

→ Vorstellung typischer Konzepte und Architekturformen  
für Parallelrechner mit praktischen Beispielen

- 6.1 Pipelining, superskalare und VLIW-Prozessoren
- 6.2 SIMD-Architekturen
- 6.3 Datenflussrechner
- 6.4 MIMD-Architekturen
  - Simultaneous Multithreading
  - eng gekoppelte Multiprozessorsysteme und Multicore-Prozessoren
  - Multicomputer (lose gekoppelte bzw. nachrichtenbasierte Multiprozessorsysteme)
  - Cluster, Grid
- 6.5 „Mischform“ GPU
- 6.6 Verbindungsnetzwerke von Parallelrechnern
- 6.7 Realisierungsbeispiele

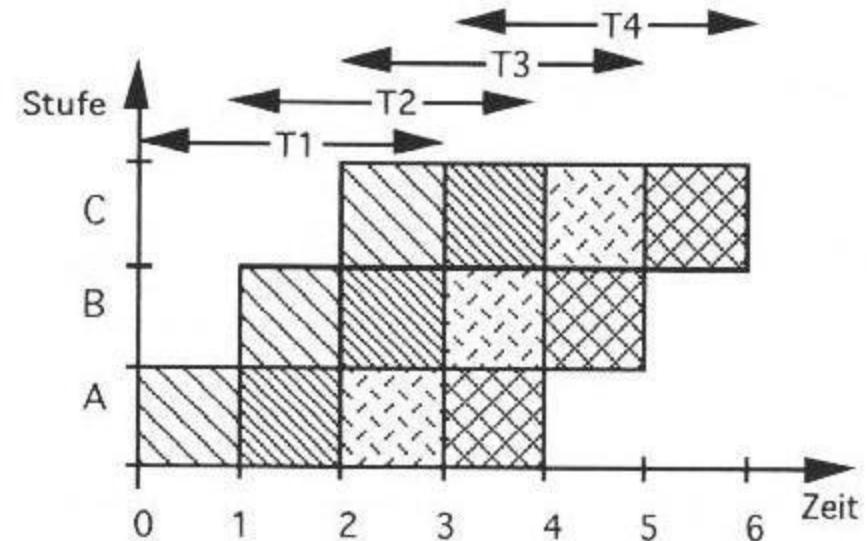
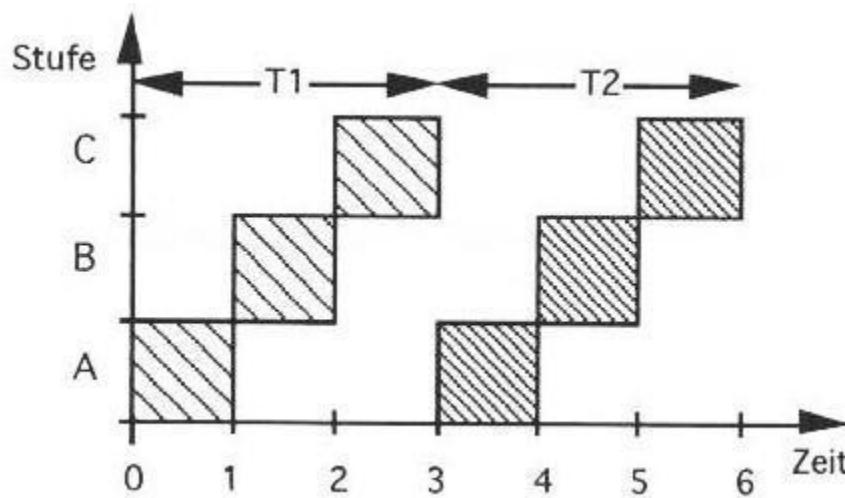
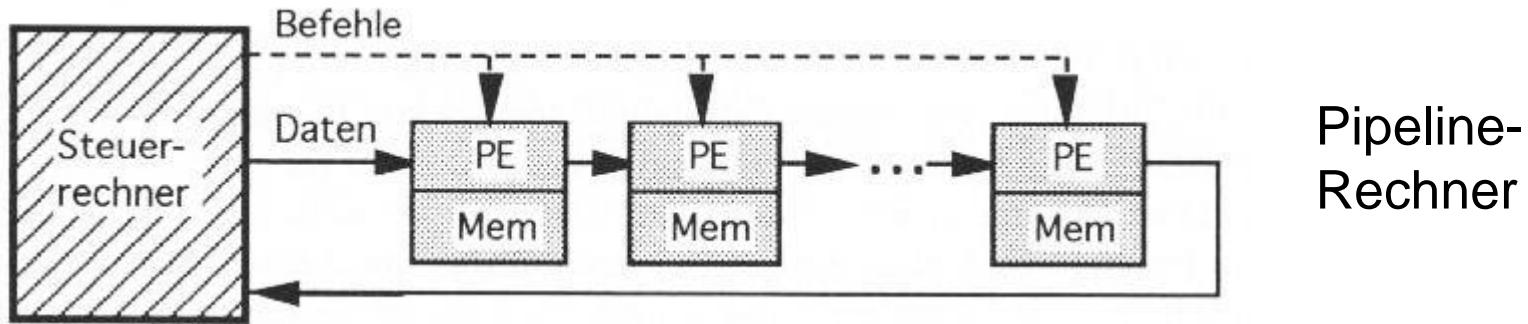
## 6.1 Pipelining, superskalare und VLIW-Prozessoren

### 6.1.1 Pipelining (I)

= Parallelität durch Fließbandverarbeitung

- ist auf verschiedenen Ebenen anwendbar
- typisch vor allem auf Ebene der Befehle bzw. Befehlsphasen
- Verarbeitung einer Instruktion wird in Teilschritte zerlegt, die von zugeordneten HW-Einheiten (= Pipelinestufen) nacheinander ausgeführt werden
- Beispiel 1: Ebene der Befehlsphasen:
  1. Lade nächsten Befehl
  2. Dekodierte Befehl
  3. Operanden holen und Befehl ausführen
  4. Ergebnis schreiben
- Beispiel 2: Ebene der Befehle:
  1. Lade Datenpaar x,y
  2. Multipliziere x und y
  3. Addiere das Produkt zu s

### 6.1.1 Pipelining (II)



Sequentielle und parallele Ausführung in einer 3stufigen Pipeline

### 6.1.1 Pipelining (III)

- Diese Pipelinestufen können (oft) parallel zueinander arbeiten, **außer** bei Abhängigkeiten im Steuerfluss oder zwischen Daten
- Die Verarbeitungsdauer auf jeder Pipelinestufe sollte etwa gleich lang sein (Vermeidung von Wartezeiten)
- eine n-stufige Pipeline hat eine Ladephase von  $(n-1)$  Schritten
- Anzahl der Pipelinestufen bestimmt max. Grad an Parallelität,
- typische Werte von 2 ... 14 (aber nicht beliebig steigerbar)
- Mikroprozessoren mit Pipelining zur Befehlsausführung:  
→ „(skalare) ILP-Prozessoren“ (Instruction Level Parallelism)
- Mikroprozessoren mit sehr vielen Pipelines: „superpipelined“

Pipelines sind ein gängiges Grundprinzip moderner Prozessoren

### 6.1.2 Superskalare und VLIW-Prozessoren (I)

= Parallelität durch mehrere unabhängige Funktionseinheiten FEs,  
wie z.B. ALUs (arithm.-log. unit), FPUs (floating point unit),  
Speicherzugriffseinheiten, Sprung-/Verzweigungseinheiten usw.

- Folge aus sequentiellen Maschinenbefehlen *eines* Programms wird (sofern möglich) mittels Hardware auf die verfügbaren FEs verteilt und dort ausgeführt

Grundprinzip  
eines  
superskalaren  
Prozessors

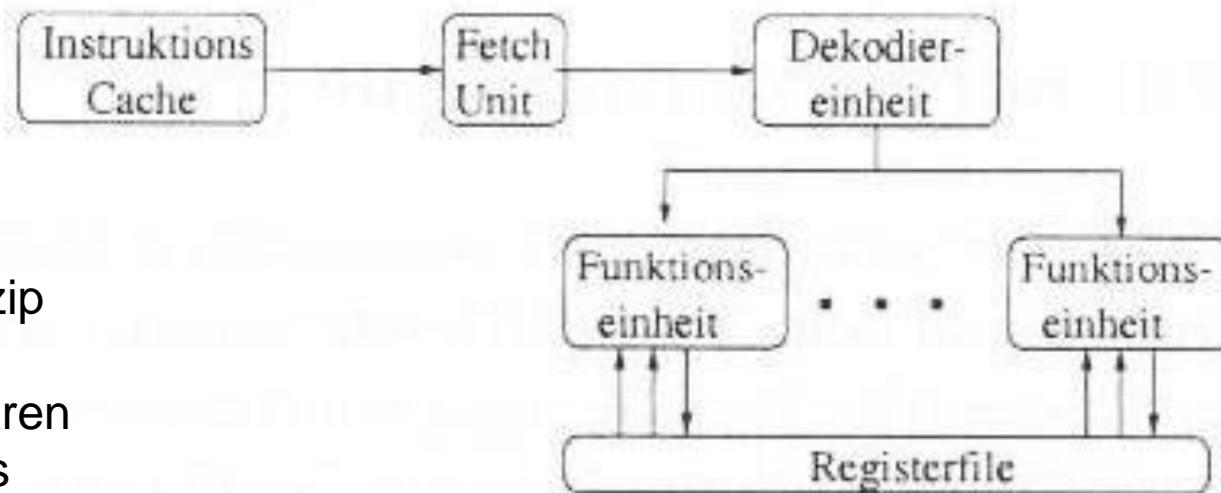


Bild-Quelle: Rauber/Rünger: parallele und verteilte Programmierung. Springer, 2000

### 6.1.2 Superskalare und VLIW-Prozessoren (II)

- Auch hier legen Abhängigkeiten zwischen Befehlen bzw. Daten enge Grenzen bzgl. der Leistungssteigerung (oft nicht mehr als 4 Befehle voneinander unabhängig)
- Hauptaspekt: Art der Zuteilung der Instruktionen an die parallelen FEs (= Scheduling!)
  - statisch:  
Zuteilung wird durch Compiler vorgegeben  
→ **VLIW (Very Long Instruction Word)-Prozessor**
  - dynamisch:  
über die Zuteilung wird erst zur Laufzeit entschieden

### 6.1.2 Superskalare und VLIW-Prozessoren (III)

- **VLIW-Prozessoren:**
  - Spezieller Compiler nötig, der zur Übersetzungszeit prüft, welche Instruktionen parallel ausgeführt werden können. Diese parallelisierbaren Instruktionen werden mit zusätzlichen Informationen versehen (die für jede FE angeben, welcher Befehl zum jeweiligen Zeitpunkt ausgeführt wird) und gruppiert (Gruppengröße entspricht Anzahl der parallelen FEs)
  - Dadurch entstehen sehr lange Befehlsworte bzw. -formate → „Very Long Instruction Word“!
  - Zuordnung Befehl → FE ist also statisch vorgegeben
  - Code für VLIW schwieriger portierbar,
  - gegenwärtig nicht mehr bevorzugt

### 6.1.2 Superskalare und VLIW-Prozessoren (IV)

- („echte“) superskalare Prozessoren:
  - Hier teilt der Prozessor selbst die Befehle den parallelen FEs dynamisch zu (nicht der Compiler)
  - Die fetch- und decode-unit können mehrere Instruktionen zeitgleich bearbeiten (aus dem Cache holen bzw. dekodieren)
  - Nach dem Dekodieren werden die Befehle in einen Puffer („Instruktionsfenster“) gelegt – ohne Beachtung von Abhängigkeiten
  - Eine Zuteilungseinheit (dispatch unit) prüft jeden Befehl bzgl. Ausführbarkeit (Operanden verfügbar?)
  - Es sollten pro Verarbeitungsschritt möglichst viele Befehle zur Ausführung an die FEs weitergegeben werden (ggf. bleiben einige oder sogar alle FEs in einem Schritt leer)
  - Dafür gibt es zwei Varianten:

### 6.1.2 Superskalare und VLIW-Prozessoren (V)

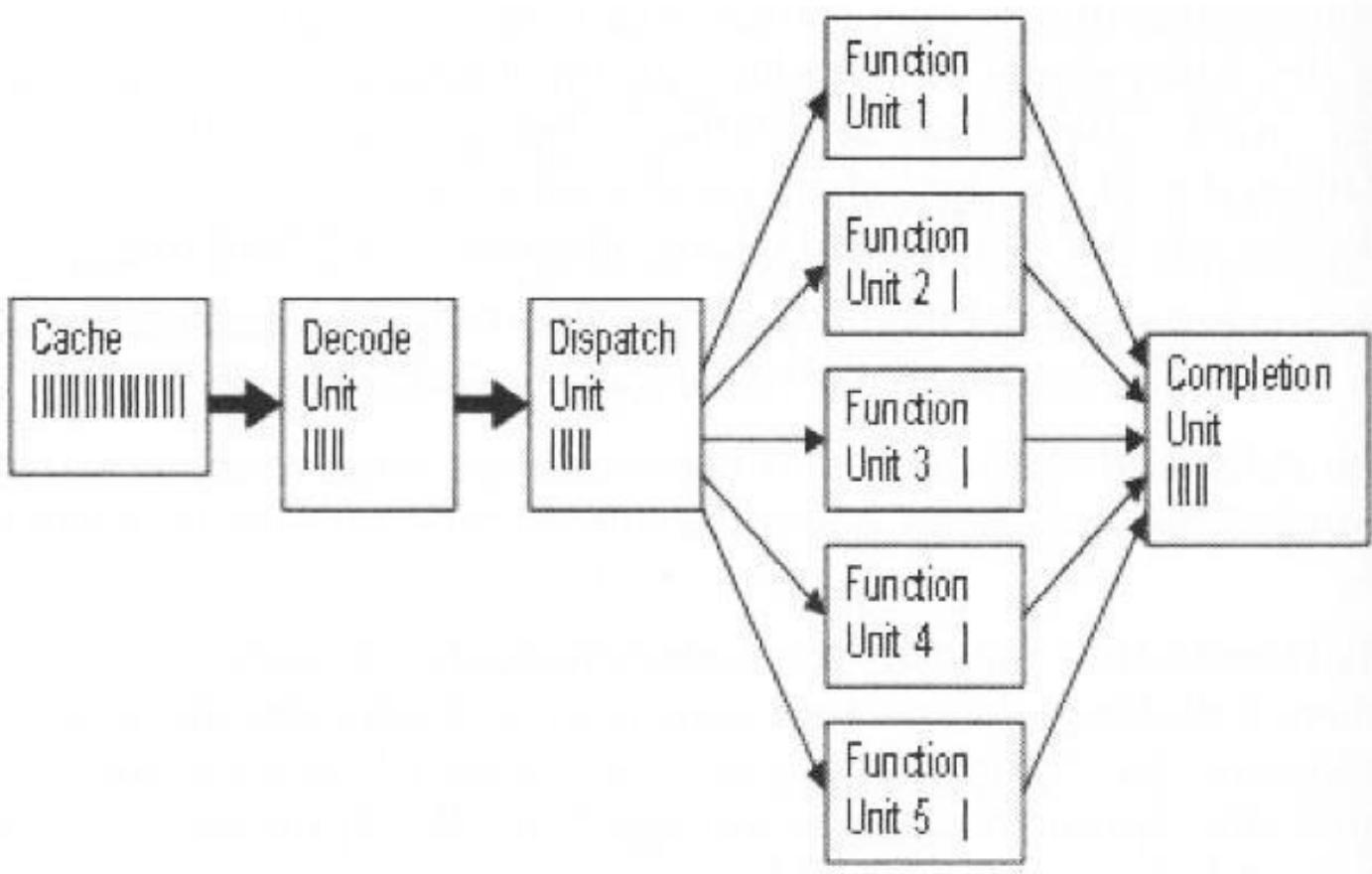


Bild-Quelle: Bengel. u.a.: Masterkurs parallele und verteilte Systeme. Vieweg/Teubner: 2008

### 6.1.2 Superskalare und VLIW-Prozessoren (VI)

1. Befehle werden in der Reihenfolge an die FEs verteilt, in der sie in den Puffer gekommen sind, d.h. ihre logische Reihenfolge bleibt erhalten, nacheinander gestartete Befehle überholen sich nicht (*„in-order-execution“*).  
möglicher Nachteil: ein im Puffer abgelegter Befehl, dessen Operanden noch nicht verfügbar sind, behindert die Ausführung von später eingetroffenen Befehlen, die schon gültige Operanden haben, also schon ausführbar wären!
2. Die Ausführung der Befehle wird auch in einer anderen als der log.-zeitlichen Folge erlaubt (*„out-of-order-execution“*), d.h. es findet dynamisch eine Umordnung statt (reorder), damit der Nachteil von Lösung 1 vermieden wird.

