RISC-LINZ

Paralleles Rechnen

Klaus Kusche

Version 3.1
1991/1992

Copyright © 1991 by Klaus Kusche. All rights reserved.

Inhalt

Abbildungsverzeichnis **3**

Vorwort **4**

1 Motivation **5**

2 Parallele Hardware **9**

2.1 Klassifizierung 9

2.1.1 Was ist parallele Hardware? 9

2.1.2 Klassische Klassifizierung 10

2.1.3 Verschiedene Klassifizierungsmerkmale 11

2.2 Kommunikationshardware 16

2.2.1 Kenngrößen eines Kommunikationsmediums 17

2.2.2 Parallele und serielle Kommunikationsmedien 18

2.2.3 Busse 19

2.2.4 Speicher 21

2.2.5 Schaltnetzwerke 22

2.2.6 Kanäle 24

2.2.7 Topologien 26

2.3 Beispiele 33

2.3.1 Pipelines 33

2.3.2 Besondere Prozessorarchitekturen 36

2.3.3 Systolische Arrays, Zellularautomaten 39

2.3.4 SIMD-Computer 40

2.3.5 Klassische Supercomputer 44

2.3.6 Flache Shared-Memory-Systeme 46

2.3.7 Andere Shared-Memory-Systeme 47

2.3.8 Hierarchische Systeme 49

2.3.9 Hypercubes 50
2.3.10 Spezielle Parallel-Mikroprozessoren 52
2.3.11 Prolog-Maschinen 58
2.3.12 Maschinen für funktionale Sprachen 59

3 Parallele Software **67**
3.1 Einleitung 67
3.2 Klassifizierung 71
3.3 Strategien zur Parallelisierung 76
3.4 Programmierumgebungen 78
3.4.1 Das Debugging 79
3.4.2 Das Performance Debugging 81
3.4.3 Benötigte Werkzeuge 83
3.5 Parallelisierende Compiler 85
3.6 Höhere parallele Sprachen 87
3.6.1 Rein funktionale Sprachen 87
3.6.2 Paralleles Prolog 93
3.6.3 Paralleles Lisp 98
3.6.4 Linda 102
3.6.5 Objekt-orientierte Ansätze 104
3.6.6 Quasiparallele Modelle 105
3.6.7 Transaction Processing 106
3.6.8 Betriebssystem-Konzepte 106
3.7 Maschinennahe Ansätze 107
3.7.1 Parallele Prozesse 108
3.7.2 Kommunikation 111

Abbildungsverzeichnis

2.1 Klassifikation von Rechnern nach Flynn 11
2.2 Globaler Speicher 13
2.3 Verteilter Speicher 14
2.4 Crossbar Switch 23
2.5 Permutations-Schaltnetzwerk 24
2.6 Vergleich von Topologien 29
2.7 Beispiele für Topologien 1 30
2.8 Beispiele für Topologien 2 31
2.9 Instruktions-Pipeline 34
2.10 Vektor-Pipeline: Gleitkomma-Addition, vereinfacht 35
2.11 SIMD Struktur 41
2.12 Der BBN Butterfly 48
2.13 Abarbeitung von Datenflußgraphen 61
2.14 Aufbau eines Datenfluß-Prozessors 62
2.15 Graph-Reduktion 65

Vorwort

Diese Vorlesung verfolgt das Ziel, eine erste Einführung in parallele Computersysteme und die darauf zur Anwendung kommenden Programmiermethoden und -sprachen zu geben.

Es wird auf einen möglichst breiten Überblick Wert gelegt, eine Spezialisierung auf einzelne Rechnertypen, Programmiersprachen oder Anwendungen wurde nach Möglichkeit vermieden.

Der erste Teil der Lehrveranstaltung gibt nach einer kurzen Motivation für die Beschäftigung mit parallelen Systemen einen Überblick über Konzepte, Grundlagen und Probleme paralleler Hardware und stellt die wichtigsten real existierenden Maschinen kurz vor.

Der zweite Teil beschreibt die Möglichkeiten, parallele Hardware zu programmieren. Dabei wird sowohl auf explizite als auch auf implizite Parallelität eingegangen. Es werden sowohl die grundsätzlichen Modelle paralleler Programmierung als auch ihre Realisierung in konkreten Programmiersprachen behandelt.

Im dritten Teil der Vorlesung werden nach Maßgabe der verbleibenden Zeit einzelne, typische Systeme genauer behandelt. Dieser Teil ist nicht im Skriptum erfaßt.

Die Vorlesung geht nicht auf parallele Algorithmen und typische Anwendungen ein. Ebenso werden mathematische Modelle paralleler Rechner und Kommunikationssysteme nicht behandelt. Auch die formalen Grundlagen paralleler Programmiersprachen gehören nicht zum Stoffumfang.

Es ist nicht Ziel der Vorlesung, Kenntnisse in paralleler Programmierung zu vermitteln. Es wird weder auf eine konkrete Programmiersprache noch auf eine konkrete Softwareentwicklungsumgebung für parallele Rechner im Detail eingegangen.

Die Vorlesung setzt nur Grundkenntnisse des ersten Studienabschnittes in Rechnerarchitektur und Programmierung voraus, Spezialwissen ist nicht erforderlich.

Dieses Skriptum enthält (noch) keinen Literaturteil, Hinweise auf weiterführende Literatur können beim Autor erfragt werden.

Kapitel 1
Motivation

Es gibt verschiedene Gründe für die Entwicklung paralleler Computersysteme, obwohl diese auf den ersten Blick vor allem eine Erhöhung der Komplexität des Hardwaredesigns, der Systemprogrammierung und auch der Anwendungsprogrammierung mit sich bringen:

Geschwindigkeitssteigerung:

Der *Geschwindigkeit* und *Größe* von Einprozessorsystemen sind natürliche *physikalische Grenzen* gesetzt (Lichtgeschwindigkeit[1], Elementarladung usw.), einer Vervielfachung von einfachen, nicht an diese Grenzen stoßenden Systemen steht jedoch keine grundsätzliche Schranke im Wege.

Man schätzt, daß Einprozessorsysteme derzeit höchstens ein bis zwei Zehnerpotenzen von der theoretischen Höchstleistung entfernt sind, und die Geschwindigkeitssteigerungen bei Einprozessor-Höchstleistungssystemen gehen bereits langsamer vor sich als vor einigen Jahren[2].

Der *Aufwand* zur Erreichung höherer Geschwindigkeiten durch Parallelisierung ist zumindest aus Hardwaresicht bereits wesentlich geringer als jener zur weiteren Steigerung der Leistung von Monoprozessoren[3].

Designvereinfachung:

---

[1] In einem Taktzyklus moderner Supercomputer schafft ein elektrisches Signal gerade noch einen halben Meter.

[2] Supercomputer und Single-Chip-Mikroprozessoren gibt es in etwa gleich lange. Während aber Mikroprozessoren seit ihrer Einführung rund 500 mal schneller wurden, konnte die Geschwindigkeit eines einzelnen Supercomputer-Prozessors im gleichen Zeitraum nur um den Faktor 5 gesteigert werden.

[3] Ein einzelner Prozessor einer Cray Y-MP ist rund 300 mal teurer als ein intel i860 Single-Chip-Prozessor, und er braucht mindestens 1000 mal mehr Platz und Strom, aber er rechnet je nach Anwendung nur rund 5–30 mal schneller.

Es ist wesentlich einfacher, schneller und kostengünstiger, ein System zu entwickeln, zu produzieren und zu warten, daß aus einer *Vielzahl von identischen, vergleichsweise simplen Grundkomponenten* besteht[4], als eines, das auf dem Zusammenspiel vieler verschiedener Komponenten beruht.

Man muß bedenken, daß die Entwicklung eines einzigen hochintegrierten Halbleiters hundert Mannjahre und mehr in Anspruch nehmen kann. Besteht ein Computer aus vielen komplexen Komponenten, muß man davon ausgehen, daß zum Zeitpunkt der Fertigstellung des Gesamtsystems die Technologie der einzelnen Teile bereits völlig veraltert ist.

Weiters kann ein Prozessor heute nur mehr mit Hilfe umfangreicher Computersimulationen und -verifikationen konstruiert werden. Daher setzt die heute verfügbare Rechnerleistung der Entwicklung noch komplexerer Systeme Grenzen.

Kostensenkung:

Neben dem Ziel, die Leistungsfähigkeit von großen Supercomputern um jeden Preis zu steigern, ist heute vor allem der Trend zu erkennen, Systeme mit der Leistungsfähigkeit von Mainframes oder kleinen Supercomputern zu entwickeln, die preislich und größenmäßig für den Einsatz am Schreibtisch im Stil von Workstations und PC's geeignet sind (*"Personal Supercomputing"*)[5].

Alle diese Konzepte beruhen nicht auf leistungsfähigeren Monoprozessoren, sondern auf parallel arbeitenden Single-Chip-Mikroprozessoren.

Scalability:

Wenn der Anwender eines Einprozessorsystems eine Steigerung der Leistung wünscht, muß er sein bestehendes System durch ein neues, größeres ersetzen. Dadurch werden einerseits frühere Investitionen wertlos, und andererseits sind einzelne, große Neuausgaben notwendig. Außerdem erhöht sich die Leistung eines Monoprozessors typischerweise nicht linear mit seinen Anschaffungskosten, sondern wesentlich langsamer.

Bei einem Parallelrechner hingegen kann man im günstigsten Fall jederzeit je nach Leistungsbedarf und finanziellen Mitteln zusätzliche Rechenleistung in kleinen Schritten zu einem bestehenden System hinzufügen, oft sogar ohne Betriebsunterbrechung.

---

[4] Viele Parallelrechner (z. B. der DAP, Transputersysteme, der nCube, usw.) bestehen nur mehr aus zwei verschiedenen Bausteinen: Einem speziell entwickelten Prozessor- und Kommunikationschip und einem Standard-Speicherchip.

[5] Um 1 Million Schilling läßt sich ein PC bereits zum Supercomputer erweitern, und zwar innerhalb des ursprünglichen Gehäuses.

Fehlertoleranz:

Die älteste Anwendung paralleler Systeme bestand darin, durch Vervielfachung von Funktionseinheiten die *Zuverlässigkeit von kritischen Systemen* (z. B. Anlagensteuerungen, Kommunikationssysteme oder Datenbanken) zu erhöhen.

Dieser Punkt gewinnt mehr und mehr an Bedeutung, nicht nur, weil Computersysteme immer essentiellere Funktionen in unserer Welt einnehmen, sondern weil das Problem von Fehlern durch immer komplexere Systeme (statistische Fehlerwahrscheinlichkeit) und höhere Integration (unvermeidbare Beeinflussung durch Höhenstrahlung etc.) immer wichtiger wird.

Heute geht man sogar so weit, daß für eine kritische Anwendung (z. B. in Flugzeugen) mehrere verschiedenartige, völlig unabhängig voneinander entwickelte, aus verschiedenen Komponenten aufgebaute und getrennt programmierte Rechner zum Einsatz kommen, damit eventuell vorhandene Design- und Programmierfehler nicht alle Systeme gleichzeitig betreffen.

Mit diesem Anwendungsbereich werden wir uns im Rest der Vorlesung nicht mehr befassen.

Restkapazitäten-Nutzung:

Aus derselben Motivation, die vor vielen Jahren zu Multiuser-Betriebssystemen für Mainframes führte, nämlich alle Komponenten eines Rechners möglichst zu jedem Zeitpunkt zu benutzen, befaßt man sich heute mit Netzwerken und parallelen Betriebssystemen:

Die Vielzahl von *PC's und Workstations* innerhalb einer Institution (oder die Vielzahl von Zentral-Rechnern an den internationalen Netzen) bieten ein enormes Potential an kostenloser, ungenutzter Rechenleistung (z. B. in den Nachtstunden). Wäre es möglich, dieses für einzelne rechenintensive Anwendungen zu nutzen, ließen sich viele Großrechner einsparen: 100 moderne Workstations entsprechen in etwa einem großen Supercomputer[6].

Theoretische Erkenntnisse:

Die Beschäftigung mit parallelen *Maschinen- und Kommunikationsmodellen*, mit der *Semantik* paralleler Programmiersprachen oder mit der *Komplexitätsanalyse* paralleler Algorithmen brachte viele neue Erkenntnisse und Einsichten, z. B. im Bereich der Berechenbarkeit.

[6] Ein amerikanisches Forschungszentrum nutzte bereits die Restkapazität von 14 verteilten VAX Computern, die durch das Internet verbunden waren, für eine Anwendung aus der Physik. Dabei wurde in etwa die Rechenleistung einer kleinen Cray erreicht, aber kostenlos und ohne Warten auf die Zuteilung von Supercomputer-Rechenzeit.

Auch mit diesem Punkt werden wir uns in der Folge nicht mehr beschäftigen.

Kapitel 2

Parallele Hardware

Nach der Definition und Klassifizierung paralleler Hardware wird das Kernstück eines jeden Parallelrechners, die Kommunikationshardware, genauer untersucht. Schließlich werden die bedeutendsten real existierenden Systeme vorgestellt.

2.1 Klassifizierung

2.1.1 Was ist parallele Hardware?

Diese Frage wird sehr oft behandelt, es wurde allerdings bis heute keine allgemeingültige, zufriedenstellende Definition gefunden (das liegt vor allem daran, daß praktisch jeder heute vorkommende Rechner im Prinzip intern in irgendeiner Form parallel arbeitet[1]). Ich werde daher der Vielzahl von existierenden Definitionen meine eigene hinzufügen:

Ein Parallelrechner ist ein Rechner,
der zu einem Zeitpunkt
mehr als ein "Benutzerdatum"

[1] Ein normaler PC enthält zumindest 4 Prozessoren: Die CPU, die FPU, den Tastaturkontroller und einen Prozessor im Platteninterface.

verarbeiten kann[2].

Dabei ist ein "Benutzerdatum" ein für den Benutzer sichtbares, natürliches, atomares Datenobjekt.

Diese etwas seltsame Definition schließt einerseits aus, daß parallel mit Rechenoperationen durchgeführte, aber für den Benutzer nutzlose oder unsichtbare Vorgänge bereits einen Parallelrechner ausmachen (z. B. das in allen heutigen Prozessoren übliche Inkrementieren des Program Counters und Durchführen von Adressrechnungen gleichzeitig mit der eigentlichen Operation, oder die parallele Ausführung von I/O-Operationen), und verhindert andererseits, daß das gleichzeitige Verarbeiten der Bits eines Speicherwortes einen Rechner schon zum Parallelrechner stempelt (jeder Rechner kann heute zwei Integers "in einem Schritt" addieren[3]).

2.1.2 Klassische Klassifizierung

Auch für die Klassifizierung und Unterteilung der Parallelrechner haben sich im Laufe der Zeit unüberschaubar viele Schemen entwickelt, die alle ihre Stärken und Schwächen haben.

Von diesen ist das Schema von ***Flynn*** heute am weitesten verbreitet und allgemein anerkannt: Es unterteilt Parallelrechner nach der Anzahl der *Instruktionen*, die gleichzeitig durchgeführt werden können[4], und nach der Anzahl der *Daten*, die gleichzeitig verarbeitet werden können. Es ergeben sich vier Klassen (siehe auch Abbildung 2.1):

SISD (Single Instruction, Single Data)**:**

Hierbei handelt es sich um die klassischen, *nichtparallelen* Rechner.

SIMD (Single Instruction, Multiple Data)**:**

In diese Kategorie fällt ein großer Teil der klassischen numerischen *Supercomputer* (Vektor- und Arrayprozessoren).

MISD (Multiple Instruction, Single Data)**:**

Diese Klasse muß nach heutigem Verständnis *leer* sein.

---

[2] Das wird uns aber nicht hindern, hier auch auf Konzepte einzugehen, die dieser Definition nicht entsprechen.

[3] Für Hardware-Techniker heißt die dazu notwendige Schaltung allerdings sehr wohl "paralleler Addierer", sie ist auf Grund der Übertragsfortpflanzung durchaus nichttrivial und ein typisches, sehr interessantes Beispiel eines parallelen Algorithmus.

[4] Für nicht-von-Neumann-Rechner ersetze man "Instruktionen" durch "Operationen".

SISD *Single Instruction* *Single Data* Sequentielle Rechner	**SIMD** *Single Instruction* *Multiple Data* Feldrechner
MISD *Multiple Instruction* *Single Data* leer!	**MIMD** *Multiple Instruction* *Multiple Data* Mehrprozessorsysteme

Abbildung 2.1: *Klassifikation von Rechnern nach Flynn*

MIMD (Multiple Instruction, Multiple Data):

Dies ist die Klasse der Parallelrechner im engeren Sinn, d. h. der *Mehrprozessorsysteme*[5].

Wie bei den meisten anderen Schemen sind die Übergänge fließend, und es kann durchaus sein, daß sich manche Rechner überhaupt nicht sinnvoll einordnen lassen.

2.1.3 Verschiedene Klassifizierungsmerkmale

Im folgenden sind (ohne Anspruch auf Vollständigkeit) verschiedenste weitere Merkmale angeführt, nach denen man Parallelrechner unterscheiden kann:

Spezialrechner oder Universalrechner?

Hier ist die Frage, ob ein Rechner universell nutzbar (d. h. beispielsweise in einer universellen, höheren Programmiersprache *programmierbar*) oder nur für eine bestimmte Anwendung einsetzbar ist.

Beispiele für parallele *Spezialrechner* sind in Anlagensteuerungen sowie in Systemen für Bild- und Signalverarbeitung (FFT-Netze, Filter) und Kommunikationsrechnern (Verschlüsselung, Vermittlung) zu finden.

[5] Wesentlich ist, daß die Operationen voneinander unabhängig sind, parallele Behandlung mehrerer zusammenhängender (z. B. logisch aufeinanderfolgender) Operationen ergibt noch keinen Multiple-Instruction-Rechner.

Systolic Arrays, *Zellularautomaten* und *Neuronale Netze* sind Grenzfälle: Sie sind zwar in irgendeiner Form steuerbar, aber im praktischen Sinn sicher nicht universell.

Numerische Superrechner sind hingegen in den meisten Fällen durchaus universell, auch wenn sie bei nichtnumerischen Anwendungen weit unter ihrer Höchstleistung bleiben.

Von-Neumann-Rechner oder nicht?

Mit dem Aufkommen von Parallelrechnern erlangten nichtklassische Verarbeitungsmethoden (*Datenfluß-*, *Reduktions-*, *Inferenz-* und andere Maschinen) erhöhte Bedeutung: Sie sind zwar grundsätzlich (d. h. auch als Einprozessor) komplexer als von-Neumann-Systeme[6], aber dafür oft wesentlich *einfacher zu parallelisieren* (das von-Neumann-Prinzip ist sowohl aus Hardware- als auch aus Softwaresicht eigentlich denkbar ungeeignet für parallele Systeme).

Pipelining oder Vervielfachung?

Beim *Pipelining* entsteht Parallelität durch *gleichzeitige Ausführung der verschiedenen für eine Operation notwendigen Schritte* in eigenen, spezialisierten (und daher normalerweise untereinander verschiedenen) Einheiten[7] (Musterbeispiel: Vektorprozessoren), bei der *Vervielfachung* entsteht Parallelität durch *gleichzeitige Ausführung mehrerer voneinander unabhängiger, gleicher oder verschiedener Operationen* auf normalerweise identischen Einheiten (Prototypen: Arrayrechner, Multiprozessoren).

Gleiche oder unterschiedliche Knoten?

Die Frage ist, ob aus *Softwaresicht* alle im System vorhandenen *Knoten ident* behandelt werden können oder nicht. Unterschiede zwischen den Knoten können sich aus verschiedensten Gründen ergeben:

Durch grundsätzliche Aufgabenteilung und Spezialisierung:

Bei paralleler Lisp-Hardware gibt es manchmal zwei unterschiedlich aufgebaute Prozessortypen für Programmausführung und Garbage Collection, ähnliches gilt für Prolog (Unifikationsprozessor usw.).

Durch Zusatzhardware in einzelnen Knoten:

Beispiele sind verschiedene Speichergrößen, Vektor-KoprozesSoren, Grafik- oder I/O-Kontroller usw..

---

[6] Das sei für uns alles, was Programme und Daten in durch explizite Adressierung angesprochenen Speichern hält.

[7] Pipelining impliziert nicht unbedingt Single Instruction, der HEP ist ein Gegenbeispiel.

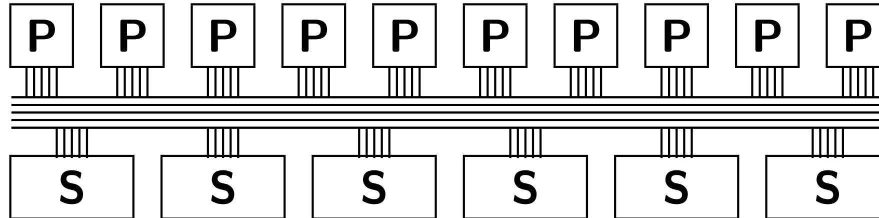


ABBILDUNG 2.2: *Globaler Speicher*

Durch die Lage im System:

Bei baumförmig aufgebauten Systemen beispielsweise ergibt sich schon durch die Lage der Knoten ein Unterschied (Wurzel, innere Knoten, Blätter), auch wenn sich dieser nicht in unterschiedlicher Hardware äußert.

Globaler, inhomogener oder verteilter Speicher?

Im wesentlichen gibt es aus logischer Sicht drei unterschiedliche Formen der Speicherorganisation:

Globaler Speicher:

Es existiert ein *einziger, großer Speicher*, jeder Prozessor sieht den Speicher in gleicher Weise und kann auf jeden beliebigen Teil des Speichers in gleicher Weise und gleich schnell zugreifen (siehe ABBILDUNG 2.2)[8].

Dieses Modell ist für den Programmierer am angenehmsten und universellsten, aber hardwaremäßig für größere Prozessorzahlen schwer zu realisieren.

Inhomogener Speicher:

Hier sind alle Varianten einzuordnen, bei denen für einen Prozessor *verschiedene Klassen von Speichern* existieren.

Ein wichtiger Vertreter sind Systeme, deren Speicher zwar logisch global, aber physikalisch auf die einzelnen Prozessoren verteilt ist, wodurch jeder Knoten auf Grund unterschiedlicher Zugriffszeiten zwischen "eigenem" und "fremdem" (*lokalem* und *nichtlokalem*) Speicher unterscheiden muß (Musterbeispiel ist der BBN BUTTERFLY, siehe ABBILDUNG 2.12).

[8] Dabei tut es nichts zur Sache, ob der Speicher tatsächlich zentral angeordnet oder hardwaremäßig auf die einzelnen Prozessoren verteilt ist.

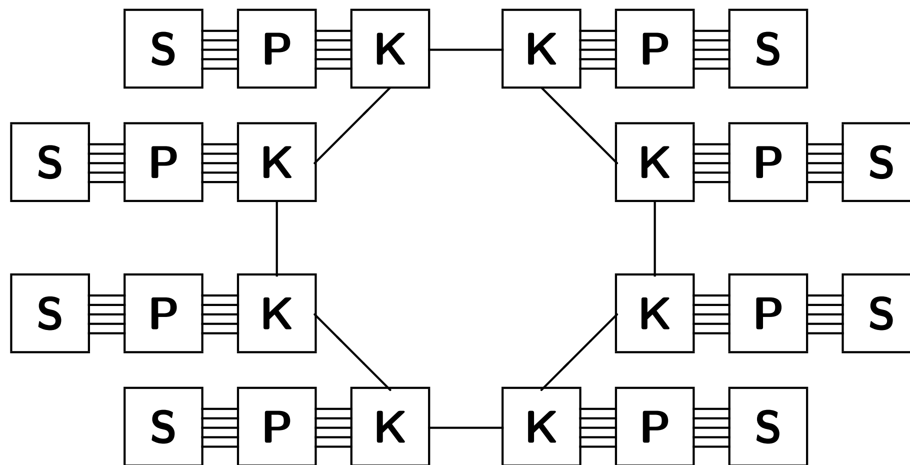


Abbildung 2.3: *Verteilter Speicher*

Auch einige hierarchisch aufgebaute Systeme fallen in diese Klasse: Oftmals haben alle Prozessoren innerhalb eines Clusters einen gemeinsamen, lokalen Speicher, auf den sie direkt zugreifen können, während sie zum Zugriff auf den Speicher anderer Cluster das globale Kommunikationssystem in Anspruch nehmen müssen.

Schließlich gibt es einige Systeme, bei denen jeder Knoten einen lokalen Speicher und zusätzlich einige mit seinen Nachbarn gemeinsame Speicher (nur für Kommunikationszwecke) besitzt.

Derartige Modelle werden bei einigen Supercomputer-Forschungsprojekten mit vielen Prozessoren verwendet, sind aber unangenehm zu programmieren.

Verteilter Speicher:

Bei diesem Modell kann jeder Prozessor nur auf den ihm *eigenen, lokalen Speicher* zugreifen, dafür aber exklusiv. Von der Existenz der Speicher auf anderen Knoten merkt er gar nichts, er kann darauf nur indirekt unter aktivem Zutun des betroffenen Prozessors zugreifen (siehe Abbildung 2.3).

Dies ist das hardwaremäßig einfachste Modell für beliebig viele Prozessoren; aus der Sicht des Programmierers ist es klar und einfach, aber relativ restriktiv.

Busse, Schaltnetzwerke oder Kanäle?

Diese Frage betrifft das Kommunikations- oder Verbindungssystem zwischen den Prozessoren untereinander oder zwischen den Prozessoren und

den Speichern:

Busse:

Das wesentliche Kennzeichen von Bussen (in der Bedeutung, in der ich sie hier gebrauche) ist, daß es sich um Verbindungswege handelt, die *von mehreren Knoten aktiv beansprucht* werden können, die aber *zu einem Zeitpunkt nur von einem Knoten tatsächlich benutzt* werden dürfen. Charakteristisch ist daher die Möglichkeit von *Zugriffskonflikten* und *Wartezeiten*.

Es können durchaus mehrere Busse in einem System existieren, und auch serielle Verbindungsleitungen wie z. B. Ethernet fallen (im Gegensatz zur technischen Bedeutung des Wortes Bus) unter diese Definition.

Ein einziger Bus ist typisch für kleine Parallelrechner mit globalem Speicher, und auch in hierarchischen Systemen werden oft Busse zur Kommunikation innerhalb eines Clusters verwendet.

Schaltnetzwerke:

Schaltnetzwerke sind Vorrichtungen, zu denen alle Komponenten eines Systems ihren eigenen, privaten Anschluß haben, und die aktiv (hardwaremäßig oder unter Software-Kontrolle) *beliebige Punkt-zu-Punkt-Verbindungen* zwischen jeweils zwei solchen Komponenten herstellen können, und zwar *mehrere gleichzeitig*.

Das kann kollisionsfrei (d. h. mit konstanter, garantierter Zugriffszeit) oder mit den gleichen Problemen wie bei Bussen geschehen.

Schaltnetzwerke kommen vor allem in großen Parallelrechnern vor, und zwar sowohl in Systemen mit globalem, als auch in solchen mit verteiltem oder inhomogenem Speicher.

Kanäle:

Bei Kanälen handelt es sich um *feste Punkt-zu-Punkt-Verbindungen*, die jeweils zwei Knoten miteinander verbinden. Kanäle sind per definitionem *frei von Zugriffskonflikten*, sie stehen den angeschlossenen Knoten jederzeit exklusiv und unabhängig von Aktivitäten auf anderen Kanälen zur Verfügung.

Praktisch alle Systeme mit verteiltem Speicher verwenden Kanäle zur Kommunikation[9].

[9] Diese Klassifikation stößt bei modernen Systemen mit hardwaremäßigem Routing in jedem Knoten oder speziellen Routing-Chips auf Probleme: Die Verbindungen zwischen den einzelnen Knoten sind zwar nach wie vor Kanäle, die der hier gegebenen Definition entsprechen; aber das Kommunikationssystem als ganzes entspricht in allen Punkten eindeutig einem Schaltnetzwerk.

Flache, nachbarliche oder hierarchische Struktur?

Diese Frage betrifft die Sicht, die die Knoten (vor allem in Systemen mit verteiltem Speicher) voneinander haben:

Flache Struktur:

Bei flachen Systemen kann *jeder Knoten jeden anderen* in gleicher Art und Weise ansprechen und mit ihm mit gleichem Aufwand kommunizieren.

Das gilt für Systeme, die auf einem Bus oder einem Schaltnetzwerk basieren, sowie für Systeme mit Kanälen, die einen Full Graph bilden.

Nachbarliche Struktur:

Hierbei sieht jeder Knoten zwei Klassen von anderen Knoten: *Nachbarn*, mit denen er direkt kommunizieren kann, und *nicht-Nachbarn*, die er nur auf dem Umweg über andere Knoten erreichen kann.

Das ist der bei Systemen mit Kanälen typische Zustand.

Hierarchische Struktur:

Eine hierarchische Struktur ist gegeben, wenn das System in *mehrere Ebenen* gegliedert ist, die ihr jeweils eigenes Kommunikationsnetz besitzen. Man nennt die Kombination mehrerer Knoten zu einem Element einer übergeordneten Struktur normalerweise einen *"Cluster"*. Kommunikationen zwischen Knoten innerhalb eines Clusters werden über das lokale Kommunikationsmedium abgewickelt, alle anderen Kommunikationen laufen über das globale Medium (und üblicherweise auch über die lokalen Medien in den betroffenen Clustern).

Diese Struktur ist typisch für einige besonders große Parallelrechner.

2.2 Kommunikationshardware

Das zentrale Problem bei der Konstruktion von Parallelrechnern ist das Kommunikationsmedium. Nach der Beschreibung der grundlegenden Kenngrößen eines Kommunikationsmediums werden wir die wichtigsten physikalischen Datenübertragungsmöglichkeiten, nämlich serielle und parallele Leitungen, gegenüberstellen. Dann werden wir uns näher mit Bussen (und Speichern), Schaltnetzwerken und Kanälen beschäftigen. Zu den Kanälen gehört auch die Untersuchung der verschiedenen Topologien.

2.2.1 Kenngrößen eines Kommunikationsmediums

Im wesentlichen läßt sich ein Kommunikationsmedium durch folgende Kenngrößen beschreiben:

Durchsatz:

Der Durchsatz gibt an, welche *Datenmenge pro Zeiteinheit* übertragen werden kann, und zwar sowohl *pro Anschluß* als auch *insgesamt*[10].

Zugriffsverzögerung:

Die Zugriffsverzögerung gibt an, wieviel *Zeit von der Beantragung einer Kommunikation bis zu ihrem tatsächlichen Beginn* vergehen kann, d. h. wieviel Zeit ein Knoten zum Zugriff auf das Kommunikationsmedium braucht. Sowohl der *durchschnittliche* als auch der *maximale* Wert sind von Bedeutung (vor allem eine fixe Obergrenze für den maximalen Wert ist wichtig).

Dieser Wert ist wesentlich für Busse und manche Arten von Schaltnetzwerken; bei Kanälen ist er per definitionem 0.

Übertragungsverzögerung:

Die Übertragungsverzögerung gibt an, wieviel *Zeit* vergeht, bis die *abgesendeten Daten tatsächlich beim Empfänger ankommen*, d. h. wieviel Zeit im Kommunikationsmedium selbst verloren geht. Auch hier müssen wieder *Durchschnitt* und (hoffentlich vorhandenes) *Maximum* betrachtet werden.

Für Busse ist dieser Wert praktisch 0, für Kanäle üblicherweise konstant, für Schaltnetzwerke und hierarchische Systeme oft abhängig von der Anzahl der Zwischenknoten oder Ebenen.

Alle diese Größen werden sinnvollerweise in Abhängigkeit von den folgenden Parametern beschrieben:

Last:

Bei allen Kommunikationssystemen, bei denen *Zugriffskonflikte* auftreten können, oder bei denen *Softwareunterstützung* zur Kommunikation notwendig ist, hängt die Leistung klarerweise von der Belastung ab. Interessant sind vor allem zwei Grenzfälle: Gleichmäßige Auslastung auf

[10] Der Durchsatz in Summe kann gleich dem Einzeldurchsatz sein (typisch für viele Busse), er kann gleich dem Produkt Einzeldurchsatz mal Anschlußzahl sein (typisch für viele Schaltnetzwerke und Netze von Kanälen), oder er kann irgendwo dazwischen liegen (Beispiel: Hierarchische Systeme; im täglichen Leben das Telefonnetz).

allen möglichen Kommunikationswegen oder Höchstlast auf einem einzigen Knoten oder Pfad.

Blockgröße:

Hier wird beschrieben, wie sich die Kommunikationseigenschaften in Abhängigkeit von der Menge der pro einzelner Kommunikation übertragenen Daten ändert (es ist offensichtlich ein Unterschied, ob der mit einer Kommunikation verbundene Overhead für ein einzelnes Byte getrieben wird oder gleich ein Megabyte auf einmal geschickt wird)[11].

Entfernung:

Bei allen hierarchischen Systemen ist es wesentlich, über wieviele *Ebenen* eine Kommunikation führt, bei nachbarlichen Systemen ist die Anzahl der *Zwischenknoten* entscheidend, bei Schaltnetzwerken die Zahl der beteiligten Schalter oder Router.

Obwohl sich diese Zahlen normalerweise für die nackte Hardware bestimmen lassen, ist es sinnvoller, sie unter Einbeziehung der zwangsweise involvierten *Systemsoftware* zu messen (d. h. beispielsweise von Systemaufruf zu Systemaufruf), denn auf diesem Niveau verwendet der Anwender die Kommunikation.

2.2.2 Parallele und serielle Kommunikationsmedien

Unabhängig davon, ob die Struktur des Kommunikationssystems einem Bus (oder mehreren Bussen), einem Schaltnetzwerk oder einem Netz von Kanälen entspricht, kann die Datenübertragung physikalisch auf zwei Arten erfolgen:

Parallele Medien:

Parallele Medien ("Busse" im herkömmlichen Sinn) übertragen die Daten und Adressen in voller Wortbreite gleichzeitig auf mehreren Leitungen.

Parallele Übertragung ist die für heutige Computer "natürlichere" Form, sie ist *konzeptuell einfach* zu realisieren (keine aufwendige Logik, nur Treiberbausteine). Der Nachteil liegt im physikalischen Bereich: Sie benötigt *viele Leitungen* und vor allem viele Anschlüsse an den Chips, und die vielen Treiberbausteine erzeugen beträchtliche Abwärme. Aus

[11] Ähnlich wie bei Vektorrechnern, wo man traditionell jene Vektorlänge als Maßzahl für den Overhead angibt, bei der genau die Hälfte der theoretisch möglichen Pipeline-Höchstleistung erreicht wird, wurde auch hier vorgeschlagen, jene Blocklänge anzugeben, bei der die Hälfte des maximal möglichen Durchsatzes erreicht wird.

diesem Grund geht man mit steigender Anzahl von Verbindungen auf serielle Übertragung über.

Parallele Datenübertragung ist prinzipiell *schneller*, sie wird meistens vom Prozessor selbst in Form normaler Speicherzugriffe durchgeführt, belegt also während des Datentransfers den Prozessor, verursacht aber dafür keinen Softwareoverhead. Daher wird parallele Übertragung vor allem bei *feinkörniger Parallelität* (häufiger Übertragung kleiner Datenmengen) eingesetzt.

Serielle Medien:

Serielle Verbindungen ("Links" o. ä.) schicken die Daten, Adressen und Steuerinformationen sequentiell über nur eine Leitung. Sie erfordern dafür aber sehr schnelle, komplexe Bauteile zur Erzeugung und zum Empfang des seriellen Datenstromes sowie zum direkten Speicherzugriff (DMA).

Sie sind auch langsamer, einerseits auf Grund der reduzierten Leitungszahl, andererseits wegen der notwendigen Serialisierung, Synchronisation usw., und auch wegen des DMA-Overheads. Auch die Programmierung ist aufwendiger, da der DMA-Kontroller initialisiert werden muß und das Ende der Übertragung meist durch einen Interrupt signalisiert wird[12]. Dafür läuft die eigentliche Übertragung *unabhängig vom und parallel zum Prozessor* ab. Serielle Übertragung kann daher bei *grobkörnigem Parallelismus* durchaus effizient sein.

2.2.3 Busse

Das grundsätzliche Problem bei Bussen besteht darin, daß ein Bus nur eine *feste Kapazität* hat, die unter allen angeschlossenen Knoten aufgeteilt werden muß. Busse stellen daher den geschwindigkeitsbestimmenden *Engpaß* im System dar, sie bewirken, daß ein System nicht beliebig ausbaubar ist, weil ab einer gewissen Grenze von zusätzlichen Knoten keine Leistungssteigerung mehr zu erwarten ist.

Diese Leistungsgrenzen sind *physikalisch bedingt*: Je mehr Knoten an einem Bus hängen, und je länger der Bus ist, umso langsamer wird er auf Grund der Signallaufzeiten und -verfälschungen, und umso mehr elektrische Energie ist zum Erzeugen der Signale notwendig, was Probleme mit der Abwärme mit

---

[12] In der Praxis beträgt die Datenrate serieller Verbindungen bestenfalls ein Zehntel des Durchsatzes von Bussen vergleichbarer Technologie. Der Overhead pro Kommunikation liegt im Bereich einiger Mikrosekunden.

sich bringt[13].

