Lehrstuhl für Informatik
Technische Universität München

Forschungs- und Lehreinheit Informatik VIII
Rechnerkommunikation, Maschinelle Deduktion
Prof. Dr. Eike Jessen

Vorlesung SS 2000

Parallelrechner

(Rechnerarchitektur II)

Version: 19. Oktober 2000

Satz und Bildgestaltung: Veit Lorenz

Inhaltsverzeichnis

Zur Einführung:

Ziele der Vorlesung

Inhalt

Literatur zur Gesamtvorlesung

Ziele der Vorlesung

Einführung in die Struktur und Arbeitsweise von Parallelrechnern, d.h. Rechnern, die an der HW/SW-Schnittstelle Parallelität vorzugeben gestatten, mit dem Ziel, die Grundlagen für

- Programmierung
- Bewertung und Auswahl
- Entwurf

von Parallelrechnern zu vermitteln, Verständnis für die Anforderungen an parallele Rechenverfahren zu erwecken, und die Zuhörer mit wichtigen, insbesondere neuen Parallelrechnern bekannt zu machen.

Literatur

Bücher:

- K. Waldschmidt (Hrsg.): Parallelrechner: Architekturen - Systeme - Werkzeuge, Teubner, 1995
  zur ganzen Vorlesung und darüber hinaus (Programmierung, Leistungsbewertung, Werkzeuge ...)
- T. Ungerer: Parallelrechner und parallele Programmierung, Spektrum, 1997
  zur ganzen Vorlesung, insbesondere auch zur Software
- W. Giloi: Rechnerarchitektur, Springer-Lehrbuch, 1993
  zur ganzen Vorlesung, mit anderen Schwerpunkten
- A. Bode, W. Händler: Rechnerarchitektur II, Springer Verlag, 1983
  zu Kap. 1, 2
- A. Bode (Hrsg.): RISC-Architekturen, BI Wissenschaftsverlag, Reihe Informatik, Bd. 60, 1992
  zu Kap. 4 (impliziter Parallelismus, Parallelismus in Mikroprozessoren), Programmierung

Bücher (englisch):

- K. Hwang, Z. Xu: Scalable Parallel Computing: Technology, Architecture, Programming, McGraw Hill, 1998
  zur ganzen Vorlesung und weit darüber hinaus (alle Aspekte der Parallelverarbeitung); empfehlenswert, bes. für weitergehende Beschäftigung mit dem Gebiet

Literatur zur Gesamtvorlesung

Die nachfolgende Vorlesung benutzt insbesondere folgende Quellen:

[Hellwagner H 98]: Hellwagner, H.: Parallelrechner, Vorlesungsskript SS 98, Universität Klagenfurt und TUM.

[Hwang Xu 98]: Hwang, K., Xu, Z.: Scalable Parallel Computing, Mc; Graw-Hill, Boston 1998.

[Jessen E 99]: Jessen, E.: Leistung von Parallelrechnern, Vorlesungsskript WS 98/99, TUM.

Aus [Hellwagner H 98] sind die nachfolgenden die Seiten Literatur entnommen. [Hwang Xu98] enthält 49 Seiten Literatur, darunter sehr viele Verweise auf WWW-Quellen.

- D. Culler, J. P. Singh, A. Gupta: Parallel Computer Architecture: A Hardware/Software Approach (Entwurf)
  http://http.cs.berkeley.edu/~culler/book.alpha/index.html
  sehr detaillierte Behandlung moderner Multiprozessoren, insbes. solcher mit gemeinsamem und verteiltem gemeinsamen Speicher
- J. Hennessy, D. Patterson: Computer Architecture: A Quantitative Approach, Morgan Kaufmann, 2nd Edition, 1996
  das beste Buch zur Rechnerarchitektur; enthält umfangreiches Kapitel zu Multiprozessoren
- D. Patterson, J. Hennessy: Computer Organization and Design: The Hardware/Software Interface, Morgan Kaufmann, 1994
  nicht spezifisch für Parallelrechner, aber sehr empfehlenswert für Verständnis eines Gesamtsystems
- G. Almasi, A. Gottlieb: Highly Parallel Computing, Benjamin/Cummings, 2nd Edition, 1994
  zu allen Aspekten der Parallelverarbeitung, mit anderen Schwerpunkten
- A. Goscinski: Distributed Operating System: The Logical Design, Addison-Wesley, 1991
  sehr gutes Buch zu verteilten Betriebssystemen
- W. Schroeder-Preikschat: The Logical Design of Parallel Operating Systems, 1995
  zu parallelen Betriebssystemen, mit Schwerpunkt PEACE
- A. Bode, M. DalCin: Parallel Computer Architectures: Theory, Hardware, Soffware, Applications, Springer Verlag, LNCS Bd. 732, 1993
  weiterführende Themen, u.a. aus dem SFB 342 an der TUM ('Werkzeuge und Methoden für die Nutzung paralleler Rechnerarchitekturen')

Tagungsbände:

- EuroPar (European Conference on Parallel Processing)
- HPCN Europe (High Performance Computing & Networking)
- ICPP (Int'l. Conference on Parallel Processing)
- Supercomputing
- ICS (Int'l. Conference on Supercomputing)
- ISCA (Int'l. Symposium on Computer Architecture)
- HPCA (High Performance Computer Architecture)
- ...

Zeitschriffen:

- IEEE Concurrency
- IEEE Transactions on Parallel and Distributed Systems
- IEEE COMPUTER
- Parallel Computing
- ...

1 Parallelität und Parallelrechner

1.1 Begriffe und Ziele der Parallelrechnerentwicklung

Parallelrechner (parallel computer): Rechner, der *Auftragssysteme* ausführt, die (in sich) nebenläufig definiert sind, und die Nebenläufigkeit für kollaterale Ausführung des Auftragssystems nutzt. (1)

Auftragssystem (task system): Menge von Aufträgen und Präzedenzrelation (vgl. Kap 1.5) zur Erfüllung *einer* Aufgabe. (2)

Gegenbeispiel: Menge der Aufträge, die gleichzeitig für viele Benutzer in einem Rechnernetz erledigt werden.

Nebenläufig (concurrent): sind zwei Aufträge, die nicht präzedent sind, d.h. es muß nicht einer fertiggestellt sein, bevor der andere begonnen wird. (3)

Kollateral (simultaneous): werden zwei Aufträge ausgeführt, wenn zu wenigstens einem Zeitpunkt beide begonnen, aber keiner beendet ist. (4)

Die Bezeichnung *parallel* ist deshalb problematisch, weil sie für verschiedene Begriffe verwendet wird. (4a)

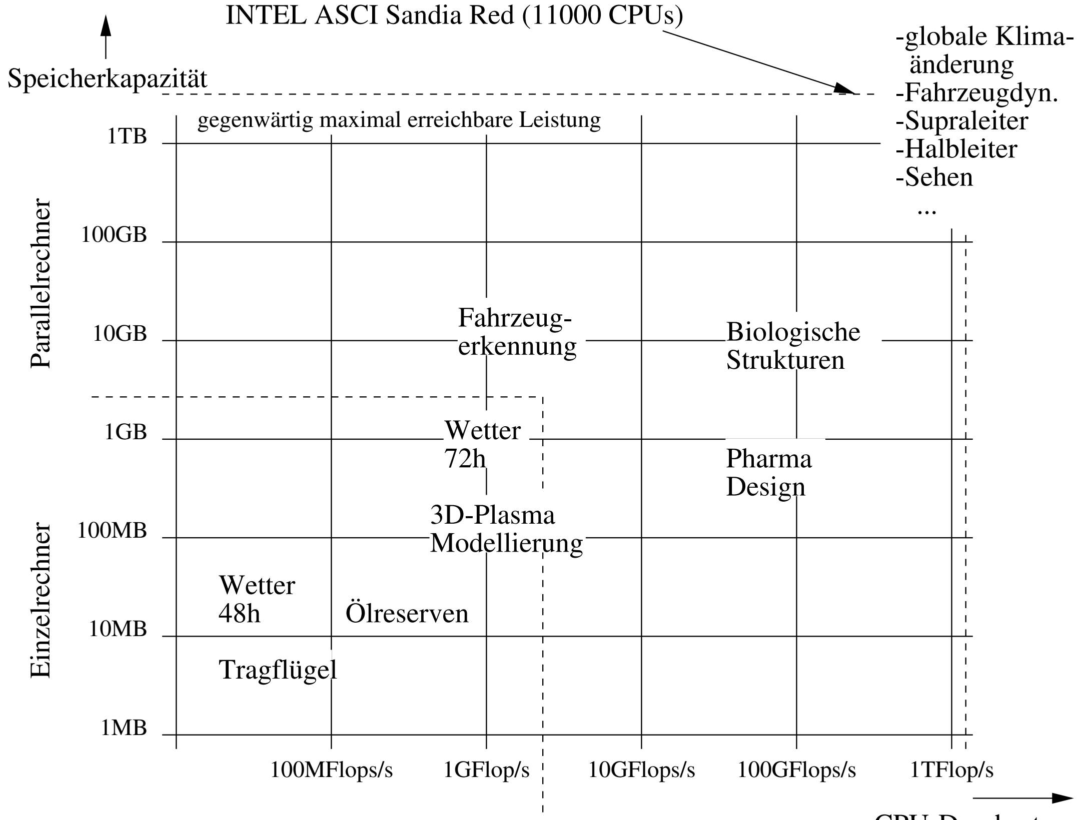
- nebenläufig ("parallele Aufträge")
- kollateral ("Parallelrechner")
- gleich beauftragt/beauftragbar ("parallele Rechenwerke"), bei Bode nebenläufig, vgl. auch "Parallelwiderstand"

Ziele der Parallelrechnerentwicklung **(4b)**

- Kürzung der Rechenzeiten für heutige Jobs
- akzeptable Rechenzeiten für künftige Probleme: insgesamt wichtiger! Seit 1995 sind Parallelrechner *die* Struktur für Höchstleistungsrechner! (bis 1995: Vektorrechner) vgl. die nachfolgende Darstellung der *Grand Challenges*!
- bessere Erweiterbarkeit/Konfigurierbarkeit als herkömmliche Rechner
- (möglicherweise:) bessere Fehlertoleranzeigenschaften als herkömmliche Rechner (Vorteile: funktionsbeteiligte Redundanz möglich, Fehlerabschottung; Nachteile: Diagnose, Wiederanlauf (mindestens bei Multirechnersystemen) schwieriger).
- natürliche Organisation für die Eingliederung von Spezialprozessoren
- wissenschaftliches Interesse
- meist bessere Redundanz aber schlechtere Recoveryeigenschaften

- mehr Bauteile, deshalb auch höhere Ausfallwahrscheinlichkeit

Die großen Herausforderungen des Rechnens (The Grand Challenges of Computing)



(4c)

1.2 Beispiele und Gegenbeispiele

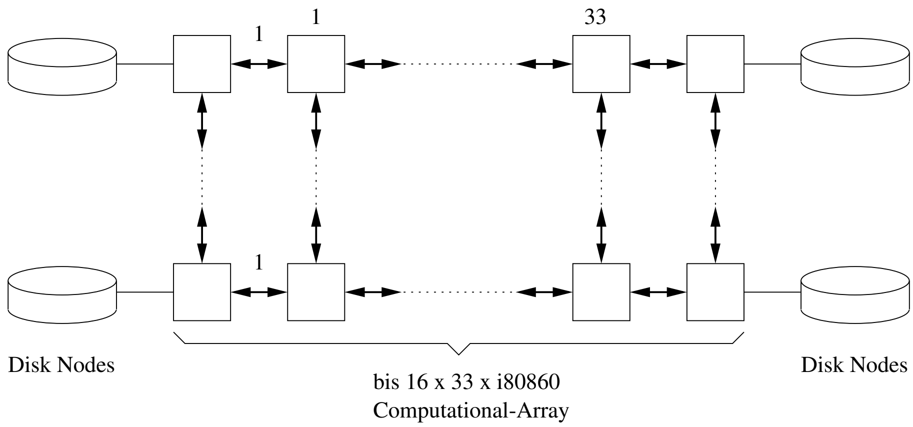
Parallelrechner nach unserer Definition sind:

a) Multirechnersysteme **(5)**

Systeme aus mehreren Rechnern, jeweils aus Prozessor (CPU) und eigenem Hauptspeicher; nur lokale Adressierung (*NORMA*: no remote memory access)

—

Beispiel: Intel Paragon (1992)



528 (16x33) Rechner in Matrixanordung, i80860 mit jeweils 16 MB, dazu 2x16 Disk Nodes; 17 GIps, 32 GFlops

—

Die Prozesse interagieren durch Interprozesskommunikation (Nachrichten zwischen den Rechnern oder auf den Rechnern, falls nebenläufige Prozesse im Zeitmultiplex auf denselben Rechnern laufen, was sinnvoll sein kann).Es gibt ein systemglobales Betriebssystem und lokale Betriebssysteme (Microkernels). (6)

Multirechnersysteme treten in 4 Spielarten auf:

- Konzentriertes Multirechnersystem geringer bis mittlerer Zahl von Rechnern (≤ 100) ([HWANG XU 98]:"Cluster"). In diese Klasse gehören die normalen Parallelrechner. (7)
- Konzentriertes Multirechnersystem höherer (> 100) Zahl von Rechnern (Heute bis etwa 10000): MPPs, Massively Parallel Processors). Diese Klasse hat immer noch einen eher experimentellen Charakter, ist aber für Erreichung von Teraflops-Durchsatz wichtig! (8)
- Netze von PCs, oder Workstations, soweit in ihnen Aufträge durch kollaterale Bearbeitung von Teilaufträgen auf Rechnern des Netzes erledigt werden. Im Allgemeinen sind die PCs oder Workstations durch ihre eigentlichen Aufgaben nur schwach ausgelastet (etwa 5%). (9a)
- Netze von Hochleistungs(parallel)rechnern, wo ein einzelner nicht ausreicht (geographisch weit verteilt, typisch 10..40ms Verzögerung durch das Netz) (9b)

Die beiden letzten Nutzungsformen heißen auch *Metacomputing*. (9c)

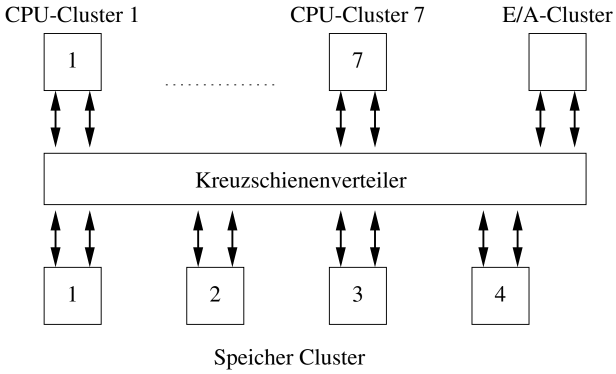
b) Multiprozessorsysteme **(10)**

Systeme aus mehreren Prozessoren und Hauptspeicher, der von allen Prozessoren adressierbar und hardwaretechnisch zugreifbar ist (dies ist eine weite Definition, der wir in dieser Vorlesung folgen). Das gilt auch für Peripherie!

Engere Definition: Systeme aus mehreren Prozessoren mit gemeinsamem Hauptspeicher, der gleichartig (z.B. gleiche Zugriffszeit) für alle Prozessoren zugreifbar ist.

—

Beispiel: Alliant FX/2800 (1990)



Bis 28+1 CPUs (Intel i80860) in 7 Clustern zu je 4 Prozessoren, Kreuzschienenverteiler bietet Zugang zu 4 Speicherclustern mit Cache, insgesamt 1 GB.

—

Diese Maschinen heißen auch (11)

UMA uniform memory access machines, oder

SMP symetric multiprocessor (schlechte Bezeichnung da sich "symmetrisch" auch auf die Betriebssystemnutzung bezieht) oder shared memory (multi)processor

Die übrigen Multiprozessorsysteme ordnen jedem Prozessor einen (schnell zugreifbaren) Hauptspeicher zu: (12)

NUMA non-uniform memory access machines, oder

DSM distributed shared memory (multi)processor.

Sie gleichen also Multirechnersystemen (die allerdings nicht global adressieren können). Der Betrieb wird so geführt, daß die Prozessoren ihre Zugriffe vorzugsweise aus dem eigenen Speicher tätigen (Prozessorzugriff 10 mal öfter als Peripheriezugriffe). (13)

Während auf einem Mehrrechnersystem die kollateralen Prozesse über (Betriebssystem / E/A-Schnittstellengestützte) *Botschaften* kommunizieren (message passing), können sie auf einem Mehrprozessorsystem gemeinsame *Speicherzellen/Variablen* benutzen (Shared Variables), was Effizienzvorteile bringt (aber i.a. trotzdem korrekte Zugriffssynchronisation verlangt!!).

UMA/SMP-Maschinen haben als wesentliches Handicap die für mehr als 16 Prozessoren (trotz Cache) nur schwer erfüllbaren *Durchsatzforderungen am gemeinsamen Speicher*. (14a)

NUMA/DSM-Maschinen (ebenso wie Mehrrechnersysteme) lassen jeweils einen Prozessor mit seinem Speicher; das Verbindungswerk / fremde Speicher tragen nur den um 100..10000 mal kleineren *Kommunikationsdurchsatz zwischen den Prozessen*. (14b)

Da alle heutigen Prozessoren Cache-Memories haben, entstehen *Kohärenzprobleme* zwischen den Caches und zwischen Caches und Hauptspeicher. Außerdem "sehen" nicht alle Prozessoren dieselbe Folge von Lese- und Schreiboperationen am Hauptspeicher (mangelnde "*Ablaufkonsistenz*").

Nicht alle Mehrprozessorsysteme sind *Parallelrechner*! (ein oder wenige Jobs laufen als nebenläufige Teilaufträge kollateral auf den Prozessoren ab). Daneben (typisch am oberen Ende von Rechnerfamilien) gibt es Mehrprozessorsysteme zur *Durchsatzerhöhung*: Das Betriebssystem verteilt die Jobs einer Warteschlange auf jeweils einen freien Prozessor; jeder Job hat höchstens einen Prozessor! (Hauptmerkmal eines Parallelrechners nicht erfüllt). (14c)

vgl. (23)

Tabellarischer Überblick über Mehrprozessor- und Mehrrechnersysteme (hiervon 3 Spielarten) aus [Hwang Xu 98]: (16)

System Characteristic	MPP	SMP	Cluster (PCs, Workstations an LAN)	Distributed Systems
Number of nodes (N)	$O(100)$ - $O(1000)$	$O(10)$ or less	$O(100)$ or less	$O(10)$ - $O(1000)$
Node complexity	Fine or medium grain	Medium or coarse grain	Medium grain	Wide range
Internode communication	Message passing or shared variables for DSM	Shared memory	Message passing	Shared files, RPC, message passig
Job scheduling	Single run queue at host	Single run queue	Multiple queues but coordinated	Indipendent multiple queues
SSI suport	Partially	Always	Desired	No
Node OS copies and Type	N (microkernel) and 1 host OS (monolithic)	One (monolithic)	N (homogeneous desired)	N (heterogenous)
Address space	Multiple (single if DSM)	Single	Multiple	Multiple
Internode security	Unnecessary	Unnecessary	Required if exposed	Required
Ownership	One organization	One organization	One or more organizations	Many organizations
Network protocol	Nonstandard	Nonstandard	Standard or nonstandard	Standard
System availability	Low to medium	Often low	Highly available or fault-tolerant	Medium
Performance metric	Throughput and turnaround time	Trunaround time	Throughput and turnaround time	Response time
Examples	Cray T3E, CM5, Berkeley Now, ASCI, VPP700	SGI Power Challenge, DEC Alpha Server, SUN Enterprise	PCs, Workstations an LAN	Rechnernetze

Vergleich von Clustern, MPP, SMP und verteilten Systemen (SSI: Single-system image, RPC: Remote procedure call, DSM: Distributed shared memory

c) Rechner mit Mehrstromprozessor (15)

Prozessor bearbeitet in starrem Wechsel Befehle mehrerer Aufträge des Auftragssystems; hardware implementierter Mikrozeitscheibenbetrieb befehlsweise. Mindert die Pipeline-Turbulenz!

nichts als eine andere Implementation eines Multiprozessors.

Eng benachbart: *Multithreading-Prozessoren* (aber hier ist das Ziel die Versteckung der Speicherzugriffszeit).

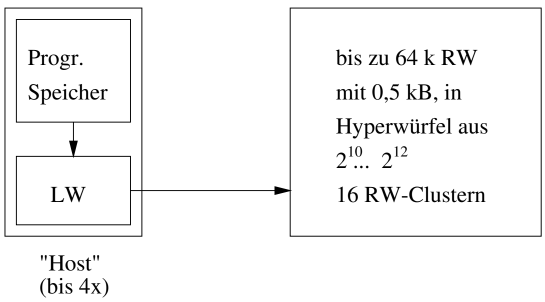
Beide Lösungen verlangen sehr kurze Umschaltzeit (befehlsweise oder nach wenigen 10 Befehlen), daher i.a. mehrfache Registersätze! (15a)

Multithreading-Prozessoren bearbeiten eine Menge von Tasks ("Threads") in einem gemeinsamen Adressraum. Bei NUMA-Maschinen treten (mit geringer Häufigkeit) nicht lokale Hauptspeicherzugriffe auf. Ähnliches gilt (mit größerer Wahrscheinlichkeit, aber kleinerem Verzug) für Cache Misses. Der Prozessor setzt den Thread ab und nimmt einen Rechenfähigen auf; vgl. Verhalten bei Plattenzugriffen bei Mehrprogrammbetrieb.

d) Feldrechner (array computer) (17)

Rechner enthält ein Leitwerk und viele Rechenwerke, regulär verbunden ("Feld") und synchron mit jeweils der gleichen Operation beauftragt. Typisch Datenspeicher in rechenwerkslokale Speicher aufgelöst. Daten oft Komponenten regulärer Datenstrukturen (z.B. Matrizen). Programmierer / Compiler organisiert die Abbildung der Daten auf das Rechenwerk / Speicherfeld. Hierzu gehört z.B.

Thinking Machines Connection Machine II



Das "Feld" ist ein Hyperwürfel aus 2^{10}, 2^{11}, 2^{12} Clustern; jeweils 16 1 Bit Rechenwerke mit je 512 B auf einem Cluster; insgesamt 2 GIps, 32 MB Datenspeicher. Programmspeicher, Leitwerk ist Teil des "Hosts", z.B. VAX 11-780

Einschub Beziehung zu anderen Klassifikationen:

Multirechner , Multiprozessorsysteme, Mehrstromprozessoren sind *MIMD-Rechner* (multiple instruction-streams, multiple data-streams), d.h. sie erzeugen mehrere Programminterpretationen mit jeweils eigenem Datenstrom. M(SISD) (19)

Feldrechner sind *SIMD-Rechner* (single instruction-stream, multiple data-streams), d.h. eine Programminterpretation arbeitet auf mehreren Datenströmen. (20)

Datenflußmaschinen werden meist nicht so klassifiziert (kein konventionell sequentielles Programm)

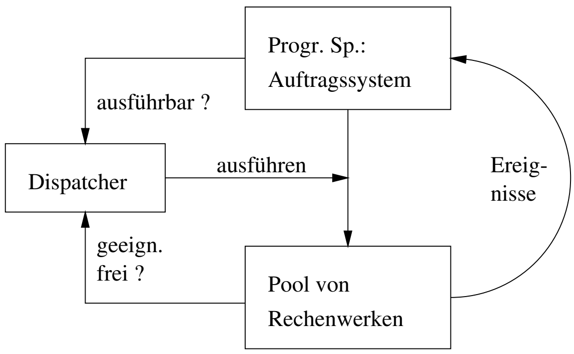
Mehrrechnersysteme, insbesondere Rechnernetze, sind Beispiele von *verteilten Systemen* (distributed systems), d.h. der *Gesamtzustand des Systems* ist im Betrieb nicht erfaßbar bzw. wird nicht erfaßt. (21)

Außerhalb der SISD-SIMD-MIMD-Ordnung:

e) Datenflußrechner **(18)**

Mit Hardwareimplementiertem Datenfluß auf der Ebene einzelner Befehle. Das "Programm" ist ein (nebenläufiges!) Auftragssystem, Freigabe von Operationen durch "Dispatcher" jeweils dann, wenn

- Argumente bereitstehen
- geeignetes Rechenwerk frei



Datenflußmaschinen (dataflow computer) sind heute ohne technische/kommerzielle Bedeutung.

Aber durch Compilation oder Analyse zur Laufzeit gewinnen heutige Hochleistungsprozessoren aus dem sequentiellen Programm des von Neumann-Typs die immanente Nebenläufigkeit zurück und arbeiten beim Start der Operationen faktisch nach dem Datenflußprinzip.

Die Fälle a),b),d) sind Parallelrechner im engeren Sinn.

Gegenbeispiele zu "Parallelrechner"

f) Rechnernetz **(22)**

in üblicher Betriebsform, d.h. die Rechner arbeiten an verschiedenene Aufgaben trotz Kommunikation, Diensterbringung.

g) Mehrprozessorsystem **(23)**

in konventioneller Betriebsform (Betriebssystem verteilt unabhängig Aufgaben (Jobs) auf die Prozessoren), weit verbreitet im oberen Leistungsbereich der Großrechner (mainframe) Familien, z.B. IBM, SIEMENS. Tendenz, Betriebsform nach b) zusätzlich zu unterstützen.

h) Pipeline-Rechner (pipeline processor/computer) (24)

sequentielles Programm, bei dem aufeinanderfolgende Befehle in Fließbandtechnik überlappt ausgeführt werden; typisch geringe Nebenläufigkeit. Programmierer/Compiler *muß Nebenläufigkeit nicht angeben*, Berücksichtigung bringt aber höhere Leistung. Beispiel: alle Prozessoren mittlerer und höherer Leistung.

Bode: "Phasenpipelining"

f) Vektorrechner (vector processor/computer) (25)

sequenzielles Programm, bei dem die (typisch breit) nebenläufigen Operationen auf den Komponenten regulärer Datenstrukturen kollateral in (arithmetischer) Pipeline abgearbeitet werden. Programmierer bzw. Compiler muß Nebenläufigkeit nicht deklarieren, er kann sie zur Verbesserung der Leistung aber berücksichtigen. Von 1975 bis 1995 der Grundtyp des Hochleistungsrechners, z.B. CRAY 2. Daher stets mit Merkmalen von h), meist auch g), neuerdings auch a) oder b). Spitzendurchsatz heute bei einigen GIps/GFlops; gute Programmierwerkzeuge.

Solche Vektorrechner, die auch die Merkmale a) oder b) (Multiprozessor oder Multirechner (hier: konzentriert, mässige Anzahl)) aufweisen, heißen auch *PVPs* (parallel vector processors). Diesen Parallelrechnern zuliebe werden in 3.3 auch Vektorrechner behandelt.

Beziehung zu anderen Klassifikationen

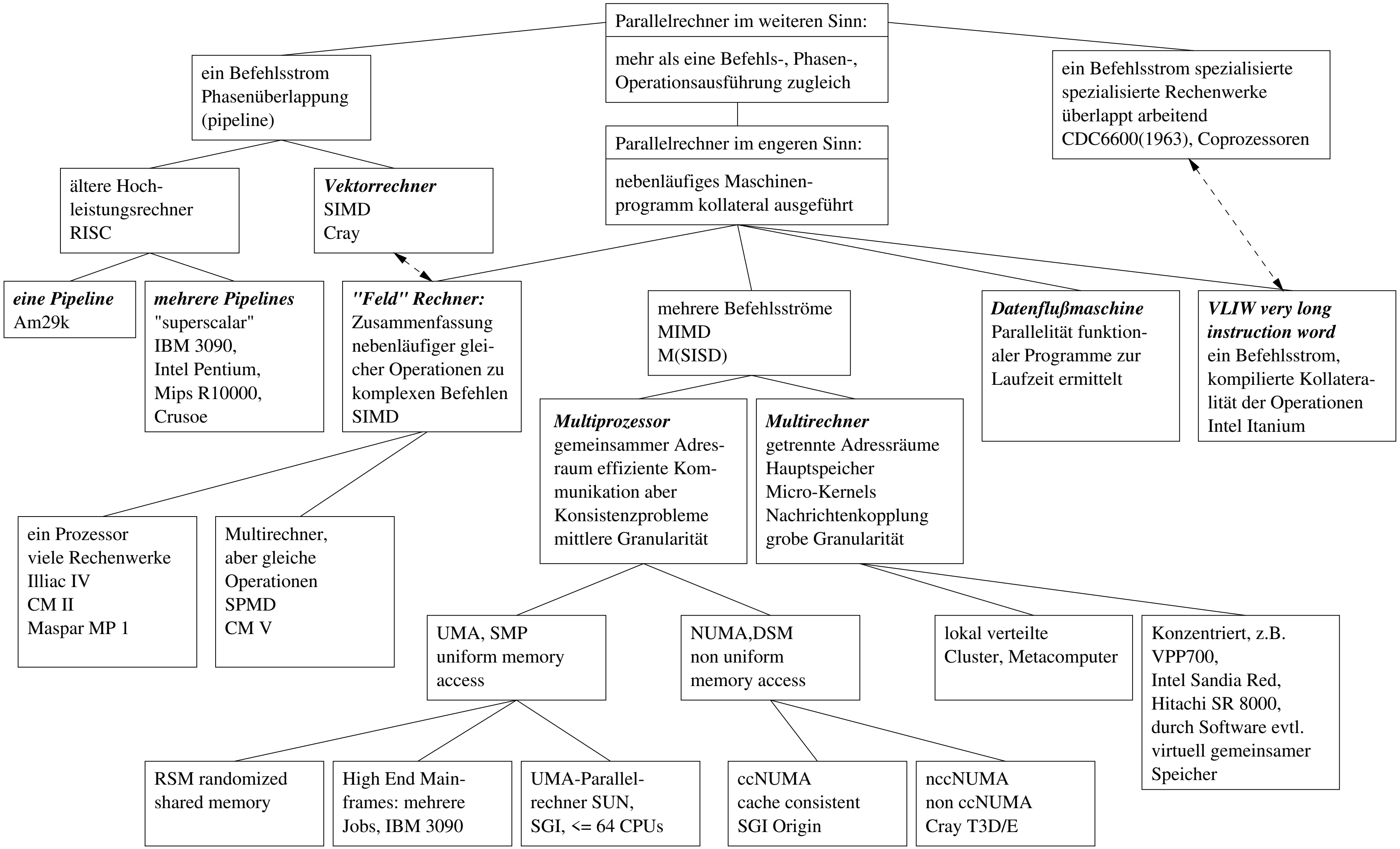
Rechnernetze und Mehrprozessorsysteme sind MIMD-Rechner.

Pipelinerechner nach h) sind SISD-Rechner, aber Vektorrechner sind SIMD-Rechner. Vektorrechner und Feldrechner sind verschiedene Implementierungen des selben SIMD-Prinzips, aber

- Feldrechner erlauben oft physische Nachbarschaft der Rechenwerke auszunutzen (sie müssen es!)
- Feldrechner überlassen die Abbildung der Datenstruktur auf das Rechenwerksfeld dem Programmierer / Compiler (deshalb sind sie hier “Parallelrechner”)

Bei unserer Parallelrechnerdefinition ist Parallelität Teil der dem Programmierer / Compiler sichtbaren Funktionalität.

(Vermutlich) Ist Weiterentwicklung zu sequentiellen (?) oder nichtprozeduralen Programmierschemen wünschenswert.

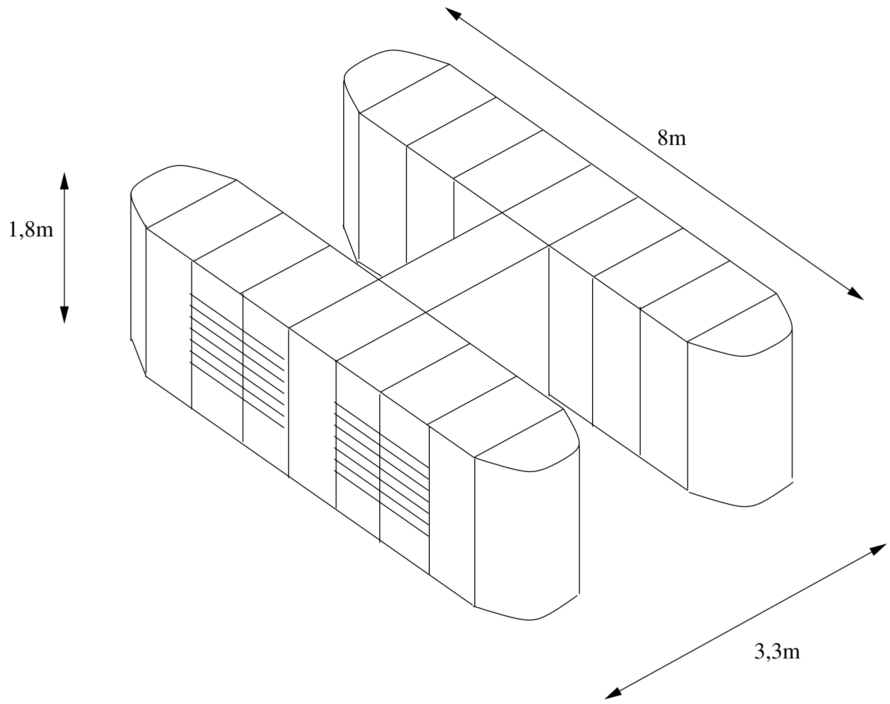

Parallelrechner im weiteren Sinn:
mehr als eine Befehls-, Phasen-, Operationsausführung zugleich
ein Befehlsstrom Phasenüberlappung (pipeline)
ein Befehlsstrom spezialisierte spezialisierte Rechenwerke überlappt arbeitend CDC6600(1963), Coprozessoren
Parallelrechner im engeren Sinn:
nebenläufiges Maschinenprogramm kollateral ausgeführt
ältere Hochleistungsrechner RISC
Vektorrechner SIMD Cray
eine Pipeline Am29k
mehrere Pipelines "superscalar" IBM 3090, Intel Pentium, Mips R10000, Crusoe
"Feld" Rechner: Zusammenfassung nebenläufiger gleicher Operationen zu komplexen Befehlen SIMD
mehrere Befehlsströme MIMD M(SISD)
Datenflußmaschine Parallelität funktionaler Programme zur Laufzeit ermittelt
VLIW very long instruction word ein Befehlsstrom, kompilierte Kollateralität der Operationen Intel Itanium
Multiprozessor gemeinsammer Adresraum effiziente Kommunikation aber Konsistenzprobleme mittlere Granularität
Multirechner getrennte Adressräume Hauptspeicher Micro-Kernels Nachrichtenkopplung grobe Granularität
ein Prozessor viele Rechenwerke Illiac IV CM II Maspar MP 1
Multirechner, aber gleiche Operationen SPMD CM V
UMA, SMP uniform memory access
NUMA,DSM non uniform memory access
lokal verteilte Cluster, Metacomputer
Konzentriert, z.B. VPP700, Intel Sandia Red, Hitachi SR 8000, durch Software evtl. virtuell gemeinsamer Speicher
RSM randomized shared memory
High End Mainframes: mehrere Jobs, IBM 3090
UMA-Parallelrechner SUN, SGI, <= 64 CPUs
ccNUMA cache consistent SGI Origin
nccNUMA non ccNUMA Cray T3D/E

1.3 Zwei Beispiele vorab: SR 8000 und ORIGIN 2000

Beispiel: HITACHI SR 8000 F1 (LRZ)

Das Leibnitz-Rechenzentrum der Bayrischen Akademie der Wissenschaften installierte im 2. Quartal 2000 die erste Ausbaustufe des *Bundeshöchstleistungsrechners*, der Benutzern in ganz Deutschland für sehr große Rechenaufgaben zur Verfügung stehen soll. Zugang über Netz. Kontingente werden vom Beirat verteilt.

Die Wahl fiel Aufgrund des Anforderungsprofils (Höchstleistungsrechner für 75% gut "vektorisierbare" Probleme, 25% anders parallelisierbare Probleme) auf die Hitachi SR 8000 F1, obwohl das Produkt noch in der Markteinführung ist.



HITACHI SR 8000 F1 am LRZ, 500 kW luftgekühlt! im Vollausbau 57MioDM, 25,2t

Dreistufige Hierarchie von Parallelität:

- 112 Knoten
- jeder Knoten ein Mehrprozessorsystem aus 8+1 CPUs, in SPMD-Modus (single program multiple data) betreibbar (synchrone Abarbeitung desselben Codes)
- übliche superskalare Pipelining-CPUs
- alle Rechner haben Platten zur Speicherverwaltung

	1.Ausbaustufe 1.Quartal 2000 schnellster Rechner in Europa	2.Ausbaustufe 2002
Zahl der Knoten (je 9 CPUs, insges. 12 GFlops/s)	112	168
Grenzdurchsatz CPUs	1,34 TFlops/s	2,02 TFlops/s
bei typischer LRZ-Last mindesten	0,4 TFlops/s	0,6 TFlops/s
Hauptspeicherkapazität	0,93 TB	1,34 TB
Grenzdurchsatz interne Kommunikation	56 GB/s	75 GB/s
Plattenkapazität	2,2 TB	2,8 TB
Grenzdurchsatz Platten	2,3 GB/s	3,0 GB/s

Die Maschine ist ausbaubar bis 512 Knoten (dann *6,1 TFlops*).

Verbindungswerk: "dreidimensionales *Crossbar*", 1 GB/s von Knoten zu Knoten, 19 μs Transportzeit für eine kurze Nachricht (ein Crossbar verbindet konfliktfrei jeden der p Eingänge mit einem beliebigen aber freien der q Ausgänge).

Knoten der Hitachi SR 8000 F1

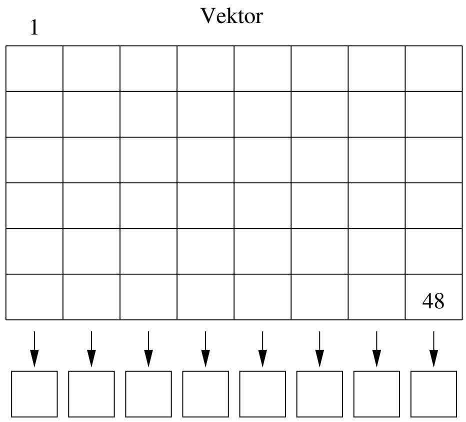
Die Knoten sind kleine Mehrprozessorsysteme aus 8+1 (Hilfsfunktionen) CPUs, die über ein Speicherzugangswerk (Crossbar mit aktiven Synchronisations- und Kommunikationsfunktionen zwischen den CPUs) auf den gemeinsamen, modularen Hauptspeicher zugreifen.

Die *CPUs sind Eigenentwicklungen* von Hitachi auf der Grundlage der IBM-Power-Prozessor-Architektur:

- 160 Gleitpunkt-Register (32 Global, 128 Thread-spezifisch)
- 32 Festpunktregister
- 128 kB Daten-Cache
- 64 kB Befehls-Cache
- prefetch-Befehle laden Blöcke in Cache vor, preload-Befehle laden Register vor
- 64b Adressierung
- Grenzdurchsatz 1,5 GFlops/s

Grenzdurchsatz 8 x 1,5 GFlops/s = 12 GFlops/s

Speicherzugangswerk enthält Kopie aller tag’s (Cache-Block-Kennzeichen) aller CPUs des Knotens, zur Konsistenzsicherung. Hauptspeicher 8 oder 16 GB.



In einer Art SPMD-Betrieb (single program multiple data) können alle 8 CPUs denselben Code annähernd synchron abarbeiten, was für Vektoroperationen eingesetzt wird (HITACHI: *PVP pseudo vector processing*) und die (vom LRZ benötigte) gute Eignung für vektorisierbare Operationen schafft.

Daneben konventioneller SMP-Multiprozessorbetrieb, mit verschiedenem Code.

Beispiel: SGI Origin 2000

SGI (Silicon Graphics), Hersteller von Graphik-Hochleistungsworkstations, kauft etwa 1996 Superrechner-Marktführer CRAY, inzwischen mit ORIGIN auch "eigener" Superrechner. Insgesamt ca. 40% Marktanteil bei Superrechnern. (inzwischen CRAY wieder verkauft!).

SGI ORIGIN: weit ausbaubarer (2..1024 Prozessoren) Multiprozessor mit sehr komplexem Interkonnektionswerk.

Allerdings fremde Speicher deutlich verlangsamt: *NUMA* (non-uniform-memory-access). Zum Ausgleich werden Blöcke von 128B in lokale Caches verlagert, mit Hardware-Konsistenz-Kontrolle (über Directories, die Aufenthaltsorte der Blöcke angeben): ccNUMA (cache-coherent NUMA).

