr_setstacksize(&s_gl_pthread_attr2, ThreadStackSize);
    if (status != 0 )
    {
       perror("set stack size of thread FAILURE ");
       exit(EXIT_FAILURE);
    }
    while(index <= workers)
    {
       cout<<"main programm kreiert Thread Nr.  "<<thrnum<<endl;
        thrNumArr[thrnum] = thrnum;
```

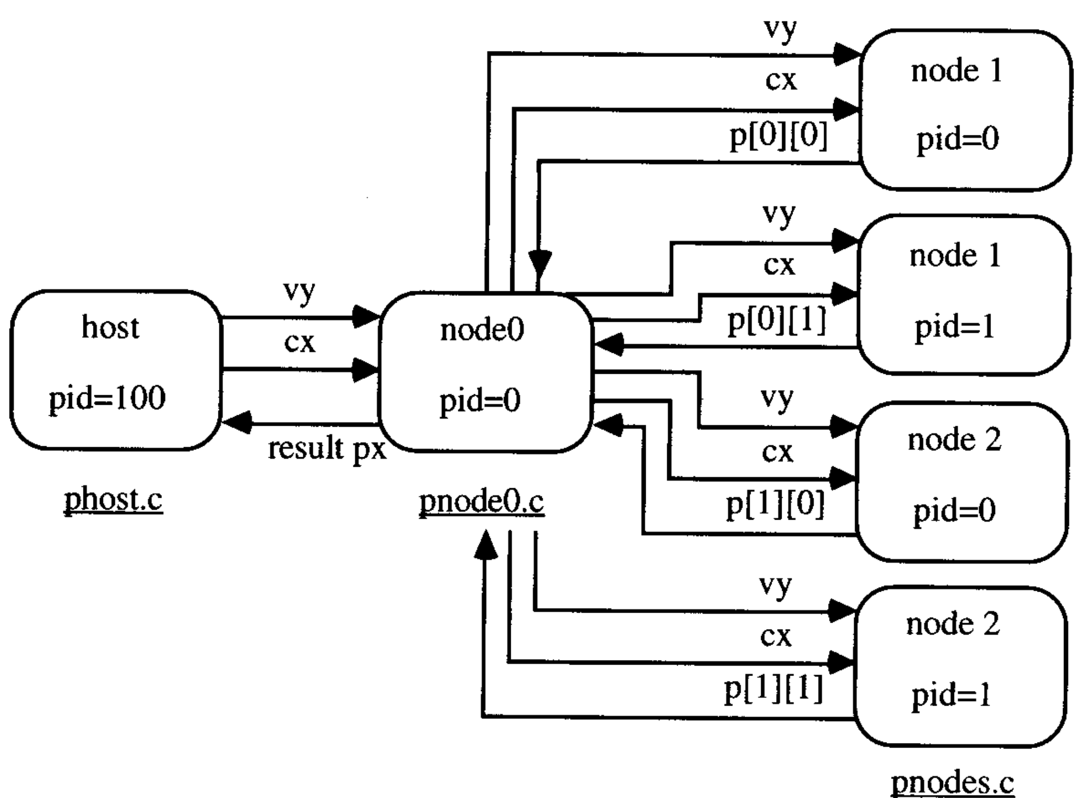
```
        status = pthread_create(&einThread[index], PTHREAD_CREATE_JOINABLE,
        &test_thread1, &thrNumArr[thrnum]);
        thrnum++;
        index++;
    }
        cout<<"main programm kreiert Thread Typ 2 mit Nr.  "<<thrnum<<endl;
        status = pthread_create(&Thread2, PTHREAD_CREATE_JOINABLE,
        &test_thread2, (void*) thrnum);

        thrnum++;
        cout<<"main programm kreiert Thread Typ 3 mit Nr.  "<<thrnum<<endl;
        status = pthread_create(&Thread3, PTHREAD_CREATE_JOINABLE,
        &test_thread3, (void*) thrnum);

        thrnum++;
        cout<<"main programm kreiert Requester Thread 1 mit Nr. "<<thrnum<<endl;
        status = pthread_create(&ReqThread1, &s_gl_pthread_attr,
        &req_thread1, (void*) thrnum);

        thrnum++;
        cout<<"main programm kreiert Requester Thread 2 mit Nr. "<<thrnum<<endl;
        status = pthread_create(&ReqThread2, &s_gl_pthread_attr2,
        &req_thread2, (void*) thrnum);



/*                                                                          */
/* Prioritäten der Threads festlegen                                        */
/* (der reine Rechenthread Thread 3 soll niedrigste Prioriät erhalten       */
/*                                                                          */
    status = pthread_setschedparam (Thread3, SCHED_OTHER, &schedparam);
    if (status !=0)
         cout<< status<<"Priorität Fehler "<<endl;
    else
        cout<<"Prioriät für Thread Typ 3 gesetzt"<<endl;