Weiters muß man die *Zugriffskollisionen* auflösen. Man erwartet von der Zugriffssteuerung, daß sie *deadlockfrei* und *fair* arbeitet (d. h. daß jeder Knoten die gleichen Chancen hat, den Bus zu benutzen); im Idealfall sollte sie eine *feste Obergrenze für auftretende Zugriffsverzögerungen* garantieren.

Ebenfalls sehr nützlich ist die Möglichkeit, identische, gleichzeitige Zugriffe verschiedener Knoten zu *kombinieren* und als einen einzigen Zugriff zu behandeln, beziehungsweise umgekehrt einem einzigen Knoten das Ansprechen mehrerer oder aller anderen Knoten in einem einzigen Zugriff zu gestatten (*multicast*, *broadcast*).

Darüber hinaus muß Vorsorge für die zur Implementierung paralleler Software notwendigen garantiert *atomaren (d. h. unteilbaren) read-modify-write Zugriffe* geschaffen werden (`test-and-set`, `fetch-and-add`), d. h. jeder Knoten muß die Möglichkeit haben, einen Wert zu lesen, zu ändern und wieder abzuspeichern, ohne daß irgendein anderer Prozessor in dieser Zeit auf diesen Wert zugreifen kann (*"bus lock"*).

Ein großes Problem ergibt sich, wenn man zur Verringerung der Busbelastung lokale Caches in den einzelnen Knoten einführt: Die in den Caches zwischengespeicherten Werte müssen zu jedem Zeitpunkt mit dem eigentlichen Wert übereinstimmen (*"Cache Consistency"*). Wenn daher irgendwo im System ein Datum abgespeichert wird, müssen sofort alle Knoten informiert werden, daß eventuell in ihrem Cache vorhandene Kopien davon jetzt ungültig geworden sind. Besonders wichtig und schwierig ist dies bei atomaren Operationen.

Ein ähnliches Problem ergibt sich bei der Verwendung lokaler *Memory Management Units* für virtuellen Speicher: Die Seitentabellen und vor allem die Modified-Bits müssen konsistent bleiben[14].

Neben der Datenübertragung sind auch noch Mechanismen für *Kontrollfunktionen* vorzusehen: Um eine Vergeudung von Rechenleistung durch Polling zu vermeiden, sollte es beispielsweise möglich sein, daß jeder Knoten auf jedem

---

[13] Derzeit liegen diese Grenzen in der Praxis bei max. 50 cm Länge und max. 30 angeschlossenen Boards. Mit konventioneller Technologie lassen sich rund 200 MB/s erreichen, in Supercomputern bis 1000 MB/s. Wenn man bedenkt, daß ein einziger moderner Mikroprozessor ohne Cache bis zu 300 MB/s braucht, um seine volle Leistung zu entfalten, und daß bei 90 % Cache Hits immer noch 30 MB/s benötigt werden, ist das sehr wenig. Als praktisch sinnvol und effizient erwiesen sich Systeme mit höchstens 32 Mikroprozessoren oder 8 Supercomputer-Prozessoren.

Auch die Verzögerungen sind beträchtlich: Während on-Chip Caches Zugriffszeiten von 10–20 ns haben, und ein lokaler, exklusiver Speicher auf derselben Platine in 50–150 ns angesprochen werden kann, benötigt ein Zugriff auf den globalen Speicher typischerweise 250–750ns; in dieser Zeit könnten einige Dutzend Instruktionen verarbeitet werden . . .

[14] Das ist einer der Gründe, warum viele der heutigen Supercomputer (z. B. Cray) noch immer ohne Caches und ohne virtuellen Speicher oder Speicher-Zugriffs-Schutz gebaut werden.

anderen Knoten einen *Interrupt* auslösen kann[15].

2.2.4 Speicher

Das Design des Speichers ist im Normalfall eng mit dem Design des Bussystems verknüpft.

Schon bei sequentiellen Systemen ist der *Speicherzugriff* heute der *zeitkritische Faktor*: Die Prozessoren könnten eine wesentlich größere Datenmenge verarbeiten als die Speicher zu liefern imstande sind[16].

Weiters ist ein vor allem bei Parallelrechnern wesentlicher Nachteil, daß bei konventionellen Systemen der Bus nach dem Anlegen der Adresse für den Zeitraum, den der Datenzugriff auf den Speicher benötigt, zwar belegt ist, aber nichts Nützliches leistet.

Um beides zu umgehen, gibt es mehrere Wege:

Pipeline-Übertragung:

Bei dieser werden bereits neue Adressen angelegt, bevor die vorhergehenden Daten übermittelt sind. Es wird in jedem Takt eine Adresse und ein Datum übertragen, wobei ein Datum der zugehörigen Adresse um eine bestimmte Zahl von Takten nacheilt.

Paket-Übertragung:

Hier wird immer gleich eine größere Anzahl von aufeinanderfolgenden Datenworten (ein Paket) nach Anlegen einer einzigen Adresse übertragen (der Zugriff auf aufeinanderfolgende Speicherzellen ist schneller durchführbar als der Zugriff auf auseinanderliegende).

Dieses Verfahren ist vor allem bei Vektorrechnern und bei Rechnern mit Cache-Speicher von Nutzen, man lädt die nächsten paar Zellen gleich “vorsorglich” in den Cache und nimmt an, daß sie ohnehin bald benötigt werden.

---

[15] Manche Systeme (z. B. ALLIANT, CONVEX, CRAY) haben darüber hinaus noch eine Vielzahl weiterer Hardwareeinrichtungen zum automatische Synchronisieren der Prozessoren untereinander, z. B. für parallel auszuführende Schleifen.

[16] Prozessoren führen typischerweise im Schnitt knapp zwei Speicherzugriffe pro Takt (d. h. alle 15–30 ns bei modernen Mikroprozessoren) aus: Einen auf den Befehl und einen für die Daten. Die heute normalerweise verwendeten Speicherbausteine erreichen Zugriffszeiten von bestenfalls 60 ns zuzüglich einer ebensolangen Pause zwischen zwei Zugriffen. Die hohen Rechengeschwindigkeiten moderner Prozessoren hängen also wesentlich von Cache-Speichern direkt am Prozessorchip ab, aber gerade Caches sind bei Parallelrechnern ein großes Problem.

Aufteilung in Speicherbänke:

Auch hier wird die Übertragung aufeinanderfolgender Speicherzellen beschleunigt: Man teilt den gesamten Speicher in n Bänke (sodaß adressmäßig aufeinanderfolgende Worte in aufeinanderfolgenden Bänken zu liegen kommen), auf die man reihum um $1/n$ der Zugriffszeit versetzt zugreift, und erreicht damit bei gleicher Zugriffszeit den n-fachen Datendurchsatz. Dieses Verfahren kommt bei fast allen Vektorrechnern zum Einsatz.

Intelligente Speicher:

Bei nichtkonventionellen Maschinen stellt sich oft heraus, daß es Möglichkeiten zum Zugriff auf den Speicher gäbe, die weniger Steuerinformation pro Zugriff als eine direkte Adressierung brauchen. Man verlagert daher die für die eigentliche Adressierung notwendige Intelligenz in den Speicher selbst und kommuniziert mit dem Speicher auf einem Protokoll höherer Ebene (Assoziativspeicher, Lisp-Speicher, Graph-Speicher für Reduktionsmaschinen, Pattern-Matching-Speicher usw.), nämlich auf jene Ebene, wo mit einem Minimum an Kontrollinformation ein Maximum an Daten angesprochen werden kann.

2.2.5 Schaltnetzwerke

Schaltnetzwerke haben im allgemeinen die Aufgabe, n Eingänge mit m Ausgängen auf alle möglichen Arten paarweise zu verbinden. Am weitesten verbreitet sind zwei Grundtypen:

Crossbar Switches:

Crossbar Switches bestehen aus einer *rechteckigen, gitterförmigen Anordnung* von $n * m$ Schaltelementen, die zu verbindenden Knoten sitzen an einer Schmal- und einer Breitseite des Rechtecks. Jeder Knoten ist mit n (bzw. m) Schaltern exklusiv und unmittelbar verbunden (siehe Abbildung 2.4).

Diese Anordnung ermöglicht eine kleine, konstante, von der Knotenzahl unabhängige Verzögerung, benötigt aber $n * m$ Schaltelemente und ist nicht beliebig erweiterbar, da die physikalischen Eigenschaften der Leitungen die Knotenzahl begrenzen.

Permutations-Schaltnetzwerke:

Diese Netzwerke treten in sehr vielen (Namens-) Varianten auf: *Shuffle* (oder *Binary*) *Exchange Network, Banyan Network, Omega Network, Delta Network, Benes Network, FFT Network,* usw.

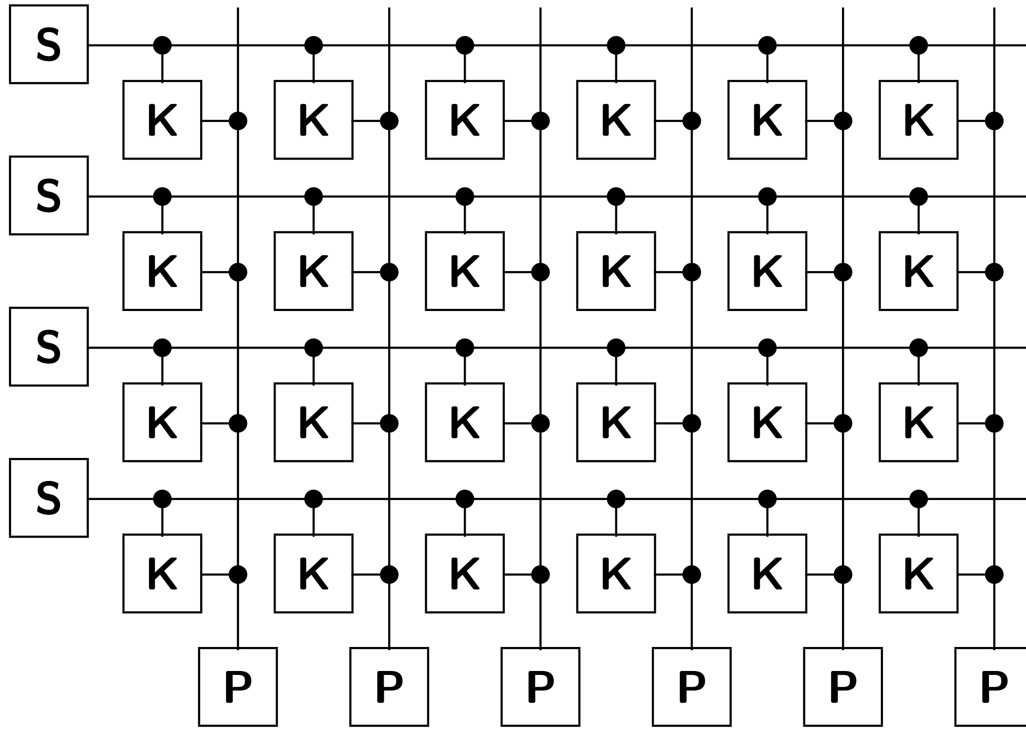


Abbildung 2.4: *Crossbar Switch*

Im wesentlichen haben sie aber eine einheitliche Struktur, nur die Details der Verbindungen sind unterschiedlich. Sie bestehen aus *Grundeinheiten*, die b Eingänge mit ebensovielen Ausgängen beliebig paarweise verbinden können (b ist üblicherweise 2 oder 4). Das gesamte Netz verbindet $n = b^t$ Eingänge mit ebensovielen Ausgängen und besteht aus t Stufen von jeweils n/b Grundeinheiten.

Die Eingänge der ersten Stufe sind die Eingänge des Gesamtnetzes, die Ausgänge der letzten Stufe seine Ausgänge, und dazwischen ist jede Stufe mit der folgenden so verbunden, daß im Endeffekt alle Wege möglich sind (hier unterscheiden sich die einzelnen Netze) (siehe Abbildung 2.5).

Der Vorteil ist, daß es im ganzen Netz nur Punkt-zu-Punkt-Verbindungen gibt und es damit leicht modular erweiterbar ist. Weiters benötigt man nur $O(n * \log(n))$ viele Elemente. Der Nachteil liegt in der mit $O(\log(n))$ wachsenden Laufzeit.

Weiters sind die heute üblichen Netzwerke von Kanälen mit Routing-Hardware auch als Schaltnetzwerke zu werten, hier gibt es ein sehr breites Spektrum[17].

[17] Diese Netze sind ein Beispiel dafür, daß Schaltnetzwerke nicht nur (wie man auf den ersten Blick oft annimmt) bei paralleler Datenübertragung, sondern mindestens ebenso häufig

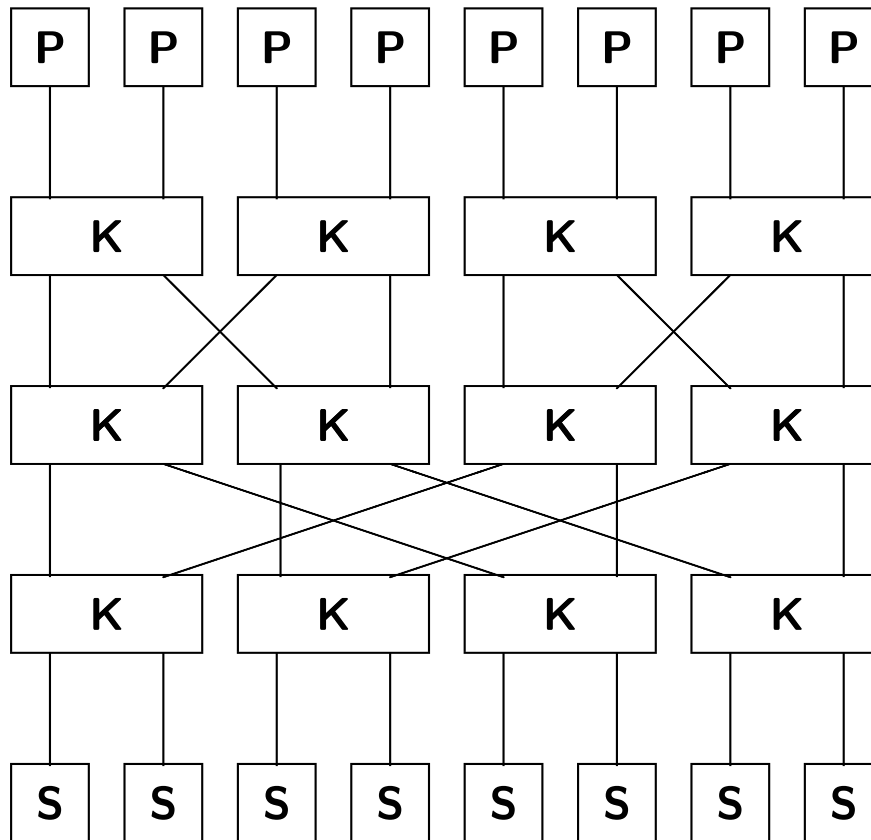


Abbildung 2.5: *Permutations-Schaltnetzwerk*

2.2.6 Kanäle

Kanäle bringen an sich aus der Sicht der Hardware kaum grundsätzliche Probleme, Daten physikalisch zwischen zwei fixen Punkten zu übertragen ist zentrale Aufgabe der Kommunikationstechnik und seit Jahrzehnten technisch gelöst. Kanäle sind typischerweise (aber nicht notwendigerweise) serielle Leitungen.

Um allerdings von einem Netz von Kanälen zu einem für den Anwender brauchbaren Kommunikationssystem zu kommen, ist eine Vielzahl weiterer Funktionen nötig:

Der direkte Speicherzugriff (DMA):

Er bewirkt, daß eine zu sendende Nachricht selbsttätig, d. h. ohne Zutun des Prozessors bei jedem einzelnen Byte, aus dem Speicher zur

auch bei seriellen Kommunikationsmedien zum Einsatz kommen.

Kanalhardware transportiert wird, und daß die Nachricht am anderen Ende der Kommunikation ebenso automatisch in einem dafür reservierten Speicherbereich abgelegt wird.

Der Prozessor selbst muß nur mehr die Übertragung initialisieren und kann sich dann anderen Prozessen zuwenden; ist die Übertragung beendet, löst die Kommunikationshardware einen Interrupt aus oder reiht den betroffenen Prozeß selbst wieder in die Warteschlange ausführbereiter Prozesse ein.

Das Routing:

Dieses umfaßt das Weiterleiten von Nachrichten zwischen nicht benachbarten Knoten auf den Zwischenknoten, aber auch das Umgehen defekter Knoten und Kanten, das gleichmäßige Verteilen der Kommunikationslast usw..

Anfänglich war *Packet-Switching* (auch *"store-and-forward"* genannt) üblich, d. h. die Nachrichten wurden von Knoten zu Knoten weitergereicht und dabei in jedem Knoten zwischengespeichert[18]. Diese Lösung benötigt keine zusätzliche Hardware, erfordert aber komplexe Software und belastet Speicher und Prozessor. Außerdem ist sie sehr langsam, da erst die ganze Nachricht fertig empfangen werden muß, bevor mit ihrer Weiterleitung begonnen werden kann.

Mit zunehmenden Hardware-Möglichkeiten setzte sich *Circuit-Switching* (auch *"Worm-Hole-Routing"* oder *"Virtual Circuit"* genannt) durch, d. h. unmittelbar vor jeder Nachrichtenübertragung wird eine (physikalische oder logische) Verbindung von einem Ende bis zum anderen durchgeschaltet; die Nachricht wird dann "in einem Ruck" übertragen, ohne daß die Zwischenknoten aktiv involviert sind. Diese Methode minimiert Verzögerungen, benötigt aber komplexere Hardware.

Das Multiplexing:

Hier geht es um die Durchführung mehrerer unabhängiger logischer Kommunikationen mehr oder weniger gleichzeitig auf einem einzigen Kanal. Dabei spielt (wie beim Routing) die *Deadlockfreiheit* und die *Fairness* eine wesentliche Rolle.

Das Buffering:

Hierbei handelt es sich um das zeitweilige Zwischenspeichern durchfließender, auf einen freien Kanal wartender, oder für einen noch nicht empfangswilligen Prozeß bestimmter Nachrichten.

---

[18] Um die Speicherverwaltung zu vereinfachen und allzu langes Belegen von Kanälen zu vermeiden, wurden die Nachrichten meist in Pakete fixer Größe unterteilt.

Ursprünglich wurden diese Funktionen softwaremäßig von den Prozessoren in den einzelnen Knoten durchgeführt. Dabei ergaben sich allerdings bald große Probleme:

- Typischerweise reichte die Geschwindigkeit der Prozessoren nicht aus, um vier, acht oder noch mehr schnelle Kanäle innerhalb der gewünschten Antwortzeit zu bedienen: Die Geschwindigkeit der Kommunikationshardware konnte nicht voll ausgenutzt werden, weil die Software zu langsam war, und die Ausführung des Anwenderprogramms kam zum Erliegen, weil die Prozessoren völlig mit Systemaufgaben ausgelastet waren.
- Für größere Netze, bei denen Nachrichten typischerweise über viele Zwischenknoten laufen, bewirkt die softwaremäßige Behandlung der Nachrichten in jedem Knoten in Summe viel zu lange Verzögerungen.

Man geht daher in letzter Zeit mehr und mehr dazu über, diese Funktionen ohne Zutun der Prozessoren in spezieller Hardware auszuführen. Diese Hardware ist sehr komplex[19], aber sie benötigt wesentlich weniger elektrische Leistung als (parallele) Busse. Auf Grund der selbstständigen Funktion läßt sich die Gesamtheit von Kanälen und Verwaltungshardware zu Recht als Schaltnetzwerk bezeichnen.

Netzwerken von Kanälen mangelt es an direkten Kommunikationsmöglichkeiten von jedem Prozessor nach außen (z. B. für Hardwarediagnose oder Monitoring und Debugging). Dem begegnet man in vielen Systemen durch einen globalen *Servicebus* oder ähnlichen von den normalen Kanälen unabhängigen Einrichtungen, über die ein Kontrollprozessor auf alle Knoten direkten Zugriff hat.

2.2.7 Topologien

An die Verbindungstopologie eines Parallelrechners ergeben sich Anforderungen aus verschiedenen Richtungen:

Aus der Sicht des Programmierers:

- Die Topologie soll *leicht vorstellbar* und *einfach zu beschreiben* sein. Im besonderen ist es für das Routing wichtig, daß man aus einer

[19] Bei modernen Systemen werden in jedem Knoten einige hunderttausend Gatterfunktionen nur für die Kommunikationslogik benötigt: Auf einem Transputerchip beispielsweise nimmt sie dreimal so viel Chipfläche wie die CPU selbst in Anspruch, und bei den INTEL Hypercubes ist in jedem Knoten eine eigene Platine mit zwei Dutzend speziell entwickelter Chips (die weit mehr kosten als der Rest des Knotens) für die Kommunikation zuständig.

gegebenen Zieladresse problemlos den Kanal ermitteln kann, auf dem die Daten abgeschickt werden müssen.

- Die Topologie soll logisch *unendlich* sein, d. h. man soll in jede Richtung beliebig weit "geradeaus" gehen können, ohne auf Ränder zu stoßen (Musterbeispiel: Torus). Das erspart das Programmieren von Sonderfällen für Randknoten.

- Die Topologie soll ein einfaches *Embedding* oder *Mapping*, d. h. eine einfache Abbildung beliebiger logischer Topologien auf die gegebene Hardware-Topologie, erlauben.

  Dazu zählt einerseits das *explizite Embedding* mittels eigener Befehle im Programm (oder durch einen intelligenten Compiler usw.), bei dem versucht wird, eine gegebene logische Topologie Knoten für Knoten möglichst optimal[20] auf eine ebenfalls bekannte Maschinentopologie abzubilden. Ein Musterbeispiel ist die H-Tree-Anordnung von binären Bäumen auf einem Rechteck.

  Andererseits gehört hierzu das *automatische Embedding*, bei dem es darum geht, die Hardwaretopologie so festzulegen, daß sich zu große oder unförmige logische Topologien in ihrer ursprünglichen Form auf Grund der Unendlichkeit der Hardware-Topologie von selbst möglichst gut auf der Hardware verteilen. Musterbeispiele sind Pipelines oder große Rechtecke, die sich in natürlicher Weise "in mehreren Lagen um einen Twisted Torus wickeln".

Aus der Sicht des Graphentheoretikers:

- Der *Grad* (d. h. die Anzahl der von einem Knoten ausgehenden Verbindungen) sollte möglichst klein und vor allem unabhängig von der Knotenzahl sein[21].

- Der *maximale Durchmesser* (d. h. die Anzahl der Kanten auf dem kürzesten Weg zwischen jenen beiden Knoten, die am weitesten voneinander entfernt sind) sowie der *mittlere Durchmesser* (d. h. die durchschnittliche Anzahl von Kanten auf dem kürzesten Weg zwischen allen möglichen Paaren von Knoten) sollen ebenfalls (betrachtet als Funktion der Knotenzahl) möglichst gering sein.

---

[20] Kriterien sind hier einerseits möglichst kurze Wege zwischen logisch benachbarten Knoten, andererseits eine möglichst gleichgroße Anzahl von Prozessen auf jedem Prozessor, d. h. keine leerstehenden Prozessoren und keine Prozessoren mit überdurchschnittlich vielen Prozessen.

[21] Hängt der Grad von der Knotenzahl ab, wie z. B. bei Hypercubes, muß bei einer Erweiterung des Gesamtsystems die Kommunikationshardware in den bestehenden Knoten um weitere Kanäle ergänzt werden.

- Die *Fehlertoleranz* (d. h. die Anzahl der Knoten oder Kanten, die ausfallen können, bevor der Graph in mehrere Teile zerfällt) soll möglichst groß sein.
- Der Graph soll *homogen* sein, d. h. jeder Knoten soll "die gleiche Sicht" des gesamten Graphen haben[22].
- Der Graph soll überall gleiche *Verbindungsdichte* haben, d. h. wenn man die kürzesten Wege zwischen allen möglichen Paaren von Knoten einzeichnet, sollen auf jedem Knoten und auf jeder Kante gleichviele Verbindungen zu liegen kommen.
- Der Graph soll in möglichst kleinen Schritten *erweiterbar* sein, ohne seine Eigenschaften zu verlieren (z. B. durch Hinzufügen einzelner Knoten anstatt nur durch Verdoppelung der Knotenzahl).

Aus der Sicht des Technikers:

- Die Topologie soll eine einfache, regelmäßige Abbildung auf die *Ebene* besitzen, um den Aufbau zu erleichtern.
- Diese Abbildung soll möglichst *kreuzungsfrei* sein und überall *gleiche Leitungsdichte* aufweisen, um die Leitungsführung zu vereinfachen.
- Weiters soll sie überall *gleiche Leitungslänge* aufweisen, da die Leitungslänge die elektrischen Eigenschaften der Leitung beeinflußt.
- Es soll eine einfache *Aufteilung* in Platinen, Einschübe, Schränke usw. möglich sein, und zwar mit möglichst vielen lokalen Verbindungen und möglichst wenigen zwischen den Aufbaueinheiten, da letztere meist zusätzlichen Hardwareaufwand erfordern.

Für Systeme mit Kanälen sind folgende Topologien gebräuchlich (siehe auch Abbildung 2.6, Abbildung 2.7 und Abbildung 2.8):

Pipeline, Ring:

Der einfachste, eindimensionale Fall. Vorteil: Einfach, nur Grad 2. Nachteil: Sehr schlechte topologische Eigenschaften.

Chordal Ring:

Das sind Ringe mit zusätzlichen, regelmäßig angeordneten Sehnen. Vorteil: Sehr gute topologische Eigenschaften, für jeden fix gegebenen Grad optimierbar. Nachteil: Schwer zu bauen (lange Leitungen), "unnatürliche" Topologie.

[22] Mathematisch läßt sich das in etwa wie folgt beschreiben: Numeriert man die Knoten durch, und gibt man einem Knoten dann eine beliebige neue Nummer, so muß es eine bijektive Abbildung der alten auf neue Knotennummern geben, sodaß der alte und der neue Graph isomorph sind.

	Grad	*Durchmesser*	*Erweiterung*
Ring	2	$\approx n$	$+1$
Chordal Ring	c	$\approx (n/c)$	$+1$
Torus usw.	4	$\approx \sqrt{n}$	$\approx +\sqrt{n}$
Baum	≤ 3	$\approx \log n$	$*2$
Recursive Diamond	4	$\approx \log n$	$*4$
Hypercube	$\log n$	$\approx \log n$	$*2$
Cube conn. Cycles	3	$\approx \log n$	$\approx *2$
Full Graph	$n-1$	1	$+1$

Abbildung 2.6: *Vergleich von Topologien*

2 D und 3 D Rechteckgitter:

Die “offensichtlichen” 2 D und 3 D Topologien. Vorteil: Leicht zu bauen, leicht verständlich, entspricht der Topologie vieler Anwendungen. Nachteil: Sehr schlechte topologische Eigenschaften, Ränder.

2 D und 3 D Torus:

Eine naheliegende Verbesserung. Vorteil: Bessere topologische Eigenschaften, keine Ränder, bessere Einbettung. Nachteil: Längere Leitungen.

Twisted Torus, Polymorphic Array, . . . :

Ähnlich dem Torus, aber die Ränder werden “versetzt” verbunden. Vorteil: Noch bessere Einbettung. Nachteil: Je besser, umso komplizierter.

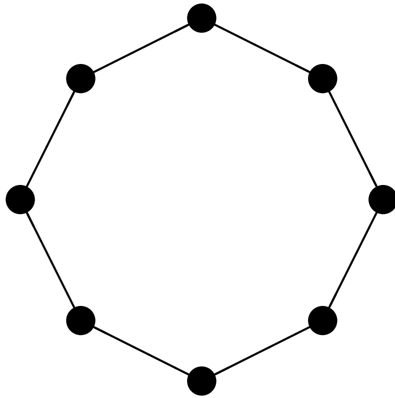
Bienenwaben- (Hex-) Gitter, Diamant- (Tetraeder-) Gitter:

Einfachste 2 D (3 D) Gitter: Nur Grad 3 bzw. 4. Vorteil: Minimaler Grad. Nachteil: Topologie für viele Anwendungen unbrauchbar, topologisch schlechter.

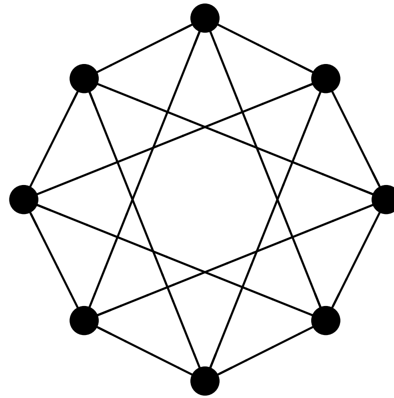
(Binär-) Bäume:

Eine aus graphentheoretischer und algorithmischer Sicht sehr naheliegende Topologie. Vorteil: Nur Grad 3, einfach aufzubauen, einfach vorzustellen. Nachteil: Sehr schlechte topologische Eigenschaften (Ränder, Engpässe, keine Fehlertoleranz), nur für wenige Anwendungen brauchbar.

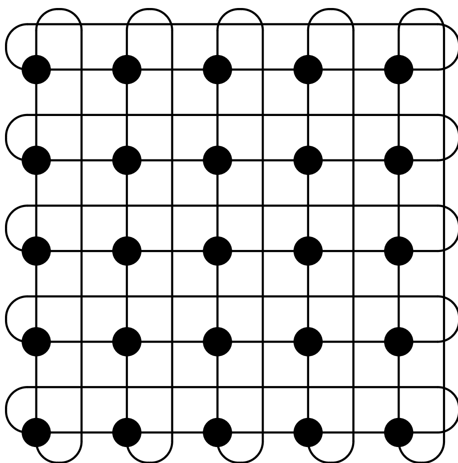
Hyperbäume, Multibäume:



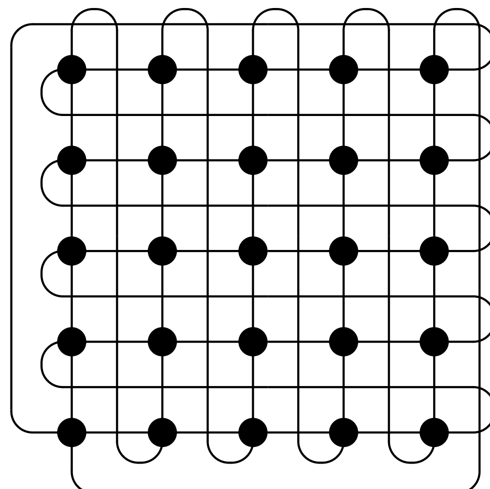
Ring



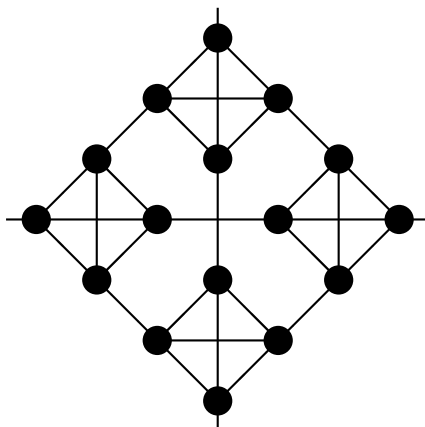
Chordal Ring



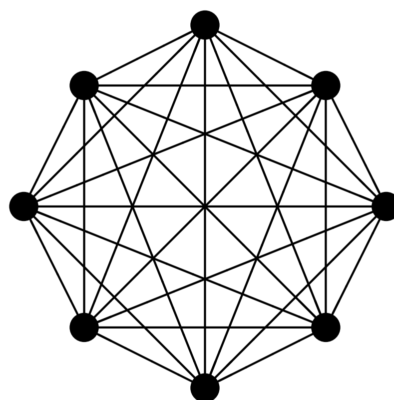
Torus



Twisted Torus

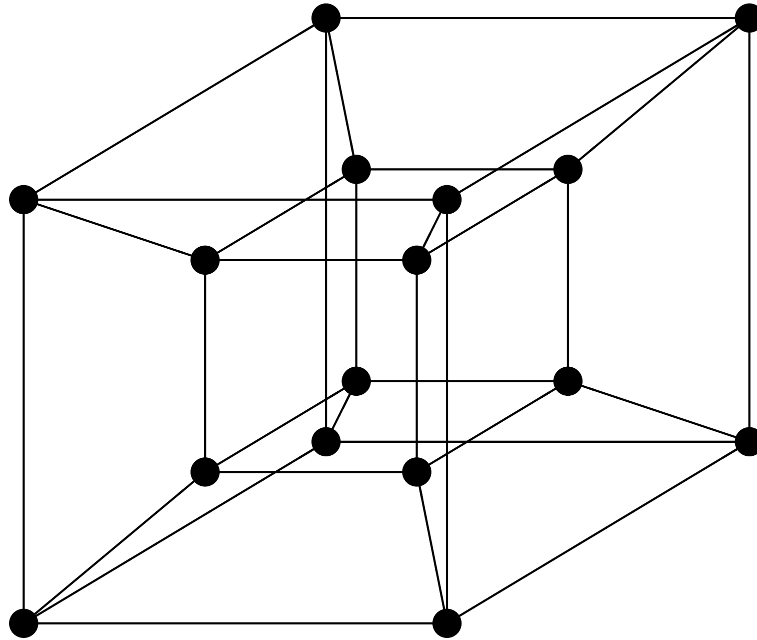


Recursive Diamond

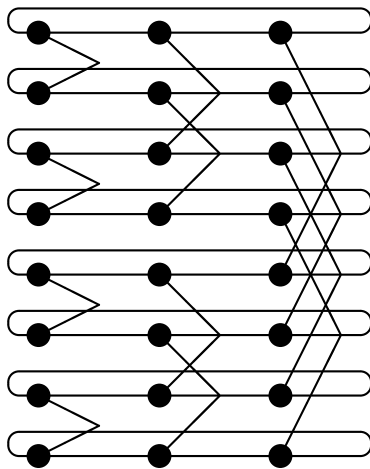


Full Graph

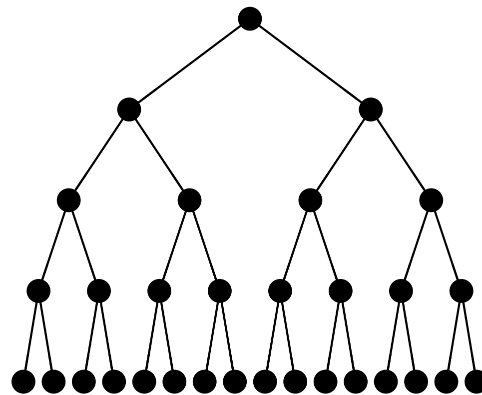
Abbildung 2.7: *Beispiele für Topologien 1*



Hypercube



Cube connected Cycles



Binärbaum

ABBILDUNG 2.8: *Beispiele für Topologien 2*

Hier handelt es sich um überlagerte Bäume oder um Bäume mit zusätzlichen Quer- und Ringverbindungen.

Recursive Diamond:

Eine Anordnung, bei der rekursiv jeweils vier Einheiten mit je vier Anschlüssen so zu einer neuen Einheit verbunden werden, daß jede mit jeder verbunden ist und vier Anschlüsse nach außen bleiben. Vorteil: Gute topologische Eigenschaften, jedes Modul hat nur vier Leitungen zu anderen Modulen. Nachteil: Schwer vorzustellen, Ränder.

Hypercube:

Eine ebenfalls rekursiv definierte Anordnung, bei der beginnend von einem Knoten zwei gleich große Hypercubes nebeneinandergestellt und ihre korrespondierenden Knoten paarweise verbunden werden (Dim. 0: 1 Knoten, Dim. 1: 1 Paar, Dim. 2: 1 Quadrat, Dim. 3: 1 Würfel, Dim. 4: Zwei ineinandergestellte und an den Ecken verbundene Würfel). Vorteil: Gute topologische Eigenschaften, gute Einbettung. Nachteil: Schwer zu bauen und zu erweitern (Grad wächst logarithmisch mit Knotenzahl).

Cube connected Cycles:

Eine rechteckige Anordnung von Knoten, bei der die Knoten waagrecht zu Ringen und senkrecht paarweise im Abstand von Zweierpotenzen verbunden sind (d. h. in der ersten Spalte mit dem Nachbarn, in der zweiten Spalte mit dem übernächsten, in der dritten Spalte mit dem Knoten vier Zeilen weiter usw.). Ein 3D Cube connected Cycle besteht beispielsweise aus $2^3 * 3$ Knoten, die wie bei einem Würfel mit schräg abgeschnittenen Ecken angeordnet und verbunden sind. Vorteil: Nur Grad 3, relativ gute topologische Eigenschaften (hypercubeähnlich). Nachteil: Sehr "unnatürlich" (Rechteck schwerer als beim Hypercube einzubetten), lange Leitungen.