## 6.2 SIMD-Architekturen

### 6.2.1 Überblick (I)

- Es gibt *nur einen Programmspeicher*, auf den (nur) eine Steuereinheit zugreift, also *nur einen Kontrollfluss*
- Zusätzlich existieren n parallele Verarbeitungseinheiten (PEs)

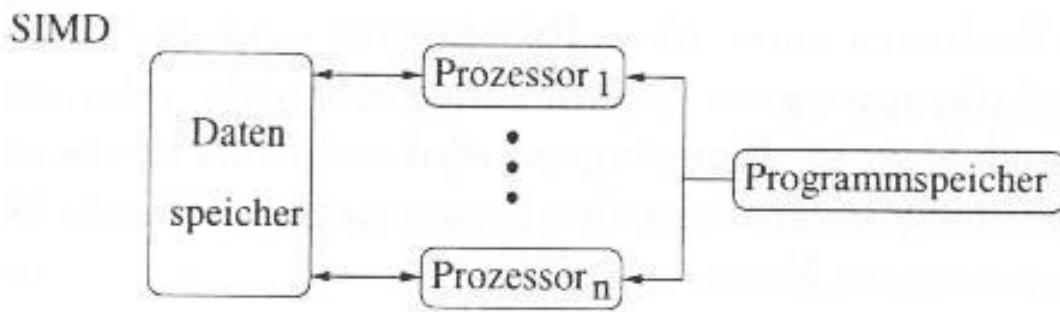
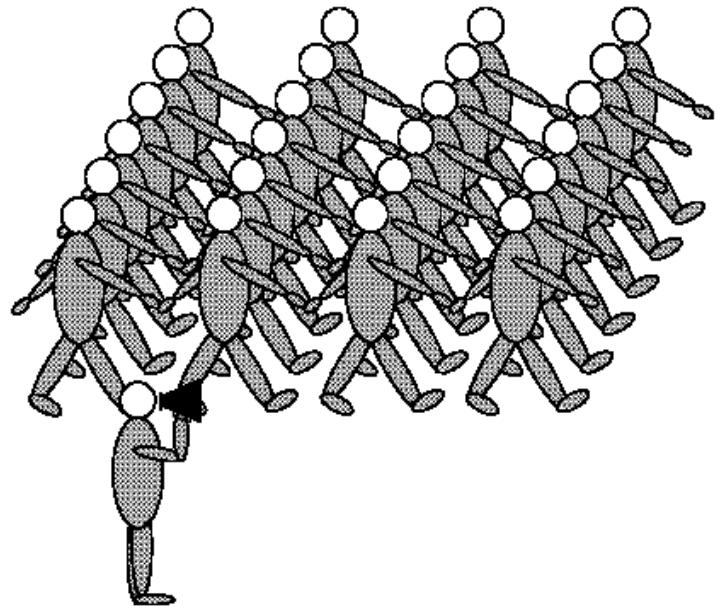
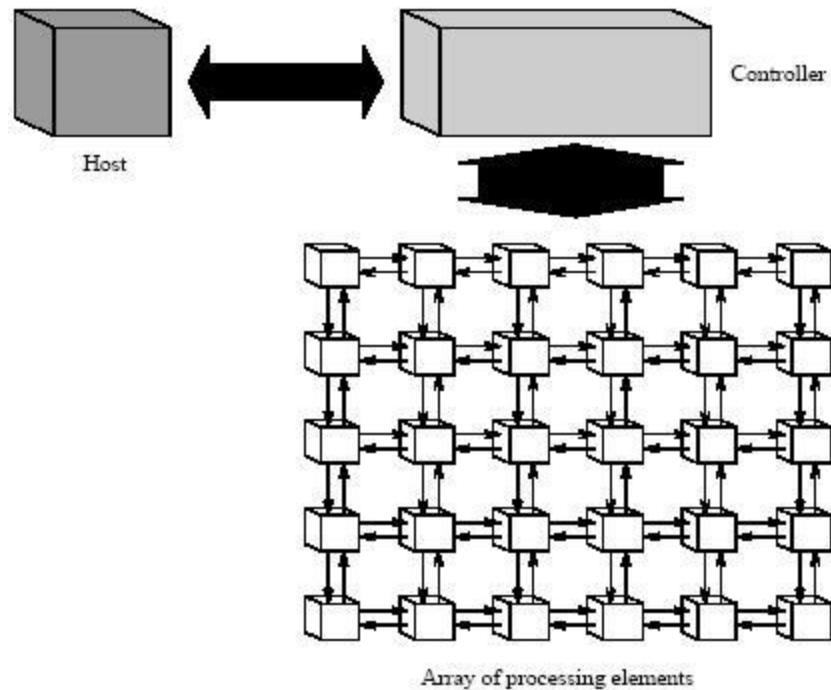


Bild-Quelle:  
Rauber/Rünger:  
parallele und verteilte  
Programmierung.  
Springer, 2000

- Anwendung zur Verarbeitung von geordneten Datenmengen (Vektoren, Matrizen usw.), dabei Ausnutzung der *expliziten Parallelität* von Vektor-/Matrix-Operationen
- v.a. bei wiss.-techn. Berechnungen, sowie Bildern, Videos

### 6.2.1 Überblick (II)



„SIMD in Action“

Quelle: epcc (Edinburgh)

### 6.2.1 Überblick (III)

- 2 Realisierungsformen:
  - Feld- bzw. Arrayrechner (z.B. ILLIAC IV, CM-2, Maspar MP-2)
  - Vektorrechner (z.B. Cray-1)
- Bewertung:
  - einfache Programmierung (nur ein Kontrollfluss!)
  - PEs sind (teure) Spezialprozessoren, daher hat diese Entwicklung nicht von der Mikroprozessorentwicklung profitiert
  - heute als eigenständige Architektur nur noch wenig verwendet
  - aber SIMD-Prinzip ist in vielen aktuellen Mikroprozessoren und vor allem in Spezialprozessoren, z.B. für Grafik, „Streaming SIMD“, integriert („SIMD-Multiprozessoren“),  
Bsp.: GPGPU = general purpose graphics processing unit  
CUDA (HW/SW Architektur von NVIDIA)

### 6.2.2 Feldrechner (Arrayrechner) (I)

- Rechner mit einer zentralen Steuereinheit (ACU, array control unit) und einem **Feld** von Verarbeitungseinheiten (PEs, processing elements)
- Alle PEs führen synchron *dieselbe Operation auf verschiedenen Daten* aus (oder sie sind vorübergehend inaktiv, Maskierung)
- PEs bestehen nur aus ALU, Speicher und Verbindungseinrichtung, können z.T. nur rel. einfache Operationen ausführen

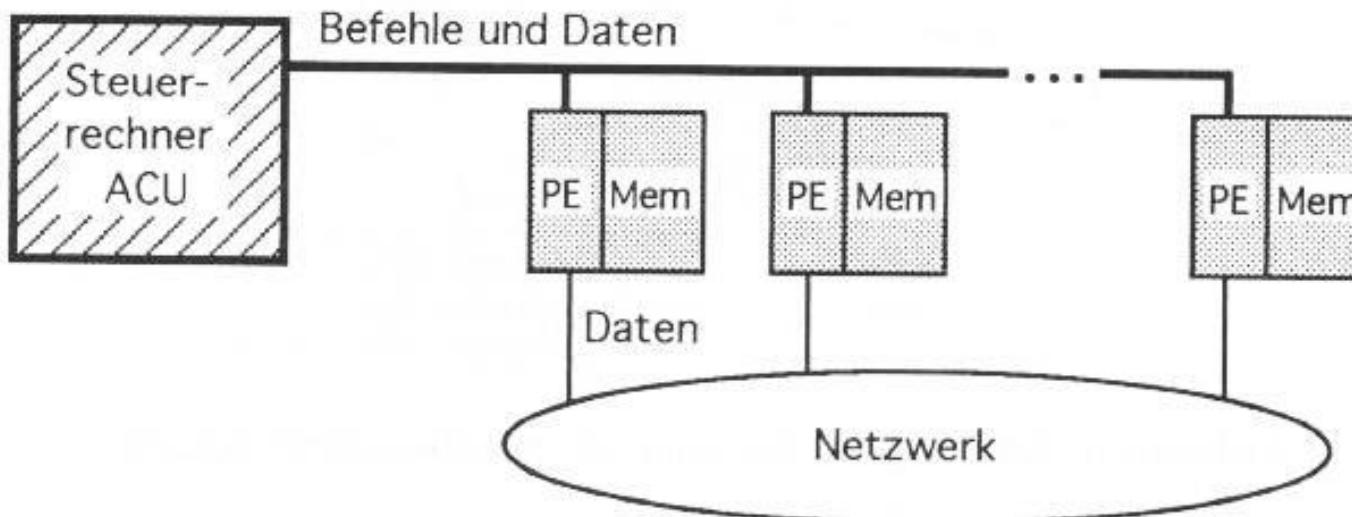


Bild-Quelle:  
Bräunl: Parallele  
Programmierung.  
Vieweg, 1993

## 6.2.2 Feldrechner (Arrayrechner) (II)

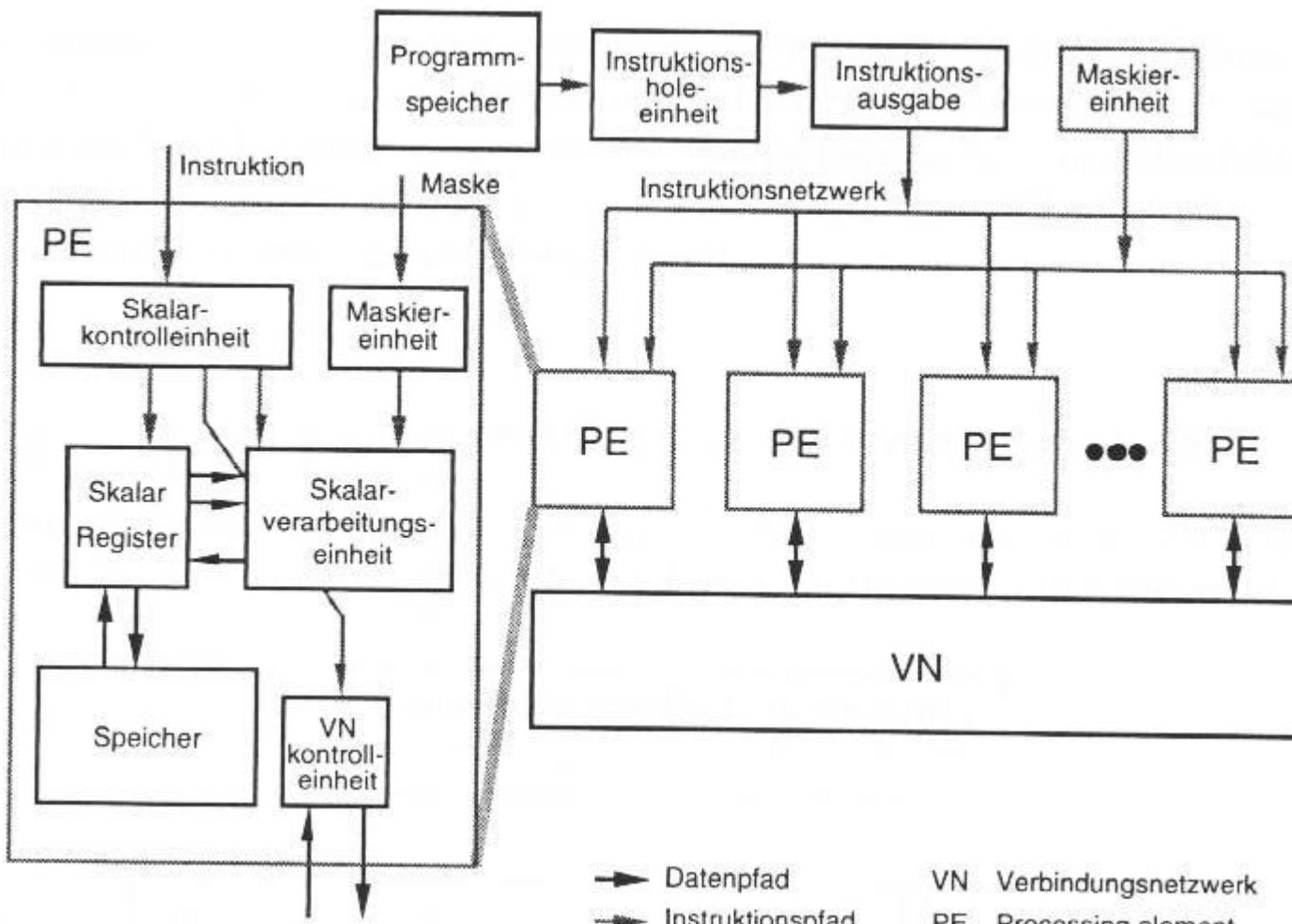


Bild-Quelle:  
Waldschmidt:  
Parallelrechner.  
Teubner, 1995

### 6.2.2 Feldrechner (Arrayrechner) (III)

Sonderform:

#### Systolisches Array

- 2- oder 3-dimensionale Gitteranordnung der PEs
- Verarbeitung der Daten taktsynchron im Pipelineverfahren entlang der Arraydimensionen, daher „wellenförmige Ausbreitung“: es werden ständig von außen neue Daten in das Array eingeschoben, in den PEs verarbeitet, zwischen den PEs weitergegeben und am Ende ausgegeben.