/*                                                                          */
/* Mittels Broadcast alle Threads der Variante einThread wecken             */
/*                                                                          */
    cout<<"Alle Threads kreiert"<<endl;
    cout<<" Main AnzOwner vor Lock "<<WeckMutex.owner<<endl;
    status = pthread_mutex_lock(&WeckMutex);
    cout<<" Main AnzOwner nach Lock "<<WeckMutex.owner<<endl;
    if (status !=0)
         cout<< status<<"Mutex Lock Main"<<endl;
    else
        cout<<"Mutex ist gelockt für Main "<<endl;
    thread_hold = 0;
    status = pthread_cond_broadcast (&WeckCond);
    if (status !=0)
         cout<<" broadcast failure"<<status<<endl;
    else
        cout<<"broadcast verschickt von main "<<endl;

    cout<<" Main AnzOwner vor Unlock "<<WeckMutex.owner<<endl;
    status = pthread_mutex_unlock(&WeckMutex);
    cout<<" Main AnzOwner nach Unlock "<<WeckMutex.owner<<endl;
    if (status !=0)
         cout<< status<<"Mutex UnLock"<<endl;
    else
        cout<<"Main unlocked Mutex "<<endl;

    cout<<"alle Threads aktiviert, Main wartet noch auf Rückmeldung "<<endl;
/*                                                                          */
/* Mittels Signal Thread2 wecken                                  */
/*                                                                          */

    status = pthread_cond_signal(&CountCond);
    if (status !=0)
```

```
        cout<<" signal failure"<<status<<endl;
    else
       cout<<"signal verschickt von main "<<endl;

//    status = mysource (x);


/* Ende aller Threads abwarten/oder herbeiführen                     */

    for (index = 0; index <= workers; index ++)
    {
       cout<<"Warte auf Ende von Thread Num "<<index+1<<endl;
       status = pthread_join(einThread[index],&exitval);
       if (status !=0)
            cout<< status<<" pthread_join"<<endl;
       cout<<"Ende von Thread Num "<<index+1<<"mit Status "<<exitval<<endl;
    }
    cout<<"Warte auf Ende von Thread Num "<<thrnum<<endl;
    status = pthread_join(Thread3,&exitval);
    if (status !=0)
        cout<< status<<" pthread_join"<<endl;
    cout<<"Ende von Thread Num "<<thrnum<<"mit Status "<<exitval<<endl;
    cout<<"Warte auf Ende von Thread Num "<<thrnum-1<<endl;
    status = pthread_cancel (Thread2);
    if (status !=0)
        cout<< status<<" pthread_cancel von Thread "<<thrnum-1<<endl;
    status = pthread_join(Thread2,&exitval);
    if (status !=0)
        cout<< status<<" pthread_join"<<endl;
    cout<<"Ende von Thread Num "<<thrnum-1<<"mit Status "<<exitval<<endl;

    cout<<"Ende"<<endl;
}
```

12 Nachrichtenorientierte Programmierung

12.1 Programmbeispiel Matrixmultiplikation

Das Programm multipliziert 2x2 Matrizen derart, daß jedes der vier Knotenprogramme genau ein Element der Resultatmatrix beisteuert. Das Programm ist auf einen Hypercubusrechner Intel iPSC/2 oder iPSC/860 zugeschnitten. Es werden der Einfachheit halber nur drei Knoten benutzt, doch läßt sich das Programm leicht skalieren. Das host-Programm läuft auf dem *System resource manager.* Es lädt das pnlode 0-Programm auf den Hypercube-Knoten 0 und das pnodes-Programm auf die Knoten 1 und 2, jeweils Prozeß 0 und 1, eines zweidimensionalen Hyperkubus. Die Kommunikationsstruktur ist in folgender Abbildung dargestellt (`pid` bedeutet Prozeßnummer, `vy` und `cx` sind die zu multiplizierenden Matrizen und px ist die Resultatmatrix).