Full Graph:

Jeder ist direkt mit jedem verbunden. Vorteil: Am leichtesten zu programmieren (Einbettung trivial). Nachteil: Grad n, n^2 viele Kanten, daher kaum zu bauen.

Permutations-Netzwerke:

Wie im Kapitel "Schaltnetzwerke" beschrieben, mit Knoten statt Schaltern. Eigenschaften: Ähnlich Cube connected Cycles. Verwendung: Nicht allgemein, praktisch nur für Spezialsysteme (FFT usw.).

Weiters gibt es noch zwei Möglichkeiten, Topologien zu variieren und zu kombinieren:

Potenzen:

Hier werden zwei gleiche oder verschiedene Typen von Topologien kombiniert, indem jeder Knoten die Summe der Einzelgrade als Grad hat und unabhängig voneinander Bestandteil beider Topologien ist. Ein Beispiel sind die oben erwähnten Bäume mit Querringen auf jeder Ebene.

Petersen's Graphs:

Hier wird eine Topologie mit einem Grad größer als drei variiert, indem man jeden Knoten durch zwei oder mehrere als Pipeline oder Ring verbundene Knoten kleineren Grades ersetzt. Nach dieser Methode lassen sich beispielsweise aus jedem Hypercube die entsprechenden Cube connected Cycles generieren.

2.3 Beispiele

2.3.1 Pipelines

Instruktions-Pipeline

Einfache Instruktions-Pipelines, d. h. das gleichzeitige Durchführen der verschiedenen zur Abarbeitung einer Instruktion notwendigen Schritte (Instruktion holen und dekodieren, Adressrechnung, Operanden holen, Operation ausführen, Ergebnis abspeichern, Befehlszeiger erhöhen) für aufeinanderfolgende Instruktionen, werden heute in praktisch jedem Prozessor verwendet, der *Befehlsstrom* fließt also durch die einzelnen Funktionseinheiten (siehe Abbildung 2.9).

Dabei gibt es allerdings zahlreiche Probleme, z. B. durch Operationen, die verschieden lang sind oder deren Ausführung verschieden lange dauert, durch Daten- oder Statusabhängigkeiten zwischen den Befehlen, durch Ladeoperationen aus dem Speicher, die ein Anhalten der Pipeline notwendig machen, oder durch bedingte Sprungbefehle, die überhaupt ein Abbrechen bereits begonnener Befehle und ein Neuladen der Pipeline erfordern (das ist der Grund, warum die harmlos aussehenden Sprünge bei den heutigen Prozessoren zu den zeitaufwendigsten Befehlen gehören).

Der Vorteil der in letzter Zeit populär gewordenen RISC-Prozessoren (MIPS R*xxxx*, Sun SPARC, Apollo DN 10000, Motorola MC 88000 usw.) mit Leistungen von bis zu 50 Mips auf einem Chip liegt darin, daß genau diese Probleme vermieden werden und die Pipeline optimal läuft, d. h. pro Takt ein Befehl ausgeführt wird:

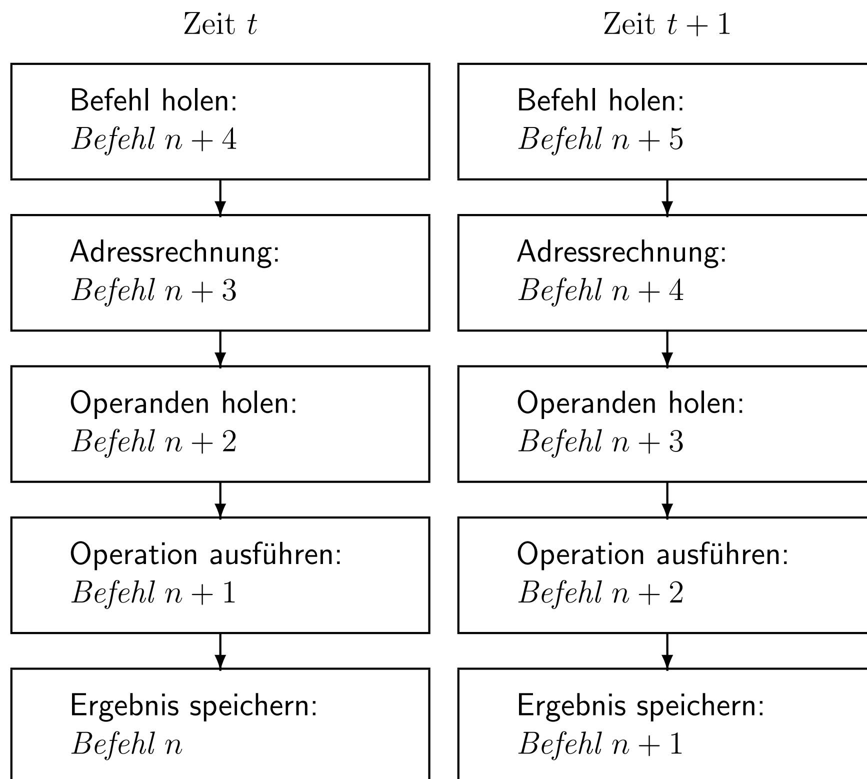


Abbildung 2.9: *Instruktions-Pipeline*

- Alle Befehle sind gleich lang und so einfach, daß sie in einem Zeittakt dekodiert und ausgeführt werden können.
- Es gibt nur wenige und sehr einfache Adressierungsarten, die ebenfalls alle in einem Takt berechnet werden können.
- Auf Datenabhängigkeiten wird zur Laufzeit nicht geachtet.
- Ladebefehle und bedingte Sprünge werden per definitionem nicht bereits beim nächsten, sondern erst beim übernächsten Befehl wirksam (*“delayed branch”*, *“delayed load”*), d. h. der Befehl unmittelbar nach einem bedingten Sprung wird in jedem Fall noch ausgeführt, und der Befehl nach einer Ladeoperation arbeitet noch auf den alten Daten.
- Ein großer Registersatz vermeidet unnötige Speicherzugriffe.
- Ein intelligenter Compiler versucht durch Umordnen des Codes, diese Eigenschaften optimal zu nutzen (notfalls müssen Leerbefehle eingefügt werden).

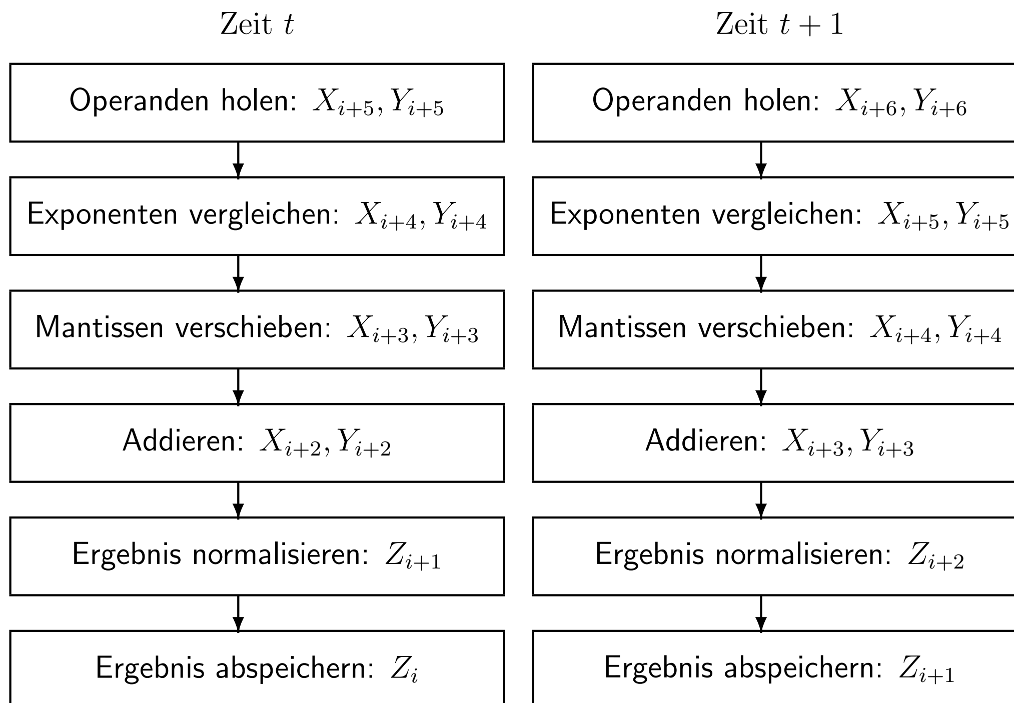


Abbildung 2.10: *Vektor-Pipeline: Gleitkomma-Addition, vereinfacht*

Vektor-Pipelines

Der nächste Schritt sind *Pipelines für Gleitkomma-Vektorarithmetik*, bei denen die einzelnen für eine Gleitkommaoperation notwendigen Schritte gleichzeitig auf aufeinanderfolgenden Elementen eines Vektors durchgeführt werden (hier fließen also die Elemente eines Vektors durch die einzelnen Funktionseinheiten, siehe Abbildung 2.10). Da Gleitkommaoperationen aus vielen Teilschritten bestehen und die Pipelines daher oft einige Dutzend Stufen haben, zahlt sich das Pipelining nur dann aus, wenn die Vektoren so lang sind, daß die Anlauf- und Entleerzeit der Pipeline am Beginn und Ende des Vektors nicht ins Gewicht fällt[23].

Diese Systeme sind heute ebenfalls Allgemeingut, man bekommt Vektor-Pipelines für 64 bit Gleitkommaarithmetik auf einigen wenigen Chips zu kaufen. Der i860 von INTEL ist ein Single-Chip-Mikroprozessor, der u. a. auch zwei 40 MFlops schnelle Vektor-Pipelines und Pipelines für grafische Operationen enthält.

[23] Eine bei Supercomputern oft verwendete Kenngröße gibt an, bei welcher Vektorlänge die Pipelines die Hälfte ihrer theoretisch möglichen Geschwindigkeit (für unendliche Vektoren) erreichen.

Derzeit gehen die Anstrengungen bei Vektorpipelines vor allem dahin, sie bezüglich Datenstrukturen universeller zu machen, sei es durch indirekte oder verkettete Adressierung (für "dünne" Matrizen), oder durch allgemeine Adressierungsfunktionen (für Band- und Dreiecksmatrizen). Das Problem dabei ist nicht die Pipeline selbst, sondern die Tatsache, daß die Speicherzugriffszeit auf nicht aufeinanderfolgende Daten sehr viel höher ist.

2.3.2 Besondere Prozessorarchitekturen

Der Denelcor HEP

Der Denelcor HEP war 1983 einer der ersten kommerziell verfügbaren Parallelrechner und erregte viel Aufmerksamkeit, ist aber inzwischen vor allem wegen mangelnder Softwareunterstützung wieder vom Markt verschwunden, obwohl er nach Aussage vieler Experten von allen konventionellen Maschinen die mit Abstand beste Unterstützung für Parallelität bietet. Er war offensichtlich seiner Zeit voraus.

Der HEP besteht aus maximal 16 Prozessoren, die mittels Schaltnetzwerk auf einen gemeinsamen Speicher (max. 1 GB) zugreifen. Zur Entlastung des gemeinsamen Speichers besitzt jeder Prozessor einen eigenen, lokalen Programmspeicher. Die Prozessoren arbeiten mit 10 MHz (konventionelle MOS-Technologie), die Gesamtleistung beträgt daher 160 Mips oder MFlops. Auch preislich und größenmäßig liegt der HEP im Bereich der damaligen Supercomputer.

Die Besonderheit des HEP liegt aber vor allem in der *Parallelität innerhalb eines einzelnen Prozessors*: Der Prozessor selbst ist wie konventionelle Prozessoren nach dem Pipelineprinzip aufgebaut, aber in den einzelnen Stufen werden nicht aufeinanderfolgende Befehle desselben Prozesses abgearbeitet, sondern *voneinander unabhängige Befehle verschiedener Prozesse*. Ein HEP-Prozessor ist imstande, bis zu 128 Prozesse hardwaremäßig zu verwalten und überlappend von jedem reihum je einen Befehl auszuführen. Der Prozessor besitzt also kein einzelnes Zustandsregister (Befehlszeiger und Statusflags), sondern es laufen bis zu 128 Prozeßzustände gleichzeitig in der ringförmigen Pipeline um.

Diese Unabhängigkeit der Befehle in den einzelnen Pipelinestufen hat einige Vorteile:

- Die oben beschriebenen Probleme durch Sprünge, Registerabhängigkeiten oder Statusabhängigkeiten treten nicht auf, aufeinanderfolgende Befehle oder Pipeline-Stufen können einander nicht beeinflussen. Die Pipeline ist daher einfacher, es gibt *kein Anhalten oder Entleeren der Pipeline.*

Um so viele Prozesse gleichzeitig und unabhängig voneinander ausführen zu können, besitzt der HEP in jedem Prozessor 2048 Register.

- Es kann eine sehr lange *Speicherzugriffszeit toleriert* werden (nämlich so lange, bis der Prozeß, der den Speicherzugriff auslöste, einmal die Pipeline durchlaufen hat und sein nächster Befehl ausgeführt wird), und es können während eines Speicherzugriffs *andere Befehle* abgearbeitet werden.

  Beim HEP geht man sogar noch weiter: Die Speicherzugriffs-Einheit nimmt einen Prozeß, der einen Speicherzugriff ausführt, einfach aus der Pipeline. Er wird intern zwischengespeichert und erst dann wieder in die Pipeline eingeschleust, wenn der entsprechende Speicherzugriff abgeschlossen ist. Jeder Prozessor kann auf diese Weise viele ausständige Speicherzugriffe gleichzeitig verwalten.

- Einzelne Prozesse können *Busy Waiting* (d. h. Warten auf ein Synchronisationsereignis durch ständiges Abfragen eines Zustandes in einer Schleife) betreiben, ohne daß der Gesamtdurchsatz wesentlich leidet: Bei n Prozessen kann ein einzelner nie mehr als $1/n$ der Zeit vergeuden, die anderen arbeiten normal weiter.

  Um diesen Vorteil zu nutzen, besitzt der HEP für jedes Register und jede Speicherstelle ein *voll/leer-Bit*, das softwaremäßig manipuliert werden kann und von der Hardware bei jedem Lesezugriff automatisch geprüft wird: Ist die zu lesende Zelle noch leer, wird der Lesebefehl unverändert in der Pipeline belassen, nach einem Pipeline-Durchlauf wird daher derselbe Lesezugriff wieder probiert, und das so lange, bis die Zelle einmal in vollem Zustand angetroffen wird[24].

Eines der größten Probleme beim HEP war die richtige Dosierung der Parallelität:

- Es gab keine vernünftige Möglichkeit, Prozesse aus der CPU-Pipeline auszulagern. Die Länge dieser Pipeline (mal der Anzahl der Prozessoren) war daher eine unüberwindliche obere Schranke für die Gesamtzahl der parallelen Prozesse im System.

- Selbst bei fast leerer Pipeline konnten die Prozesse nicht wesentlich schneller rotieren als bei gut gefüllter Pipeline. Bei sequentiellen Programmen konnte der eine verbliebene Prozeß daher nur einen Bruchteil der Prozessorleistung nutzen, in den restlichen Takten geschah nichts.

[24] Die Implementierung von Locks und Semaphoren ist damit sehr einfach möglich.

Der Multiflow

Der MULTIFLOW der gleichnamigen Firma, der 1987 auf den Markt kam und inzwischen wieder verschwand, ist das Musterbeispiel eines *VLIW* (*Very Long Instruction Word*) -Computers. Es handelt sich dabei um einen ganz normalen, sequentiellen 32-bit-Prozessor, bei dem die Struktur der Befehle so geändert wurde, daß alle Rechenwerke des Prozessors (z. B. Integer-ALU, Integer-Multiplizierer, Gleitkomma-Addierer und -Multiplizierer, Vergleicher, Speicherzugriffs-Einheit usw.; beim MULTIFLOW sind es bis zu 28 Funktionseinheiten, von denen einige auch mehrfach vorhanden sind) *von jedem Befehl unabhängig voneinander gesteuert werden.* Beim MULTIFLOW ergeben sich dadurch *bis zu 1024 bit lange Befehle.*

Das Ziel ist das gleiche wie bei Pipelines und dem HEP: Eine möglichst gute Ausnutzung der vorhandenen Hardware. Mit relativ geringem zusätzlichem Hardwareaufwand (größere Befehls-Einheit, großer Multiport-Registersatz) erreicht man so, daß jede Einheit bei jedem Schritt etwas tut, während bei konventionellen Prozessoren im Prinzip die gleichen Funktionseinheiten vorhanden sind, aber bei jedem Befehl nur eine davon benutzt wird.

Der Benutzer sieht auch keinen Parallelrechner, sondern eine ganz gewöhnliche UNIX-Maschine (sogar mit einem ganz gewöhnlichen UNIX-Debugger). Der *Compiler* übernimmt es, auf Grund von Datenfluß-Analysen den Code so umzuordnen, daß möglichst viele Einheiten gleichzeitig genutzt werden. Das geht sogar so weit, daß Sprünge vorhergesagt und Berechnungen "auf Vorrat" durchgeführt werden, um dann mit Hardware-Unterstützung ev. wieder rückgängig gemacht zu werden (der MULTIFLOW-FORTRAN-Compiler umfaßt mehr als 300.000 Zeilen C Code!!!).

In der Praxis wird bei typischen FORTRAN-Anwendungen eine Maschinenausnutzung von 75–90 % erreicht; es ist ganz erstaunlich, daß ein sequentieller 10 MHz Prozessor in ganz konventioneller Technologie (CMOS) bis zu 60 MFlops Dauerleistung erreichen kann.

Probleme gibt es allerdings bei FORTRAN-Code, der nicht von menschlichen Programmierern, sondern von irgendwelchen Preprozessoren oder Codegeneratoren stammt, die Auslastung sinkt auf bis zu 20 %, da vor allem die Sprungvorhersage mit dem "unnatürlichen" Sprungverhalten nicht zurecht kommt.

INTEL hat sich nach dem Ende des MULTIFLOW viele der Hardware- und Compiler-Konzepte gesichert: Der i870, der Nachfolger des i860, soll eine VLIW-Architektur auf einem Chip bieten und mit Hilfe derartiger Compilertechnologie programmiert werden. Der in einem eigenen Kapitel beschriebene iWARP von INTEL und der CMU verwendet bereits jetzt das VLIW-Prinzip, allerdings in geringerem Ausmaß.

Superscalar-Prozessoren

Zu dieser Klasse gehören einige der neuesten Mikroprozessoren, z. B. IBM's RS6000-Familie, der INTEL i860, oder der T9000 von Inmos. Auch hier wird versucht, zu jedem Zeitpunkt möglichst viele Einheiten des Prozessors zu nutzen, aber nicht, indem man in jedem Befehl mehrere Einheiten anspricht, sondern indem man aufeinanderfolgende Befehle, die unabhängige Einheiten benutzen, gleichzeitig ausführt.

Auch hier ist zur optimalen Nutzung dieser Fähigkeit ein maschinenspezifischer, sehr komplexer Compiler nötig, der die Abfolge der Instruktionen für diesen Zweck optimiert[25].

2.3.3 Systolische Arrays, Zellularautomaten

Systolische Arrays sind *geometrische, gitterförmige* Anordnungen (üblicherweise zweidimensional, z. B. rechteckig oder hexagonal) von einfachen, identen Rechenelementen, die *synchron* arbeiten und durch die entlang der Gitterkanten getaktet Informationen von Knoten zu Knoten gepumpt werden.

Ursprünglich waren die Rechenelemente *nicht* programmierbar und führten nur eine fixe Funktion aus (z. B. eine Multiplikation), und die Daten flossen durch das Gitter. Das klassische Beispiel ist die *Matrizenmultiplikation auf hexagonalem Gitter*.

Dann gestaltete man die Prozessoren programmierbar, das systolische Array näherte sich der SIMD-Maschine. Es blieb jedoch der wesentliche Unterschied, daß die einzelnen Zellen nicht in jedem Takt einen von einem zentralen Steuerwerk gelieferten Befehl ausführen, sondern daß bei jedem Takt die Daten in der gleichen Art und Weise fließen, und daß auf ihnen stets die gleiche, vorher programmierte Funktion ausgeführt wird (diese Funktion kann entweder aus rein kombinatorischer Logik oder aus einem endlichen Automaten bestehen).

Derzeit gibt es zwei kommerziell angebotene Chips, die eine Matrix programmierbarer Zellen enthalten. Auf einem Chip sind dabei typischerweise 1024 einfache endliche Automaten integriert.

Die letzte Überlegung geht in die Richtung, die Daten in den Knoten feststehen zu lassen (bzw. nur auf Befehl zum Nachbarn weiterzuschicken) und die Instruktionen durch das Gitter zu takten[26].

---

[25] Nur der T9000 enthält einen "Instruction Grouper", der Befehle hardwaremäßig beim Ausführen umordnet und zusammenfaßt und dadurch auch bei Binärcode, der für die Vorgängerprozessoren compiliert wurde, eine recht gute Prozessorauslastung erzielt.

[26] In einer zweiten Richtung können dann ev. noch Instruktionsmasken geschoben werden, um die durchfließenden Instruktionen an bestimmten Prozessoren wirksam oder unwirksam werden zu lassen.

2.3.4 SIMD-Computer

SIMD-Computer bestehen aus einem *einzigen Steuerwerk* mit Programmspeicher, das wie bei einem konventionellen Prozessor für das Laden und Dekodieren von Befehlen zuständig ist[27], aber aus *vielen Rechenwerken* mit jeweils lokalen Registern und lokalem Speicher, die alle *gleichzeitig* dieselbe vom Steuerwerk befohlene Operation auf ihrem *lokalen* Speicher ausführen. Meistens sind diese Rechenwerke auch in Form eines Torus *nachbarlich miteinander verbunden*, um alle auf einen Ruck, d. h. in einem Schritt, in die gleiche Richtung Daten weiterzuleiten (siehe Abbildung 2.11).

Die Wortbreiten von Steuerwerk und Rechenwerken sind voneinander unabhängig: Das Steuerwerk und der Programmspeicher sind immer relativ konventionell aufgebaut (z. B. 16 oder 32 bit breit), während die einzelnen Rechenwerke oft nur aus *1 bit Prozessoren* bestehen. Das bedeutet unter anderem, daß Integer- und Gleitkommaoperationen sequentiell, d. h. Bit für Bit, durchgeführt werden müssen und daher aus vielen einzelnen Befehlen bestehen (dafür wird die Operation auch auf einem ganzen Array von Zahlen auf einmal durchgeführt).

Um dieses Konzept etwas flexibler zu machen, besitzen fast alle im folgenden genannten Maschinen zwei Hardware-Erweiterungen in jedem Knoten:

Instruction-Enable-Flag:

Mit dieser Einrichtung ist es möglich, abhängig von vorausgegangenen boolschen Operationen Instruktionen bedingt auszuführen; abhängig von diesem Flag und einem Bit im Befehl führt jede Zelle für sich die momentane Operation aus oder ignoriert sie.

Indirect Adressing Register:

Dieses Register erlaubt das Berechnen und Verwenden indirekter Adressierung und damit flexiblere Datenstrukturen. Es führt zwar in einem Takt nach wie vor jede Zelle dieselbe Operation aus, aber auf jeweils anderen Elementen des lokalen Speichers.

Diese Rechner werden nicht universell eingesetzt, sondern üblicherweise von einem Frontend-Rechner aus betrieben. Auf ihnen läuft daher auch kein eigenständiges Betriebssystem, sondern nur jeweils ein einzelnes Applikationsprogramm. Der Einsatzbereich umfaßt Anwendungen, die Datenparallelismus auf dichten Feldern und Matrizen mit lokalem Verhalten bieten (numerische Anwendungen, Bild- und Signalverarbeitung, Suchalgorithmen usw.). Da nur die Datenstrukturen parallelisiert werden, der Programmablauf selbst aber

[27] Es führt auch alle skalaren Befehle selbst aus.

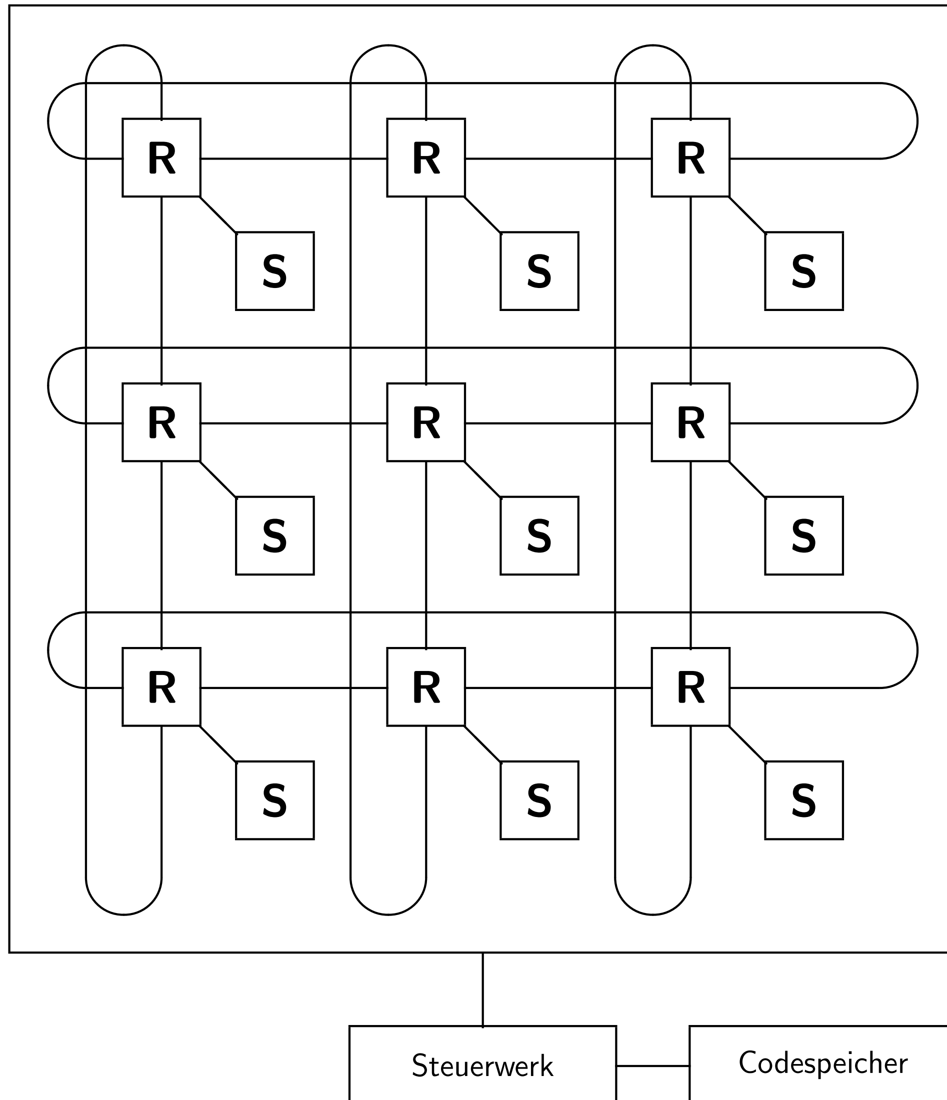


Abbildung 2.11: *SIMD Struktur*

strikt sequentiell bleibt, treten viele mit paralleler Programmierung verbundene Probleme bei diesen Rechnern nicht auf.

Durch den einfacheren Aufbau der einzelnen Knoten und die strikt synchrone Arbeitsweise sind SIMD-Rechner üblicherweise schneller und billiger als gleichgroße MIMD-Systeme.

Der ILLIAC-4

Der ILLIAC-4 war der erste *Supercomputer* mit einer Spitzenleistung von über 100 MFlops. Der Rechner wurde vor allem für Strömungssimulationen entwickelt. Es wurde nur ein Prototyp gebaut und 1972 im Ames Research Center der NASA in Betrieb genommen[28].

Der Rechner enthielt 64 Prozessoren, die in einem $8 * 8$ *Torus* angeordnet waren und jeweils aus einem 64 bit Gleitkomma-Rechenwerk bestanden. Diese Prozessoren ließen sich auch als 128 32 bit Gleitkommaprozessoren verwenden.

Der ILLIAC-4 füllte eine eigene Halle; die Technologie war für heutige Verhältnisse steinzeitlich: 80 ns Zykluszeit, 256 bit (!) Speicherchips, nur 1 MB RAM insgesamt, dafür aber ein ganzes Feld von großen, untereinander synchronisierten Festkopf-Magnettrommelspeichern (d. h. pro Spur ein eigener Kopf), die eine Dauertransferrate von mehr als 50 MB/s boten — viel mehr als die ersten Cray's und CDC's.

Der IBM GF-11

Dieser Rechner wird im IBM Labor Yorktown Heights entwickelt. Auch er füllt eine große Halle, und zwar mit 576 individuellen 64 bit Gleitkommaprozessoren (davon 64 in Reserve, um Ausfälle zu kompensieren), die über ein *Schaltnetzwerk* verbunden sind.

Dieser Rechner ist auf physikalische Anwendungen spezialisiert, bei denen man trotz der Leistung von 11 GFlops mit bis zu einem Jahr Laufzeit rechnet. Es wurde daher großes Augenmerk auf Fehlertoleranz und Ausfallsicherheit gerichtet.

Der AMT DAP

Dieser Rechner ist als *Koprozessor für Workstations* gedacht, ist also wesentlich billiger und paßt unter den Schreibtisch. Er wurde von der englischen Computerfirma ICL vor einiger Zeit entwickelt und wird jetzt von der ebenfalls englischen Firma Active Memory Technologies gebaut und vertrieben.

Er enthält maximal 4096 1 bit Prozessoren, die in einem zweidimensionalen $64 * 64$ *Torus* angeordnet sind. Jeweils 64 solcher Prozessoren sind auf einem einzigen CMOS-Chip integriert. Es stehen insgesamt maximal 64 MB Speicher zur Verfügung.

[28] Es dauerte allerdings bis 1985 (lang nach seiner Demontage), bis der letzte vom ILLIAC gehaltene Geschwindigkeitsrekord fiel.

Die Taktrate beträgt 10 MHz, der Rechner bietet also abhängig von der Wortbreite der zu verarbeitenden Daten eine Spitzenleistung von 200 MFlops oder 2000 Mips (Integer) oder 40 Giga Pixel Instruktionen pro Sekunde (Grafik).

Der Rechner bietet zwei zusätzliche Hardwareeinrichtungen:

- Eine *Broadcast-Möglichkeit*, um die korrespondierenden Speicher aller Prozessoren mit einem einzigen Befehl zu laden.
- Eine *parallele I/O-Einrichtung*, die maximal 400 Mbit/s durch das Prozessorfeld schleust, z. B. für Grafik oder Signalverarbeitung.

Die Thinking Machines Connection Machine

Ursprünglich wurde das Konzept am MIT für AI-Anwendungen entwickelt, über die CM-1 führte die Entwicklung dann zur hier beschriebenen CM-2, einem kommerziellen Produkt von Thinking Machines Cooperation.

Dieser Rechner besteht aus 2^{16} Prozessoren, ebenfalls nur 1 bit breit, von denen je 16 auf einen Chip passen, und aus insgesamt maximal 512 MB Speicher.

Er hat viele Besonderheiten:

- Je 32 Prozessoren sind zusätzlich mit einem eigenen Weitek *Gleitkommaprozessor* verbunden.
- Jeder Hardwareprozessor kann fast beliebig viele *"virtuelle Prozessoren"* emulieren, ohne daß die Software irgend etwas davon merkt (außer dem verringerten Speicherplatz pro virtuellem Prozessor). Man kann daher Programme schreiben, die eine Million Prozessoren oder mehr verwenden.
- Es gibt *zwei unterschiedliche Kommunikationsnetzwerke*: Ein einfaches, schnelles, nachbarschaftliches, das auf einem zweidimensionalen Torus basiert, und ein langsames, das Leitungen in Hypercube-Topologie benutzt, aber durch hardwaremäßiges Routing beliebige Punkt-zu-Punkt-Kommunikationen erlaubt. Der Gesamtdurchsatz der Kommunikation beträgt 3 GB/s.
- Für Ein- und Ausgabe stehen verteilte, parallele Disk- und Grafiksysteme zur Verfügung.

Der Rechner liegt preis- und größenmäßig im Bereich von *Supercomputern*, bei speziellen Anwendungen bringt er auch eine Dauerleistung von bis zu 20 GFlops.

Mit derart massiv parallelen Systemen entwickelten sich auch *viele neue Algorithmentypen und -ideen*: Simulierte man Strömungsmodelle bisher beispielsweise numerisch, so ist es auf der CM möglich, den Raum in genügend feine Gitterpunkte zu teilen, die auf jeweils einen (virtuellen) Prozessor abgebildet werden, und das strömende Medium durch herumfliegende Partikel direkt darzustellen: Liefert eine Kommunikation ein 1-Bit, wird das als eintreffendes Teilchen interpretiert; ein 0-Bit steht im wahrsten Sinn des Wortes für nichts. Daraus ermittelt jeder Prozessor, wann und in welche Richtungen er Teilchen weiterzumelden hat.

2.3.5 Klassische Supercomputer

Die Zeit der Vektor-Supercomputer begann um 1978 mit der Cray-1 und der Control Data Cyber 205. Heute ist bereits die zweite Generation am Markt (Cray X-MP, Y-MP und 2, CDC Eta-10, sowie Systeme von Fujitsu, Hitachi, IBM und NEC).

Sie haben folgende gemeinsame Kennzeichen:

- Verwendung *extrem schneller* (Taktzeiten rund 5 ns), aber *niedrig integrierter* Logik (ECL, bipolar; oft nur einige Dutzend Transistoren pro Chip).
- Großer *Platzbedarf* (einige Schränke), großer *Leistungsbedarf* (oft über 100 KW), oft aufwendige Kühlverfahren[29].
- *Rechengeschwindigkeit* 300 MFlops bis 10 GFlops, *Kosten* von 10 Mill. US $ aufwärts.
- Ursprünglich einer, heute bis zu *16 Prozessoren*.
- Jeder Prozessor ist mit einigen *Vektor-Pipelines* ausgerüstet.
- Ein *großer* (max. 4 GB), *globaler, homogener Speicher* mit trickreichen Kombinationen von *Vektorregistern*, *Caches*, *Pipeline-Bussen* und *Speicherbänken*, um die für die Pipelines benötigte Datenrate bereitzustellen (mehr als 1 GB/s!)[30].

---

[29] Nur die Fujitsu ist luftgekühlt; die Cray-2 ist in Freon tauchgekühlt; die Eta-10 ist in flüssigem Stickstoff tauchgekühlt (Temperaturen um −200 Grad Celsius haben den erwünschten Nebeneffekt, daß CMOS-Schaltkreise fast dreimal so schnell arbeiten wie bei Zimmertemperatur); die anderen Systeme sind mit Wasser oder Freon röhrengekühlt.

[30] Ein besonders leuchtendes Beispiel ist der Alliant Minisupercomputer: Bis zu 8 Prozessoren greifen über einen Crossbar-ähnlichen Switch auf 4 gemeinsame, voneinander unabhängige Caches zu, die wiederum über einen gemeinsamen Bus auf den in Bänken organisierten Hauptspeicher Zugriff haben. Unabhängig davon haben noch maximal 12 I/O-Prozessoren auf diesen Hauptspeicher direkten Zugriff.

- Zusätzlich oft ein noch größerer, aber billigerer und langsamerer *Erweiterungsspeicher* (zur Entlastung der Plattenspeicher).
- Hardwareunterstützung für *Parallelität* und *Synchronisation*, vor allem für feinkörnige Parallelität und für *parallele Abarbeitung von Schleifen.*
- Anschlußmöglichkeit für ein Hyperchannel-Netzwerk (ein genormtes lokales Netz im Gigabit-Bereich speziell für Supercomputer).

Diese Systeme sind auf Grund ihrer Architektur (Vektor-Pipelines) nur für die Verarbeitung *numerischer Daten in Vektor- oder Arrayform* gut geeignet, dafür ist für diesen Bereich die Software schon relativ ausgereift (Unix- oder IBM-kompatible Betriebssysteme, *automatisch vektorisierende und parallelisierende Compiler für* Fortran *und* C, parallele und vektorielle, in Assembler geschriebene *Unterprogramm-Bibliotheken* für Vektor- und Matrizenoperationen, sowie viele fertige Anwenderpakete).

Bei günstigen Anwendungen läßt sich damit eine Maschinenausnutzung (d. h. eine Rechengeschwindigkeit relativ zur theoretischen Höchstgeschwindigkeit) von bis zu 95 % erreichen, beim ersten Versuch kommt man aber meist kaum über 10 %, und mit 50 % muß man schon sehr zufrieden sein. Die theoretische Höchstgeschwindigkeit für rein skalare Anwendungen ist meist nur ein Zehntel der Vektorleistung oder weniger, für symbolische Anwendungen ist die Performance auf Grund der ungeeigneten Speicherstruktur oft noch wesentlich schlechter.