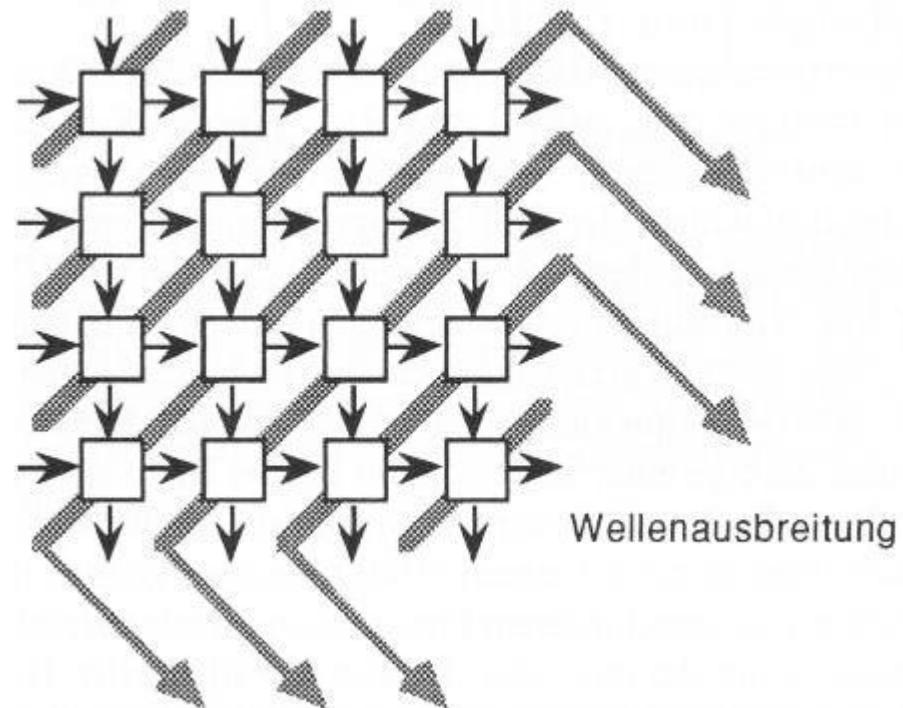


Bild-Quelle:  
Bräunl: Parallele  
Programmierung.  
Vieweg, 1993

### 6.2.2 Feldrechner (Arrayrechner) (IV)

Bsp.: Systolisches Array zur Multiplikation von zwei 3x3 Matrizen

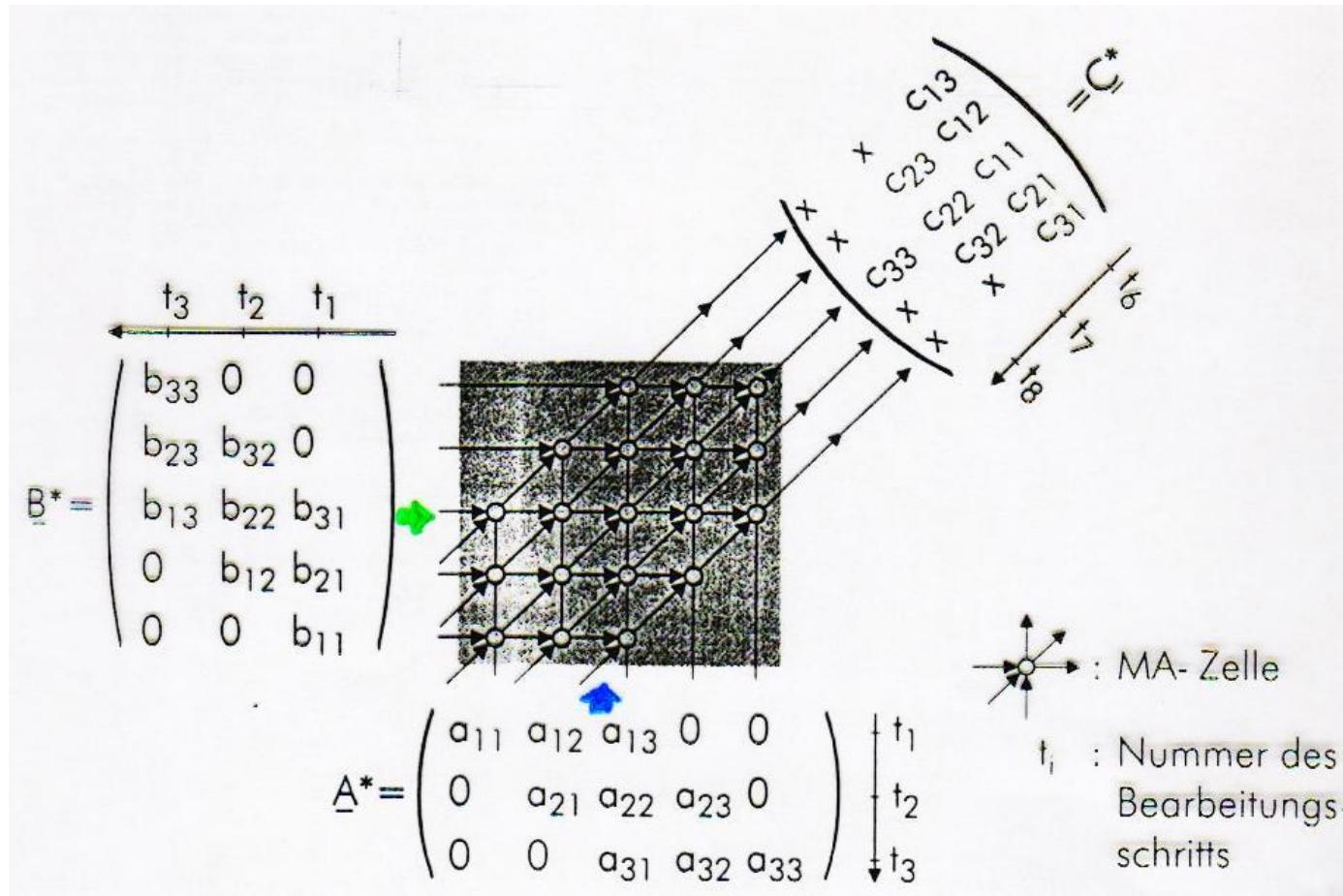


Bild-Quelle: Heidepriem: Echtzeitsysteme, Bd. 1. Oldenbourg-Verlag 2000

### 6.2.3 Vektorrechner (I)

- ähnlich wie Feldrechner, optimiert zur Vektorverarbeitung
- kein globales Verbindungsnetzwerk zwischen den PEs
- PEs enthalten Vektorregister, über denen komponentenweise arithmet./logische Operationen ausgeführt werden, d.h. es entsteht eine pipelineartige Verarbeitung der Vektor-Elemente
- zwischen PEs erfolgt einfacher Datenaustausch über spez. Verbindungsleitungen
- vereinfachter Aufbau:

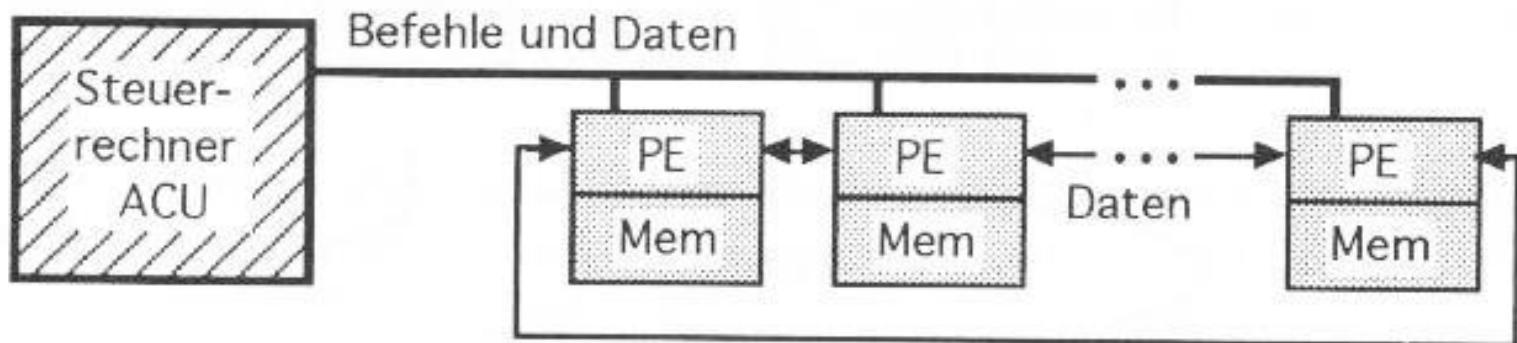


Bild-Quelle: Bräunl: Parallele Programmierung. Vieweg, 1993

### 6.2.3 Vektorrechner (II)

Detailaufbau:

- Vektoreinheit mit mehreren pipelineartig arbeitenden PEs zur Vektorverarbeitung: *ein* Befehl kann *zwei* Vektoren verknüpfen, die Elemente der Vektoren werden in der Pipeline sequentiell aber überlappend verarbeitet; nach Einschwingzeit der Pipeline kann pro Takt ein Ergebnis berechnet werden, d.h. max. Parallelitätsgrad  $\approx$  Anzahl Pipelinestufen
- zusätzlich: Skalareinheit zur Bearbeitung skalarer Befehle („Nicht-Vektor-Befehle“)
- Vektor- und Skalareinheit können parallel arbeiten
- Datenparallelität (in den Vektoren) kann mittels Pipelines gut umgesetzt werden
- einfache sequentielle Programmierung, Compiler kann bei Vektorisierung helfen (z.B. Ersetzen von Operationen in Schleifen durch Vektoroperationen)

### 6.2.3 Vektorrechner (III)

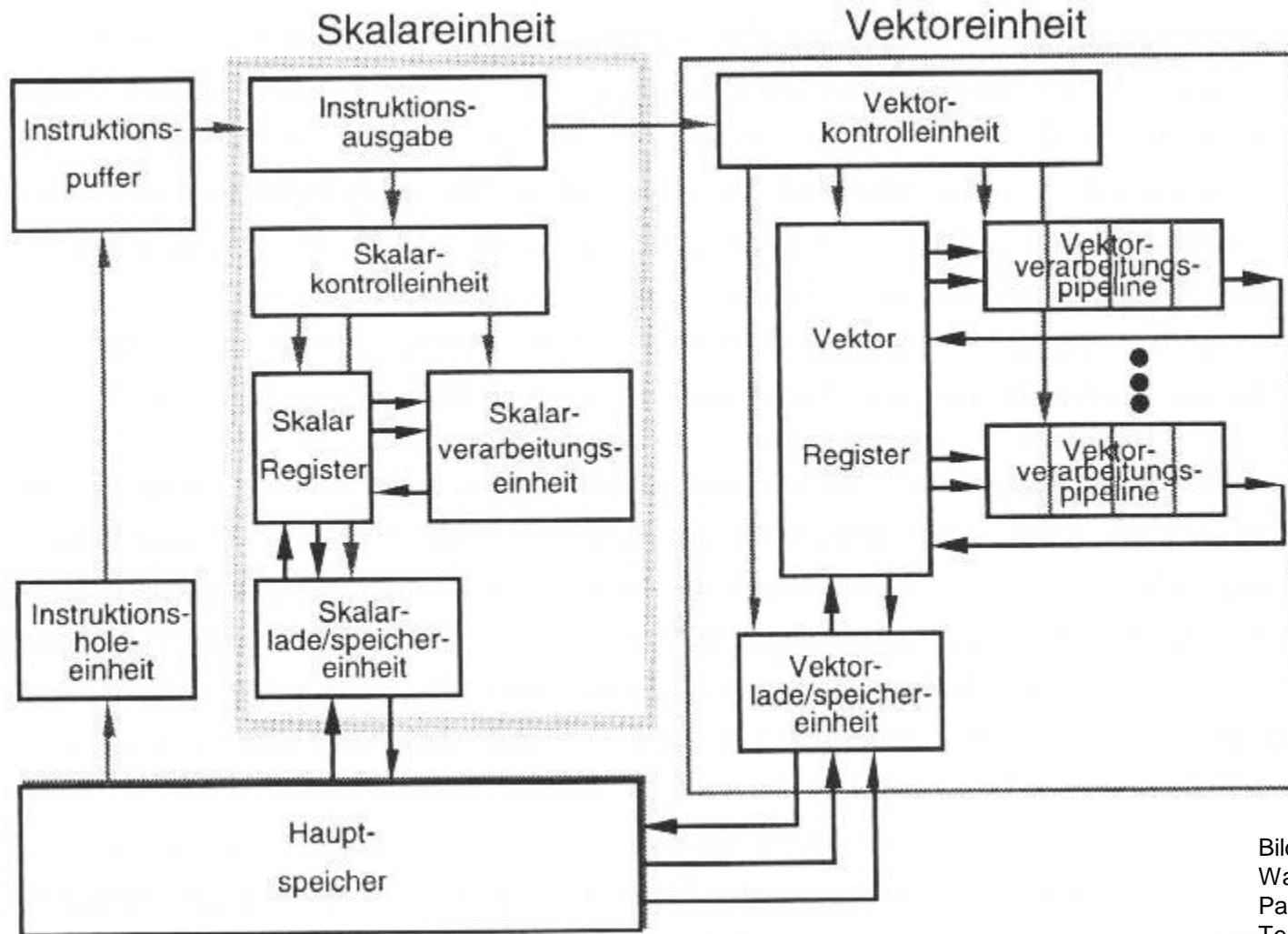


Bild-Quelle:  
Waldschmidt:  
Parallelrechner.  
Teubner, 1995

### 6.2.3 Vektorrechner (IV)

Beispiel:

- „für alle i von 1 bis n:  $C[i] = A[i] + B[i]$ “
- im Vektorrechner: **ein** Vektorbefehl addiert die Vektoren A und B elementweise und legt das Ergebnis in C ab:  
 $C(1:n) = A(1:n) + B(1:n)$
- Falls eine Addition 4 Takte benötigt (= 4 Pipelinestufen), entsteht folgender Ablauf:

	1	2	3	4	5	6
Stufe 1	A1, B1	A2, B2	A3, B3	A4, B4	A5, B5	A6, B6
Stufe 2		A1, B1	A2, B2	A3, B3	A4, B4	A5, B5
Stufe 3			A1, B1	A2, B2	A3, B3	A4, B4
Stufe 4				A1, B1	A2, B2	A3, B3



Zeit

### 6.3 Datenflussrechner (I)

- basieren *nicht* auf dem Von-Neumann-Prinzip!
- sie haben keinen Programmzähler und keinen zentralen Speicher!
- Abarbeitung eines Programms wird durch *Datenfluss* bestimmt:  
→ ein Befehl kann (erst) dann ausgeführt werden, wenn alle seine Operanden verfügbar sind,  
Ergebnisse eines Befehls, die wiederum Eingabedaten eines anderen Befehls sind, werden nicht gespeichert, sondern gleich weitergegeben
- Programmierung mittels spez. Datenfluss-Sprachen,  
z.B. LAU, VAL, Id, Lucid, Lustre, SISAL
- spezieller Compiler erkennt Datenabhängigkeiten und generiert Binärkode, in dem diese berücksichtigt sind  
dabei wird jede Datenabhängigkeit eindeutig markiert  
dadurch können verschiedene Codesegmente später parallel ausgeführt werden, z.B. durch mehrere Threads

### 6.3 Datenflussrechner (II)

#### Ringstruktur eines Datenflussrechners

- nach jeder Aktualisierung des Speichers erfolgt (aufwändige) Überprüfung bzgl. nun ausführbarer Befehle
- kann durch (begrenzten) Assoziativspeicher beschleunigt werden