```
//      host program
#include <cube.h>
main()

int k,i,j;
double px[2] [2],vy[2] [2],cx[2] [2],inbuf;
long outmsg_id, inmsg_id;

vy[01 [0]=1.0; vy[0] [1]=3.0;
vy[1] [0]=2.0; vy[1J [1]=4.0;
```

```
cx[0] [0]=5.0; cx[0] [l]=7.0;
cx[1] [0]=6.0; cx[1] [11=8.0;

setpid(1OO)
load("pnodeO",O,O);
load("pnodes",1,O);
load("pnodes",1,1);
load("pnodes",2,O);
load("pnodes',2,1);

// use of an asynchronous send Operation
outmsg_id=isend(0,vy, 4*sizeof(double) ,0,0);
if ( !msgdone (outmsg_id)) msgwait (outlmsg_id);

// use of a synchronous send Operation
csend(l,cx, 4*sizeof (double) ,0,0);

// synchronous receive operation
 crecv(-1,px,4*sizeof(double)); // recv result matrix from node 0

// pnode0 program for node 0
// receive 2 matrices from phost, distribute work to nodes,
// collect results from nodes, compress and send
// resultmatrix to phost

#include <cube.h>
main()
{
int k,i,j;
double px[2] [2],vy[2] )2],cx[2) [2],inbuf;
long outnsg_id, inmsg_id;

// synchronous receive Operation, matrices from phost
crecv(0,vy,4*sizeOf(double));
crecv(1,cx,4*sizeOf(double));

//     synchronous send Operations
k=2;
for(i=l;i<3;i++)          //      node 1 & 2
    for(j=0;j<2;j++)      //      process 0 & 1
{
csend(k,vy,4*sizeof(double) ,i,j);
k++;
csend(k, cx,4*sizeOf (double) ,i,j);
k++
}

i=0;
while (i< 4)
{
    cprobe(-1);       // wait until a message of any type arrives
                      // asyncbronous receive Operation
                      // receive message of any type
    inmsg_id=irecv(-l, &inbuf, sizeof (double));

    i++
                      // wait until receive Operation is completed
    msgwait(inmsg~id);

                      // infotype returns the type of the message
    switch ( infotype ())
       {
       case   10     : px[0][0]=inbuf; break;
       case   11     : px[0][l]=inbuf; break;
       case   20     : px[l][0]=inbuf; break;
       case   21     : px[l][l]=inbuf; break;
       }
}
```

```
// send result matrix to phost
csend(99,px,4*sizeOf(double) myhost() ,100);
}


// pnodes program for node 1&2 process 0 & 1

#inolude <oube.h>
main()
{
int k;
double res,vy[2] [2] ,cx[2] [2];
res=0. 0;

crecv(-1,vy,4*sizeof(double));
creov(-1,cx,4*sizeOf(double));

for ( k=0 ; k<2 ; k++)
   res =  vy[mynode()-1] [k] * cx[k] [mypid()];

csend(10*mynode()+ mypid() ,&res,sizeof(double) ,0,0);
}
```

Programmierung nachrichtengekoppelter Systeme:

- Im Gegensatz zur Programmierung speichergekoppelter Systemen (nahezu) beliebige Skalierbarkeit
- Größere Unabhängigkeit von der zugrundeliegenden Hardwarearchitektur! Auf allen MIMD-Systemen und sogar auf heterogenen Workstation-Clustern einsetzbar
- Aber, wie im obigen Beispiel zu sehen ist: viel " Handarbeit“ bei der Implementation der Kommunikation sowie viele proprietäre Programmierschnittstellen

12.2 Nachrichtengekoppelte Programmierschnittstellen

Man kann folgende nachrichtengekoppelte Programmierschnittstellen zur parallelen und verteilten Programmierung unterscheiden:

- maschinenabhängige: IBM's MPL (früher EUI genannt), Meiko's CS, nCUBE's PSE und Intels's NX-Lib
- maschinenunabhängige aus dem Kontext der Workstation-Cluster: EXPRESS, P4, PARMACS, PVM, und MPI (als standardisierte Schnittstelle); Spezialfälle: Sun RPC und DCE RPC

Einige der letztgenannten werden von Parallelrechner-Herstellern wieder maschinenabhängig (d. h. für einen speziellen Parallelrechner) implementiert.

12.3 Implementierung nachrichtengekoppelter Programme

Ein nachrichtengekoppeltes Programm besteht aus mehreren (sequentiellen) Programmen/Prozessen. Besonderes Augenmerk gilt bei der Programmentwicklung

- der Synchronisation zwischen den einzelnen Prozessen sowie
- der Organisation der Schreib-/Lese-Zugriffe eines einzelnen Prozessors auf die Speicher der anderen Prozessoren mittels der verfügbaren Message-Passing Befehle der Programmierschnittstelle.

Eine nachrichtengekoppelte Programmierschnittstelle setzt sich dabei aus einer sequentiellen Programmiersprache (C, Fortran) und einer Message-Passing-Bibliothek zusammen.

12.4 MPI Der Message Passing Interface Standard

Entwickelt seit Januar 1993 vom MPI-Forum unter Mitwirkung von ca. 60 Personen aus über 40 Organisationen aus Industrie, Forschungsinstituten und Universitäten aus Amerika und Europa. Nach vorhergehender Bekanntgabe der vorläufigen Spezifikation wurde die Version 1.0 Juni 1994 verabschiedet und veröffentlicht. Die Version 1.1 entstand durch Korrektur kleinerer Fehler und Präzisierung der Dokumentation und wurde im Juni 1995 herausgegeben. Die Version 2 des MPI Standards ist ein Versuch, MPI sinnvoll zu ergänzen (nicht zu ändern!). Ein Draft von MPI 2 wurde auf der " Supercomputing '96\-Tagung veröffentlicht, die endgültige Version ist im August 1997 erschienen. Entwickelt als portable, effiziente und flexible nachrichtengekoppelte Programmierschnittstelle. Es sollte ein weitverbreiteter Standard zur Entwicklung von SPMD/MIMD Message-Passing Programmen für Parallelrechner, heterogene Workstationcluster und heterogene Netzwerke geschaffen werden.

12.4.1 MPI Beispielprogramm : Berechnung von Pi