CPUs sind MIPS R 10000:

- je 400 MFlops/s bei 200 MHz
- je CPU 2 GB Hauptspeicher

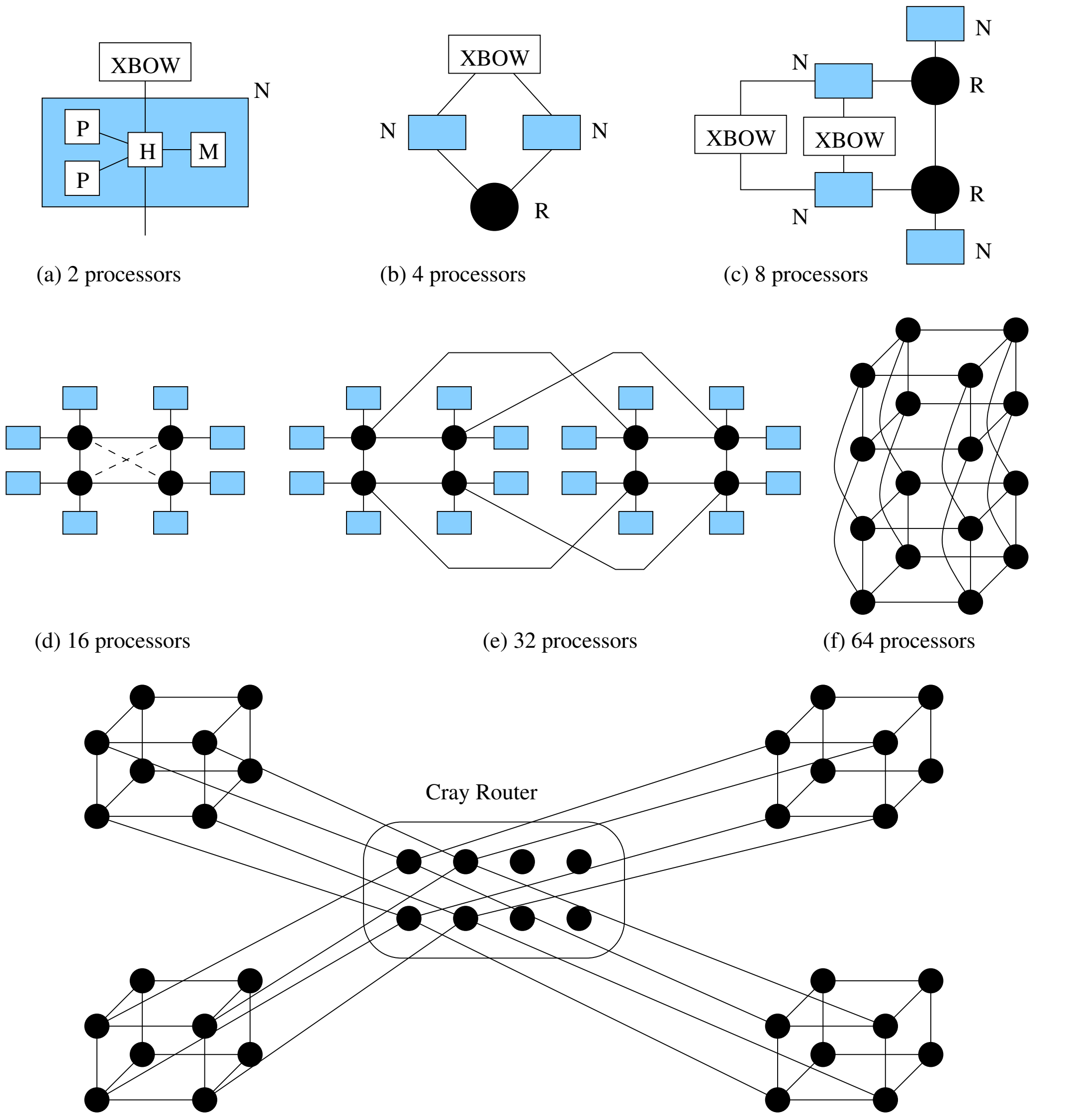
bis 128 CPUs => 51,2 GFlops/s, 256 GB

Interkonnektionsnetz

"Fetter" Hyperwürfel mit 2 Rechnern je Knoten, Dimension 6 (64 Knoten). "Fett" heißt: zusätzliche Würfeldiagonalen sichern, daß bei Vergrößerung der Grenzdurchsatz proportional mit der Zahl der Knoten wächst.

Ausführliche Behandlung der SGI ORIGIN in 4.3.2.1!

Aus [Hwang Xu 98]:



(a) 2 processors (b) 4 processors (c) 8 processors

(d) 16 processors (e) 32 processors (f) 64 processors

(g) 128 processors (For clarity, only 16 connections to the left four metarouters in the Cray router are shown)

Topologie des in der ORIGIN verwendeten "fetten" Hypercubes. (N: Nodes aus 2 Prozessoren Hub und Speicher, M: Speicher, P: Processor, H: Hub, XBOW: Crossbar I/O Switch, R: Router) Nach Laudon und Lenoski, Proc. of the 24th Int'l Symp. Computer Architecture, 1997

Konfiguration d: "fett"

1.4 Geschichte und Stand des Gebietes

Geschichte:

1957	Zuse's SIMD-Patent
1958	IBM AN/FSQ 31,32 : Doppelrechnersysteme für SAGE (semi automatic ground environment, Luftverteidigung)
1960	Mehrrechnersystem Ramo-Wooldridge RW 400 Mehrstrom-Prozessor Honeywell H 800
1961	IBM 7030 "Stretch" Pipeline-Prozessor
1962	Westinghouse SOLOMON-Feldrechner
1963	Mehrprozessorsystem Burroughs 825 D
1972	ILLIAC IV Feldrechner mit 64 Rechenwerken, Goodyear STARAN Assoziativer Rechner
1974	Vektorrechner Control Data Star 100, LAU Datenflußmaschine (Konzept), C.mmp (CMU): 16-Prozessor PDP 11
1975	Lokale Netze
1976	Cray-1 Vektorrechner

Ab 1976 erhalten Mehrprozessoranlagen (aber noch nicht als Parallelrechner), Vektorrechner und (weniger) Feldrechner eine kommerzielle Bedeutung.

1980	Vektorisierende Compiler für Vektorrechner
1981	Caltec Cosmic Cube: erster Multirechner als Parallelrechner
1983ff	Intel IPSC 1, 2, 860 : Hypercube-Multirechner
1985	Thinking Machines: Connection Machine-2: Hypercube-Feldrechner mit 64 k Rechenwerken
1987	University of Illinois: Cedar: Virtuell gemeinsamer Speicher
1989	Cray Y-MP: Multivektorrechner
1990	MasPar MP-1: Mehrprozessorsystem als Parallelrechner
1990	Kendall Square Research KSR-1 : COMA
1992	Connection Machine 5: SPMD/MIMD-Maschine mit Fat-Tree-Verbindungswerk, Intel Paragon: Multirechner auf ebenem Gitter Stanford University: DASH: ccNUMA

1993 Cray T3D Multivektorprozessor mit 1024 DEC Alpha-Rechner als 3D-Torus

1996 Intel ASCI: erster Teraflop-Rechner, etwa 10000 Prozessorelemente

Ab 1995 verdrängen die Parallelrechner die (Poly)Vektorprozessoren von ihrem Platz als Superrechner. Zugleich tritt durch MPI und PVM eine gewisse Stabilisierung der Programmierparadigmen für message passing ein.

Stand des Gebietes

- Rechenverfahren für Parallelrechner theoretisch intensiv erforscht (z.B. auf PRAM- oder BSP-Idealmaschienen); technische Verfahren wachsend verstanden und verbreitet; Nutzung wegen Abhängigkeit von den verschiedenen Maschinenarchitekturen schwierig!
- Programmiersprachen für Datenparallelität ($\sim$SIMD), Fortran-Offsprings gut, sonst schwach stabilisiert; viele Ansätze bekannt. Automatische Parallelisierung bei SIMD gut, bei MIMD schwach beherrscht.
- Werkzeuge für interaktiv/experimentelle Verfahrens/Programmentwicklung verbreiten sich.
- Betriebsstrategien für Allokation der Daten/Programmstrukturen, Kommunikation, Parallelitätssteuerung, Mehrprogrammbetrieb in Entwicklung, verbreiten sich, allerdings auch stark Architekturabhängig.
- "Konnektionismus" (massive Parallelität schwacher Prozessoren) ist out; statt dessen mäßige Parallelität starker Prozessoren in.
- Übergang zu Zustandsmikroprozessoren, bis 10^4.
- Interkonnektionswerk wichtigstes Strukturmerkmal
- Instabiler Herstellermarkt, bescheidene Installationszahlen. Hardwarezuverlässigkeit befriedigend.

1.5 Modellvorstellungen

1.5.1 Berechnungsstrukturen: Auftragsysteme

Auftragssysteme sind ein sehr mächtiges Mittel zur Beschreibung der durch einen Algorithmus induzierten Last und sehr nützlich für die Analyse der Parallelisierbarkeit.

Parallelrechner nutzen, daß Aufträge (jobs) in Teilaufträge (tasks) zerfallen, die nebenläufig sind.

Auftragssystem $AS = (\mathbb{A}, \lessdot)$ (27)

$\mathbb{A}$ Menge der Teilaufträge

$\lessdot$ *Präzedenzrelation*, $a_i, a_j \in \mathbb{A} \wedge (a_i, a_j) \in \lessdot :$ a_i ist *präzedent* zu a_j, "$a_i \lessdot a_j$".

$(a_i \lessdot a_j) \vee (a_j \lessdot a_i) :$ a_i, a_j sind *sequentiell* sonst sind sie *nebenläufig*.

Als *Synchronisation* bezeichnet man eine Maßnahme zur Sicherung der Präzedenz.

Für die Abarbeitung auf Parallelrechnern (auch Vektorrechner, Phasenpipelining) sollte das Auftragssystem möglichst wenige sequentielle Teilaufträge haben, d.h. $| \lessdot | \ll \frac{n(n-1)}{2}$ bei n Teilaufträgen.

Das Auftragssystem muß aber Forderungen genügen, daß das *Resultat* unabhängig von der parallelen Ausführung bleibt! Genauer:

Die Berechnung erfolge in einem globalen Zustandsraum, Variablenmenge $\mathbb{V}$. Jeder Teilauftrag a_i kopiert einen *Eingabebereich* $E_{a_i} \subseteq \mathbb{V}$ auf private Variable, rechnet dann, kopiert private Variable auf einen *Ausgabebereich* $A_{a_i} \subseteq \mathbb{V}$; lesen, rechnen, schreiben. Lesen und Schreiben sind Ereignisse ohne Zeitverbrauch.

Eine *Ausführungsfolge* eines Auftragssystems ist eine Folge von n Lese- und Schreibereignissen, unter Einhaltung der Präzedenz unter den Teilaufträgen.

Ein Auftragssystem ist *funktional* (determiniert), wenn die Folge der Wertzuweisungen jeder Variablen $\in \mathbb{V}$ unabhängig von der Ausführungsfolge ist. (diese Eigenschaft geht weiter, als daß das Resultat unabhängig von der Ausführungsfolge ist!! aber einfachere Voraussetzungen!)

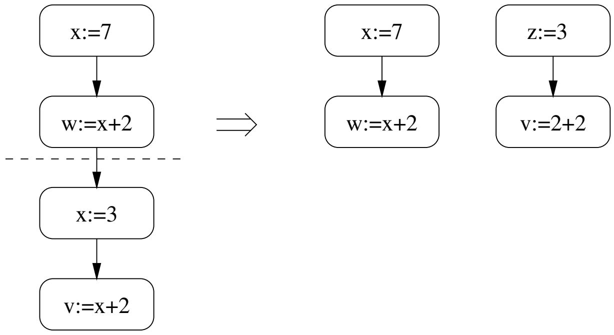
Hinreichende Voraussetzung für die Funktionalität des Auftragssystems ist die *Störungsfreiheit* aller Teilauftragspaare. a_i und a_j sind störungsfrei, wenn (a_i, a_j) sequentiell oder $(E_{a_i} \wedge A_{a_j}) = (A_{a_i} \wedge E_{a_j}) = (A_{a_i} \wedge A_{a_j}) = \emptyset$, d.h. $E_{a_i} \wedge E_{a_j} \neq \emptyset$ ist zulässig!! (213)

Bei der Prüfung auf Störungsfreiheit gibt es noch folgende Erleichterungen bezüglich der zweiten Bedingung: (214)

- Eine Eingabevariable, die ohne Einfluß auf die Ausgabe ist, darf in E gestrichen werden.
- Eine Ausgabevariable, deren Wert durch die Task nicht geändert werden kann, darf in A gestrichen werden.
- Wenn der aktuelle Wert einer Variablen unter keiner Ausführungsfolge mehr gelesen wird, darf sie für den nachfolgenden Gebrauch umbenannt werden.

—

Beispiel:



(27a)

—

"Privatisierung" der Variablen, vgl. auch Register-Renaming.

In Kenntnis der genannten Bedingungen kann (z.B. durch einen vektorisierenden/parallelisierenden Compiler) die Präzedenz so reduziert werden, daß das Auftragssystem *maximal nebenläufig* ist. Die Ausführungsfolge wird

- Auf Datenflußmaschinen erst zur Laufzeit gefunden, nach Maßgabe der gerade freien Betriebsmittel; ebenso auf Hochleistungs-CPUs.
- Auf anderen Maschinen bei der Codeerzeugung festgelegt, nach Maßgabe der vom Compiler als frei geplanten Betriebsmittel. Bei Parallelrechnern ist das Problem erheblich durch die Zuordnung von Teilaufträgen zu geeignet in der Kommunikationstruktur gelegenen Werken kompliziert!

Auswege: günstige Kommunikationsverhältnisse, dynamischer Lastausgleich, randomized shared memory, echte Mehrprozessorsysteme.

Nachbemerkungen zum Auftragssystem:

- Das Auftragssystem ist prinzipiell nicht aus dem Programm erschließbar (*Turing's* Halteproblem).
- *Bedingte* Präzedenzen (a_i gibt a_j unter einer Bedingung frei) und *disjunktive* Bedingungsverknüpfung (a_i oder a_k muß a_j freigeben; Auftrag endet mit a_i oder a_k) sind erforderlich, wenn Auftragssystem als *Programmierparadigma* ($\rightarrow$ Datenflußmaschine).
- Auch die betriebsmittel/schedulingbedingte Synchronisation ist (in einfachen Fällen) in das Auftragssystemmodell einbringbar.
- Nicht modellierbar sind spontane Unterbrechungen.

1.5.2 Maschinen: PRAM und BSP (bulk synchronous parallelism)

[Hwang Xu 99, S.13-23]

PRAM (parallel random access machine) (28)

ist eine Abstraktion eines MIMD-Rechners. Die abstrakte Maschine besteht aus

- k Prozessoren mit festem lokalen Programm, die synchron in jedem Zyklus der Maschine eine (oder keine) atomare Operation ausführen.
- einem gemeinsamen Datenspeicher von m Zellen, für alle Prozessoren gleich zugänglich (UMA!), für jede Operation werden anfangs alle benötigten Werte gelesen, am Ende alle Ergebnisse geschrieben.
- Ein/Ausgabe-Strom in/aus Datenspeicher.

Die PRAM-Modelle spezialisieren sich noch nach der Behandlung der gleichzeitigen Zugriffe auf dieselbe Zelle:

- concurrent read/write: CR/CW oder
- exclusive read/write: ER/EW

Für CW gibt es verschiedene Annahmen zur Konfliktauflösung bei abweichenden Werten.

Zweck des PRAM-Modells:

Einfache Entwicklung und Bewertung (Zeit-, Raumkomplexität) von Algorithmen in einer radikal vereinfachten Umgebung, als standardisierte Plattform:

- reale MIMD-Rechner rechnen asynchron (nur Feldrechner sind operationssynchron)
- die PRAM benutzt ungeschützte Operationen an gemeinsamen Variablen (vgl. 1.5.1: i.a. nicht funktional!) ohne Synchronisationsoperationen. Reale Rechner müssen bei gemeinsamen Variablen die Präzedenz sichern. Oft (Multirechnersysteme!) verwenden sie Botschaften statt gemeinsame Variablen zur Kommunikation.
- reale MIMD-Rechner brauchen zusätzlichen Aufwand für Erschaffung, Gruppierung, Verschiebung von Prozessoren.

Für die PRAM sind sehr viele Algorithmen entwickelt und auf ihre Komplexität untersucht worden. Viele der Algorithmen sind nichtdeterministisch und deshalb nicht auf technischen Maschinen einsetzbar.

—

Beispiel ($a_1 .... e_2$: Konstanten, k: Zahl der Prozessoren) (29)

	sequentielle Maschine	PRAM
Addition von n Zahlen	$a_1 * (n-1)$	$a_2 * \left(\left\lceil \frac{n}{k} \right\rceil - 1\right) + a_3 * ld(k)$
Suchen	$b_1 * ld(n)$	b_2
Sortieren	$c_1 * n * ld(n)$	$c_2 * ld(n)$
Matrixmutliplikation	$d_1 * n^{2,4}$	$d_2 * ld(n) \quad \left(k \geq \frac{n^3}{ld(n)}\right)$
Erfüllbarkeit einer Aussagenlogischen Formel	$e_1 * 2^n$	$e_2 * n$

Das Additionsbeispiel ist typisch für Divide-and-Conquer-Algorithmen.

—

Das Bulk Synchronous Model **(30)**

(BSP) ist etwas realistischer, aber spezieller. Es führt ein realistischeres Modell für Synchronisierung und Kommunikation ein:

- Die Berechnung besteht aus einer Serie von *Supersteps*, innerhalb derer asynchrone parallele Rechenprozesse auf lokalen Daten arbeiten.
- Jeder Superstep wird durch eine globale Synchronisation abgeschlossen ("Barrier Synchronisation"), die erst beendet ist, wenn alle parallelen Prozesse des Supersteps beendet sind.
- Auf die Synchronisation folgt eine Schreib- und Lesephase, in der Ergebnisse der parallelen Berechnungen auf die globalen Variablen übertragen bzw. deren Werte auf lokale Variable übertragen werden; entsprechendes gilt für Botschaftenkommunikation. Dann erst beginnt der nächste Superstep.

Das BSP-Modell

- erfaßt eine spezielle, aber in der Parallelrechnernutzung nicht unrealistische Ablaufstruktur
- erlaubt den Zeitaufwand für Synchronisation und Kommunikation einzubinden
- löst sich von der PRAM-Forderung, daß alle Operationen synchron ablaufen

1.6 Programmierung von Parallelrechnern

[Hwang Xu 98]

1.6.1 Probleme und Grundformen

Die Entwicklung von Parallelrechnern hatte immer einen weiten Vorsprung vor der Entwicklung von parallelen Verfahren, Algorithmen, Programmen und Werkzeugen. Der Grund liegt u.a. darin, daß die technologischen und wirtschaftlichen Parameter der Rechnerentwicklung ungleich günstiger verlaufen als der wesentliche Parameter der Softwareentwicklung: Seit Beginn der industriellen Entwicklung von Rechnern haben sich im Mittel je Jahr verbessert:

Prozessordurchsatz: Faktor 1,5

Kosten je Operation: Faktor 1,5

Größtes noch funktionsfähiges Programm: Faktor 1,2

Parallelrechnerprogrammierung *enthält alle Probleme des sequentiellen Programmierens*, dazu aber

a. nebenläufige Zergliederung

b. Synchronisation

c. Kommunikation

Die Betriebsmittelzuteilung (vor allem Prozessor/Rechnerzuteilung) sind viel schwieriger als auf Systemen, wo ein Benutzerauftrag nur einen Prozessor/Rechner besitzt (was auch auf high-end-Mehrprozessorsystemen der Mainframe-Familien der Fall ist). Die Probleme a, b, c stellen sich (in stark abgeminderter Form) allerdings in jeder Multitasking-Organisation, also z.B. in technischen Realzeitsystemen, On-line-Transaktionssystemen und natürlich in Mehrprozeß-Betriebssystemen. Die Betriebsmittelzuteilung ist bei Parallelrechnersoftware nicht so weit automatisierbar wie bei Einrechnersoftware.

Für sequentielle Programmierung besteht ein universell akzeptiertes (imperativ-prozedurales) Programmiermodell aus Variablen, Anweisungen, Bedingungen. Moduln, seit Fortran (1957), dem mit ein paar Erweiterungen fast alle Sprachen folgen (Algol, Cobol, Simula, Pascal, C, C++, Java, ...). Das erlaubt die Übertragung von Verfahren, und in aller Regel sichert es die Verfügbarkeit aller gängigen Programmiersprachen auf allen Prozessoren. In der parallelen Programmierung gibt es das nicht; mühsam etablieren sich in den letzten Jahren vier Programmiermodelle:

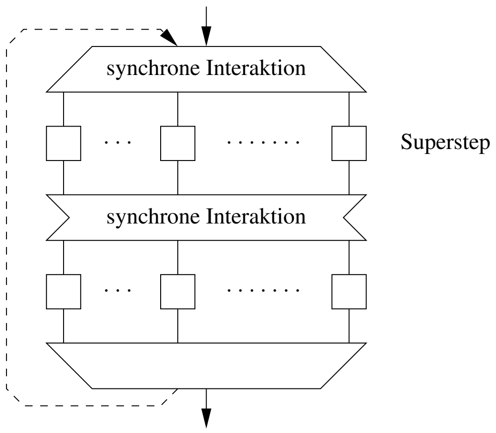
- ohne explizite Parallelität (laß es den Compiler selbst versuchen!)
- mit expliziter Parallelität:
  - Datenparallelität (data parallelism, SIMD, SPMD)
  - Nachrichtenkopplung (message passing, MIMD)
  - Gemeinsame Variable (shared variables, MIMD)

Die vier Programmiermodelle haben eigene Programmiersprachen und sind (vorzugsweise) für bestimmte Maschinenstrukturen geeignet: (31)

Programmiermodell	Sprachen	Maschinenstrukturen
ohne expliziten Parallelismus	(alle)	alle (technisch erfolgreich bei Vektorrechnern)
mit explizitem Parallelismus		
Datenparallelität	Fortran 90, High Performance Fortran (HPF)	Feldrechner, Vektorrechner
Nachrichtenkopplung	Parallel Virtual Machines (PVM), Message Passing Interface (MPI)	Mehrrechnersysteme (konzentriert oder verteilt, mäßig oder massiv parallel)
Gemeinsame Variable	X3H5,OpenMP	Mehrprozessorsystem, über virtuell gemeinsamen Speicher auch Mehrrechnersysteme

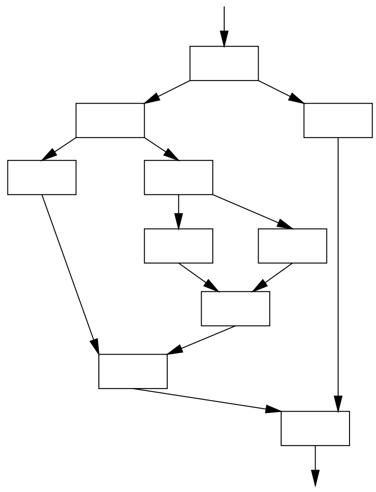
Wichtige Grundbilder paralleler Algorithmen

Phasenparallelität, Serien-Parallelstruktur, Bulk-Synchronous (BSP). **(41)**



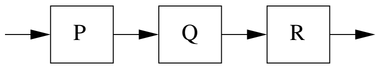
Verschnitt vor synchroner Interaktion! d.h. Minderauslastung bei Einprogrammbetrieb. Bei Einprogrammbetrieb auch zyklisch (hier nicht die parallele Schleife, sondern serielle Schleife!).

Divide and Conquer **(42)**



Rekursive Zerteilung und Sammlung, meist erheblicher Verschnitt! Nur bei Hintergrundlast (Mehrprogrammbetrieb auf Parallelrechner!) effizient.

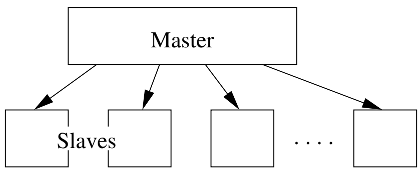
Pipeline **(43)**



z.B. Signalverarbeitung, z.B. Grundwasserfluß → Kontaminationsfluß → Visualisierung

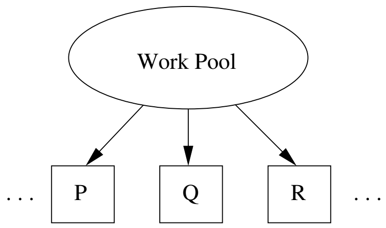
gibt kaum Beispiele hoher Parallelität

Process Farm **(44)**



Master arbeitet seriell, verschickt parallelisierbare Arbeit an seine Slaves. Zentrale Verarbeitung, wobei (von der Streuerung abgesehen) Parallelität eine untergeordnete Rolle spielt.

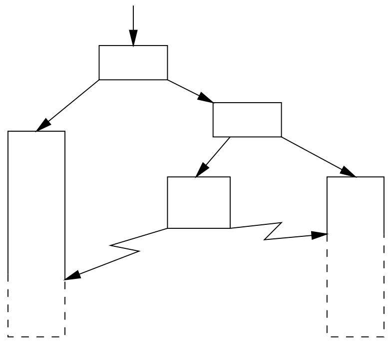
Work Pool **(45)**



Die Prozesse P,Q,R,... holen sich Arbeit aus dem Pool und legen abspaltbare Arbeit dort ab. Pool mit gewisser Synchronisation. Bessere Balancierung der Auslastung der Prozesse!

"Auftragsanziehung" - statt Auftragsvergabe

Wettbewerb **(46)**



Oftmals gibt es mehrere Wege zur Lösung, und es ist nicht bekannt, welcher Weg erfolgreich ist und wie lang jeder Weg ist. Beispiele liefern alle nichtdeterministischen Algorithmen, insbesondere viele Such/Beweisalgorithmen.

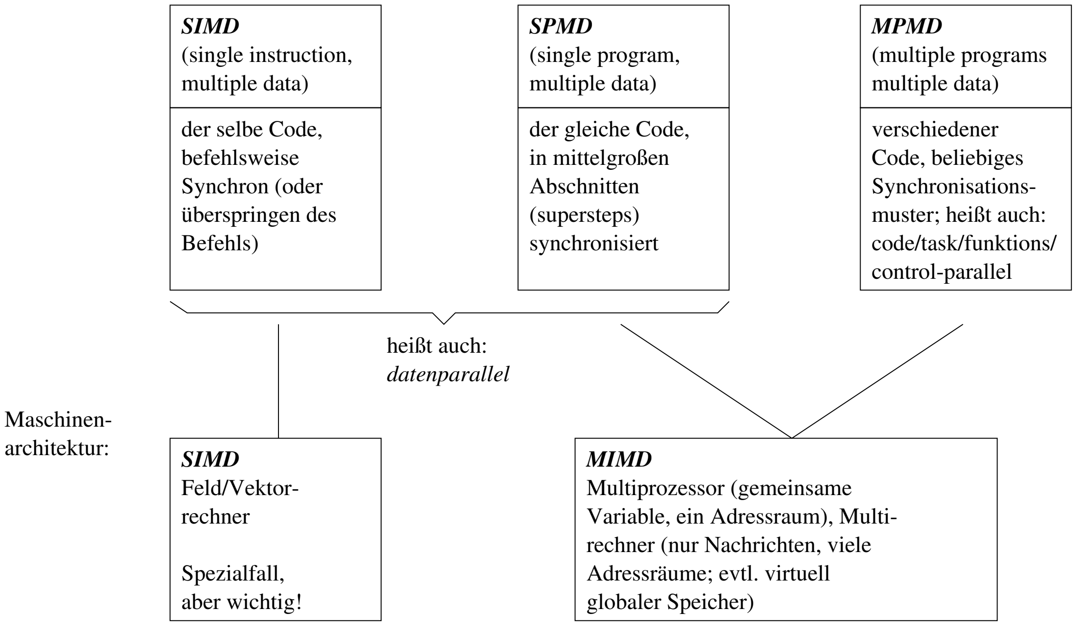
Dann startet man parallel eine Menge von Wegen und läßt den, der das Ziel als erster erreicht, die anderen stoppen.

Wenn unter der ausgewählten Menge (zufällig) einer ist, der sehr kurz ist bis zum Ziel (im Verhältnis zur mittleren Länge), dann ist das Verfahren sehr attraktiv! Insbesondere kann bei Einsatz von k Werken unter Umständen die Ausführungszeit um mehr als k verkürzt werden (Superlinearer Speedup).

Die zuerst betrachteten Grundbilder von Algorithmen ((41)..(45)) sind sämtlich auf das Prinzip der *Aufteilung der Last* (realisiert durch Aufteilung von Code und/oder Daten) gegründet. Wettbewerbsverfahren beschreiten k *vollständige* sequentielle (!) Lösungswege (also *keine Entwicklung eines parallelen Programms*!) und verlangen (außer beim Start und am Ziel) *keine Kommunikation*! Also besonders geeignet für Netze von Rechnern, wo Verzögerung und geringer Durchsatz die Kommunikation hemmen.

1.6.2 Parallele Sprachkonstrukte

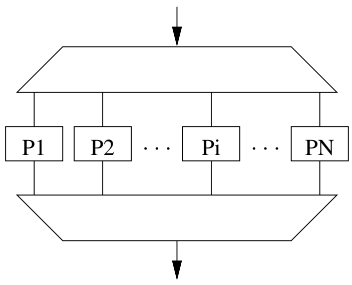
Eine Grundlegende Unterscheidung von Nutzungsformen der Parallelität, die teilweise mit den schon gemachten Unterscheidungen zusammentrifft, orientiert sich an der (Un)Gleichheit der in den parallelen Zweigen ausgeführten Codes:



(34)

Parallele Sprachstrukturen

Paralleler Block **(35)**

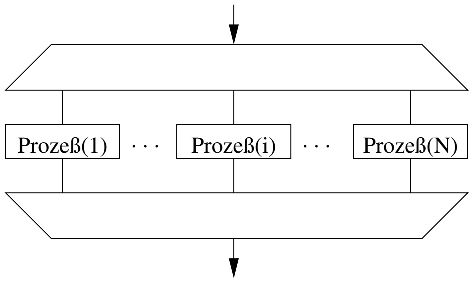



```
parbegin P1, P2,...Pi,...PN parend
```

Endet mit Synchronisation (global, barrier synchronisation).

Parallele Schleife **(36)**

Der Index-Bereich wird auf Prozesse bzw. Werke (das sind (SIMD)Rechenwerke, (MIMD)Prozessoren, Rechner) verteilt. vgl. auch (33)




```
parfor (i:=1; i<=M; i++) {prozeß(i);}
```

Häufig sind SPMD, dann prozeß(i)=prozeß und SIMD, dann prozeß nur eine Operation (für gewisse i unter Umständen keine Operation).

Beipiel datenparallel

statt

```
parfor(i:=1;i<=N;i++) {C[i]=A[i]+B[i];}
```

auch

```
forall(i:=1,N) C[i]=A[i]+B[i];
```

oder Array-Variante

```
C[1..N]=A[1..N]+B[1..N];
```

Massive Parallelität beruht weitgehend auf parallelen Schleifen in SPMD (SIMD), dank großer regulärer Datenstrukturen (partielle Differentialgleichungen, Multimediaprozesse, ...).

Auf Workstation/PC-Netzen i.a. keine übergreifende Programmiersprache, sondern Rückgriff auf (netzweite) Kommandosprache, z.B. (37)

```
run P1 on node 1
run P2 on node 2
run P3 on node 3
```

Beispiel für parallelen Block (ohne Synchronisation!) (38)

```
pid:=my_processor_id();
numproc:=number_of_processors();
for(i:=pid;i<=N;i:=i+numproc) foo(i);
```

Beispiel für parallele Schleifen über die Narrenfunktion foo(i); der obige Code ist für alle Prozessoren/Rechner gleich!

Gruppierung **(39)**

Oft ist es notwendig, eine Teilmenge von Prozessen als Gruppe anzusprechen; Bezeichner, Zahl der Mitgliedsprozesse, Index ("Rang") eines Mitgliedsprozesses, Formierung/Auflösung von Gruppen.

Statische und dynamische Parallelität **(40)**

Die Menge der parallel ausführbaren Prozesse kann zur Übersetzungszeit bekannt sein: *statische Parallelität,* oder nicht: *dynamische Parallelität.* Für Letzteres ist die Erschaffung, evtl. auch Terminierung von Prozessen zur Laufzeit erforderlich:

```
create, terminate, fork, join...
```

Im einfachsten Fall statischer Parallelität ist die Lebensdauer aller Prozesse die des parallelen Programmlaufes.

Herkömmliche sequentielle Programmiersprachen geben oft den Rahmen für die Implementation der Paradigmen. Häufig werden C oder Fortran verwendet (Grufties von 28 bzw. 43 Jahren, so etabliert sind sequentielle Paradigmen!). Drei Techniken von Bedeutung sind: Bibliotheksanschlüsse, Spracherweiterung, Compile-Zeit-Direktiven (Pragmas): (32)

	Bibliothek	Spracherweiterung	Direktiven
Beispiel	MPI (message passing interface) PVM (parallel virtual machines) (beides Nachrichtenkopplung)	Fortran 90 (datenparallel)	HPF (high performance Fortran) (datenparallel)
Vorteile	leicht implementierbar, kein neuer Compiler	erlaubt Progr. Analyse/Prüfg. und Optimierung durch Compiler	kann auf sequentiellen Rechnern einfach übergangen werden
Nachteile	keine Progr.-Analyse/Prüfg. und Optimierung durch Compiler	neuer Compiler kann schwer zu implementieren sein	im übrigen mittlere Position zwischen den beiden Anderen

Beispiele zur Eingliederung paralleler Konstrukte in Programmiersprache ([Hwang Xu 98], S.57): (33)

```
for ( i=0; i<N; i++ ) A(i]=b[i]*b[i+1];
for ( i=0; i<N; i++ ) c[i]=A[i]+A[i+1];
```

(a) A sequential code fragment

```
id = my_process_id();
p = nurnber_of_processes();
for( i=id; i<N; i=i+p ) A[i]=b[i]*b[i+1];
barrier();
for( i=id; i<N; i=i+p ) c[i]=A[i]+A[i+1];
```

(b) Equivalent parallel code using library routines my_process_id(), number of_processes (), and barrier()

```
A(0:N−1) = b(0:N−1) * b(1:N)
c = A(0:N−1) + A(1:N)
```

(c) Equivalent code in Fortran 90 using array operations

```
#pragma parallel
#pragma shared ( A, b, c )
#pragma local ( i )
{
  #pragma pfor iterate (i=0; N ; 1)
  for ( i = 0; i < N; i++ ) A[i] = b[i] * b[i+1];
  #pragma synchronize
  #pragma pfor iterate (i=0; N ; 1)
  for ( i = 0; i < N; i++ ) c[i] = A[i] + A[i+1];
}
```

(d) Equivalent code using pragmas in SGI Power C

Zur Erläuterung:

(a) Paarweiser Produkte aus b[0..N] werden auf A[0..N-1] übertragen, *dann* ebenso paarweise Summen aus A[0..N] (!) auf c[0..N-1].

Parallelisierung durch *Verteilung der Schleifendurchläufe* (parallele Iteration); Dazwischen implizite globale Synchronisation (barrier).

(b) Mit 3 Bibliotheksroutinen gelöst; jeder Prozess nimmt sich den Durchlauf mit Index my_process_id, my_process_id+p,...

(c) Radikale Reduktion der Notierung durch Array-Operationen in Fortran 90.

(d) Pragma Lösung: Das ursprüngliche C-Programm ist durch 6 Compiler-Direktiven (die Makros einsetzen/die Übersetzung steuern) aufgeweitet. Die Pragmas können bei Ausführung auf Einzelprozessoren gestrichen werden!

1.6.3 Granularität, Parallelitätsgrad, Partitionierung, Mapping, Scheduling

Granularität (Körnung) ist die Zahl der Operationen oder die Ausführungszeit, die ein Teilauftrag zwischen zwei Interaktionen mit anderen Teilaufträgen durchläuft. Das Verhältnis der Dauer eines "Korns" (Teilauftrag) zur Dauer einer Interaktion ist ein *grobes* Maß für die mögliche Effizienz der parallelen Berechnung (Grob: weil Überlappbarkeit der Interaktionen mit (anderen) Rechnungen und Grenzdurchsatz des Verbindungswerkes noch wichtig). (47)

Typische Werte: (48)

Granularität

1 Operation (10ns) nur für SIMD geeignet!

< 200 Operationen (2μs) nur für Multiprozessoren!

< 20000 Operationen (200μs) konzentrierte Multirechner

> 20000 Operationen (200μs) lokale Netze

> 20*10^6 Operationen (200ms) Fernnetze

Parallelitätsgrad (dop, degree of parallelism) ist die Zahl der aktuell kollateral ausgeführten Zweige/Prozesse des Programmlaufes. Später wesentlich stärkere Unterscheidung ($\rightarrow$ 1.7): Zahl der tätigen Werke (Rechenwerke, Prozessoren, Rechner) Zahl der aktuell nebenläufigen oder kollateralen Aktionen (Zweige, Prozesse, Teilaufträge,...). (49)

Partitionierung (partitioning) ist der Vorgang der Zerlegung eines Programmlaufes in Teilaufträge. Dabei gibt es Gestaltungsfreiheit, wie weit man nebenläufige Teilaufträge abspaltet oder mit anderen zusammenfaßt! Im Prinzip gilt: (50)

hohe Parallelität		***niedrige Parallelität***
kleine Granularität	$\Longleftrightarrow$	große Granularität
kurze Ausführungszeit		lange Ausführungszeit
aber viel Interaktion		
(kontraproduktiv)		

Lastverteilung ist die räumliche (mapping) und zeitliche (scheduling) Zuteilung von (Teil)Aufträgen zu Funktionseinheiten, insbesondere den Werken in P unseres Rechnermodells ($\rightarrow$ 1.7). (51)

Statische Verteilung geschieht *vor Auftragsstart*, verlangt *Information* über Verlauf der Auftragsbearbeitung (von Programmierer, Compilation) und über die Betriebssituation vor dem Start. (52)

Achtung! prinzipielle Berechenbarkeitsschranken! Problem Interaktivität!

Auch wenn die Auftragslast vorab bekannt ist (wäre), sind die typischen Planungsprobleme im wesentlichen nur enumerativ (d.h. praktisch kaum) oder suboptimal lösbar. vgl. LPT

Lastverteilung kann als (Hilfs)Ziel *Lastgleichheit* anstreben. Daher oft: Lastverteilung $\sim$ load balancing.

Aber es gibt viele Beispiele, die zeigen, daß Lastgleichheit kontraproduktiv ist (wegen Migrationsaufwand, Kommunikationslokalität, ...)

Lastverteilung ist stark abhängig vom *Ziel*, z.B.

- Bedienzeitminimierung
- Durchsatzmaximierung (damit Minimierung der mittleren Verweilzeit bei gegebener mittlerer Füllung!)
- Fehlertoleranz
- Programmierbarkeit

Dynamische Verteilung geschieht (ereignisorientiert, periodisch) nach Auftragsstart. Verlangt *Betriebsinformation* über z.B. (53)

- Auslastung
- Füllung (Warteschlange)
- Wartezeit

Möglichst zusätzlich über den Verlauf der künftigen Auftragsbearbeitung.

Entscheidungsverfahren über *Lastmigration*, sind abhängig von

- Betriebsinformation
- Migrationsaufwand zu Zielfunktionseinheit
- Nachbarschaften (Crossbar, Bus, Delta:keine!)(Ring, Hypercube, Baum:Ja!)
- bestehende Berechnungskontexte (Caches, Hauptspeicher)

Was muß migriert werden?

- Registerinhalte (idealer Mehrprozessor, RSM)
- Daten (SIMD)
- Daten und Code (sonst)
  - "alles"
  - Arbeitsmenge
  - eine Seite zum Starten

Bedienzeit (Ausführungszeit für *einen* Programmlauf allein auf dem Parallelrechner) (54)

- Kritischer Weg kommt auf ein Werk, wird nicht durch Migration auf andere unterbrochen!

- Dazu nebenläufige Teilaufträge werden unter Beachtung der Kommunikation und der Rechenlast verteilt
- (Steht's prüfen ob Verteilung überhaupt lohnt!)

Übrigens: Es gibt Beispiele (bei Nichtverdrängbarkeit), wo man zeigen kann, daß noch nicht mal die intuitiv überzeugende Regel: Wenn Arbeit da ist, gib sie einem freien Prozessor! optimal ist (und zwar auch unter Vernachlässigung allen Interaktions- und Transportaufwandes!).

Übrige Ziele von Lastverteilung: (55)

Durchsatz (Füllung > 1; Durchsatzmaximierung minimiert mittlere Verweilzeit (Little))

- Auslastung wird bereits durch Überlagerung von Aufträgen gehoben!
- Hauptziel der Lastverteilung: Engpassbeseitigung, vgl. Routing in Netzen.
- Nebenziel: Bedienzeitgünstige Verteilung (niedrige Verweilzeit bei kleiner Füllung)

Fehlertoleranz

- Verteilung verschiedener Versionen auf unabhängige Einheiten
- durch Verteilung gute Diagnose- und Recoverystützen

Programmierbarkeit

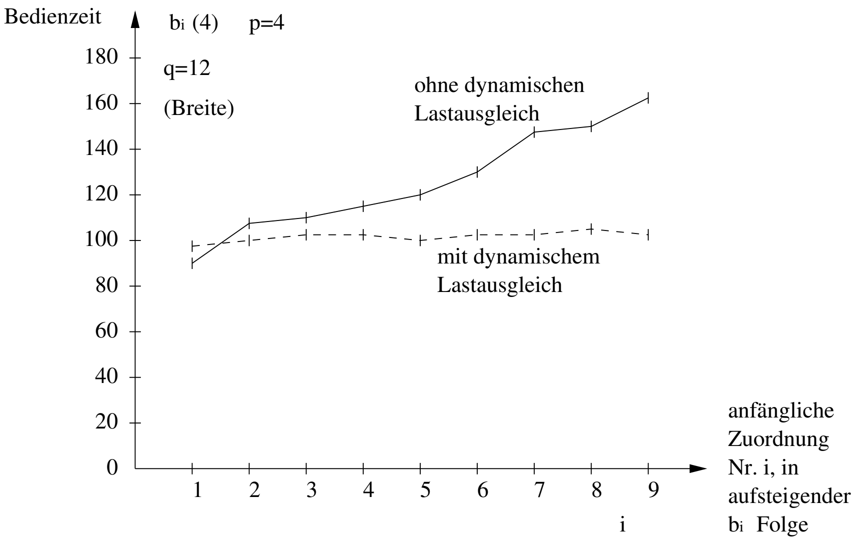
- keine Information vom Programmierer verlangen
- Verteilung soll Testbarkeit, Ergebnis nicht beeinflussen

Dynamischer Lastausgleich [Ludwig T 92] (56)

Die Auftragsbearbeitung wird laufend auf Indikatoren für schlechte Betriebsmittelnutzung geprüft. [Ludwig T 92] beschreibt u.a. ein Experimentiersystem zum Studium von Lastausgleichsverfahren auf Intel IPSC/2 (80386-Hypercube-Multirechner). Das System:

- mißt ausgesuchte Indikatoren (z.B. Freizeiten, Knotenwarteschlange, globalen Ankunftsprozeß, Kommunikationsdurchsätze...)
- bewertet sie
- plant Ausgleichsmaßnahmen
- führt die Maßnahmen durch: Teilauftragsverschiebung (Speicherseiten für Code, Daten)

Typisches Resultat aus [Ludwig T 92]



(57)

Die Experimente zeigen Beispielhaft:

1. Ohne dynamischen Lastausgleich hängt die Bedienzeit b kritisch von der anfänglichen Zuordnung ab (im Beispiel etwa 1:2).
2. Dynamischer Lastausgleich kürzt fast überall die Bedienzeit.
3. Bei sehr guter anfänglicher Anordnung kann dynamischer Lastausgleich die Bedienzeit erhöhen.