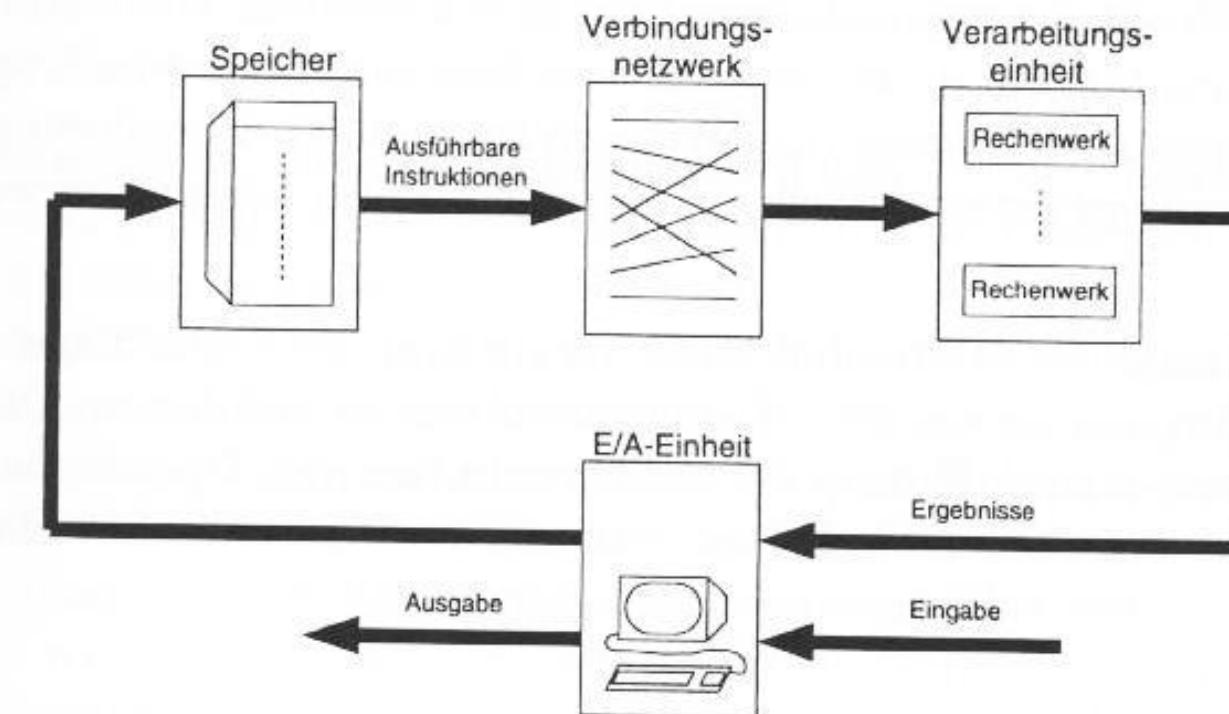
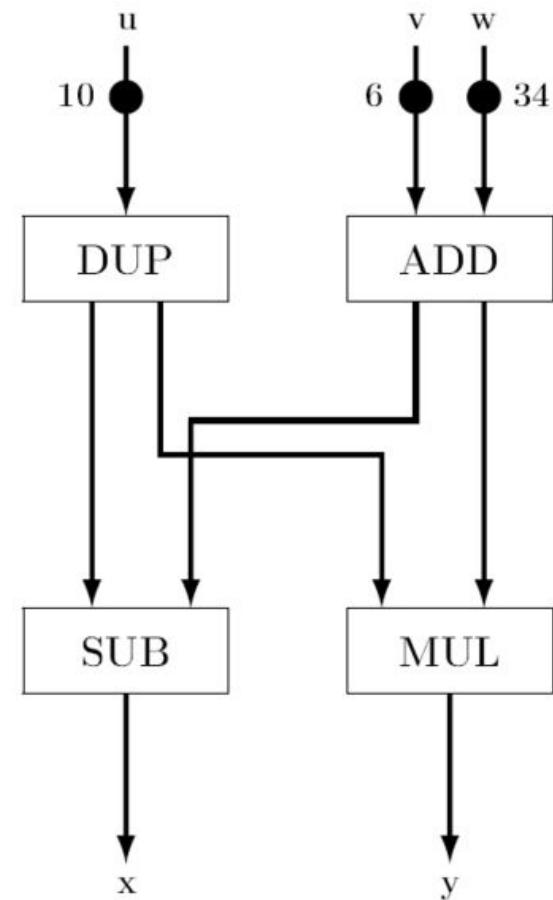


Bild-Quelle:  
Waldschmidt:  
Parallelrechner.  
Teubner, 1995

### 6.3 Datenflussrechner (III)

- Datenfluss kann durch *Datenflussgraph* modelliert werden
  - gerichteter Graph
  - Knoten = Befehle
  - Kanten = Datenabhängigkeit
  - bewertete Token auf den Kanten  
= Eingabedaten für Befehle
  - ein Knoten „feuert“, wenn genug Token auf den Eingangskanten sind
  - es können mehrere Knoten parallel feuern
- Auswertung eines Datenflussgraphen entspricht einer math. Funktion
- Beispiel:  
Input:  $u, v, w$ ;  
 $x = u - (v + w);$   
 $y = u * (v + w);$   
Output:  $x, y$ ;



Quelle für das Beispiel und das Bild: Wikipedia

### 6.3 Datenflussrechner (IV)

Programmiersprachen für Datenflussrechner

- benötigen keine Konstrukte für Parallelität, weil sie diese implizit enthalten!
- Compiler muss Datenabhängigkeiten erkennen können
- daher keine Mehrfachbenutzung von temporären Variablen (= Prinzip der Einmalzuweisung), um unnötige scheinbare Datenabhängigkeiten zu vermeiden (d.h. ein Befehl kann nicht ausgeführt werden, weil die Speicherstelle für das Ergebnis noch mit einem Wert belegt ist, der später noch gebraucht wird); dies erhöht aber den Speicherbedarf insgesamt
- Seiteneffektfreiheit durch Verbot globaler Variablen

Gesamtbewertung:

- als eigenständige Architektur in der Praxis nicht durchgesetzt,
- aber maßgebliche Beeinflussung der Entwicklung moderner µP

## 6.3 Datenflussrechner (V)

zwei Architekturformen:

- statische Datenflussrechner
  - ein Knoten pro Kante
  - einfache Ermittlung von Knoten, die feuern können
  - eingeschränkte Parallelität bzgl. Schleifeniterationen
  - geringe Programmiersprachen-Unterstützung
  - nur wenige Maschinen, z.B. MIT Static Dataflow Archit, DDM1, DDP
- dynamische Datenflussrechner
  - wenn der selbe Teilgraph von mehreren Token gleichzeitig durchlaufen werden könnte (z.B. für verschiedene Durchläufe der selben Schleife), wäre höhere Parallelität möglich,
  - dazu müssten aber die Token markiert werden (mittels Tags) und ein Knoten feuert nur, wenn genug Token mit demselben Tag vorliegen; dies erfordert eine (aufwändige) Prüfung der Token mittels „Token matching“
  - Beispiele: MIT Tagged Token Dataflow Archit., Monsoon, EM-4

## 6.4 MIMD-Architekturen

### 6.4.1. Überblick (I)

MIMD = Multiple Instruction, Multiple Data

- „Volle“ Parallelität
- gegenwärtig am häufigsten benutzte Architektur
- mit vielen verschiedenen Ausprägungen, z.B.
  - Multithreaded Architecture (MTA) bzw. Simultaneous Multithreading (SMT)
  - (eng gekoppelte) Multiprozessorsysteme und Multicore-CPUs
  - (lose gekoppelte) Multiprozessorsysteme (= Multicomputer)
  - Cluster, Grid
- Hauptmerkmal: auch Befehlsstrom ist verteilt, d.h. es gibt keine zentrale Steuerinstanz!  
Verarbeitungseinheiten besitzen lokale Autonomie (eig. Bef.-zähler)

### 6.4.1. Überblick (II)

Merkmale bzw. Klassifizierungskriterien für MIMD-Systeme:

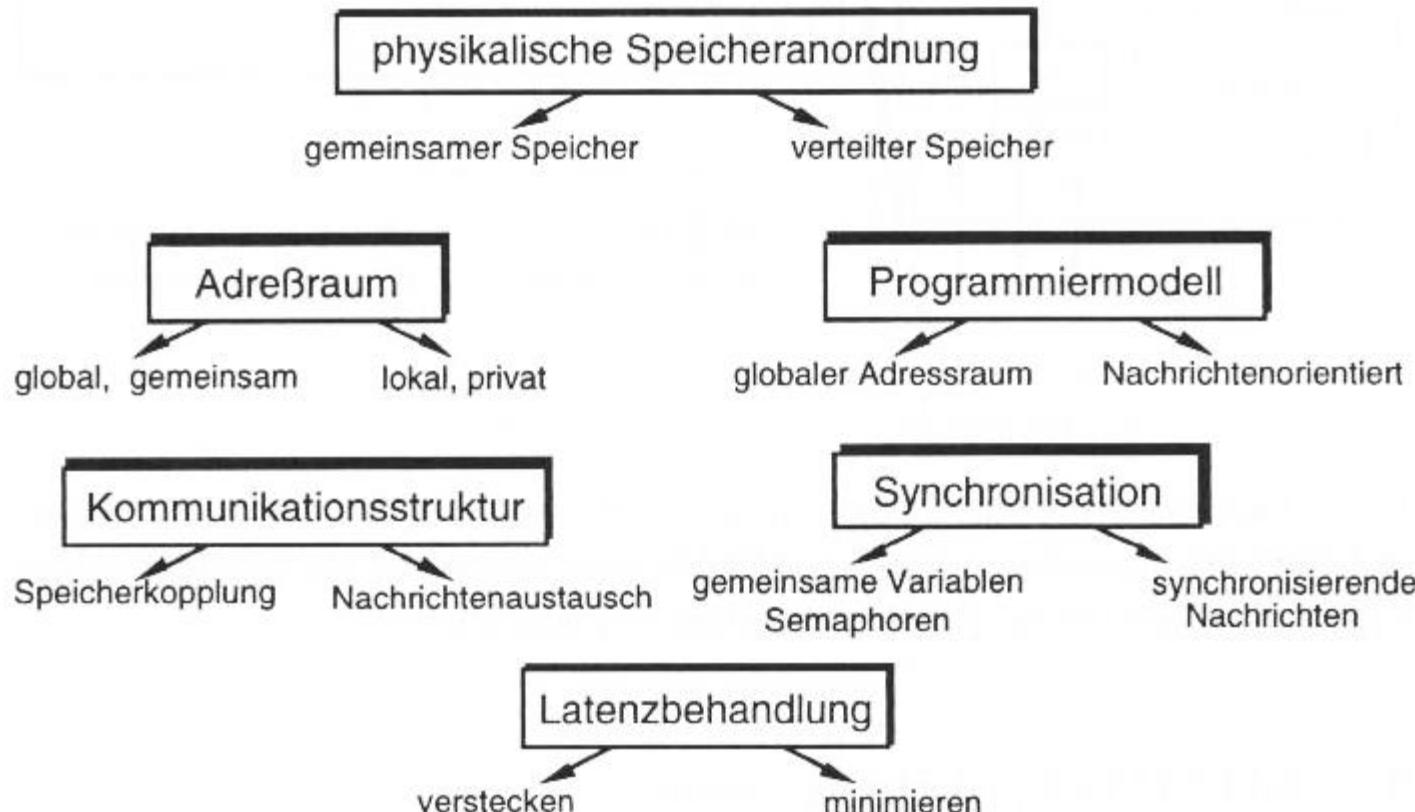


Bild-Quelle: Waldschmidt: Parallelrechner. Teubner, 1995

### 6.4.1. Überblick (III)

Merkmale bzw. Klassifizierungskriterien für MIMD-Systeme:

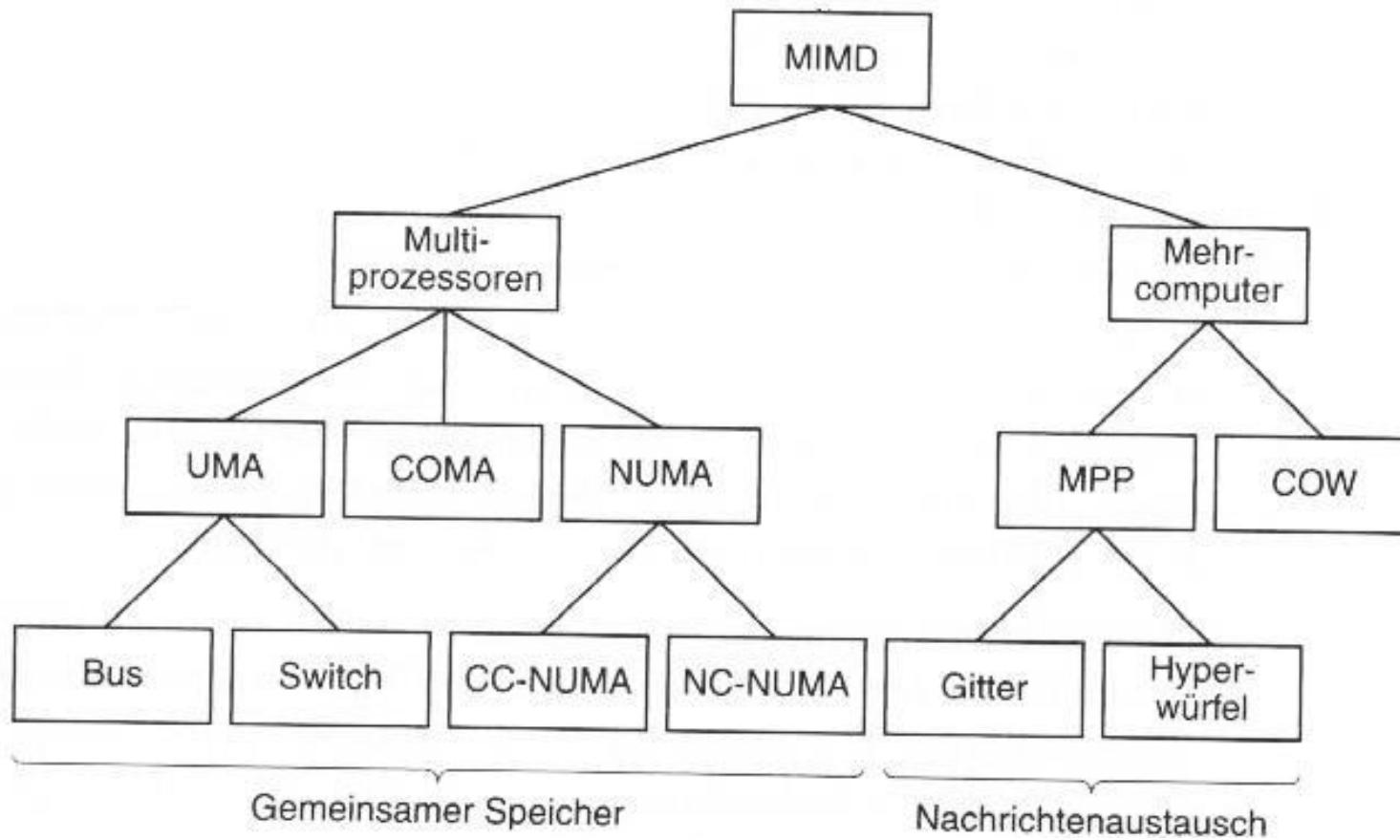
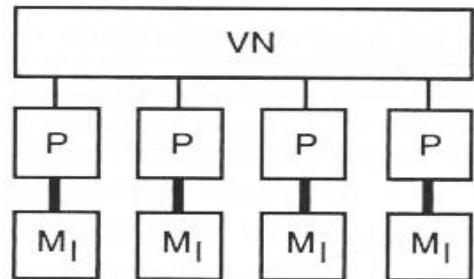
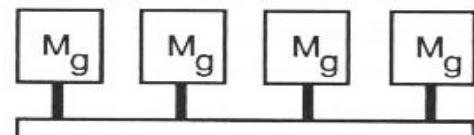


Bild-Quelle: Teil einer Abb. aus Tanenbaum/Goodman: Computerarchitektur. München: Pearson-Studium, 2001

### 6.4.1. Überblick (IV)

- bisherige Klassifikationen nicht einheitlich
- wichtigstes Differenzmerkmal: **physikal.** Speicheranordnung  
(Achtung: sie kann sich von der logischen – d.h. der Sicht auf die logischen Adressräume - unterscheiden!!)
  - gemeinsamer (phys.) Speicher für alle Prozessoren,  
„shared memory architectures“  
speichergekoppelte Systeme
  - (Mischformen)
  - kein gemeinsamer (phys.) Speicher für alle Prozessoren,  
also **verteilter Speicher**  
„distributed memory architectures“  
nachrichtengekoppelte Systeme