```
!*********************************************************************
!   fpi.f90 - compute pi by integrating f(x) = 4/(1 + x**2)
!
!  Variables:
!
!    pi          the calculated result
!    n           number of points of integration.
!    x            midpoint of each rectangle's interval
!    f            function to integrate
!    sum,pi  area of rectangles
!    tmp       temporary scratch space for global summation
!    i            do loop index
!*************************************************************************
program Main
  !
  ! use MPI
  !.. Implicit Declarations ..
  implicit none
  include 'mpif.h'
  !
  !.. Local Scalars ..
  integer, parameter :: master = 0
  integer :: i,n,myid,numprocs,ierr,rc
```

```
  double precision, parameter :: pi25dt = 3.141592653589793238462643d0
  double precision :: a,h,mypi,pi,sum,x
  !
  !.. Intrinsic Functions ..
  intrinsic ABS, DBLE
!.. Statement Function to integrate
  double precision :: f
  f(a) = 4.d0 / (1.d0+a*a)
     call MPI_INIT( ierr )                              !Initialisierung der MPI-Umgebung
     call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )  !Prozesskennung erfragen
call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )   ! Zahl der Geschwister erfragen    !
  ! ... Executable Statements ...
  do
     if ( myid .eq. 0 ) then
         write (UNIT=*, FMT='(''Enter the number of intervals: (0 quits)'')')
         read (UNIT = *, FMT = '( I10)')  n
    end if
    call MPI_BCAST(n,1,MPI_INTEGER,master,MPI_COMM_WORLD,ierr) ! N wird an alle geschickt
     if (n <= 0) exit                              ! check for quit signal
     h = 1.0d0 / n                                 ! calculate the interval size
     sum = 0.0d0
     do i = myid+1, n, numprocs                    !z.B. n=2, h = 0.5
         x = h * (DBLE(i)-0.5d0)                   ! X1 = 0.5*0.5 = 0.25,  x2=0.5*1.5=0.75
         sum = sum + f(x)                          !sum1 = 4/1+0.25*0.25)                =
3.7647,
      end do                                       !sum2 = 3.7647+4/(1+(0.75*0.75)) = 6.3247
      mypi = h * sum                                !mypi = 0.5* 6.3247  = 3.1624
    ! Für alle  mypi wird das arithmetische Mittel gebildet
    call MPI_REDUCE(mypi,pi,1,MPI_DOUBLE_PRECISION,MPI_SUM,master,MPI_COMM_WORLD,ierr)
    if (myid .eq. 0) then
         print*, 'Pi ist :',  pi,'Fehler ist:',  ABS(pi-pi25dt)
    endif
 end do
 call MPI_FINALIZE(rc)              !Freigeben der Systemresourcen end
```

12.5Datenparalle Programmierung: Open MP

OpenMP umfaßt eine Sammlung von Compiler-Direktiven, Bibliotheks-Funktionen und definierten Umgebungsvariablen zur Ausnutzung der parallelen Eigenschaften von **Shared-Memor**y-Maschinen. OpenMP ist für Fortran sowie C / C++ definiert. **Ziel:** Bereitstellung eines portablen, herstellerunabhängigen parallelen Programmiermodells.

Synchronisation: Verwendet das sogen. **fork-join**-Modell für die parallele Ausführung. Ein Programm startet als ein einzelner **Thread** , der **Master Thread**. Beim Erreichen des ersten parallelen Konstrukts erzeugt der Master Thread eine Gruppe von Theads und wird selbst Master der Gruppe. Nach Beendigung des parallelen Konstrukts synchronisiert sich die Gruppe, und nur der Master Thread arbeitet weiter. Die Parallelisierung erfolgt auf Schleifenebene.

Beispiel: Fortran-Unterprogramm, wie es häufig bei der numerischen Berechnung diskreter zweidimensionaler Probleme auftritt.

Es initialisiert zwei Felder für die diskrete numerische Lösung u(i,j) und die exakte Lösung f(i,j). Die Doppelschleife wird von den verf ügbaren Prozessoren parallel verarbeitet ***(parallel do),*** wobei die Indexbereiche f ür i und j verteilt werden. Die beiden Felder sowie die Größen n, m, dx, dy und alpha stehen allen Prozessen global zur Verfügung (werden ***geshared),*** die Größen i,j,xx,yy sind pro Prozeß lokal ***(private).***

12.5.1 OpenMP Beispielprogramm