Minisupercomputer und Grafiksupercomputer

In neuerer Zeit hat sich eine neue Systemklasse herausgebildet, die statt maximaler Leistung um jeden Preis unter Ausreizung der technologischen Möglichkeiten versucht, mit *derselben Architektur wie Supercomputer* ein *optimales Preis/Leistungsverhältnis* zu erreichen: Die Minisupercomputer (wichtigste Vertreter: Convex, Alliant).

Sie bringen rund *ein Zehntel der Leistung eines State-of-the-Art Supercomputers*, sind aber wesentlich kleiner und billiger, da sie anstatt extrem schneller, niedrig integrierter Logik *hochintegrierte Bauelemente* verwenden und üblicherweise mit *Luftkühlung* auskommen[31].

---

[31] Aktuellstes Beispiel ist die Convex C3: Sie verwendet in den meisten zeitkritischen Funktionseinheiten Gallium-Arsenid-Technologie. Diese ist zwar etwas langsamer als die schnellstmöglichen herkömmlichen Halbleiter, erlaubt aber wesentlich höhere Integrationsdichten und erzeugt nur einen Bruchteil der Abwärme.

An einigen Orten wurden bereits bestehende Supercomputer durch eine C3 an Stelle eines Supercomputers der neuesten Generation ersetzt; die C3 dürfte in etwa jene Leistung bieten, die vor 2–3 Jahren von den damaligen Supercomputern erreicht wurde.

Der nächste Schritt sind Minisupercomputer, deren Prozessoren auf einem Chip integriert sind: Alliant und intel propagieren gemeinsam den *PAX-Standard* als Grundlage paralleler Software, der aus einer Kombination des i860-Befehlssatzes mit den von Alliant entwickelten Befehlen zur Parallelitätsverwaltung und Synchronisation besteht. Es existiert auch schon ein Zusatzchip zum i860, der diese Parallel-Befehle implementiert. Alliant hat bereits die Serie FX/2800 vorgestellt, die maximal 28 i860 Prozessoren in einem Gerät vereint.

Die *Hochleistungs-Grafiksubsysteme für Workstations* liegen leistungsmäßig ebenfalls im Supercomputerbereich (50–500 Mips bzw. MFlops), haben aber auf einigen wenigen Boards Platz und sind wegen ihrer spezialisierten Hardware auf Grafikanwendungen beschränkt[32]. Auch hier wird üblicherweise eine Kombination von *Vektorpipelines* und *Parallelität mit einigen wenigen Prozessoren* eingesetzt.

2.3.6 Flache Shared-Memory-Systeme

Neben zahlreichen Forschungsprojekten fallen in diesen Bereich die folgenden kommerziellen Systeme: Der Multimax der Firma Encore und die Balance sowie die Symmetry der Firma Sequent. Diese Systeme gibt es seit etwa 1985, sie liegen leistungsmäßig zwischen Mainframes und Supercomputern (wenn die Prozessoren mit Gleitkomma-Zusatzhardware ausgestattet sind, werden Leistungen von 200 MFlops erreicht), preislich und größenmäßig aber deutlich darunter. Das Haupteinsatzgebiet liegt vor allem im Bereich von großen Unix-Servern und Datenbanksystemen, wo sie konventionellen Rechnern weit überlegen sind (durch mehrere, parallele I/O- und Disk-Prozessoren wird auch eine entsprechende I/O-Leistung geboten).

Ausschließlich in der kommerziellen Datenverarbeitung ist die Firma Tandem tätig, die hochausfallsichere Systeme für Datenbankanwendungen mit derartiger Struktur herstellt.

Diese Systeme bestehen üblicherweise aus bis zu 32 32 bit Mikroprozessoren (intel, National Semiconductor, Motorola), eventuell mit Gleitkomma-Zusatzhardware, die mit großen, aufwendigen *Caches* ausgerüstet sind und über einen *gemeinsamen Bus* auf einen *globalen Speicher* zugreifen. Sie sind vergleichsweise einfach aufgebaut.

Eine ganz ähnliche Architektur, aber einen ganz anderen Hintergrund haben *Super-Workstations* (Silicon Graphics, Solbourne): Sie verwenden weniger (max. 8), aber wesentlich leistungsfähigere RISC-Prozessoren (MIPS

[32] Einige dieser Systeme enthalten nicht einmal universell programmierbare Prozessoren, sie bieten nur einige fest verdrahtete Operationen.

R3000, Sun Sparc), sonst aber die gleichen Konzepte, um Leistungen im Bereich von 300 Mips für technisch-wissenschaftliche Anwendungen, Grafik und Workstation-Netz-Server zur Verfügung zu stellen.

Der iAPX432

Dieses Projekt von intel in den Jahren 1980 bis 1985 resultierte im ersten kommerziell verfügbaren, speziell für Parallelrechner entworfenen Mikroprozessor. Obwohl dieser Prozessor sowohl wegen seiner Architektur als auch wegen seiner Integrationsdichte ein Meilenstein war, blieb der kommerzielle Erfolg auf Grund zu geringer Geschwindigkeit aus[33].

Es handelte sich beim iAPX432 um einen 32 bit Prozessor mit extrem komplexem Befehlssatz (der Höhepunkt der CISC-Entwicklungen[34]) und sehr aufwendigen Speicherverwaltungs- und Schutzmechanismen; alle in Hochsprachen und Betriebssystemen üblichen Objekte und Datenstrukturen wurden direkt von der Hardware verwaltet (eines der Hauptziele war eine optimale Unterstützung von Ada).

Der Prozessor war für den Aufbau von Parallelrechnern mit globalem Speicher gedacht, er unterstützte Parallelität, Kommunikation und Synchronisation in Hardware. Sogar transparente Parallelität war möglich: Es konnten zur Laufzeit Prozessoren hinzugefügt oder entfernt werden, und die Software verteilte sich automatisch auf die gerade zur Verfügung stehenden Prozessoren. Auch ein Redundanz-Modus für Hochsicherheitssysteme wurde hardwaremäßig unterstützt: Alle Anschlüsse zweier Prozessoren wurden paarweise miteinander verbunden, die beiden Prozessoren arbeiteten völlig synchron. Einer betrieb seine Pins wie üblich, während der andere alle seine Ausgänge in Eingänge verwandelte und die von seinem Partner erzeugten Signale mit seinen eigenen verglich. Ergab sich ein Unterschied, wurde ein Fehler signalisiert.

2.3.7 Andere Shared-Memory-Systeme

Der BBN Butterfly

Dieser Rechner der Firma BBN (Bolt-Beranek-Newman) aus dem Jahr 1987 besteht aus maximal 256 MC 68020 / MC 68881 Prozessoren mit je 4

[33] Es gelang intel nicht, den Prozessor auf einem einzigen Chip unterzubringen. Es wurden zwei Chips benötigt, und die Kommunikation zwischen diesen erwies sich als großer Engpaß.

[34] Die Befehle hatten beispielsweise bitweise variable Länge; manche konnten hundert und mehr Bits umfassen.

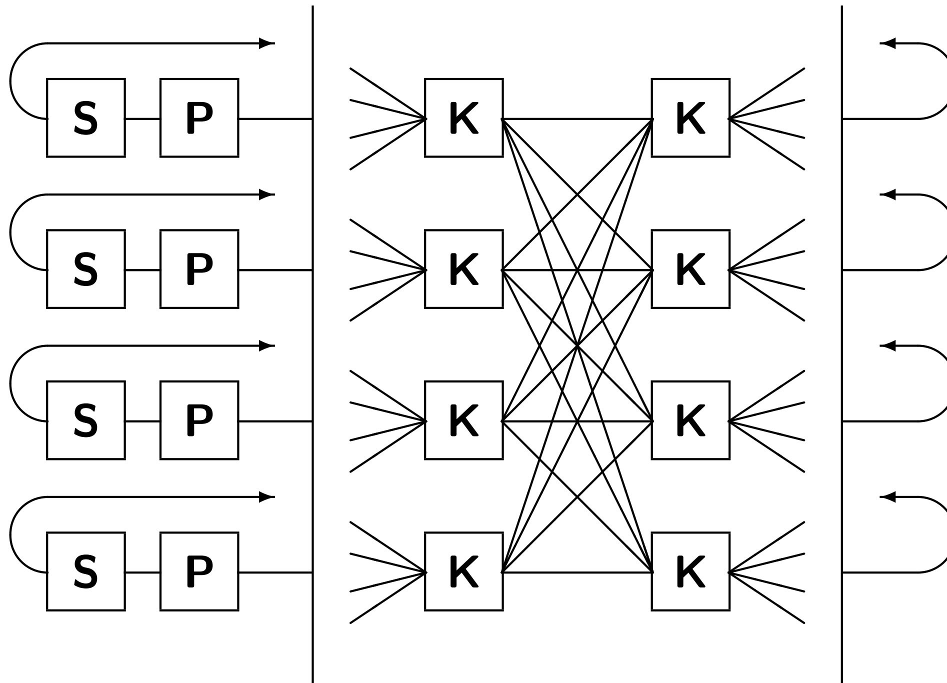


Abbildung 2.12: *Der* BBN Butterfly

MB RAM, wobei jeder Prozessor über ein *Banyan-Netzwerk* auf jeden Speicher zugreifen kann[35]. Um Platz zu sparen, ist der Switch mit 4 bit breiten Datenpfaden ausgestattet, die 32 bit Daten und Adressen werden darauf stückchenweise (und relativ langsam) übertragen[36]. Dafür war es möglich, ein $4 * 4$ Switchelement auf einem einzigen Chip unterzubringen. Die Geschwindigkeit beträgt rund 600 Mips.

Der IBM RP-3

Hier handelt es sich um ein Forschungsprojekt für Supercomputer im IBM-Labor Yorktown Heights.

512 IBM/RT Prozessoren (32 bit RISC Single-Chip Prozessoren) mit Vektor-Koprozessor, Cache und Memory Management Unit werden über ein intelligentes Schaltnetzwerk mit einem *globalen Speicher* verbunden.

Das Netzwerk und der Speicher bieten folgende Fähigkeiten:

[35] Auf seinen lokalen Speicher kann jeder Prozessor direkt, d. h. ohne Netzwerk, zugreifen, siehe Abbildung 2.12.

[36] Ein Zugriff auf den lokalen Speicher ist zumindest zwanzigmal schneller.

- *Fetch-and-add* Grundoperation.
- *Kombination simultaner Zugriffe*, auch für Fetch-and-add Zugriffe.
- *Verwaltung mehrerer ausständiger Speicherzugriffe* für jeden Prozessor.
- *Spezielle Logik für Cache und MMU Consistency.*

Diese Switches sind *sehr aufwendig*, sie nehmen um einiges mehr Platz ein als die Prozessoren und der Speicher und sind *wassergekühlt*. Man entschloß sich, die Funktionen auf *zwei getrennte Switch-Systeme* aufzuteilen: Ein schnelles, einfaches, niedrig integriertes für die trivialen Operationen (Banyan-Switch mit $4 * 4$-Elementen), und ein langsames, hoch integriertes für die komplexen Operationen (Omega-Switch mit $2 * 2$-Elementen).

2.3.8 Hierarchische Systeme

Der Cedar

Der Cedar der University of Illinois at Urbana/Champaign baut auf dem Alliant FX/8 auf. Dieser Rechner (ein Minisupercomputer) enthält 8 Prozessoren mit Vektor Pipelines, die über ein kompliziertes Bus- und Cachesystem auf einen gemeinsamen Speicher zugreifen.

Für den Cedar werden maximal 128 solcher Alliant FX/8 über einen *Omega Switch* mit 1024 Anschlüssen mit einem *globalen Speicher* verbunden (dazu wird der Speicher in jeder Alliant mit zusätzlichen Interfaces ausgestattet). Als Besonderheit bietet der globale Speicher *full/empty-Bits* ähnlich dem HEP.

Der Suprenum

Der Suprenum, ein staatlich angeregtes und gefördertes Projekt in Deutschland (GMD, FIRST Berlin, Suprenum GmbH), sollte in einem System resultieren, das vor allem für Mehrgitter-Methoden (ein Näherungsverfahren für Differentialgleichungen in der Strömungslehre) eine Rechenleistung im GFlop-Bereich zur Verfügung stellt.

Ein besonderer Schwerpunkt wurde bei diesem Projekt auf die *Software* (High-Level-Sprachen, intelligente Compiler, usw.) gelegt.

Im Unterschied zum Cedar handelt es sich beim Suprenum um ein System mit verteiltem Speicher. Die Hardware besteht aus maximal 256 *Clustern*, die durch serielle Kanäle zu einem zweidimensionalen $16 * 16$ *Torus* verbunden

sind. Jeder Cluster besteht aus 16 Prozessoren an zwei gemeinsamen Bussen[37] sowie einem *Plattenspeicher* und umfangreicher *Monitoring- und Debugginghardware*. Jeder Prozessor besteht aus einem MC 68020 mit einer Weitek Vektor Pipeline und 8 MB RAM sowie spezieller Kommunikationshardware.

Die Hardware wurde kurz nach ihrer Fertigstellung und der Auslieferung von zwei Kleinsystemen wieder vom Markt genommen, sie war ein Beispiel für ein Projekt, das zum Zeitpunkt seiner Beendigung bereits überholt war.

2.3.9 Hypercubes

Die iPSC Hypercubes

Die intel iPSC1 und iPSC2 Hypercube-Systeme haben ihren Ursprung in Entwicklungen des Caltech (*"Caltech Cosmic Cube"*). Sie umfassen preislich, größenmäßig und leistungsmäßig den Bereich von Superworkstations bis zu Supercomputern (Spitzenleistung 5 GFlop).

Der iPSC1 bestand im wesentlichen aus maximal 128 i80286/87 Prozessoren, die mittels handelsüblicher Ethernet-Kontroller (8 pro Knoten) miteinander kommunizierten.

Der iPSC2 umfaßt maximal 128 i80386/87 Prozessoren mit Cache und 1–16 MB RAM sowie (optional) einem Weitek Skalar- oder Vektor-Koprozessor sowie einem SCSI Platteninterface mit eigenem Prozessor.

Jeder Knoten hat noch immer 8 *serielle Verbindungen*: 7 für den Hypercube und 1 für Eingabe/Ausgabe-Funktionen. Allerdings wurden die Ethernet-Kontroller durch einige sehr komplexe speziell für den iPSC entwickelte Chips ersetzt. Einerseits verdoppelte das die Geschwindigkeit (jetzt 20Mbit/s), andererseits wurden alle für Routing, Multiplexing, DMA usw. nötigen Funktionen in die *Hardware* verlegt, was die Prozessoren wesentlich entlastet und dem Programmierer beliebige Punkt-zu-Punkt-Verbindungen direkt zur Verfügung stellt.

Eine kompatible Erweiterung des iPSC2 ist der iPSC860: Er verwendet neue Prozessorboards mit einem i80860 als CPU und 8 bis 64 MB Hauptspeicher. Die Spitzengeschwindigkeit beträgt damit 80 MFlops pro Prozessor. Die Kommunikationshardware blieb gleich; es können 8 bis 128 Prozessoren und ebensoviele Plattensysteme zusammengeschlossen werden, und auch gemischte Konfigurationen mit iPSC2 und iPSC860 Prozessorkarten in einem Gerät sind möglich.

---

[37] Die Busse dienen ausschließlich der *Nachrichtenübertragung*, nicht dem Speicherzugriff mittels direkter Adressierung. Sie sind ident, jedes Board ist mit beiden Bussen verbunden und bekommt dynamisch den zugeteilt, der gerade frei ist. Die Verdopplung dient also vor allem der Geschwindigkeitssteigerung, nicht der Betriebssicherheit.

Auch die iPSC's sind keine eigenständigen Universalrechner für Mehrbenutzerbetrieb; sie werden von einem Frontend aus (einer Sun Workstation) mit einzelnen Anwenderprogrammen geladen. Deshalb läuft auf den einzelnen Knoten auch kein Betriebssystem im üblichen Sinn, sondern nur ein kleiner Kommunikationskernel[38].

Mittelfristig arbeitet intel mit Unterstützung der DARPA am Nachfolgeprojekt *"Touchstone"*: Dieses soll 16 bis 2048 i860 Prozessoren mit je 64 MB lokalem Speicher sowie verteilten Plattenspeichern und Grafik (die bei den iPSC's noch nicht verfügbar ist) umfassen. Es verwendet keinen Hypercube mehr, sondern ein normales Rechteckgitter mit einer Geschwindigkeit von mindestens 100 MB/s pro Kanal, das aus Softwaresicht beliebige direkte Punkt-zu-Punkt-Verbindungen bietet. Als Software ist eine Weiterentwicklung von CMU's Mach geplant.

Die nCubes

Auch diese kommerziellen Produkte stammen von den Caltech-Entwicklungen ab.

Der nCube 1 besteht aus maximal 1024 Knoten, die einen Hypercube bilden (wieder mit zusätzlichen I/O-Leitungen). Jeder Knoten besteht nur aus 9 Chips: Einem speziellen 16 bit Prozessor, der auch die gesamte Kommunikationslogik (14 Kanäle) enthält, sowie 8 Speicherchips für 1/2 MB RAM. Das gesamte System passt unter einen Schreibtisch.

Im Herbst 1989 wurde das Nachfolgemodell nCube 2 vorgestellt. Dieses System besteht aus 64–8192 Prozessoren, die ebenfalls wieder in Hypercube-Topologie miteinander verbunden sind. Wie beim Vorgängermodell ist die gesamte Kommunikationslogik (DMA-gestützt und unabhängig vom Prozessor) gemeinsam mit dem Prozessor[39], dem Gleitkomma-Koprozessor und der Speicherverwaltung auf einem einzigen Spezialchip untergebracht (rund 1 Million Transistoren!). Wie beim iPSC2 beträgt die Kommunikationsgeschwindigkeit 20 Mbit/s; und auch hier wird das Routing und Multiplexing hardwaremäßig erledigt. Im Unterschied zum iPSC, bei dem für einen Knoten zwei große Platinen benötigt werden, hat ein Knoten (mit 1–64 MB RAM) hier auf einer Scheckkarte Platz. Weiters stehen parallele I/O-Systeme, Plattenspeicher und Grafikinterfaces zur Verfügung.

Die nächste Generation soll bis zu 2^{16} Knoten erlauben; sie wird in gleicher Art und Weise mit einem speziellen Prozessor- und Kommunikationschip konzipiert

[38] Es ist allerdings die Portierung von Chorus, einem kommerziell angebotenen Unix-kompatiblen Betriebssystem für Systeme mit verteiltem Speicher, geplant.

[39] Es handelt sich um einen 7.5 Mips schnellen 32 bit Prozessor mit einer 2 MFlop schnellen 64 bit Gleitkommaeinheit.

sein und wieder eine Hypercube-Topologie verwenden.

Die Software und Betriebsweise ist ähnlich den iPSC's; ein Schwerpunkt von nCube sind Implementierungen von Datenbanksystemen wie Oracle.

Die FPS T Serie

Die T Serie der amerikanischen Firma Floating Point Systems kam 1985 als *"Scaleable Supercomputer"* heraus, wurde aber auf Grund von Softwareproblemen gleich darauf wieder eingestellt.

Ein Rechner konnte maximal 2^{14} Knoten umfassen (das größte installierte System umfaßte 64 Knoten). Jeder Knoten enthielt einen Transputer, dessen Links durch zusätzliche Logik auf 16 Kanäle aufgeteilt wurden, eine Weitek Vektor Pipeline, und 8 MB RAM. Zusätzlich war für je 8 Knoten ein Serviceprozessor mit einem Platteninterface vorgesehen, der u. a. für die Implementierung von Wiederaufsetzpunkten durch regelmäßiges Dumpen des RAM zuständig war.

2.3.10 Spezielle Parallel-Mikroprozessoren

Die Inmos Transputer-Familie

Die 1984 angekündigte Transputerfamilie des englischen Halbleiterherstellers Inmos umfaßt derzeit eine Reihe kompatibler Bausteine, die jeweils folgende Funktionen auf einem einzigen Chip bieten:

- 16 oder 32 bit Mikroprozessor (rund 15 Mips schnell).
- 64 bit IEEE-Standard Gleitkomma-Koprozessor (rund 2 MFlops schnell).
- 2 oder 4 KB RAM.
- Interface zu externem Speicher mit Kontroller für DRAM's (4 GB Adressraum, 50MB/s schnell).
- 4 bidirektionale, serielle, DMA-unterstützte, vom Prozessor unabhängig arbeitende *"Links"* für Zweidraht-Punkt-zu-Punkt-Verbindungen zu anderen Transputern (Geschwindigkeit 2 MB/s pro Link und Richtung)[40].

[40] Im Unterschied zu den Hypercubes müssen Routing, Multiplexing und Buffering softwaremäßig erledigt werden. Dadurch ergibt sich eine Verzögerung von einigen μs pro Zwischenknoten, im Vergleich zu weniger als 1 μs bei den iPSC's und nCubes. Andererseits liegt auch der Overhead in den Endknoten nur in diesem Bereich, während der Aufruf der Kommunikationsfunktionen bei den Hypercubes jeweils 100-200 μs dauert.

- Hardware-Unterstützung und eigene Maschinenbefehle für parallele Prozesse und Kommunikation (Prozeßwechsel in weniger als 650 ns[41]).
- Timer und automatisches, hardwaremäßiges Timeslicing von Prozessen.

Um die Netzwerktopologie von Transputersystemen flexibler zu gestalten, entwickelte Inmos einen *Link Switch*: Dieser Baustein erlaubt es, unter Softwarekontrolle 32 Links auf jede beliebige Art und Weise zu verbinden. Reicht das nicht aus, können mehrere solcher Bausteine in beliebiger Art und Tiefe kaskadiert werden.

Ursprünglich waren die Transputer vor allem für Steuerungen und andere dedizierte Systeme gedacht. Ab 1987 kamen jedoch sehr viele auf dem Transputer basierende universelle Multiprozessor-Systeme auf den Markt (Meiko, Parsytec u. v. a. m.). Im Prinzip lassen sich mit den heute angebotenen Systemen Computer beliebiger Größe aufbauen, in der Praxis umfassen die größten derzeit installierten Systeme rund 1000 Prozessoren[42].

Die rasche Verbreitung (der Transputer gehört zu den meistverkauften Mikroprozessoren, und es gibt mehr Parallelrechner auf Transputerbasis als es Parallelrechner aller anderen Typen zusammen gibt) liegt weniger an der Neuheit oder Überlegenheit des Konzeptes (beides hält sich in Grenzen), sondern daran, daß der Transputer von vorneherein mit dem Ziel *“Parallelität für das Volk”* entwickelt wurde:

- Mit dem Transputer stand erstmalig ein Baustein zur Verfügung, mit dem jeder mit minimalem Aufwand und minimalen Kenntnissen einen Parallelrechner höchster Leistung bauen konnte: Man braucht nur Transputerchips (und ev. ein paar Speicherchips) zusammenzulöten; es ist kaum zusätzliche Logik nötig, und vor allem entfällt das komplizierte Design einer Kommunikationshardware.
- Der Markt war und ist heiß umkämpft: Derzeit produzieren rund 200 Hersteller Hardware auf Transputerbasis, und zumindest 50 Firmen entwickeln Software für derartige Systeme. Ähnlich wie im PC-Bereich kristallisierte sich ein de-facto-Standard für Hardware- und Software-Schnittstellen heraus, die Hardwarekomponenten sind also untereinander misch- und austauschbar, und die Software ist portabel.

  Das hatte zur Folge, daß sich das Preisniveau für Transputersysteme bei einem Viertel bis einem Zehntel vergleichbar leistungsfähiger anderer Systeme einpendelte.

[41] Zum Vergleich: Ein Prozeßwechsel unter Unix benötigt fast 1 ms, also das Tausendfache!

[42] Spezialrechner für bestimmte Aufgaben aus der Signalverarbeitung umfassen bis zu 4000 Transputer; sie bestehen nur aus Transputerchips ohne irgendeine externe Logik.

- Man setzte von Anfang an stark auf die PC-Welt, es gibt sehr viele Transputereinschubkarten (und sogar Bastler-Kits) für PC's, Mac's usw., die kaum teurer als ein PC selbst sind und einen Einstieg in die Parallelrechnerei erlauben. Damit konnte einerseits der Markt der Kleinfirmen und sogar der Privatanwender und Freaks angesprochen, andererseits aber vor allem die Verbreitung in Lehre und Forschung forciert werden.

  Ziel ist einerseits das *"Personal Supercomputing"*, d. h. die Verfügbarkeit von Supercomputerleistung im PC-Stil[43], andererseits die Erhöhung der Grafikleistung von PC's und ähnlichen Systemen[44].

- Die Software entwickelte sich ähnlich: Sie ist zwar primitiv und fehlerhaft und bietet nur sehr begrenzte Möglichkeiten, aber sie stand von Anfang an zur Verfügung, nutzte die Fähigkeiten des Transputers optimal und erlaubte es, sehr effiziente Programme zu entwickeln. Außerdem war sie ebenfalls um Zehnerpotenzen billiger als alles Vergleichbare, lief auf jedem PC, und hatte nie mit Ausfuhr-, Geheimhaltungs- oder Lizenzproblemen zu kämpfen.

  Heute können Transputer sowohl als reine Anwendungs-Rechner in Kombination mit irgendeinem Frontend betrieben werden, als auch als selbstständige (sogar mehrbenutzerfähige) Universalrechner mit einem Unix-ähnlichen Betriebssystem.

- Es gibt sehr viele Interfaces (Grafik, Disk, Ethernet, Frame Grabber, IEEE-488 Bus, A/D- und D/A-Wandler, parallele und serielle Schnittstellen usw.), was dem Transputer weitere Einsatzbereiche, vor allem in Industrie, Forschung und Lehre, öffnete.

- Weiters verschaffte es dem Transputer viel Auftrieb, daß er in Großbritannien praktisch zur nationalen Religion erhoben wurde und massive Unterstützung durch Staat und Universitäten fand.

Zumindest zwei Firmen bieten derzeit auch Systeme auf Transputerbasis an, bei denen jeder Knoten zusätzlich einen intel i860 Prozessor enthält: Der i860 trägt die Rechenlast, und der Transputer dient als Kontroll- und Kommunikationsprozessor. Auch Kombinationen von Transputern mit speziellen Gleitkomma-Vektor-Chipsets, Signalverarbeitungs-Prozessoren oder Grafikprozessoren werden angeboten.

---

[43] Zehn Transputer mit je 4 MB RAM sind heute auf einer einzigen PC-Karte zu haben, und für 1 Million Schilling bekommt man schon mehr als 100 MFlops.

[44] Der Transputer ist zwar speziellen Grafiksystemen unterlegen, wenn es um die reine Grafikleistung geht, aber er bietet von allen universellen Mikroprozessoren mit Abstand die beste und schnellste Unterstützung für grafische Operationen und eine ausgewogene Kombination von universeller Rechenleistung und Grafikleistung.

Die Cogent XTM

Die XTM-Familie von Workstations und Rechenleistungs-Servern, die im Jahr 1987 von der amerikanischen Firma Cogent vorgestellt wurde, basiert auf Transputern, realisiert aber ein für Transputer sehr ungewöhnliches Kommunikationskonzept:

- Die Transputer sind sowohl über Links als auch mit einem gemeinsamen Bus verbunden.
- Es gibt keine fix verdrahteten Links; alle Links sind so über Link Switches geführt, daß zwischen je zwei Transputern eine direkte Link-Verbindung hergestellt werden kann.
- Mit diesen Link Switches wird allerdings nicht die für ein Anwenderprogramm jeweils günstige Topologie fix programmiert (wie das sonst üblich ist), sondern es werden einzelne Verbindungen bei Bedarf durchgeschaltet und nach erfolgter Kommunikation wieder abgebaut.
- Kurze Nachrichten werden direkt über den Bus übertragen, ohne daß die Links involviert sind.

  Für lange Nachrichten (oder für kurze Nachrichten, die für einen Knoten außerhalb des lokalen Systems, d. h. nicht an diesem Bus, bestimmt sind) wird zuerst eine Anfrage über den Bus an den Kontrollprozessor gesendet. Dieser errichtet eine direkte Link-Verbindung über die Link Switches und bestätigt danach die Anfrage über den Bus. Daraufhin wird die Nachricht über dieses Link direkt zum Zielknoten übertragen. Nach erfolgter Kommunikation baut der Kontrollprozessor die Linkverbindung wieder ab.
- Um das ganze System homogen zu erhalten, basieren auch die Verbindungen zwischen den einzelnen Workstations und Servern auf Links (bei großen Entfernungen über Glasfasern), und nicht auf einem Standardnetzwerk wie z. B. Ethernet.

Auch das Softwarekonzept der XTM ist unüblich: Es handelt sich um ein eigens entwickeltes Betriebssystem, das sich für den Benutzer ähnlich wie Unix verhält ("it walks like Unix, it talks like Unix"), bei dem aber *Linda* als zentrales Konzept für Parallelität und Kommunikation sowohl auf Anwenderebene als auch systemintern, aber auch für die Organisation des Filesystems usw. dient.

Die neue Transputer-Generation

Inzwischen ist die zweite Generation von Transputern (T9000) verfügbar. Sie ist softwaremäßig mit ihren Vorgängern kompatibel: Alte Binärcodes laufen unverändert; die Neuerungen und Erweiterungen sind für bestehende Programme transparent. Auch hardwaremäßig wurden Bausteine entwickelt, die eine Kombination von Transputern der ersten und zweiten Generation in einem System erlauben.

Im wesentlichen wurden drei Ziele verfolgt: Steigerung der Leistung, Behebung bestehender Mängel und Unzulänglichkeiten, und Erweiterung der Fähigkeiten. Das resultiert in folgenden Merkmalen:

- Spitzen-Rechenleistung bei 50 MHz 200 MIPS oder 25 MFlops, Dauerleistung 70 MIPS oder 15 MFlops[45].
- Superskalarer Prozessor mit automatischem Umordnen der Befehle (auch auf altem Binärcode und ohne spezielle Compilerunterstützung halbwegs effizient).
- Unterstützung neuer Parallelitätskonstrukte durch zusätzliche Maschinenbefehle:
  - ∗ Semaphore.
  - ∗ n-zu-1 Kanäle für die Implementierung von Serverprozessen mit beliebig vielen Klienten.
- 16 KB Speicher on-Chip, wahlweise als echtes RAM, als Cache, oder halbe/halbe. Logik für Cache-Konsistenz in Systemen mit gemeinsamem Speicher.
- 4 DMA-gestützte Links mit je 10 MB/s pro Link und Richtung, Paritätsprüfung aller Daten- und Statustransfers.
- Eigener Kommunikations-Prozessor:
  - ∗ Aufteilung in Pakete und Erzeugung von Paket-Headern für ausgehende Nachrichten, Analyse der Header und Zustellung der Nachrichten für eingehende Pakete.
  - ∗ Multiplexing mehrerer logischer Kanäle auf einem Link.
  - ∗ Verwaltung der Acknowledgements, Buffering von Paketen für noch nicht empfangswillige Prozesse.

[45] Die MFlops beziehen sich auf skalare 64 bit Gleitkommaoperationen, es wurde keine eigene Vektor-Einheit implementiert.

   * Erkennen physikalischer Übertragungsfehler und Timeouts, Unterstützung softwaremäßiger Fehlerbehandlung.

- Speicherverwaltungseinheit zum Speicherschutz und zum Mapping von logischen auf physikalische Adressen (aber noch keine Unterstützung für demand-paged virtual memory). Unterscheidung zwischen System- und Anwender-Prozessen, transparentes Trapping privilegierter Instruktionen.
- Verbesserter DRAM-Kontroller: Unterstützung von page mode DRAM usw..
- Eigene, unabhängige Links zur Systemkonfiguration und -kontrolle und zum Debugging, kaskadierbar ohne zusätzliche Hardware.

Auf Routinglogik auf dem Prozessor wurde bewußt verzichtet, dazu dient ein eigener Baustein (C104):

- 32 Links, intern durch einen $32 * 32$ Crossbar Switch verbunden (alle Links können gleichzeitig in beide Richtungen aktiv sein).
- Verbindbar in beliebiger Topologie, kaskadierbar in beliebiger Tiefe.
- Wormhole Routing, Verzögerung kleiner als 1 μs.
- Routing für jedes Paket einzeln, Bestimmung des Ausgangslinks durch Suche des Paket-Headers in Intervall-Tabellen[46]. Auf Wunsch Löschung des Headers für hierarchische Netze mit mehreren Headern pro Paket, dadurch Routing transparent für den Empfänger.
- Buffering von Paketen, die für ein gerade anderweitig belegtes Ausgangslink bestimmt sind.
- Möglichkeit zum Zusammenfassen mehrerer Links zu Gruppen: Pakete für eines dieser Links werden auf einem beliebigen freien Link der Gruppe gesendet.
- Möglichkeit zur Erzeugung zufälliger Paket-Header für *“Random Routing”* zur Vermeidung von Hot Spots.
- Separate Links zur Systemkonfiguration und -kontrolle.

[46] Dies unterscheidet den C104 wesentlich vom alten Link Switch C004, bei dem permanente Verbindungen zwischen Paaren von Links von außen programmiert wurden, unabhängig von den auf den Links übertragenen Daten.

Durch die Kombination von Transputern und Routern ist es möglich, daß jeder beliebige Prozeß auf irgendeinem Transputer im Netz mit einem einzigen Maschinenbefehl ohne zusätzlichen Softwareoverhead direkt mit jedem beliebigen anderen Prozeß im Netz kommunizieren kann.

Basierend auf der neuen Transputergeneration gibt es einerseits wieder eine Familie von Modulen mit standardisierten, herstellerunabhängigen Hard- und Softwareschnittstellen, andererseits werden bereits komplette Großsysteme mit einigen tausend Transputern angeboten.

Der intel iWarp

Der iWarp ist ein Forschungsprojekt von intel mit DARPA-Unterstützung in Zusammenarbeit mit der CMU (Carnegie-Mellon University). Das Ziel ist ein Chip zum Aufbau von Systemen zur parallelen Bild- und Signalverarbeitung mit 10 bis 1000 Prozessoren.

Dieser Chip enthält einen Integer- und einen Gleitkommaprozessor, einen Cache und ein Speicherinterface, sowie 4 vom Prozessor unabhängige, parallele (nicht serielle!) Punkt-zu-Punkt Kommunikationskanäle mit Routing- und Multiplexing-Unterstützung (wahlweise Leitungs- oder Paketvermittlung), die eine Datenrate von 40 MB/s je Kanal und Richtung bieten.

Das Besondere an den Kommunikationskanälen ist, daß sie sowohl alle synchronisiert zum Aufbau ein- oder zweidimensionaler systolischer Systeme verwendet werden können, als auch unabhängig voneinander den Austausch einzelner Nachrichten wie z. B. beim Transputer ermöglichen. Im systolischen Betrieb werden die Daten ohne zusätzlichen Software- und Synchronisationsoverhead direkt in die oder von den CPU-Registern übertragen (der iWarp enthält einen großen Multiport-Registersatz).

Alle Funktionseinheiten des iWarp können in jedem einzelnen Maschinenbefehl nach dem VLIW-Prinzip gleichzeitig und unabhängig voneinander angesprochen werden.

2.3.11 Prolog-Maschinen

Bei parallelen Prolog-Maschinen sind zwei Grundtypen zu unterscheiden:

Inhomogene Prolog-Maschinen:

Diese Maschinen bestehen aus einigen *wenigen, unterschiedlichen Prozessoren*, von denen sich jeder um eine einzelne der für die Abarbeitung von Prolog-Programmen notwendigen Grundoperationen (Unifikation, Speicherverwaltung, ...) kümmert.

Es gibt vor allem in Japan einige funktionierende Systeme dieses Typs; da die Prozessorzahl aber naturgemäß fix und relativ klein (maximal acht) ist, ist mit dieser Methode keine wesentliche Leistungssteigerung gegenüber sequentieller Prolog-Hardware zu erzielen.

Homogene Prolog-Maschinen:

Diese Maschinen bestehen aus *vielen gleichartigen Prozessoren*, die imstande sind, verschiedene Pfade eines Prolog-Programmes echt parallel und unabhängig voneinander in Angriff zu nehmen.

Auf diesem Gebiet ist trotz großer Anstrengungen (Japans *"Fifth Generation"* Projekt) bis jetzt noch kein entscheidender Durchbruch gelungen, es ist noch nicht einmal ein einheitlicher, erfolgversprechender Ansatz zu erkennen. Das liegt zum einen daran, daß auch das Gebiet sequentieller Prolog-Prozessoren noch nicht ganz abgeklärt und ausgereift ist, und zum anderen vor allem daran, daß viele der abstrakten parallelen Maschinenmodelle einen *großen globalen Speicher* (und manchmal auch eine globale Abarbeitungskontrolle) voraussetzen.