### 6.4.1. Überblick (V)

Schema für weitere Betrachtungen:

	physikalischer Speicher gemeinsam	physikalischer Speicher verteilt
logischer Adressraum gemeinsam	<p>Shared Memory Systeme <b>UMA</b></p> <p>Multithreaded Architectures, eng gekoppelte Multi- prozessoren, SMP, Multicore-Prozessoren</p>	<p>Distributed <u>Shared</u> Mem. Systeme (Mischform) <b>NUMA</b></p> <ul style="list-style-type: none"><li>• <b>NCC-NUMA,</b></li><li>• <b>CC-NUMA,</b></li><li>• <b>COMA</b></li></ul> <p>lose gekoppelte Multi- prozessoren</p>
logischer Adressraum verteilt	(leer)	<p>Distributed Memory Syst. <b>NORMA</b></p> <p>massiv parallele Systeme, Multicomputer, Cluster, Grid</p>

### 6.4.2. UMA-Architekturen

#### 6.4.2.1 allgemeine Merkmale von UMA

- **UMA = Uniform Memory Access**
- d.h. alle Prozessoren haben eine gleichförmige Zugriffslatenz zum physikalisch gemeinsamen Speicher mittels Load/Store-Befehlen
- dieser Effekt kann durch verschiedene Techniken erreicht werden

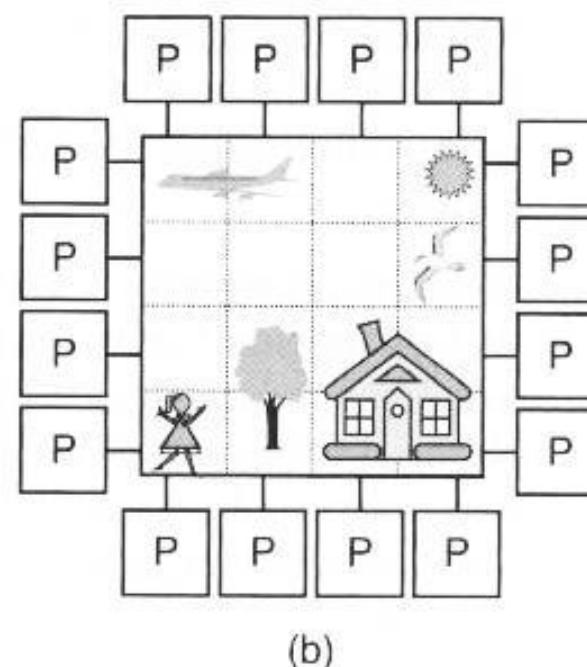
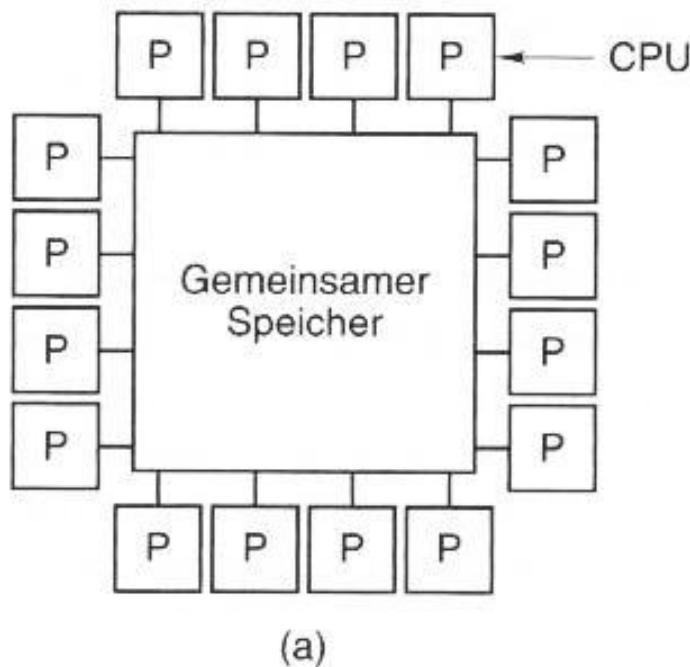


Bild-Quelle:  
Tanenbaum/Goodman:  
Computerarchitektur.  
München: Pearson-  
Studium, 2001

### 6.4.2.2 Simultaneous Multithreading SMT (I)

- → *hardwareseitiges* Multithreading innerhalb eines Prozessors
- Fähigkeit eines Prozessors, durch getrennte Pipelines und zusätzliche Registersätze mehrere Threads gleichzeitig auszuführen
- Aber: alle Threads nutzen Speicher, Bus und Caches gemeinsam!
- erweitert (mit rel. einfachen Mitteln) die Parallelität auf Befehls-ebene (vgl. Pipelines in superskalaren Architekturen) durch die Parallelität auf Threadebene
- Idee: Geschwindigkeitsgewinn durch „Verbergen“ von Latenz-Zeiten bzgl. Speicher- bzw. Netz-Zugriffen: reines Pipelining nutzt nur verschiedene unabhängige Befehle **eines** Prozesses/Threads aus; SMT verzahnt die Ausführung unabhängiger Befehle **mehrerer** Threads
- Typ. Gewinn ca. 15-30%, also nicht so leistungsfähig wie echte Multicore-Prozessoren

### 6.4.2.2 Simultaneous Multithreading SMT (II)

Idee:

Verbergen von Latenzzeiten durch Umschaltung zu anderen Threads

Lösungswege:

- interleaved multithreading:
  - Prozessor bearbeitet reihum eine (feste) Zahl von Threads
  - Kontextwechsel *nach jeweils einem Befehl* (feinkörnig!)
  - geeignete Festlegung der Threadanzahl nötig, damit zwischen zwei Aktivierungen desselben Threads genug Zeit bleibt, um die für die Durchführung seines letzten Befehls nötigen Speicherzugriffe auszuführen
- Block-Multithreading:
  - Kontextwechsel *nach einem Block von Befehlen*, d.h. erst dann, wenn ein Befehl mit nicht-lokalen Daten erreicht wird, der zum Warten des Prozessors führen würde (grobkörnig)
- Weitere: z.B. Kontextwechsel bei nicht erfolgreichen Cachezugriffen

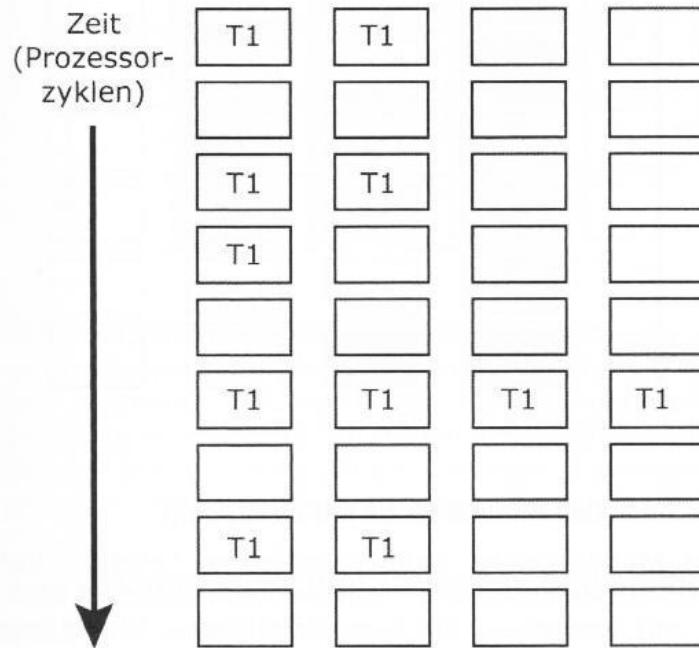
### 6.4.2.2 Simultaneous Multithreading SMT (III)

- Voraussetzung: Hardware-Unterstützung des Kontextwechsels!  
diese führt ggf. zu Spezialprozessoren: mehrere Befehlszähler und komplette Registersätze (je 1 pro Thread!) nötig!
- Abstand bzgl. Geschwindigkeit Prozessor - Speicher wächst weiter, daher ist eine noch höhere Threadanzahl nötig, um den Parallelisierungseffekt zu erreichen
- für Programmierer: im Programm „genügend“ parallelisierbare Threads „beschaffen“. Er braucht sich aber um Lokalität der Speicherzugriffe nicht zu kümmern.
- Beispiele:
  - Ältere Multithreaded Architectures (MTAs) waren z.B.: HEP (max. 16 Tasks mit je 128 Threads), TERA, Alewife, MANNA
  - Jüngere SMT-Architekturen: Intels Hyper-Threading Technologie (HTT): Pentium 4, Xeon; IBM Cell-Prozessor, POWER5, POWER6

### 6.4.2.2 Simultaneous Multithreading SMT (IV)

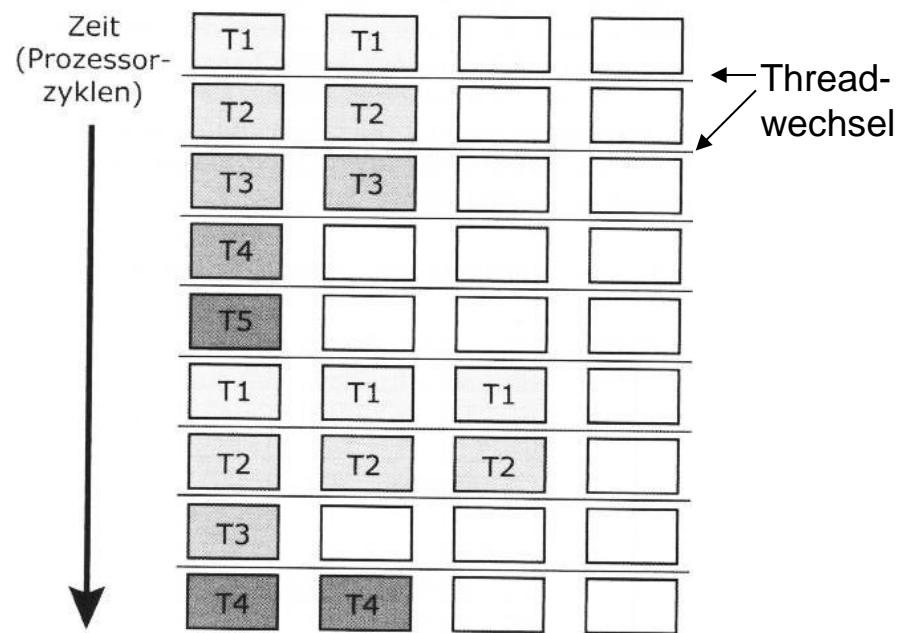
Beispiel: 4 parallele FEs in einem Prozessor

Superskalar-Prozessor:



FEs führen Befehle aus *einem* Thread aus, gelingt (nur) z.T. parallel, evtl. bleiben sogar ganze Zyklen frei weg. Speicherlatenz

Multithreaded Architecture:



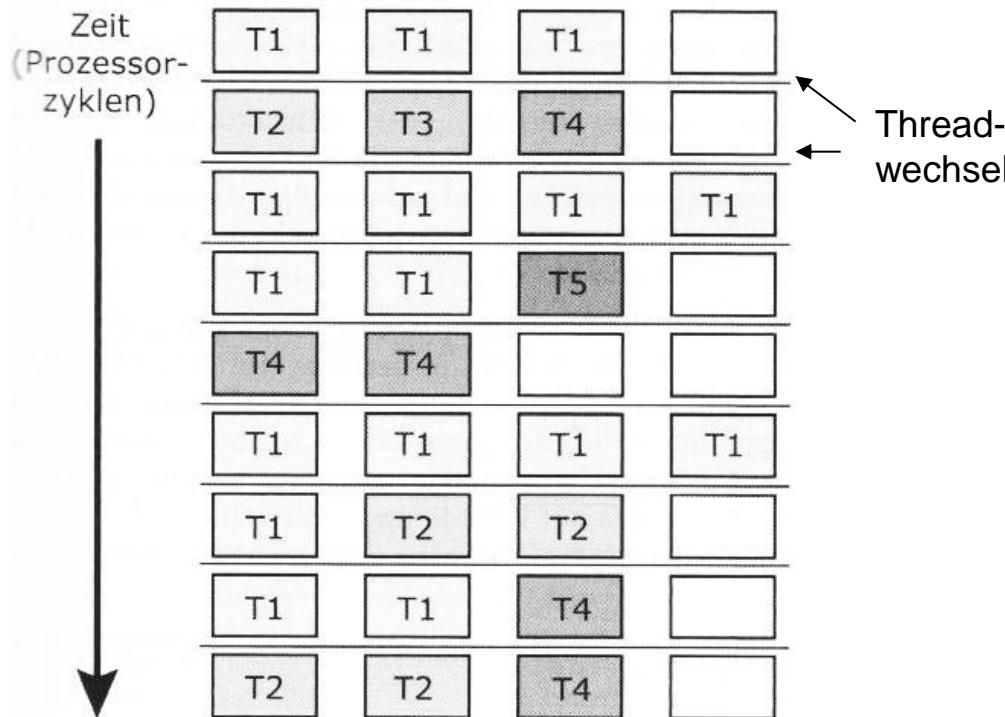
FEs führen Befehle aus *mehreren* Threads aus, dadurch sind komplett Leerzyklen vermeidbar

Bild-Quelle: Bengel: Masterkurs parallele und verteilte Systeme  
Vieweg/Teubner: 2008

### 6.4.2.2 Simultaneous Multithreading SMT (V)

Simultaneous Multithreading = Kopplung beider Verfahren:

Ausnutzung der möglichen Parallelität innerhalb eines Threads (durch die Superskalarität) und der Parallelität zwischen verschiedenen Threads (durch Threadwechsel)



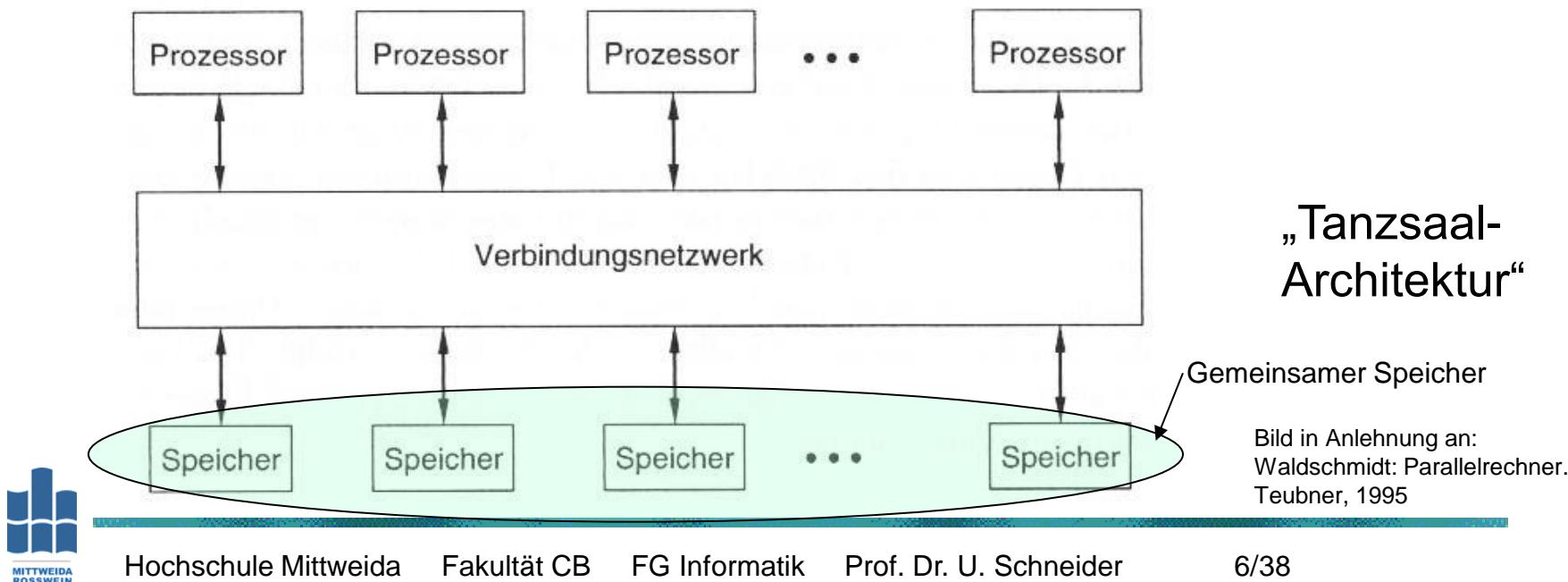
Threads mit *vielen* unabhäg. Befehlen profitieren von der Superskalarität.