```
subroutine initialize (n,m,dx,dy,u,f,alpha)
```

```
! Initializes data. Assumes exact solution is u(x,y) = (1-x^2)*(1-y^2)
implicit none
integer n,m
real u(n,m),f(n,m),dx,dy,alpha
integer i,j, xx,yy
dx = 2.0 / (n-1)
dy = 2.0 / (m-1)
!$omp parallel do
!$omp& shared(n,m,dx,dy,u,f,alpha)
!$omp& private(i,j,xx,yy)
do j = 1,m
    do i = 1,n
       xx = -1.0 + dx * (i-1) ! -1 < x < 1
       yy = -1.0 + dy * (j-1) ! -1 < y < 1
       u(i,j) = 0.0
       f(i,j) = -alpha *(1.0-xx*xx)*(1.0-yy*yy)- 2.0*(1.0-xx*xx)-2.0*(1.0-yy*yy)
    enddo
enddo!$omp end parallel do
return
end
```

```
!*********************************************************************
!   fpi.f90 - compute pi by integrating f(x) = 4/(1 + x**2)
!
!  Variables:
!
!    pi       the calculated result
!    n        number of points of integration.
!    x        midpoint of each rectangle's interval
!    f        function to integrate
!    sum,pi   area of rectangles
!    tmp      temporary scratch space for global summation
!    i        do loop index
!*****************************************************************************
!
program Main
! use MPI
!.. Implicit Declarations ..
   implicit none
! #include "mpif.h"
   include 'mpif.h'
!.. Local Scalars ..
  integer, parameter :: master = 0
  integer :: i,n,myid,numprocs,ierr,rc
  double precision, parameter :: pi25dt = 3.141592653589793238462643d0
   double precision :: a,h,mypi,pi,sum,x
    !.. Intrinsic Functions ..
  intrinsic ABS, DBLE
!.. Statement Function to integrate
  double precision :: f
  f(a) = 4.d0 / (1.d0+a*a)
      call                          MPI_INIT(                          ierr
! Initialisierung der MPI-Umgebung
      call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )     ! Prozesskennung erfragen
     call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr ) ! Zahl der Geschwister erfragen    !
!... Executable Statements ...
  do
     if ( myid .eq. 0 ) then
         write (UNIT=*, FMT='(''Enter the number of intervals: (0 quits)'')')
        read (UNIT = *, FMT = '( I10)')  n
    end if
    call MPI_BCAST(n,1,MPI_INTEGER,master,MPI_COMM_WORLD,ierr) ! N wird an alle geschickt
     if (n <= 0) exit       ! check for quit signal
     h = 1.0d0 / n          ! calculate the interval size
     sum = 0.0d0
     do i = myid+1, n, numprocs        !z.B. n=2, h = 0.5
          x = h * (DBLE(i)-0.5d0)       ! X1 = 0.5*0.5 = 0.25,  x2=0.5*1.5=0.75
         sum = sum + f(x)              !sum1 = 4/1+0.25*0.25)    = 3.7647,
      end do                            !sum2 = 3.7647+4/(1+(0.75*0.75)) = 6.3247
     mypi = h * sum                    !mypi = 0.5* 6.3247  = 3.1624
! Für alle  mypi wird das arithmetische Mittel gebildet
    call MPI_REDUCE(mypi,pi,1,MPI_DOUBLE_PRECISION,MPI_SUM,master,MPI_COMM_WORLD,ierr)
    if (myid .eq. 0) then
         print*, 'Pi ist :',  pi,'Fehler ist:',  ABS(pi-pi25dt)
    endif
 end do
 call MPI_FINALIZE(rc)        !Freigeben der Systemresourcen
```

13 Bibliographie

[DUD 93] Duden, Informatik, Mannheim 1993

[Gil 93] **Giloi W.K., Rechnerarchitektur, 2. Auflage 1993, Springer-Verlag Berlin-Heidelberg-New York**

[GMD 93] Der GMD Spiegel 3/93, MANNA - Beiträge zur Parallelverarbeitung, St. Augustin 1993

[IBM FOR] IBM Parallel FORTRAN, in IBM System Journal, No. 4/1988

[IBM Sei] Seismic computations on the IBM 3090 Vector Multiprozessor, in IBM System Journal, No. 4/1988

[IBM RES] IBM Journal of Research and Development, Volume 35, November 1991

[Mär 94] **Märtin Christian, Rechnerarchitektur, Struktur, Organisation, Implementierungstechnik, Hanser Studienbücher der Informatik, 1994**

[Ung 97] **Theo Ungerer, Parallelrechner und parallele Programmierung, SpektrumAkademischer Verlag 1997**

14 Glossar

Barrieren-Synchronisation