[Ludwig T 92] zeigt auch:

- grobes Meßraster (0,5...1s) reicht aus
- CPU-Leerzeit und CPU-Warteschlange sind brauchbare Indikatoren

Interaktionsmuster **(62)**

$1:1$ (Punkt zu Punkt) ist in Netzen das Schema überhaupt

$1:n$ Multicast, Broadcast (gleiche Information), scatter (verschiedene Informationen)

$m:1$ gather, reduction (unter Zusammenziehung, z.B. "wenigstens einer > 0?", Summenbildung o.Ä.)

$m:n$ total exchange (jeder schickt jedem eine verschiedene Information), permutation (jeder Sender hat einen eigenen Empfänger), shift.

Die Interaktionsmuster sind hier für Kommunikation formuliert, gelten aber ebenso für Synchronisation!

Interaktion ist erforderlich und kann, sowohl mit gemeinsamen Variablen als auch mit Nachrichten (Botschaften), realisiert werden.

1.6.4 Interaktion

Die einzelnen Teilaufträge einer Berechnung treten in Wechselwirkungen (Interaktionen). Diese sind: (58)

***Synchronisation*:** Aufbauen von Wartebedingungen zwischen den ansonsten asynchronen Teilaufträgen, anhalten der Berechnung und freigeben bei erfüllter/sich erfüllender Bedingung. wait(e), signal(e)/send(e); atomic...end; (Ausschluß jeder Unterbrechung; alles oder nichts); critical region...end; barrier(<Gruppenbezeichnungen>). (59)

***Kommunikation*:** Interaktion besteht im Transport einer Nachricht, ist mehr oder weniger für Sender und/oder Empfänger synchronisierend, vgl. Vorlesung "Verteilte Anwendungen". (60)

***Aggregation*:** schwach synchronisierende globale Kommunikation von n Sendern auf einen Empfänger, z.B. n Ergebnisse an Zentrale. (61)

Kompetitive und Kooperative Interaktion

Interessant ist die folgende Gegenüberstellung von kompetitiver Interaktion von Prozessen/Threads unter einem Betriebs/Kommunikationssystem (Kampf um knappe Betriebsmittel) und innerhalb eines parallelisierten Jobs auf einem Parallelrechner (Arbeitsteilung (auch bei Wettbewerbsparallelität!)), vgl. [Hwang Xu S. 85] (62)

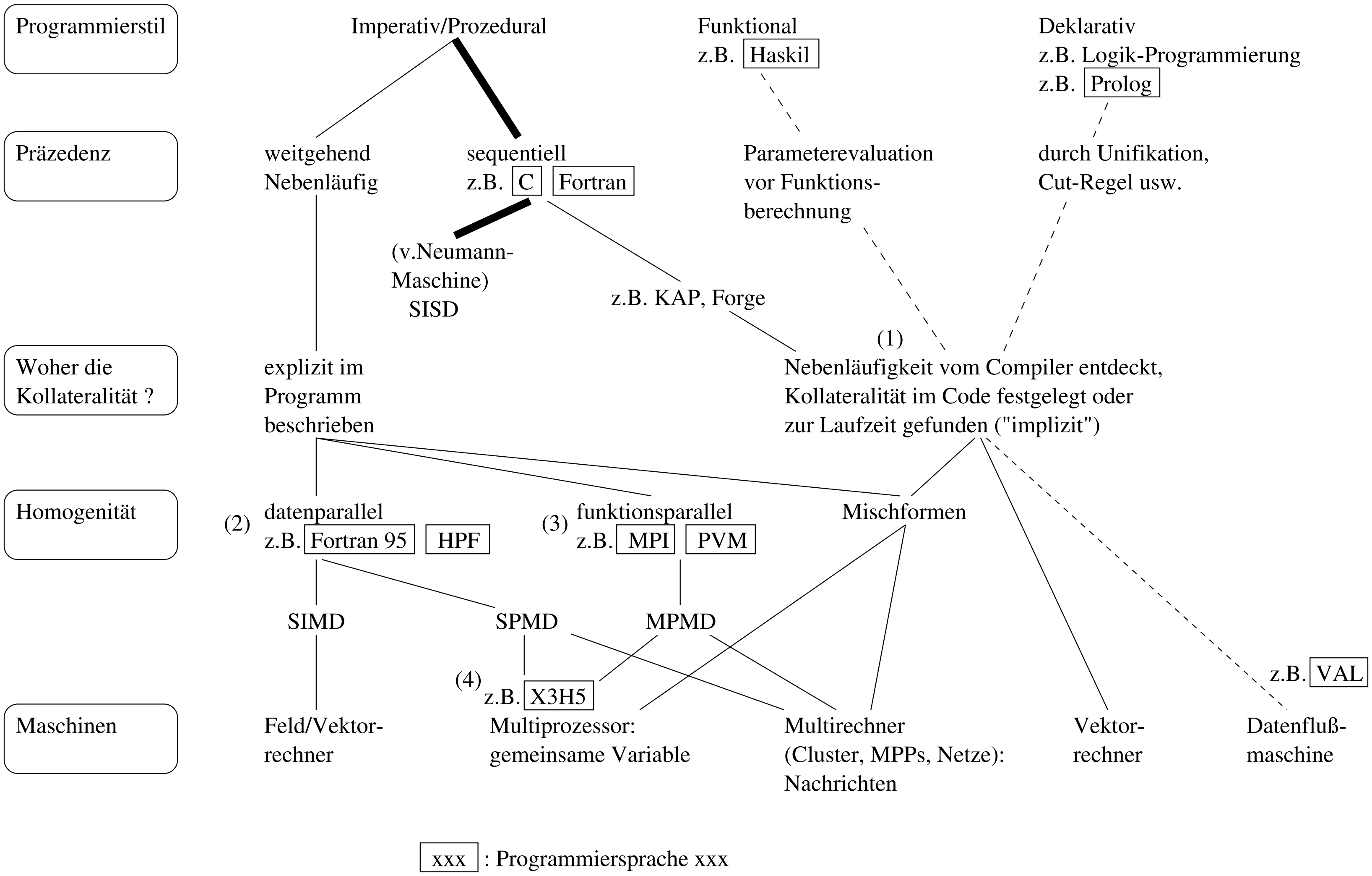
	kompetitive Interaktion	kooperative Interaktion
Beispiele	Betriebssystem Kommunikationss.	Job auf Parallelrechner
Granularität	grob heterogen	fein, mittel homo/heterogen
typische Interaktionen	kritische Regionen, Producer-Consumer Reader/Writer Unterbrechung!	barrier, Broadcast, Permutation, Shift, Reduction, Work Pool
Elementaroperationen	Lock, Semaphor, Event, Region, Monitor	barrier, fetch and add
Prozeßsystem	endet nicht, nicht funktional	terminiert, funktional

1.6.5 Die vier Paradigmen der Parallelrechnerprogrammierung

Wie schon in (31), werden in der Parallelrechnerprogrammierung vier Paradigmen unterschieden

ohne explizite Parallelität		(hauptsächlich Vektorrechner)	(1)
mit expliziter Parallelität	$\longrightarrow$	datenparallel	(2)
	$\longrightarrow$	funktionsparallel (Nachrichtenkommunikation)	(3)
	$\longrightarrow$	codeparallel (gemeinsame Variable)	(4)

Die vier stehen unglücklicherweise auf durchaus verschiedenen begrifflichen Niveaus; weshalb wir unsere bisherigen Klassifikationen nochmals integrieren: (63)


Programmierstil
Präzedenz
Woher die Kollateralität ?
Homogenität
Maschinen
Imperativ/Prozedural
Funktional
z.B. Haskil
Deklarativ
z.B. Logik-Programmierung
z.B. Prolog
weitgehend Nebenläufig
sequentiell
z.B. C Fortran
Parameterevaluation vor Funktions- berechnung
durch Unifikation, Cut-Regel usw.
(v.Neumann- Maschine) SISD
z.B. KAP, Forge
(1)
explizit im Programm beschrieben
Nebenläufigkeit vom Compiler entdeckt, Kollateralität im Code festgelegt oder zur Laufzeit gefunden ("implizit")
(2)
datenparallel
z.B. Fortran 95 HPF
(3)
funktionsparallel
z.B. MPI PVM
Mischformen
SIMD
SPMD
MPMD
(4)
z.B. X3H5
z.B. VAL
Feld/Vektor- rechner
Multiprozessor: gemeinsame Variable
Multirechner (Cluster, MPPs, Netze): Nachrichten
Vektor- rechner
Datenfluß- maschine
xxx : Programmiersprache xxx

Vergleichende Bewertung der vier Paradigmen (vgl. [Hwang Xu 98, S. 625]) durch den Programmierer: (64)

Kriterium	(1) nicht explizit	(2) daten-parallel	(3) Nachrichten-kommunikation	(4) gemeinsame Variable
Leichtigkeit der Parallelisierung	++	+	-	o
Allokation (durch Programmierer)	++	+	o	++
Kommunikation	++	+	-	+
Synchronisation	++	++	o	-
Terminierung	++	++	+	o
Funktionalität (Determiniertheit)	++	++	+	o
Korrektheit	++	+	+	o
Effizienz	-	++	o	+
Portabilität	++	+	o (MPI-, PVM+)	o
technische Beherrschung (Tools)	-	+	+	o
Beipiele für Sprachen/Compiler	KAP, Forge (Compiler)	Fortran 95, HPF	MPI, PVM	X3H5

zu den Paradigmen:

Für den *impliziten Parallelismus* als Basis der (automatischen) Programmierung spricht neben den angegebenen Kriterien auch noch

- der immense Vorrat verstandener und erprobter Software
- die Ausbildungssituation
- die guten Tools und Methoden für die Entwicklung sequentieller Software

Für *Datenparallelität* als Parallelisierungsansatz spricht noch:

- wichtige Anwendungsklasse (z.B. partielle Differentialgleichungen)
- Anwendungsklasse liefert hohe und sehr hohe Parallelität
- auf Vektorrechnern sind diese Probleme ohne Allokation (Abwicklung in der Zeit!) lösbar, abgesehen von Rücksichten auf zugriffsgünstige Adressierungsfolgen im Speicher
- unkomplex, da ein Prozeß, einfaches Synchronisationsschema

Für *Nachrichtenkommunikation* spricht:

- Strukturierung kann förderlich sein (Allokation, Scheduling, explizite Synchronisation und Kommunikation)

- *aber* i.a. nur für grobe Granularität brauchbar
- hohe Maschinenparallelität gibt es nur auf Multirechnern

Für *gemeinsame Variable* spricht:

- Nähe zur sequentiellen Programmierung
- keine Allokation, kein Verschiebungaufwand bei Umverteilung (es ist lediglich ein anderer Prozessor zuzuteilen!)
- *aber* (bei mangelhafter Synchronisation) Gefahr von Nichtdeterminiertheit
- daß gemeinsame Variablen die Programmentwicklung erleichtern, verglichen mit Nachrichtenkommunikation, ist unbelegte Parallelrechnerfama.

1.6.6 Programmierung ohne explizite Parallelität

KAP: Kap ist ein (kommerzieller) Compiler, der Programme für einen Multirechner aufbereitet. Diese Techniken beruhen auf einer Analyse des Kontrollflusses und des Datenflusses im Programm. Die "Privatisierung" von Variablen ist eine wichtige Technik (vgl. Beispiel in 1.5.1). Untersuchungen über die Wirksamkeit [Hwang Xu 98, S. 618]) zeigen, daß der Compiler etwa um den Faktor 10 gegenüber "manueller" Transformation des Algorithmus zurück bleibt.

Dagegen sind sogenannte vektorisierende Compiler für Vektorrechner, die Fortran 77 oder HPF oder C verarbeiten, sehr erfolgreich (seit 1980) und haben wesentlich zu dem Erfolg dieser Maschinen beigetragen. Allerdings nutzen sie in der Hauptsache Schleifen über Feldern zur Parallelisierung aus, was einfach ist.

1.6.7 Datenparallelität: Fortran 90/95 und High Performance Fortran

- *Ein* Prozeß in *einem* Adressraum (aber, abhängig von Maschinenparallelität und Array-Abmessungen, dynamisch viele Instantiierungen der Kollateralen Zweige, die gleich oder bedingungsabhängig sein können).
- Synchronisation statementweise, implizit, global; bei SIMD sogar befehlsweise, durch Globalsteuerung der Maschine (k Rechenwerke oder k Vektorelemente) gegeben.

Fortran 90/95: Erweiterung von Fortran um Operationen auf Arrays und Teilarrays: *elemntare*, höhere (z.B. Matrixoperationen, Aggregationen). Diese werden parallelisiert.

High Performance Fortran HPF: Obermenge zu Fortran 90/95: komplexere Array-Operationen, flexiblere Indexmangen statt Schleifen, Direktiven für Verteilung, Anpassung an den Zielrechner.

1.6.8 Nachrichtenkommunikation: MPI und PVM

MPI (Message Passing Interface) und PVM (Parallel Virtual Machine) sind die weitaus wichtigsten aus den zahlreichen Sprach/Programmiersystemen für Nachrichtenkommunikation (Message Passing). MPI ist standardisiert. Eine Integration beider Entwicklungen erscheint möglich. Beide werden von allen Parallelrechnerherstellern unterstützt.

MPI ist eine Programmbibliothek von etwa 200 Funktionen, Vorzugsweise an Fortran und C anschließbar. MPI benutzt statische und (neuerdings) dynamische Prozesse, keine Threads. MPI besitzt zahlreicher Synchronisations-, Kommunikations-, und Aggregationsoperatoren. MPI baut auf 4 orthogonalen Konzepten auf: Nachrichten Typen, Prozessgruppierungen, Kommunikationsoperatoren, virtuelle Topologie.

PVM ist ein autonomes Programmsystem. Es integriert Programme, die in Fortran, C oder Java geschrieben sind. PVM erlaubt die Konstruktion einer virtuellen Maschine, die auf heterogene physische Maschinen mit unterschiedlichen Betriebssystemen abgebildet wird.

1.6.9 Gemeinsame Variable: X3H5 - Ansatz

Die Programmierung von Parallelprogrammen auf Mehrprozessorsystemen unter Verwendung gemeinsamer Variablen kann im Grundsatz mit den vorhandenen sequentiellen Sprachen (mit maschinennahen Prozeduren und Systemaufrufen) geschehen. Die Hersteller stellen verschiedene proprietäre Programmiersysteme ergänzend bereit (z.B. SGI Power C, Cray MPP).

Es existiert ein ANSI-Standard (ANSI: American National Standardisation Institute), *X3H5*, der von den Herstellern in dieser Form nicht unterstützt wird, aber in die von vielen Herstellern unterstützte OpenMP-Initiative übergeht. *OpenMP* bietet: schlichte Parallelitätskonzepte (Block, Schleife), Behandlung verwaister Sohn-Prozesse, Gültigkeitsregeln, Persistenz, Synchronisation, kritische Regionen, Locks, Atomizität. Wegen des Overheads für grobe Granularität geeignet.

1.7 Leistung von Parallelrechnern

vgl. [Jessen E 98]: Vorlesungsskript "Leistung von Parallelrechnern"

1.7.1 Begriffe

Wiederholung des Wichtigsten aus der Rechnerarchitekturvorlesung:

Verweilzeit (delay, latency, ...): (physikalische) Zeitdauer von Übernahme/Übergabe eines Auftrags von/an/durch die erledigende Funktionseinheit: "y". (65)

Füllung (population): Zahl der Aufträge, die augenblicklich ihre Verweilzeit in der Funktionseinheit zubringen: "f". (66)

Bedienzeit (service time): Verweilzeit eines Prozesses bei Füllung 1 (er ist allein!): "b". (67)

Kapazität (capacity): maximale Füllung einer Funktionseinheit: "k". (68)

Exclusive Funktionseinheit : Kapazität = 1.

Durchsatz (throughput): Zahl der von der Funktionseinheit in einem Zeitintervall erledigten Aufträge, geteilt durch die Dauer des Intervalls: "d". (69)

Grenzdurchsatz (bandwidth): maximaler Durchsatz der Funktionseinheit: "c". (70)

Auslastung (utilisation): Quotient von tatsächlichem Durchsatz zu Grenzdurchsatz: "ρ". [0...1] (71)

Little's Formel : Die Füllung einer Funktionseinheit sei am Anfang und am Ende eines Betrachtungszeitraums Null. Für die Mittelwerte von *Füllung* $(\overline{f})$ und *Verweilzeit* $(\overline{y})$ und für den *Durchsatz* d gilt:

$$\overline{f} = \overline{y} * d \tag{72}$$

$E[F]$, $E[Y]$ und $E[D]$ seine zeitkonstante, beschränkte Erwartungswerte der Zufallsvariablen F, Y, D (Füllung, Verweilzeit, Durchsatz). Dann gilt

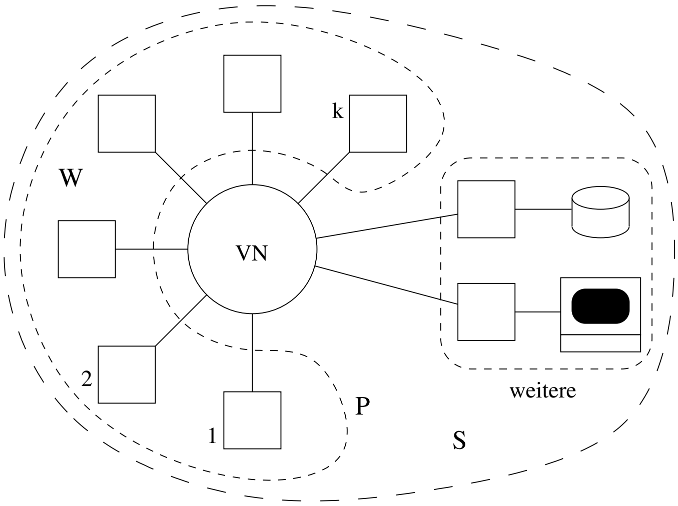
$$E[F] = E[Y] * E[D].$$

Belegung (occupancy): einer Funktionseinheit ist die Summe der Bedienzeiten, die der Auftrag in v_i "Besuchen" der Funktionseinheit i in Anspruch nimmt. v_i heißt *Besuchszahl* (visit ration). Bei jedem Besuch löst der Besuch einen *Teilauftrag* aus. (73)

Engpaß (bottleneck): ist die Funktionseinheit i eines Systems, bei der der Quotient $\frac{c_i}{v_i}$ minimal ist (dabei ist c_i Grenzdurchsatz von i in Teilaufträgen und v_i mittlere Besuchszahl von i). Äquivalent: Engpaß ist die höchstausgelastete Funktionseinheit: $c_s = \min\left[\frac{c_i}{v_i}\right]$. (74)

1.7.2 Einfaches Parallelrechnermodell und Speedup

Unser einfachstes Modell enthält:



(75)

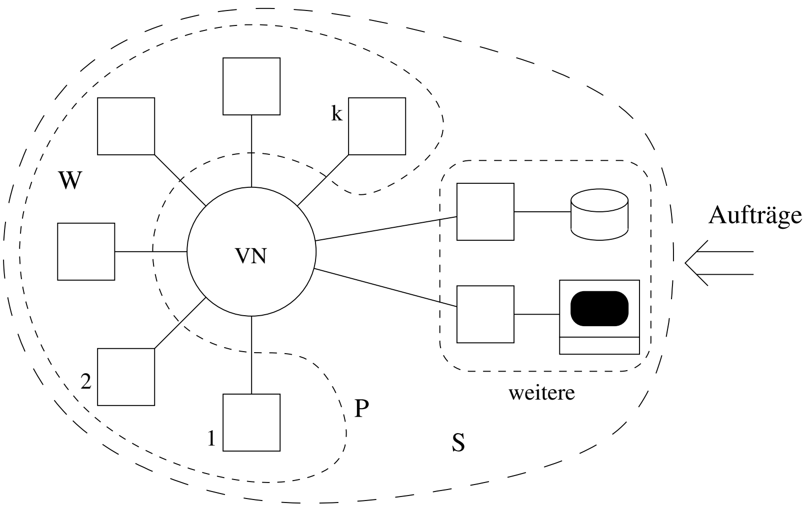
Funktionseinheit P mit k exklusiven gleichen Werken W (Rechenwerk beim Feldrechner, Zentralprozessor beim Multiprozessor, Rechner beim Multirechner), von denen einem Auftrag an das Gesamtsystem S gerade p zugewiesen werden. Typisch ($p \leq k$, aber ($p > k$ durch Multiplexing möglich ("virtual processors", falls die Werke Prozessoren sind).

Funktionseinheit VN (Verbindungsnetz)

"Weitere" Funktionseinheiten (z.B. E/A), die aber nicht mit Parallelität p betrieben werden.

Dem Modell fehlen:

- für Multiprozessoren: Speichermoduln (parallel!)
- für Feldrechner: Leitwerk



Aufträge an S sind gekennzeichnet durch:

Besuchszahlen (Teilaufträge) v_W, v_{VN}, $(v,\ \ldots)_{weitere}$

Bedienzeiten (Teilaufträge) $b_W, b_{VN}, (b, \ldots)_{weitere}$

Grenzdurchsätze (Teilaufträge)

Auslastungen bei gegebenem Durchsatz $\rho_W, \rho_{VN}, (\rho, \ldots)_{weitere}$

Die Last ist eine *Belegungsmenge* $\{\overline{x_i} = \overline{v_i} * \overline{b_i}\}$ mit einem (u.U. komplexen) Präzedenzschema. Besonders wichtig "Arbeit" w (work) der "parallelen Werke W, $w = \overline{x_W} = \overline{v_W} * \overline{b_W}$, z.B. CPU-Zeitsumme des Auftrags an S.

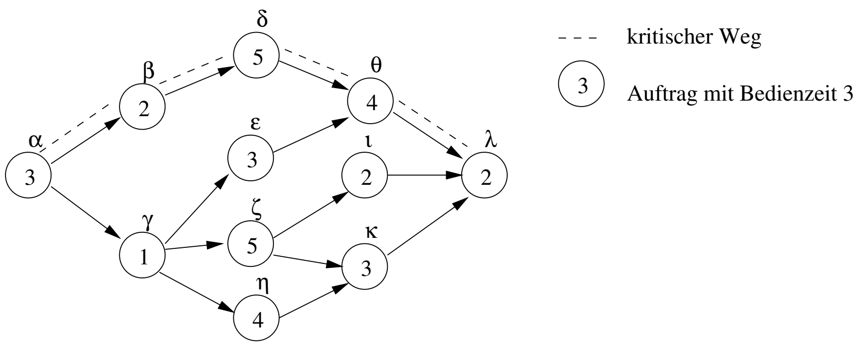
Aufträge an S bestehen (u.a.) aus nebenläufigen Teilaufträgen an die Werke W in P.

Unterscheide:

k	*Kapazität* (Zahl der Werke) von P, konstant
$p(t)$	dem Auftrag *zugestandene Parallelität* i.a. $p(t) \leq k$, aber $p(t) > k$ möglich! meist $p(t) = p$ konstant (76)
$r(t)$	vom Auftrag aktuell *genutzte Zahl von Werken* W bzw. Zahl der aktuell bedienten Teilaufträge. Mittelwert $\overline{r} = \frac{w}{b_p(p)}$ (77)
$f_p(t)$	aktuelle *Füllung* von P in *Teilaufträgen*, diese können rechenbereit warten oder bedient werden, also $f(t) \geq t(t)$
$q(t)$	*Breite* des Auftrags, $q(t) = r(t) = f(t)$ falls P so groß, daß nie ein Teilauftrag auf das Freiwerden eines Werkes warten muß $q(t) \in (q_{min}, q_{max})$. $q_{min} \geq 1$, falls kein Warten auf VN, weitere. Also $q(t) = r(t) = f(t)$ falls $p \geq q_{max}$. (78)
ρ_p	*Auslastung von* P, bei Einprogrammbetrieb ist $\rho_p = \frac{\overline{r}}{k}$, sonst $\rho_p = \frac{\overline{r}}{p}$. ρ_p gibt relative Zahl der genutzten weiteren an. (79)

—

Beispiel:

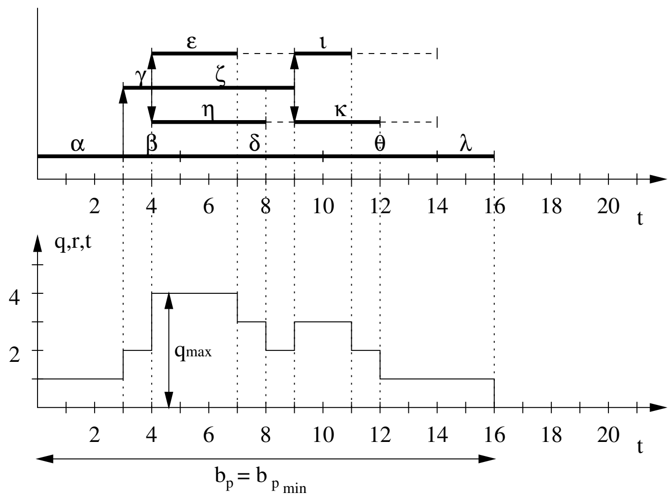


(80)

Auftrag aus $v_w = 11$ Teilaufträgen $\alpha \ldots \lambda$, Bedienzeiten: $w = \sum_{i=1}^{11} b_i = 34$, Kritischer Weg bestimmt $b_{p_{min}}$

Ausführung unter $p \geq 4$

$q_{max} = 4$ "Gantt-Diagramm", $q = r = f_p$

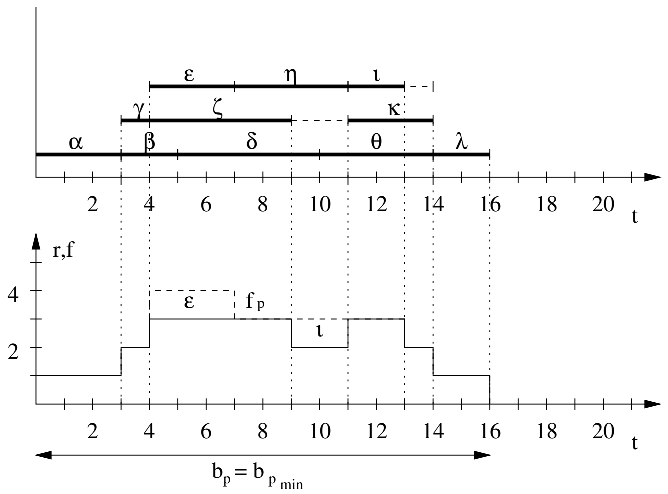


$\overline{r} = \overline{q} = \frac{w}{b_{p_{min}}} = \frac{34}{16} = 2,125$

bei $p = 4$, $\rho_p = \frac{\overline{r}}{p} = \frac{2,125}{4} = 0,53$

Ausführung unter $p = 3$

$q \neq r \neq f_p$, $f_p = 2,44$

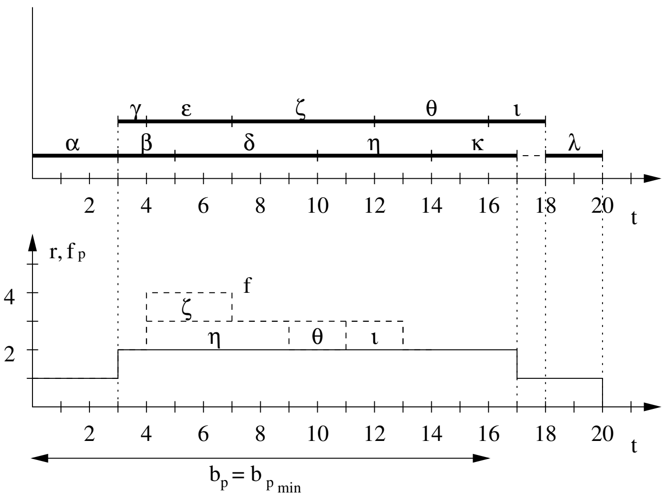


$\overline{r} = \frac{w}{b_{p_{min}}} = \frac{34}{16} = 2,125$ (Glück!)

bei $p = 3$, $\rho_p = \frac{\overline{r}}{p} = \frac{2,125}{3} = 0,71$

Ausführung unter $p = 2$

$q \neq r \neq f_p,\ f_p = 2,45$



$\overline{r} = \frac{w}{b_{p_{min}}} = \frac{34}{20} = 1,7$

bei $p = 2$, $\rho_p = \frac{\overline{r}}{p} = \frac{1,7}{2} = 0,85$

Die Füllung f enthält auch die auf Zuteilung eines Werkes wartenden Teilaufträge.

—

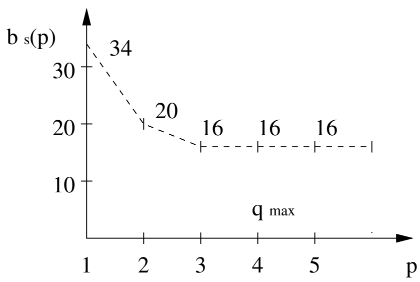
Für die Beurteilung des parallelen Rechnens sind Bedienzeit und Grenzdurchsatz ebenso natürliche Kennzeichen wie für konventionelles Rechnen.

Wir ermitteln $b_s(p)$, $\rho_s(p)$, $c_s(p)$ für das Beispiel (58).

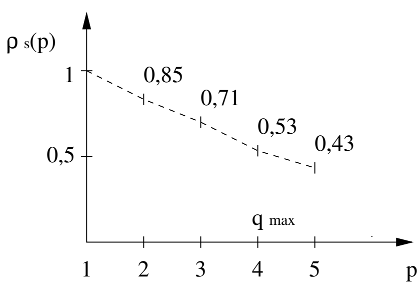
—

Beispiel: S hier P (81)

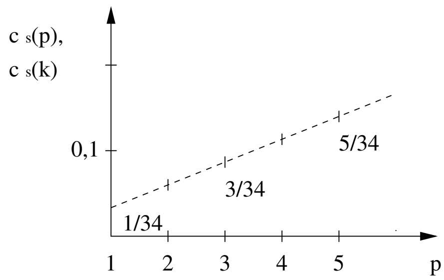
Bedienzeit $b_s(p) = b_p(p)$ für $p = 1$ ist $b_s(1) = w = 34$



Auslastung $\rho_s(p) = \rho_p(p)$ bei Einprogrammbetrieb, vgl. (58)



Grenzdurchsatz $c_s(p) = c_s(k) = c_p(p)$



Wir fahren die p Werke (k Werke) unter Mehrprogrammbetrieb. Jeder Auftrag erfordert $w = 34$. p bzw. k Werke können $\frac{p}{w}$ bzw $\frac{k}{w}$Aufträge je Zeiteinheit erledigen.

—

Tatsächlich hat sich $c(k)$ (oder $c(p)$) nicht eingebürgert (da an Mehrprogrammbetrieb bisher wenig Interesse), und $b_s(p)$ wird tatsächlich meist relativ zu $b_s(1)$ betrachtet, und

$$s(p) = \frac{b_s(1)}{b_s(p)} \tag{82}$$

wird suggestiv *Speedup* genannt.

Die Speedupdefinition enthält einige Probleme, wie wir sehen werden:

- langsame Algorithmen können besseren Speedup liefern als schnelle
- langsame Werke können höheren Speedup liefern als schnelle

Über Speedup und Verbesserungsvorschläge sind massenhaft Artikel geschrieben worden, die den Speedup aber nicht umbringen konnten. Vgl. Hinweis "(4)" in der Literaturliste.

Schon aus der Bedienzeitdefinition folgt, daß Speedup nur bei Einprogrammbetrieb (oder bei Partitionierung von P, so daß p Werke dem Auftrag allein zugeteilt sind) ein sinnvoller Begriff ist.

Speedup

$$s(p) = \frac{b_s(1)}{b_s(p)} \qquad p\text{: zugestandene Parallelität}$$

für $b_s(1)$ kann man heranziehen:

Bedienzeit des nebenläufigen Algorithmus bei $p = 1$, dann *relativer Speedup*. Beurteilt wird das Parallelrechensystem unter der durch den nebenläufigen Algorithmus definierten Last.[1]

Bedienzeit des schnellsten sequentiellen Algorithmus, dann *absoluter Speedup*. Beurteilt wird das Parallelrechensystems unter der durch den nebenläufigen Algorithmus definierten Last und die Güte des nebenläufigen Algorithmus.

[1] Also hat ein durchaus schlechter paralleler Algorithmus, einen prächtigen relativen speedup, wenn er nur bei $p = 1$ genügend viel schlechter als schon bei $p > 1$ ist.

Effizienz

Für die nachfolgende Definition setzen wir Einprogrammbetrieb und Beauftragung von nur P voraus.

Effizienz $e(p)$ ist $e(p) = \frac{s(p)}{p}$ (83)

Es besteht folgende Beziehung zu ρ_p, $\overline{r}$: wobei wir (realistisch) $w = w(p)$ annehmen:

$$\rho_p = \frac{\overline{r}}{k} = \frac{\left(\frac{w(p)}{b_p(p)}\right)}{k} = \frac{w(p)}{p * b_p(p)} * \frac{p}{k} \text{ wegen (79)}$$

mit $b_p(p) = \frac{w(1)}{s(p)}$ (wegen (82), Auftrag nur an P, d.h. $b_s = b_p$, $b_s(1) = w(1)$!)

folgt

$\rho_p = \frac{w(p)}{w(1)} * \frac{p}{k} * \frac{s(p)}{p} = \frac{w(p)}{w(1)} * \frac{p}{k} * e(p)$ (84)

entsprechend

$\overline{r} = \frac{w(p)}{w(1)} * s(p) = \frac{w(p)}{w(1)} * p * e(p)$ (85)

und

$s(p) = \frac{w(1)}{w(p)} * k * \rho_p = \frac{w(1)}{w(p)} * \overline{r}$ (86)

und

$e(p) = \frac{w(1)}{w(p)} * \frac{k}{p} * \rho_p = \frac{w(1)}{w(p)} * \frac{\overline{r}}{p}$ (87)

sehr oft wird nur der simple Fall $p = k$, w konstant betrachtet:

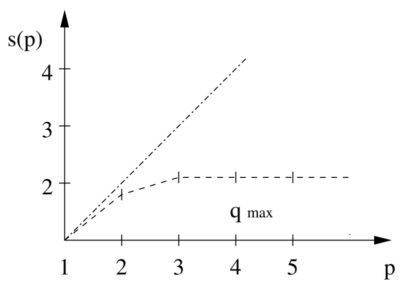
$\rho_p = e(p), \qquad s(p) = \overline{r}$ (88)

—

Beispiel wie (80), (81) (89)

p	$b_s(p)$	$s(p)$
1	34	1
2	20	1,7
3	16	2,125
4	16	2,125
5ff	16	2,125

relativer Speedup



im Beispiel ist:

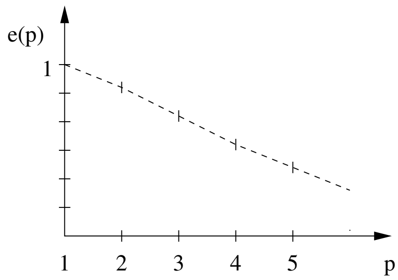
$s(1) = 1$ (das gilt immer)

$w(p) = w$ konstant (90)

$s(p) = \overline{r}$ gilt, falls nur P beauftragt und w konstant) (91)

$s_{max} = \frac{w}{b_{p_{min}}}$ (gilt falls nur P beauftragt) (92)

Effizienz $e(p) = \frac{s(p)}{P}$



—

Nun ordnen wir die Voraussetzungen unserer Überlegungen besser !

1.7.3 Modellklassen und einfache Resultate Gesichtspunkte für die Einteilung von Parallelrechnermodellen

A) *Einprogrammbetrieb/Mehrprogrammbetrieb*

Fast alle Modelle betrachten (implizit) Einprogrammbetrieb. So auch unser Beispiel (80). (93)

B) *global produktive Bedienung in* P

(schon früher: produktiv ist Bedienstrategie, die nie bei freier Bedieneinheit einen Auftrag warten läßt) Global produktive Bedienung heißt: nie wartet ein Teilauftrag, wenn ein Werk frei ist. (94)

Ob und wie aufwendig global produktive Bedienung erreichbar ist, hängt ab von:

- können alle Werke auf alle Teilaufträge gleich zugreifen, und es gibt keine werkslokalen Berechnungskontexte? (einfach!)
- oder nicht ? (schwierig!)
  - ist die jetzige und künftige Teilauftragsverteilung bekannt ? (Strategie wirksam!)
  - oder nicht ? (Strategie nicht wirksam!)
  - ist die Auftragsverlagerung schnell durchführbar (dann nützlich)
  - oder nicht (dann evtl. bei Abschluß überholt!)

B ist auch i.a. nicht vorauszusetzen!

C) Die Belastung betrifft *nur die parallele Einheit* P. So auch in Beispiel (80). Wenn auch andere (VN, $Weitere$). Dann ist zu unterscheiden: (95)

- vollständig überlappt mit b_p (dann keine Auswirkung auf die Gesamtbedienzeit b_s, vgl. (67). Allerdings u.U. Auswirkungen auf c_s, falls VN oder $Weitere$ Engpaß).
- nicht (vollständig) überlappt mit b_p (dann Erhöhung der Gesamtbedienzeit b_s).

D) Die zugestandene *Parallelität* p ist *praktisch unbeschränkt*: (96)

$$\begin{array}{lll} p \geq q_{max} & \Rightarrow\ r = q = f & \text{falls (B)} \\ & \Rightarrow\ \overline{r} = \overline{q} = \overline{f} = \frac{w}{b_{p_{min}}} & \text{falls (C) nur } P \end{array}$$

sonst

$$\begin{array}{lll} p < q_{max} & \Rightarrow\ f \geq r & \text{d.h. es warten u.U. Teilaufträge} \\ & \Rightarrow\ \overline{r} = \frac{w}{b_p} & \text{falls (C) nur P} \end{array}$$

E) *Die Belegungen* $w = v_p * \overline{b_p}, v_{VN} * \overline{b_{VN}}, v_{Weitere} * \overline{b_{Weitere}}$

- sind gegenüber p *konstant* (so auch Beispiel (58))
- wachsen mit p (typisch!) wegen Prozesskreierung/Terminierung
  - CPU-Zeit für Synchronisation, Kommunikation, virtuell globalen Speicher, dynamische Lastverteilung
  - CPU-Zeit wegen Redundanz, Informationsarmut des nebenläufigen Verfahrens
  - VN-Zeit für Synchronisation, Kommunikation, virtuell globalen Speicher, dynamische Lastverteilung

  (alle diese Effekte können mit r wachsen, d.h. mit p, solange $p \leq q_{max}$)
- *nehmen ab* mit p. Das ist möglich bei einigen nichtdeterministischen Algorithmen , wenn die kürzesten Bedienzeiten klein gegenüber den mittleren sind (vgl. (46)). Betroffen ist primär w Damit evtl. auch $v_{VN} * \overline{b_{VN}}, v_{Weitere} * \overline{b_{Weitere}}$

Untersuchung von $b(p)$, $c(p)$, $s(p)$ im Rahmen von Modellen der Gesichtspunkte A...E
Es sei

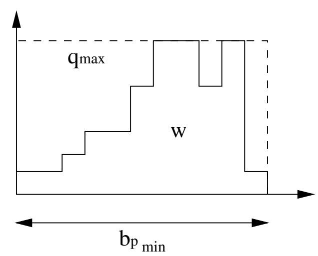
A: Einprogrammbetrieb

B: global produktive Bedienung in P

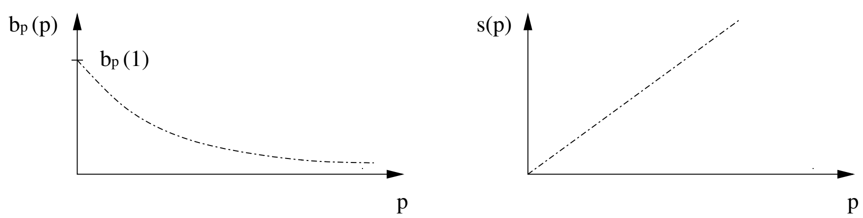
C: nur P wird beauftragt

D: $p \geq q_{max} \vee \; p < q_{max}$ und (d.h. alle Fälle)

E: $w(p)$ konstant, steigt, fällt (d.h. alle Fälle); endlich



Annahme $b_{p_{min}} = 0$, d.h. die Bedienzeit des längsten Weges ist Null. Dann ist
$q_{max} = \infty$, da $w \leq q_{max} * b_{p_{min}}$ (97)
wenn $b_{p_{min}} = 0$, dann ist ideale Verteilung auf p Werke möglich:

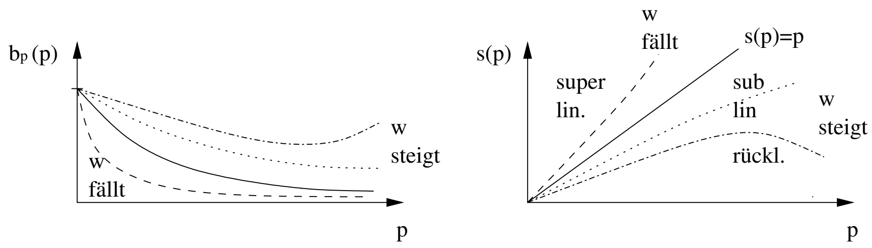


$b_p(p) = \frac{w}{p}$ $\quad w$ konstant (98)

$s(p) = \frac{b(1)}{b(p)} = \frac{w}{\left(\frac{w}{p}\right)} = P$ "linearer Speedup" (99)

(Annahme $b_{p_{min}} = 0$, ideale Verteilung (66)):

Setzen wir $w(p)$ fällt/steigt voraus, dann erhalten wir *superlinearen/sublinearen* oder sogar *rückläufigen* Speedup. (100)



Später sehen wir, daß auch z.B. das Verbindungswerk rückläufigen Speedup herbeiführen kann. Entsprechend wird bei k Werken, die alle zugeteilt werden $(p = k)$ der Grenzdurchsatz beeinflußt. (101)

$$c_p(p) = \frac{k}{w(k)} \begin{cases} \text{steigt superlinear} \\ \text{linear} \\ \text{schwächer als linear} \end{cases} \qquad (103)$$

Die Effizienz $e(p)$ ist entsprechend $<$ oder ≥ 1. Die Auslastung $\rho_p(p)$ ist dagegen wegen der Voraussetzung $b_{p_{min}} = 0$ und (94) steht's 1 !