Falls mehrere Threads mit *wenigen* unabhäg. Befehlen vorliegen, können sie in einem Zyklus parallel ausgeführt werden

Bild-Quelle: Bengel: Masterkurs parallele und verteilte Systeme  
Vieweg/Teubner: 2008

### 6.4.2.3 Eng gekoppelte Multiprozessoren (I)

- Hauptmerkmal: **physikalisch gemeinsamer (globaler) Speicher**, d.h. jeder Prozessor hat Zugriff auf den gesamten Speicher (der wiederum aus einzelnen Speichermodulen besteht)
- Es wird ein einheitlicher, **gemeinsamer logischer Adressraum** für alle Prozessoren gebildet, auf den diese mit gewöhnlichen Load- und Store-Befehlen zugreifen können
- Verbindung zwischen Prozessoren und Speicher via Netzwerk



### 6.4.2.3 Eng gekoppelte Multiprozessoren (II)

- Klassisches **UMA**-System  
(d.h. alle Prozessoren haben eine gleichförmige Zugriffslatenz zum Speicher)
- Problem: das Verbindungsnetzwerk (hier: Speicherbus) wird zum „Flaschenhals“ des Systems (Speicherzugriffe viel langsamer als Prozessorschritte)!
- Daher ist die Zahl der Prozessoren bei eng (speicher-)gekoppelten Multiprozessorsystemen stark begrenzt!
- Programmierung einfach, da gemeinsame Variablen und einfache Synchronisationsverfahren (Locks, Semaphore) sowie nur „lokale“ (und daher schnelle) Kommunikationsverfahren nutzbar sind (wie bei Einprozessorsystemen!)

### 6.4.2.3 Eng gekoppelte Multiprozessoren (III)

Sonderform eines eng gekoppelten Multiprozessorsystems:

**SMP = symmetrisches Multiprozessorsystem**

- Alle Prozessoren haben gleiche Funktionalität und gleiche Sicht auf den „Gesamtrechner“
- Prozessoren sind evtl. mit lokalem Cache ausgerüstet (um den Speicherbus teilweise zu entlasten), ggf. Sicherung der Cache-Kohärenz (Cache-Konsistenz) nötig [→ später]!
- Gleiche Zugriffszeit jedes Prozessors zum gemeinsamen Speicher
- Gleiche Zugriffszeit jedes Prozessors zu seinem Cache
- Speicherbusbasierte SMP-Systeme üblich mit 16 (max. 64) Prozessoren, also wegen Busbreite stark begrenzte CPU-Zahl
- Erweiterungen/Leistungsverbesserungen nur mittels anderer Verbindungseinrichtungen (kein Bus, sondern z.B. Kreuzschienenverteiler, Mehrebenen-Netzwerke [→ später]) möglich, aber teuer und daher letztlich auch begrenzt

### 6.4.2.3 Eng gekoppelte Multiprozessoren (IV)

SMP:

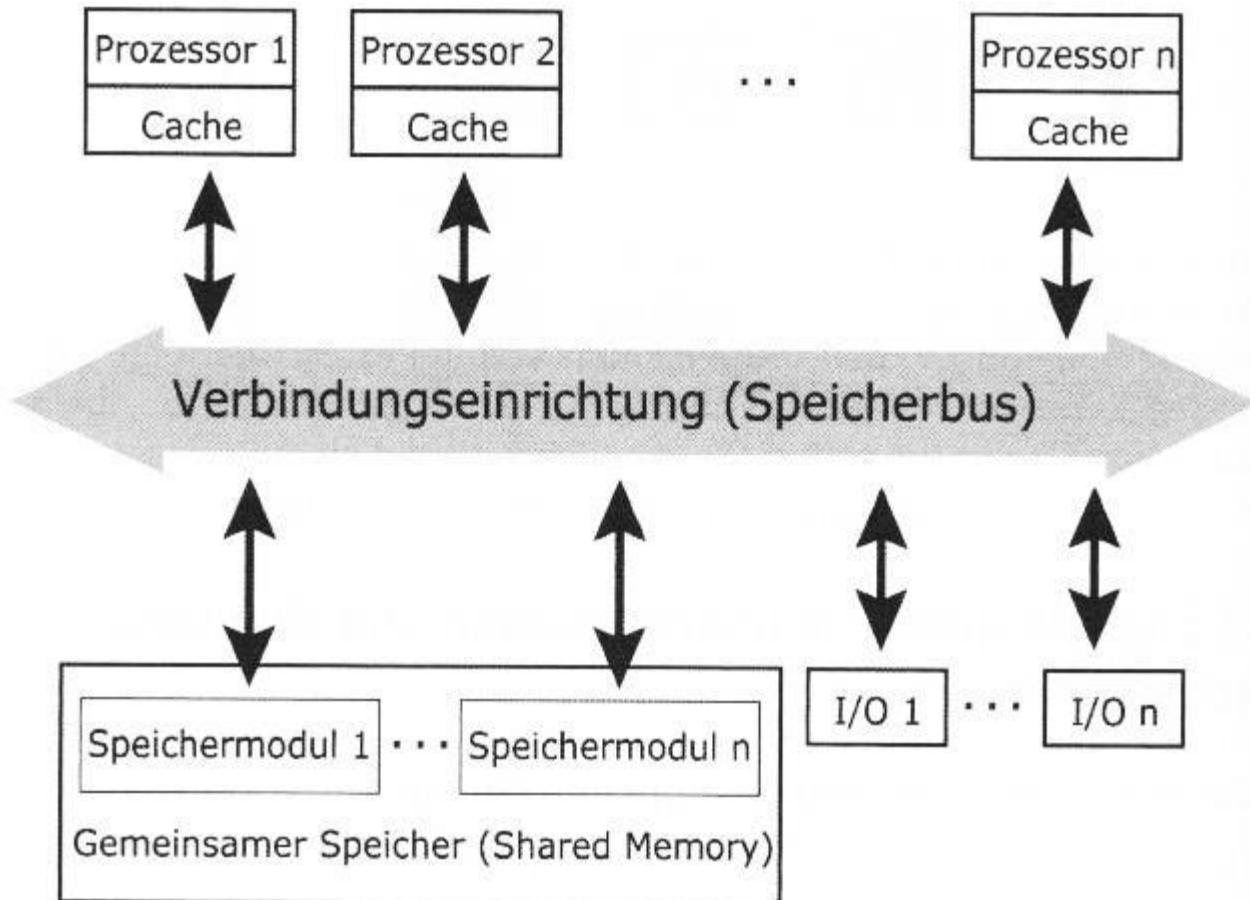


Bild-Quelle: Bengel:  
Masterkurs parallele und verteilte Systeme  
Vieweg/Taubner: 2008

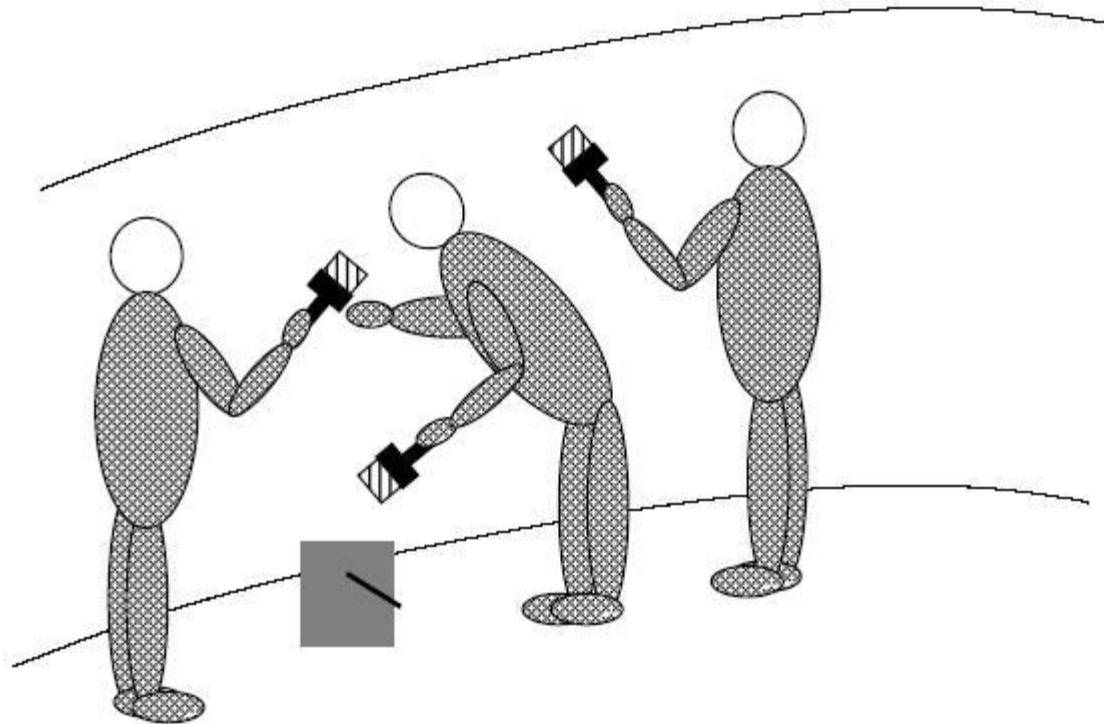
### 6.4.2.3 Eng gekoppelte Multiprozessoren (V)

SMP:

- Die vielen Prozessoren sind für Applikation weitgehend unsichtbar
- Häufiges Programmiermodell: Multithreading (POSIX pthreads!)
- Alle Prozessoren arbeiten mit einem einzigen BS, dieses verteilt Prozesse/Threads auf die (gleichen) Prozessoren
- Probleme:
  - „CPU-Hopping“, wenn Prozesse mehrfach den Prozessor wechseln;
  - „Cache–Thrashing“ durch ständigen Wechsel der Cache-Inhalte durch unterschiedliche Prozesse
- Möglicher Ausweg: BS achtet auf Prozessor-Affinität
- Anwendung oft bei Servern (inkl. Multiuser-Betrieb)
- Bsp.-Systeme: Sun Enterprise, SGI Challenge, ...

### 6.4.2.3 Eng gekoppelte Multiprozessoren (VI)

„SMP in Action“:



Quelle: epcc (Edinburgh)

### 6.4.2.3 Eng gekoppelte Multiprozessoren (VII)

Weitere Sonderform der eng gekoppelten Multiprozessorsysteme:

**Multicore-Prozessoren (Mehrkern-Prozessoren)**

auch: Multiprocessor Systems on-chip (MPSoC)

- Jeder Kern arbeitet wie ein eigener Prozessor, der eng mit weiteren Prozessoren (auf dem selben Chip) gekoppelt ist

aktuelle Systeme:      2 Kerne: Dual-Core Prozessoren

                          4 Kerne: Quad-Core Prozessoren

                          6 Kerne: „??“, z.B. Intel Xeon X760

demnächst:            8 Kerne: Oct-Core Prozessoren ....

                          ... und mehr (Prognose Intel für 2015: >100 Kerne!)

technologischer Hintergrund:

Die durch höhere Integrationsdichte auf dem Chip „gewonnene“ Fläche wird nicht mehr zur weiteren Steigerung der Taktrate benutzt (u.a. wegen Energieverbrauch und Wärmeproblem), sondern zur Veränderung der Prozessorarchitektur selbst

### 6.4.2.3 Eng gekoppelte Multiprozessoren (VIII)

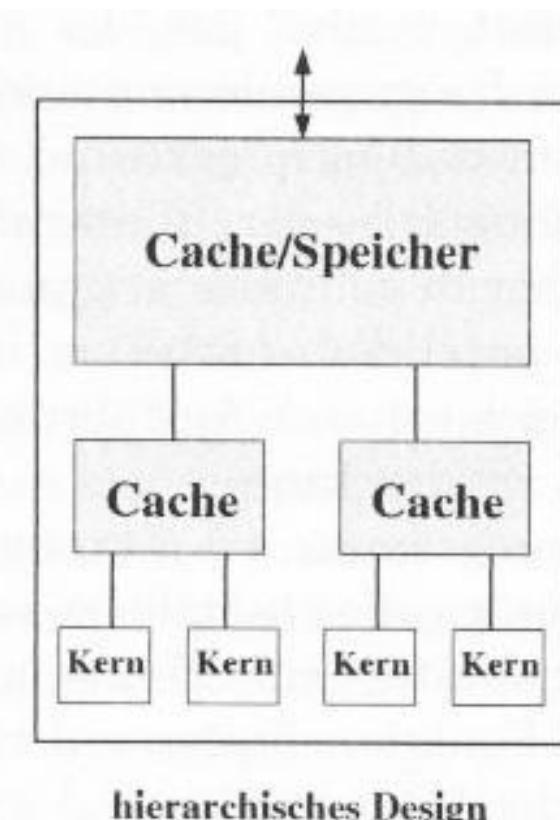
Multicore-Prozessoren:

- Implementierungsvarianten:
  - Hierarchisches Design
  - Pipeline-Design
  - Netzwerkbasiertes-Design

### 6.4.2.3 Eng gekoppelte Multiprozessoren (IX)

a) Multicore-Prozessoren mit hierarchischem Design:

- Mehrere Prozessorkerne teilen sich mehrere Caches mit unterschiedl. Kapazität, die hierarchisch angeordnet sind



Beispiel:

Quadcore mit 3 stufigem Cache

- L1 Cache für jeden Core
- L2 Cache für je 2 Cores gemeinsam
- Hauptspeicher für alle Cores gemeinsam

konkrete Systeme mit hierarch. Design:

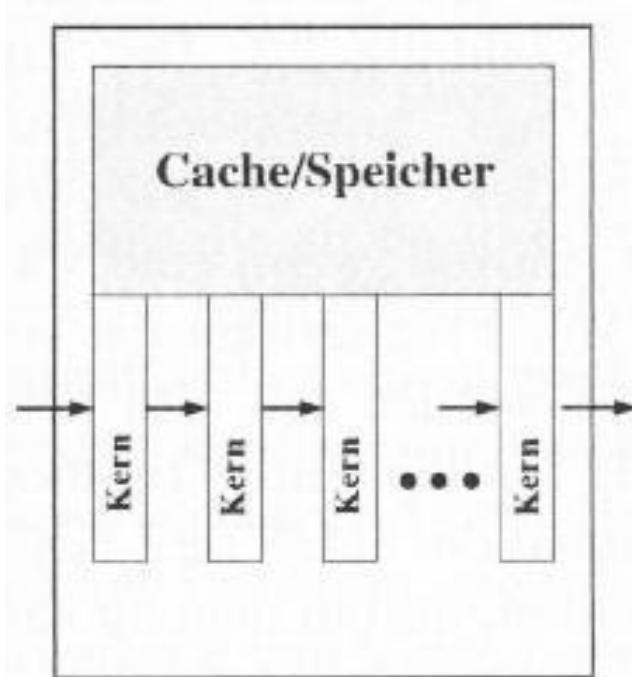
- IBM Power 5 (2 Kerne mit je 2 SMT)
- Intel Core 2 (2 oder 4 Kerne, geplant >8 Kerne)
- Sun Niagara 2 (8 Kerne mit je 8 SMT-Threads)

Bild-Quelle: Rauber/Rünger: Multicore:  
Parallele Programmierung. Berlin: Springer, 2008

### 6.4.2.3 Eng gekoppelte Multiprozessoren (X)

b) Multicore-Prozessoren mit Pipeline-Design:

- Daten werden durch mehrere Kerne in Art einer Pipeline nacheinander verarbeitet und vom letzten Kern in den Speicher geschrieben