Die *Barrieren-Synchronisation* synchronisiert eine Vielzahl von Kontrollfäden so, daß beim Erreichen einer Barriere alle Kontrollfäden abgearbeitet sein müssen, bevor das Programm weiter fortschreitet. Dazu führt jeder Kontrollfaden am Ende eine WAIT-Instruktion aus. Kontrollfäden, die zuerst fertig sind, verbleiben solange in dem Wartezustand, bis der zeitlich letzte Kontrollfaden ebenfalls den Wartezustand erreicht hat.

Datenparallelität

Parallel ausführbare Operation auf Datenstrukturobjekten. Ein Vektor ist beispielsweise ein einfaches Datenstrukturobjekt. Eine Datenparallelität ist dann gegeben, wenn strukturierte Datenmengen, wie z.B. Arrays oder Vektoren zu verarbeiten sind, wobei von den Operationen solcher Datenstrukturtypen von vornherein bekannt ist, wieweit sie parallel ausgeführt werden können. [Gil 93, S.143]

Datenstruktur-Architektur

Es handelt sich bei Datenstruktur-Architekturen um Rechnerarchitekturen, deren Operationsprinzip darin besteht, daß Datenstrukturtypen als elementare Einheiten der Maschine definiert werden und als solche unmittelbar verarbeitet werden können. Besitzt die Programmiersprache dieselben Typen, so führt die Maschine die Operationen dieser Typen unmittelbar parallel aus, und es bedarf dafür keiner Programmstrukturanalyse. Ein Merkmal von Datenstruktur-Architekturen ist die Existenz eines speziellen Strukturprozessors. In der einfachsten Form ist dies ein Adreßgenerator. Die einfachste Form einer Datenstruktur-Architektur ist die *Vektormaschine*.

Parallelarbeit in einer Datenstruktur-Architektur bedeutet, daß die in den einzelnen Schritten der komplexen Rechenoperationen der Strukturdatentypen auftretenden Elementaroperationen gleichzeitig ausgeführt werden. Um dies zu ermöglichen, wird man entweder ein *Array von Rechenelementen* oder aber einen *Pipeline-Prozessor* vorsehen.

Funktionaler Programmierstiel

Ein funktionales Programmiersystem kennt nur eine einzige Operation, die *Applikation*. Wenn f eine Funktion und x ein Wert ist, so ist f: x eine Applikation, deren Resultat der Wert ist, der sich durch die Anwendung der Funktion f auf den Wert x ergibt. In einem solchen Programm gibt es *keine Variablen im üblichen Sinne*, sondern nur geschachtelte Ausdrücke, wobei solche Ausdrücke eine erhebliche Länge annehmen und aus Schachtelungen erheblicher Klammerungstiefe bestehen können. [Gil93, S.238]Dieser funktionale Programmierstiel wird bei sogenannten *Reduktionsmaschinen*, einer Variante der Datenflußarchitekturen verwendet.

Bekannte funktionale Programmiersprachen sind z.B. LISP und LOGO, die allerdings auch imperative Sprachelemente enthalten[DUD93,S.545]

Kommunikationslatenz

Die Latenzzeit eines Kommunikationssystems ist die Zeit, die zwischen der Einleitung eines Kommunikationsvorganges und dessen Abschluß vergeht.

Leichtgewichtiger Prozeß

Teil eines Gesamtprogrammes, in dem keine Sprunganweisungen vorkommen dürfen - Basisblock.

Leichtgewichtiger Prozeß höherer Komplexität

Es dürfen Sprunganweisungen und Anweisungen zum Laden oder Speichern auf einen externen Speicher vorkommen [Gil93, S 350]

Leichtgewichtiger Prozeß niedrigster Komplexität

Es dürfen weder Sprunganweisungen noch Anweisungen zum Laden oder Speichern auf einen externen Speicher vorkommen [Gil93, S 350]

Kontrollfluß

Ein Programm besteht bei imperativen Programmiersprachen (Fortran, Pascal, Modula2...) aus einem Kontrollfluß und einem Datenfluß. Dementsprechend haben diese Sprachen sogenannte Kontrollkonstrukte (Schleifen, Bedingungen usw.). Visualisiert wird der Programmablauf mittels sogenannter Programmablaufpläne oder Programmflußpläne

Programmparallelität

Eine implizite Programmparallelität ist immer dann gegeben, wenn es in dem Algorithmus eine Menge von Operationen gibt, innerhalb derer es auf die Reihenfolge der Ausführung nicht ankommt. [Gil 93, S.143]

Dazu folgendes FORTRAN-Beispiel:

```
1       DO WHILE (X  .NE.  Stop)
2           Y := S(X)
3           IF (X+L(X)  .EQ.  Y) THEN
4              L(X) = L(X) + L(Y)
5              S(X) = S(Y)
6            ELSE
7              X = Y
8            ENDIF
9       ENDDO

        Die beiden Befehle 4 + 5
        sind parallel ausführbar
```

RISC

Die *Reduced Instruction Set Computer* Architektur zeichnet sich gegenüber den tradtionellen CISC (*Complex Instruction Set Computer*) Prozessoren durch relativ geringe Befehlsanzahl, einfaches Befehlsformat, Phasenpipelining, Load/Store Architektur und kleines CPI (Cycles per Instruction) aus.

Schwergewichtiger Prozeß

Task

Scoreboard

Datenkonsistenzkonflikte können in einer Pipeline auftauchen, wenn ein nachfolgender Befehl auf Daten eines vorausgegangenen Befehls zugreifen will, diese aber noch nicht im Registerfile zugänglich sind. Diese Konflikte können gelöst werden, wenn der nachfolgende Befehl so lange verzögert wird, bis das gewünschte Datum verfügbar ist. Die Hardware nennt man Scoreboard, die für jedes Register Buch führt, ob der Inhalt verfügbar ist und falls nicht, die Befehle, die in die Pipeline eingeführt werden, so lange anhält, bis der Konflikt beseitigt ist (die Daten im Registerfile stehen). Das Scoreboard besteht für jedes Registerfile aus einem Bitvektor mit n Bits, in dem jeweils ein Bit genau einem der Register zugeordnet ist. Zu Beginn einer Instruktionsausführung wird das Bit für das Zielregister der Instruktion gesetzt, nach Fertigstellung des Resultats wird das Bit zurückgesetzt.

Superskalar-Pipeline

Die Superskalar Pipeline ist bestimmendes Merkmal der *Superskalar-Architektur*. Sie zeichnet sich dadurch aus, daß mehrere Ausführungseinheiten parallel mehrere Befehle gleichzeitig verarbeiten können. Die Auswertung, ob die eingelesenen Befehle parallel oder sequentiell bearbeitet werden können, trifft ein *Sceduler* in Zusammenarbeit mit einem *Scoreboard* zur Laufzeit. Dabei werden die auftretenden Datenkonsistenz- und Steuerflußkonflikte berücksichtigt.

Die einzelnen Ausführungseinheiten haben meist eine unterschiedliche Funktionalität, so daß Hardware eingespart werden kann, ohne daß der Grad an Parallelität in gleichem Maße sinkt.

symmetrisch - asymmetrisch:

Sind alle Prozessoren bez¨üglich ihrer Rolle im System vergleichbar, so bezeichnet man es als symmetrisches Multiprozessorsystem. Ein asymmetrisches Multiprozessorsystem ist gegeben, falls die Prozessoren verschiedene, spezialisierte Funktionen ausüben. Heutzutage wird häufg der Begriff symmetrischer Multiprozessor symmetric multiprocessor) in einem engeren Sinne fürr speichergekoppelte Multiprozessoren mit gleichartigen Prozessoren verwendet.

VLIW-Architektur

Very Long Instruction Word Architektur. Es werden mit einem Befehlswort mehrere Befehle an die CPU übertragen und damit mehr als eine Ausführungseinheit ange-

sprochen. Ein VLIW-Prozessesor kann in einem Takt sogar mehrere gleichartige Befehle abarbeiten (z.B. 3 FPUs). Verwand mit superskalarer Architektur

von Neumann-Architektur

Die von Neumann-Architektur stellt die klassische Rechnerarchitektur dar. Sie besteht im Prinzip aus dem Prozessor und einem Speicher. Sowohl die Befehle als auch die Daten sind in diesem Speicher abgelegt. Da diese Schnittstelle zum Speicher bei heutigen Rechnern die erreichbare Systemleistung begrenzt, spricht man vom von Neumann-Flaschenhals, der durch den Übergang zur Harward-Architektur vermieden wird.