2.3.12 Maschinen für funktionale Sprachen

Mit der Untersuchung rein funktionaler Sprachen (siehe Kapitel "Parallele Software") kamen auch Maschinenmodelle auf, die sich für die Abarbeitung solcher Sprachen (aber auch z. B. für Prolog) sehr gut eignen und vor allem in einfacher und natürlicher Weise eine Parallelverarbeitung gestatten. Sie unterscheiden sich allerdings wesentlich vom von-Neumann-Modell und sind daher zur Ausführung konventioneller Programmiersprachen (z. B. Fortran) ungeeignet.

Es gibt viele verschiedene Ansätze, aber im wesentlichen zwei Grundtypen: *Datenfluß-Maschinen* und *Reduktions-Maschinen.* Von beiden existieren zahlreiche universitäre Prototypen, kommerzielle Produkten sind geplant, aber noch nicht realisiert.

Datenfluß-Maschinen

Die "Maschinensprache" von Datenflußmaschinen sind *Datenflußgraphen.* Die Knoten sind die *Operationen* (z. B. arithmetische Operationen, Selektion von Feldelementen, Vergleiche, `if`-Selektion usw.), die Kanten die *Datenabhängigkeiten* (d. h. von einer Operation führt je eine gerichtete Kante zu all jenen Operationen, die das Ergebnis dieser Operation unmittelbar weiterverwenden). Für die Eingabe und die Ausgabe gibt es ebenfalls Kanten.

Der “Speicherzustand” von Datenflußmaschinen ist eine Menge von *Token*. Jedes Token enthält ein Datum und *Tags*. Die Tags bestimmen im wesentlichen die *Operation*, für die das Datum bestimmt ist (entspricht dem Programmzeiger konventioneller Maschinen) und den *Kontext*, zu dem es gehört, um die Token, die für die gleiche Operation bestimmt sind, aber zu unterschiedlichen Rekursionsinstanzen oder Schleifeninstanzen gehören, auseinanderzuhalten (entspricht dem Stapelzeiger konventioneller Maschinen)[47]. Es gibt keinen direkt adressierbaren Speicher, keine Register und vor allem keinen Programmzeiger.

Die Abarbeitung ist *feinkörnig-parallel* (auf der Ebene einzelner Instruktionen oder Datenelemente), sie ist nur durch die Token gesteuert: Die Token wandern entlang der Kanten des Graphen; sind alle für eine Operation benötigten Token (aus dem gleichen Kontext) eingelangt, wird diese Operation (unabhängig von allen anderen) ausgeführt. Sie konsumiert dabei die Eingangstoken und liefert auf jeder ausgehenden Kante ein neues Token mit dem Ergebnis und den entsprechenden Tags (mit dem gleichen Kontext) für die Nachfolgeoperationen ab (siehe Abbildung 2.13). Jede Instanz einer Instruktion ist sozusagen ein eigener Prozess, die Synchronisation erfolgt ausschließlich durch die Datenabhängigkeiten.

Eine Datenflußmaschine besteht grundsätzlich aus beliebig vielen identischen Prozessoren, die durch ein Netzwerk verbunden sind, das Token anhand der Tags von jedem beliebigen Prozessor an jeden anderen liefern kann.

Ein Prozessor besteht aus folgenden Einheiten (siehe Abbildung 2.14):

Der Token Queue:

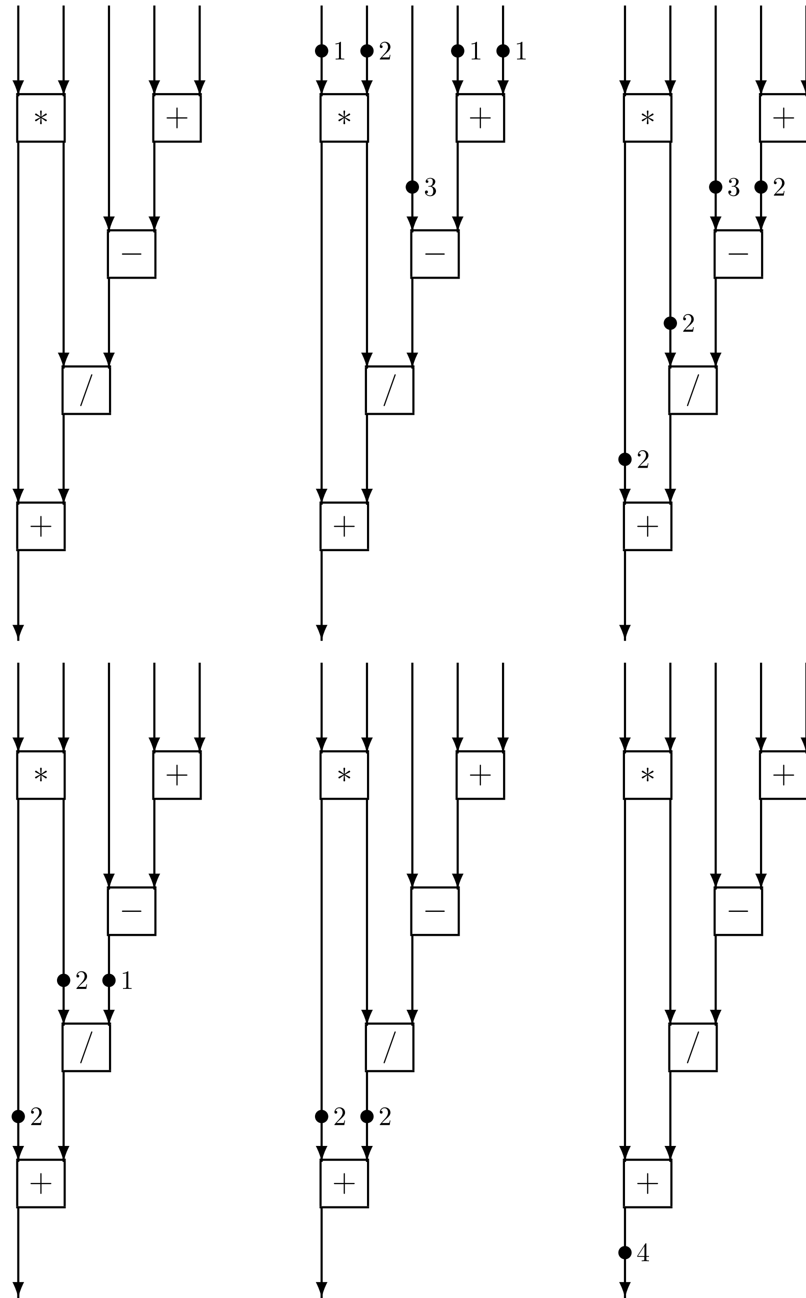
Sie puffert die aus dem Netz ankommenden Token, da die Wait-Match-Einheit Token mit konstanter Geschwindigkeit konsumiert, Token aber in sehr stark variierender Menge und Rate eintreffen können.

Der Wait-Match-Einheit mit dem Waiting Token Speicher:

Sie prüft, ob schon alle anderen zu der Zieloperation eines ankommenden Token gehörenden Token eingetroffen sind. Ist dies der Fall, werden alle diese Token an die Befehlseinheit weitergereicht, ansonsten wird das Token zwischengespeichert, bis die anderen Operanden ankommen.

Der Befehlseinheit mit dem Befehlsspeicher:

[47] Historisch begann die Entwicklung mit sogenannten “statischen” Datenflußmaschinen, die im Unterschied zu den hier beschriebenen “Tagged-Token” Datenflußmaschinen noch keine Tags kannten und daher nur eine Instanz von jedem Codestück verwalten konnten. Sie konnten daher keine Schleifen, keine Rekursion und keine Funktionen höherer Ordnung bearbeiten.



Abarbeitung von $f(1,2,3,1,1)$, $f(u,v,x,y,z) := (u*v)+(u*v)/(x-(y+z))$.

Abbildung 2.13: *Abarbeitung von Datenflußgraphen*

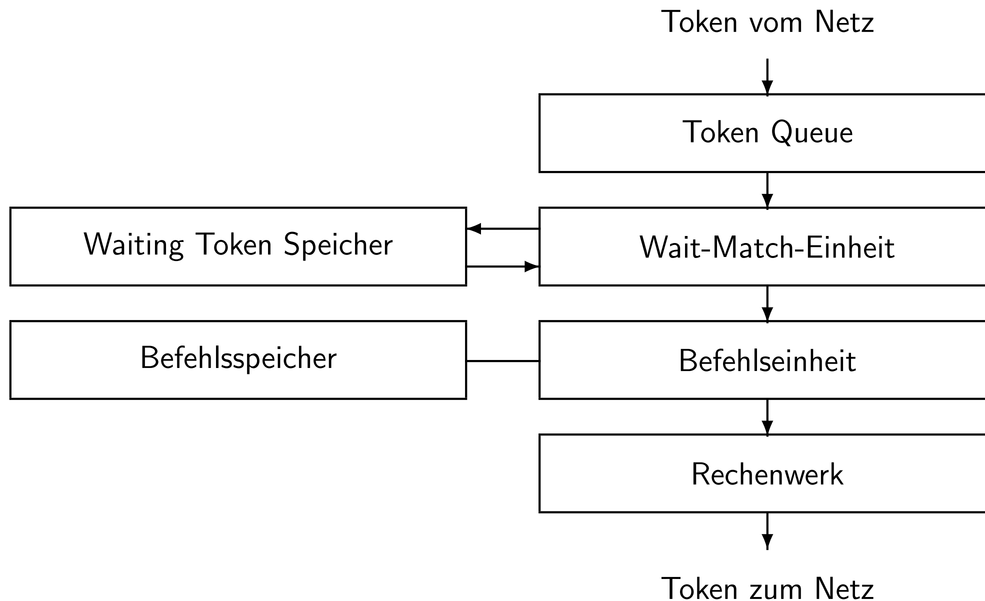


Abbildung 2.14: *Aufbau eines Datenfluß-Prozessors*

Sie stellt den zu einem Satz zusammenpassender Token gehörenden Befehl (und eventuell notwendige Konstanten etc.) anhand der in den Token enthaltenen Zieladresse zur Verfügung, enthält also den Datenflußgraph.

Dem Rechenwerk:

Es führt auf den Daten der Eingangstoken die angegebene Operation aus, ermittelt die neuen Tags, generiert die Ausgangstoken und leitet sie an das Netz weiter.

Die Wait-Match-Einheit ist der heikle Punkt, weil man nicht weiß, ob ein Token gespeichert oder weitergeleitet wird; alle darauffolgenden Einheiten lassen sich ähnlich wie bei konventionellen Prozessoren als starr getaktete Pipeline ausführen.

Diese Arbeitsweise hat zusätzlich zu den grundsätzlichen Vorteilen funktionaler Programmiersprachen einige Vorteile:

- Sie holt das *Maximum an Parallelität* aus jedem Algorithmus.
- Es gibt *keinen Overhead* für die parallele Abarbeitung, das Abarbeitungsmodell im sequentiellen und im parallelen Fall ist genau das gleiche.

- Solange genügend Parallelität vorhanden ist, wächst die Geschwindigkeit praktisch *linear* mit der Prozessorzahl.
- Solange genügend Token vorhanden sind, ist es egal, wie lange ein einzelnes Token auf dem Weg von seiner Erzeugung zu seiner Zieloperation braucht. Die Maschine toleriert daher *große Speicher- und Kommunikations-Verzögerungen.*
- Sie ist etwas einfacher zu realisieren als die Reduktionsmaschine.

Es gibt auch einige Probleme:

- Im Prinzip ist für das Aufsammeln aller zu einer Operation gehörenden Eingangstoken ein *Assoziativspeicher* nötig, denn im wesentlichen müssen die Tags (Zieloperation und Kontext) gematcht werden[48].
- Alle Daten müssen *fixe Größe* haben, denn sie müssen in einem Token Platz finden[49].
- Die *Beschränkung explosiver Parallelität*, die *Lastverteilung* und vor allem die *Speicherverwaltung* ist noch nicht zufriedenstellend gelöst, der Speicherbedarf in den einzelnen Speichern pulsiert sehr stark.
- Das Kommunikationsnetz zum Austausch von Token zwischen den Prozessoren muß trotz des *feinkörnigen Datentransfers* (einzelne Token) *sehr hohen Durchsatz* haben[50].

Beispiele von Datenflußmaschinen sind die TTDA (1986) sowie die ETS ("Monsoon") (1988) am MIT, die Manchester Dataflow Machine (1982, eingestellt) und die ETL-Sigma 1 in Japan. Die Sigma-1 arbeitet seit 1986, sie erreichte mit 128 Prozessoren Geschwindigkeiten über 600 MFlops.

Reduktions-Maschinen

Die "Maschinensprache" von Reduktionsmaschinen sind *Rewrite Rules*, und zwar für Graphen (in der Gestalt, wie man z. B. arithmetische Ausdrücke als

[48] Am MIT ist es allerdings gelungen, ein Maschinenmodell zu entwerfen, bei dem bereits im Compiler absolute Adressen für zu matchende Token vergeben werden und nur ein ganz gewöhnlicher Speicher vorausgesetzt wird.

[49] Die meisten heutigen Datenflußmaschinen sind um *Struktur- oder Arrayspeicher* erweitert (die Token enthalten dann nur mehr die Pointer), aber das erfordert großen zusätzlichen Aufwand.

[50] Im Prinzip ist das von einer Datenflußmaschine verwendete Speichermodell ja ebenfalls ein globales.

Binärbaum mit den Operationen als inneren Knoten und den Operanden als Söhnen darstellen kann).

Der "Speicherzustand" von Reduktionsmaschinen ist ein *Graph*. Am Anfang repräsentiert dieser Graph den eingegebenen Ausdruck; er wird so lange *durch Anwenden der Rewrite Rules umgeformt*, bis keine Rule mehr anwendbar ist, und repräsentiert dann das Ergebnis. Selbstverständlich kann der Graph an vielen Stellen gleichzeitig und unabhängig voneinander bearbeitet werden (siehe Abbildung 2.15).

Dabei gibt es im wesentlichen zwei Typen von Rewrite Rules:

- Teilgraphen, deren Wurzel eine *elementare* Operation (z. B. Integer-Addition) ist, werden nur umgeformt, wenn die Operanden (Subgraphen) bereits fertig berechnet sind. Der Teilgraph wird dann durch die Repräsentation des Ergebnisses ersetzt, das durch Anwendung der Operation auf die Operanden berechnet wird.
- Teilgraphen, deren Wurzel eine *benutzerdefinierte* Funktion ist, werden durch den Graphen, der die Definition dieser Funktion repräsentiert, ersetzt, wobei statt der formalen Parameter an allen Stellen die Graphdarstellung der aktuellen Argumente substituiert wird. Dieser Graph wird dann weiter behandelt.

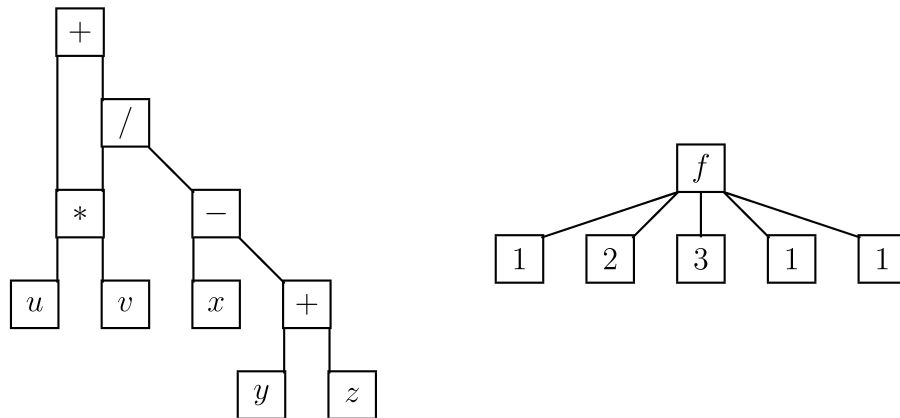
In der Praxis bestehen diese Maschinen aus einem großen zentralen *Graph-Speicher*, auf den *viele identische Prozessoren* Zugriff haben. Jeder Prozessor entnimmt in einer ständigen Schleife einen reduzierbaren Teilgraph, wendet eine Rewrite-Rule an, und stellt das Ergebnis wieder zurück in den zentralen Speicher.

Der größte Vorteil liegt darin, daß es kaum Load-Balancing-Probleme gibt: Wenn ein Prozessor nichts zu tun hat, sucht er sich einfach den nächstbesten reduzierbaren Teilgraphen.

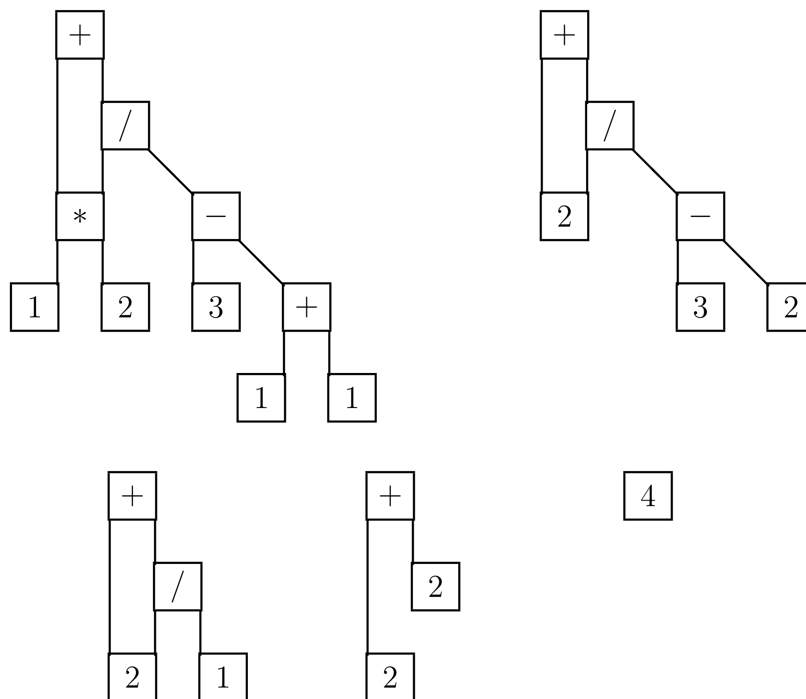
Das Konzept hat allerdings einige Nachteile:

- Ein *großer, globaler Speicher* ist schwierig zu bauen, und die *Speicherverwaltung* für die Graphen (parallele Garbage Collection usw.) ist auch nicht einfach.
- Man kann zwar alle *Strategien* für die Abarbeitung funktionaler Sprachen (eager/lazy, strict/nonstrict) leicht nachbilden[51], aber die in der Praxis optimale Strategie wurde noch nicht gefunden: Einerseits gilt es, unendliche Äste zu vermeiden und den Speicherverbrauch zu begrenzen,

[51] Datenflußmaschinen sind in dieser Hinsicht nicht so flexibel: Sie arbeiten prinzipiell stets eager.



Darstellung von $f(u, v, x, y, z) := (u*v)+(u*v)/(x-(y+z))$ und $f(1, 2, 3, 1, 1)$.



Abarbeitung.

ABBILDUNG 2.15: *Graph-Reduktion*

andererseits kann eine zu restriktive Strategie die vorhandene Parallelität zu sehr einschränken.

- Die Implementierung der Strategie erhöht die Anzahl der notwendigen *Speicherzugriffe* stark: Normalerweise muß bei jedem Zugriff auf einen Knoten auch auf dessen Vater oder Söhne zugegriffen werden, um irgendwelche Flags oder Zähler zu ändern.

- Das Problem der unnötigen *Mehrfachauswertungen* identischer Funktionen mit identischen Argumenten ist ebenfalls noch nicht zufriedenstellend gelöst.

Das Zentrum der Reduktionsmaschinen ist Großbritannien, die Alice vom Imperial College London ist die bekannteste und älteste Vertreterin: Sie ist als hierarchisches System konzipiert, bei dem rund 20 Reduktionseinheiten mit einem gemeinsamen "Packet Pool" (Graphspeicher) verbunden sind; mehrere Pools untereinander tauschen Informationen durch ein intelligentes Netz aus[52].

Neuere Projekte sind Flagship, ein Gemeinschaftsprojekt von mehreren Universitäten und Firmen in Großbritannien, und Grip, eine Entwicklung des University College London, die auf konventionellen Mikroprozessoren, aber intelligenten, aktiven Speichern besteht.

[52] Derzeit gibt es einen Prototypen, der aus einigen wenigen Reduktionseinheiten an nur einem Pool besteht, die einzelnen Reduktionseinheiten sind aus einigen fest programmierten Transputern aufgebaut.

Kapitel 3

Parallele Software

In diesem Kapitel werden wir nach einem Überblick über die grundsätzlichen Probleme und Methoden paralleler Programmierung die Anforderungen an Programmierumgebungen für parallele Systeme besprechen. Dann werden wir auf die wichtigsten parallelen Programmiersprachen und -konzepte eingehen.

3.1 Einleitung

Das eigentliche *Problem* bei Parallelrechnern und der Hauptgrund für ihre unerwartet langsame Verbreitung liegt heute nicht darin, sie zu bauen, sondern darin, sie zu *programmieren.*

Das erkennt man auch daran, daß Parallelrechner bisher ihre *größte Verbreitung und Akzeptanz* dort gefunden haben, wo ihre *Parallelität vor dem Benutzer weitgehend verborgen* bleibt:

- Vor allem bei numerischen Anwendungen wird das Erzeugen parallelen Codes oft von *parallelisierenden und vektorisierenden Compilern* übernommen, der Anwender programmiert weiterhin sequentiell in einer konventionellen Programmiersprache.

- Viele Multiprozessorsysteme (Encore, Sequent, etc.) werden heute vor allem als Hochleistungs-Multiuser-Server für Unix und *Datenbankanwendungen* eingesetzt, um den *Systemdurchsatz als Ganzes* zu stei-

gern. Für den einzelnen Benutzer verhalten sich solche Systeme allerdings weiterhin wie ein konventioneller Monoprozessor.

Dieses Problem wurde auch in einigen markanten Zitaten verdeutlicht:

“A giant step backwards in programming . . . ”

Das bezieht sich vor allem darauf, daß ein Großteil der heute verwendeten parallelen *Programmiermodelle* unmittelbar die *Hardwarestruktur* widerspiegeln und jede Abstraktion vermissen lassen.

“Parallel programming is programming in communication assembler.”

Hier wird bekrittelt, daß die Programmierwerkzeuge und -sprachen für fast jede heute zur Verfügung stehende Parallelarchitektur ein Unikum, *eine Insel für sich*, darstellen, und daß parallele Programme dadurch *kaum portabel* sind — ein Zustand, von dem man glaubte, daß er 1960 mit der Einführung höherer Programmiersprachen für immer überwunden sei.

“One thing more to go wrong.”

Diese Bemerkung will verdeutlichen, daß der Programmierer nun zusätzlich den *zeitlichen Ablauf der einzelnen parallelen Prozesse relativ zueinander* beachten muß.

Interessanterweise sind viele der bei der parallelen Programmierung auftretenden Probleme (z. B. Parallelität und Speicherverwaltung innerhalb eines Knotens, Kommunikation über Netzwerke, Fehlertoleranz, lastabhängiges Verteilen von Prozessen auf Netzwerken, . . .) *auf dem Niveau von Betriebssystemen und Netzwerkprotokollen bereits eingehend untersucht und größtenteils auch befriedigend gelöst.* Diese Möglichkeiten sind aber derzeit *für den Anwendungsprogrammierer nicht in vernünftiger Form nutzbar*, sie fanden noch nicht Einzug in Programmiersprachen oder Unterprogrammbibliotheken.

Zusammenfassend lassen sich folgende Hauptprobleme paralleler Software beobachten:

Brauchbare parallele Programmiersprachen fehlen.

Das liegt großteils daran, daß man sich derzeit noch nicht einmal über ein vernünftiges abstraktes paralleles Maschinen- und Programmiermodell im klaren ist; die parallelen Komponenten heutiger Programmiersprachen wurden meist *ad hoc* entsprechend der vorhandenen Hardware konzipiert.

Brauchbare parallele Programmierumgebungen fehlen.

Vor allem das *Debugging* paralleler Software erweist sich als ungleich schwieriger als im sequentiellen Fall[1].

Es gibt noch keine Standards.

Heutige parallele Programme sind praktisch *nicht portabel*; um sie auf einen anderen Parallelrechner zu übertragen, sind umfangreiche Änderungen des Source Codes, oft sogar des Algorithmus, notwendig.

Im Idealfall sollte parallele Software sogar zwischen grundverschiedenen Hardwaretypen (von Shared-Memory-Supercomputern über Hypercubes bis zu Workstations an einem lokalen Netz, und selbstverständlich auch zwischen Uniprozessoren) übertragbar sein, und zwar schlimmstenfalls mit einer Neukompilation. Das hätte viele wesentliche Vorteile:

- Der Anreiz, *parallele Software zu entwickeln*, wird größer, weil sich ein breiterer, stabilerer Markt auftut.
- Der Anreiz, *parallele Hardware einzusetzen*, wird größer, weil ein breiteres Softwareangebot zur Verfügung steht.
- Der Anreiz, *sich mit paralleler Programmierung zu beschäftigen*, wird größer, weil man sich auf jeder parallelen Maschine sofort zu Hause fühlt.
- Dadurch wird auch das Angebot an Programmierern für parallele Systeme größer, und es wird möglich, diese Programmierer an verschiedenen Systemen ohne lange Lernphase einzusetzen.

Erste Ansätze in diese Richtung sind im Programmiermodell LINDA der Universität Yale und in den *funktionalen und logischen Programmiersprachen* zu erkennen, ebenso im FORTRAN-8x (oder -9x ?) Standard.

Heute ist es so, daß sogar eine *Änderung der Knotenzahl oder der Verbindungstopologie* innerhalb eines Systems (z. B. durch Erweiterungen oder Hardwaredefekte) in vielen Fällen eine *Änderung des Programms* bedingt, obwohl diese Dinge automatisch zur Laufzeit behandelt werden könnten und sollten.

Fehlertoleranz fehlt.

Einerseits fällt darunter das soeben angesprochene Problem der *veränderten Hardwaretopologie durch Defekte* (die Fehlerwahrscheinlichkeit steigt im günstigsten Fall linear mit der Zahl der Bauelemente, und die ist bei Parallelrechnern praktisch unbegrenzt). Im Extremfall muß man annehmen, daß *die Maschinenkonfiguration zur Ausführungszeit zum Zeitpunkt der Programmierung unbekannt ist und sich sogar während der Ausführung ändern kann.*

[1] Zu diesem Punkt folgt noch ein eigener Absatz.

Auf der anderen Seite treten in größeren Systemen auch sehr viel häufiger sogenannte *Soft Errors*, das heißt kurzfristige, reversible, nicht reproduzierbare Störungen oder Datenverluste beispielsweise durch kosmische Höhenstrahlung, auf. Es ist durchaus denkbar, daß *die mittlere Zeit zwischen zwei Soft Errors bald unter die Laufzeit rechenintensiver Anwendungen sinkt*[2]; man muß daher annehmen, daß solche Fehler während eines Programmlaufes mit an Sicherheit grenzender Wahrscheinlichkeit auftreten.

Hier wäre es notwendig, daß Compiler und Betriebssystem automatisch (d. h. unsichtbar für den Programmierer) *Wiederaufsetzpunkte* innerhalb des Programms einbauen und im Fehlerfall die verlorengegangenen Berechnungen neu starten[3].

Eine Umerziehung ist notwendig.

Die Denkweise der Programmierenden und vor allem der Lehrenden bedarf einer Umstellung. Das *"Denken in* FORTRAN*"* ist so tief verwurzelt, daß *ein Großteil der in einem Problem natürlich vorhandenen Parallelität schon beim Übergang von der Problemspezifikation zu einer ersten konkreten Algorithmusidee verloren geht*, ohne dem Programmierer jemals zu Bewußtsein gekommen zu sein.

Natürlich muß dann Parallelität wieder künstlich dazukonstruiert werden, was einerseits als schwierig empfunden wird, und andererseits zu krampfhaften, unnatürlichen Programmen führt.

Der Ruf ist verdorben.

Viele ad hoc eingeführte, unüberlegte und oft sogar gefährliche Sprachmittel und Parallelisierungsstrategien haben dazu geführt, daß *die parallele Programmierung als besonders kompliziert, fehleranfällig und unsicher gilt*[4].

Viele Probleme sind schwer oder unlösbar.

Viele der im Rahmen paralleler Programmierung auftretenden Probleme sind entweder noch ungelöst oder bereits als schwer oder unlösbar erkannt.

Beispiele:

---

[2] Bei Speichern im Gigabyte-Bereich liegt die mittlere Zeit zwischen zwei reversiblen Fehlern bereits unter einer Stunde.

[3] FPS hat bei seiner T Series automatisches Checkpointing des RAMs auf die verteilten, lokalen Disks vorgesehen.

[4] In den USA sind einige — zum Teil auch kommerzielle — Parallelprojekte aus Softwaregründen spektakulär in die Hose gegangen, was die Investitionsfreude stark reduzierte.

- Das *Mapping-Problem* (Abbilden der logischen auf die physikalische Topologie) ist NP-vollständig (auch im einfachsten, statischen Fall).
- Die *Deadlockfreiheit* beliebiger Programme ist für die meisten praktisch verwendeten Kalküle nicht algorithmisch entscheidbar.
- *Beidseitiger Kommunikations-Nichtdeterminismus* (z. B. OCCAM's `ALT` erweitert um Output-Guards) ist verteilt und demokratisch (d. h. ohne zentrale Steuerprozesse) nicht implementierbar (von fair erst gar nicht zu sprechen).
- *Deadlockfreie und faire Kommunikation* ist im allgemeinsten Fall auf Netzen von Kanälen nicht implementierbar, wenn in jedem Knoten nur endlich viel Speicher zur Verfügung steht.

3.2 Klassifizierung

Wie bei der Hardware kann auch die folgende Aufstellung nur eine Hilfestellung bei der Einordnung paralleler Software sein, es gibt noch kein vollständiges, allgemein anerkanntes Schema.

Implizite oder explizite Parallelität?

Bei implizit parallelen Systemen braucht sich der Programmierer selbst nicht um die Parallelität kümmern, sie bleibt für ihn unsichtbar. Hier gibt es zwei wesentliche Ansätze:

Very high level Programmiersprachen:

PROLOG, funktionale Sprachen, Rewrite-Rule-Systeme, PURE LISP, etc., bieten auf Grund ihrer Semantik von vornherein reichlich Möglichkeiten zu paralleler Abarbeitung. Es ist daher im Prinzip einfach, entweder explizit parallelen Maschinencode zu erzeugen oder von vornherein ein implizit paralleles Hardwaremodell zu Grunde zu legen.

Vektorisierende und parallelisierende Compiler:

Diese versuchen, durch Datenflußanalyse, symbolische Exekution o. ä. aus Programmen in konventionellen, sequentiellen Programmiersprachen (FORTRAN, C, auch LISP, etc.) parallelen Code zu erzeugen.

Bei expliziter Parallelprogrammierung muß der Programmierer selbst die gewünschte Parallelität in seinen Programmen definieren. Dazu gibt es mehrere Wege:

Annotationen:

Hierbei handelt es sich syntaktisch gesehen um normale Kommentare (sie beeinflussen daher die Semantik nicht und werden einfach ignoriert, wenn das Programm für ein nichtparalleles System kompiliert wird), die dem Compiler Hinweise zur Unterstützung bei seinen Parallelisierungsversuchen (oder zur Unterdrückung unerwünschter Parallelisierung) geben. Dieser Weg ist typisch für Fortran-Dialekte auf konventionellen Supercomputern.

Bibliotheken paralleler Unterprogramme:

Derartige Bibliotheken stehen vor allem im numerischen Bereich für Vektor- und Matrizenoperationen zur Verfügung: Das Benutzerprogramm selbst bleibt rein sequentiell, nur die Bibliotheksfunktionen (die meist vom Hardwarehersteller selbst handkodiert und hochoptimiert mitgeliefert werden) arbeiten intern parallel.

Parallele Datentypen:

Ähnlich funktionieren *Datentypen, die parallel verarbeitet werden*: Für den Benutzer verhalten sie sich wie herkömmliche Datenstrukturen (z. B. Arrays oder Matrizen), er sieht nur eine veränderte Deklaration, aber weiterhin ein sequentielles Programm. Intern allerdings werden diese Datentypen automatisch verteilt und parallel bearbeitet. Diese Methode kommt vor allem auf SIMD-Maschinen zum Einsatz.

Parallele Schleifen:

So wie eine spezielle Deklaration den Compiler veranlaßt, ein Feld parallel zu behandeln, kann ein spezielles *Schleifen-Schlüsselwort* (bei sonst unveränderter Syntax und Semantik) dem Compiler anzeigen, daß die einzelnen Iterationen einer Schleife parallel auszuführen sind. Solche Konstrukte treten vor allem bei SIMD-Rechnern und konventionellen Supercomputern auf.

Parallelität mittels spezieller Systemfunktionen:

Der Aufruf spezieller Systemfunktionen (für das Kreieren oder Stoppen paralleler Prozesse, das Austauschen von Nachrichten, die Synchronisation usw.) sind der übliche Weg, Möglichkeiten zur parallelen Programmierung zu konventionellen Programmiersprachen (C, Fortran, Lisp usw.) hinzuzufügen, ohne die Sprache selbst zu verändern. Dieser Weg wird vor allem auf MIMD-Systemen beschritten.

Explizite syntaktische Sprachkonstrukte:

Solche Konstrukte werden meist in speziellen Parallel-Programmiersprachen (wie Occam, Concurrent Pascal, etc.) eingeführt, sie

bieten die umfangreichsten Möglichkeiten und auch die am leichtesten lesbare, natürlichste Notation.

Statische oder dynamische Parallelität?

Ein wichtiges Kriterium für die Verwendbarkeit und Universalität paralleler Softwaresysteme ist die Möglichkeit, *zur Laufzeit* die Struktur der Parallelität (Anzahl und Plazierung paralleler Prozesse sowie deren Kommunikationsmöglichkeiten untereinander) vom Programm selbst aus beliebig zu verändern, beispielsweise durch das Erzeugen neuer Prozesse (*dynamische Parallelität*).

Müssen alle Prozesse und deren Verbindungen schon *zur Übersetzungszeit* fixiert werden (wie z. B. in OCCAM), spricht man von *statischer Parallelität.*

Händisches Mapping?

Bei expliziter Parallelität ist zu unterscheiden, ob nur die logische Parallelität oder auch die *Abbildung der logischen Struktur auf die zur Verfügung stehende Hardware* beschrieben werden muß[5] (*"Mapping"*- oder *"Embedding"*-Problem).

Wenn dies der Fall ist, ist die Frage, ob diese Zuordnung *in das Programm selbst* einfließt (schlecht, z. B. wegen der Wartbarkeit), *in einem getrennten File* spezifiziert wird, oder erst *zur Link- oder Ladezeit* festgelegt werden muß[6].

Korn der Parallelisierung:

Ein Algorithmus läßt sich auf verschiedenen Niveaus in parallele Komponenten zerlegen (geordnet nach zunehmender Korngröße):

Parallelität innerhalb zusammengesetzter Datenstrukturen:

Hier werden die einzelnen Elemente einer Datenstruktur (üblicherweise eines Feldes oder einer Liste) parallel bearbeitet. Musterbeispiel sind Vektorrechner und SIMD-Systeme.

Parallelität innerhalb einzelner Instruktionen:

Hier bewirkt eine einzelne Instruktion mehrere parallele Operationen. Typischer Vertreter ist der MULTIFLOW.

---

[5] In OCCAM zum Beispiel muß jeder Prozeß und jeder Kommunikationskanal statisch im Quellcode auf einen bestimmten Transputer bzw. auf ein bestimmtes Link gelegt werden.

[6] Wie bereits erwähnt, ist das Mappingproblem auch für statische Parallelität NP-vollständig.

Parallelität auf dem Niveau einzelner Instruktionen:

Hier stellt jede Instruktion für sich einen einzelnen Prozeß dar. Beispiele sind vor allem die nichtklassischen Maschinen (Datenfluß-, Reduktions- und Prolog-Maschinen) und teilweise auch der HEP.

Parallele Abarbeitung von Schleifen:

Hier werden verschiedene Iterationen einer einzelnen Schleife gleichzeitig auf verschiedenen Prozessoren in Angriff genommen. Dieses Verfahren kommt vor allem bei Supercomputern zur Anwendung.

Parallele Codeblöcke:

Hier werden ganze Anweisungsfolgen innerhalb einzelner Funktionen parallel abgearbeitet. Beispiele dieser Technik sind das `PAR` in Occam und das `cobegin` in Concurrent Pascal sowie Konstrukte in manchen parallelen Fortran-Dialekten.

Parallele Funktionsauswertung:

Hier ist die kleinste Einheit paralleler Abarbeitung ein ganzer Funktionsaufruf; mehrere Funktionsauswertungen können parallel durchgeführt werden. Im Normalfall sind die parallel auszuwertenden Funktionsaufrufe Teil eines eigenen Sprachkonstruktes (`pcall` in manchen Lisp-Varianten, `eval` in Linda), oder sie resultieren aus impliziter Parallelität (parallele Auswertung aller Argument-Ausdrücke in Funktionsaufrufen bei Lisp und funktionalen Sprachen).