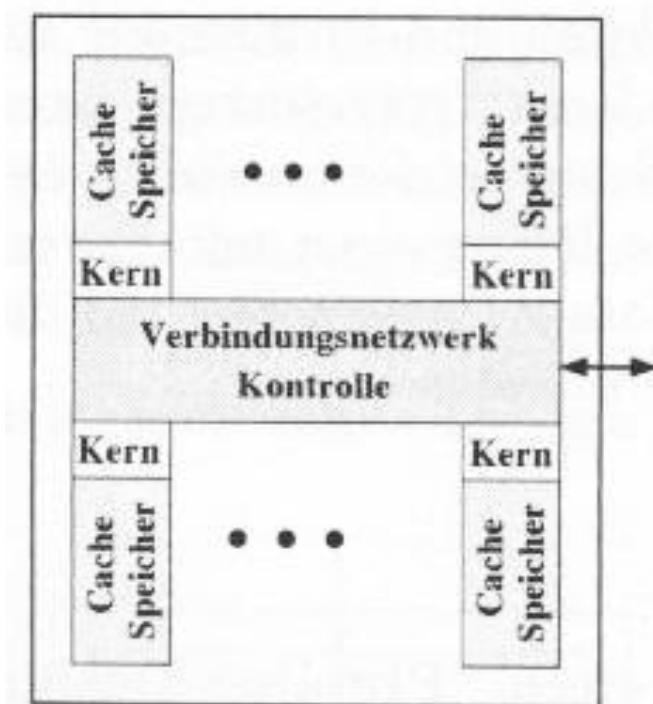


Beispiel:  
Multicore mit Pipeline-Design

konkrete Systeme mit Pipeline-Design:  
vor allem spezialisierte Prozessoren für  
Netzwerksteuerung oder Grafik, z.B.  
Xelerator Netzwerkprozessoren  
• X10 (200 Prozessorkerne),  
• X11 (bis 800 Kerne)

### 6.4.2.3 Eng gekoppelte Multiprozessoren (XI)

- c) Multicore-Prozessoren mit netzwerkbasierterem Design:  
→ Kerne und ihre lokalen Caches sind über ein Chip-internes Verbindungsnetzwerk mit den anderen Kernen gekoppelt



Netzwerkbasiertes Design

Beispiel:  
Multicore mit netzwerkbasiertem Design

konkrete Systeme mit Netzwerk-Design:

- Intel Teraflop-Prozessor mit 80 Kernen  
(Intel Tera-Scale-Initiative)

### 6.4.2.3 Eng gekoppelte Multiprozessoren (XII)

Multicore-Prozessoren: Gleichartigkeit der Kerne?

- *Symmetrische oder homogene Multicore-Prozessoren:*
  - alle Kerne sind gleich
  - ein Programm kann auf jedem beliebigen Kern laufen
- *Asymmetrische oder heterogene Multicore-Prozessoren:*
  - unterschiedliche Kerne für verschiedene Aufgaben
  - etwa wie klassische Prozessoren plus Co-Prozessoren
  - Programm läuft nur auf dem „passenden Kern“
  - Anwendung vor allem für „embedded Audio/Video-Systeme“, z.B. bei TV, DVD-Player, Camcorder Spielekonsolen,...
  - Beispiel: Cell-Prozessor

### 6.4.2.3 Eng gekoppelte Multiprozessoren (XIII)

Multicore-Prozessoren: Beispiel Intel Core 2 Duo:

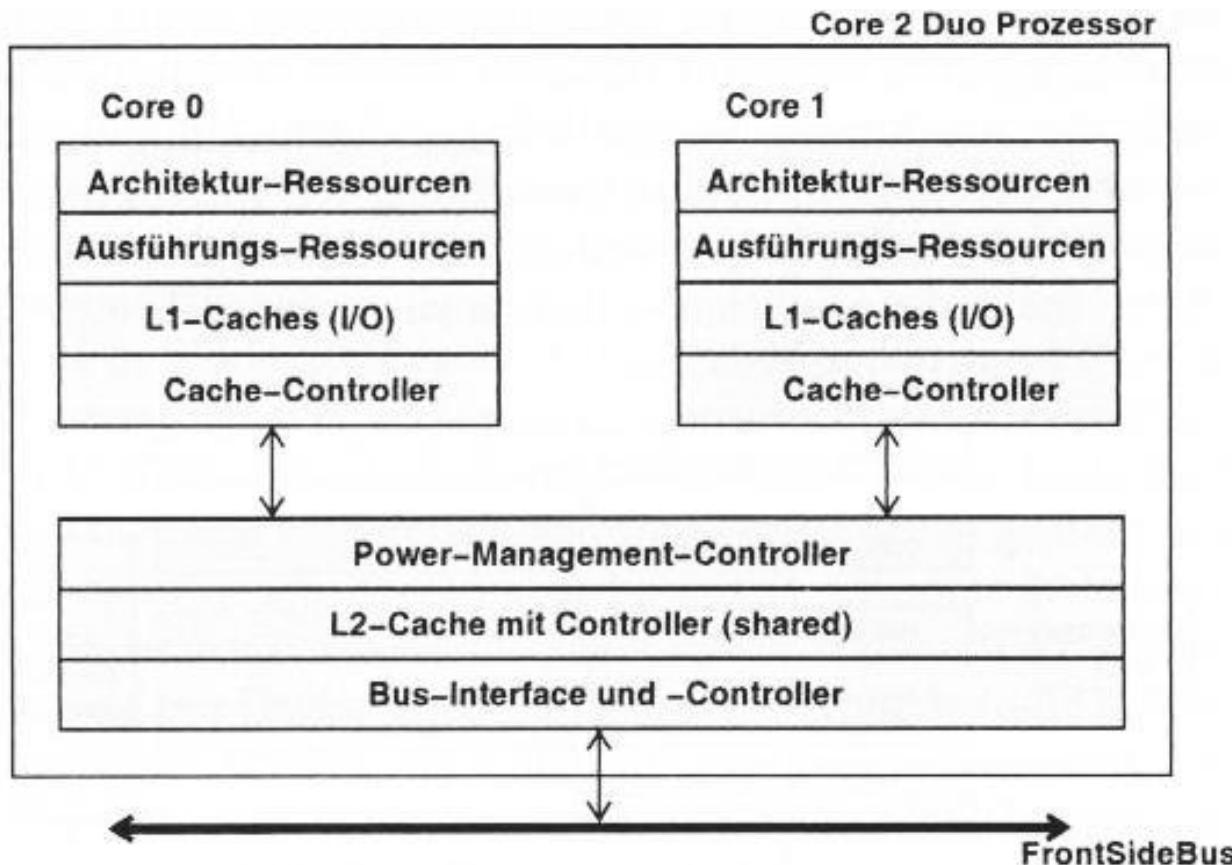
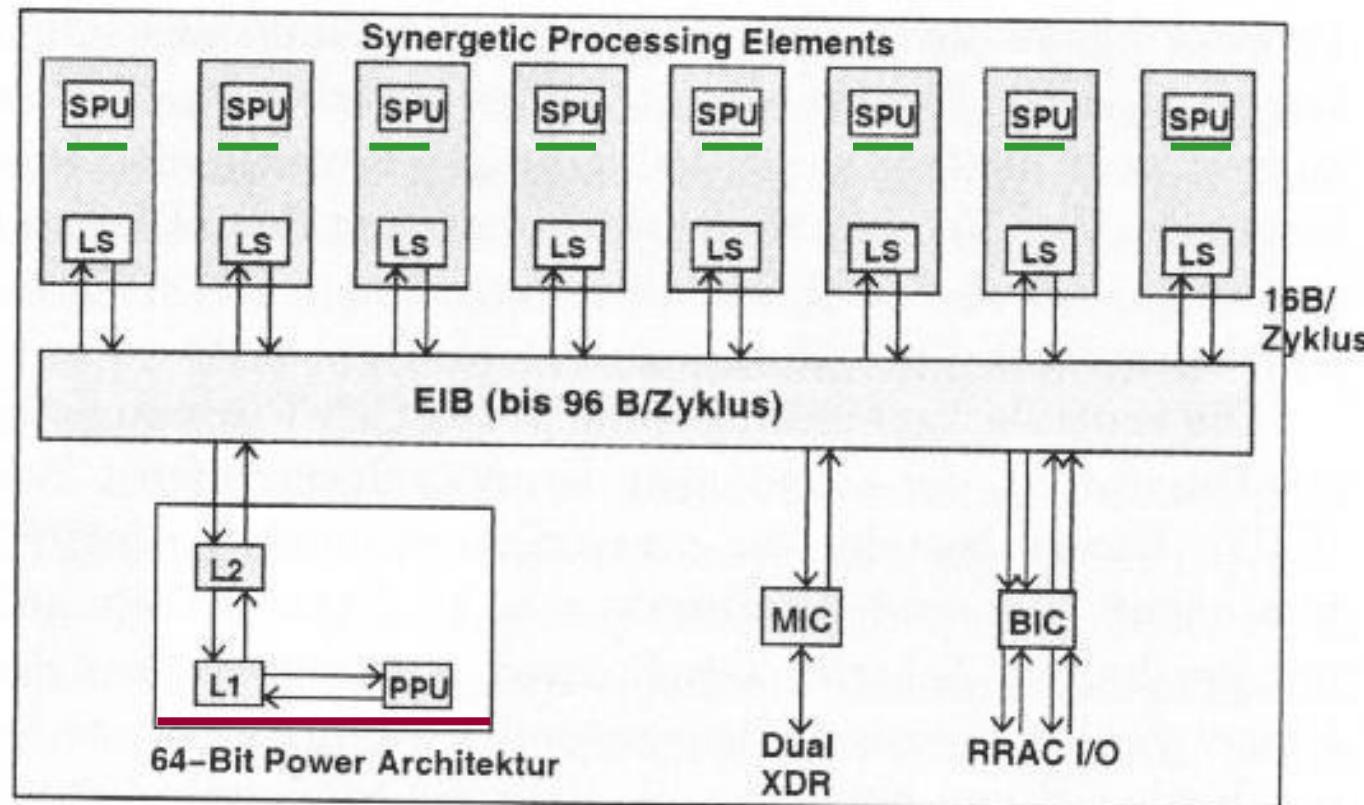


Bild-Quelle: Rauber/Rünger: Multicore:  
Parallele Programmierung. Berlin: Springer, 2008

### 6.4.2.3 Eng gekoppelte Multiprozessoren (XIV)

Multicore-Prozessoren: Beispiel Cell Broadband Engine CBE  
(Cell Prozessor: 1 Power-CPU + 8 SIMD-Prozessoren)



SPU: Synergetic Processing Unit (Vektorprozessor, 4xFK + 4xGK Ops gleichzeitig)

LS: Local Storage

EIB: Element Interconnect Bus

PPU: Power Processing Unit (64Bit Prozessor, 2xSimMultithr. SMT)

MIC: Memory Interface Controller

BIC: I/O-Bus Interface Controller

### 6.4.2.3 Eng gekoppelte Multiprozessoren (XV)

Multicore-Prozessoren: Beispiel heterogener Multiprozessor  
für DVD-Player

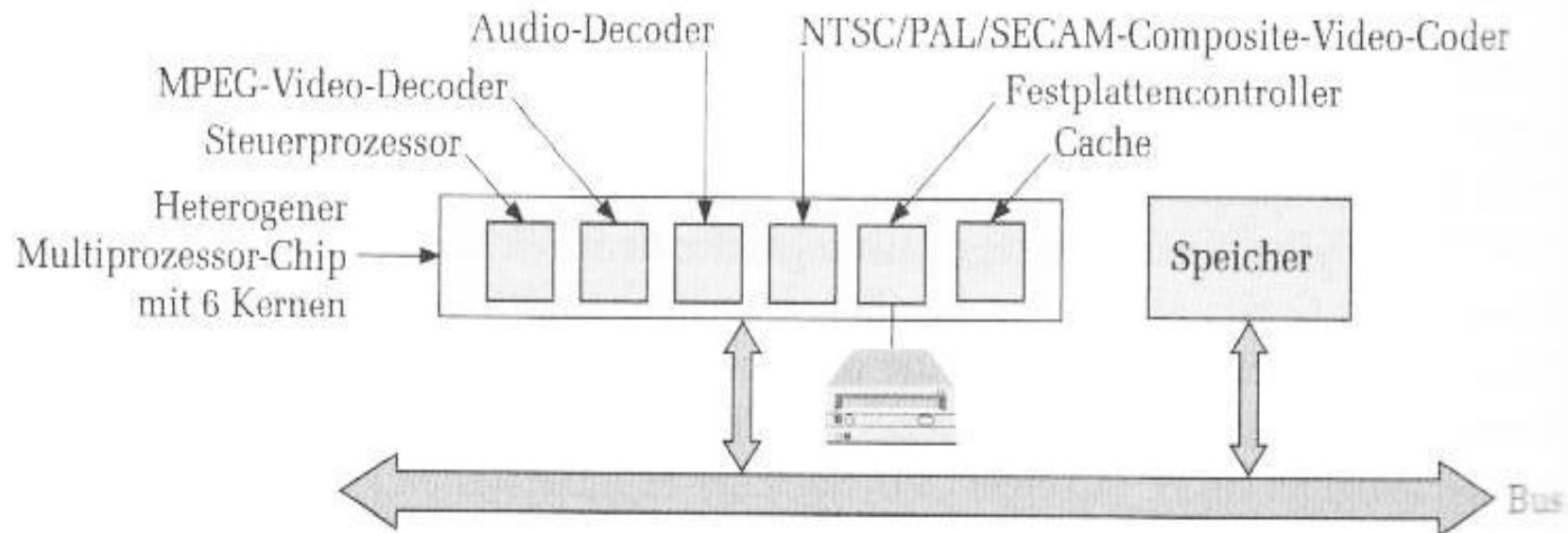


Abbildung 8.9: Die logische Struktur eines einfachen DVD-Players enthält einen heterogenen Multiprozessor mit mehreren Kernen für unterschiedliche Funktionen.