Nachbemerkungen zu superlinearen Speedup.

Zunächst sieht es so aus, als sei *absoluter* superlinearer Speedup nicht möglich.

Denn angenommen

$$s(p) = \frac{b_p(1)}{b_p(p)} > p$$

Dann tausche man den für $p = 1$ benutzten Algorithmus aus gegen eine processor sharing-Simulation des Parallelrechners aus p Werken, bei jeder der p Zweige der parallelen Rechnung mit $\frac{1}{p}$ Rechenleistung verfolgt wird; es ist dann $b_{p_{ProcSh}}(1) = p * b_p$, also der absolute Speedup

$$s(p) = \frac{b_{p_{ProcSh}}(1)}{b_p(p)} = \frac{p{*}b_p(p)}{b_p(p)} = p \; ! \qquad (104)$$

Realistisch ist das Prozessorsharing aber (wegen der Umschaltverluste!) deutlich langsamer als nur um den Faktor p, zumal die Zeitscheibe klein gewählt werden muß.

(relativer superlinearer Speedup ist natürlich möglich!)

Nun Übergang zu (etwas) realistischerer Voraussetzung:

$b_{p_{min}} = 0$ ist in Realität nicht gegeben !

$b_{p_{min}} = 0$ setzt eine vollständig nebenläufige Last voraus. Bei Existenz einer nichtleeren Präzedenzrelation ist aber b_p durch die *Bedienzeitsumme des kritischen Weges*, $b_{p_{kritW}}$, nach unten beschränkt!

$b_{p_{min}} = b_{p_{kritW}}$ [vgl. Beispiel (80), $b_{p_{kritW}} = 10$]

Trotz der Existenz von $b_{p_{min}}$ kann die maximale Breite, q_{max}, noch unbeschränkt sein, wenn nämlich ein Rest in w ideal parallelisierbar ist!

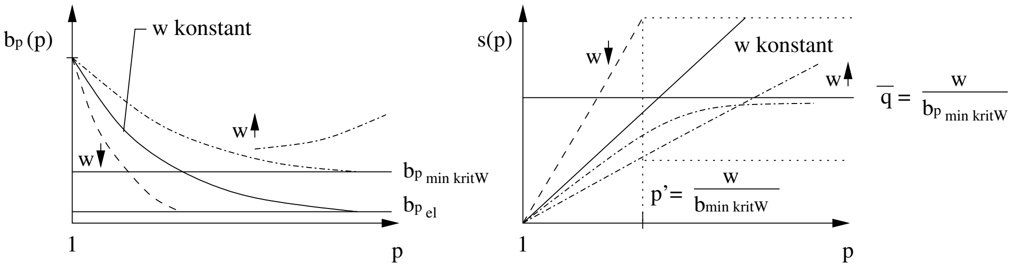
Tatsächlich ist auch das nicht realistisch, weil die kleinste vergebbare Lasteinheit eine von Null verschiedene Bedienzeit $b_{p_{el}}$ (el: elementar) hat. Also ist q_{max} nach oben beschränkt durch

$$q_{max} \leq \frac{w}{b_{p_{el}}} \tag{105}$$

und die Bedienzeit nach unten durch

$$b_{p_{min}} = \max\{b_{p_{kritW}}, b_{p_{el}}\}. \tag{106}$$

Auswirkung der Schranken für $b_{p_{min}}$:



i.a. $$b_{p_{min\ kritW}} \gg b_{p_{el}} \tag{107}$$

vgl. Bsp. (80): $$\overline{q} = \frac{w}{b_{p_{min}}} \tag{108}$$

Die Veränderung $w(p)$ hat auch (schlecht generell bestimmbare) Auswirkungen auf $w/b_{p_{min\ kritW}}$, da $b_{p_{min\ kritW}}$ auch mit w steigt/fällt. Falls $w/b_{p_{min\ kritW}} = const$, folgt $p' = const$.

Im folgenden wird wie (wie schon früher) steht's $b_{p_{min}} = b_{p_{min\ kritW}}$ gesetzt. (siehe nachfolgenden "Exkurs Speedup bei Wettbewerksparallelität")

Wir erkennen jetzt zwei *obere Schranken* für den Speedup, und wissen, daß jenseits von $p' = \frac{w}{b_{min}}$ eine Vergrößerung von p keine wesentlichen Vorteile bringt.

—

Exkurs Speedup bei Wettbewerksparallelität

Der Algorithmus erlaubt Varianten mit Bedienzeit B (Zufallsvariable).

Der Speedup ist

$$s(p) = \frac{E[B]}{E\big[\min_{Stichprobe\ p}\{B\}\big]},$$

wobei $E\,[B]$ mit $p = 1$, keine Vorkenntnis; und $E\,[\min_{Stichprobe\ p}\{B\}]$ mit $p \neq 1$ keine Vorkenntnis.

Dann ist $\lim_{p\to\infty}\ s(p) = \frac{E[B]}{b_{min}} = s_{max}$.

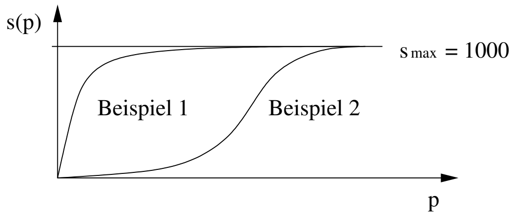
—

Beispiel 1: $P[B = 1] = P[B = 1999] = 0,5$

$$E[B] = 1000 \quad b_{min} = 1 \quad s_{max} = 1000$$

Beispiel 2: $P[B = 1]$ sehr klein, $P[B = 1000]$ sehr groß

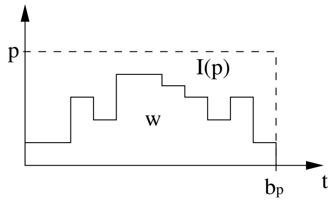
$$E[B] = 1000 \quad b_{min} = 1 \quad s_{max} = 1000$$



für kleine p gilt: $\begin{cases} 1: & E[w] \ll E[B] \quad (w \downarrow,\ superlinear\,!) \\ 2: & E[w] \approx E[B] \quad kein\ speedup \end{cases}$

—

Man kann noch, für konstantes w, eine untere Schranke für den Speedup bestimmen ([Eager et al 89]). Dazu benutzen wir eine Abschätzung der Freizeitsumme $I(p)$ bei Parallelität p.



(109)

Da

$$p * b_p = w + I(p)$$

folgt

$$b_p(p) = \frac{1}{p}(w + I(p))$$

damit ist $(b_p(1) = w)$ $(I(1) = 0)$

$$s(p) = \frac{w*p}{w+I(p)} = \frac{\overline{q}*b_{p_{min}}*p}{\overline{q}*b_{p_{min}}+I(p)} = \frac{\overline{q}*p}{\overline{q}*\frac{I(p)}{b_{p_{min}}}} \tag{110}$$

wobei wir $w = \overline{q} * b_{p_{min}}$ benutzt haben.

Jetzt wird $I(p)$ abgeschätzt,

$$I(p) \leq b_{p_{min}} * (p-1) \text{ } (\rightarrow \text{S.71 LvPr) (global produktive BS)} \tag{111}$$

und wir erhalten

$$s(p) \geq \frac{p*\overline{q}}{p+\overline{q}-1} = \frac{p}{1+(p-1)*\frac{1}{\overline{q}}} = \frac{p}{1+(p-1)*\frac{b_{p_{min}}}{w}} \tag{112}$$

die Schranke ist asymptotisch korrekt.

$$s(1) = 1 \quad s(\infty) = \overline{q}$$

und wird in Beispielen auch für $p \neq 1, \infty$ auch erreicht. vgl. (80).

—

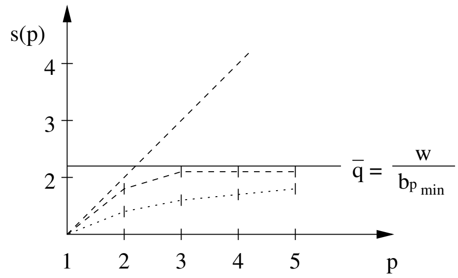
Beispiel wie (80)

wir tragen die untere Schranke in die Speedup-Funktion ein: $s(p) \geq \frac{p*\overline{q}}{p+\overline{q}-1}$ (113)

$\overline{q} = 2,125$ (vgl. (80))

$w = const.$

$b_{p_{min}} = 16$



—

Wenn über das Verfahren mehr bekannt ist als $\overline{q}$ (oder w, $b_{p_{min}}$), kann man genauere Vorhersagen machen. Wir verschieben das im Wesentlichen auf Kapitel 6 und betrachten nur noch einen (wichtigen) Sonderfall:

$$w(p) = w_1 + w_2 \quad (114)$$

Dabei sei w_1 nicht und w_2 ideal parallelisierbar.

"Amdahl-Modell" [Amdahl G 67]

Wie bisher: Einprogrammbetrieb, global produktive Bedienung, nur P wird beauftragt.

Es sei $w(p) = w_1 + w_2$ konstant gegen p und w_1 nicht und w_2 ideal parallelisierbar

also

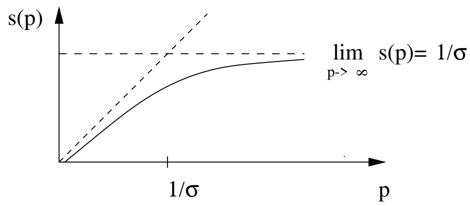
$b_p(1) = w$

$b_p(p) = w_1 + \frac{w_2}{p}$

$$s(p) = \frac{w}{w_1 + \frac{w_2}{p}} = \frac{p}{p * \frac{w_1}{w} + \frac{w - w_1}{w}} = \frac{p}{1 + (p-1) * \frac{w_1}{w}} \quad (115)$$

$\frac{w_1}{w}$ ist der relative Anteil der Bedienzeitsumme, der sequentiell ist. Wir nennen ihn σ:

$$s(p) = \frac{p}{1 + (p-1) * \sigma} \quad (116)$$



(117)

Amdahls Speedup-Gesetz

Da $b_{p_{min}}$ hier w_1, ist $\overline{q} = \frac{w}{b_{p_{min}}} = \frac{w}{w_1} = \frac{1}{\sigma}$, d.h. (116) erfüllt (112)

$$s(p) = \frac{p}{1 + (p-1) * \sigma} \quad \text{Amdahls Speedupgesetz}$$

ist eigentlich nur eine Interpretation des viel allgemeineren (und der Menschheit lange bekannten) folgenden Sachverhalts:

Eine Größe G sei durch einen Parameter p zu verkleinern, der einen Summanden G_1 unverändert läßt, aber $G - G_1$ beliebig klein macht. G ist höchstens im Verhältnis $\frac{G}{G_1}$ verkleinerbar. (118)

—

Beispiel: Wenn die Laufzeit eines Programmes zu 60% aus nicht überlappbaren Plattenzugriffen besteht und zu 40% aus Zentraleinheit-Zeit, dann kann auch die schnellste Zentraleinheit nur die Laufzeit um $\frac{1}{0,6} = 1,67$ kürzen. (119)

Beispiel: (wie [58](80)): Es ist $\sigma = \frac{b_\alpha + b_\lambda}{w} = \frac{3+2}{34} = 0,147$. Ein Speedup von $\frac{1}{\sigma} = 6,8$ ist aber unerreichbar, da $w_2 = 29$ nicht ideal parallelisierbar. (120)

—

1.7.4 Allgemeine Aussagen und Einbeziehung von Kommunikation

Wir stellen in einer Übersicht die grundlegenden Resultate für den folgenden Modellfall zusammen:

A Einprogrammbetrieb

B global-produktive Bedienstrategie (also idealisierte Lastverteilung)

C Betrachtung nur der parallelen Einheit P

D $p \leq q_{max}$ (also keine Sättigung durch mangelnde Breite)

Dabei geben wir noch einen Parallelitätsgrad p' an. Dessen (Weite)Überschreitung nicht lohnt, weil die Auslastung sinkt, ohne daß noch eine wesentliche Bedienzeitminderung eintritt.

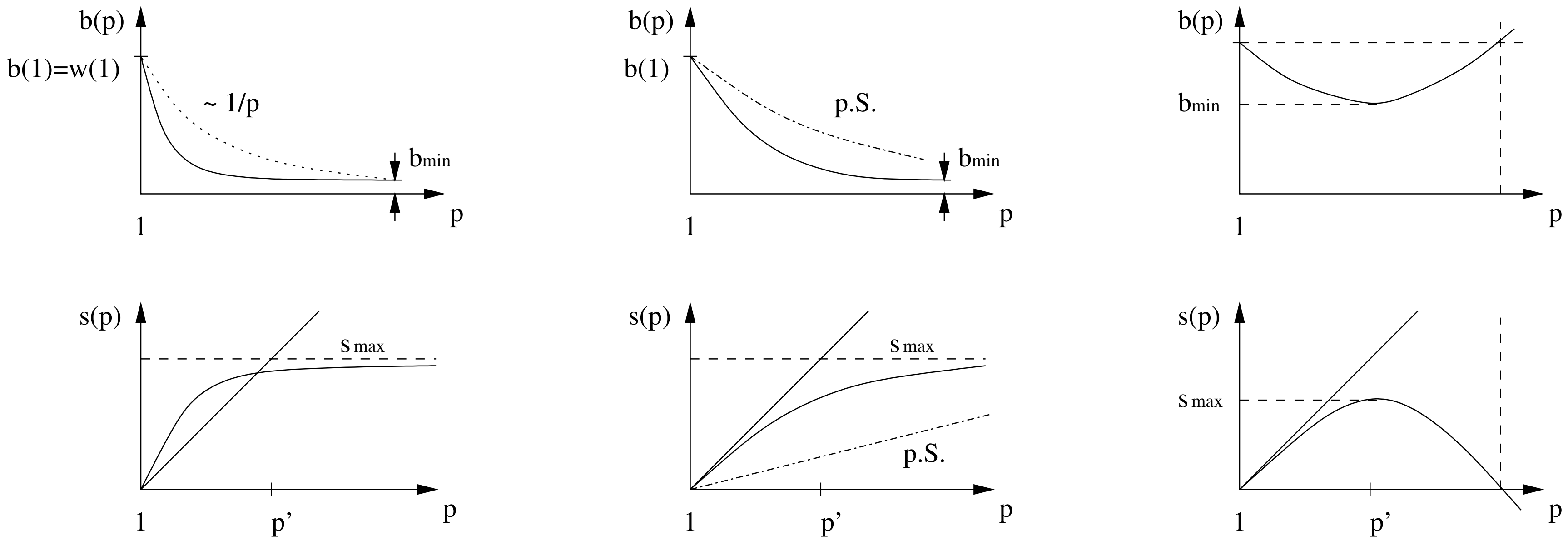
Später betrachten wir zusätzlich Kommunikation (d.h. C entfällt!), was auf NUMA und NORMA-Maschinen wichtig ist.

Die Rechenarbeit w(p)

sinkt mit p
(beste Anlaufcharakteristik)

ist konstant oder steigt nicht stärker, als sie parallelisierbar ist (z.B. O(p))

steigt stärker, als sie parallelisierbar ist



superlinear

p.S.: pessimistische Schranke

b(p) steigt nicht (aber Anomalien sind bekannt !).

$s_{max} = p' = w(1)/\ b_{min}$

b_{min} ist die Länge des kriteischen Wegs, > 0 !

Speedup ist gesättigt.

b(p) steigt nicht monoton.

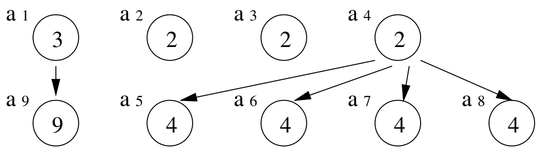
Steigt typisch für große p.

$s_{max} = p' = w(1)/\ b_{min}$

Speedup ist rückläufig !

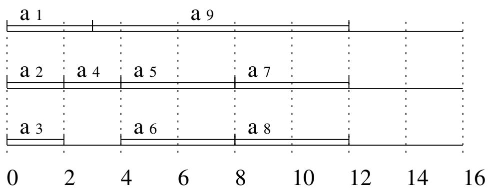
Zwei Anomalien, zum Staunen, aus Coffman, E., Denning, P.: Operating System Theory, Prentice Hall 1973

Auftragssystem:

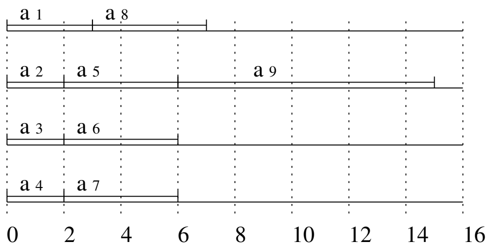


(122)

(A) $p = 3$, $w = 34$, $b(3) = 12$

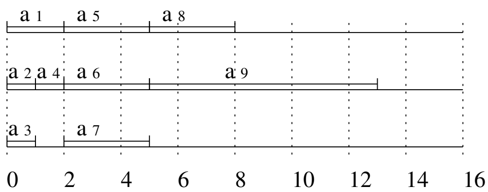


(B) $p = 4$, $w = 34$, $b(4) = 15$



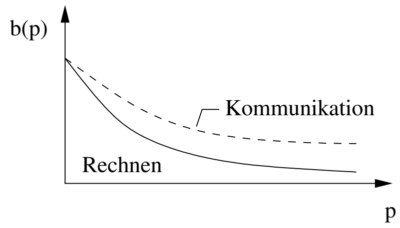
$b(4) > b(3)$ (!), aber Speedup $s(4) < s(3)$

(C) $p = 3$, $w = 25$, $b(3) = 13$



Alle b_i sind um 1 kleiner: $w = 25 < w = 34$. Aber $b = 13 > 12$ (!), kleinere Last liefert höhere Bedienzeit!

- Sogenannte *List-Serve-Anomalien*: strenge Einhaltung einer (präzedenzkonformen) Folge a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 bei der Vergabe.
- In der Informatik gibt es noch mehr solche "fast-nie"-Fälle: FIFO-Seitentauschanomalie, Anomalie der unproduktiven Bedienstrategie (nichts tun kann bedienzeitoptimal sein).



Nun Erweiterung auf eine zweite Resource: Verbindungsnetz VN: von Bedeutung bei NUMA. NORMA:

- Kommunikation bei $p = 1$ i.a. nicht vorhanden, also kein superlinearer Speedup durch schrumpfende Kommunikation möglich!
- Bei grober Granularität (p klein) kann die Kommunikation mit Rechnen überlappbar sein, nicht mehr bei feiner Granularität und niemals auf dem kritischen Weg!
- Kommunikation führt zu rückläufigem Speedup, wenn nicht gut parallelisierbar (also Crossbar, Fat-Tree; nicht Bus)

Rechnen: Rechenarbeit $w(p)$

Kommunikation: Belegung $x_{VN}(p) = \overline{b_{VN}}(p) * \overline{v_{VN}}(p)$

Rechnen sinkt mit p, Kommunikation steigt nicht stärker als sie parallelisierbar ist, oder in Bilanz $v(p)$, $x(p)$ sinkend	Alle konstant, oder nichts steigt stärker als parallelisierbar (z.B. $O(p)$)	Wenigstens ein Anteil steigt stärker als er parallelisierbar ist (z.B. $x(p) \sim p^2$, aber VN ist Crossbar)
superlinear: vgl. (121)	Wie (121) Mitte. Wahrscheinlich besteht ebenfalls eine pessimistische Schranke.	Speedup ist *rückläufig*.
wie (181): $s_{max} = p' = b(1)/b_{min}$		$s_{max} = p' = b(1)/b_{min}$

i.a. $b(1) = w(1)$ (keine Kommunikation ohne Parallelisierung)

b_{min} für $p \to q_{max}$	b_{min} für mittlere p

1.7.5 Benchmarks

nach [Hellwagner H 98].

Beispiele für Parallelrechner-Benchmarks.

Linpack **(125)**

- Lineares Gleichungssystem mit $n = 100, 1000, \ldots, max$ Unbekannten für gegebene Maschine.
- Sehr reguläres Problem, gibt hohe aber für andere Anwendungen nicht repräsentative Leistungswerte.
- Weit verbreitet, für verschiedene Systeme verfügbar.

Parkbench **(126)**

- Programmiermodelle Fortran77+PVM und HPF
- Quellprogramme liegen vor, zwei Arten von Resultaten: unverändert, optimiert.
- Schichtenkonzept von Benchmarks:
  - Low-Level: Timer-Auflösung, Gleitpunktdurchsatz, Speicher-, Kommunikations- und Synchronisationsleistung (barrier).
  - Kernels: Matrix-Operationen, Fast-Fourier Transformation
  - Compact Applications (nur Vorschläge)
  - HPF Compiler Benchmarks: z.Z. keine
- Schichtenmodell soll Leistungsmodell ermöglichen, für Vorhersage, aber aufwendige Benchmark

NAS Parallel Benchmarks **(127)**

Aus dem "Numerical Aerodynamic Simulation Program" der NASA (tatsächlich numerische Lösung partieller Differentialgleichungen, keine Simulation!), typisch strömungsdynamische Anwendungen.

Paper & Pencil Benchmark

Nur Spezifikation der Aufgabe und Problemgröße, Implementierung steht frei, aber Code Samples als Anleitung werden veröffentlicht.

5 Kernel Benchmarks

- EP (Embarrassingly Parallel): mißt vorwiegend Gleitpunktleistung, kaum Kommunikation.

- MG (Multigrid): 3D-Multigitter-Verfahren, strukturierte Kommunikation
- CG (Conjugate Gradient): Approximation des kleinsten Eigenwertes einer großen, dünn besetzten Matrix, unstrukturierte Kommunikation im Gitter
- FT (Fourier Transformation): Lösung dreidimensionaler partieller Differentialgleichungen, Kommunikation über große Distanzen
- IS (Integer sort)

3 Pseudoanwendungen: itterative Lösung von partiellen Differentialgleichungen:

- LU (Lower/Upper Triagonalmatrix (Gaußelimination)
- SP (Pentadiagonal Solver)
- BT (Block Tridiagonal Solver)

3 Problemgrößen

- Sample Codes: zum Testen
- Class A: mittlere Problemgröße
- Class B: große Problemgröße

Richtlinien für Darstellung und Einreichung von Benchmark-Resultaten:

- genaue Beschreibung der HW/SW Konfiguration
- genaue Beschreibung der Algorithmen und der Implementierung
- Quell-Programme
- Ausgabe-Protokolle der Benchmarks

NAS Parallel Benchmarks sind für technisch-wissenschaftliche Anwendungen sehr gut akzeptiert.

Benchmark code	Problem size	Memory (Mw)	Time (sec)	MFLOPS
Embarrassingly parallel (EP)	2^{24}	0.1	11.6	120
Multigrid (MG)	32^3	0.1	0.1	128
Conjugate gradient (CG)	1400	1.0	1.2	63
3-D FFT PDE (TF)	64^3	2.0	1.2	160
Integer sort (IS)	2^{16}	0.3	0.2	30.5
LU solver (LU)	12^3	0.3	3.5	28
Pentadiagonal solver (SP)	12^3	0.2	7.2	24
Block tridiagonal solver (BT)	12^3	0.3	7.2	34

Tabelle 1.1 NAS Parallel Benchmarks Sample Code Statistics

Benchmark code	Problem size	Memory (Mw)	Time (sec)	MFLOPS
Embarrassingly parallel (EP)	2^{28}	1	151	147
Multigrid (MG)	256^3	57	54	154
Conjugate gradient (CG)	14000	10	22	70
3-D FFT PDE (TF)	$256^2 \times 128$	59	39	192
Integer sort (IS)	2^{23}	26	21	37.2
LU solver (LU)	64^3	8	344	189
Pentadiagonal solver (SP)	64^3	6	806	175
Block tridiagonal solver (BT)	64^3	6	923	192

Tabelle 1.2 NAS Parallel Benchmarks Class A Statistics

Benchmark code	Problem size	Memory (Mw)	Time (sec)	MFLOPS
Embarrassingly parallel (EP)	2^{30}	18	512	197
Multigrid (MG)	256^3	59	114	165
Conjugate gradient (CG)	75000	97	998	55
3-D FFT PDE (TF)	$512 \times 256 \times 256$	162	366	195
Integer sort (IS)	2^{25}	114	126	25
LU solver (LU)	102^3	121	1973	162
Pentadiagonal solver (SP)	102^3	36	2160	207
Block tridiagonal solver (BT)	102^3	160	3554	203

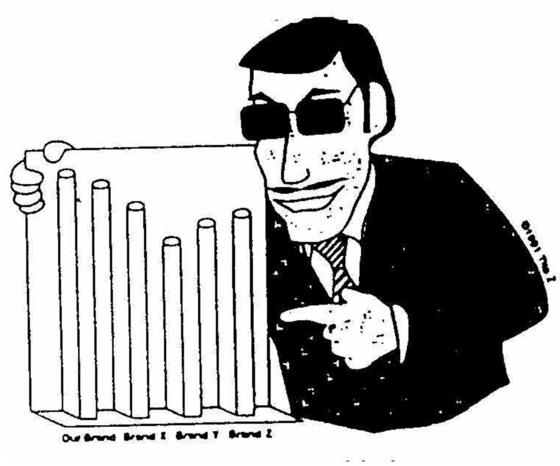
Tabelle 1.3 NAS Parallel Benchmarks Class B Statistics

1.7.6 Advice from a top - salesman

(Quelle: Supercomputing Review, Aug. 1991: D.H. Bailey: Twelf ways to fool the masses when giving performance results on parallel computers)

Twelf Ways to Fool the Masses When Giving Performance Results on Parallel Computers

by D.H. Bailey



1. Quote only 32·bit performance results, not 64·bit results.

We all know that it is hard to obtain impressive performance using 64-bit floating-point arithmetic. Some research systems do not even have 64-bit hardware.Thus, always quote 32 bit results, and avoid mentioning that you're doing so. Better still, compare your 32-bit results with 64-bit results on other systems. This 32-bit arithmetic may or may not be appropriate for your application, but the audience doesn't need to be bothered with such details.

2. Present performance figures for an inner kernel, then represent these figures as the performance of the entire application.

It is difficult to obtain high performance on a complete, large-scale scientific application, timed from beginning of execution through completion. There is often a great deal of data movement and initialization that depresses overall performance rates. A good solution is to present results for an inner kernel of an application, which can be souped-up with artificial tricks. Then simply imply in your presentation that these rates are equivalent to the overall performance of the entire application.

3. Quietly employ assembly code and other low-level language constructs.

It is often hard to obtain good performance from straightforward Fortran or C code that employs the usual parallel programming constructs, due to compiler weaknesses on many highly parallel computer systems.Thus, feel free to employ assembly-coded computation kernels, customized communication routines and other low-level code in your parallel implementation. Don't mention such usage, though, since it might alarm the audience to learn that assembly-level coding is necessary to obtain respectable performance.

4. Scale up the problem size with the number of processors.

Graphs of performance rates versus the number of processors have a nasty habit of trailing off. This problem can be remedied by plotting the performance rates for problems whose sizes scale up with the number of processors. The important point is to omit any mention of this scaling in your plots and tables. Disclosing this fact might raise questions about your implementation's efficiency.

5. Quote performance results projected to a full system.

Few labs can afford a full-scale parallel computer - such systems cost millions of dollars. Unfortunately, the performance of a code on a scaled-down system is often not very impressive. There is a straightforward solution to this dilemma: Project your performance results linearly to a full system and quote the projected results without justifying the linear scaling. Be very careful not to mention this projection, however, since it could seriously undermine your performance claims should the audience realize you did not actually obtain your results on real full-scale hardware.

6. Compare your results against scalar, unoptimized code on Crays.

It really impresses the audience when you can state that your code runs several times faster than a Cray, still the world's dominant supercomputer. Unfortunately, with a little tuning many applications run quite fast on Crays. Therefore, you must be careful not to do any tuning on the Cray code. Do not insert vectorization directives, and if you find any, remove them. In extreme cases. it may be necessary to disable all vectorization with a command-line flag. Also, Crays often run much slower with bank conflicts, so be sure that your Cray code accesses data with large, power-of-two strides whenever possible. It is also important to avoid multitasking and autotasking on Crays - imply in your paper that the one-processor Cray performance rates you are comparing against represent the full potential of a $25 million Cray system.

7. When direct run-time comparisons are required, compare with an old code on an obsolete system.

Direct run-time comparisons can be quite embarrassing, especially if your parallel code runs significantly slower than an implementation on a conventional system. If you are challenged to provide such figures, compare your results with the performance of an obsolete code running on obsolete hardware with an obsolete compiler. For example, you can state that your parallel performance is "100 times faster than a VAX 11/780." A related technique is to compare your results with those on another, less capable parallel system or minisupercomputer. Keep in mind the bumper sticker, "We may be slow, but we're ahead of you."

8. If MFLOPS rates must be quoted, base the operation count on the parallel implementation, not on the best sequential implementation.

We know that MFLOPS rates of parallel codes are often not very impressive. Fortunately, a few tricks can make these figures more respectable. The most effective scheme is to compute the operation count based on an inflated parallel implementation. Parallel implementations often perform far more floating-point operations than the best sequential implementation.

Often millions of operations are masked out or merely repeated in each processor. Millions more can be included simply by inserting a few dummy loops that do nothing. Including these operations in the count will greatly increase the resulting MFLOPS rate and make your code look like a real winner.

9. Quote performance in terms of processor utilization, parallel speedups or MFLOPS per dollar.

As mentioned above, run-time or even MFLOPS comparisons of codes on parallel systems with equivalent codes on conventional supercomputers often are not favorable. Thus, whenever possible use other performance measures. One of the best is "processor utilization" figures. It sounds great when you can claim that all processors are busy nearly 100 percent of the time, even it what they are actually busy with is synchronization and communication overhead. Another useful statistic is "parallel speedup" - you can claim "fully linear" speedup simply by making sure the single-processor version runs just slow enough. Include synchronization and communication overhead on the single-processor version. even though this code is not necessary when running on only one processor.

A third statistic that many in the field have found useful is "MFLOPS per dollar." Be sure not to use "sustained MFLOPS per dollar," i.e., actual delivered computational throughput per dollar, since these figures often are not favorable to new computer systems.

10. Mutilate the algorithm used in the parallel implementation to match the architecture.

Everyone is aware that algorithmic changes are often necessary when we port applications to parallel computers.Thus, in your parallel implementation, it is essential that you select algorithms that exhibit high MFLOPS performance rates, without regard to fundamental efficiency. For example, partial differential equation applications with explicit linear system solvers typically run at rather high MFLOPS rates on parallel computers.

Unfortunately, such algorithmic changes often result in a code that requires far more time to complete the solution - explicit methods, for example, sometimes converge 1.000 times slower than implicit or multigrid methods. For this reason, you must be careful to downplay your changes to the algorithm. Otherwise, the audience might wonder why you employed such an inappropriate solution technique.

11. Measure parallel run times on o dedicated system, but measure conventional run times in a busy environment.

There are a number of ways to further boost the performance of your parallel code relative to the conventional code. One way is to make many runs on both systems, then publish the best time for the parallel system and the worst time for the conventional system. Another is to time your parallel computer code on a dedicated system and time your conventional code in a normal loaded environment. After all, your conventional supercomputer is very busy, and it is hard to arrange dedicated time. If anyone in the audience asks why the parallel system is freely available for dedicated runs but the conventional system isn't, change the subject.

12. If all else fails, show pretty pictures and animated videos, end don't talk about performance.

It sometimes happens that the audience starts to ask all sorts of embarrassing questions. These people simply have no respect for the authorities in our field. If you are the object of such disrespect there is always a way out simply conclude your technical presentation and roll the videotape. Audiences love razzle-dazzle color graphics, and this material often helps deflect attention from substantive technical issues.

Acknowledgments

The author wishes to acknowledge helpful contributions and comments by the following persons: R. Bailey, E. Barszcz, R. Fatoohi, P. Frederickson, J. McGraw, J. Riganati, R. Schreiber, H. Simon, V. Venkatakrishnan, S. Weerstunga, J. Winget and M. Zosel.

David H. Baily is with the Numerical Aerodynamic Simulation (NAS) Systems Division at NASA Ames Research Center, Moffett Field, CA 94035.

1.8 Der Markt : T0P500

→ http://www.top500.org

Erste 10 Plätze Top 500 Supercomputer Nov. 99

(alle sind Parallelrechner)

Rang	Produkt, installiert	Ort	Einsatz	Zahl CPU's	Durchsatz Linpack	GFlops/s Peak
1	Intel ASCI RED, 1997/99	Sandia Albuquerque	Forschung	9632	2.400	3.200
2	IBM Blue Pacific SST, SP 604e ASCI, 1999	Lawrence-Livermore	Forschung	5808	2.100	3.900
3	SGI ASCI Blue Mountain, 1998	Los Alamos	Forschung	6144	1.600	3.100
4	Cray/SGI T3E 1200, 1998	"Government"	Geheim	1084	890	1.300
5	Hitachi SR8000/128, 1999	Uni-Tokio	Forschung/ Lehre	128	880	1.000
6	Cray/SGI T3E 900	"Government"	Geheim	1324	820	1.200
7	SGI Origin 250 Mhz	Los Alamos	Forschung	2048	690	1.000
8	Cray/SGI T3E 900	Naval Occe-anographic Off. Bay. S.Louis	Wetter	1084	680	980
9	Cray/SGI T3E 1200	Deutscher Wetterdienst, Offenbach	Wetter	812	670	970
10	IBM SP Power 3, 222 Mhz	Uni-San Diego	Forschung/ Lehre	960	560	850

Produkte nach Rang

Rang	Produkt	Jahr	Zahl CPU's	Durchsatz Linpack	GFlops/s Peak
1	Intel ASCI Red	97/99	9632	2.400	3.200
2	IBM SP 604e ASCI Blue Pacific	99	5808	2.100	3.900
3	SGI ASCI Blue Mountain	98	6144	1.600	3.100
4	SGI/Cray T3E 1200	98	1084	890	1.300
5	Hitachi SR 8000 /128	99	128	870	1.000
7	SGI Origin 2000	99	2048	690	1.000
8	IBM SP3 222	99	960	560	850
15	Fujitsu VPP 800/63	99	63	480	500
24	Hitachi CP/PACS	99	2048	370	300
32	Fujitsu VPP 5000	96	31	290	300
33	SUN HPC 4500	98	720	270	480
34	Compaq Alpha Serv.	99	512	270	510
36	Fujitsu VPP 700	99	128	270	310
40	NEC SX-4	97	128	240	260
67	SUN HPC 10000	99	256	140	200
73	NEC SX-5	99	16	120	130
101	SGI/Cray T3D	94	1024	100	150
112	HP V 2500	99	218	84	260

2 Verbindungswerke

Literatur:

Richter, H.: Verbindungswerke für parallele und verteilte Systeme, Spectrum, Heidelberg 1997
[Hwang Xu 98]: Kap. 6, S.273...S.342
[Jessen E 98] (LvPr-Vorlesung) Kap.5, S.98...S.137A
[Hennesy Patterson 96]: Kap. 7, S.563...S.634
[Hellwagner H 98]: Kap. 5.1, S.5.1..S32

2.1 Bedeutung und Anforderungen

Verbindungswerke bestimmen ganz wesentlich Leistung und Kosten eines Parallelrechners. (141)

Die Verwendung eines Prozessors im Parallelrechner erweitert die Anforderungen an den Prozessorentwurf nicht wesentlich:

- Interprozessoralarme,
- Unterteilbarkeit von Lese/Modifizier/Schreibzyklen am Hauptspeicher,
- Unterstützung von Cache-Koheränzmechanismen,
- Adressraum, konsistente Zugriffsfolgen am Speicher.

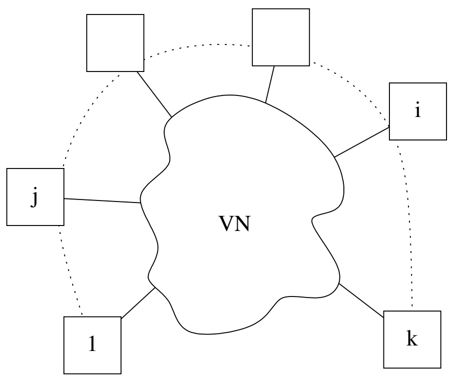
Parallelrechner sind oft zugleich Höchstleistungsrechner. Daher gibt es weitere Anforderungen an Prozessoren:

- hoher Grenzdurchsatz,
- gute Unterstützung von SIMD-Funktionen (Vektor-Pipelines); oft zugleich für Multimedia wünschenswert.

Anforderungen an Verbindungswerke

Das Verbindungswerk kann die Elemente einer Gruppe intern verbinden (gruppenintern) oder Elemente einer Gruppe mit Elementen einer anderen Gruppe (gruppenextern).

Gruppenintern: k Werke/Rechner sind untereinander zu verbinden ("symmetrisch"). (142)

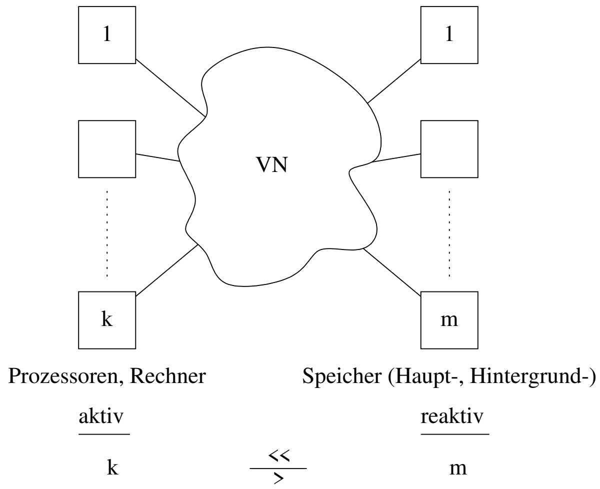


VW Verbindungswerk, k Kapazität, $i \in [1..k]$ ist mit allen $j \in [1..k]$, $i \neq j$ zu verbinden.

Die Beziehung ist symmetrisch (Verbindungsaufbau, Übertragung). Es gibt also bis $k * (k-1)$ Wege.

Beispiele: Rechnernetze, Multirechnersystem, Feldrechner.

Gruppenextern: k Prozessoren/Rechner mit m Speicher ("asymmetrisch"). (143)



Es können

- $\min(k, m)$ Verbindungen bestehen (falls je Prozessor/Rechner höchstens ein Speicher belegt wird)
- m Verbindungen bestehen (falls je Prozessor/Rechner mehrere Speicher belegt werden, vgl. Pipeline, Cache)

"belegt": mit Lesen, Schreiben beauftragt.

vgl. auch Rechnerarchitektur I - Vorlesung: Technik des vorübergehenden Freigebens von VN bei Leseoperationen: Pipelining von VW - Speicher - VW.

Beispiel: Mehrprozessorsysteme, Seitentransporte bei gemeinsamem verteiltem Speicher

Grenzdurchsatz $c = c$ (k, p, m, l, Verteilung) (144)

p von k seinen für den betrachteten Auftrag/die betrachteten Aufträge aktiv; es kommt i.a. wesentlich auf die Verteilung der Quellen und Senken der Verbindung an ! Der Rest sei frei, d.h. produziere keinen bzw. nicht störenden Verkehr.

m (bei gruppenexterner Verbindung von Elementen)

l Länge der Nachrichten

Der Einfluß von k, p, m beschreibt die *Skalierbarkeit* des (Prinzip des) Verbindungswerkes: hier vor allem $c \sim p$ wichtig, d.h. Grenzdurchsatz steigt mit der Zahl der Quellen. Ähnliche Skalierbarkeitsforderungen:

- Verweilzeit steige nur schwach mit p, k.
- Kosten steigen nur schwach mehr als k, m.
- Durchsatz steigt proportional zu p bzw. k.