Parallele Module oder Tasks:

Hier werden ganze Programmmodule mehr oder weniger unabhängig voneinander bearbeitet. Typische Vertreter finden sich bei parallelen Erweiterungen konventioneller Programmiersprachen (C unter Unix, Modula-2, etc.).

Als letzte, grobkörnigste Stufe könnte das parallele Bedienen mehrerer Jobs oder Benutzer auf Betriebssystemniveau gelten. Da es sich dabei aber nicht um für den Anwender sicht- oder nutzbare Parallelität handelt, wollen wir nicht weiter darauf eingehen.

Ein anderes Maß für die Korngröße läßt die Struktur der Parallelität außer acht und beschränkt sich auf zwei ganz objektiv und statistisch erfaßbare Größen:

Datenmenge pro Kommunikation:

Hier geht es darum, wie viele Bits oder Bytes bei einer einzelnen Kommunikation im Schnitt übertragen werden. Bei SIMD-Rechnern können das durchaus einzelne Bits sein, bei Systemen mit verteiltem Speicher werden erst Nachrichten ab Kilobyte-Größe als sinnvoll und effizient angesehen.

Instruktionen pro Kommunikation oder Synchronisation:

Hier geht es darum, wie viele Instruktionen typischerweise sequentiell und unabhängig von anderen Prozessen zwischen zwei Kontakten mit anderen Prozessen (Kommunikation, Synchronisation, Prozeßanfang oder -ende usw.) ausgeführt werden. Das eine Extrem sind wieder SIMD-Rechner, bei denen das einige wenige Instruktionen sein können, das andere Systeme mit verteiltem Speicher, bei denen erst sequentielle Blöcke von einigen zehntausend Instruktionen als vernünftig erachtet werden.

Art der Synchronisation:

Auch hier handelt es sich um eher vage Begriffe mit fließenden Grenzen:

Datenabhängige Synchronisation:

Hier wird der Ablauf der Abarbeitung einzig durch die *Verfügbarkeit der jeweils gerade benötigten Daten* gesteuert, und zwar implizit: Sind die Daten noch nicht da, wird der Zugriff automatisch verzögert. Beispiele sind funktionale Programmiersprachen, paralleles Prolog (soweit es die Unifikation betrifft) und Futures, aber auch Message Passing, Streams und Linda.

Für die formale Behandlung der Semantik und auch für die praktische Programmierarbeit ist es dabei ein wesentlicher Unterschied, ob es eine Möglichkeit gibt, zu prüfen, ob ein Datum schon zur Verfügung steht oder noch nicht, und abhängig von solchen Abfragen den Programmablauf zu steuern (das ist der weit heiklere Fall)[7], oder ob ein Prozeß in jedem Fall blockiert ist, bis die Daten eintreffen.

Programmabhängige Synchronisation:

Hier wird an expliziten Synchronisationspunkten gewartet, bis ein bestimmtes Programmstück fertig abgearbeitet ist. Hierzu zählen Occam's `PAR`, Concurrent Pascal's `cobegin` und Unix's `wait`, sowie alle auf parallelem Funktionsaufruf (oder Methodenaufruf bei objektorientierten Ansätzen) basierenden Modelle.

Ressourcenabhängige Synchronisation:

Hier wird der Programmlauf implizit durch die (exklusive) Verfügbarkeit bestimmter Ressourcen gesteuert. Beispiele sind Semaphore, Mutual Exclusion und Monitore.

[7] Linda's `inp` und `readp` sowie Occam's `ALT SKIP` sind typische Beispiele für solche Prüfmöglichkeiten.

Diese Methode ist aus semantischer Sicht mit Abstand am schlimmsten, da sie laufzeitabhängig und damit normalerweise nichtdeterministisch, nichtreproduzierbar und relativ unabhängig von der eigentlichen Programmlogik ist.

3.3 Strategien zur Parallelisierung

In diesem Abschnitt werden die verschiedenen Konzepte besprochen, nach denen bei der Parallelisierung eines gegebenen Algorithmus vorgegangen werden kann.

Nach der Algorithmenstruktur:

Hier bilden typischerweise verschiedene Teile des Gesamtalgorithmus (der oft dem ursprünglichen, sequentiellen Algorithmus ähnelt) die einzelnen Prozesse. Das resultiert meist in grobkörnigem Parallelismus (parallelisiert werden Programmmodule und äußere Schleifen oder Rekursionen) mit einer fixen und relativ geringen Zahl von Prozessen und unregelmäßiger Kommunikationsstruktur.

Das Problem bei diesem Ansatz ist, den Algorithmus so zu zerteilen, daß möglichst viele Teile gleichzeitig aktiv sein können und alle gleichzeitig aktiven Teile in etwa gleich viel zu tun haben, denn der rechenintensivste sequentielle Teil bestimmt als Flaschenhals die erreichbare Effizienz.

Nach der Datenstruktur:

Hier werden die gleichen Operationen parallel auf allen Elementen einer Datenstruktur (meist eines Feldes) ausgeführt. Dieses Konzept ist typisch für numerische Anwendungen sowie für die Bild- und Signalverarbeitung, es resultiert in regelmäßigen, geometrischen Netzen vieler identer Prozesse und in feinkörniger Parallelität mit synchronem Kommunikationsverhalten, da meist die innersten Schleifen parallelisiert werden.

Durch die Zuordnung mehrerer Datenelemente auf einen einzelnen Prozeß (d. h. durch die Aufteilung des Feldes in Blöcke anstatt in einzelne Elemente) läßt sich die Korngröße exakt den Bedürfnissen anpassen.

Nach dem Pipelineprinzip:

Das Pipelineprinzip kombiniert die beiden obrigen Ansätze; es wird sowohl der Algorithmus als auch die Datenstruktur aufgespalten: Aufeinanderfolgende Teile des Algorithmus werden parallel aneinandergereiht, und die Daten wandern element- oder blockweise durch die einzelnen

Stufen. Es resultiert eine lineare Anordnung verschiedener Prozesse mit einfacher, regelmäßiger Kommunikation.

Auch dieses Konzept findet vor allem bei numerischen Problemen und in der Signalverarbeitung Anwendung, aber auch im Compilerbau. Es hat die gleichen Probleme wie eine Parallelisierung nach der Algorithmenstruktur: Die Prozeßzahl ist fix und oft zu gering, die Pipeline benötigt eine gewisse Anlauf- und Entleerzeit, und die rechenintensivste Stufe bildet den Engpaß, nach dem sich der Durchsatz richtet.

Nach der "Processing Farm"-Methode:

Hier arbeitet eine feste, der jeweiligen Hardware optimal angepaßte Anzahl identischer Kopien des gesamten sequentiellen Programms gleichzeitig und unabhängig voneinander auf verschiedenen, ebenfalls voneinander unabhängigen Teilen des Problems. Ein zentraler Master-Prozeß zerteilt und verteilt die Eingabe portionsweise an die Rechenprozesse und sammelt die Ergebnisse. Das geschieht nicht reihum oder in einem fixen Schema, sondern ganz nach Bedarf und Rechengeschwindigkeit: Sobald ein Rechenprozeß sein Ergebnis abgeliefert hat und untätig wird, bekommt er unabhängig von den anderen eine neue Aufgabe zugewiesen.

Wenn es anwendbar ist (vor allem im Grafikbereich, aber auch bei Such- und Optimierungsproblemen und anderen Aufgaben mit vielen Eingabesätzen), hat dieses Verfahren viele Vorteile: Es ist einfach zu programmieren, da *der ursprüngliche Algorithmus* verwendet wird und innerhalb des Algorithmus *keine Parallelität oder Kommunikation* (und daher auch keine bestimmte Maschinentopologie) nötig ist, die *Parallelität* läßt sich *nach Wunsch* und unabhängig von Algorithmus und Datenstrukturen einstellen, und die *Effizienz* liegt nahe beim Optimum, da jeder Prozeß sofort ein neues Stück Arbeit zugeteilt bekommt, wenn er untätig wird, selbst wenn die einzelnen Berechnungen sehr unterschiedlich lange dauern. Die Kommunikation ist sternförmig und eher grobkörnig.

Diese Methode ist allerdings anspruchsvoll, was die Mächtigkeit und Ausdruckskraft der benötigten Kommunikationskonstrukte betrifft: Sie basiert auf Nichtdeterminismus.

Nach dem Divide-and-Conquer-Prinzip:

Hier werden die zwei oder mehr rekursiven Aufrufe in einem nach dem Divide-and-Conquer-Prinzip arbeitenden Algorithmus parallel zueinander ausgeführt. Derartige Algorithmen treten beim Suchen, Sortieren und Optimieren, bei Backtrackverfahren sowie bei Algorithmen auf Bäumen oft auf.

Der Vorteil ist, daß sich die Prozeßzahl (wächst exponentiell mit der Tiefe) und die Korngröße (sinkt ebenso mit der Tiefe) leicht kontrollie-

ren läßt: Ab einer gewissen Schranke wird nicht mehr parallel, sondern sequentiell wie im herkömmlichen Algorithmus weitergearbeitet.

Das Problem bei diesem Ansatz ist, daß die einzelnen Zweige oft sehr verschieden viel Rechenaufwand benötigen, und daß baumförmige Topologien wie sie hier auftreten bei Systemen mit verteiltem Speicher und automatischem Mapping oft zu Problemen führen.

Durch "Speculative Parallelism":

Diese Methode wird angewendet, wenn es zur Lösung eines Problems mehrere, voneinander unabhängige Möglichkeiten gibt (entweder durch verschiedene Algorithmen oder durch die Bearbeitung der Daten in verschiedener Reihenfolge), bei denen nicht von vorneherein festgelegt werden kann, welche die zielführende oder schnellste ist. Es werden einfach alle Lösungsmöglichkeiten gleichzeitig probiert, und das erste Ergebnis wird verwendet. Alle anderen Versuche werden abgebrochen sobald ein Versuch ein positives Resultat geliefert hat.

Ähnlich der Processing Farm Methode hat dieser Ansatz viele Vorteile: Er verwendet die ursprünglichen, rein sequentiellen Algorithmen und eine einfache, grobkörnige Kommunikationsstruktur, und er lastet die Maschine optimal aus (da alle Prozessoren aktiv sind, bis eine Lösung gefunden wird, und nachher sofort frei werden). Weiters ist das der einzige Fall, in dem häufig *superlinearer Speedup* zu beobachten ist: Benötigen die einzelnen Lösungsmethoden sehr unterschiedliche Rechenzeiten, kann es sein, daß man im sequentiellen Fall sehr viel Zeit mit langsamen oder erfolglosen Versuchen vergeudet, bevor man zum erfolgreichen Fall vordringt, während die Laufzeit des parallelen Programms stets von der Rechenzeit der am schnellsten zum Ziel führenden Variante bestimmt wird.

Auch die Nachteile sind vergleichbar: Man braucht nicht nur nichtdeterministische Kommunikationsprimitiva, sondern auch eine Möglichkeit, die nicht mehr benötigten Berechnungen abzubrechen.

3.4 Programmierumgebungen

Ein wesentliches Hindernis für die Verbreitung paralleler Programmierung war und ist das Fehlen brauchbarer Programmierumgebungen für das *Entwickeln*, *Testen* und *Optimieren* paralleler Software.

3.4.1 Das Debugging

Beim Debugging paralleler Software ergeben sich viele grundsätzlich neue Probleme[8]:

Nichtdeterminismus:

Im Unterschied zu sequentieller Programmierung, wo der Programmablauf durch genau eine Zeitachse beschrieben werden kann, hat bei paralleler Software *jeder Prozeß seinen eigenen, lokalen, von den anderen Prozessen unabhängigen Zeitbegriff.* Das Verhalten des Programmes als Ganzes kann aber sehr wohl von der zeitlichen Relation der Prozesse untereinander abhängen[9], und diese kann bei jedem Lauf anders sein, was folgende Probleme mit sich bringt:

- Fehler sind üblicherweise *nicht eindeutig reproduzierbar.*
- Selbst wenn kein Fehler auftritt, kann jeder Programmlauf *andere Ergebnisse* bringen.
- Da der *Debugger* selbst auch Ressourcen in Anspruch nimmt, *ändert* er auch *das Verhalten des zu untersuchenden Programms.*

  Traces und Profiles sind aus ähnlichen Gründen nicht sinnvoll: Ihre Erstellung und Abspeicherung ändert den zeitlichen Ablauf des Programms und belastet das Kommunikationsmedium.
- Tritt in einem Prozeß ein Fehler auf, kann man wenig über den *Zustand der anderen Prozesse zum Zeitpunkt des Fehlers* aussagen. Es ist auch nicht klar, was mit diesen Prozessen geschehen soll.
- Klassisches *Single-Stepping* ist ebensowenig sinnvoll wie das Retten, Modifizieren und Wiederherstellen des Speicherzustandes mittels *Dumps*, weil es einen *Speicherzustand im eigentlichen Sinn nicht gibt* (die einzelnen Prozesse ändern ihren Speicher unabhängig voneinander).

Deadlocks:

Deadlocks sind ein Phänomen, das nur bei paralleler Programmierung auftritt: *Zwei oder mehr Prozesse warten zyklisch aufeinander und sind*

[8] Ganz allgemein ist bei parallelen Programmen der Zusammenhang zwischen Fehler und Ursache noch schwerer herzustellen als im sequentiellen Fall: Die Ursache kann in anderen Prozessen liegen, u. a. auch in schon terminierten Prozessen.

[9] Beispiele: Linda's `inp` und `readp`, Occam's `ALT`, fast alle Primitive zum exklusiven Zugriff auf gemeinsamen Speicher, ...

dadurch für immer blockiert[10]. Schließlich kommt in den meisten Fällen jede Aktivität zum Erliegen, ohne daß das Programm endet.

Es ist bewiesenermaßen für die meisten Modelle paralleler Programmierung nicht möglich, die Anwesenheit oder Abwesenheit von Deadlocks von vorneherein festzustellen, man ist oft auf das Probieren angewiesen. Selbst wenn der Deadlock bereits aufgetreten ist, ist es meist noch sehr schwierig, seinen Ausgangspunkt und die schuldigen Prozesse ausfindig zu machen: Welche Prozesse sind primär, d. h. innerhalb der zyklischen Abhängigkeit, am Deadlock beteiligt, und welche sind nur deshalb inaktiv, weil sie auf einen dieser Prozesse warten?

Ein weiteres Problem besteht darin, daß bei ernsthaften Verklemmungen im Kommunikationssystem *die Systemsoftware und der Debugger selbst auch betroffen* sind und sich die parallele Hardware dadurch jeder Einflußnahme und jeder Analyse durch den Benutzer entzieht.

Ein ähnliches Problem sind *Livelocks*, bei denen gewisse Ressourcen (Prozessorzeit, Speicher, Puffer, Kommunikationsmedien, etc.) durch fehlerhafte Prozesse überbelegt und monopolisiert werden, sodaß die anderen Prozesse (schlimmstenfalls auch der Debugger) nicht mehr zum Zug kommen. Dieser Fall ist äußerlich noch schwerer als ein Deadlock zu erkennen, da ja Aktivität vorliegt.

Informationsflut:

Einerseits sind statt einem plötzlich *viele Prozesse* im Auge zu behalten (für die üblichen Fullscreen-Debugger wird der Bildschirm sehr schnell zu klein), und auch die *Speichergröße* erreicht neue Dimensionen[11], andererseits sind auch einige zusätzliche Informationen anzuzeigen, die im sequentiellen Fall nicht existieren, beispielsweise der *Zustand der einzelnen Prozesse* (rechnend, bereit, wartend (worauf?), ...) und der *Zustand bzw. die Aktivität der Kommunikationsverbindungen.*

Das Erstellen von Traces und Profiles scheitert ebenfalls oft daran, daß bei einer Vielzahl von Prozessen die Informationen schneller anfallen, als sie abgespeichert werden können.

Parallele Ein- und Ausgabe:

Bei manchen parallelen Systemen kann auch der Zugriff auf Files oder den Bildschirm parallel (und auch nichtdeterministisch) erfolgen. Dazu kommt noch, daß die Ein- und Ausgabe des Debuggers gleichzeitig mit der des Programmes ablaufen muß.

---

[10] Ein klassisches Beispiel sind die *"Dining Philosophers"* mit fünf Philosophen und fünf Gabeln an einem runden Tisch.

[11] Manche Parallelrechner haben viel mehr Hauptspeicher als Plattenspeicher, wodurch das Arbeiten mit Dumps zum Problem wird.

3.4.2 Das Performance Debugging

Selbst wenn ein paralleles Programm korrekt funktioniert, heißt das noch lange nicht, daß es wie erwünscht arbeitet: In vielen Fällen wird nur ein Bruchteil der theoretischen Maschinenleistung erreicht, die parallele Hardware wird nur sehr schlecht ausgenutzt[12]. Für solche Probleme hat sich der Begriff *"performance bugs"* eingebürgert, zur Bekämpfung solcher Bugs sind Werkzeuge zur Beobachtung und statistischen Auswertung der Abarbeitung erforderlich.

Das zentrale Gesetz für alle derartigen Bemühungen lautet:

> *Wenn zeitlich gesehen die Hälfte eines Programms sequentiell ist, kann das Programm als Ganzes bestenfalls doppelt so schnell werden, selbst wenn der parallelisierbare Teil unendlich schnell berechnet wird.*

Unter anderem stellen sich folgende Aufgaben:

Analyse des Algorithmus:

Hier geht es darum, Einsicht in das Verhalten eines parallelen Algorithmus zu gewinnen, um beispielsweise verschiedene Algorithmen und Parallelisierungsstrategien vergleichen zu können.

- Ein wesentliches Kriterium ist der zeitliche Umfang des *sequentiellen Anteils.*
- Noch genauere Aufschlüsse gibt das *Parallelitätsprofil*, das die Anzahl der parallel ausführbaren Prozesse oder Operationen zu jedem Zeitpunkt angibt.
- Ebenfalls interessant ist *unnütze Parallelität*, die zur Beschleunigung des Gesamtalgorithmus nichts beiträgt und schlimmstenfalls durch den vermehrten Overhead sogar eine Verlangsamung bewirkt.

Optimierung des Algorithmus:

Selbst wenn der Algorithmus im Prinzip klar ist und den Vorstellungen entspricht, sind normalerweise noch *Unebenheiten in der Parallelisierung* auszubügeln:

- Oft sind Programme durch *unbeabsichtigte Abhängigkeiten* zwischen Prozessen (zuviele Synchronisationen, ungünstige Reihenfolge von

[12] In der Praxis zeigt sich, daß der erste Versuch üblicherweise weniger als 10 % Maschinenauslastung erreicht und oft sogar langsamer als ein sequentielles Programm arbeitet.

Kommunikationen) unnötig sequentiell[13].

- Ein anderes Problem sind *zeitliche Ballungen*, d. h. gleichzeitige Kommunikations- oder I/O-Aktivitäten vieler Prozesse[14].
- Das Endziel ist eine optimale Verzahnung von Berechnungen und Kommunikationen, sodaß immer einige Prozesse ausführbereit sind, während andere gerade kommunizieren. Dazu sind unter anderem ungleich lange Rechenzeiten voneinander abhängiger Prozesse zu vermeiden.

Optimierung der Prozeßzahl und der Korngröße:

Hier geht es darum, den besten Mittelweg zwischen zu kleinem Korn (und damit zu hohem Verwaltungs- und Speicheraufwand) und zu grobem Korn (mit zuwenig Parallelität und zu langen Synchronisations-Wartezeiten) zu finden.

Optimierung der Blockung:

Im Falle von Datenparallelismus stellt sich die Frage, wie viele Datenelemente am besten in einem Prozeß bearbeitet bzw. bei einer Kommunikation übertragen werden, und ob die Aufteilung zeilenweise, blockweise, zufällig oder sonstwie am günstigsten ist.

Optimierung von Mapping und Load Balancing:

Hier wird versucht, durch Umverteilung von Prozessen eine möglichst gleichmäßige Belastung von Prozessoren, Speichern und Kommunikationsmedien zu erreichen und Engstellen zu vermeiden. Weiters ist es sinnvoll, die Länge der am stärksten belasteten Kommunikationswege zu minimieren.

Optimale Nutzung von Maschineneigenschaften:

Vor allem bei Supercomputern muß die Struktur und Größe von Feldern und Schleifen sorgfältig an Vektorregister, Caches, Speicherbänke usw. angepaßt werden[15].

---

[13] Der erste, naheliegende Ansatz, eine synchrone Pipeline in OCCAM zu parallelisieren, endet beispielsweise fast immer mit einem Programm, das auf Grund der Kommunikation nur rein sequentiell abläuft, weil immer nur ein Prozeß aktiv sein kann.

[14] Wenn beispielsweise tausend Prozesse in jeder Sekunde ein paar Mal gleichverteilt auf eine gemeinsame Variable zugreifen, ist das kein Problem, tun sie das auf Grund irgendwelcher zeitlichen Abhängigkeiten immer alle gleichzeitig, kann das sehr wohl eine deutliche Verzögerung bewirken.

[15] Auf Supercomputern ist der Code von besonders viel verwendeten und daher sorgfältig optimierten Bibliotheksfunktionen für Matrizen-Grundoperationen oft hundertmal länger als das entsprechende sequentielle Programm!

Finetuning von Systemparametern:

Beispiele von solchen für das parallele Verhalten wesentlichen Parametern sind die Paketgröße der Kommunikation oder die Prioritäten von Prozessen.

3.4.3 Benötigte Werkzeuge

Unter anderem wären folgende Unterstützungen hilfreich:

Interaktive Parallelisierungshilfen:

Da eine vollautomatische Parallelisierung oft keine zufriedenstellenden Ergebnisse liefert, ist eine Programmierumgebung, die den Programmierer beim händischen Parallelisieren unterstützt, sinnvoll.

Unter anderem werden folgende Ideen für ein solches Werkzeug diskutiert:

- Analyse des Daten- und Kontrollflusses, Anzeige von Parallelisierungsmöglichkeiten bzw. Parallelisierungshindernissen.
- Korrektheitsprüfung händisch durchgeführter Programmumformungen.
- Grafische Darstellung der Topologie, der Datenaufteilung, der Datenabhängigkeiten usw..
- Durchführung der Routine-Codierung: Automatische Datenaufteilung, automatisches Einfügen von Wiederaufsetzpunkten, automatischer Schutz globaler Zugriffe durch kritische Regionen usw..

Grafische Echtzeit-Beobachtung:

Hier geht es darum, laufend *auf einen Blick* einen ersten Eindruck von der *momentanen Auslastung des Systems während der Programmausführung* zu bekommen. Ein erster Schritt sind *Leuchtdioden* auf jedem Knoten, die die Aktivität des Prozessors und der Kommunikationsverbindungen signalisieren. Auf Workstations kommt beispielsweise eine farbige Darstellung des Gesamtnetzes in Frage: “Hot spots” (stark belastete Knoten oder Verbindungen) in rot, “cold spots” in blau.

Statistische Auswertung im Nachhinein:

Neben den im sequentiellen Fall üblichen Profile-Daten (Ausführungshäufigkeiten und -zeiten) und den offensichtlichen Erweiterungen für parallele Systeme (Anzahl und Umfang der Kommunikationen usw.) ist vor allem eine exakte *Erfassung der Prozeß-Wartezeiten* (welcher Prozeß

wartete wann wie lange worauf?) und der *ungenutzten Prozessorzeiten* wesentlich.

Zusatzhardware für Debugging:

Wesentlich ist, daß das Debugging, Tracing und Profiling die Programmausführung nicht beeinflußt:

- Keine Modifikation des Programms.
- Keine Änderung des zeitlichen Ablaufs.
- Keine zusätzliche Kommunikation.
- Funktion trotz Deadlocks und Livelocks.
- Funktion trotz defekter Prozessoren oder Kommunikationsverbindungen.

Dazu ist es notwendig, hardwaremäßig ein eigenes, von den normalen Kommunikationsmedien unabhängiges Kommunikationsnetz für Systemaufgaben zur Verfügung zu stellen (beispielsweise einen *Servicebus*).

Simulations-Software:

Es sollte möglich sein, Informationen über das zu erwartende Verhalten eines parallelen Programms zu gewinnen, ohne das Programm wirklich auf der Zielhardware auszuführen.

Einerseits könnte man dadurch ein Programm debuggen, analysieren und optimieren sowie Varianten des Programms vergleichen, ohne (im Fall von großen Systemen meist sehr knappe und teure) Parallelrechner-Rechenzeit zu verbrauchen, und andererseits könnte man die Einflüsse von Prozessorzahl, Speicherzugriffszeit, Kommunikationsleistung usw. untersuchen, um so die für eine Anwendung optimale Hardware im Voraus zu bestimmen.

Für diese Zwecke ist es wesentlich, daß das virtuelle Zeitverhalten der Prozesse zueinander und die relative Geschwindigkeit von Prozessoren und Kommunikationssystem korrekt nachgebildet wird, die absolute Geschwindigkeit ist nicht so wichtig. Die Simulation kann daher auf einem kleineren Parallelrechner (oder einem Teil des Zielsystems) oder auf einem rein sequentiellen System erfolgen.

Verifikations-Software:

Noch mehr als im sequentiellen Fall gilt beim Vorhandensein von Nichtdeterminismus, daß sich die Abwesenheit von Fehlern, im speziellen von Deadlocks, wenn überhaupt, dann nur durch formale Verifikation feststellen läßt, nicht aber durch Testen.

Leider ist eine vollständige formale Verifikation von parallelen Programmen theoretisch nur sehr eingeschränkt möglich und in der Praxis noch völlig undurchführbar, aber schon ein Programm, das in der Art von `lint` nach offensichtlichen Fehlern und potentiellen Problemen sucht, wäre eine große Hilfe:

- Deadlocks.
- Nichtterminierende, unerreichbare oder ungebremste Prozesse.
- Stellen mit zeit- oder implementierungsabhängigem Verhalten.
- Ungeschützte Zugriffe auf gemeinsame Ressourcen, I/O-Konflikte.
- Fehlende Freigaben nach exklusiven Zugriffen.
- Partnerlose Kommunikationen, Kommunikationen mit Typfehlern.

3.5 Parallelisierende Compiler

Diese Compiler wurden vor allem dazu entwickelt, aus bereits bestehenden FORTRAN-Programmen (*“Dusty Deck* FORTRAN*”*) mit einem Minimum an Modifikationen ein Maximum an Geschwindigkeit und Maschinenausnutzung auf *Supercomputern* zu erreichen. Darin liegt auch ihr größter Vorteil: Sie erlauben es, bestehende Programme (und auch Programmierer) weiterzuverwenden, und sie vermitteln auf für den Benutzer einfache Art und Weise erste Erfolgserlebnisse auf Vektor- und Parallelrechnern.

Inzwischen gibt es solche Compiler auch schon für C, ADA usw., sogar automatisch parallelisierende COMMON LISP Compiler sind in Entwicklung. Auch der Übergang von SIMD-Systemen und Wenigprozessorsystemen mit globalem Speicher (wie es die Supercomputer sind) zu massiv parallelen MIMD-Systemen auch mit verteiltem Speicher ist in Arbeit.

Die Methoden basieren alle auf umfangreichen *Analysen des Kontroll- und Datenflusses* und auf *automatischen Code-Umstellungen* und bewirken im wesentlichen:

- *Das Aufteilen von Datenstrukturen.*
- *Das Parallelisieren, Ausrollen, Verschmelzen usw. von Schleifen.*
- *Das Parallelisieren voneinander unabhängiger Codeteile* (zum Teil auch schon über Funktionsgrenzen hinweg).

Diese Techniken haben aber auch noch viele Schwächen:

- Auf Anhieb (d. h. ohne Modifikationen des bestehenden Programms) erreicht man üblicherweise bestenfalls 10 % der theoretisch möglichen Rechengeschwindigkeit, Werte von 90 % und mehr sind zwar bei einigen Problemen erreichbar, aber nur mit umfangreichen Programmanpassungen.

  Diese Anpassungen an Compilereigenschaften und Maschinenstruktur (Cache-Organisation, Vektorregister-Größe, Prozessorzahl, Bank- oder Seitengröße etc.) führen allerdings zu einem verkrampften Programmierstil und zu unleserlichen Programmen[16].

- Die heutigen Methoden arbeiten nur bei *Zählschleifen* (FORTRAN `DO`) zufriedenstellend, `while`-Schleifen und Rekursionen werden praktisch nie parallelisiert.

  Auch bei Zählschleifen übersehen die Compiler oft Parallelisierungsmöglichkeiten, weil sie natürlich den ungünstigsten Fall gegenseitiger Beeinflussung von Speicherzellen annehmen müssen und nur der Programmierer genau weiß, welche Seiteneffekte wirklich auftreten und welche nicht.

  Kernprobleme sind *Seiteneffekte* (Call by Reference, nichtlokale Variablen), *überlagerte Variablen und Felder* (FORTRAN `EQUIVALENCE`), *Pointer und Adressarithmetik sowie indirekte Index-Ausdrücke*, und vor allem *destruktive Listen-Operationen* (`setcar`, `setcdr` usw. in der LISP-Familie). Weiters sind alle *Ein- und Ausgabeoperationen* Hindernisse für die Parallelisierung.

- Der Benutzer hat wenig Überblick und auch kaum Kontrollmöglichkeiten über die vom Compiler vorgenommene Parallelisierung[17]. Das erschwert sowohl die Erzeugung optimaler Parallelität (Körnigkeit, Balance, ...) als auch das Debugging: Der *Zusammenhang mit dem Quelltext geht verloren.*

---

[16] Der FORTRAN-Quellcode einer hochoptimierten Bibliotheksfunktion für Matrizenmultiplikation auf Supercomputern umfaßt normalerweise rund ein Dutzend Seiten, obwohl sich der eigentliche Algorithmus in vier Zeilen angeben läßt: Je nach der Größe der beiden Matrizen im Vergleich zu den Vektor-Registern und den Caches werden normalerweise verschiedene Fälle mit total unterschiedlicher Struktur unterschieden, und in jedem Fall werden die ursprünglichen Schleifen in mehrere Segmente aufgespalten.

[17] Es gibt allerdings schon Ansätze zu interaktiven Vektorisierungs- und Parallelisierungs-Environments: Das System teilt dem Programmierer mit, was es erfolgreich bearbeitet hat und was nicht (und vor allem warum nicht), und der Programmierer kann sofort die Wirksamkeit anderer Programmvarianten probieren.

3.6 Höhere parallele Sprachen

Hier wollen wir alle Programmiersprachen und -modelle zusammenfassen, die mehr oder weniger hardwareunabhängig sind und die eigentliche Implementierung der Parallelität und Kommunikation vor dem Benutzer verbergen.

3.6.1 Rein funktionale Sprachen

Diese Sprachen — auch *"applikative Sprachen"* genannt — umfassen ausgehend vom Urvater FP u. a. ML, Miranda, SASL, Hope, VAL, Id, Haskell und Pure Lisp, nicht aber das übliche Lisp. Auf ähnlichen Konzepten basieren *Term-Rewrite-Sprachen*, *Sprachen für algebraisch spezifizierte abstrakte Datentypen*, *Datenflußsprachen* usw..

Bei funktionalen Sprachen entsprechen die Programme *Funktionen im mathematischen Sinn*:

- *Funktionen sind frei von Seiteneffekten*, es gibt nur *Call by value* und *lokale Variablen*[18].

  Jeder Ausdruck liefert ein Ergebnis, hat aber sonst keine Auswirkungen. Dieses Ergebnis hängt nur von den Argumenten ab, nicht von der Vorgeschichte der Berechnung oder anderen Einflüssen außerhalb der Funktion (*"referential transparency"*).

- Es gibt *keine Zuweisung*; ein Name ("Variable" ist ein bei funktionalen Sprachen unangebrachter Ausdruck, weil eben nichts "variabel" ist) wird einmal und für immer mit einem Wert verbunden ("single assignment"), ist also eine Konstante.

Funktionale Sprachen haben folgende Vorteile:

Sie sind deterministisch:

Das bedeutet, daß ein funktionales Programm zu einer bestimmten Eingabe *stets dieselbe Ausgabe* liefert; das Ergebnis ist per definitionem unabhängig von Grad und Art der Parallelisierung, von Abarbeitungsreihenfolge und relativer Rechengeschwindigkeit der einzelnen Prozesse usw..

[18] Wie weiter unten erwähnt, gibt es eigentlich überhaupt keine Variablen, sondern nur Konstanten. Aber auch für diese gelten *strikt lexikalische Gültigkeitsbereiche*: Jeder neu definierte Namen kann nur ihm untergeordnete Funktionen und Ausdrücke beeinflussen, er kann keine Rückwirkungen 'nach außen' haben.

Das ist für das Debugging und die Korrektheitsprüfung von Programmen ein großer Vorteil (wenn es einmal funktioniert hat, dann funktioniert es immer).

Sie bieten ein Maximum an impliziter Parallelität:

Auf der einen Seite wird die Abarbeitungsreihenfolge nur durch die vom Algorithmus vorgegebenen, essentiellen *Datenabhängigkeiten* bestimmt; es gibt weder einen Kontrollfluß im üblichen Sinn, der zusätzliche Sequentialisierungen bedingt, noch Möglichkeiten zu Seiteneffekten, die eine Parallelisierung verhindern.

Auf der anderen Seite bieten funktionale Sprachen die Möglichkeit, bis herunter auf das Niveau einzelner Grundoperationen zu parallelisieren, kein anderes Konzept bietet ein derartig breites Spektrum für die Körnigkeit der Parallelität.

Sie haben eine klare und einfache formale Semantik:

Üblicherweise wird die Semantik funktionaler Sprachen durch einen *Rewrite-Kalkül* angegeben, der einerseits gut formal zu behandeln ist, andererseits aber nahe genug an den tatsächlichen Verarbeitungsmethoden liegt, um durch Transformationen auch die Korrektheit von Implementierungen zu beweisen.

Sie sind sehr high-level:

Viele dieser Sprachen bieten unter anderem folgende Fähigkeiten:

Funktionen höherer Ordnung:

Das bedeutet, daß auch Funktionen ganz normale Datenobjekte (“first class citizens”) sind und beispielsweise als Funktionsargumente oder -ergebnisse übergeben werden können. Damit ist es möglich, Funktionen zu schreiben, die andere Funktionen als Argumente mitbekommen und dann intern aufrufen oder zu neuen Funktionen kombinieren (einfachstes Beispiel ist die `map`-Funktionsfamilie in Lisp).

Curried Functions:

Dieses Konzept besagt, daß Funktionen angewendet auf nur einen Teil ihrer Argumente wieder vollwertige, eigenständige Funktionsobjekte mit entsprechend geringerer Stelligkeit ergeben (auf diese Weise würde beispielsweise die Addition mit dem einen Argument 1 die Nachfolgerfunktion ergeben).

Generische Funktionen und Datentypen, Polymorphismus:

Im Unterschied zu Lisp beruhen funktionale Sprachen meist auf *strenger, aber impliziter Typprüfung* (d. h. der Compiler führt eine

Typprüfung ähnlich wie in Pascal durch, aber die Typen werden nicht vom Benutzer deklariert, sondern automatisch vom Compiler hergeleitet).

Neben den aus konventionellen Sprachen bekannten Konzepten der Modularisierung und der abstrakten Datentypen bieten funktionale Sprachen zusätzlich die Möglichkeit, mit generischen Funktionen und Typen zu arbeiten, d. h. gleich eine ganze Klasse von Datentypen und darauf arbeitenden Funktionen (z. B. die Klasse aller Listen-Typen, gleich welcher Repräsentation und welchen Elementtyps, oder die Klasse aller Polynome) in einer einzigen Definition zu beschreiben.

Auch *Operator Overloading*, d. h. die Möglichkeit, Funktionen gleichen Namens, aber verschiedener Argumenttypen, zu definieren, von denen dann automatisch die richtige ausgewählt wird, wird oft ermöglicht.

Streams:

Streams sind Datenstrukturen, die einer *nicht-strikten Liste mit lazy evaluation* ähneln: Sie sind eine beliebig (theoretisch auch unendlich) lange Folge einzelner Elemente, wobei jedes Element erst dann berechnet wird, wenn es tatsächlich benötigt wird.

Pattern Matching:

Manche funktionale Sprachen erlauben es, anstatt der formalen Parameter Terme anzugeben, und so eine Funktion für die einzelnen möglichen Fälle getrennt *in mehreren Teilen zu definieren.*

Zur Laufzeit werden dann die aktuellen Argumente mit den formalen Parametern gematcht, und die für diese Argumente zuständige Funktionsdefinition wird ausgeführt. Das erlaubt einen Prolog-ähnlichen Programmierstil.