Diese Architektur wurde nach John von Neumann benannt, der in den 40er Jahren mit anderen den Übergang von der Rechnerprogrammierung durch Schalter und Steckverbindungen auf das Ablegen eines Programmes in einem Speicher anregte.

15 Index

A

Akteure · 63, 64
Aktivitäten · 39, 61, 63, 98, 100, 101, 105
Amdahls Gesetz · 33

B

Barrieren-Synchronisation · 2, 48, 49, 128

C

Cache-Kohärenzprotokoll · 68
Compiler · 6, 26, 27, 37, 40, 41, 42, 44, 46, 56, 57, 58, 61, 73, 95, 96, 97
  Loop Jamming · 2, 43
  Register-Optimierung · 2, 41
  Schleifen-Aufrollen · 2, 41

D

Datenabhängigkeitsanalyse · 6, 37, 44, 64
Datenflußarchitekturen · 3, 6, 39, 61, 62, 128
Datenflußgraph · 45, 61, 62
Datenflußprinzip · 61, 62, 63, 64
Datenkonflikte · 6
Datenkonsistenzkonflikt · 53
Datenparallelität · 36, 128
Datenstruktur-Architektur · 40, 128
DSM · 7, 21, 23

E

Ebenen der Parallelität · 2, 24, 25, 28
  Anweisungsebene · 2, 6, 25, 26, 28, 41, 43, 44, 52, 97
  Blockebene · 25, 26, 28, 96
  Jobebene · 25
  Programmebene · 25, 27
  Prozeßebene · 2, 6, 25, 44, 66, 76
Effizienz · 2, 29, 30, 31, 32, 34, 96

F

Feldrechner · 20, 21, 95, 97
  SIMD · 20, 21, 22
Flynnsche Klassifikation · 2, 19
Fork/Join-Modell · 2, 47
Funktionaler Programmierstiel · 128
Funktionspipelining · 8, 52

G

Granularität · 25, 28, 73, 78

H

horizontale Parallelisierung · 46
Horizontale Parallelisierung
  Vektorisierung. · 45, 46

I

Iteration · 39, 46, 97

K

Kohärenz · 68
Kommunikation
  No-Wait-Send · 51
  Remote Procedure Call · 51
  Rendezvous · 51, 77
Kommunikationslatenz · 3, 44, 50, 76, 77, 128
Kontrollabhängigkeitsgraph · 43
Kontrollfluß · 129

L

Leichtgewichtiger Prozeß · 128
Loop Jamming · 2, 43

M

MESI-Protokoll · 68
Metacomputer: · 22
MIMD · 20, 21, 22, 66, 126
  Nachrichtenorientierte Systeme · 7, 21, 50, 73
  speichergekoppelte Systeme · 7, 21, 94, 96, 97, 100, 129
MPI · 4, 7, 24, 94, 95, 96, 125, 126

N

Nachrichtenorientierte Systeme · 7, 21, 50, 73
Nebenläufigkeit · 8, 98
Non-uniform-communication-architecture-Modell · 23
Nonuniform-memory-access-Modell · 23
No-remote-memory-access-Modell · 23
No-Wait-Send · 51

P

Parallelität
  Ebenen · 2, 24, 25, 28
Parallerechner
  MIMD · 20, 21, 22, 66, 126
Phasenpipeling · 52
POSIX · 100, 108
Programmparallelität · 37, 129
Prozeß
  Leichtgewichtiger · 128
  Schwergewichtiger · 129

R

Register-Optimierung · 2, 41
Remote Procedure Call · 51
Remote Process Invocation · 51
Rendezvous · 51, 77
Resourcenkonflikt · 53
RISC · 129

S

Schleifen-Aufrollen · 2, 41
Schnüffel-Logik · 68
Schwergewichtiger Prozeß · 129
SIMD · 20, 21, 22
SISD · 20
SMP · 7, 21, 23
speichergekoppelte Systeme · 7, 21, 96
Superskalare Prozessoren · 2, 3, 6, 28, 52, 56
Superskalar-Pipeline · 129
Synchronisation · 2, 4, 7, 21, 23, 25, 30, 31, 44, 47, 49, 51, 57, 62, 63, 66, 73, 75, 77, 85, 100, 101, 104, 126
    Remote Process Invocation · 51
  Barrieren-Synchronisation · 2, 48, 49, 128
  Fork/Join-Modell · 2, 47
  No-Wait-Send · 51
  Remote Procedure Call · 51
  Rendezvous · 51, 77

T

Thread · 4, 7, 25, 94, 97, 98, 99, 102, 103, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122
Threads · 7

U

Uniform-memory-access-Modell · 23

V

Vektorisierung. · 45, 46
Vektorrechner · 6, 7, 20, 21, 22, 26, 28, 59, 97
  SIMD · 20, 21, 22
Vertikale Paralle
  lisierung · 39, 46, 97
Vertikale Parallelisierung · 46
VLIW-Architektur · 130
von Neumann-Architektur · 130

W

Workstation-Cluster · 22, 94, 125

Z

Zündregel-Semantik · 6, 61, 62