Eine Vergleichsgröße für den Grenzdurchsatz erhält man, wenn man den minimalen Grenzdurchsatz aller Schnitte bestimmt, die den Rechner in zwei Teile (annähernd) gleicher Zahl von Werken teilt: Biselektions-Grenzdurchsatz (bisection channel width, bisection bandwidth). (145)

Verweilzeit $y = y$ (k, p, m, l, Verteilung) (146)

der Nachrichten im Verbindungswerk. Verbindungsnetze, in denen die Verweilzeit nie größer als die Bedienzeit ist, heißen *nichtblockierend*, sonst *blockierend*. (dann ist die Bedienstrategie ("Wer wird als erster bedient?") wichtig). Das Blockieren kann den lokalen *Verlust* der Nachricht bedeuten; (147)

bei Erkennung des Verlustes muß der Sender die Nachricht abermals verschicken ! ($\rightarrow$Quittung, time-out, Wiederherstellung der Nachrichtenfolge).

Verklemmungsfreiheit (148)

Erhaltung der Reihenfolge von Nachrichten auf jeder Verbindung (149)

Fehlertoleranz (z.B.) zweifache Erreichbarkeit) (150)

Zahl der Ports der verbundenen Elemente (gering halten!) (Skalierbarkeit!) (151)

Ausbaubarkeit der Zahl der verbundenen Elemente, k, m. Möglichst ohne Rückwirkung auf die Zahl der Ports. Siehe auch Bemerkung zur Skalierbarkeit bei Grenzdurchsatz. (152)

Homogenität: gleicher Grenzdurchsatz/Verweilzeit für alle Verbindungen ! Falls (wie bei großen k, m) nicht oder nicht leistungsfähig herstellbar: günstige *Abbildbarkeit* der *problemgegebenen Kommunikationsstruktur* auf das Verbindungswerk.

2.2 Lösungen

Lösungsraum: (153)

- Topologie
- feste oder geschaltete Verbindungen
- bei geschalteten Verbindungen:
  - Fehlertoleranz durch alternative Wege
  - einstufig/mehrstufig
  - falls alternative Wege: Routing (zentral/dezentral, adaptiv/nichtadaptiv)
  - Nachrichten/Paketschaltung (mit Varianten)
  - Zweiweg/Vielweg-Schalter
- Leitungsgrenzdurchsatz, Grenzdurchsatz der Schalter
- Verzögerungszeit der Schalter

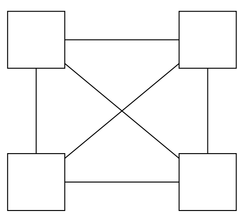
In der Literatur oft (inkonsistente) Unterscheidung:

statisches Verbindungswerk: Schalterfunktionen sind den Werken (Rechner, Prozessor, Rechenwerk) $1:1$ zugeordnet.

dynamisches Verbindungswerk: Schaltfunktionen sind in autonomen Schaltern. Sie sind keinem oder mehreren Werken zugeordnet (entspricht etwa: indirekten Netzen).

2.2.1 Topologien

I. Einstufige direkte Verbindung

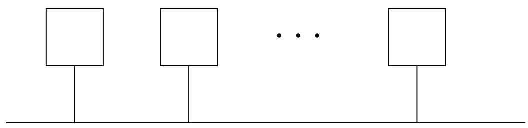


(154)

vollständige Vermaschung

gruppenintern!

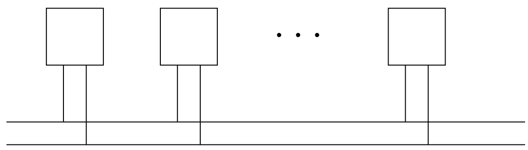
II. Einstufig indirekte Verbindung



(155)

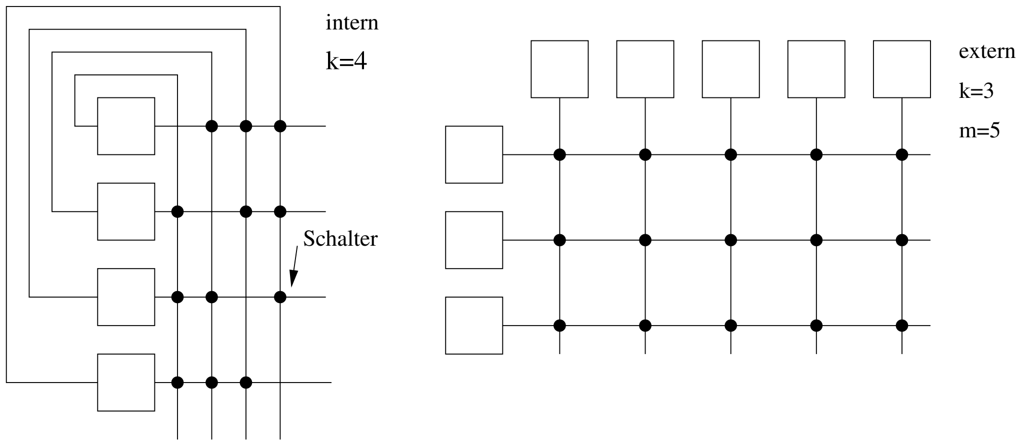
Bus (wirkt wie ein Schalter, daher hier: "indirekt")

gruppenintern/gruppenextern!



(156)

z-facher Bus (hier $z = 2$)

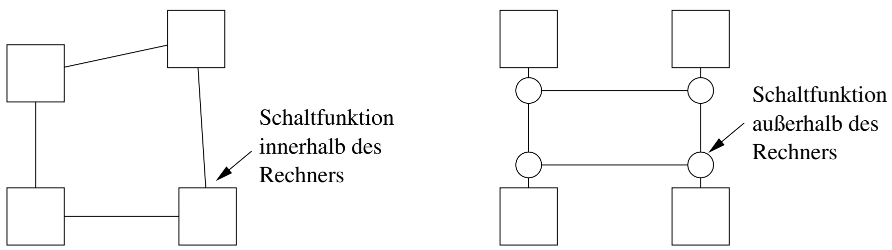


(157)

Kreuzschienen-Verteiler (crossbar)

Kreuzschienen-Verteiler und Busse sind auch als Hierarchien wichtig !

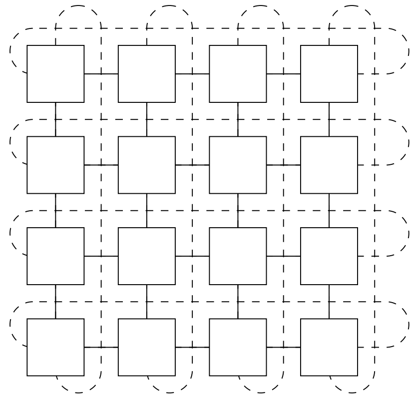
III. Mehrstufig indirekte Verbindungen



(158)

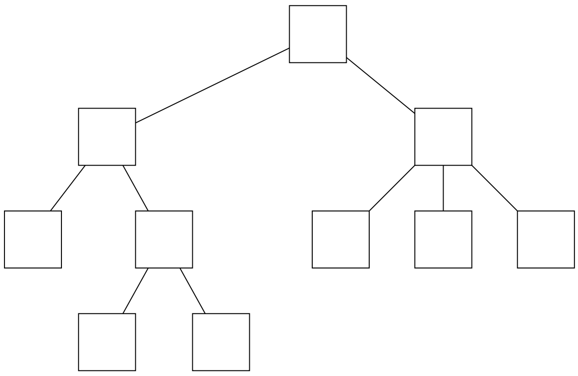
Ring; $O(n)$

Für unsere Klassifikation ist es hier (und im Folgenden) unerheblich, ob die Werke zusätzlich Schaltfunktionen leisten, oder ob dafür (besser!) besondere Schalter dienen.



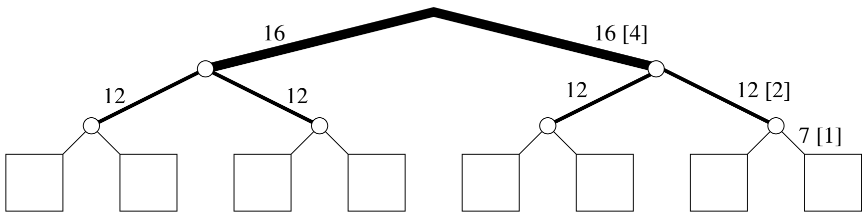
(159)

Gitter/Torus; $O(\sqrt{n})$, auch in höheren Dimensionen als $z = 2$



(160)

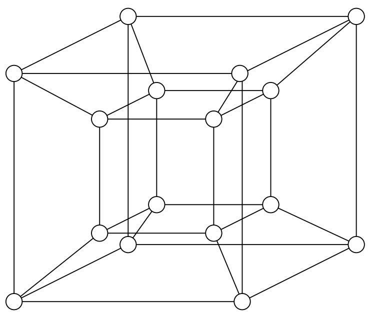
Baum; $O(\log(n))$ von Wurzel zu Blatt, $O(2 * \log(n))$ von Blatt zu Blatt



(161)

Fat Tree

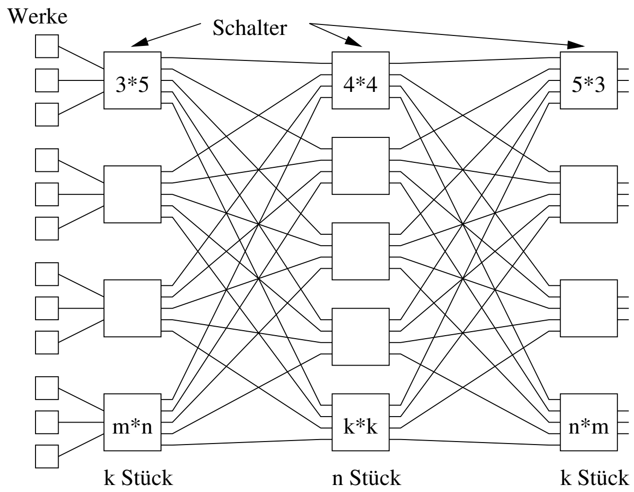
Werke nur als Blätter; jede der $k * (k-1)$ Verbindungen (jeder mit jedem) oder $k/2$ Verbindungen (jeder mit einem) [in Klammern]



(162)

Hyperwürfel (hier Dimension 4); $O(\log(n))$ - schlechtere Ausbaubarkeit als Baum, wegen Portanzahl

Variante: Ringe an den Ecken: cube-connected-cycles; Vorteil Portanzahl unabhängig von Dimension !

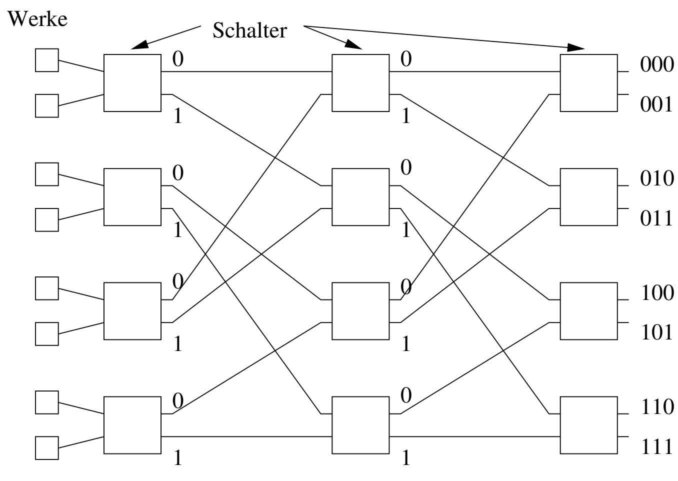


(163)

Clos-Netz

3 stufig, aus Crossbar-Bausteinen, hier 3×5, 4×4.

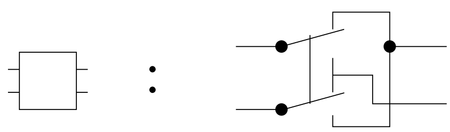
$n \geq 2 * m - 1$: blockierungsfrei, (im Bild: $m = 3$, $n = 5$)



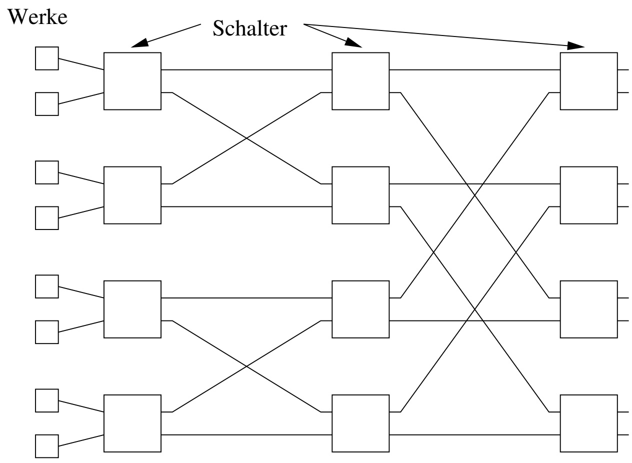
(164)

Omega-Netz

Adressbitgesteuerte Zweier-Schalter



Permutationsstufen: perfect shuffle



(165)

Banyan-Netz

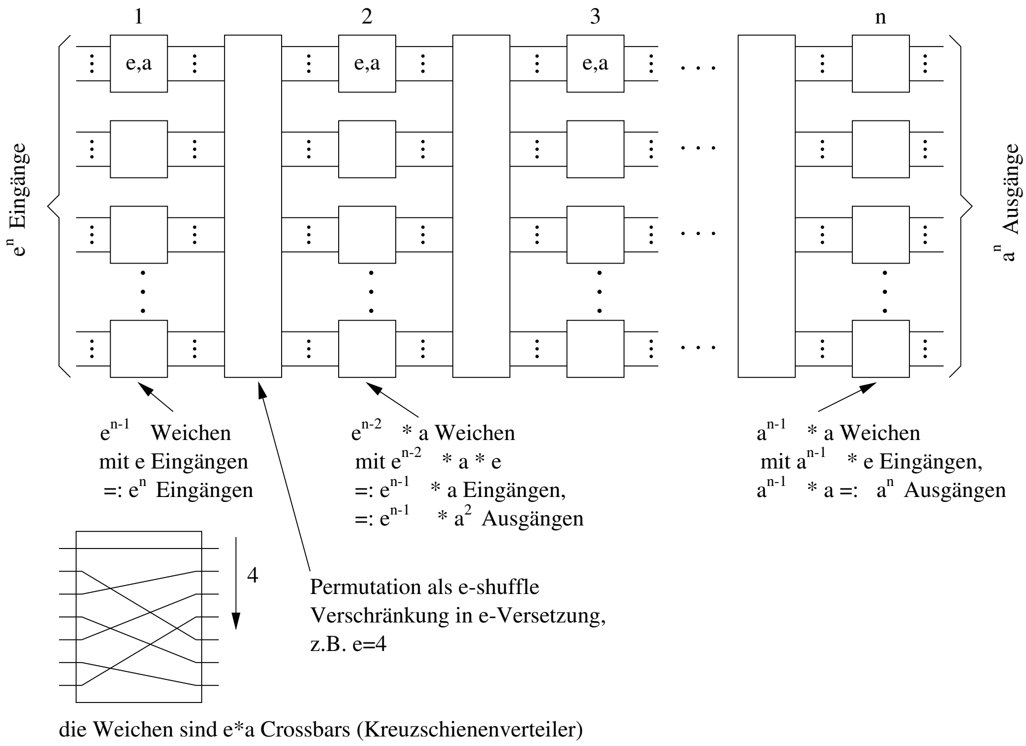
Schalter wie Omega-Netz

Permutationsstufen: butterfly

Es gibt zahlreiche, einander teils überlappende Klassen von vielstufigen Schaltern/Permutationsnetzen. Sind sie aus gleichen Stufen aufgebaut (wie das Omega-Netz), dann kann an ihrer Stelle auch ein einstufiges rückgekoppeltes Netz treten !

Verallgemeinerung:

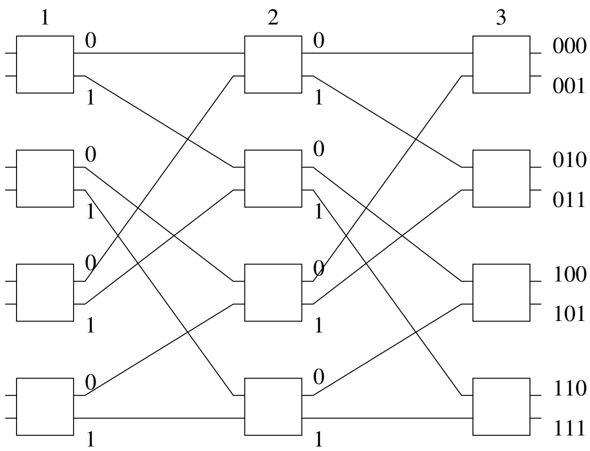
Delta-Netze: Eine Folge von Adress-Stellen gesteuerten Weichen und speziellen Permutationsgliedern (shuffle)



(165a)

—

Beispiel $n = 3$, $e = 2$: Omega



2-shuffle "perfect shuffle"

—

2.2.2 Bewertung

[Hellwagner H 98]: Stand der Technik bei Verbindungsnetzen in massiv parallelen Rechnern:

- Trend zu 2D/3D-Gittern, mehrstufigen Schalternetzen oder Crossbars; weg von Hypercubes
- bei hohem Knotengrad noch bitserielle Übertragung mit 1..10 MB/s je Kanal, sonst Byte/Wort-parallel mit bis zu 300 MB/s je Kanal
- Nachrichtenschaltung vorwiegend adaptive Paketvermittlung mit Wormhole "Routing"
- Hardware-Verzögerungen im Netzwerk (ohne Blockierungen!) wenige Mikrosekunden, auch darunter
- Software-Aufsetzzeiten im Bereich einiger -zig Mikrosekunden

In *jüngster Zeit* oft *mehrschichtige Verbindungswerke*: Knoten als Bus/Crossbar/Ring, Gesamtsystem als Gitter/Hypercube/mehrstufiges Crossbar. (165b)

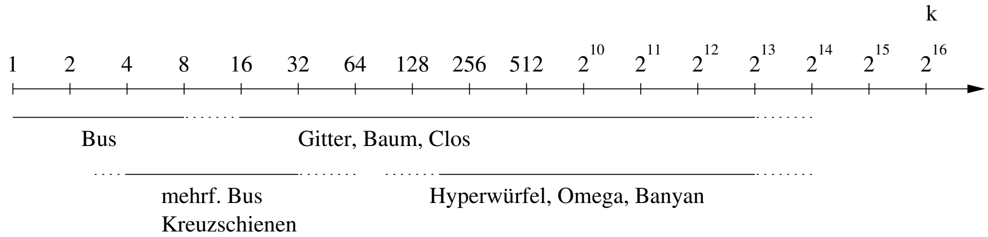
Topologie	ext. int.	Stufen	$c(k)$	$b(k)$	Block.	Fehler-toleranz	homogen	Ports	Kosten	Ausbau	Beispiel (Bemerkungen)
I. Direkt											
vollst. vermascht	i	0	$k(k-1)c_0$	$O(1)$	nein	nein	ja	$k+1$	$O(k^2)$	$--$	
Speicherkopplung	e	0		$O(1)$	ja	ja	ja/nein	$< m$	$O(k * m)$	$-$	(m Speichermoduln) EGPA, DIRMU, Cray Y-MP
II. Einstufig											
Bus	i,e	1	c_0	$O(1)$	ja	nein	ja	1	$O(k)$	$++$	Sequent, Siemens MX500, Encore Multimax
z-facher Bus	i,e	1	$z * c_0$	$O(1)$	ja	ja	ja	z	$O(k * z)$	$-$	Suprenum (Hierarchie)
Kreuzschienen-Verteiler	i,e	1	$k * c_0$	$O(1)$	nein	nein	ja	1	$O(k^2)$	$-$	Alliant FX/2800, B 825 D, Fujitsu VPP 700 (Hierarchie), Cray Y-MP, SUN E 10000
III. Mehrstufig											
Ring	i,e	$< k$	$< k * c_0$	$O(k)$	ja	ja	nein	2	$O(k)$	$++$	Convex-C1-XP, KSR 1
Gitter/Torus	i	$< \sqrt{k}$	$< k * c_0$	$O(\sqrt{k})$	ja	ja	nein	4	$O(k)$	$+$	Illiac IV, DAP, IPSC 860, Supremum, Cray T3D/E, Intel ASCI RED
Baum	i,e	$\log(k)$	$< k * c_0$	$O(\log(k))$	ja	nein	nein	3	$O(k)$	$++$	(3 Ports: Binärbaum), DADO
Fat Tree	i,e	$\log(k)$	$\frac{n}{2} * c_0$	$O(\log(k))$	nein	nein	nein	1	$O(k)$?	$-$	(1 Port falls nur 1 Kommunikation) CM-5
Hyperwürfel	i	$\log(k)$	$< k * c_0$	$O(\log(k))$	ja	ja	nein	$ld(k)$	$O(k*\log(k))$	$--$	Intel IPSC I, II, CM-2, SGI Origin
Clos-Netz	i,e	3	$\leq k * c_0$	$O(1)$	ja/nein	ja	ja	1	$O(k^{\frac{3}{2}})$	$-$	IBM SP 2 [Richter H 97]
Omega-Netz	i,e	$\log(k)$	$< k * c_0$	$O(\log(k))$	ja	nein	ja	1	$O(k*\log(k))$	$-$	CEDAR, N.Y.Ultracomputer, IBM RP 3
Banyan-Netz	i,e	$\log(k)$	$< k * c_0$	$O(\log(k))$	ja	nein	ja	1	$O(k*\log(k))$	$-$	BBN Butterfly, IBM RP3, TRAC
Delta-Netz	i,e	$\log(k)$	$< k * c_0$	$O(\log(k))$	ja	nein	ja	1	$O(k*\log(k))$	$-$	

2.2.3 Leistung

Rohe Bewertung:

1. Es gibt *keine* insgesamt optimale Struktur.

2. Nach der Zahl k zu verbindender Werke: (167)



3. Für große k etwa erreichbar (168)

- Grenzdurchsatz $\sim k$,
- Bedienzeit $\sim \log(k)$,
- geringe Blockierwahrscheinlichkeit,
- Kosten $\sim k * \log(k)$,
- mäßige Ausbaubarkeit,
- Fehlertoleranz: z.B. doppeltes Netz,
- einigermaßen effiziente Abbildung der Struktur der Anwendungskommunikation auf das Verbindungswerk.

4. Vom *Grenzdurchsatz des Verbindungswerkes* c ist zu unterscheiden der *Grenzdurchsatz je Verbindung*, der höchstens c_0 (Grenzdurchsatz von Schalter, Leistung) ist, wenn Verbindungen nicht gesplittet werden, was bei Parallelrechnern unüblich ist.

5. Grenzdurchsatz und Bedienzeit hängen von der *Dienstschnittstelle* ab (entsprechend OSI-Referenzmodell). Die Prozessorzeiten für die Protokollabarbeitung können groß gegen die Übertragungszeit sein und den Grenzdurchsatz empfindlich schmälern. Volumenunabhängige "Sockel"zeiten kommen durch Signallaufzeiten (Nachricht, Quittung, Zustandsmeldungen), eventuell auch durch Zwischenspeicherung (teilweise üblich!) in jedem Fall hinzu.

6. In Situationen, in denen p verschiedene Kommunikationsschritte zugleich anstehen, z.B. bei Eröffnung einer synchronisierten Parallelphase, wird die Dauer eher durch den Grenzdurchsatz bestimmt als durch die Bedienzeit: (169)

$t_{Kommunik.} \approx \frac{p}{c(k)}$

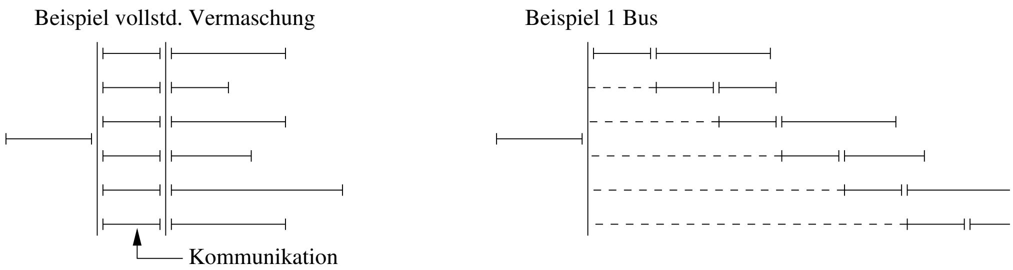
Damit ergeben sich $t_{Kommunik.}$

$O(1)$ vollständige Vermaschung

$O(1)$ Kreuzschienenverteiler

$O\left(\left\lceil \frac{p}{z} \right\rceil\right)$ z-Bus

$O(p)$ Bus, Ring



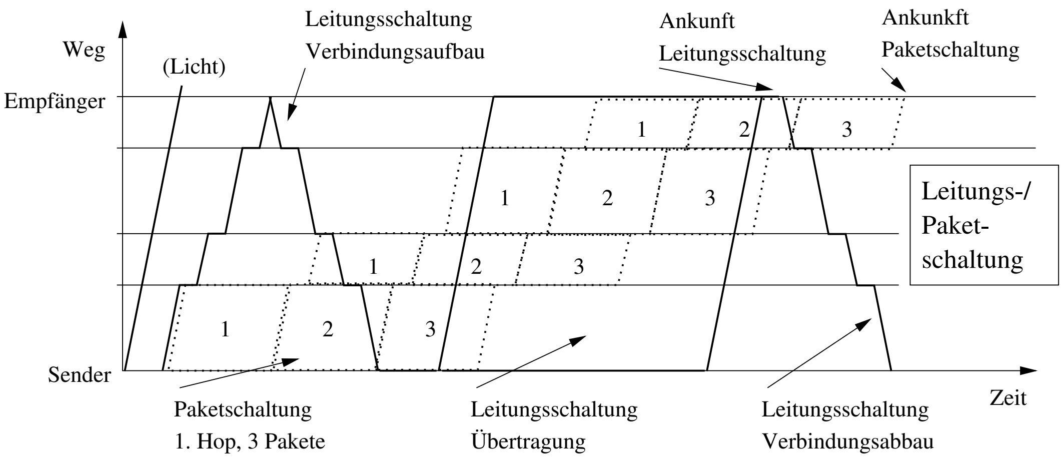
Weitere Einschränkungen durch Sender/Empfänger möglich (z.B. *eine* Nachricht auf einmal empfangen/senden !).

7. Man kann durch Auflösung der Nachrichten in Pakete bzw. in kleine Zellen *bessere Bedienzeiten gegen niedrigeren Grenzdurchsatz tauschen*; vgl. Vorlesung Rechnernetze:

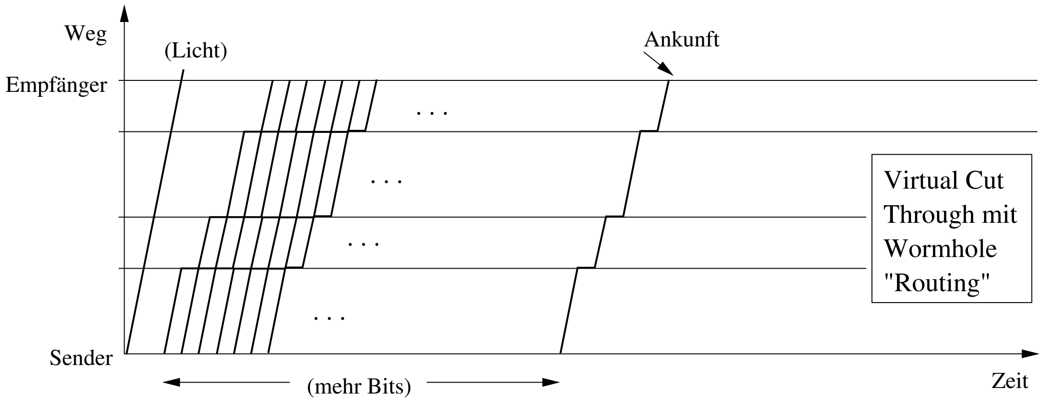
Leitungsschaltung: Verbindungsaufbau - Übertragungsphase - Verbindungsabbau. In Parallelrechnern überwiegt Auf/Abbau der Verbindung wegen kleiner Laufzeit weniger als in WAN's ! (169a)

Paketschaltung: Mehr Overhead in der übertragenen Information (Adressen...), aber (169b)

- datagram-Technik: kein Verbindungsaufbau, paketweise unabhängiges Routing, Kürzung der Bedienzeit
- virtual channel: weniger Overhead in der Übertragung, aber dank Reservierung keine Blockierung/Deadlock in Übertragungsphase
- virtual cut through: wie Datagram, aber bei nicht-Blockierung Vermittlung ohne Zwischenspeicherung (on the fly)
- wormhole "routing" (kein Routing!): wie virtual cut through, aber es gibt keine End - zu - End - Verbindung, nur Pufferreservierung zwischen Beginn und Ende der Nachricht. Verwendung sehr kleiner Zellen für "Flits" (flow control digits).



(169c)



(169d)

8. *Asymptotische Analyse* von Verbindungswerken: Eine fundamentale Einsicht des Leistungsanalyse, angewandt auf Verbindungswerke und das Problem des latency hiding.

Asymptotische Analyse:

Ein System S wird mit voneinander unabhängigen Aufträgen beschickt, die in Konkurrenz Teilaufträge an den Einheiten i ($i \in [1..m]$) des Systems auslösen und sich i.a. behindern.

Beispiel: Verbindungswerk mit vielen Paketen, die zu übermitteln sind.

Die Einheiten des Systems sind durch Teilauftragsgrenzdurchsätze c_i und Besuchszahlen v_i gekennzeichnet. Dann ist der Grenzdurchsatz von S $c_s = \min\left\{\frac{c_i}{v_i}\right\}$ vgl. (74) (170)

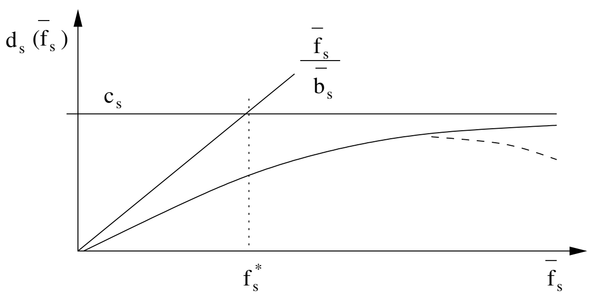
und die Einheit(en) i mit $\frac{c_i}{v_i} = c_s$ heißt *Engpaß von* S.

Das System S möge eine mittlere Bedienzeit $\overline{b_s}$ haben. Eine mittlere Füllung $\overline{f_s}$ von Aufträgen, einen Durchsatz d_s und eine mittlere Verweilzeit $\overline{y_s}$. Dann gilt (Little's Formel, vgl. (72))

$d_s = \frac{\overline{f_s}}{\overline{y_s}} \leq \frac{\overline{f_s}}{\overline{b_s}}$ da $\overline{y_s} \geq \overline{b_s}$ (171)

Außerdem gilt $d_s \leq c_s$

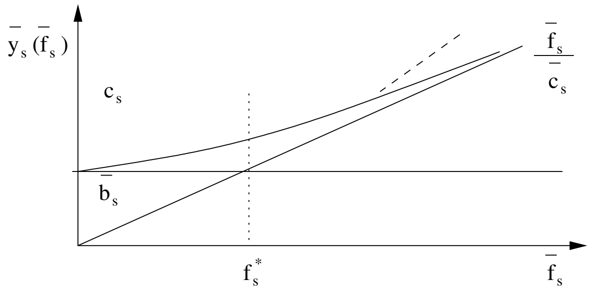
Also hat die Durchsatzfunktion $d_s = d_s(\overline{f_s})$ die Form..



(172)

entsprechendes gilt für die Verweilzeit

$\overline{y_s} = \frac{\overline{f_s}}{d_s} \geq \frac{\overline{f_s}}{c_s}$ und $\overline{y_s} \geq \overline{b_s}$ (173)



(174)

Die Größe f_s^* ("Sättigungsfüllung")

- erlaubt zwischen schwach belastetem System ($\overline{f_s} < f_s^*$) und stark belastetem System ($\overline{f_s} > f_s^*$) zu unterscheiden
- gibt an, wieviel Aufträge mindestens im System sein müssen, damit der Durchsatz ds.. in die Nähe von cs.. kommt
- berechnet sich als $f_s^* = \overline{b_s} * c_s$ (175)

Die Annäherung der Kurven an die Asymptoten (und damit die Leistung des Systems) ist umso besser, je

- weniger (fast) Engpässe bestehen,
- weniger die Bedienzeiten b_i in den Einheiten streuen,
- je verweilzeitgünsiger die Bedienstrategien in den Einheiten i sind.

Für das Verbindungswerk ist es offenbar wichtig, den Verkehr so zu lenken, das $\frac{c_i}{v_i}$ überall etwa gleich ist (Vermeidung von ausgeprägten Engpässen, hot spots). Dies kann durch Routing oder geeignetes Mapping geschehen.

—

Beispiel: Randomized Shared Memory, RSM. Ein Multirechner mit k Rechenwerken wird als Multiprozessor globalem verteiltem Speicher betrieben. Die Adressen werden *zufällig* auf die Speicher verteilt, um ausgeprägte Engpässe im Verbindungswerk zu vermeiden. Bei Zugriff auf einen entfernten Speicher wird der Thread ausgesetzt, bis der Zugriff erledigt ist; ein Anderer wird aufgenommen. (176)

Jeder Prozessor verlange je Taktzeit des mehrstufigen ($ld(k)$ Stufen) Verbindungswerkes einen Speicherzugriff (also geforderter Durchsatz je Prozessor $d_s' = \frac{1}{a}$; a Taktzeit Verbindungswerk). (177)

Über die $ld(k)$ Stufen ist die Zugriffszeit (Lesen, Vernachlässigung der Zugriffszeit im Speicher selbst)

$b_s = 2 * a * ld(k)$ (178)

Je Verbindung ist

$c_s' = \frac{1}{a}$ (179)

Für alle k Prozessoren ist

$c_s = \frac{k}{a}$ (z.B. Delta-Netz) (180)

Um c'_s zu erreichen, müssen je Prozessor

$f^*_s = b'_s * c'_s = 2 * a * ld(k) * \frac{1}{a} = 2 * ld(k)$ (181)

(mindestens!) unterwegs sein. Also muß der Prozessor $2 * ld(k)$ Threads bereithalten, d.h. er benötigt $2 * ld(k)$ Registersätze, um kurze Umschaltzeit zu ermöglichen.

Auf jedem Prozessor laufen also (ideal) $2 * ld(k)$ Threads, damit *trotz* der hohen Speicherzugriffszeit von $2 * a * ld(k)$ der Prozessor seinen vollen Durchsatz erreicht ($2 * ld(k)$ Threads von je $\frac{1}{2*ld(k)}$ Durchsatz). Wenn alle Threads aus *derselben hochparallelen Aufgabe* genommen werden, läuft die Maschine mit $k * ld(k)$ Threads (alle Prozessoren zusammen) mit voller Leistung ! Die hohe Zugriffszeit $2 * a * ld(k)$ bleibt verborgen ! Die Technik wird daher auch als latency hiding bezeichnet. (182)

—

2.3 Beispiele

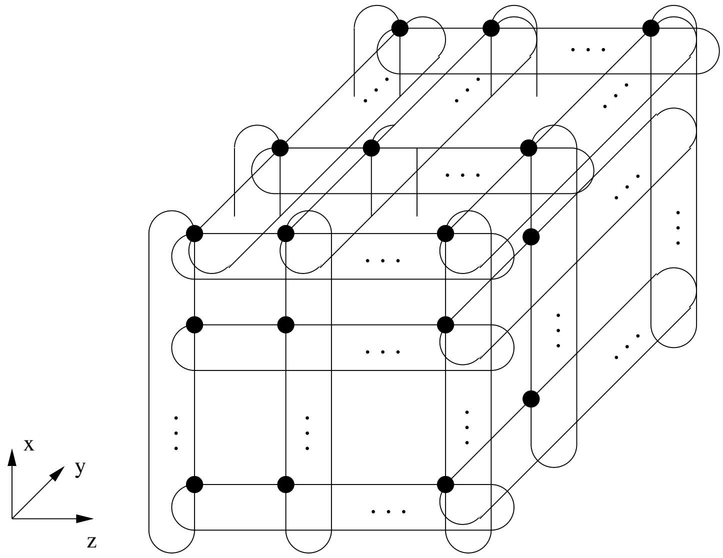
2.3.1 Stand der Technik: tabellarische Übersicht von Hochleistungsparallelrechnern
2.3.2 Cray T3D/T3E
2.3.3 Intel ASCI RED
2.3.4 SGI Origin 2000 vgl. Kapitel 1.3

2.3.1 Stand der Technik

Hersteller Produkt	CPU Typ Freq.	CPU c_{CPU} GFlops/s	CPU max. Anzahl	Rang	Anz. CPU	Linpack GFlops/s	Verbindungswerk
(SGI) Cray T3D	DEC Alpha 21064	0,15	1024	108	512	51	3D-Torus (Quader mit wrap-araound-Verbindungen) aus 2er-Knoten. Separates Synchronisationsnetz als Baum
(SGI) Cray T3E	DEC Alpha 21164 500 MHz	0,9	2048	2	1048	890	-“-
Fujitsu VPP 700	proprietär Vektor 147 MHz	2,2	256	48	56	110	flache Crossbar- Hierarchie
IBM ASCI Blue Pacific		1	3072	4	1344	470	?
IBM SP2	Power PC 604 u. A.	0,5	1024	33	512	157	Crossbar-Hierarchie
Intel ASCI RED	Pentium Pro 200 MHz	0,2	9072	1	9152	1400	zweifaches 2D-Gitter aus (2×2)-Knoten
SGI Origin 2000	Mips R10000 200 MHz	0,4	128	34	512	150	“fetter” Hyperwürfel aus 4er-Knoten
SUN Enterprise 10000	Ultra Sparc 1 333 MHz		64	41	256	120	16×16 Crossbar für 4er-Knoten

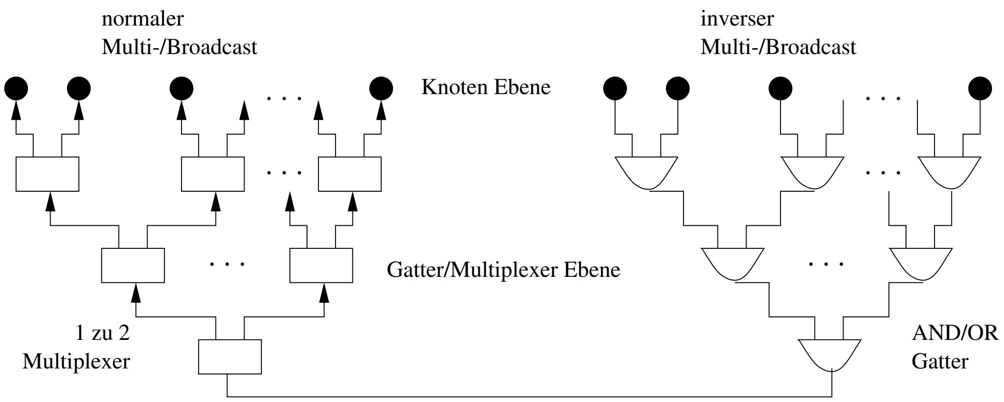
2.3.2 SGI/Cray T3D/T3E

Das Verbindungswerk ist ein 3D-Torus und ein Baum-Synchronisationswerk: Bilder aus [Richter H 97]



(184)

Datentransfertorus der Cray-Parallelrechner

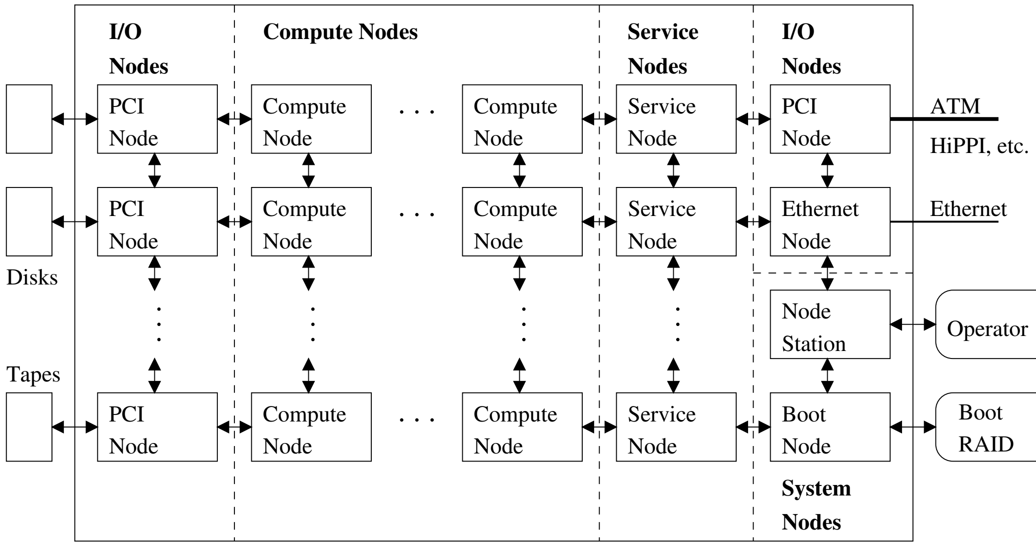


(185)

Die Bestandteile des Synchronisationsnetzes

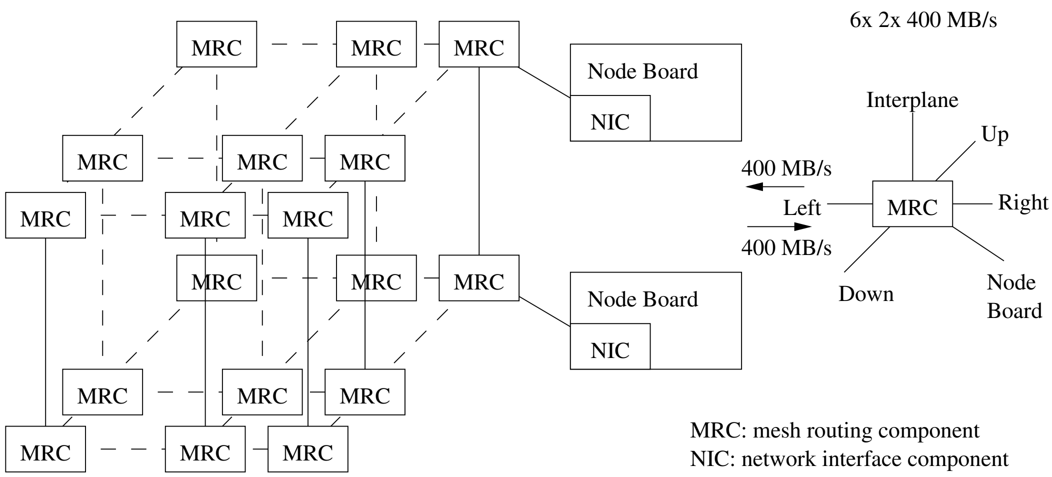
2.3.3 Intel ASCI Option RED

Das Verbindungswerk ist eine (zwei/mehrschichtiges) ebenes Gitter: Bilder aus [Hwang Xu 98]:



(186)

The ASCI Option Red system block diagram. (Courtesy of Mattson et al. Proc. of the 10th Int'l Parallel Processing Symp [433], 1996)



(187)

The Option Red Interconnection architecture (Courtesy of Mattson et al. Proc. of the 10th Int'l Parallel Processing Symp [433], 1996)

3 Single Instruction Stream - Multiple Data Stream (SIMD)

3.1 Datenparallelität und zugehörige Maschinentypen

Datenparallelität: (188)

SIMD: Auf allen in die Bearbeitung einbezogenen Daten wird dieselbe oder keine Operation vollzogen. Das Befehlswerk ist eine zentrale Funktion. Synchronisierung nach jeder Operation.