Funktionale Sprachen sind aber auch noch mit vielen Problemen behaftet:

Schlecht geeignet für konventionelle Hardware:

Funktionale Sprachen sind auf konventionellen Parallelrechnern nur sehr schlecht implementierbar[19]: Ihre Speicherstruktur und Abarbeitungsstrategie entsprechen nicht dem von-Neumann-Modell, die Parallelität ist (wenn man sie in vollem Umfang ausnutzt) für heutige Begriffe viel

[19] Effiziente sequentielle Implementierungen sind hingegen relativ einfach zu bewerkstelligen, wenn die Auswertung strict und eager vorgenommen wird (was leider die am wenigsten mächtige Strategie ist).

zu feinkörnig[20], und auch die Notwendigkeit eines großen globalen Speichers[21] entspricht nicht gerade den technischen Gegebenheiten.

Man hofft, daß sich die Situation mit der Verfügbarkeit von Datenfluß- und Reduktionsmaschinen bessert.

Zu viel Parallelismus:

Nutzt man wirklich den gesamten impliziten Parallelismus aus, ergibt sich ein explosiver Speicherbedarf. Es gilt daher, gerade soviel Parallelität zu erzeugen, daß die Maschine optimal genutzt wird (d. h. daß keine Prozessoren leer stehen).

Keine modifizierbaren Datenstrukturen:

Im Prinzip lassen sich einzelne Komponenten einer Datenstruktur nach dem Erzeugen der Struktur *nicht mehr ändern*; es ist notwendig, eine *neue Kopie der gesamten Struktur* mit dem einen geänderten Element anzulegen. Das erfordert sehr hohen Kopieraufwand und Speicherbedarf.

Besonders für numerische Anwendungen ist es ein großes Problem, daß es vor allem Arrays im herkömmlichen Sinn nicht gibt: Im Prinzip müßten alle Elemente eines Arrays zum Zeitpunkt seiner Erzeugung bekannt sein, sie können nachträglich nicht mehr belegt oder verändert werden[22].

Einige Problemklassen werden durch diese Einschränkungen auch grundsätzlich getroffen, beispielsweise die *akkumulierenden Probleme*[23].

Auch einige klassische Aufgaben der *Systemprogrammierung* basieren auf modifizierbarem Speicher, beispielsweise die Garbage Collection[24]. Ebenso ist jede *Ein- oder Ausgabeoperation* grundsätzlich ein Seiteneffekt, jedes *Filesystem* ein modifizierbarer Speicher. Beide sind daher mit der Semantik funktionaler Sprachen unvereinbar.

Fehlender Nichtdeterminismus:

---

[20] Natürlich steht es jeder Implementierung frei, mit einem gröberen als dem semantisch möglichen minimalen Korn zu arbeiten, aber das im-Voraus-Abschätzen von Laufzeit und Speicherbedarf einzelner Programmfragmente in Programmiersprachen mit dynamischen Speicherstrukturen und Rekursion ist noch nicht zufriedenstellend gelöst.

[21] Auch hier kann man natürlich andere Implementierungsstrategien wählen, aber die Semantik legt einen globalen Speicher nahe.

[22] Manche Sprachen bieten Erweiterungen in diese Richtung (z. B. die I-structures von Id), aber auf Kosten der formalen Semantik.

[23] Musterbeispiel: Zähle, wie oft jeder Buchstabe in einem Text enthalten ist!

[24] Daher ist es notwendig, diese Aufgaben bei nichtklassischen Maschinen entweder direkt von spezieller Hardware ausführen zu lassen, oder die Maschinenarchitektur für solche Systemverwaltungsaufgaben um von-Neumann-ähnliche Möglichkeiten zu erweitern.

Für manche Algorithmentypen, vor allem für den *"speculative parallelism"*, bei dem ein Problem gleichzeitig mit verschiedenen Lösungsmethoden bearbeitet wird und nur das als erstes eintreffende Ergebnis von Interesse ist, wäre Nichtdeterminismus und eine Möglichkeit zum expliziten Abwürgen nicht mehr benötigter Berechnungen notwendig.

Endlose Zweige:

Berechnet man Ausdrücke allzu eager (d. h. auf Verdacht, bevor feststeht, ob sie jemals wirklich gebraucht werden), können sich auch bei korrekten Programmen endlos rekursive Zweige ergeben.

Umerziehung der Programmierer:

Um gute und effiziente funktionale Programme zu schreiben, sind ganz andere Denkweisen, teilweise sogar andere algorithmische Ideen, notwendig; dies dürfte einer der Hauptgründe für die bis jetzt geringe Akzeptanz solcher Sprachen sein.

Auch die Methoden des Debuggings müssen sich ändern, wenn es plötzlich keinen Programmzähler und keine Variablen mehr gibt.

Mehrfachberechnungen:

Ein naiver Programmierstil (und/oder unintelligente Compiler) führen oft zu unnötigen Mehrfachauswertungen von Funktionen mit identen Argumentwerten (klassisches Beispiel: Berechnung der Fibonaccizahlen in der zweifach rekursiven Form).

Fehlende Schleifen:

Da eine Schleife darauf beruht, daß sich der Zustand bei jedem Durchlauf ändert (sonst wäre keine Termination möglich), es in funktionalen Sprachen aber keinen Zustand gibt, gibt es im Prinzip auch keine Schleifen[25]. Deren *Ersetzung durch Rekursionen* wirkt oft stilistisch unschön und braucht zusätzlichen Speicherplatz.

Die einzelnen funktionalen Programmiersprachen unterscheiden sich vor allem in der Abarbeitungssemantik:

Lazy oder eager Evaluation?

Bei *eager Evaluation* wird eine Operation *so bald wie möglich* ausgeführt (d. h. sobald alle benötigten Operanden verfügbar sind), unabhängig davon, ob ihr Ergebnis benötigt wird oder nicht.

Bei *lazy Evaluation* wird eine Operation erst dann begonnen, wenn feststeht, daß ihr *Ergebnis tatsächlich benötigt* wird.

---

[25] Viele Sprachen bieten aber schleifenähnliche Erweiterungen.

Entscheidend ist das bei *Streams* (einen potentiell unendlichen Stream eager zu berechnen ist nicht sinnvoll, es vergeudet Zeit und Speicherplatz[26]) sowie bei `if`: Stets beide Zweige auszuwerten ist nie sinnvoll, oft sogar gefährlich: Ein Zweig enthält oft einen rekursiven Aufruf, was zu endloser Rekursion führen würde. Ähnliches gilt für `and` und `or`: Abhängig vom ersten Ausdruck ist der zweite oft undefiniert.

Strikte oder nicht-strikte Funktionen?

Hier ist die Frage, ob man erst dann beginnt, eine Funktion auszuwerten, wenn alle Argumente fertig berechnet sind (strikt), oder ob man unabhängig von den Argumenten sofort beginnt, so weit wie möglich loszurechnen (nicht-strikt).

Vor allem wenn auf Grund von Verzweigungen manche Argumente in der Funktion überhaupt nicht verwendet werden, ergibt sich ein wesentlicher Unterschied (beispielsweise wenn gerade die Berechnung dieser Argumente zu einem Fehler führt).

Strikte oder nicht-strikte Datentypen?

Ähnlich wie bei den Funktionen geht es hier darum, ob alle Komponenten einer Datenstruktur fertig berechnet sein müssen, um mit der Struktur als Ganzes weiterrechnen zu können (strikt), oder ob mit der Datenstruktur als "black box" sofort weitergearbeitet werden kann, obwohl ihre Komponenten noch unbekannt sind (nicht-strikt).

Diese Frage beeinflußt vor allem die erzielbare Parallelität.

Diese Strategien beeinflussen unter anderem

- *Mächtigkeit der Sprache und Zeitkomplexität der Programme*,
- *Möglichkeiten für zyklische Datenstrukturen*,
- *überflüssige und mehrfache Berechnungen*,
- *erreichbare Parallelität und Speicherverhalten*,
- *Auftreten von endlosen Rekursionen und Datenstrukturen*,
- *Auftreten und Fortpflanzung von Fehlern und undefinierten Werten und*
- *das Terminationsverhalten*[27].

---

[26] Gerade bei Streams geht man auch Mittelwege (man rechnet in begrenztem Maß auf Vorrat), denn reine lazy evaluation bewirkt eine unnötig sequentielle Verarbeitung.

[27] Bei einigen Sprachen und Implementierungen kann es vorkommen, daß das Endergebnis geliefert wird lange bevor die Berechnung terminiert (wenn sie überhaupt terminiert).

Explizit parallele funktionale Sprachen

Neben den bisher besprochenen Sprachen, die auf optimale Nutzung impliziter Parallelität ausgelegt sind, gibt es auch *funktionale Sprachen für explizite Parallelprogrammierung*, z. B. FP2. Diese Sprachen verknüpfen Funktionen durch spezielle Funktoren zu neuen Funktionen, wobei die einzelnen Teilfunktionen in einer durch den Funktor bestimmten Art parallel ausgeführt werden. Dieser Ansatz zeichnet sich durch eine besonders schöne und klare Semantik für parallele Berechnungen aus; scheint aber in der Praxis keine Bedeutung zu erlangen.

3.6.2 Paralleles Prolog

In PROLOG gibt es viele verschiedene Quellen der Parallelität:

OR-Parallelismus:

Hier wird versucht, die Köpfe *aller für ein Goal in Frage kommenden Klausen gleichzeitig* mit dem Goal zu unifizieren, und alle erfolgreich unifizierten Klausen parallel und unabhängig voneinander weiterzuuntersuchen. Es werden also alle möglichen Alternativen gleichzeitig untersucht.

AND-Parallelismus:

Hier werden in jedem Schritt *alle zu beweisenden Goals*, d. h. alle noch verbliebenen Teilprobleme, gleichzeitig bearbeitet.

Parallele Unifikation:

Es gibt auch Ansätze, innerhalb einer einzigen Unifikation eines Goals mit einem Kopf parallele Algorithmen anzuwenden, die resultierende Parallelität ist allerdings sehr feinkörnig.

Datenparallelität:

Diese Form der Parallelität tritt in Form von *Streams* auf. Das sind im Prinzip beliebig lange Listen, die allerdings nicht als Ganzes, sondern Element für Element verarbeitet werden:

Der Stream wird vom *Producer* erzeugt, indem an die den Stream repräsentierende Variable ein Paar-Term gebunden wird, der aus dem Kopfelement und einer neuen Variablen für den noch unbestimmten Rest des Streams besteht. Dann ruft sich der Producer mit dieser Variablen als Stream selbst rekursiv auf.

Der *Consumer* verarbeitet den Stream, indem er im Kopf einen Paar-Term mit zwei Variablen enthält. Er wird daher bei der Unifikation

angehalten, bis der Stream vom Producer auf einen Paar-Term gebunden wird. Die erste Variable enthält dann das zu verarbeitende Element, die zweite repräsentiert den Rest des Streams (eine noch ungebundene Variable) und wird am Ende des Bodies wieder für den rekursiven Selbstaufruf verwendet. Eine Consumer-Klause mit der leeren Liste anstatt eines Paar-Terms im Kopf wird vom Unifizierer gewählt, wenn der Stream zu Ende ist.

Wenn die Elemente des Streams noch ungebundene Variablen enthalten, ist auch ein Datenaustausch vom Consumer zurück zum Producer möglich.

Im wesentlichen gibt es drei verschiedene Ansätze für paralleles PROLOG: Reines PROLOG, annotiertes PROLOG und explizit paralleles PROLOG.

Reines Prolog

Bei diesem Ansatz werden konventionelle, den üblichen PROLOG-Konzepten entsprechende PROLOG-Programme ("dusty deck") durch einen intelligenten Compiler oder Interpreter *automatisch implizit parallel* abgearbeitet.

Im Prinzip sollte das — der rein deklarativen PROLOG-Semantik entsprechend — problemlos möglich sein und die Semantik gegenüber der sequentiellen Abarbeitung nicht ändern.

In der Praxis hat der Ansatz folgende Nachteile:

Unübliche operationale Semantik:

Die meisten PROLOG-Programme sind unter der Annahme der *üblichen operationalen* PROLOG-*Semantik* geschrieben worden, d. h. sie funktionieren nur korrekt, wenn die in Frage kommenden Klausen textuell von oben nach unten geprüft und die Goals innerhalb einer Klause von links nach rechts ausgewertet werden.

Unendliche Zweige:

Aus dem vorigen Punkt ergibt sich oft auch, daß diese Programme bei Anwendung deklarativer Semantik eine Vielzahl *unendlicher Zweige* enthalten — man müßte diese Zweige automatisch erkennen und einbremsen, ohne dabei essentielle Berechnungen zu treffen.

Explizite Seiteneffekte:

Viele PROLOG-Programme enthalten *explizite Seiteneffekte*, z. B. den Cut-Operator (`!`) oder Manipulationen der Datenbasis (`assert` und `retract`). Diese sind mit paralleler Verarbeitung unvereinbar (in welcher Reihenfolge sollen die Seiteneffekte ausgeführt werden?).

Nichtdeterminismus:

Will man nicht alle, sondern nur eines aller möglichen Ergebnisse, so ist das Resultat oft *extrem nichtdeterministisch* (abhängig von Implementierung, relativer Rechengeschwindigkeit der Prozesse zueinander, etc.).

Kollidierende Variablenbindungen:

Für jede Variable müssen viele Bindungen gleichzeitig verwaltet werden:

- Durch den Or-Parallelismus kann eine Variable durch verschiedene Klausen gleichzeitig an mehrere verschiedene Werte gebunden werden. Jeder einzelne dieser Werte stellt für sich eine gültige Lösung dar und muß unabhängig von den anderen weiterbearbeitet werden.
- Durch den And-Parallelismus kann eine Variable durch verschiedene Goals innerhalb eines Bodies an verschiedene Werte gebunden werden. Hier ergibt sich die Lösung als die Menge all jener Variablenbelegungen, die in allen Goals gemeinsam auftreten. Diese müssen herausgefiltert, alle anderen Bindungen ignoriert werden.

Speicherbedarf:

Auch die Implementierung stößt auf Schwierigkeiten: Durch die breite Suche ergibt sich ein *extremer Speicherbedarf*; noch dazu wird der Speicher als global vorausgesetzt.

Vertreter dieses Typs parallelen Prologs sind über die experimentelle Phase noch nicht hinausgekommen.

Annotiertes Prolog (Committed Choice Prolog)

Die Probleme der Suchsteuerung und der Variablenbindungen wurden hier auf den *Programmierer* abgeschoben, der durch spezielle Operatoren, Deklarationen usw. im Programmtext den *Ablauf selbst kontrollieren* muß:

Aufteilung in Guard und Body:

Der Body einer jeden Klause wird durch den *Commit Operator* (|) in das sogenannte *Guard* (vorne) und den eigentlichen Body getrennt. Das steuert den Or-Parallelismus: Die Guards aller in Frage kommenden Klausen werden parallel und unabhängig voneinander ausgeführt, bleiben aber nebenwirkungsfrei, d. h. sie binden keine Variablen[28]. Die erste Klause,

[28] Die meisten Varianten haben noch weitere Einschränkungen zur einfacheren Implementierung: Die Unifikation des Kopfes und die Auswertung der Guards dürfen überhaupt keine Variablen binden, sondern nur bereits anderweitig gebundene Variablen verwenden;

die erfolgreich zum Commit-Operator vordringt, wird ausgewählt: Ihre Bindungen werden allgemein gültig, und ihr Body wird weiter verfolgt. Die Bearbeitung aller anderen Klausen wird abgebrochen als hätte sie nie stattgefunden.

Sequentialisierung:

Spezielle *"sequential And"* und *"sequential Or"* Operatoren können dazu verwendet werden, die übliche "von oben nach unten, von links nach rechts"-Semantik für einzelne Programmteile wieder einzuführen.

Festlegung der Variablenbindung:

Zur Vermeidung von Problemen mit Bindungen muß die Variablenrichtung definiert werden, entweder indem man einzelne Vorkommen von Variablen als *"read only"* kennzeichnet, oder indem man für jedes Prädikat von vorneherein deklariert, welche Variablen es selbst bindet und welche es nur konsumiert (*"Mode declaration"*). Stößt man bei der Abarbeitung auf eine Variable, die das betreffende Prädikat nicht selbst binden darf, die aber noch nicht anderweitig gebunden ist, so wird diese Auswertung verzögert, bis diese Variable an einer anderen Stelle gebunden wird[29].

Dieser Ansatz bringt einige Vorteile:

- Er ist einfach und sehr *effizient* implementierbar, sogar effizienter als normales, sequentielles Prolog[30].
- Die Abarbeitung ist leicht nachvollziehbar, im Normalfall sogar *deterministisch*; die Parallelität läßt sich ebenso exakt kontrollieren wie explizite Seiteneffekte (I/O, `assert`/`retract`).

Manche empfinden es allerdings als Nachteil, daß die resultierenden Sprachen praktisch *nur mehr syntaktische Gemeinsamkeiten* mit Prolog haben:

- Die Semantik unterscheidet sich wesentlich sowohl von der idealen, denotationalen Prolog-Semantik als auch von der üblichen "von oben nach unten, von links nach rechts"-Semantik:

---

die Prädikate im Guard müssen elementar, d. h. eingebaut und nicht vom Benutzer definiert, sein (man nennt diese Prologs dann *"flach"*, weil innerhalb der Guards keine Rekursion auftreten kann), es darf stets maximal eines aller in Frage kommenden Guards erfolgreich sein usw..

[29] Auch hier gibt es weitergehende Einschränkungen, beispielsweise die, daß statisch anhand des Programmtextes überprüfbar sein muß, daß jede Variable nur von genau einem Prädikat gebunden werden kann.

[30] Es gibt Implementierungen, die bis herunter auf Maschinencode compilieren.

  * Die Abarbeitung ist deterministisch, es gibt *kein Backtrack* mehr: Es wird *maximal eine Lösung* berechnet, und wenn ein Guard ungünstig entscheidet, kann es durchaus sein, daß die Abarbeitung mit “failed” abbricht, obwohl andere Klausen sehr wohl eine Lösung geliefert hätten.
  * Die übliche Semantik des `not` (“Negation by Failure”) ist ebenfalls nicht mehr gegeben, und den Cut-Operator gibt es auch nicht mehr.

- Eine der herausragendsten Eigenschaften von Prolog, nämlich die Vertauschbarkeit von Eingabe und Ausgabe, geht ebenfalls verloren: Variablen sind nicht mehr in jede Richtung verwendbar, sondern müssen schon im Programmtext als *Eingabe oder Ausgabe fixiert* werden[31].

Diese Unterschiede beeinflussen den Programmierstil stark (Programme degenerieren praktisch zu `if-then-else`-Ketten), und sie haben auch das typische Einsatzgebiet verändert: Anstatt in AI-Anwendungen (z. B. Expertensystemen) findet man diese Sprachen in rechenintensiven Problemen, ja sogar in der Systemprogrammierung.

Die Vertreter dieser Gruppe stellen den Großteil der heute als paralleles Prolog bezeichneten Systeme, zu ihnen gehören GHC (Guarded Horn Clauses) vom japanischen ICOT, Parlog vom Imperial College London und Concurrent Prolog vom israelischen Weizmann-Institut.

Weiters gehört die Sprache Strand, die kommerziell für eine Vielzahl von Rechnern, aber auch zum Einsatz auf lokalen Netzen sequentieller Rechner angeboten wird, zu dieser Familie.

Explizit paralleles Prolog

Hier handelt es sich um normales Prolog mit der üblichen, sequentiellen operationalen Semantik, das um einige *spezielle, eingebaute Prädikate zur Erzeugung und Synchronisation paralleler Prozesse sowie zur Kommunikation* (z. B. mittels ganz konventionellem Message Passing) erweitert wurde, ähnlich wie beispielsweise C oder Fortran durch Bibliotheken mit Systemfunktionen für Parallelverarbeitung erweitert wurden.

Diese Sprachen bewegen sich zwar softwaretechnisch auf niedrigerem Niveau, aber sie sind einfach zu implementieren und einfach zu durchschauen. Sie erwiesen sich für bestimmte Anwandungsgebiete als hervorragend geeignet.

Der wichtigste Vertreter ist CS-Prolog der Firma Multilogic.

---

[31] Parlog kommt beispielsweise zur Laufzeit ohne jede Unifikation aus!

3.6.3 Paralleles Lisp

Bei parallelem Lisp gibt es die größte Variantenvielfalt, es gibt kaum ein Konzept paralleler Programmierung, das nicht mit Lisp kombiniert wurde:

Implizite Parallelität:

Auf der einen Seite gibt es Ansätze, die relativ *nahe bei rein funktionalen Programmiersprachen* stehen und auch mit ähnlichen Konzepten arbeiten. Sie beziehen sich vor allem auf Pure Lisp oder Scheme.

Auf der anderen Seite gibt es Versuche, normales Lisp (Common Lisp, Portable Standard Lisp) in vollem Umfang mittels intelligentem Compiler (ähnlich parallelisierenden Fortran-Compilern) zu parallelisieren, z. B. an der Universität Bath. Diese beruhen auf zwei Konzepten:

Datenfluß- und Side-Effect-Analyse:

Sie gibt an, wo man parallelisieren darf (im wesentlichen wird jede Funktion als "sicher" oder "unsicher" markiert).

Gewichtsanalyse:

Basierend auf Rekursionsaufrufen und Schleifen ordnet sie jeder Funktion eine Korngrößen-Kennzahl ("Gewicht") zu, die helfen soll, zu entscheiden, wo sich die Parallelisierung auszahlt.

Datenflußkonzepte:

In Qlisp gibt es die sogenannten PMI-Funktionen (*"partially, multiply invoced functions"*). Diese Funktionen können ihre Argumente in beliebiger Reihenfolge einzeln und unabhängig voneinander erhalten (*"curried functions"*), und verhalten sich dabei selbst stets als normales Objekt, egal, ob sie bisher keine, einige oder alle Argumente bekommen haben, können also unter anderem zwischen Prozessen hin- und hergeschickt werden. Dadurch ist es möglich, Funktionen zu schreiben, bei denen die Argumente eines Aufrufes von verschiedenen Prozessen geliefert werden, wodurch sich eine datenflußähnliche Verarbeitungsweise ergibt.

Nicht-strikte Datentypen:

Das von funktionalen Sprachen bekannte Konzept der nicht-strikten Datentypen beziehungsweise der Streams wird auch in Lisp angewendet.

Es wurde schon mit allen vier möglichen Varianten des `cons` experimentiert: Beide Teile strikt (wie üblich), beide Teile nicht-strikt, vorne strikt und hinten nicht (Streams) und umgekehrt.

Nicht-strikte Funktionen:

In Qlisp können Funktionsargumente automatisch in Futures verpackt werden, wodurch sich exakt der Effekt nicht-strikter Funktionen ergibt: Der Funktionsrumpf kann zu rechnen beginnen, bevor die Argumente fertig ausgewertet sind.

Explizite parallele Evaluierung von Funktionsargumenten:

Diese Form der Parallelität tritt u. a. in Multilisp auf: Bei `(pcall` *fun arg1 arg2 ...*`)` werden die aufzurufende Funktion und die Argumente parallel ausgewertet, dann wird die Funktion normal mit den so berechneten Argumenten aufgerufen und ihr Ergebnis als Ergebnis von `pcall` zurückgegeben.

Beim `qlambda` in Qlisp geht man noch weiter: Es verhält sich wie ein `qlet` (parallele Auswertung der Argumente, nicht-strikte Auswertung des Rumpfes), liefert sofort ein Future als Funktionsergebnis, und besitzt zusätzlich auf Wunsch noch Monitor-Eigenschaften, d. h. zu jedem Zeitpunkt kann maximal ein Aufruf von `qlambda` aktiv sein.

Explizites Kreieren paralleler Prozesse:

Explizites, dynamisches Kreieren paralleler Prozesse und alle damit zusammenhängenden Kontrollfunktionen (verzögern, killen, suspendieren usw.) werden in vielen Dialekten geboten, z. B. in Spur Lisp und Qlisp.

In Qlisp hat jedes Konstrukt, das implizit oder explizit parallele Prozesse erzeugt, einen zusätzlichen Parameter: Ist er `nil`, wird das Konstrukt rein sequentiell (wie das zugrunde liegende normale Lisp-Konstrukt) ausgeführt, sonst parallel. Da der Programmierer über spezielle Funktionen den Belastungszustand des Systems (Anzahl der wartenden Prozesse usw.) abfragen kann, ist es möglich, dynamisch stets genau die für die optimale Nutzung des Rechners angemessene Parallelität und nicht mehr zu erzeugen.

Explizite parallele Zuweisung:

Hier ist Qlisp der typische Vertreter: Es gibt eine ganze Familie von Konstrukten (`qlet`, `qprogn` usw.), die sich im wesentlichen so wie ihre sequentiellen Namensvettern verhalten, aber die Initialisierungen der lokalen Variablen parallel vornehmen. Weiters gibt es von diesen Konstrukten auch Varianten, bei denen auch die Auswertung des Rumpfes parallel zu den Initialisierungen begonnen wird (*nicht-strikt*). Zu diesem Zweck werden die lokalen Variablen erst einmal mit Futures initialisiert, und erst wenn im Rumpf der tatsächliche Wert benötigt wird, wird automatisch gewartet, bis die entsprechende Initialisierung fertig ist.

Futures und Delays:

Diese beiden Konstrukte sind eine explizite, sehr universelle Form nichtstrikter Datentypen und Funktionen; sie wurden erstmals in Multilisp eingeführt.

Ein Aufruf von `(future` *expr*`)` liefert sofort ein Datenobjekt (ein *"Future"*, d. h. ein "Versprechen, in Zukunft einmal ein konkreter Wert zu sein"), mit dem beliebig weitergerechnet werden kann, solange sein Wert nicht benötigt wird (es kann beispielsweise als "black box" in Datenstrukturen eingebaut sowie als Funktionsargument oder -ergebnis übergeben werden). Parallel dazu (und unabhängig davon) wird mit der Auswertung der *expr* begonnen.

Nach außen hin unterscheidet sich ein Future nicht von einem normalen Datenobjekt: Wird auf seinen Wert zugegriffen, bevor dieser berechnet ist, wird dieser Zugriff automatisch verzögert, bis der Wert fertig berechnet ist. Dann wird das Future durch den aktuellen Wert ersetzt und verhält sich auch wie dieser.

`(delay` *expr*`)` verhält sich genauso, allerdings wird mit der Berechnung von *expr* erst dann begonnen, wenn der Wert wirklich benötigt wird ("lazy evaluation").

In Spur Lisp gibt es — unabhängig von den dort ebenfalls vorhandenen expliziten parallelen Prozessen — ebenfalls Futures, ebenso in Qlisp. Dort geht man noch weiter: Zur leichteren Realisierung von And- und Or-Parallelismus kann ein Future in Qlisp mehrere Prozesse enthalten. Beim Or-Parallelismus wird das erste positive Resultat als Wert des Futures betrachtet, und alle anderen Prozesse dieses Futures werden abgebrochen. Beim And-Parallelismus werden die Werte der Teilprozesse mit einer beliebigen Funktion zum Wert des Futures kombiniert. Weiters verwenden viele Konstrukte von Qlisp implizit Futures (siehe `qlet` und `qlambda`).

Bis jetzt gibt es von allen Sprachen, die Futures bieten, nur Implementierungen auf Maschinen mit gemeinsamem Speicher. Versuche auf Systemen mit verteiltem Speicher stehen noch aus, dürften aber sehr schwierig sein.

Das Hauptproblem bei Futures ist, daß *jede* Operation, die den Wert eines Operanden benötigt, zuerst prüfen muß, ob der Operand vielleicht ein Future ist. Das bewirkt einen signifikanten *Geschwindigkeitsnachteil* gegenüber Lisp-Implementierungen ohne Futures, vor allem auf konventioneller Hardware, *selbst für Programme, die rein sequentiell sind und keinerlei Futures verwenden.*

Mul-T, eine auf Futures basierende Parallelisierung des Scheme-Dialektes T, die u. a. auf dem Encore Multimax läuft, ist das momentan

wohl beste und effizienteste parallele Lisp, es basiert auf dem hochoptimierenden Orbit-Compiler. Hier betragen die Verluste für die Future-Tests mindestens 100 %. Selbst wenn der Compiler Datenfluß-Analysen durchführt und die Future-Tests an jenen Stellen, an denen auf Grund vorheriger Tests ohnehin nie ein Future auftreten kann, entfernt, bleibt eine Geschwindigkeitseinbuße von rund 65 %.

In manchen Systemen (z. B. in parallel PSL) verlegte man sich daher auf einen Kompromiß: Man läßt die Future-Tests in den einzelnen Operationen weg; diese dürfen daher nur mit tatsächlichen Werten (oder ausgewerteten Futures) aufgerufen werden, nicht aber mit noch in Auswertung befindlichen Futures. Überall, wo das passieren könnte, muß der Programmierer vorher *explizit* durch den Aufruf von `(touch` *expr*`)` veranlassen, daß gewartet wird, bis das Future *expr* fertig berechnet ist. Das Einfügen dieser `touch`es ist allerdings eine sehr unangenehme und fehleranfällige Methode, jedes vergessene `touch` hat fatale Folgen.

Parallele Datentypen:

In *Lisp für die Connection Machine gibt es spezielle Datentypen für Felder, die verteilt auf die einzelnen Prozessoren abgespeichert werden und deren Elemente parallel bearbeitet werden.

Linda Lisp:

Eine Kombination von Lisp mit Linda wird derzeit an mehreren Orten für unterschiedlichste Maschinentypen (u. a. Transputer) entwickelt.

Explizites Message Passing:

Lisp erweitert um Funktionen für Message Passing gibt es u. a. für die intel Hypercubes und für Transputer (laut Ankündigungen). Dabei wird so vorgegangen, daß auf jedem Knoten ein komplettes, eigenständiges Lisp-System (mit eigenem Gültigkeitsbereich für Namen und eigenem lokalen Speicher) arbeitet und Daten in externer Darstellung (d. h. in Textform) ähnlich wie bei einem Zugriff auf Files ausgetauscht werden.

Explizite gemeinsame Variablen:

Experimente in diese Richtung gibt es vor allem im Bereich der klassischen Supercomputer (C-Lisp, ZLisp).

Signals:

Sowohl in Qlisp als auch in Spur Lisp gibt es Funktionen, mit denen ein Prozeß eine Art Interrupt (*"signal"*, *"event"*) an einen anderen Prozeß schickt, dieser beginnt daraufhin sofort mit der Ausführung der für dieses Ereignis vorgesehenen Funktion.

Ein Problem, das allen parallelen Lisps gemeinsam ist, ist die schon im sequentiellen Fall komplizierte Semantik von statischen und dynamischen *Bindungen von Namen*: Wie werden welche Bindungen an neu kreierte Prozesse vererbt und was passiert, wenn der bindende Prozeß ein Scope verläßt oder endet, aber andere Prozesse noch die Bindungen benötigen? Weitere kritische Punkte sind destruktive Listenoperationen sowie `catch` und `throw` zwischen Prozessen.

Bisher hat noch keines der oben angeführten Systeme weite Verbreitung erlangt. Der Hauptgrund dürfte darin liegen, daß entweder das verwendete Parallel-Konzept nicht harmonisch zu Lisp paßt (Message Passing, gemeinsame Variablen), oder daß die Implementierung extrem ineffizient ist (Multilisp beispielsweise läuft auf dem gleichen Prozessor um Größenordnungen langsamer als konventionelles Lisp, und alle nicht-strikten Varianten haben mit ähnlichen Problemen zu kämpfen).

3.6.4 Linda

Linda ist eine Entwicklung von Prof. David Gelernter an der Universität Yale[32]. Es ist ein *sprachunabhängiges Modell für Parallelität und Kommunikation*, das gute Chancen hat, das Standard-Modell für high-level-Parallelität in prozeduralen Sprachen (bisher: C, C++, Fortran, Lisp, Modula-2, Postscript) zu werden.

Es behandelt Parallelität und Kommunikation gemeinsam, deckt einen weiten Bereich von Korngrößen ab, und ist einfach genug, um sowohl als theoretisches Modell mit klarer Semantik[33], als auch als praktisches Programmierwerkzeug mit effizienter Implementierung zu dienen[34].

Es ist relativ universell und so anschaulich, daß es auch ohne spezielles Wissen von Anwendern erfolgreich benutzt werden kann. Auch von der Mächtigkeit her ist es imstande, die meisten anderen Modelle (gemeinsame Variablen einschließlich atomic Update und Semaphoren, Message Passing, Streams etc.) zu simulieren, kontrollierter Nichtdeterminismus ist ebenfalls möglich.

Linda ist ein Modell für *explizite Parallelität*[35], aber es ist nicht so hardware-

[32] Linda ist nicht zu verwechseln mit den *“symmetric languages”*, die ebenfalls von Prof. Gelernter entwickelt wurden. Bei diesen Sprachen handelt es sich um ein Konzept, das auf der Vereinheitlichung von Daten- und Programmstruktur beruht und impliziten Parallelismus bietet. Es hat eine gewisse Verwandtschaft mit Datenflußkonzepten.

[33] Meines Wissens wurde allerdings bis jetzt kein Versuch einer formalen Semantikdefinition für Linda unternommen.

[34] Linda und seine Erweiterungen (z. B. mit mehreren, benannten Tuple Spaces) sind über ein einzelnes Programm hinaus verwendbar: Das Modell eignet sich auch zur Repräsentation und Implementierung von Betriebssystemen, Filespeichern, Netzwerken usw..

[35] Trotzdem benötigt Linda für eine wirklich effiziente Implementierung einen intelligenten Compiler, der in seiner Komplexität nahe an die mit Datenflußanalyse arbeitenden automa-

nahe wie gemeinsame Variablen oder Message Passing; es läßt sich auf einem weiten Spektrum von Hardware (sowohl Systeme mit gemeinsamem als auch mit verteiltem Speicher, aber auch Monoprozessoren oder Computer an Netzwerken) effizient implementieren. Es ist auch vom Betriebssystem weitgehend unabhängig.

Das Modell von Linda beruht auf dem *Tuple Space*, einem "Sack" (einem Multiset, d. h. einer Menge, in der auch identische Elemente in ihrer vollen Anzahl erhalten bleiben) von beliebig, aber endlich vielen *"Tupeln"* (d. h. endlichen Folgen von getypten Datenelementen)[36].

Dieser Tuple Space ist *logisch global* (kann aber durchaus auf verteiltem Speicher implementiert werden), auf ihn wird aber nicht mittels Adressierung oder Reihenfolge zugegriffen, sondern mittels *Pattern Matching*[37]: Zwei Tupel matchen, wenn die Länge ident ist, die Typen paarweise übereinstimmen, und auch die bereits vorhandenen Werte gleich sind; wo noch Variablen sind, erhalten diese den korrespondierenden Wert aus dem anderen Tupel zugewiesen.

Für den Zugriff auf den Tuple Space gibt es 6 primitive Operationen, die zur jeweiligen Sprache hinzugefügt werden:

`out`

Mit `out` wird ein Tupel, das die angegebenen Werte enthält, zum Tuple Space hinzugefügt, dann rechnet der aufrufende Prozeß normal weiter.

`eval`

`eval` fügt wie `out` ein Tupel zum Tuple Space dazu, nur mit dem Unterschied, daß die Argumentwerte nicht gleich innerhalb des aufrufenden Prozesses ausgewertet werden, sondern parallel dazu von eigens dafür geschaffenen, unabhängigen Prozessen. Der aufrufende Prozeß kann hier noch früher, nämlich sogar bevor die Werte des Tupels fertig berechnet sind, weiterrechnen[38].

Das ist der einzige Weg, in Linda *Parallelität* zu erzeugen. Tupel, die noch nicht fertig berechnete Werte enthalten, werden *Live Tupel* genannt (im Unterschied zu den normalen Datentupeln). Sie sind *unsichtbar*, nehmen also nicht am Pattern Matching teil. Erst wenn alle Werte fertig

tisch parallelisierenden Compiler herankommt.

[36] In manchen Varianten, die mehrere Tuple Spaces erlauben, ist `tuplespace` selbst wieder ein normaler Datentyp, der u. a. als Elementtyp in Tupeln auftreten kann, wodurch sich Tuple Spaces verschachteln lassen.

[37] In der Praxis analysiert der Compiler im Voraus alle Zugriffsoperationen und ersetzt das allgemeine Pattern Matching so weit wie möglich durch einfache, direkte Zugriffe oder Hashcoding, ohne daß sich die Semantik dadurch ändert.

[38] Dieses Konzept ist mit der parallelen Auswertung von Funktionsargumenten in Lisp und funktionalen Sprachen verwandt.

berechnet sind, verwandelt sich das betroffene Live Tupel in ein normales, sichtbares Datentupel, als ob es mit `out` erzeugt worden wäre.

Live Tupel und die dazugehörigen Prozesse sind in Linda black boxes: Linda interessiert sich nicht für deren Innenleben, sie sind anonym und unabhängig voneinander, ja sie wissen nicht einmal voneinander, sie können nur indirekt mittels Tuple Space miteinander kommunizieren.

`in`

`in` mit einer Liste von Werten und Variablen (einem *Pattern*) sucht im Tuple Space nach einem Tupel, das das gegebene Pattern matcht. Gibt es kein solches Tupel, wird der aufrufende Prozeß so lange verzögert, bis eines zum Tuple Space hinzugefügt wird. Gibt es mehrere, wird eines davon zufällig gewählt. Den Variablen werden dann die entsprechenden Werte aus dem Tupel zugewiesen, das Tupel wird aus dem Tuple Space entfernt, und der aufrufende Prozeß rechnet weiter.

`read`

`read` verhält sich wie `in`, bis auf den Unterschied, daß das gematchte Tupel im Tuple Space bleibt.

`inp` **und** `readp`

Diese beiden Primitive verhalten sich wie `in` und `read`, wenn ein passendes Tupel im Tuple Space existiert, ansonsten kehren sie anstatt zu verzögern sofort zurück. Sie geben zusätzlich einen boolschen Wert zurück: `true`, wenn ein Tupel gefunden wurde, und ansonsten `false`.

3.6.5 Objekt-orientierte Ansätze

Eine Parallelisierung von objektorientierten Sprachen ergibt sich auf einfache und natürliche Weise: Man betrachtet *jedes Objekt als eigenständigen Prozeß*, und die *Kommunikation ergibt sich automatisch durch das in objektorientierten Sprachen verwendete Austauschen von Nachrichten zwischen Objekten.*

Bei konventionellen objektorientierten Sprachen ist die Abarbeitung wegen der Art des Nachrichtenaustausches allerdings immer sequentiell, auch wenn viele Objekte gleichzeitig existieren. Damit wirklich Parallelität auftritt, gibt es zwei Möglichkeiten[39]:

[39] Daneben gibt es natürlich explizite konventionelle Parallelität, beispielsweise die Prozesse und Semaphoren in Smalltalk, und Möglichkeiten zur impliziten Parallelität, beispielsweise durch das gleichzeitige Auswerten der Argumente einer Nachricht, wenn sichergestellt werden kann, daß sich diese nicht durch Side-Effects beeinflussen.

Verzögerte Antworten:

Hier wird das *Senden einer Nachricht und das Empfangen der Antwort syntaktisch und zeitlich getrennt*, der Sender kann andere Aktionen ausführen (und auch mit anderen Objekten kommunizieren), während seine Nachricht bearbeitet wird[40].

Am saubersten ist dieses Konzept in Concurrent Smalltalk verwirklicht: Ähnlich den *"Futures"* in parallelen Lisp-Dialekten wird jede Nachricht sofort mit einem Objekt beantwortet, das als Platzhalter für die tatsächliche Antwort dient. Dieses Objekt reagiert nur auf eine Nachricht, nämlich `receive`: Es antwortet mit dem Objekt, das die Antwort auf die ursprüngliche Nachricht ist, und bewirkt wenn nötig eine Verzögerung, bis dieses Objekt fertig berechnet ist.

Objekte mit Innenleben:

Hier beginnt jedes Objekt automatisch ein Programm auszuführen, sobald es kreiert wird. Es kann daher *auch dann aktiv sein, wenn es nicht gerade eine eingegangene Nachricht bearbeitet*, beziehungsweise kann es nach Beantwortung einer Nachricht noch weitere Aktionen ausführen[41].

Der bedeutendste Vertreter ist POOL von Philips im Rahmen von Esprit 415.

Ein ähnliches, von vorneherein auf Parallelität ausgerichtetes Konzept liegt den *Actors*, einem parallelen, auf Objekten beruhenden Programmiermodell, zu Grunde. Hier erzeugt ein Objekt bei Bedarf eine neue, idente Kopie von sich selbst und ist deshalb imstande, mehrere Nachrichten gleichzeitig zu bearbeiten.

3.6.6 Quasiparallele Modelle

Neben Modellen, die für die Programmierung echt paralleler Hardware gedacht sind, gibt es auch zahlreiche Ansätze von Programmiersprachen für Anwendungen, die zwar *mehrere Prozesse enthalten*, aber *nur auf Monoprozessoren abgearbeitet werden*, d. h. wo zu einem Zeitpunkt nur jeweils ein Prozeß rechnet.

[40] Die Methode, *Funktionsaufruf und Erhalt des Ergebnisses syntaktisch und zeitlich zu trennen*, wurde auch schon zur *Parallelisierung konventioneller, prozeduraler Sprachen* mit Erfolg verwendet. Es handelt sich hier praktisch um eine eingeschränkte Form des Message Passing.

[41] Andererseits bedeutet dies in Abweichung von üblichen objektorientierten Sprachen, daß in jedem Objekt nun *explizit* programmiert werden muß, wann es auf welche Nachrichten wartet.

Solche Anwendungen umfassen beispielsweise Betriebssysteme; die bekannteste Programmiersprache in diesem Bereich ist wohl Modula-2 mit dem Konzept der *Koroutinen*[42].

Wir werden diesen Bereich hier nicht weiter behandeln.

3.6.7 Transaction Processing

Für die Implementierung von *Transaction Processing Systemen*, d. h. Systemen, die vielen Benutzern gleichzeitig den Zugriff auf eine gemeinsame Datenbasis gestatten (Flugbuchung, Geldtransfer, ...), wurden auch zahlreiche spezielle Programmiersprachen entwickelt.

Die Grundziele unterscheiden sich allerdings deutlich von denen der parallelen Programmierung: Der Schwerpunkt liegt auf *Ausfallsicherheit* und *Konsistenz verteilter Datenbestände*; der einzelne Benutzer sieht weiterhin nur ein rein sequentielles System.

Dieses Fachgebiet wird auch *"distributed programming"* genannt, wir wollen hier nicht näher darauf eingehen.

3.6.8 Betriebssystem-Konzepte

Erfreulicherweise gibt es auch auf dem Niveau der Betriebssysteme maschinenunabhängige Ansätze, deren Ziele praktische Verwendbarkeit und Kompatibilität mit bestehenden Standards ist:

NCS (Network Computing System):

Hier handelt es sich um eine Sammlung von hardware- und systemunabhängigen Funktionen und Protokollen zum verteilten Rechnen auf durch ein TCP/IP-basiertes Netz verbundenen Computern, ähnlich wie NFS verteilte Filesysteme auf solchen Netzen definiert.

Mach:

Mach ist ein Unix-kompatibles Betriebssystem, das primär auf sequentiellen Rechnern und MIMD-Systemen mit globalem Speicher läuft, sich aber auch auf MIMD-Systeme mit verteiltem Speicher erweitern läßt.

[42] Wie weit sich Ada mit dem ursprünglich vorgesehenen Konzept des *Rendevouz* in der Praxis für echt parallele Anwendungen eignet, oder ob es ebenfalls auf quasiparallele Programme beschränkt bleibt, entzieht sich meiner Kenntnis. Alle mir bekannten echt parallelen Ada-Applikationen verwendeten jedenfalls andere, zu Ada nachträglich hinzugefügte Parallelitätsfunktionen.

Mach ist eine Entwicklung der CMU (Carnegie Mellon University), es ist völlig frei von AT&T-Code und wird kostenlos an Forschungseinrichtungen abgegeben. Auch das Betriebssystem OSF-1 beruht auf Mach.

Das zentrale Konzept von Mach ist *shared virtual memory*, d. h. Vortäuschung von gemeinsamem Speicher durch Paging zwischen den einzelnen Prozessen und Prozessoren.

Chorus:

Chorus ist ein kommerziell entwickeltes Betriebssystem aus Frankreich speziell für MIMD-Systeme mit verteiltem Speicher (Transputersysteme, Hypercubes). Es ist voll kompatibel zu den gebräuchlichen Unix-Standards.

3.7 Maschinennahe Ansätze

Die Bezeichnung “maschinennahe Ansätze” ist hier willkürlich gewählt, wir wollen damit alle Modelle zusammenfassen, wo der Benutzer explizit mit parallelen Prozessen und deren Kommunikation zu tun hat, und wo die Kommunikation direkt die Hardwarestruktur widerspiegelt (z. B. Message Passing und Kommunikationskanäle oder gemeinsame Variablen und globaler Speicher)[43].

Im wesentlichen kann man bei diesen Modellen zwei Teilbereiche unabhängig voneinander untersuchen:

- **Das Erzeugen paralleler Prozesse.**
- **Die Kommunikation zwischen den Prozessen.**

Gelegentlich tritt auch noch die *Synchronisation* als eigenständiges Konzept hinzu; sie sollte aber implizit in den Konstrukten für Parallelität und Kommunikation enthalten sein.

Zwei Konzepte für eigenständige Synchronisation sind besonders verwerflich:

Interrupts:

Das vor allem in Unix unter dem Begriff `signal` gebräuchliche Konzept, daß ein Prozeß die momentane Arbeit eines anderen unterbrechen und ihn zur sofortigen Ausführung einer bestimmten Funktion veranlassen kann, führt zu besonders fehleranfälligen und schwer zu durchschauenden Programmen.

[43] Auf die Möglichkeiten der Simulation eines Modells auf der jeweils anderen Hardware werden wir noch eingehen.

Busy Waiting, Polling:

Hierbei wird eine Variable, ein Kommunikationskanal oder ein Prozeßzustand in einer Schleife ständig geprüft, bis der gewünschte Zustand eintritt. Dieses Konzept ist zwar semantisch harmlos, aber es kann nutzlos beliebig viel Rechenzeit verbrauchen.

3.7.1 Parallele Prozesse

Im wesentlichen gibt es hier zwei Ansätze:

Aufspalten des Programmablaufes:

Ein explizites syntaktisches Konstrukt im Programmtext (Musterbeispiel: Occam's `PAR`) spaltet den Programmablauf in mehrere gleichberechtigte, voneinander unabhängige Pfade auf.

Am Ende dieses Konstruktes wird automatisch gewartet, bis alle Zweige fertig berechnet sind, dann wird sequentiell weitergerechnet.

Diese Art von Parallelität spielt sich innerhalb des Rumpfes einer Funktion auf dem Niveau einzelner Statements ab und ist daher relativ feinkörnig.

Die auf diese Weise entstandenen Prozesse sind normalerweise anonym, d. h. es gibt keinen Namen und keine Referenz für sie, und sie wissen voneinander nichts.

Ähnliche Formen der Parallelität treten beispielsweise bei den parallelen Schleifenkonstrukten auf, wo die einzelnen Iterationen des Schleifenrumpfes parallel bearbeitet werden.

Explizites Kreieren neuer Prozesse:

Das geschieht meist durch den Aufruf einer Systemfunktion (`create` oder ähnliches[44]), wobei der aufrufende Prozeß normalerweise sofort weiterrechnen kann, und der neu geschaffene Prozeß unabhängig davon seine Arbeit beginnt und beendet, wodurch sich eine baumförmige Prozeßhierarchie ergibt.

Hier gibt es keine implizite Synchronisation, der kreierende Prozeß sieht nichts vom Fortgang seines Sohnes, und es kann sowohl der Vater vor dem Sohn enden als auch umgekehrt. Es gibt allerdings oft explizite Synchronisationsfunktionen, die den aufrufenden Prozeß bis zur Beendigung des angegebenen Prozesses verzögern.

---

[44] Das in der Praxis derzeit verbreitetste Parallelkonstrukt, nämlich `fork` in Unix, ist eine besonders unleserliche Variante dieses Konzeptes.

Normalerweise wird dieser prozeßkreierenden Funktion der Name jener Funktion, die vom neu kreierten Prozeß ausgeführt werden soll, mitgegeben, die resultierende Parallelität ist daher eher grobkörnig, eine Funktion ist die kleinste Einheit.

Als Ergebnis liefert die prozeßkreierende Funktion eine Referenz auf den neu kreierten Prozeß, mit der dieser in Zukunft beispielsweise für Kommunikationszwecke, zur Synchronisation oder zwecks Abbruch angesprochen werden kann.

Eine weitere Frage ist, wie weit Prozesse nur abstrakte Gebilde sind beziehungsweise wie weit sich der Programmierer um deren physikalische Eigenschaften kümmern muß:

Plazierung von Prozessen:

Die Frage ist, ob der Programmierer beim Kreieren eines Prozesses einen *Prozessor* angeben muß, auf dem dieser Prozeß ausgeführt werden soll, oder ob das System in der Lage ist, anhand der derzeitigen Systemlast oder sonstiger Heuristiken automatisch eine günstige Plazierung vorzunehmen.

Geht es automatisch, ist weiters die Frage, ob Prozesse ein für allemal fest plaziert werden, oder ob sie während der Abarbeitung zwecks Load Balancing auf *andere Prozessoren verlagert* werden können.

Bei händischer Plazierung gibt es zwei grundsätzliche Möglichkeiten, den zu verwendenden Prozessor anzugeben:

Mit absoluten Prozessoridentifikationen:

Hier ist jeder Prozessor im System mit einer Kennung versehen, und bei jedem neu zu kreierenden Prozeß wird die Kennung des zu verwendenden Prozessors angegeben. Occam verwendet beispielsweise diese Methode, die Prozessoren sind einfach statisch durchnumeriert.

Mit relativen Ortsangaben:

Hier wird der für den neuen Prozeß zu verwendende Prozessor angegeben durch den Pfad, auf dem er relativ zum Standort und zur Orientierung (Blickrichtung) des kreierenden Prozesses zu erreichen ist (ähnlich wie bei Logo's *"Turtle Graphics"*).

Dieses Verfahren wird beispielsweise in Concurrent Prolog verwendet, es ist für dynamische oder rekursive Parallelität sehr viel besser geeignet.

Speicherverwaltung:

Viele heutige Systeme (vor allem am Transputer) erfordern es, jedem Prozeß bei seiner Erzeugung *bestimmte Speicherbereiche für Stack, Heap usw. fix zuzuordnen*, die Prozesse können bei Bedarf nicht dynamisch Speicher nachfordern. Das ist eine wesentliche Einschränkung, die sich vor allem auf die Speicherauslastung negativ auswirkt.

Ein weiteres Kriterium bei der Analyse paralleler Prozesse ist die Frage, wieviel zwei miteinander verwandte Prozesse gemeinsam haben beziehungsweise was neugeschaffene Prozesse von ihrem Vater erben.

Das betrifft einerseits den *Speicher* (gemeinsame Variablen, gemeinsamer Heap mit global gültigen Pointern), andererseits *Zugriffsrechte und Schutzattribute* für Files und Devices, aber auch für die Kommunikation mit anderen Prozessen, sowie Prioritäten und Interruptfunktionen und -masken.

Da diese Dinge alle relativ aufwendig zu handhaben sind (kopieren von Speicherbereichen, modifizieren von Speicherverwaltungstabellen und Statusregistern usw.) und daher das Kreieren eines neuen Prozesses, aber auch einen simplen Prozeßwechsel sehr langsam machen, bietet man heute in vielen Systemen (MACH sowie einige Softwareprodukte für den Transputer) zwei Schichten an:

Tasks:

Das sind komplette, selbstständige Einheiten mit *eigenen Adressräumen* und *allen Schutzmechanismen*. Sie sind daher aufwendig (*"heavyweight processes"*) und sollten nur für grobkörnige Parallelität verwendet werden.

Threads:

Hier handelt es sich um einzelne, parallele Abläufe innerhalb eines einzigen Tasks. Sie enthalten im wesentlichen *nur einen eigenen Programmzeiger und einen eigenen Stack*, alles andere haben sie untereinander gemeinsam und von dem Task, zu dem sie gehören, geerbt. Da sie auch einen gemeinsamen Adressraum benutzen, müssen auf einem System mit verteiltem Speicher zwangsweise alle Threads eines Tasks auf demselben Prozessor angesiedelt sein.

Threads sind sehr einfach und schnell zu behandeln (*"lightweight processes"*) und daher auch für feinkörnigere Parallelität geeignet[45].

Ein wesentliches Problem bei expliziten parallelen Prozessen ist die Möglichkeit, von einem Prozeß direkt in den Ablauf eines anderen einzugreifen, um ihn beispielsweise zu verzögern oder abzubrechen. Hier gilt dasselbe wie in der konventionellen Programmierung für die Abhängigkeiten zwischen Funktionen oder Modulen: Je weniger Prozesse voneinander wissen, und je weniger

[45] In der Praxis beträgt der Geschwindigkeitsunterschied oft 1 zu 1000 und mehr.

sie sich gegenseitig beeinflussen können, umso besser ist es aus programmiertechnischer Sicht und auch für die Festlegung einer formalen Semantik. Der Idealfall wäre hier wie bei Funktionen, daß ein Prozeß nur bei seiner Erschaffung und seiner Termination mit der Außenwelt kommuniziert und sonst völlig isoliert arbeitet.

Ein Musterbeispiel für ein unsicheres Konstrukt ist das in vielen Sprachen vorhandene *"Killen" von Prozessen*, d. h. das Abbrechen eines Prozesses von außen ohne sein Wissen. Es ist semantisch praktisch nicht zu beherrschen:

- Was passiert, wenn dieser Prozeß gerade in *Kommunikationen* oder *Synchronisationen* verwickelt ist, oder wenn andere Prozesse auf ihn warten (der garantierte Deadlock)?
- Was geschieht mit den von diesem Prozeß belegten *Betriebsmitteln* (Files, Speicher, usw.)?
- Was geschieht mit den von diesem Prozeß geschaffenen *Subprozessen*?
- Wie verhindert man das unbemerkte Abhandenkommen von nur diesem Prozeß bekannten *Möglichkeiten und Rechten* (z. B. Pointer, Zugriffsschlüssel, Identifikationen anderer Prozesse, ...)?

3.7.2 Kommunikation

Hier gibt es zwei grundsätzliche Ansätze, die unmittelbar aus der Hardware abgeleitet sind:

- **Gemeinsame Variablen** (auf Systemen mit globalem Speicher)
- **Message Passing** (auf Systemen mit verteiltem Speicher)

Beiden Modellen gemeinsam ist das praktisch höchst unbefriedigend gelöste Problem der gemeinsamen dynamische Datenstrukturen mit Pointern und Garbage Collection.

Manche Sprachen bieten allerdings auch beide Modelle. Ein Musterbeispiel ist Chill, die Standardsprache für die Programmierung von Telekommunikationssystemen:

`region` bietet sicheren Zugriff auf gemeinsame Datenstrukturen nach dem Monitor-Prinzip.

`signal` bietet direktes, asynchrones Message Passing zwischen zwei Prozessen, die mit ihrer Identifikation angegeben werden.

`buffer` sind Mailboxen beliebiger Größe, auf die beliebig viele Prozesse sendend und empfangend zugreifen können und die daher Nichtdeterminismus zur Folge haben.

`start` kreiert explizit neue Prozesse.

`event` erlaubt es, Prozeß-Warteschlangen explizit zu manipulieren.

Ada bietet mit seinem *Rendezvouz* eine Art Mittelding, das am ehesten mit den Remote Procedure Calls verglichen werden kann[46]. Es ist relativ high-level und maschinenunabhängig und auch (durch seine vielen Optionen) sehr universell. Nichtdeterminismus läßt sich damit ebenfalls ausdrücken. Andererseits verwendet es strikte Synchronisation, es ist daher sicherer in der Anwendung als viele andere Konstrukte und auch einer formalen Semantik leichter zugänglich.

Gemeinsame Variablen

Gemeinsame Variablen an sich sind einfach zu handhaben: Sie werden irgendwie speziell deklariert[47] (`shared`, `common`, `public`, ...) und verhalten sich sonst syntaktisch und meist auch semantisch wie ganz normale Variablen.

Das Problem besteht darin, den Zugriff auf gemeinsame Variablen so zu regeln, daß einem Prozeß bei Bedarf das *kontrollierte Lesen, Verändern und Zurückschreiben* einer dieser Variablen möglich ist, ohne daß andere Prozesse in der Zwischenzeit ebenfalls unbemerkt den Wert dieser Variablen modifizieren[48].

Besonders wichtig ist das bei *zusammengesetzten Datenstrukturen* (Queues, Listen usw.), wo mehrere Komponenten geändert werden müssen, um von einem konsistenten Wert zu einem anderen zu gelangen[49].

---

[46] Daneben gibt es in Ada auch niedrige Primitive für gemeinsame Variablen.

[47] Es gibt auch Sprachen, bei denen die Variablen per default gemeinsam sind und private Variablen speziell deklariert werden müssen.

[48] Wie man sich leicht überlegen kann, führt die zeitliche Verzahnung zweier Änderungen einer Variablen dazu, daß am Ende ein falscher Wert in dieser Variablen gespeichert ist: Nehmen wir an, daß zwei Prozesse unabhängig voneinander die gleiche Variable um 1 erhöhen wollen. Wenn der zweite Prozeß die Variable liest, bevor der erste Prozeß den neuen Wert abspeichert, speichern beide den um 1 erhöhten alten Wert ab. Die Variable enthält somit einen falschen Wert; der korrekte wäre der alte Wert plus 2.

[49] Ein Beispiel ist das Anhängen eines Elementes an eine verkettete Liste, wobei sowohl der Zeiger auf das Listenende als auch der Folgezeiger im bisherigen letzten Element geändert werden müssen. Werden diese Zeiger von zwei Prozessen überlappend und unabhängig voneinander manipuliert, kann das zu einer inkonsistenten Liste führen (d. h. der Zeiger auf das Listenende zeigt nicht auf das tatsächliche Ende der Liste, oder die Liste hat ein Loch), oder das Ergebnis ist zwar eine korrekte Liste, aber eines der beiden anzuhängenden Elemente ist verloren gegangen.

Es sind also Konstrukte vorzusehen, die ein *“Atomic Update”*, also eine *unteilbare Änderung*, garantieren: *Wenn ein Prozeß eine gemeinsame Variable oder Datenstruktur ändern will, muß allen anderen Prozessen vom ersten Lesen bis zum letzten Schreiben jeder Zugriff auf diese Variable oder Datenstruktur verwehrt werden.*

Hier gibt es zahlreiche Ansätze (um nicht zu sagen verwirrend viele), die größtenteils ähnliche Mächtigkeit bieten (d. h. ein Modell als Grundoperation reicht normalerweise aus, um die anderen darauf aufbauend zu implementieren), aber bisher wurde noch keines als Ideal oder Standard anerkannt. Die wichtigsten sind (geordnet nach absteigender Hardwarenähe):

Explizite Locks:

Hier wird durch *spezielle Maschinenbefehle* für bestimmte Bereiche im Programm (normalerweise nur eine oder einige wenige Instruktionen lang) ein *exklusives Zugriffsrecht auf den gesamten Speicher* angefordert.

Diese Befehle aktivieren unmittelbar *spezielle Hardwareeinrichtungen*, die die Exklusivität sicherstellen bzw. einen Prozeß (genauer gesagt den betroffenen Prozessor), der ein Lock anfordert, so lange hardwaremäßig anhalten, bis alle anderen den Zugriff freigegeben haben. Da es sich hier um *busy waiting* handelt (d. h. der blockierte Prozessor kann in der Zwischenzeit keine anderen Prozesse ausführen), sind Locks nur für sehr feinkörnige Operationen sinnvoll; für zusammengesetzte Datenstrukturen sind sie ungeeignet.

Read/modify/write-Befehle:

Diese Befehle sind eine Alternative zu expliziten Locks: Anstatt beliebige Befehle mit expliziten Locks zu schützen gibt es hier einen oder einige wenige Befehle, die grundsätzlich *atomar*, d. h. ohne daß andere Prozesse gleichzeitig auf den Speicher zugreifen können, ausgeführt werden. Die wichtigsten Beispiele sind `test-and-set` (lade den alten Wert einer Variable und speichere einen neuen) sowie `fetch-and-add` (lade den alten Wert und speichere den alten Wert plus einem Inkrement).

Diese Befehle aktivieren die gleichen Hardwaremechanismen wie explizite Locks.

Semaphore:

Semaphore sind spezielle *Zählvariablen* verbunden mit einer *Prozeß-Warteschlange*, auf denen nach einmaliger Initialisierung (meist mit 0 oder 1) nur die Operationen ‘erhöhe’ und ‘erniedrige’ erlaubt sind, wobei der Wert nicht unter 0 sinken darf.

Versucht ein Prozeß, eine Semaphore mit dem Wert 0 zu erniedrigen, bleibt der Wert 0, und dieser Prozeß wird in die Warteschlange eingereiht

und deaktiviert. Ist der Wert der Semaphore größer 0, wird er erniedrigt, und der Prozeß kann sofort weiterarbeiten.

Erhöht ein Prozeß die Semaphore, werden zuerst entsprechend viele auf diese Semaphore wartende Prozesse wieder aktiviert, und erst wenn kein Prozeß mehr auf die Semaphore wartet, wird ihr Wert tatsächlich erhöht.

Semaphore sind das einfachste Konstrukt auf höherer Ebene, das aufbauend auf einem der beiden vorigen Primitiva in Software implementiert wird.

Kritische Regionen:

Kritische Regionen sind die wichtigste *Anwendung von Semaphoren*: Hier wird wie beim Lock (aber auf grobkörnigerem Niveau) mit Hilfe von Semaphoren der *exklusive Zugriff vor allem auf einzelne, gemeinsame Datenstrukturen für ein bestimmtes Programmstück* sichergestellt. Das geschieht entweder explizit durch Semaphore (die auf 1 initialisierte Semaphore wird erniedrigt, dann erfolgt der Zugriff auf die Datenstruktur, und zuletzt wird die Semaphore wieder erhöht), oder durch ein eigenes Sprachkonstrukt, wobei normalerweise jede kritische Region benannt wird und zusammengehörige kritische Regionen den gleichen Namen tragen.

Die kritischen Regionen für verschiedene globale Datenstrukturen sind unabhängig voneinander; jede Datenstruktur hat ihre eigene Semaphore. Von den zu einer Datenstruktur gehörenden kritischen Regionen kann zu jedem Zeitpunkt nur eine betreten werden, aber es können sich durchaus (im Unterschied zu Locks) mehrere Prozesse gleichzeitig in kritischen Regionen verschiedener Datenstrukturen befinden.

Monitore:

Auch hier geht es um den Schutz von Datenstrukturen. Diese werden als *abstrakte Datentypen* implementiert, sie sind also nur über Zugriffsfunktionen ansprechbar. Im Unterschied zu normalen abstrakten Datentypen stellt ein Monitor aber auch noch sicher, daß zu jedem Zeitpunkt *nur eine Zugriffsfunktion aktiv* ist; weitere Aufrufe werden verzögert, bis vorhergehende beendet sind[50].

Der wichtigste Kritikpunkt an gemeinsamen Variablen ist, daß sie zu schwer durchschaubaren, fehleranfälligen, nicht nachvollziehbaren Programmen führen: Jeder Prozeß kann anonym zu jedem Zeitpunkt mit jeder Variable beliebiges anstellen, ohne daß andere Prozesse darüber informiert werden. Umgekehrt ist es nicht möglich, anhand des Programmtextes für einen beliebigen

[50] Die in der Praxis bedeutendste parallele Programmiersprache mit gemeinsamen Variablen, die um 1977 bei Xerox für die Systemprogrammierung entwickelte Sprache Mesa, verwendet ebenfalls Monitore.

Variablenzugriff dessen Vorgeschichte (d. h. wer zuletzt diese Variable wie modifiziert hat) festzustellen. Dementsprechend schwierig ist es auch, eine formale Semantik anzugeben. Fairness ist ebenfalls kaum zu garantieren.

Message Passing

Das Prinzip des Message Passing besteht darin, daß *mit eigenen Befehlen (“senden” und “empfangen”) eine Nachricht explizit von einem Prozeß zu einem anderen übertragen* wird.

Auch beim Message Passing gibt es viele Varianten, die sich in einigen Punkten unterscheiden:

Art der Synchronisation:

Hier ist das eine Extrem die *ungepufferte Kommunikation mit strikter Synchronisation*: Der Sender einer Nachricht muß warten, bis der Empfänger die Nachricht vollständig übernommen hat (wie in Occam).

Das andere Extrem ist die *asynchrone Kommunikation mit unbegrenztem Puffer*: Der Sender kann nach dem Senden einer Nachricht sofort weiterarbeiten und auch beliebig viele Nachrichten absenden, unabhängig davon, ob und wie schnell jemand diese Nachrichten konsumiert[51].

In der Praxis ergibt sich oft ein Mittelding: Der Sender darf dem Empfänger vorauseilen, aber nur um eine bestimmte Anzahl von Nachrichten, dann wird er verzögert. Das rührt daher, daß die Puffer für noch nicht empfangene Nachrichten üblicherweise nur endliche Größe haben.

Bei jeder gepufferten Kommunikation stellt sich die Frage, ob das Kommunikationssystem stets *die Reihenfolge der Nachrichten erhält*, oder ob die Nachrichten auch in einer anderen als der Sendereihenfolge beim Empfänger eintreffen können.

Art der Adressierung:

Hier geht es darum, anzugeben, wohin die Nachrichten geschickt bzw. woher sie empfangen werden:

Adressierung mittels Prozeßidentifikation:

Hier gibt der Sender den Prozeß, der die Nachricht empfangen soll, direkt an, und ebenso spezifiziert der Empfänger im Normalfall, von welchem Prozeß er auf eine Nachricht wartet[52].

---

[51] Hier können sich natürlich Probleme mit Speichergröße und Fairness ergeben, ein eifriger Prozeß kann den Prozessor monopolisieren und den Speicher überfluten.

[52] Eine ebenfalls vorkommende Variante besteht darin, daß der Empfänger keinen Absender angibt und jede an ihn gerichtete Nachricht unabhängig von Sender akzeptiert, und dann

Diese Methode impliziert, daß es zwischen je zwei Prozessen nur eine logische Verbindung geben kann.

Kanäle, Ports, Queues, Mailboxes usw.:

Hier gibt es *eigene, benannte Kommunikationswege*, an die die Nachrichten geschickt und von denen sie empfangen werden; die kommunizierenden Prozesse wissen nicht unmittelbar voneinander, sondern kennen nur mehr diese Nachrichtenträger.

Eine wichtige Frage ist hier, ob diese Verbindungen *statisch* im Programm deklariert sein müssen (wie z. B. in Occam), oder ob sie *zur Laufzeit* nach Belieben eingerichtet und aufgelöst werden können.

Eine weitere Frage ist, ob jeder dieser Nachrichtenträger genau einen Sender und einen Empfänger miteinander verbindet, oder ob *mehrere Prozesse* an ihn Nachrichten senden bzw. von ihm Nachrichten empfangen können (wodurch sich *Nichtdeterminismus* ergibt und Fragen der Fairness usw. auftauchen).

Getagte Nachrichten:

Hier wird jede Nachricht beim Senden mit einem Mascherl versehen (*"Tag"*, *"Type"*), und der Empfänger gibt an, auf welche Art von Nachrichten er gerade wartet. Dieser Mechanismus wird oft in Kombination mit einer der vorhergehenden Methoden verwendet, um von einem Partner oder Kanal bestimmte Nachrichten auszuwählen und andere zurückzuweisen.

Darüber hinaus gibt es manchmal auch noch Broadcast- oder Multicast-Möglichkeiten, bei denen eine Nachricht an alle oder eine bestimmte Gruppe von Prozessen gleichzeitig verschickt wird.

Art der Datenübertragung:

Hier ist einerseits die Frage, ob die Daten wirklich übertragen oder nur "by reference" weitergegeben werden (d. h. ob Modifikationen durch den Empfänger Rückwirkungen auf den Sender haben können)[53], und andererseits geht es darum, ob die übertragenen Daten irgendeiner Typprüfung oder einem Protokoll unterliegen, und welche Arten von Daten überhaupt übertragen werden können (kritisch sind dynamische Datenstrukturen und Pointer sowie Files, aber auch Identifikationen von Prozessen oder Kanälen).

---

anhand der gleichzeitig empfangenen Identifikation des Senders entscheidet, was mit der Nachricht geschehen soll.

[53] Bei Systemen mit globalem Speicher bildet die *"copy on write"*-Methode den optimalen Mittelweg, z. B. in Mach.

Art des Nichtdeterminismus:

Praktisch jedes Message Passing Modell enthält eine Form von Nichtdeterminismus[54], wobei es verschiedene Möglichkeiten gibt, Nichtdeterminismus einzuführen:

- Es gibt Kommunikationskanäle, die von mehr als einem Prozeß beschickt oder gelesen werden können (impliziter Nichtdeterminismus durch die mehr oder weniger zufällige Reihung der Nachrichten).
- Es gibt Befehle, die explizit prüfen, ob ein Kanal bereit zur Kommunikation ist (oder eine Mailbox Daten enthält usw.), oder die nach einer gewissen Zeit einen erfolglosen Kommunikationsversuch abbrechen, und es dem Programmierer ermöglichen, abhängig davon verschiedene Programmpfade einzuschlagen.
- Es gibt Befehle, die gleichzeitig mehrere Kommunikationspartner oder Kanäle anzusprechen versuchen und daraus einen einzigen, der willig zur Kommunikation ist, auswählen (entweder zufällig oder abhängig von der zeitlichen Reihenfolge der Kommunikationswilligkeit). Ein Beispiel ist das `ALT` in OCCAM.

Auch hier stellen sich mehrere Fragen:

Ist der Nichtdeterminismus fair?

Fairness bedeutet, daß ein Prozeß, der an einer nichtdeterministischen Kommunikation beteiligt und auch tatsächlich bereit zur Kommunikation ist, *nicht beliebig lange übergangen* werden kann, sondern tatsächlich einmal ausgewählt wird (das impliziert, daß auch ein besonders eifriger Prozeß nicht die Kommunikation monopolisieren kann). Im besten Fall sollten alle Prozesse gleich große Chancen haben.

Gibt es Prioritäten?

Das bedeutet, daß vom Benutzer in seinem Programm festgelegt werden kann, welche Prozesse bei nichtdeterministischer Wahl zuerst oder bevorzugt berücksichtigt werden sollen.

Im Unterschied zu gemeinsamen Variablen sind beim Message Passing jene Punkte in einem Programm, wo eine Kommunikation stattfindet, aus dem

[54] Ist dies nicht der Fall, d. h. gibt es nur Sende- und Empfangskonstrukte in ihrer reinen Form und keine andere Kommunikation zwischen Prozessen, so ist die Sprache deterministisch, d. h. Verhalten und Ergebnis eines Programmes hängen nicht vom zeitlichen Ablauf der Prozesse und ihrer relativen Geschwindigkeit zueinander ab.

Programmtext klar zu ermitteln. Eine Nachricht wird genau einmal abgesendet und einmal empfangen; jede Kommunikation verläuft von Punkt zu Punkt in eine Richtung.

Weiters müssen beide Partner aktiv an einer Kommunikation beteiligt sein; es ist nicht möglich, Daten eines Prozesses ohne sein Wissen zu modifizieren. Daher ist bei jeder Kommunikation feststellbar, welche Prozesse involviert sein können. Das erleichtert sowohl das visuelle Überprüfen und Nachvollziehen von Programmen als auch die Definition einer formalen Semantik und die Verifikation von Programmen.

Obwohl das Konzept des Message Passing theoretisch genauso mächtig ist wie das der gemeinsamen Variablen, wird es in der Praxis oft als restriktiver empfunden: Es gibt viele Anwendungen, die sich mit Message Passing nur mit großem Aufwand oder großem Effizienzverlust parallelisieren lassen. Das liegt auch daran, daß jede einzelne Kommunikation beim Message Passing sehr viel mehr Overhead verursacht (ein Systemaufruf im Vergleich zu einem Variablenzugriff), und Message Passing daher nur für grobkörnige Parallelität sinnvoll ist.

Es ist kein Problem, Message Passing ohne Effizienzverlust auf der Basis von gemeinsamen Variablen zu implementieren, es eignet sich also gleichermaßen für Systeme mit verteiltem oder globalem Speicher und wird auch in der Praxis auf beiden Systemtypen verwendet.

Der umgekehrte Weg, nämlich die Realisierung gemeinsamer Variablen und der dazu gehörenden Schutzmechanismen auf einem System, das nur Message Passing bietet (d. h. einem System mit verteiltem Speicher), ist zwar theoretisch in vollem Umfang möglich, aber ungleich schwieriger und ineffizienter:

- Ein Weg besteht darin, einen Prozeß mit der Verwaltung der gemeinsamen Variablen zu betrauen. Alle anderen Prozesse müssen dann alle Lese- und Schreiboperationen, aber auch Anforderungen von Locks usw., explizit mittels eigener Messages durchführen.

- Eine andere Möglichkeit, die u. a. im Betriebssystem MACH verfolgt wird, ist die Ausnutzung des Konzeptes des virtuellen Speichers: Der Speicherbereich, der die gemeinsamen Variablen enthält, wird bei Bedarf (d. h. wenn ein Zugriff erfolgt) vom Betriebssystem seitenweise mittels Message Passing auf den jeweiligen Knoten kopiert. Beschreibt ein Knoten eine global bekannte Seite, werden alle Kopien als ungültig markiert und bei Bedarf erneut kopiert.

  Dieses Konzept ist für das Anwenderprogramm transparent und verbirgt die tatsächliche Maschinenarchitektur, und es ist sogar in vielen Fällen hinreichend effizient, aber es braucht entsprechende Unterstützung durch das Betriebssystem und die Hardware.