- *Feldrechner*: k Rechenwerke mit lokalen Datenspeichern an einem Verbindungswerk (Gitter, Hypercube).
- *Assoziativrechner*: k assoziative und rechenfähige (Langwort-) Zellen eines Speichers. Nicht mehr aktuell.
- *Vektorrechner*: s-stufige arithmetisch-logische Pipeline führt überlappend dieselbe Operation an den Daten durch, oder keine (maskengesteuert). Kein echter Parallelrechner.

SPMD: Betriebsform eines MIMD (Multiprozessor, Multirechner): Kurze, vorwiegend dieselbe Operationsfolge auf allen Prozessoren/Rechnern, danach globale Synchronisation.

Datenparallelität (insbesondere SIMD)

- ist eine stark eingeschränkte, bei Verzweigungen ineffiziente Parallelität, nicht aber bei gleichförmiger Verarbeitung regulärer Datenstrukturen
- bedient eine sehr wichtige Anwendungsklasse (insbesondere partielle Differentialgleichungen) und liefert dort Rechenverfahren hoher Breite
- verlangt häufig nur lokale Kommunikation, wenn die Datenstruktur auf das Verbindungswerk gut abgebildet werden kann. Hier bewähren sich Gitterstrukturen, vgl. Cray T3D/E.

—

Beispiel: In einem datenabhängig stark verzweigenden Verfahren erreicht ein SIMD paralleler Rechner nur einen Speedup von $O(\log(p))$. (189)

—

Etwa 1960 äußerte der amerikanische K.I.-Pionier Marvin Minsky die Vermutung, hochparallele Systeme erlaubten nur einen Speedup $O(\log(p))$. Viele Erkenntnisse (und Beispiele der Vorlesung) sprechen *gegen* die Vermutung.

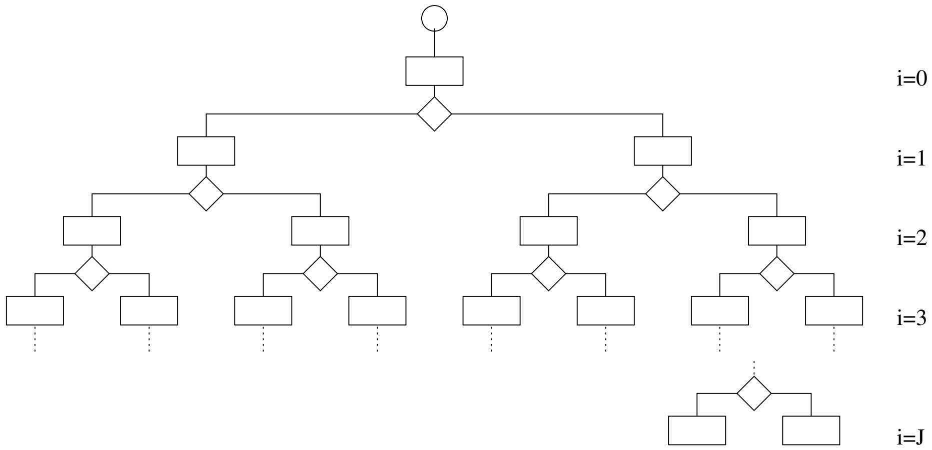
Auf Flynn (1968) geht das nachfolgende $O(\log(p))$-Beispiel für einen Feldrechner zurück. Man mache sich klar, daß das Beispiel

- *nicht für Vektorrechner* gilt
- auf einem MIMD als Beispiel vollständiger Nebenläufigkeit (6.4.) nicht wesentlich unterhalb $s(p) = p$ laufen dürfte

—

Demonstrationsbeispiel für Minsky's Vermutung auf Feldrechnern/Assoziativrechnern. (190)

Das Programm habe die folgende Struktur:



Das Programm hat 2^J Ausgänge. Es mögen alle Ausgänge einmal gewählt werden bei $n = 2^J$ Datensätzen. Eine SISD-Maschine macht $v_{SISD} = n * (J + 1) = 2^J(J + 1)$ Schritte, also $b(1) = 2^J(J + 1)$ für den Job.

Ein Feldrechner/Assoziativrechner stelle $p = 2^J$ Werke/Zellen zur Verfügung. Er durchläuft jedes Kästchen genau einmal mit 2^{J-1} Datensätzen, macht also $v_{FR/AR} = P(1+2+...+2^J) = p * (2^{J+1} - 1)$ Schritte, in der Bedienzeit $b(p) = 2^{J+1} - 1$. Also ist der Speedup

$$s(p) = \frac{b(1)}{b(p)} = \frac{2^J(J+1)}{2^{J+1}-1} = \frac{J+1}{2-2^{-J}} = \frac{ld(p)+1}{2-\frac{1}{p}} \quad (191)$$

Das gilt nicht für Vektorrechner oder MIMD !

—

3.2 Feldrechner und Vektorrechner

Feldrechner	*Vektorrechner*
Gitter / Torus / Hypercube-Topologie	eine oder mehrere Vektor-Pipelines
Leistungssteigerung durch k Rechenwerke ($k \in [64...64k]$) sehr groß, aber begrenzt durch Verteilung/Sammlung, Kommunikation	Leistungssteigerung durch geringe Stufigkeit der Pipelines begrenzt, außer dem Start-up (Einlauf-Zeit)
synchroner Zugriff auf verteilten Datenspeicher	versetzte Zugriffe auf zentralen Speicher
schwierige, i.a. veränderliche Abbildung der Berechnungen auf Werke und Zeit	einfache Abbildung der Berechnungen auf die Zeit
schwache parallelisierende Compiler	gute vektorisierende Compiler
billige Wiederholteile	keine Wiederholteile
Redundanz nicht nutzbar	keine Redundanz
seit 1962 (Solomon), 1972 (ILLI AC IV)	seit 1974 (Star 100), 1976 (Cray 1)

3.3 Feldrechner: Connection Machine CM-2

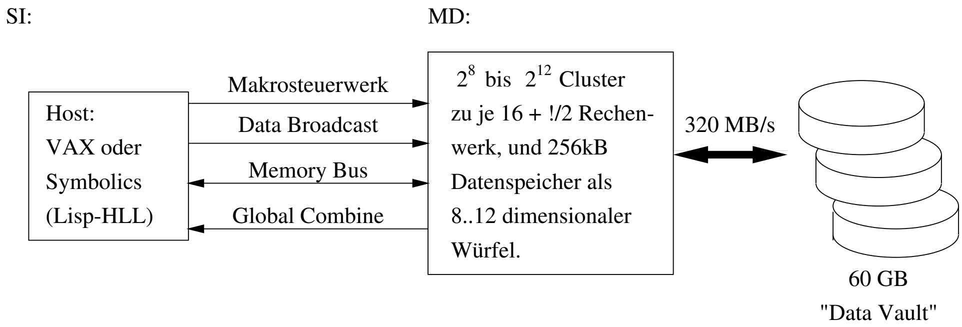
Die CM-2 entstand aus einer Laborentwicklung am M.I.T., von Hillis (CM-1). Daraus Firmengründung "Thinking Machines". CM-2 war ein gut verkauftes Produkt, ca. 50 verkauft.

Die CM-2 war Manifest der *konnekionistischen* Denkweise: nicht mehr die Sequenz von Elementaroperationen, wie bei von Neumann würde die Eroberung neuer Reiche der Berechnung erlauben, sondern die "synergistische" Verkoppelung sehr vieler (einfacher) Zellen, wie im Gehirn. Nie wieder ist eine Maschine mit so vielen Werken (64k) gebaut worden!

Die spätere CM-5 war eine MIMD-Maschine, vorzugsweise für SPMD.

Connection Machine CM-2

Literatur: [Hwang K 93] Hwang, K.: Advanced Computer Architecture, McGraw Hill 1993



(192)

Host (Betriebssystem, Leitwerk (!) emittiert Makrosteuerworte an alle Cluster gleichzeitig, dort von lokalem Sequencer in Mikrosteuerworte umgesetzt.

Cluster zu 16 *bitseriellen* Rechenwerken (CM-1: 1 Bit/μs, d.h. 2^{-5} Mips je RW, bei 32 Bit Operanden, ganze CM-1 etwa 2048 Mips Peakrate).

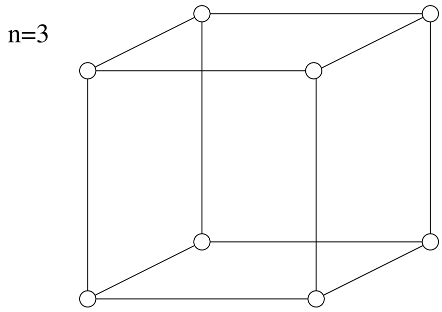
Je zwei Cluster teilen sich ein 32 Bit-Gleitpunkt-Rechenwerk von etwa 14 Mips. D.h. 28 GFlops für die voll ausgebaute Maschine (Peakrate ! realistisch 20 %). Je zwei Cluster teilen sich 512 kB Datenspeicher. Also 1 GB für die voll ausgebaute Maschine.

Die 16 1 Bit-Rechenwerke sind mit Registern (u.a. für Bedingung, ob an der aktuellen Operation beteiligt) mit dem Router

- für die Kommunikation in einem Gitter von 4×4 Werken (News, north east west south)
- für Hyperwürfel-Kommunikation

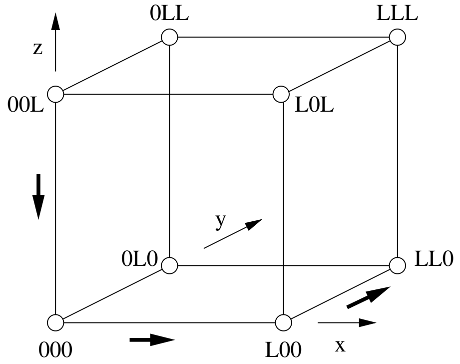
auf einem Chip zusammengefaßt. Die Werke können im Zeitmultiplex je bis zu 64 virtuelle Werke darstellen. Damit kann die Parallelität p über die physische Parallelität $k \leq 2^{16} = 65536$ Werke erhöht werden.

Hyperwürfel (Hypercube)



Verbindet 2^n Knoten mit $n * 2^{n-1}$ Kanten der Weglänge (höchstens) n, bei n direkten Nachbarn. D.h. n "Ports" je Knoten.

Inter-Cluster-Kommunikation nach Prinzip der Paketvermittlung. (193)



Routing durch Auswerten der Adressbits. Vgl. obiges 3-dimensionales Beispiel (Abbildung) von OOL nach LLO.

Ort	Differenz (Antivalenz) zur Adresse LLO		Routingmöglichkeiten
OOL (Start)	xyz=	LLL	$x, y, \underline{z}$
OOO		LLO	$\underline{x}, y$
LOO		OLO	$\underline{y}$
LLO		OOO (Ziel!)	

Die Auswahl aus x, y, z kann nach Belegung der Leitungen erfolgen.

Etwa 250 μs für 32 Bit-Nachricht über mittleren Abstand.

Technischer Aufbau der CM-2: Würfel $(1,5m)^3$, 28 kW, 1,2 t

Programmierung über Host (Compiler,...) in C*, *LISP (Erweiterungen von C, Common LISP in Hinblick auf Parallelität)

Anwendungen:

- Information Retrieval (z.B. 15000 Artikel zu im Mittel 400 Wörtern gegen 200 Wörter prüfen, bestpassende 20 finden in 60 ms)
- Bildverarbeitung
- Festigkeits- /Elastizitätsmodelle (Methode der finiten Elemente)
- Graphik
- Zellulare Automaten als Modelle
- Aufstellen von Regeln aus Beobachtungen

3.4 Vektorrechner: Cray YMP, C-90

Vektorrechner sind keine Parallelrechner nach unserer ursprünglichen Definition. Es sind SIMD-Maschinen, da sie Befehle haben, deren Operanden Vektoren sind, und die Komponenten der Vektoren werden kollateral in Pipelines verarbeitet.

Vektorrechner gibt es seit 1972 (*Control Data Star 100*, Texas Instruments TI ASC); technische Bedeutung erhielten sie als Produkte der Firma Cray (*Cray-1*, 1975). *Seymour Cray* war bereits

Schöpfer der Control Data CDC 6600 (1963; eine revolutionäre Maschine: Hauptprozessor mit 10 funktionsspezialisierten Rechenwerken, 10 Nebenprozessoren (Ein-/Ausgabe, Betriebssystem), load/store-Prinzip, Kathodenstrahl-Displays für Operateure, Flüssigkeitskühlung, mit 2,2 Mips 3x schneller als bisherige Rechner).

Cray: Cray-1 (1975), Cray-2 und Mehrprozessoranlagen, Cray-X-MP, Y-MP (1988), C-90 (1992) waren 1976-1995 die führenden Superrechner; anfangs als Einzel-Vektorprozessor, ab X-MP als Multivektorprozessor (PVP, poly vector processor); Ab C-90 zusätzlich mit (Sklaven-) MIMD-Rechner auf der Basis von DEC Alpha-Prozessoren; "MPP" an der C-90; später T3D, T3E.

Mitte der 90er waren japanische Vektorrechner schneller, was in den USA Regierungsaktionen zur Förderung der US Amerikanischen Superrechner auslöste.

Die Vektorrechner setzten zugleich alle vorhandenen strukturellen Mittel zur Leistungssteigerung ein und benutzten bis tief in die 90er hinein diskrete Bauelemente und sehr aufwendige Kühl- und Packungstechnik.

Der *Fujitsu/Siemens VPP 700* ist ein massiv-paralleler Rechner aus Vektorrechnern, deren Prozessoren für diesen Zweck entwickelt wurden, vgl. 4.4.4. ! Unter dem Namen Multimedia bzw. SIMD halten Vektoroperationen Einzug in konventionelle Prozessoren, vgl. Pentium MMX, Pentium III.

Vektorrechner (wie alle SIMDs) haben einen wesentlich höheren Operationsdurchsatz (MFlops) als Befehlsdurchsatz (Mips). Sie haben eine bezüglich Zugriffszeit verhältnismäßig anspruchslose Speicherschnittstelle, da die Operandenzugriffe (Vektorkomponenten!) vorhersagbar sind ("streaming"). Der Speicher braucht hohen Grenzdurchsatz (also verschränkt). Oft kommen die Maschinen ohne Cache aus; aber Vektor-Register für Zwischenwerte.

Problematisch: Anlaufzeit (setup time) bei kurzen Vektoren; Leistungsfähigkeit bei nicht vektorisierbarem Anteil.

Stand "klassische" Vektorrechner-Technologie im Jahre 2000:

Vektor CPUs erreichen "nur" 0,5...1,5 GFlops/s (4 Pipes, 300 MHz). Höhere Leistung nur durch Parallelisierung von Prozessoren, vgl. Hitachi SR 8000.

—

Beispiel Cray SV-1 (1999): Ring (800 Mb/s) aus bis zu 32 Multiprozessor-Knoten (SMPs). Knoten aus höchstens 8 CPUs a 1,2 GFlop/s (2 Pipes, 300 MHz) *und* höchstens 6 "Multi-Streaming Processors" (4 CPUs im synchron-parallel-Betrieb (wie Hitachi), damit 4,8 GFlop/s), also je Knoten 32 x 1,2 = 38 GFlop/s peak; Etwa 1 TFlop/s im voll ausgebauten Ring, bei 1 TB Hauptspeicher

oder

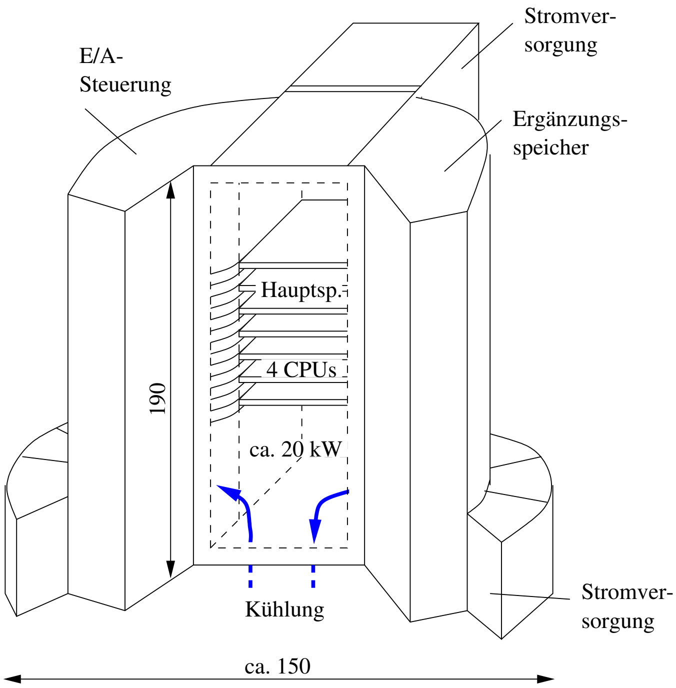
Beispiel Cray SV-2 (2000): Multirechner auf Gitter, Nachfolger der T3E (siehe Kap. 2.3.2.), bis 1200 Rechner zu 1,2 GFlop/s Peak.

—

SGI hat inzwischen Cray wieder verkauft an *TERA*.

Beispiel zur Aufbau-/Packungstechnik eines neuen Hochleistungsrechners:

Cray YMP im Leibniz-Rechenzentrum München (4 CPUs a max. 366 Mflop/s)



- Mit Peripherie ca. 25' Mio DM
- 2 Jahre in Betrieb
- Ein-Schicht-Anwesenheitswarnung
- Alle 2 Wochen 4h vorbeugende Warnung (80%: Peripherie)
- Alle 3 Monate marginal checks (Spannung, Frequenz)
- Zentrale Einheit nur flüssigkeitsgekühlt (Fluorinert), mit 2-stufigem Austauscher Wasser/Luft.
- Diagnose (Programm, Hardware) identifiziert schadhafte CPU bzw. Speicherchip.
- Eine CPU als Ersatzteil vorrätig (1 Mio DM?).
- 5000 h mean time between failure für Zentraleinheit, aber häufiger vorsorglicher Bausteintausch.

Cray Y-MP (1988)

Mehrprozessorsystem aus 8 Prozessoren mit je 4 Pipelines für Vektor/Skalar/Adressrechnung.

Peakrate 366 MFlops je Prozessor, 3 GFlops für System (mittlerer CPU-Durchsatz etwa 20...30%). 6 ns Taktzeit.

Load-Store Prinzip (setzt die Setup-Zeit für den Zugriff der ersten Vektor-Komponente hinunter, wichtig für kurze Vektoren).

Mehrstufiger Kreuzschienen Verteiler zwischen 8 Prozessoren mit je 4 Anschlüssen und 32 Anschlüssen des zu 256fach verschränkten Speichers.

1 GB Hauptspeicher (30ns), 4 GB Ergänzungsspeicher (25 μs).

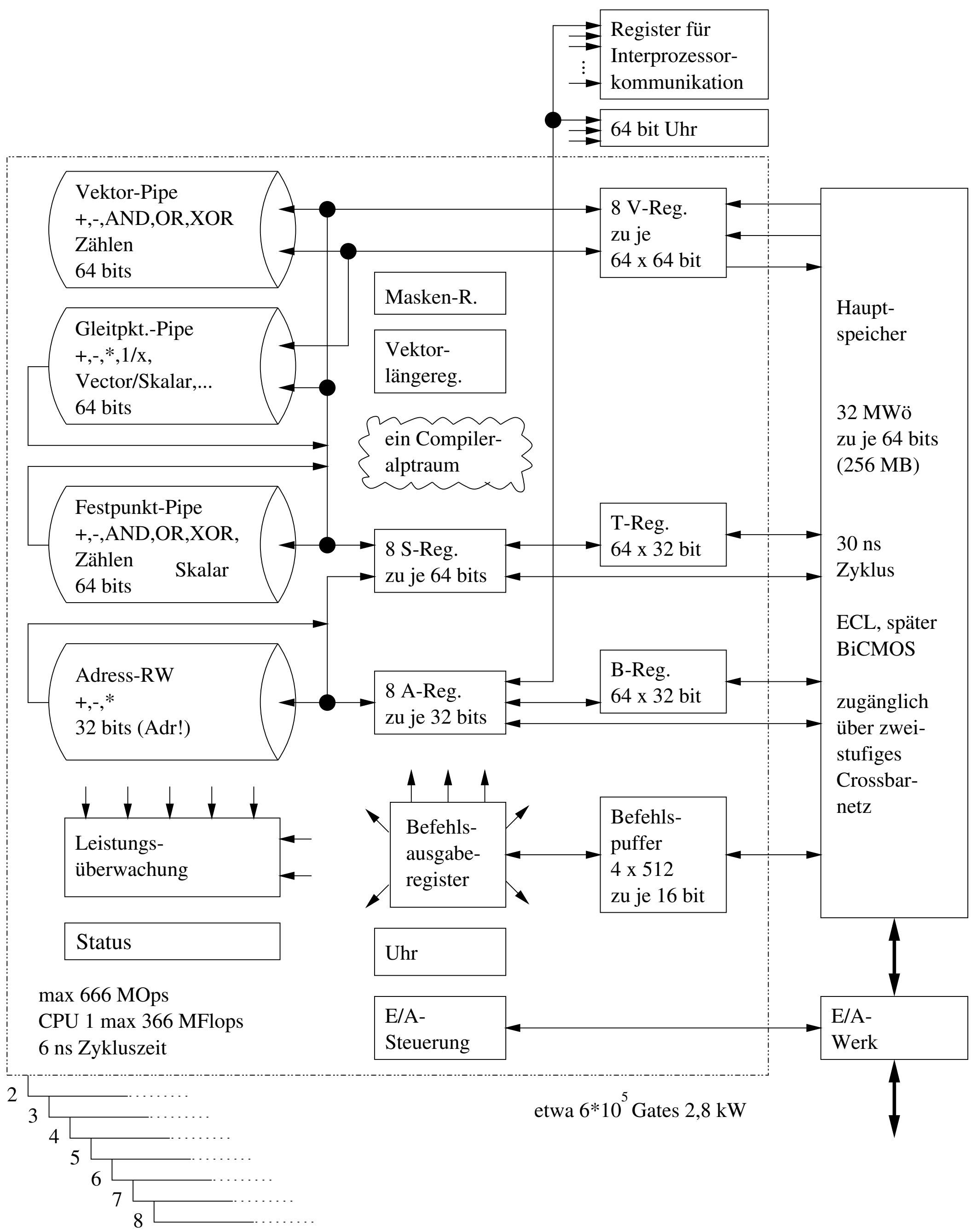
Cray C-90 (1992)

Wie Y-MP aber 16 Prozessoren. 520 MFlops/CPU bzw. 1 GFlops Peakrate bei 4,2 ns Taktzeit.

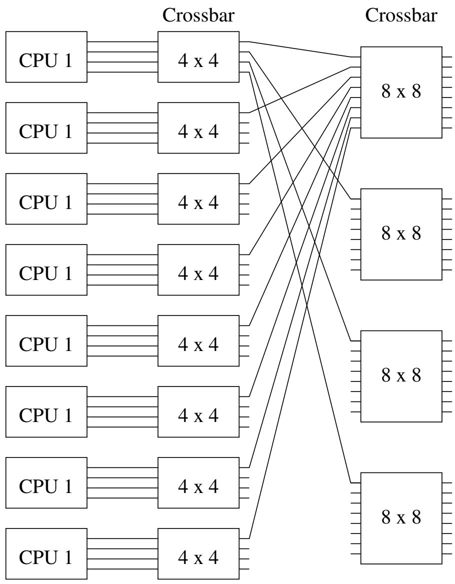
2 GB Hauptspeicher.

Aufbau von Multi-C-90-Konfigurationen durch gemeinsamen Zugriff auf den Ergänzungsspeicher und ein Semaphore Unit.

Cray Y-MP



Crossbar (Kreuzschienenverteiler)-Netz zwischen 8 Prozessoren und 32 Anschlüssen des 256fach verschränkten Speichers bei Y-MP: (195)



An den 8×8 Crossbars sind die Moduln des Hauptspeichers angeschlossen, z.B. erster Anschluß an Modul 0, 4, 8, 12, 16, 20, 24, 28.

Ein $k \times k$-Kreuzschienenverteiler verbindet jeden der k Eingänge mit einem der k Ausgänge, kann also bis k Nachrichtenströme aufrechterhalten.

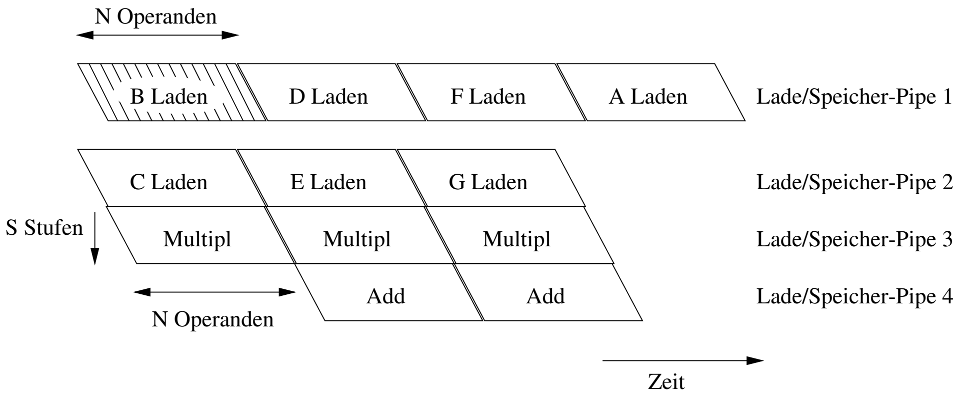
Elementare Beispiele zu *vektorisierendem Compiler und Chaining* (196)

```
DO 10 I=1,N
  A(I)=B(I)*C(I)+D(I)*E(I)+F(I)*G(I)
10 CONTINUE
```

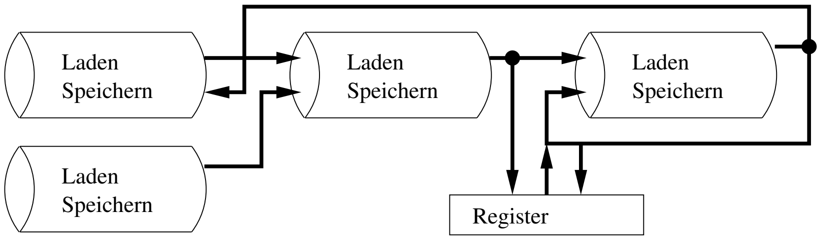
wird transformiert in

$X_i=B_i*C_i \qquad i:\ 1\to N$
$Y_i=D_i*E_i \qquad i:\ 1\to N$
$Z_i=F_i*G_i \qquad i:\ 1\to N$
$X_i=X_i+Y_i \qquad i:\ 1\to N$
$A_i=X_i+Z_i \qquad i:\ 1\to N$

zeitliche Ordnung:



Laden/Speichern-Pipe entfällt hier, wenn Operanden/Ergebnis im Register steht!



Typische Schwierigkeiten bei Bedingungen in Schleifen durch Binärvektoren (Masken) behandelt: (197)

```
DO 10 I=1,N
    IF (E(I).LE.0.0) GO TO 9
    A(I)=B(I)*C(I)+D(I)*E(I)+F(I)*G(I)
9   CONTINUE
10  CONTINUE
```

wird transformiert in

```
Mi=E(I).GT.0.0   i: 1→N
Xi=Bi*Ci         i: 1→N
Yi=Di*Ei         i: 1→N
Zi=Fi*Gi         i: 1→N
Xi=Xi+Yi         i: 1→N
Ai=Xi+Zi  unter Mi  i: 1→N
```

Unter M_i heißt: Anweisung wird für M_i=0 nicht durchgeführt

Die Maske dient auch dazu, schwach besetzte Vektoren für die Bearbeitung zu komprimieren. Ist auch für unser Beispiel nützlich.

4 Multiple Instruction Stream-Multiple Data Stream (MIMD, M(SISD))

4.1 Klassifikation, Erscheinungen

4.2 Konsistenz und Kohärenz

4.2.1 Probleme, Begriffe

4.2.2 Cache - Kohärenzverfahren

4.2.3 Ablaufkonsistenz im gemeinsamen Speicher

4.3 Multiprozessoren

4.3.1 Uniform Memory Access (UMA, SMP) (Intel SHV, Sun Ultra Enterprise 10000)

4.3.2 Non-Uniform Memory Access (NUMA, DSM)

4.3.2.1 ccNUMA: Cache Coherent NUMA (SGI Origin 2000)

4.3.2.2 COMA: Cache-only Memory Architecture (KSR 1,2)

4.3.2.3 nccNUMA: Non-Cache Coherent NUMA (SGIICray T3/E)

4.4 Multirechner (Beispiel: Intel ASCI Option Red, IBM RS 6000 SP, Workstation-Cluster)

Literatur: [Ungerer T 97] (Gut!), [Hwang Xu 98], [Hellwagner H98]

Die Welt der MIMD's:

- MIMD (M(SISD))
  - Multiprozessoren: ein Adressraum! Speicherkopplung/gemeins. Hauptspeicher Ablaufkonsistenz im gemeins. Hauptspeicher! Effiziente Kommunikation möglich mittlere bis grobe Granularität.
    - UMA uniform memory acc. (SMP shared memory proc.) Cache Kohärenz ! Bus snooping / Verzeichn. bis zu 64 CPUs Beispiele: Intel, SUN, SGI Power Challenge. Keine Lastverschiebung, ein Betriebssystem
    - NUMA non uniform memory access (DSM distrib. shared memory); Physisch ein Multirechner, aber globaler Speicher. Latency hiding! Mikrokernels, aber single system image
      - cc-NUMA cache consistent NUMA. Cache Kohärenz (!) über Verzeichnisse. Beispiel: SGI Origin 2000
      - COMA cache only memory access. Alle Speicher wie Cache; Inhalte wandern. Beispiel: Kendall Square Res. KSR 1
      - ncc-NUMA non cache consistent NUMA. Caches nur Rechner-lokal kohärent. Beispiel: SGI / Cray T3D / E
  - Multirechner: NORMA. Jeder Rechner hat eig. Adressraum, Hauptspeicher, Betriebssystem. Nachrichtenkopplung: keine effiziente Kommunikation, grobe Granularität.
    - MPP massiv parallele Systeme, konzentriert, mehr als 100 Rechner. Beispiel: Intel ASCI RED, Teraflops !
    - Worksation Cluster (Farms) Sicherheit! Kommunikationsengpaß Beispiel: Viele!
    - Metacomputing Multirechner aus (heterogenen) Superrechnern Beispiele: Wenige
    - Cluster, konzentriert, bis zu 100 Rechern. Beispiel: IBM RS/6000 SP

4.2 Konsistenz und Kohärenz

4.2.1 Probleme, Begriffe

Konsistent heißt ein Zustand / eine Informationsstruktur / ein Ablauf, der in allen seinen Komponenten *semantisch korrekt* ist. (199)

—

Beispiele: Eine Datenbank ist vor Beginn und nach Ende einer Transaktion konsistent. (200)

Wenn es von einer Datenstruktur mehrere Exemplare gibt (Kopien), dann verlangt die Konsistenz, daß ihr Wert gleich ist.

Die von Neumannsche Maschine führt eine serielle Operationsfolge mit Lese- und Schreiboperationen durch. Überholungen der Lese- und Schreiboperationen sind inkonsistent.

—

In den beiden letzten Beispielen werden faktisch in Rechnern *Konsistenzverletzungen aus Leistungsgründen* hingenommen!

Konsistenzverletzungen können aus Leistungsgründen hingenommen werden, wenn trotzdem (auf höherer Ebene, nach hinreichend langer Zeit) korrekte (evtl. akzeptabel korrekte) Ergebnisse erarbeitet werden.

—

Beispiele: Zeitweise Abweichung zwischen Wörtern im Cache und im Hauptspeicher, zwischen Seiten im Hauptspeicher und auf der Platte. (201)

Aufschieben von Schreiboperationen gegenüber Leseoperationen am Hauptspeicher. (202)

—

Verfahren, die Inkonsistenzen beheben oder unsichtbar machen, heißen *Kohärenzverfahren*. Manchmal wird auch "Kohärenz" wie "Konsistenz" gebraucht.

Kohärenzverfahren sind in Multiprozessoren besonders kritisch und schwierig; in Einprozessor- und Mehrrechnersystemen sind die Probleme einfacher!

In *verteilten Systemen* gibt es eine weitere Spielart von Inkonsistenz: kein Beobachter kann eine große Teilmenge des globalen Zustandes zu einem Zeitpunkt mit Sicherheit korrekt erfassen: ein Prozessor hat faktisch asynchron streuende und nicht globale Zugriffsmöglichkeiten; auch wenn Zustandsänderungen konsistent verliefen, könnte er sie i.a. nicht konsistent beobachten! *Perspektivische Inkonsistenz* (203)

Also (204)

(4.2.1) Konsistenz und faktische Verletzung (Leistungsgründe!) und perspektivische Verletzung

(4.2.2) Inkonsistente Inhalte in Caches und Hauptspeicher, in Ein- und *Mehr*prozessorsystemen

(4.2.3) Nicht konsistente oder perspektivisch nicht konsistente Schreib / Lesefolgen im gemeinsamen Hauptspeicher in Mehrprozessorsystemen. Verschiedene Grade von Konsistenz

Kohärenzprotokolle

Gemeinsamer Bus: snooping / MESI-Kohärenzprotokoll

Kein gemeinsamer Bus: verzeichnisorientierte Protokolle (directories)

strikte Konsistenz

sequentielle Konsistenz

Prozessorkonsistenz

schwache Konsistenz

Freigabe / Entrykonsistenz

Konsistenzhierarchie

4.2.2 Cache - Kohärenzverfahren

Voraussetzungen für das Auftreten von Inkonsistenzen Instanzierungen von Objekten (Kopien): (205)

1. Es gibt mehr als eine Instanzierung (d.h. Existenz von Kopie(n)) *und*
2. wenigstens ein Zugreifer verändert eine Instanzierung (schreibt auf eine Kopie) *und*
3. der Zugreifer verändert nicht alle Instanzierungen zugleich

Abhilfen (205a)

1. Auf Kopien verzichten, vgl. Randomized Shared Memory Machine (176), i.a. nicht akzeptabel!
2. Auf Schreiben verzichten: nicht akzeptabel.
3. Alle Instanzierungen (hinreichend) gleichzeitig (store through) ändern oder die nicht geänderten Instanzierungen ungültig machen (invalidate).

Kohärenzverfahren mit ungültig machen:

One writer - n readers:

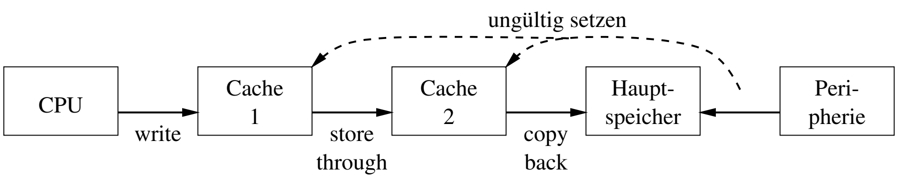
Jede Instanzierung ist in einem von zwei Zuständen:

- sie ist die einzige und darf beschrieben werden (durch ihren write owner), oder
- sie ist eine von n Instanzierungen, die gelesen werden (durch ihre read owners)

Das Verändern regelt die Zustandsübergänge mit folgenden Funktionen:

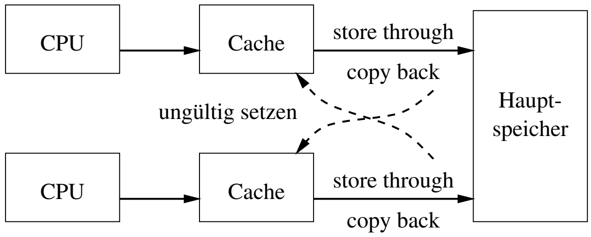
- Ungültigsetzen von n-1 Instanzierungen bei Übergang einer Instanzierung in den Schreibzustand.
- Herstellen von Instanzierungen (auf Anforderung neuer Zugreifer und bei Übergang in den Lesezustand).
- Verzeichnis(se) von Ort und Zustand und owner der Instanzierung.

Einprozessor **(206)**



Typische Regelung bei Einprozessor (und k mal bei (k-fachem) Multirechner)

Multiprozessor (hier $k = 2$) **(207)**



(vereinfacht: nur 1 Cache, keine Peripherie)

"Ungültig setzen" betrifft Kopien von Cache Blöcken, die von einem anderen Prozessor in den Speicher geschrieben wurden. Wird erkannt über

- mithören (snooping), am Bus, oder
- Verzeichnis(se), sonst

Mithörprotokoll (snooping protocol) **(208)**

Beruht auf Mithören (Adresse Schreiben/Lesen) am gemeinschaftlichen Bus:

Dieser serialisiert alle Vorgänge am Hauptspeicher, daher kann jeder Prozessor/Cache die Transaktionen aller anderen beobachten.

—

Spezielles Beispiel:

MESI-Protokoll, genannt nach dem vierwertigen Attribut der Cache-Blöcke

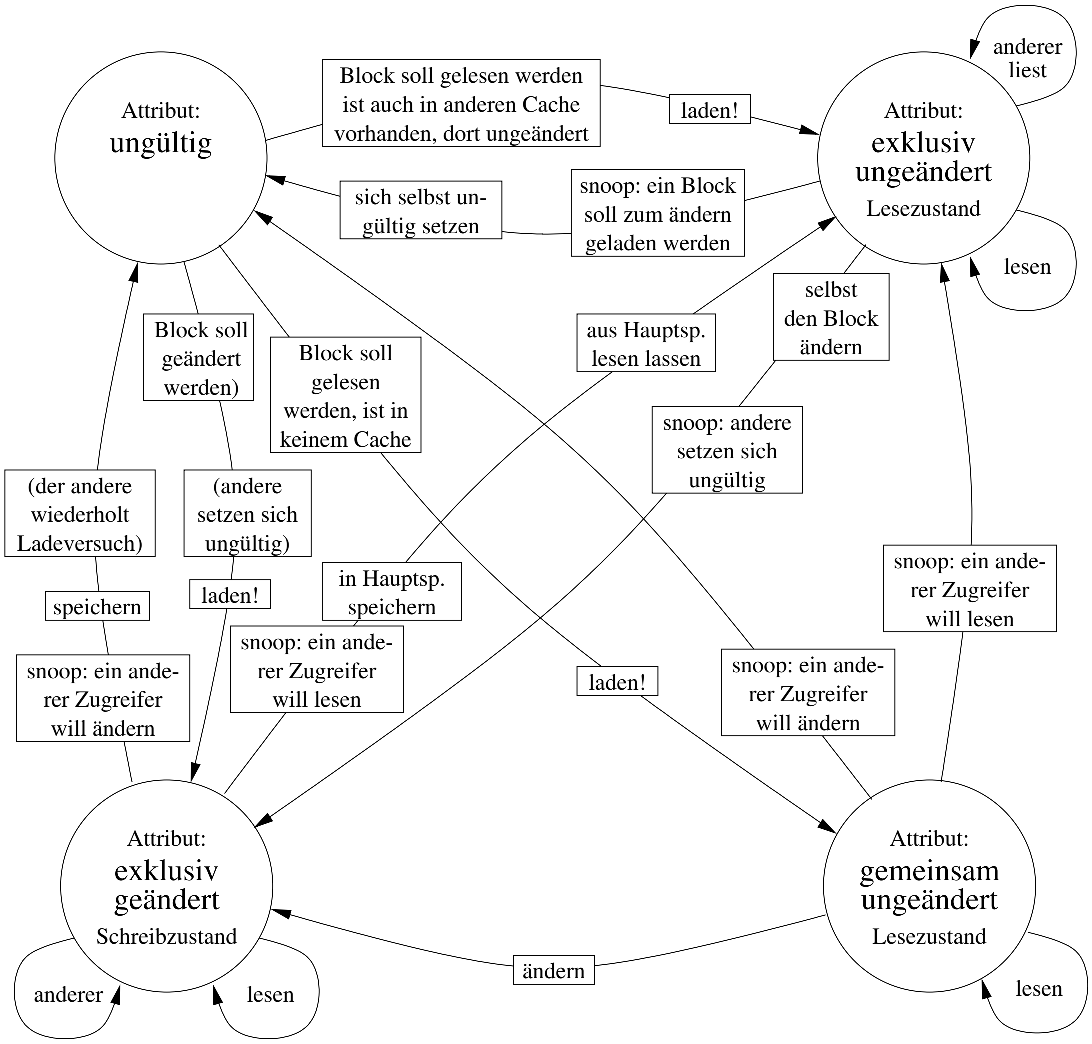
- Exclusive modified (exclusiv, geändert)
- Exclusive unmodified (exclusiv, ungeändert)
- Shared unmodified (gemeinsam, ungeändert)
- Invalid (ungültig)

Ein One Writer - n Reader - Protocol.

—

Begrenzt auf SMP (UMA) mit Bus!

MESI-Protokoll Attributübergänge der Cacheblöcke und Aktionen



(209)

Verzeichnis-basierte Verfahren (directory based protocol) **(210)**

Sind für die Cache-Konsistenz erforderlich, wenn kein Bus als Verbindungswerk gewählt wird:

- UMA (SMP) mit Crossbar
- NUMA - Multiprozessoren : dann cc-NUMA (Cache coherent - NUMA)

Verzeichnisse geben zu jedem Block an:

- wo er steht (mehrere / kein Ort möglich!)
- sein Attribut

Zentrale und dezentrale (hierarchische) Lösungen (Engpaßvermeidung, Skalierbarkeit), *Hardware*-/Software-basiert oder *Hybrid*.

↪ Beispiel SGI ORIGIN 2000 cc-NUMA SMP/NUMA mit directories

4.2.3 Ablaufkonsistenz im gemeinsamen Speicher

Die meisten heutigen Prozessoren ordnen Speicherzugriffe freizügig um und genehmigen sich zusätzliche Zugriffe (Lookahead, Cacheblock laden), um mehr Prozessordurchsatz zu erzielen. Sie verwenden dabei Verfahren, die *sie selbst* vor semantischen Fehlern schützen.

Solche Freizügigkeiten sind:

Gegenseitige Überholung der Fertigstellung von Operationen: (211)

Führt dazu, daß der nach Programmsemantik letztzuschreibende Wert von einem Vorläufer überschrieben wird. Aber Vorteil: Freigabe von Rechenwerksbetriebsmitteln, Vermeidung unnötigen Wartens bei nachfolgenden Leseoperationen auf derselben Zelle. *Abhilfe*: reorder buffer am Prozessor"ausgang", Merken unerledigter Schreibvorgänge, vgl. nächster Punkt.

Verzögertes Abspeichern aus Cache bei store through: (212)

Das Cache Memory wartet die Erledigung von Schreibaufträgen an den Hauptspeicher nicht ab, setzt die Aufträge in einen Schreibpuffer; Leseaufträge an den Hauptspeicher werden durch Adressvergleich darauf überprüft, ob im Schreibpuffer ein (oder mehrere) Auftrag steht; dann wird dessen Wert benutzt. Damit *Abhilfe* in kritischen Fällen (schützt nicht die anderen Prozessoren!)

Nichtblockierendes Cache Memory: (213)

Wenn am Cache Memory ein MISS auftritt, werden wartende andere Aufträge (auf anderen Adressen) weiterbedient. Führt zu irregulären Zugriffsfolgen;

Abhilfe in besonderen Programmsituationen durch Fence (Zaun)-Befehle, die alle anstehenden Lese- und Schreibaufträge erledigen, bevor der Befehl durchgeführt wird (z.B. Testen und Setzen des Inhalts einer Speicherzelle). Auch diese Abhilfe beseitigt nur die eigenen Probleme des Prozessors. (214)

Im Folgenden werden, ausgehend von strikter Konsistenz, fortschreitend weitherzigere *Konsistenzmodelle* dargestellt, die mehr Leistung erlauben (vor allem späte Vollendung von Schreiboperationen gestatten), aber semantisch komplexer und fehleranfälliger sind. Die Probleme stellen sich weitgehend unabhängig von der Cache-Kohärenz und von dem Problem der Synchronisation bei Zugriff auf gemeinsame Variable (Funktionalität ! vgl. 1.5.1)

Strikte Konsistenz: Für jeden Zugreifer und jeden Lesezugriff wird der letztgeschrieben Wert geliefert. (215)

(Ist wegen der vorgenannten Freizügigkeiten und wegen der perspektivischen Inkonsistenz nicht gegeben; Durchsetzung würde den Multiprozessor erheblich bremsen)

Sequentielle Konsistenz: Jeder Zugreifer sieht eine Speicheroperationsfolge, die ein Zeitmultiplex der Speicheroperationsfolgen aller Zugreifer ist: Jede Operationsfolge jedes einzelnen Prozessors wird korrekt beobachtet, aber der Zeitmultiplex kann vom Betrachter abhängen. (216)

—

Beispiel: Überholungen zwischen den Prozessen sind zulässig. (Distanz der Prozessoren zu den Speichern darf verschieden sein.)

—

Prozessor-Konsistenz: (217)

Wieder wird am Speicher die Zugriffsfolge jeden Prozessors korrekt vollzogen und von ihm auch richtig beobachtet. Fremde Zugreifer erhalten aber i.a. Bilder von Zugriffsfolgen mit Vertauschungen, z.B. auch von Schreiboperationen. (erlaubt also schwankend versetzte Lesezugriffszeiten)

Schwache Konsistenz: (218)

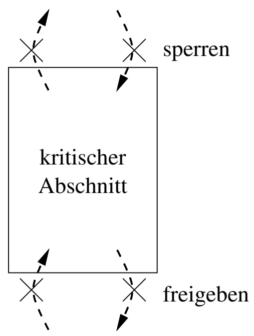
Die strikte Konsistenz wird bei Ausführung von Synchronisationsoperationen (sperren, freigeben) garantiert. Jede dieser Operationen wird erst ausgeführt, wenn alle ausstehenden Lese- und Schreiboperationen ausgeführt sind. Keine Lese- oder Schreiboperation darf vor eine Synchronisationsoperation gezogen werden. Die Folge der Sychronisationsoperationen ist sequentiell konsistent.

Das Motiv hinter der schwachen Konsistenz ist, daß "verantwortungsvoller" Umgang mit gemeinsamen Variablen ohnehin nur unter Sperren geschehen kann, und entsprechend wird lediglich sichergestellt, daß die Situation bei Eintritt in den kritischen Abschnitt konsistent ist und daß Operationen den kritischen Abschnitt nicht verlassen können (Ausführung vor Eintritt oder nach Austritt des kritischen Abschnitts).

Setzen der kritischen Abschnitte ist Sache des Programmierers/Compilers. Der Gewinn an Sicherheit/Funktionalität (vgl. 1.5.1) wird mit Overhead und mit Verlust an Nebenläufigkeit bezahlt!

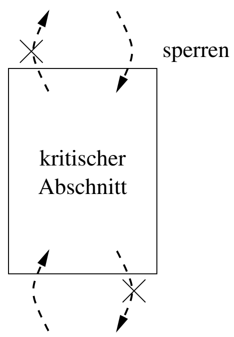
Freigabe-Konsistenz (218)

Die schwache Konsistenz läßt sich gefahrlos weiter abschwächen. Die Serialisierung in der Umgebung von Synchronisationsoperatoren schützt mehr als notwendig!



(219)

Ferner ist das Vorziehen einer Sperroperation vor eine Operation an einer Variablen und einer Operation an einer Variablen vor eine Freigabeoperation unbedenklich.



(220)

Entry-Konsistenz (221)

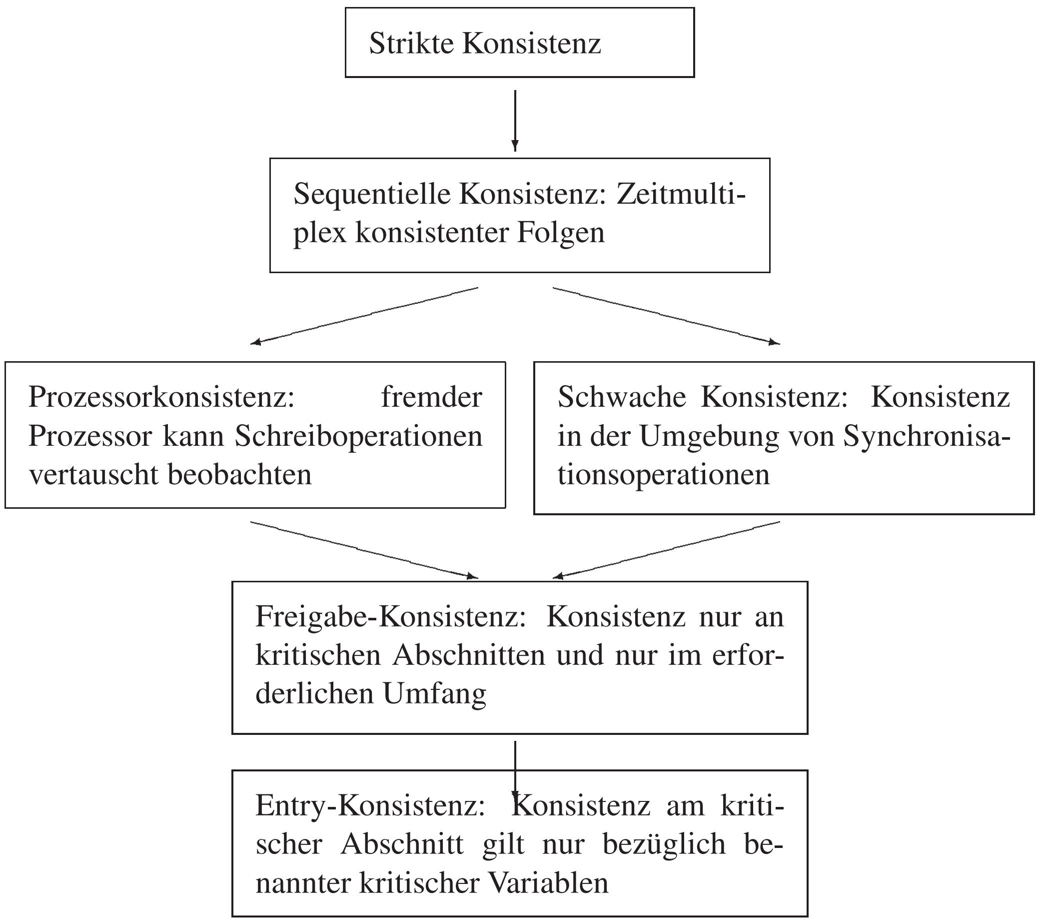
Eine weitere Abschwächung (aber auch Komplexitätssteigerung) ergibt sich, wenn die Serialisierung am kritischen Abschnitt nur bezüglich einer kritischen Variablen garantiert wird; die Sperrvariable (Semaphor, Lock) wird bezüglich dieser Variablen deklariert.

Ob eines der abgeschwächten Konsistenzmodelle Vorteile bringt, hängt stark von der Anwendung ab!

Abschließende Bemerkung zu den Konsistenzmodellen: Besondere Konsistenzprobleme haben auch die Autoren: Man vergleiche [Hennesy, Patterson 96] mit [Hwang Xu 98] mit [Ungert 97].

Hierarchie der Konsistenzmodelle **(222)**

$\boxed{\text{a}} \longrightarrow \boxed{\text{b}}$ heißt: a ist konsistenter als b



4.3 Multiprozessoren

Wir wissen schon ziemlich viel: (223)

- ein Adressraum
- UMA/SMP: Physische Speicher (i.a. viele Moduln, gleich zugänglich, gleich erreichbar)
- NUMA, cc-NUMA, COMA: Physisch viele Speicher, ungleich zugänglich
- Programmierparadigma gemeinsame Variable (aber nicht SPMD, Nachrichtenkopplung möglich)
- Probleme Cache-Kohärenz und Ablauf-Konsistenz am Speicher

4.3.1 Uniform Memory Access (UMA, SMP shared memory processor) (224)

Wieder wissen wir schon ziemlich viel:

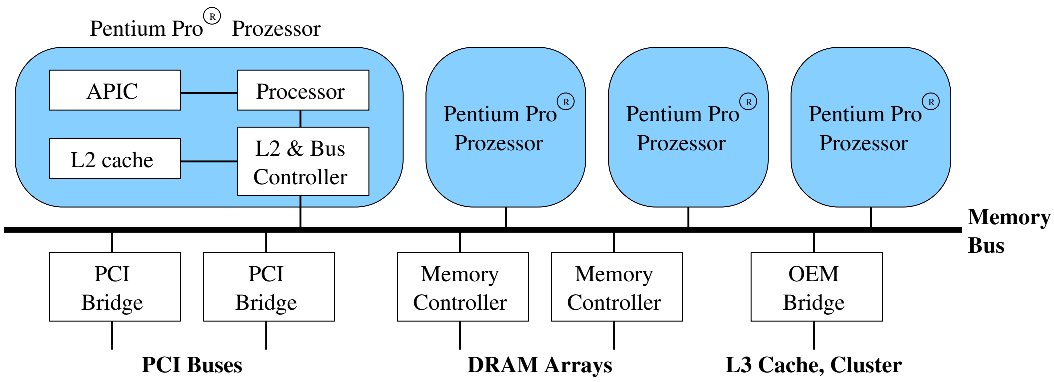
- Verbindungswerk und Speicherzugang kritisch: 8 Prozessoren...64 Prozessoren (Bus/mehrfacher Bus/Crossbar, bei 64 Prozessoren (Sun Enterprise 10000), 4-Prozessor-Blöcke an 16x16 Crossbar)
- Externes (asymmetrisches) Verbindungsproblem
- ein Betriebssystem, eine Auftragsverwaltung
- seit 1963 (Burroughs 825 D)
- Programmiermodell gemeinsame Variable, Kommunikation hoher Effizienz über physisch gemeinsamen Speicher, SPMD und Nachrichtenkopplung gut simulierbar
- Lastverteilung einfach (globale Information bei einem Betriebssystem, kein Lasttransport)
- keine Allokation erforderlich, wohl aber Scheduling
- mittlere bis grobe Granularität
- wenn Bus, dann Cache-Kohärenz über Snooping (z.B. MESI-Protokoll) einfach erreichbar
- nicht für Höchstleistung ! In Top 500 rangiert SUN HPC 10000 mit 256 Prozessoren auf Platz 41 bei 120 Gflop/s Linpack.
- Weitere Beispiele: SGI Power Challenge, DEC/COMPAQ Alpha Server, HP Exemplar, INTEL Standard High Volume Server HVS

Beispiel: INTEL SHV (standard high volume) Server / IBM Netfinity / COMPAQ / DELL... SMPs mit Buskopplung, Snooping und bescheidener Kapazität: 2, 4, 8 (ab Ende 1999) Prozessoren, Anwendungen:

- Mutliprogramming
- Uniprogramming
  - als Server, viele Aktivierungen des dienstleistenden Programms gleichzeitig: Multithreading (ein Adressraum!)
  - als Parallelrechner (selten; eher in den großen SMP Systemen, die aber eher den NUMA zuzurechnen sind).

Heute typisch Pentium III XEON-Prozessoren eingesetzt, Flagschiff der Pentiumlinie (550 MHz)

[Hwang Xu 98]:



(226)

Typische Konfiguration (hier INTEL SHV mit Pentium Pro):

- Prozessor mit APIC (advanced programmable interrupt controller)
- PCI-Bus-Schnittstellen für zwei Busse
- OEM (original equipment manufacturer, Wiederverkäufer)- Brücke für Integration größeres System, z.B. Parallelrechner aus SMP-Knoten
- Memory Bus teilt sich in 6 Subbusse auf : Arbitration, Request, Error Check, Completion, Response, Data, die Pipeline-artig betrieben werden (Durchsatz !)

—

Beispiele von SMPs mittlerer Komplexität [Hwang Xu 98]: (227)

Comparison of Five Commercial SMP Systems

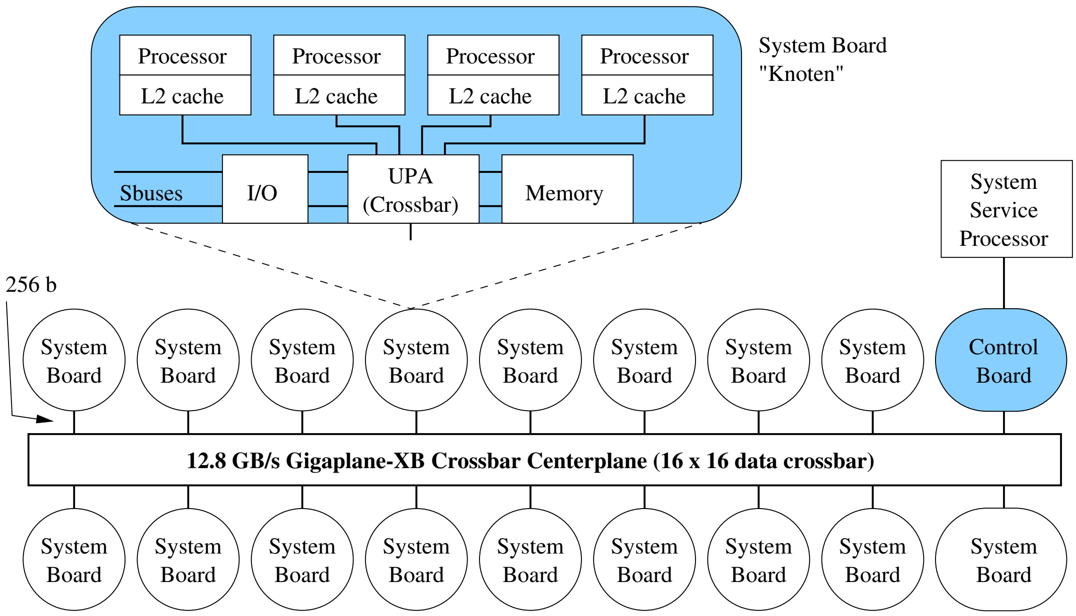
System Characteristics	DEC Alphaserver 8400 5/440	HP 9000/ T600	IBM RS6000/R40	Sun Ultra Enterprise 6000	SGI Power Challenge XL
No.processors	12	12	8	30	36
Processor type	437 MHz Alpha 21164	180 MHz PA 8000	112 MHz PowerPC 604	167 MHz UltraSPARC 1	195 MHz MIPS R10000
Off-chip cache per processor	4 MB	8 MB	1 MB	512 KB	4 MB
Max memory	28 GB	16 GB	2 GB	30 GB	16 GB
Interconnect bandwidth	Bus 2.1 GB/s	Bus 960 MB/s	Bus+Xbar 1.8 GB/s	Bus+Xbar 2.6 GB/s	Bus 1.2 GB/s
Internal disk	192 GB	168 GB	38 GB	63 GB	144 GB
I/O channels	12 PCI buses, each 133 MB/s	N/A	2 MCA, each 160 MB/s	30 Sbus, each 200 MB/s	6 Power Channel-2 HIO, each 320 MB/s
I/O slots	144 PCI slots	112 HP-PB slots	15 MCA	45 Sbus slots	12 HIO slots
I/O bandwidth	1.2 GB/s	1 GB/s	320 MB/s	2.6 GB/s	320 MB/s per HIO slot

Große Konfigurationen kommen mit Bus nicht mehr durch, brauchen Crossbar ("Xbar").

—

Beispiel: SUN Ultra Enterprise 10000 (Starfire) (228)

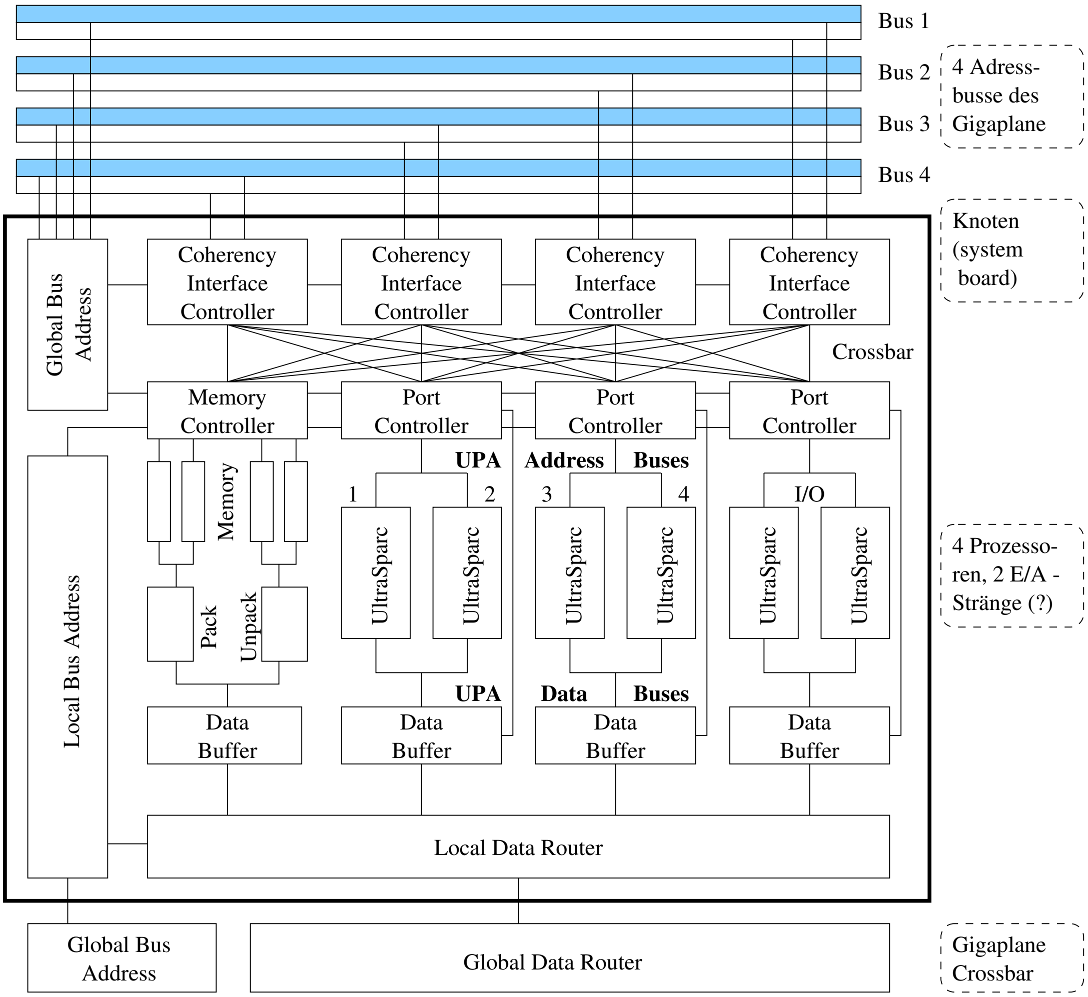
System aus 16-SMP Knoten mit verteiltem Speicher (aber ein Adressraum), bereits mit schwacher NUMA-Charakteristik. Jeder Knoten (System board) enthält 4 250MHz Ultra-Sparc-Prozessoren. Verbindung der 16 Knoten durch das Gigaplane XB (Crossbar) Centerplane (Crossbar und 4 Adressbusse , für Snooping)



(229)

The Sun Ultra Enterprise 10000 architecture. (Courtesy of Sun Microsystems, [600], 1997)

Knoten (system board)



(230)

UPA: Ultra Port Architecture [Hwang Xu 98]

Sun Ultra Enterprise 10000

—

4.3.2 Non-Uniform Memory Access (NUMA, DSM)

Wieder wissen wir schon ziemlich viel, vgl. auch 4.3 (223), über Multiprozessoren: (231)

- ein Adressraum
- Programmierparadigma gemeinsame Variable (aber auch SPMD, Nachrichtenkopplung möglich)
- Probleme Cache-Kohärenz und Ablaufkonsistenz am Speicher und im Gegensatz zu 4.3.1 (224) UMA (Uniform Memory Access)

- Durchsatz und Bedienzeit des Verbindungswerks weniger kritisch; Skalierbarkeitsbereich von wenigen Prozessoren bis 1024; dabei Prozessoren oft zu mehreren im Knoten zusammengefaßt.
- cc-NUMA Cache-Kohärenz wieder durch Hardware gesichert, aber nicht über Bus-Snooping, sondern über Verzeichnisse (directories).
- Internes (symmetrisches) Verbindungsproblem
- Basisbetriebssysteme (Microkernels) auf allen Knoten, machen Verwaltung der lokalen Threads, des Speichers, des Prozessors (der Knotenprozessoren) evtl. der Ein- /Ausgabe.
- wie bei SMP, ist die Kommunikation über den gemeinsamen Speicher organisatorisch einfach; sie ist aber deutlich langsamer als bei SMPs.
- Lastverteilung ist nun mit Lasttransport belastet. Soweit homogener Zugang besteht (z.B. Crossbar, nicht aber Ring!), brauchen Nachbarschaften nicht berücksichtigt zu werden. Dann gibt es für die Allokation nur den Freiheitsgrad: innerhalb eines Mehrprozessorknotens /außerhalb.

4.3.2.1 cc-NUMA: Cache-Coherent NUMA Beispiel: SGI Origin 2000

(vgl. S. 19ff)

Literatur: Laudon, J. , Lenoski, D.: The SGI Origin: A cc-NUMA highly scalable Server

über http://www.sgi.com/tech/origin.html

[Laudon, Lenoski 97] Laudon, J., Lenoski, D. : The SGI Origin: A cc-NUMA highly scalable Server, Proc. International Symposium on Computer Architecture (ISCA), 1997, IEEE Computer Society Press; http://www.sgi.com/Technology/compcon/isca.pdf

- allgemeine Charakteristik
- hierarchische Verbindungsarchitektur
- Prozessoren
- Speicher-Konsistenz und Cache-Kohärenz
- Seitentransporte im verteilten Speicher
- Leistung der Speicherhierarchie

SGI Origin 2000: Allgemeine Charakteristik

Cache-konsistenter NUMA Parallelrechner mit Directories als einzigem Kohärenz-Instrument. Prozessoren Mips R 10k/12k und sehr komplexe vierschichtige Verbindungsstruktur aus Hyperwürfeln und Crossbar, ausbaubar bis 1024 Prozessoren (dann etwa 400/500 GFlop/s Spitzendurchsatz). In der Top 500-Liste vom Juni 99 erscheint eine SGI Origin 2000 mit 2048 Prozessoren (691 GFlop/s Linpack, 1024 GFlops Spitzendurchsatz) auf Platz 6 (!) (Versuchsmaschine, Los Alamos), eine 1024-Prozessor-Maschine auf Platz 27 (265/328 GFlop/s).

Unter den Top 500 sind 98 SGI Origin 2000, häufigste Maschine; installiert 1997...1999

Für 2000 ist eine Weiterentwicklung auf der Grundlage der INTEL-Itanium-Prozessoren angekündigt (→ S. 4.3.1)

SGI-Origin 2000: Hierarchische Verbindungsarchitektur (232)

Stufe 5: 128 ... 1024 CPUs: Hyperwürfel bis Dimensions 5 verbindet bis 32 Hyperwürfel der Dimension 3 (a 32 CPUs)

Stufe 4: 64 CPUs: Hyperwürfel Dimension 4, verbindet 16 Doppelknoten

Stufe 3: 32 CPUs: Hyperwürfel Dimension 3, verbindet 8 Doppelknoten

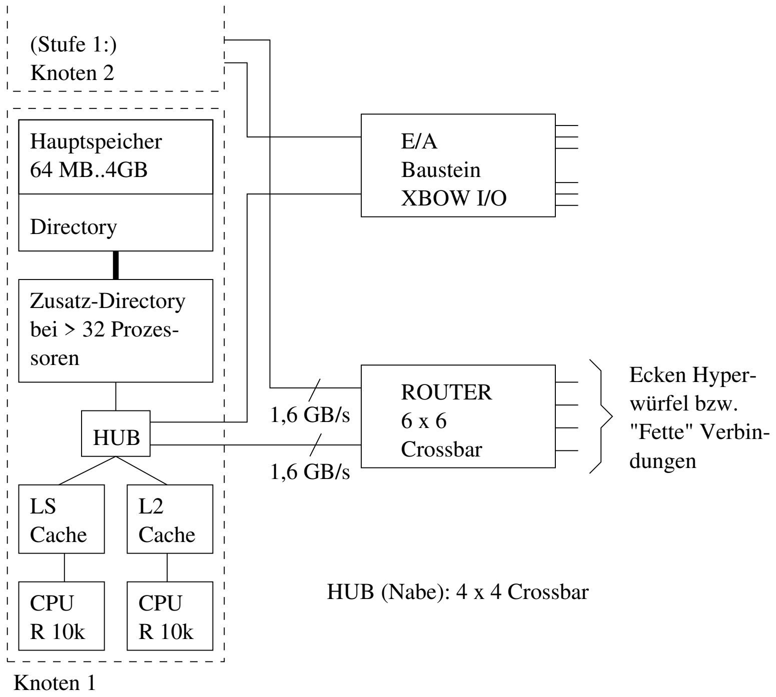
Stufe 2: 4 CPUs: Doppelknoten, verbindet über Router"(6x6 Crossbar) 2 Knoten und 4 Hyperwürfel-Doppelknoten (d.h. 4 Ports)

Stufe 1: 2 CPUs: Knoten, verbindet über Crossbar 2 CPUs und Speicher (64 MB bis 4 GB, mit Directory) und E/A-Baustein.

Freie Router Ports sind für Diagonalen der Hyperwürfel bis Dimension 3 einsetzbar ("fetter Hyperwürfel").

SGI Origin 200: Knoten und Doppelknoten (Stufe 1,2) (233)

(Stufe 2:)

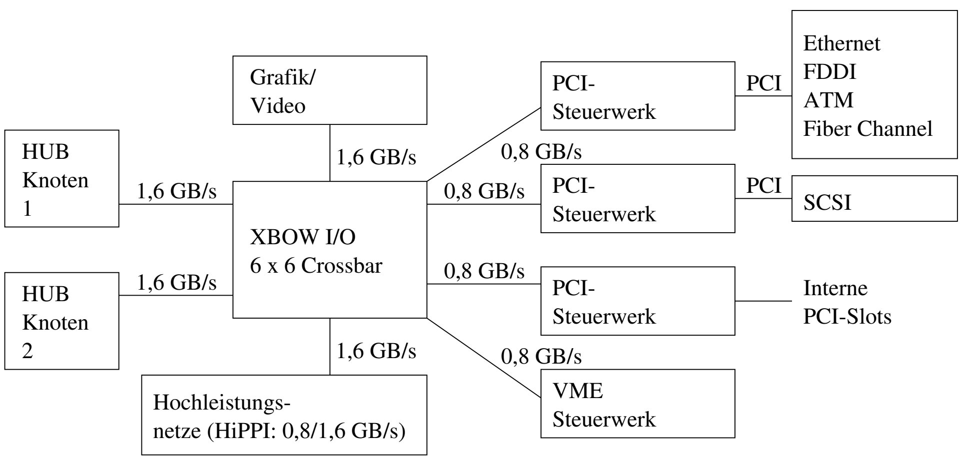


HUB (Nabe): 4x4 Crossbar

SGI Origin 200 hat verteilte (!) Ein/Ausgabe (skalierbar, aufwendig!) Knoten ist *eine* Steckeinheit

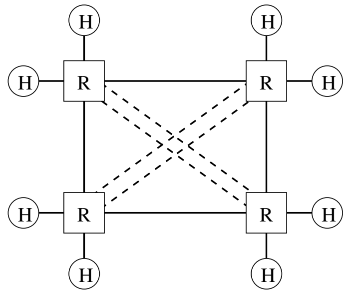
SGI Origin 2000: E/A-Baustein (234)

XBOW I/O (Crossbar 6x6, ähnlich dem Router)



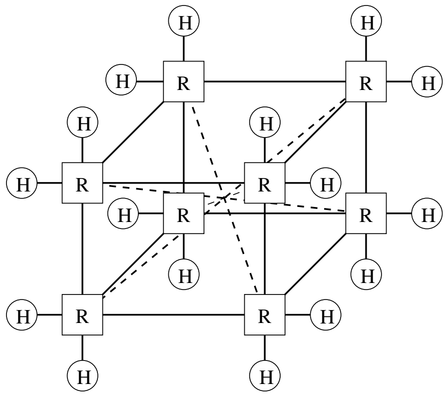
SGI-Origin: Übergang zu *"fetten"* Hypercubes bei 2 und 3 Dimensionen: Einbeziehen von "Xpress Links Options" unter Verwendung des 5./6. Routerports.

Dimension 2 (Stufe 2): 16 CPUs **(235)**



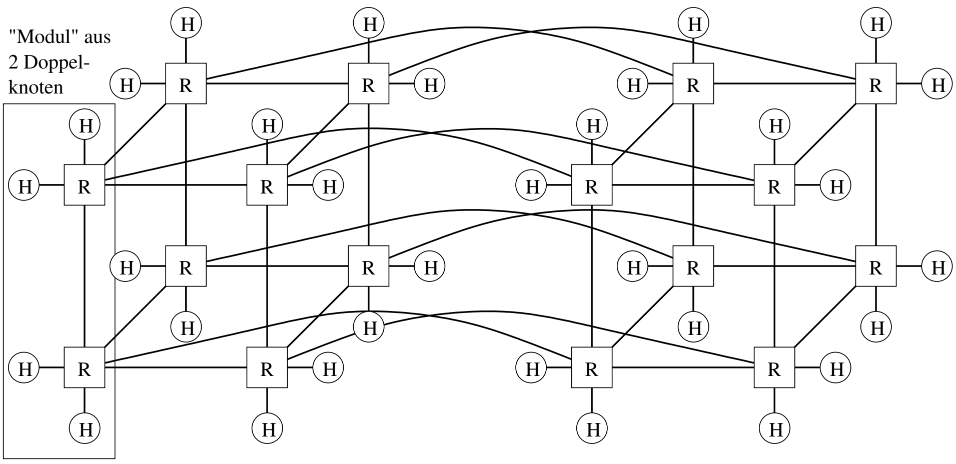
Hier auch doppelte Diagonalen möglich. (H:Hub des Knoten, R: Router)

Dimension 3 (Stufe 3): 32 CPUs **(236)**



Jeder Hub bindet zwei Prozessoren, Speicher und Directory an! (H:Hub des Knoten, R: Router)

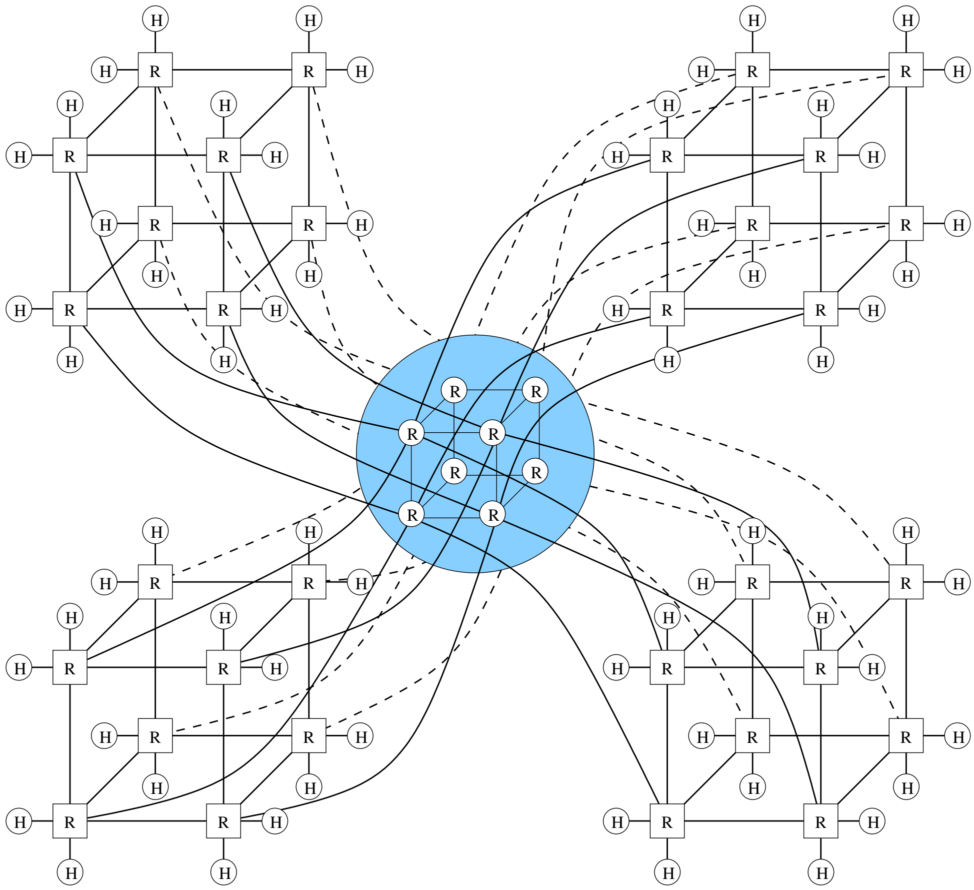
Stufe 4: 64 CPUs in 16 Doppelknoten als 4-dimensionaler Würfel **(237)**



Zwei über Craylink direkt verbundene 32-Processor Systeme (Hyperwürfel). Die Verbindung der Würfel erfolgt unter Benutzung des einen freien Links je Router. (H:Hub des Knoten, R: Router)

Stufe 5: (Beispiel) 128 CPUs **(238)**

Zentraler 3-dimensionaler Würfel ("Cray Router") verbindet (Crossbar) die korrespondierenden Ecken von vier 3-dimensionalen Würfeln: 4x32 CPUs (sind die zentralen Links stark genug (d.h. dreifach)? vermutlich nicht!)



SGI Origin 2000 (Aufbau Technik) **(239)**

Rack

2-128 CPUs
51.2 GFlops
256 GB Memory
81.9 [102.4] GB/s IO
20.5 [26] GB/s Bisection
74 TB Fibre Channel
14 TB Ultra SCSI

Deskside

1-8 CPUs
3.2 GFlops
5.1 [6.4] GB/s IO
2.6 [3.2] GB/s Bisection
18.5 TB Fibre Channel
3.5 TB Ultra SCSI

Tower

1-4 CPUs
1.6 GFlops
4 GB Memory
2.6 [3.2] GB/s IO
1.3 [1.6] GB/s Bisection
20 TB Fibre Channel
4.4 TB Ultra SCSI

SGI Origin 200: Prozessoren

Benutzt werden Mips R10000 / R12000 Prozessoren.

(240)

MIPS R10000	MIPS R12000
200/250 MHz	300 MHz
0,35 μ	0,25 μ
300 mm^2	200 mm^2
5 Pipelines 2 Festpunkt 2 Gleitpunkt 1 Lade/Speicher	5 Pipelines 2 Festpunkt 2 Gleitpunkt 1 Lade/Speicher
64 x 64b Register adressierbar, abgebildet auf 128 x 64b physische Register	64 x 64b Register adressierbar, abgebildet auf 128 x 64b physische Register
spekulative Ausführung bedingter Sprünge bis Tiefe 4	(komplexere Sprungvorhersage)
32kB/32kB nichtblockierende Caches	32kB/32kB nichtblockierende Caches
400 MFlops	500 MFlops

Vgl. Vorlesung Rechnerarchitektur.

SGI Origin 2000: Speicherkonsistenz und Cache-Kohärenz

Die Origin 2000 bietet *sequentielle Konsistenz*, d.h. alle Prozessoren liefern korrekte Zugriffsfolgen, sehen einen (i.a. verschiedenen) Zeitmultiplex dieser Zugriffsfolgen.

Die *Cache-Kohärenz* wird (bis in die 2-Prozessor-Knoten hinunter) ausschließlich durch Directories realisiert., die den Speichern zugeordnet sind. Dort wird für jeden im Speicher vorhandenen *128B-Block* der Kohärenzzustand und ein Eigentümervektor (1 Bit je Prozessor) geführt. Hier ist der *Heimatort* des Blocks gespeichert. Die Kohärenzzustände sind:

- ohne Besitzer
- exclusiv
- gemeinsam
- vergiftet (Seite wird gerade in anderen Speicher verlagert)
- busy/wait (in Umstellung des Zustandes, mehrere wartende Anforderungen)

Cache-Kohärenz

Lese-Fehlzugriff (read-miss), Datenblock.

Lesezugriff auf den eigenen Speicher bzw. Heimatort, Fallunterscheidungen:

- Block *ohne Eigentümer*: Lesen, Zustand exclusiv, Eigentümerbit setzen.
- Block *gemeinsam*: Lesen, Eigentümerbit setzen.
- Block *exclusiv für Anderen*: Leite Leseauftrag an fremden Prozessor weiter; dieser verschickt Kopie des Blocks an den Anforderer, zugleich wird Block in den Speicher des Heimatortes geschrieben (copy back), der Zustand wird gemeinsam.

(invariante Blöcke (Code) werden ohne weiteres kopiert)

Schreibzugriff auf das Cache-Memory: Fallunterscheidung:

- Block ist vorhanden und exclusiv, dann schreibe in das Cache
- Block ist vorhanden aber nicht exclusiv, dann fordere "upgrade" an, um exclusiv Zugriff zu erhalten
- Block ist nicht vorhanden, dann fordere "read exclusiv" an.

upgrade und read exclusiv wenden sich an den *Heimatort* des Blocks:

Fallunterscheidung:

- bisher gemeinsam: mit Hilfe des Eigentümervektors die anderen Kopien des Blocks ungültig setzen. Dann Block des Anforderers exclusiv setzen.
- bisher exclusiv: bisheriger Eigentümer wird veranlaßt, Kopie direkt an Anforderer zu schicken; am Heimatort wird Eigentümervektor umgestellt.

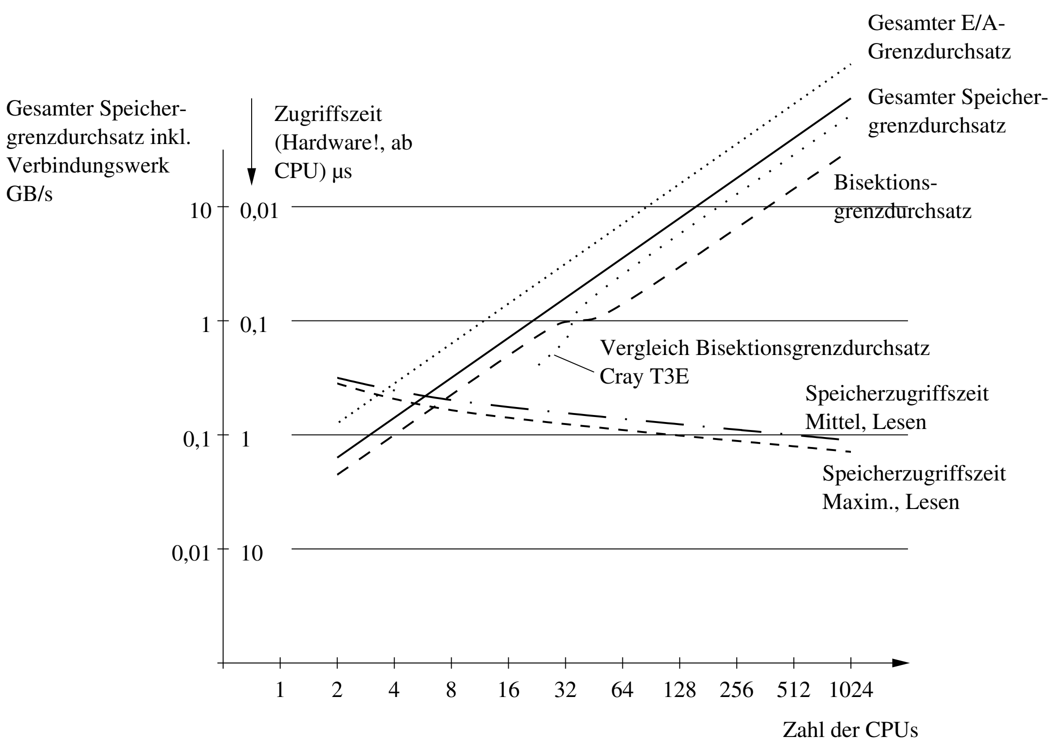
SGI Origin 2000: Seitentransporte im verteilten Speicher

Eine "block transfer engine" transportiert automatisch Seiten *innerhalb* des verteilten Speichers. Aus den Seiten werden zwar 128 B-Kopien an Caches verschickt, die Seiten bleiben aber zusammenhängend am Heimatort.

- Bei *invarianten Seiten* (Code) werden ganze Seiten an andere Prozessoren als Kopien verschickt.
- *Beschreibbare Seiten* werden erforderlichenfalls *verlagert*. Dazu ist die Maschine in *Regionen* (ein Knoten, bei mehr als 64 Knoten: 8 zusammenhängende Knoten) eingeteilt, und die Seite wandert (über ein Schwellwertkriterium, einstellbar !) in die Region der Zugriffsfrequenz.

SGI Origin 2000: Leistung der Speicherhierarchie

(241)



Vergleich SGI/Cray T3E (nccNuma) hat etwas besseren Bisektionsgrenzdurchsatz (homogener dreidimensionaler Torus), etwas schlechtere Zugriffszeit.

Zugriffszeiten der Gesamthierarchie (von Leseanforderung an Cache Memory 1 bis Eintreffen des Wortes):

(242)

Cache Mamory 1		5,1	ns	
Cache Memory 2		56.4	ns	
lokaler Hauptspeicher		310	ns	
entfernter Hauptspeicher	(4 CPUs)	540	ns	Mittelwerte
	(8 CPUs)	707	ns	-“-
	(16 CPUs)	726	ns	-“-
	(32 CPUs)	773	ns	-“-
	(64 CPUs)	867	ns	-“-
	(128 CPUs)	945	ns	-“-

Die guten Werte für 16 und 32 CPUs kommen unter Benutzung der Express-Links (fetter Hypercube) zustande.

4.3.2.2 COMA: Cache-only-Memory-Architecture: Beispiel KSR-1,2.

Ansatz der COMA Maschine (KSR-1)

Jeder Hauptspeicher wird wie ein vollassoziativer Cache betrieben, mit Blöcken von 128B und globaler virtueller Adressierung. Ihm ist ein Directory zugeordnet, das Auskunft über den Inhalt gibt und über virtuelle Adressen abfragbar ist. Die Blöcke werden in einem Demand - Fetching Verfahren herangeschafft:

- Code als Kopien (es gibt dann also mehrere Instanzierungen nach Bedarf)
- Beschreibbare Blöcke als physische Migration (unter Löschung von evtl. im Prozessor - Cache des Senders vorhandener Kopie).

Es gibt also einen beschreibbaren Block nur auf höchstens einem Knoten und daher kein Kohärenz-Problem außer dem zwischen Prozessor-Cache und Hauptspeicher ("Cache")

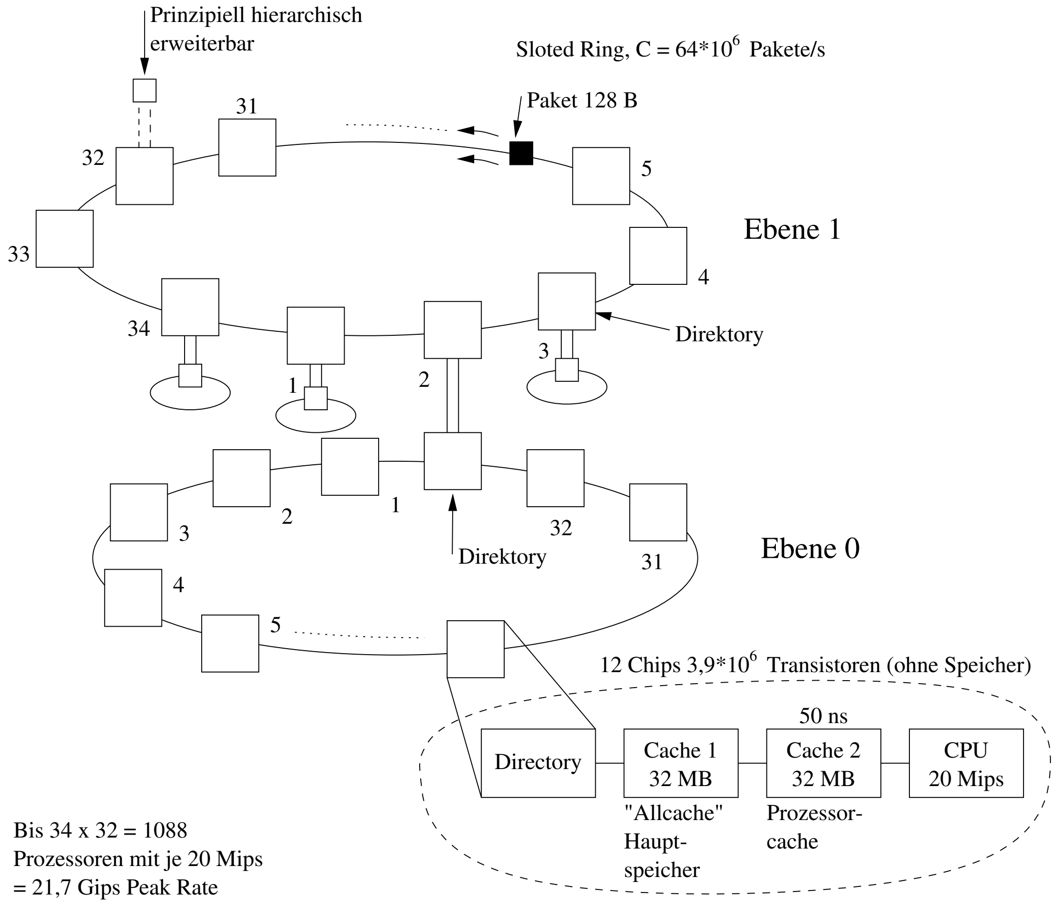
Vorteil des Schemas: Einfache Lösung des Kohärenz-Problems, Daten wandern automatisch zum Zugreifer.

Nachteil des Schemas: Besitzerwechsel bedeutet physischen Transport; unhandlich für beschreibbare Blöcke mit langen nur-lese-Phasen vieler Zugreifer. Aufwendige Seitenersetzung: Daten werden als Seiten vom Hintergrund geladen und verdrängt; Aufwendige(r) Verdrängungs Entscheidung/Vorgang, sobald Blöcke der Seite abgewandert sind.

—

Beispiel eines COMA-Multiprozessors: *Kendall Square Research KSR-1*

(243)



Die Directories enthalten die virtuellen Adressen der im Knoten bzw. im Ring gespeicherten Objekte. Ein nichtlokaler Zugriff (miss) beginnt mit Suche im eigenen Ring, notfalls Umstieg über Ebene 1. Zugriffszeit (minimal):

Prozessor Cache	0,1 μs	
"Cache" (Hauptsp.)	1,0 μs	
lokaler Ring	7,5 μs	kein Kontext-Switch in CPU, sondern Warten
entfernter Ring	28,5 μs	kein Kontext-Switch in CPU, sondern Warten

—

4.3.2.3. nccNUMA: non Cache Coherent NUMA (Beispiel: SGI/Cray T3E)

Die Gewährleistung der Cache-Kohärenz kann auch oberhalb des Rechners, in der Software, geschehen. Hierauf beschränkt sich ein weit verbreiteter Hochleistungs-Parallelrechner, die SGI/Cray T3E.

—

Beispiel: SGI/Cray T3E-1200 (244)

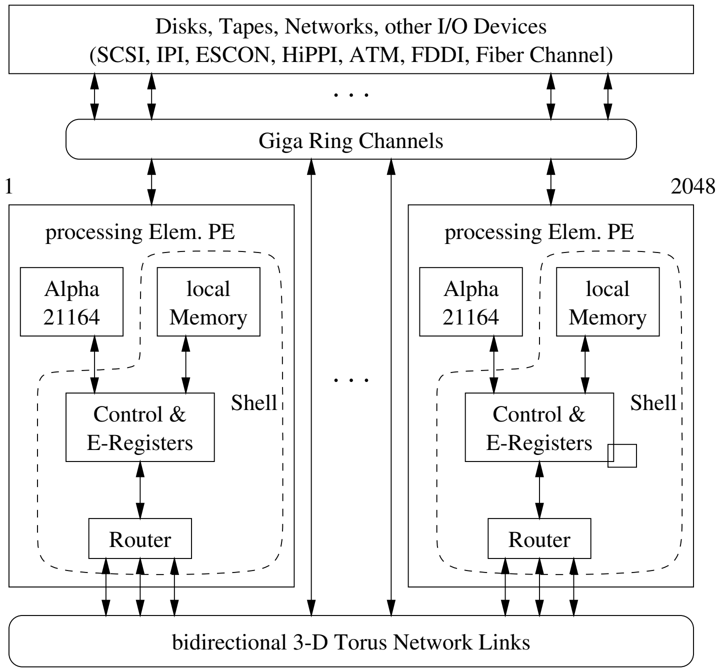
Hochleistungs-Parallelrechner auf

- dreidimensionalem Torus mit besonderem Synchronisationsnetz (Baum → S. 2.3.2), mit

- DEC/COMPAQ Alpha 21164-CPU
- 6 bis 2048 Knoten (bis 2,4 TFlop/s, 4 TB; größte Konfiguration bisher 1084 Knoten; Nr. 3 der Top 500 (Juni 99), erreicht 892 GFlop/s Linpack (1,3 TFlop/s Spitze), mit
- sehr hohem E-/A-Durchsatz (bis 128 GB/s)
- ab etwa 1 Mio DM

T3E-1200: Logische Struktur

(245)

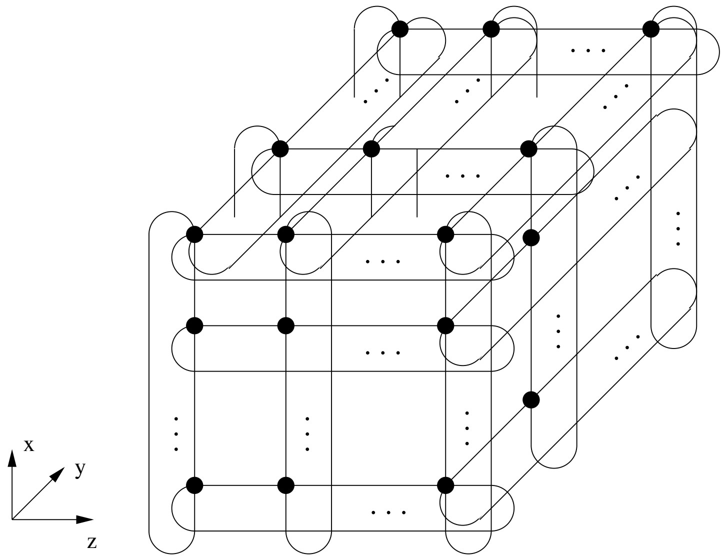


- Alpha Prozessor verpackt in Shell (Speicher, Adressierung, Steuer- und Zwischen-Register für Kommunikation).
- Lokaler Speicher bis 2 GB, 50 ns
- Router mit 6 Ein- / Ausgängen für 3-Dimensionalen Torus und Zugang zu Rechnern extra
- Zugang zu Peripherie für je 4 Rechner über gemeinsamen Gigaring (gegenläufiger Doppelring, 2 x 269 MB/s)

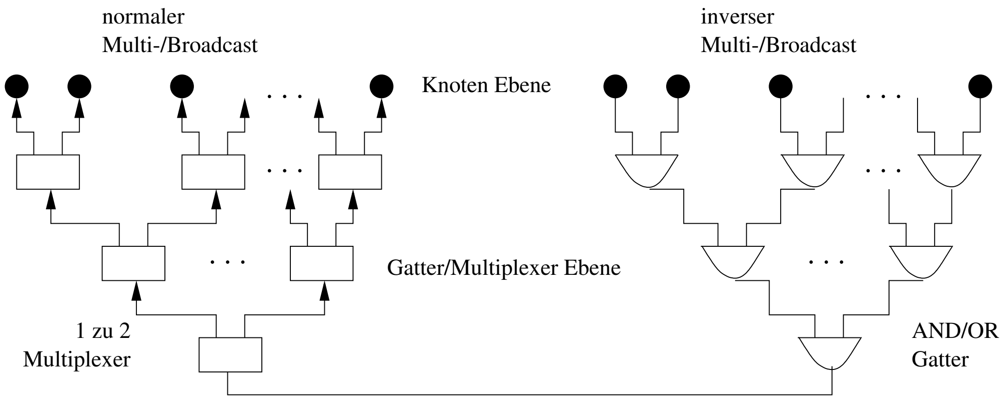
T3E-1200: Verbindungswerk

(246)

vgl. S. 2.3.2



Transportnetzlinks 650 MB/s, bidirektional, 1 64b-Wort alle 12 ns. Größte Erstreckung 16 Knoten. Jeder Router leistet bis 6 x 2 x 650 MB/s. Bisektions-Grenzdurchsatz 122 GB/s.



Broadcast- und Synchronisationsnetz: auf T3D separat, auf T3E integriert in Transportnetz

T3E-1200: Prozessoren, Betriebssystem

(247)

DEC/COMPAQ Alpha 21164 A

bis 4 Befehle je Takt (2 Gleitpunkt)

600 MHz, 1200 Mflop/s, 2400 Mips (Grenzleistung)

Caches	8 kB / 8kB	Stufe 1
	96 kB	Stufe 2

Shell erweitert Adressraum!

Die Konfiguration enthält zusätzliche Prozessoren

- als globale Server des Betriebssystems (Unicos)

- als Redundanz (durch Rerouting inkorporierbar)

Auf den Benutzerprozessoren läuft je ein Mikrokernel (lokale Systemsteuerung, Aufrufe, Unterbrechungen)

4.4 Multirechner (Intel Sandia ASCI Option Red, IBM RS 6000 SP)

Also jetzt (vgl. (198), (223)): (248)

- Im Parallelrechner hat jeder Rechner seinen eigenen Maschinenadressraum (ob mit Software eine Art globaler Speicher erzeugt wird (unhandlich ohne globalen Adressraum!), kümmert uns hier nicht).
- Jeder Rechner hat seinen eigenen Speicher und keinen (Hardware-) Zugang zu fremden Speichern: No remote memory access (NORMA)
- Aber wir haben auch kein Problem mit Ablaufkonsistenz am Speicher, und die Cache-Kohärenz ist kein größeres Problem als auf jedem Uni-Prozessor
- k Rechner haben k Betriebssysteme, ein Mikrokernel/Server-Konzept ist aber ähnlich brauchbar wie auf NUMA-Maschinen, mit globalen Diensten/Benutzungsschnittstelle.
- Kommunikation ausschließlich über Nachrichten. Damit verbietet sich feine/mittlere Granularität.
- Aber, unbeschwert von gemeinsamem Speicherzugang und Kohärenz, lassen sich hohe Kapazitäten (Parallelitätsgrade) erreichen: Hier sind die größten/schnellsten Maschinen zu finden!

Zwei wesentliche Ausprägungen:

konzentrierte Systeme

- massiv parallel (≥ 100 CPUs)
- "Cluster"([Hwang Xu 98]): ≤ 100 CPUs.

verteilte Systeme

- PC/Workstation-Netze
- Metacomputing (unklare Abgrenzung, typisch WAN-gekoppelte Hochleistungsrechner, trotzdem single system image)

—

Beispiel: INTEL/SANDIA ASCI Option Red (249)

1994 startete das Department of Energy ein Projekt zur Entwicklung von Rechnern mit mehr als 1 TFlop/s Durchsatz. Das Projekt läuft 10 Jahre und kostet 1 Mrd. \$. Die Rechner sollen u.a. für die Simulation von Kernwaffentests genutzt werden (also partielle Differentialgleichungen, typisch SIMD/SPMD, hohe Breite erreichbar). Drei Verträge wurden abgeschlossen:

INTEL: 55 M$, Option Red, > 1 TFlop/s
(mit SANDIA Laboratory, Betriebsaufnahme 1997, Weiterentwicklung des Paragon-Schemas, vgl. 1.2)

IBM: 93 M$, Blue Pacific, 3 TFlop/s
(IBM-Parallel-Projekt Deep Blue), IBM lieferte im Oktober 98 eine erste Stufe mit 7200 CPUs (800 MFlop/s, Power 3)

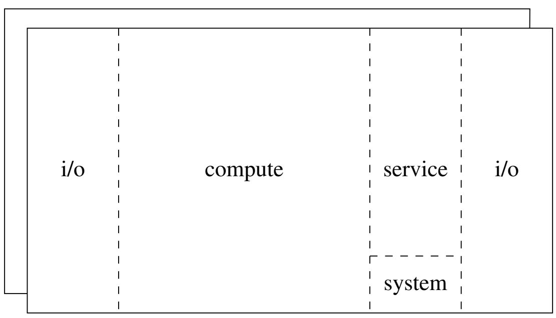
SGI/Cray: 111 M$, Blue Mountain, > 4 TFlop/s
ccNUMA-Doppel-Rechner, Inbetriebnahme 1999 (?), 1 GFlop/s Mips - CPUs.

INTEL ASCI Option Red

Schnellster Computer der Erde nach Top 500 Liste von Juni 99; 1,3 TFlop/s[2] unter Linpack, 1,8 TFlop/s Spitzendurchsatz.

- 4608 Knoten zu je 2 Pentium Pro CPU (200 MHz, 400 MFlop/s), 1999 Übergang zu Xeon CPU (?), aufgeteilt in
  - 4536 Compute Nodes (Produktion)
  - 32 Server Nodes (Programmentwicklung, interaktive Aufgaben)
  - 24 Input/Output Nodes
  - 2 System Nodes (u.a. System-Aufbau)
  - Ersatzknoten
- 594 GB Hauptspeicher, 640 Platten (ca. 1 TB)
- 100 qm Stellfläche, 800 kW (!)

Schemabild des Layouts der Doppelebene des Verbindungswerks, vgl. (186), (187)



—

Beispiel: IBM RS / 6000 SP bzw. SP2 scalable power (250)

IBM RS/6000 SP ist ein Multi-Rechner-System auf der Basis von RS/6000-Workstations, mit eigener Peripherie, gekoppelt über E/A-Kanäle (zum Vergleich: T3E und ASCI Option Red benutzen Speicherbus gekoppelte Rechner).

[2] Basis Pentium Pro; Xeon 2,1 TFlop/s Linpack

IBM RS/6000 SP ist zugleich Basis der Blue Pacific-Maschine. IBM RS/6000 SP ist (größtenteils in kleinen Konfigurationen) mehr als 6000 mal verkauft worden.

Verschiedene Knotentypen, z.B.:

- Power 2 66,7 MHz (<= 267 MFlop/s), 2x32 kB Cache, 3 Varianten (wide, thin, thin 2 node)
- Power 2 sc 0,5 GFlop/s: new wide node
- Power 2 sc als 2...8xMultiprozessor-Knoten: high node
- Power 3: Basis der Blue Pacific; 512 Knoten erreichen 288 GFlop/s Linpack.

Verbindungswerk: Ein- oder mehrstufiges Crossbar, dabei gewisse Nachbarschaften bevorzugt. Laufzeit kleiner Nachrichten (nur Hardware!) 0,5 ... 1 ms. Je Anschluß 40 MB/s Duplex Grenzdurchsatz. TCP/IP oder (effizienter!) IBM User Level Communication Protocol.

Hilfsnetz: Ethernet oder FDDI

Das System läuft unter AIX und ist sehr flexibel partitionierbar.

Wir werfen noch einen Blick auf die Zeitcharakteristiken von Systemdiensten auf einer isolierten AIX-Power2 Workstation und auf p-Knoten Maschine:

Systemdienste unter AIX auf RS/6000 **(251)**

Ausführungszeit von Systemdiensten auf Power 2-CPU unter AIX (*Uniprozessor*!)

Prozesserschaffung	1,4	ms
do. light weight Process	170	μs
do. Thread	40	μs
Kontext (Prozess)wechsel	21	μs
Pipe-Transportzeit	138	μs
Pipe-Durchsatz	34	MB/s
send-receive	40	μs

Ausführungszeit von *Parallelrechnerdiensten* auf 400-Knoten-SP2 (Messungen bis 256 Knoten):

Knoten - Knoten - Kommunikation (MPL mit US-Protokoll), Zeitbedarf bei Nachrichtenlängen m:

$t = 46\mu s + \frac{m}{28{,}6 MB/s}$

Weitere *Parallelrechnerdienste* (z.B. $p = 64$ Knoten, $m = 1kB$; $ld(p) = log_2(p)$)

Broadcast: m Byte an p Knoten

$t = 52\mu s * ld(p) + (0,029 * ld(p)) * \frac{m*\mu s}{B}$

(Beispiel: $t = 52 * 6 + (0,029 * 6) * 1024 = 490\mu s$)

Gather/Scatter: empfangen/senden von verschiedener Nachricht von/an p Knoten

$t/\mu s = 17 * ld(p) + 15 + (0,025 * p - 0,02) * m$

(Beispiel: $t = 17 * 6 + 15 + (0,025 * 64 - 0,02) * 1024 = 1735\mu s$

Total Exchange: jeder sendet m Byte an jeden

$t/\mu s = 80 * ld(p) + 0,03 * p^{1,29} * m$

(Beispiel: $t = 80 * 6 + 0,03 * 64^{1,29} * 1024 = 7000\mu s$)

Barrier Synchronistion (nur Rechenzeit!!)

$t/\mu s = 94 * ld(p) + 10$

(Beispiel: $t = 94 * 6 + 10 = 574\mu s$)

Zum Vergleich: eine Power 2 - CPU macht in $1000\mu s$ (höchstens) 267 kFlops.

Nachbemerkung zu 4.4 Multirechner:

Konsequent müßten sich hier Abschnitte zu Netzen von Arbeitsplatzrechnern/PCs, als Parallelrechner und zu Metacomputern anschließen.

Diese Gebiete haben jedoch nicht zu eigentümlichen Lösungen geführt; sie sind eine Herausforderung an

- Hochleistungsnetze
- Verfahren (Kommunikationsarm, insbesondere robust gegen die unvermeidliche Laufzeit!)
- Betriebssoftware

und werden daher hier (mit diesem Hinweis!) nicht behandelt.

5 Datenflußrechner

(252)

Das Datenflußprinzip ist ein fundamentales, nicht - von Neumannsches, nicht - sequentielles, variablenfreies, funktionales Berechnungsprinzip, Aufbauend auf den *Auftragsystemen* (1.5).

Die technische Bedeutung liegt in

- Schaltnetzen
- Datenflußrechnern (seit etwa 1972 bis 1985, kamen über Laborstadium nicht weg)
- Steuerung der Kollateralität (Parallelität) in Hochleistungsprozessoren
- affinen selbstsynchronisierenden modularen Softwarestrukturen.

Also jetzt:

5.1 Grundidee

5.2 Bedingte Ausführung und Iteration

5.3 Bewertung

5.1 Grundidee des Datenflußschemas

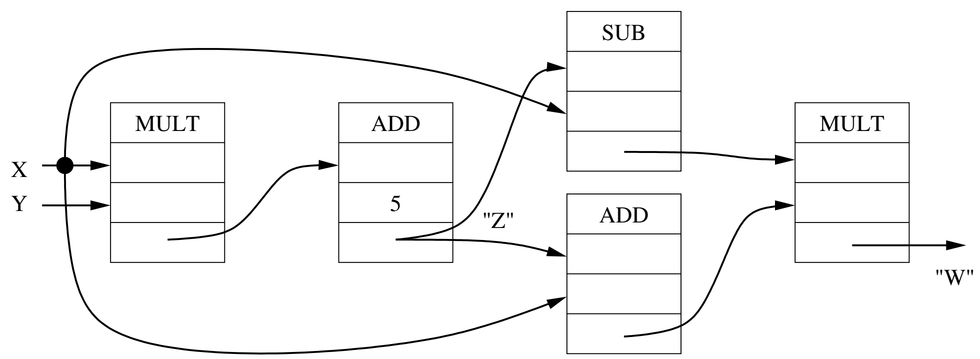
Eine Berechnungsvorschrift wird als maximal nebenläufiger Präzedenzgraph formuliert. (253)

z.B.

Z:=X*Y+5

W:= (Z-X)*(Z+X)

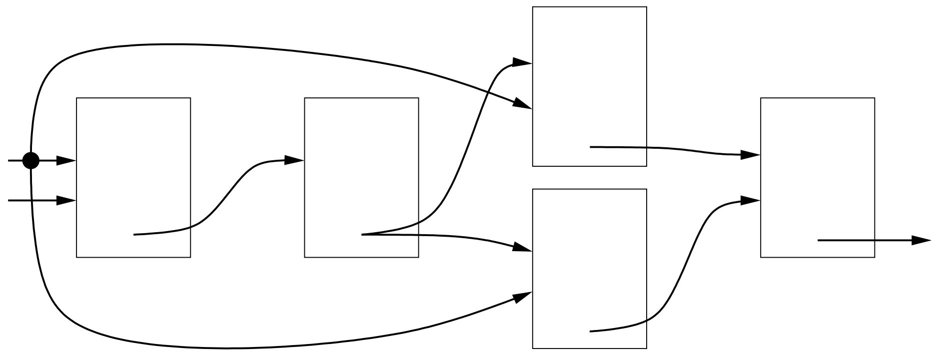
formuliert als



Dabei:

Knoten (activity, packet): Elementare Funktionen des Berechnungsschemas (Teilauftrag).

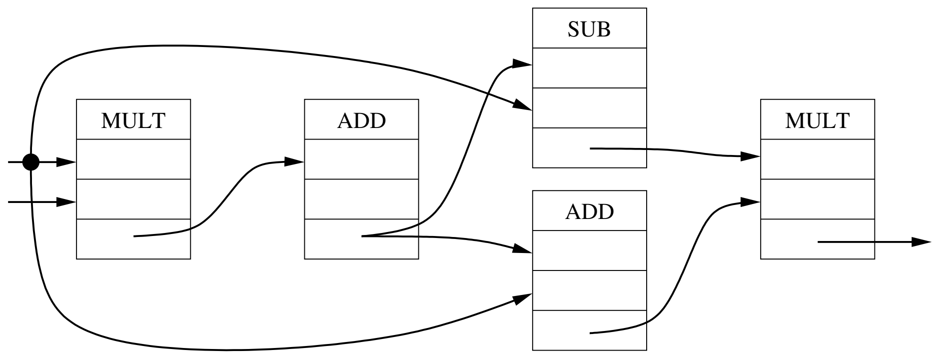
Kanten: Präzedenzen, die durch die Ergebnis/Argument - Beziehung eintreten.



Die Wertübertragung Ergebnis → Argument geschieht durch *einmaliges* Schreiben *einer* Aktivität auf ihren Wertträger, der nachfolgend evtl. vielfach gelesen wird: Wertträger ist *single assignment Variable*. (254)

Die Maschine ist *datengetrieben* (data driven). Sie bearbeitet eine Aktivität, sobald die Argumente vorliegen (und ein Werk frei ist!), d.h. kollaterales Berechnungsverfahren, maximale Nutzung der zur Laufzeit erzielbaren Nebenläufigkeit.

Die Beziehung verläuft *trotz* der Kollateralität *funktional*, da alle Aktivitäten *"störungsfrei"* sind: Werte gehen nur zwischen sequentiellen Aktivitäten über, die Aktivitäten sind seiteneffektfrei. (255)

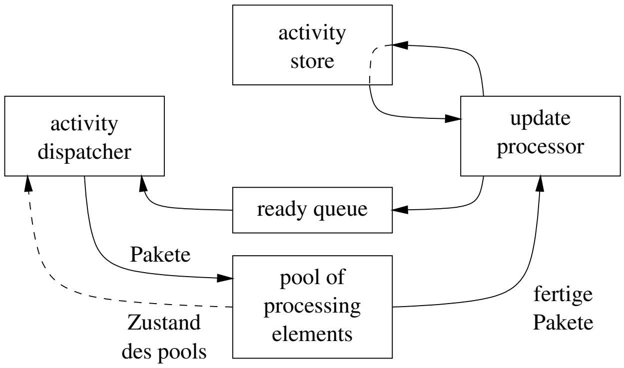


Statische und dynamische Abbildung auf die Maschine **(256)**

Statisch: Der Berechnungsgraph ist der Plan für die Maschine

- Folge: Spezialrechner, lohnend bei Folgeverarbeitung, z.B. in der Signalverarbeitung: digital signal processor DSP.
- Bei homogenem, "breitem" Aufbau, mit Synchronisation, Übergang in "systolisches Array".

Dynamisch: Die Maschine besitzt einen Pool von Verarbeitungswerken, die ausführbare Knoten (Pakete) zugeteilt erhalten.



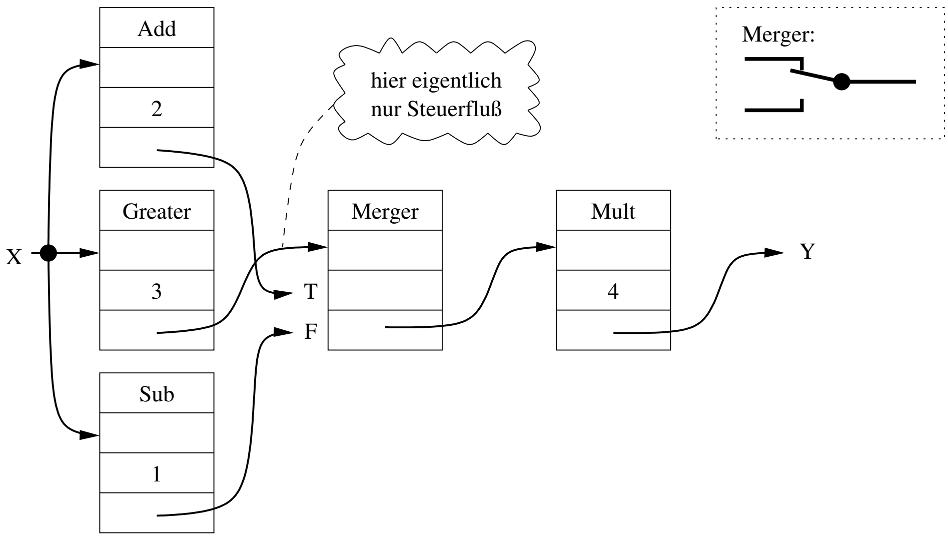
Dennis' Datenflußmaschine 1978

5.2 Bedingte Ausführung und Iteration

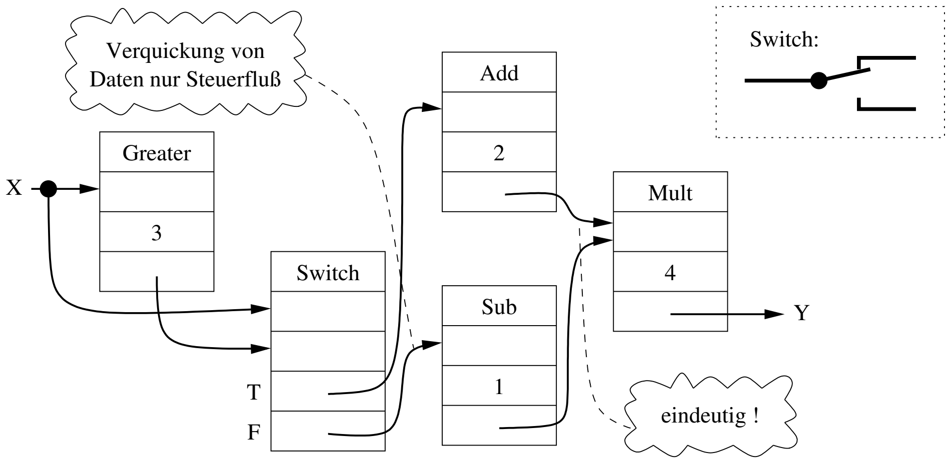
Grundidee wird um bedingte Übergänge und zyklische (!!) Präzedenzen erweitert: (257)

Beispiel für Bedingung:

```
y:= (if x>3 then x+2 else x-1) * 4
```



data driven: unnötige Berechnung von ADD oder SUB



demand driven: unnötige Berechnung vermieden, aber mehr Bedienzeit

Nachbemerkung zu der Antinomie data - driven / demand - driven: (258)

demand-driven: weniger Operationen, mehr Bedienzeit, weniger Parallelität

data-driven: mehr Operationen, mehr Parallelität; die Bedienzeit ist bei unbeschränkten Betriebsmitteln kürzer als bei demand - driven Berechnungen; bei beschränkten Betriebsmitteln kann sie länger sein (Verschwendung von Bedienzeit an irrelevante Berechnungen) wegen Wartezeiten (Grenzdurchsatz)

Man sagt auch lazy evaluation statt demand - driven, eagar evaluation statt data - driven.

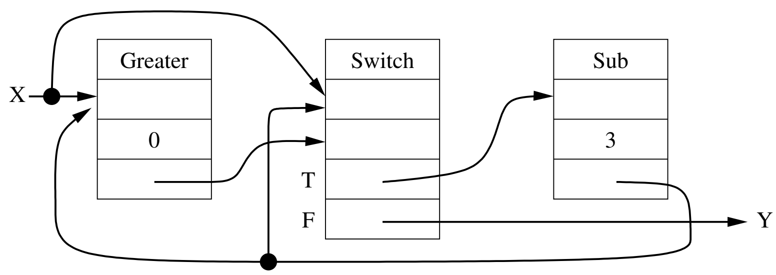
Noch eine Struktur gegen das Grundschema:

Schleifen **(259)**

- Zählschleifen im einfachen Fall bei Compilation abwickelbar
- Induktive Schleifen erlauben erst zur Laufzeit den Datenflußgraphen azyklisch zu formulieren!

Beispiel:

```
y:= while x>0 do x-3
```



(Das Beispiel erlaubt schadlos einen zyklischen Graphen! Die Verletzung der single - assignment - Philosophie bedeutet in Wahrheit ökonomische Verwendung von Speicherzellen! Allerdings ist der von links kommende x-Wert ebenso wie die von unten kommenden nach Aufnahme in "greater" zu tilgen!)

Schleifen der Art

```
for i:= 2 to n do
  begin
    a[i]:= 2 * a[i] + a[i-1];
    b[i]:= c[i] * d[i] - b[i-1]
  end
```

kann man in zwei Schleifen zerlegen, d.h. es ist möglich, kollateral auszuführen.

```
a[2]:= 2 * a[2] - a[1]
b[3]:= c[3] * d[3] - b[2]
```

d.h. Aktivitäten verschiedener Durchläufe können sich überholen.

Lösung: Die Wertübertragung geschieht durch Botschaften, die außer dem Wert und dem Bezeichner einen Generationsindex enthalten (coloured token machine). Die Aktivitäten verknüpfen jeweils einen vollständigen Satz von Werten (tokens) derselben Generation ("Dynamische Datenflußmaschine Arvind 1984). Parallelisiert mehr als ein Compiler kann! (260)

5.3 Bewertung

Das Datenflußschema ist bezüglich des Pools von Werken einfach erweiterbar und fehlertolerant, da es sich dynamisch der Zahl der verfügbaren Werke anpaßt.

Redundante Berechnungen, mit Voting über Argument - Inkarnationen, sind einfach einbringbar. Das "Programm" ist invariant.

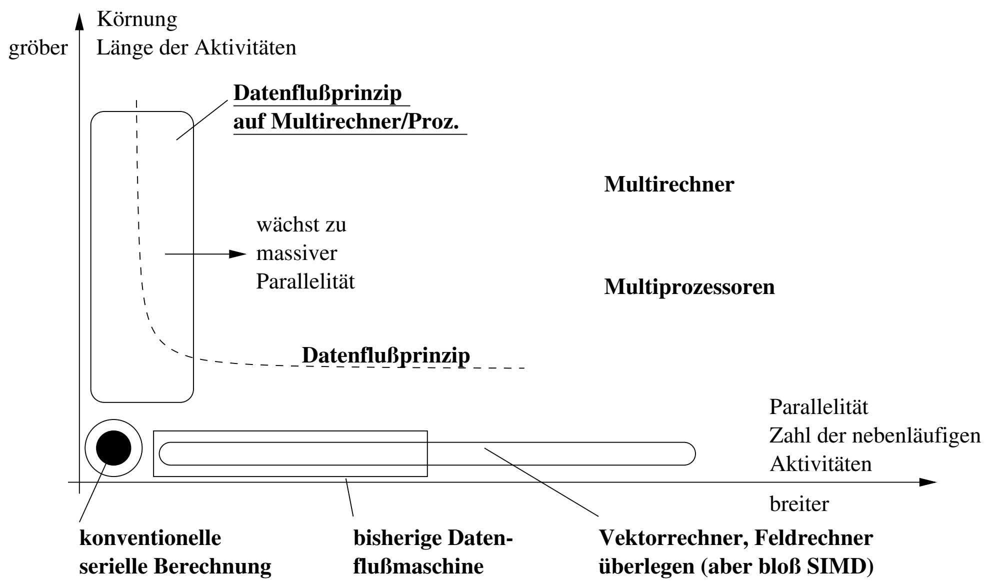
Da Information zur Laufzeit für die Parallelisierung genutzt wird, ist die Nebenläufigkeit ("Breite") potentiell größer als bei Festlegung durch den Programmierer und Compiler.

Kommunikation und Synchronisation: Die Aktivitäten kommunizieren / werden synchronisiert ausschließlich zu Beginn / zu Ende. Der Aufwand für Kommunikation / Synchronisation muß klein sein im Verhältnis zum Berechnungsaufwand der Aktivität. Danach bemißt sich die richtige "Korngröße" (granularity) der Aktivitäten.

Bei kleiner Korngröße (z.B. eine oder wenige Maschinenoperationen), so mit - Datenflußmaschine (Dennis), ist die Effizienz zu klein im Verhältnis zu (compilierten) Registeroperationen eines konventionellen Pipeline - Prozessors. Große Parallelität bei kleiner Korngröße gibt es überwiegend bei Operationen auf regulären Datenstrukturen, was Vektormaschinen sehr effizient bewältigen.

Bei grober Körnung stellt jede Aktivität eine längere Programmausführung dar, die man billiger und universeller auf einem Multirechner oder Multiprozessor, mit Software - Synchronisation / Kommunikation durchführt. Das Datenflußprinzip ist dann ein Programmierschema (wie ein endlicher Automat oder ein neuronales Netz oftmals).

Also



Nachbemerkung vom Standpunkt der (262)

Datenflußmaschine	Hochleistungsmikroprozessor
Berechnungsvorschrift ist compilierter Datenflußplan	Berechnungsvorschrift ist compiliertes sequentielles Programm auf Registervariablen
	Datenflußplan wird zur Laufzeit vom Prozessor gefunden durch • Streichung entbehrlicher Präzedenzen • Privatisierung von Variablen (register renaming)
datengetriebene Ausführung	datengetriebene Ausführung