Bild-Quelle: Tanenbaum: Computerarchitektur.  
München: Pearson Studium, 2006

## 6.4.2.3 Eng gekoppelte Multiprozessoren (XVI)

### Multicore-Prozessoren: weitere aktuelle Beispiele

Prozessor	Intel Core 2 Duo	IBM Power 5	AMD Opteron	Sun T1
Prozessorkerne	2	2	2	8
Instruktionen pro Zyklus	4	4	3	1
SMT	nein	ja	nein	ja
L1-Cache I/D in KB per core	32/32	64/32	64/64	16/8
L2-Cache	4 MB shared	1.9 MB shared	1 MB per core	3 MB shared
Taktrate (GHz)	2.66	1.9	2.4	1.2
Transistoren	291 Mio	276 Mio	233 Mio	300 Mio
Stromverbrauch	65 W	125 W	110 W	79 W

## 6.4.3 NUMA-Architekturen

### 6.4.3.1 allgemeine Merkmale von NUMA (I)

**NUMA = Non-Uniform Memory Access**

d.h. die Prozessoren haben keine gleichförmige Zugriffslatenz zum physikalischen Speicher

- Prozessoren verfügen jeweils über private (lokale) Speichermodule
  - also kein physikalisch gemeinsamer Speicher, daher hat bei jedem Speicherzugriff eines Prozessors die „Entfernung“ des Speichers Einfluss auf die Zugriffszeit,  
→ unterschiedl. Zugriffszeiten:
    - schnell zum lokalen Speicher,
    - langsam zum entfernten Speicher
  - aber virtuell gemeinsamer Adressraum („vorgetäuscht“ durch geeignete Kohärenzprotokolle u.ä.), damit einheitliche Sicht für Programmierer, Speicherzugriff via Load/Store-Befehle
- „**Distributed Shared Memory**“ (DSM), „**Virtual Shared Memory**“

### 6.4.3.1 allgemeine Merkmale von NUMA (II)

NUMA typisch für *lose gekoppelte* Multiprozessorsysteme

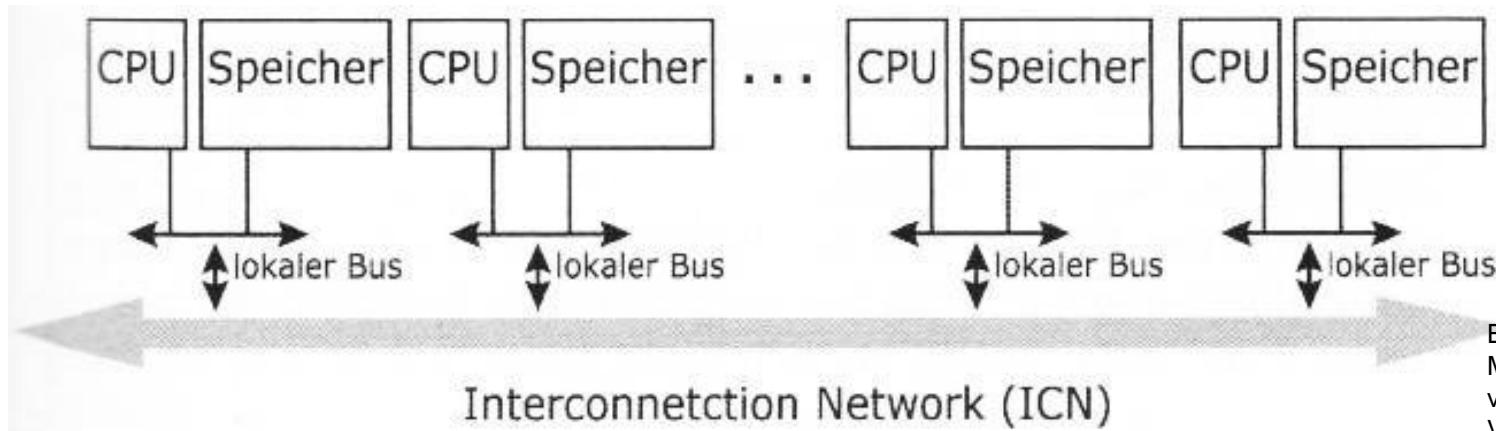


Bild-Quelle: Bengel:  
Masterkurs parallele und  
verteilte Systeme  
Vieweg/Teubner: 2008

Vorteil: Verzicht auf gleichförmige Speicherzugriffszeit erlaubt bessere Skalierung, d.h. Prozessoranzahl dadurch nicht beschränkt UMA-Programme laufen auch auf NUMA (evtl. langsamer)

Unterscheidung von NUMA-Systemen bzgl. Caches:

- NCC-NUMA: Non Cache Coherent NUMA
- CC-NUMA: Cache Coherent NUMA
- COMA: Cache Only Memory Architecture

### 6.4.3.2 NCC-NUMA (I)

- Es gibt keine Cache-Kohärenz, d.h. keine HW-Unterstützung dafür
- Lokaler Cache ist also nur für lokale Daten des Prozessors nutzbar
- Schreibzugriffe zum Speicher anderer Prozessoren immer nur erneut (und langsamer) über das Verbindungsnetz möglich

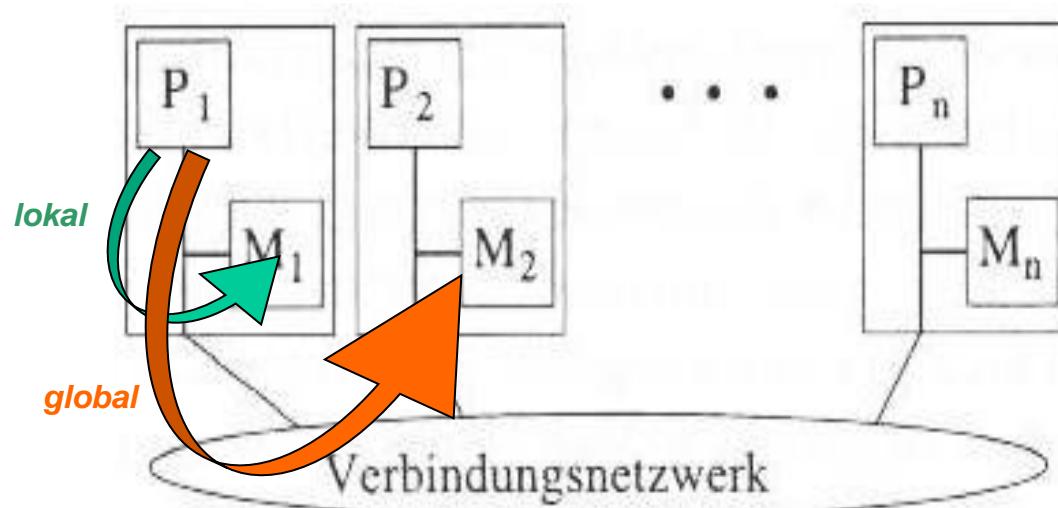


Bild-Quelle: Rauber/Rünger:  
Parallele und verteilte Programmierung.  
Berlin: Springer, 2000 (ergänzt)

- Realisierungsbeispiele:
  - Carnegie Mellon Cm\*
  - Cray T2D/T3E

### 6.4.3.3 CC-NUMA (I)

- Cache-Kohärenz mittels HW gesichert
  - Typische Lösung: Bei jedem Zugriff auf eine Cachezeile wird ein Verzeichnis (Datenbank) danach befragt, wo und in welchem Zustand (modifiziert?) sich die Cachezeile befindet.
  - Das Verzeichnis muss bei jedem Speicherzugriff abgefragt werden, daher extrem schnelle Implementierung mittels Spezial-HW nötig!
  - Damit können im lokalen Cache eines Prozessors auch Daten aus entfernten Speichern anderer Prozessoren zwischengespeichert werden
- [Verzeichnisbasierte Multiprozessoren \(Directory-based Multiprocessors\)](#)
- Verlangsamung bei entfernten Zugriffen wird durch Caching rel. gut verborgen, trotzdem gibt es Cache-Fehler, falls nicht alle geforderten Daten in den Cache passen. Leistungsverlust auch, wenn mehrere Prozessoren schnell nacheinander auf dieselbe Speicherstelle zugreifen wollen.

### 6.4.3.3 CC-NUMA (II)

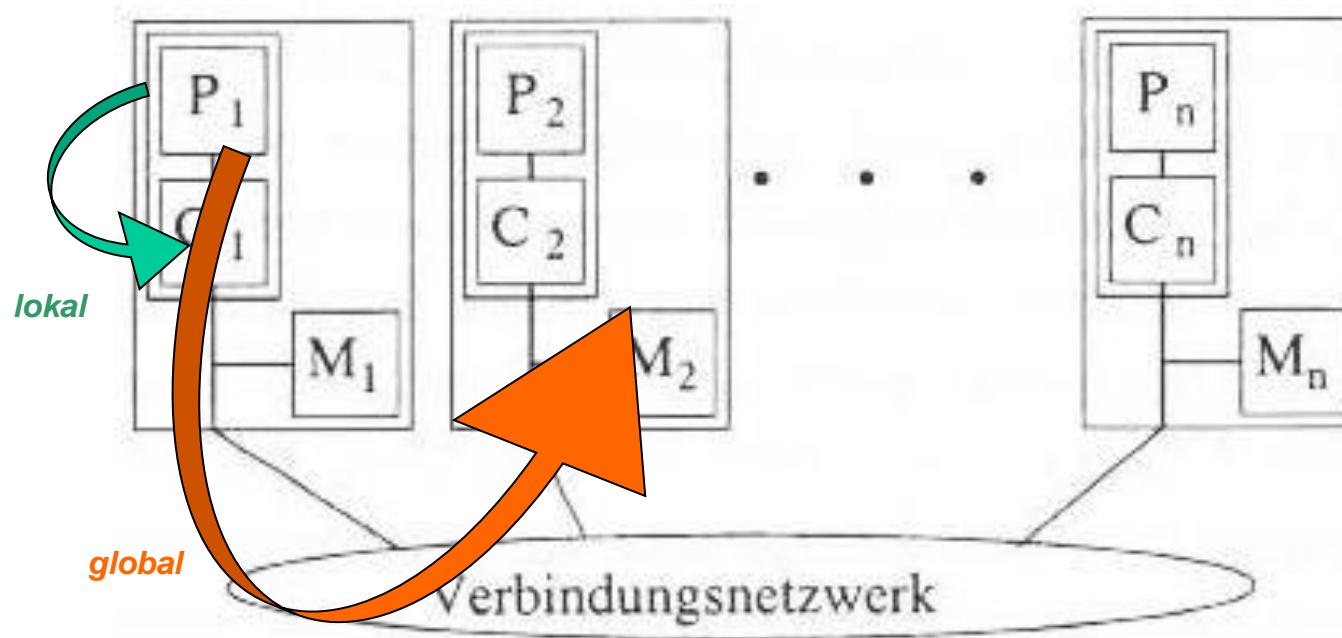


Bild-Quelle: Rauber/Rünger:  
Parallele und verteilte Programmierung.  
Berlin: Springer, 2000 (ergänzt)

- Realisierungsbeispiele:
  - Stanford DASH, FLASH
  - SGI Origin 2000, SGI Altix
  - Sun Fire (z.B. E25K)
  - AMD Multiprozessorsysteme mit Opteron

### 6.4.3.4 COMA (I)

- Gesamter Hauptspeicher jedes Prozessors wird als kohärenzgesicherter Cache benutzt!
- Der physische Adressraum wird in Cachezeilen unterteilt, diese „wandern“ quasi im Gesamtsystem umher, d.h. Variablen haben keine feste Adressen
- „Attraction Memory“ = Speicher, der nur die gerade benötigten Zeilen an sich zieht; da nur Zugriffe auf lokalen Speicher stattfinden, sind diese immer attraktiv...
- Beispiel: P1 will auf Speicherelement zugreifen, das z.Zt. im Cache von P2 liegt. Dieses Element wird nun als Kopie im Cache von P1 abgelegt. Falls später die ursprüngliche Kopie im Cache von P2 ungültig wird (z.B. wegen Überschreiben), dann wird die Kopie im Cache von P1 als (evtl. einzige) gültige Kopie markiert. Zugriffswünsche anderer Prozessoren nach diesem Element werden dann an P1 weitergeleitet. Die letzte Kopie eines Elements darf nicht gelöscht werden ...
- Nachteil: evtl. hoher Zeitaufwand bzgl. Suche

#### 6.4.3.4 COMA (II)

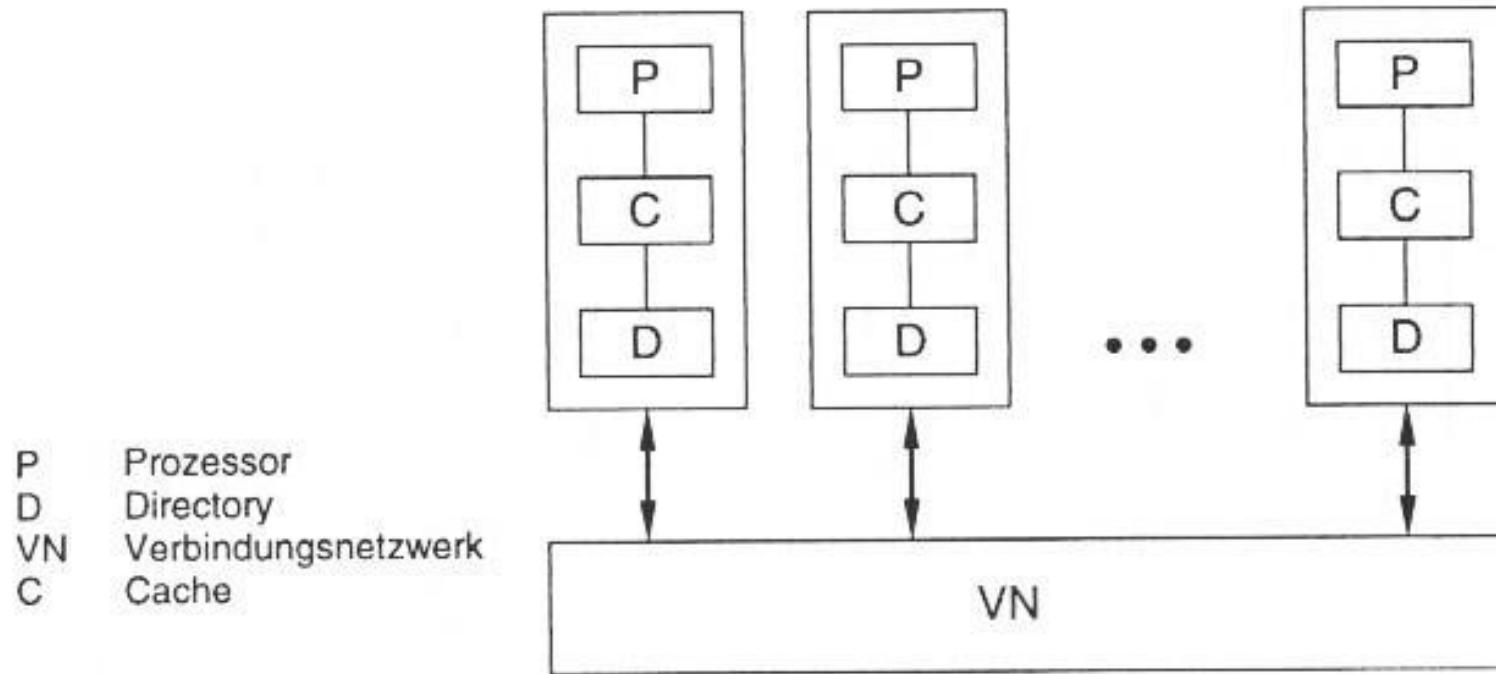


Bild-Quelle: Waldschmidt:  
Parallelrechner. Teubner, 1995

- Realisierungsbeispiele:
  - Kendall Square Research KSR-1, KSR-2
  - Data Diffusion Machine
  - SDAARC

### 6.4.3.5 Cache-Kohärenzprotokolle (I)

Problem (Beispiele):

- Zwei Prozessoren haben beide die Adresse x in ihrem Cache und verändern den Wert der Variablen auf dieser Adresse. Danach schreiben sie ihr Ergebnis in den Speicher – welcher Wert gilt jetzt?
- Ein Prozessor holt Daten aus dem Speicher eines anderen Prozessors, führt damit Berechnungen aus und schreibt die Ergebnisse in seinen lokalen Cache. Der Cache des Prozessors, von dem die Daten stammen (und vielleicht auch noch weitere Caches im System) wissen von dieser Änderung nichts ...

Forderungen:

- **Konsistenz:** Im Hauptspeicher und in den Caches dürfen zu keiner Zeit unterschiedliche Kopien desselben Datenblocks liegen
- **Kohärenz** = abgeschwächte Forderung in Multiprozessorsystemen: Beim Lesen eines Blockes, der durch mehrere Zugriffe verändert wurde, muss immer der zuletzt geschriebene Wert geliefert werden

### 6.4.3.5 Cache-Kohärenzprotokolle (II)

Datenkohärenz ist bei Multiprozessorsystemen gegeben, wenn

1. zwar jeder Cache eine Kopie von Daten aus dem Hauptspeicher haben darf
2. aber nur ein Cache eine *veränderte* Kopie der Daten besitzen darf, jedoch nur solange, wie kein anderer Prozessor diese Daten liest

Lösung:

Beim Schreiben neuer Werte in den Cache müssen auch dieselben Werte in allen anderen Caches und im Hauptspeicher neu geschrieben werden!

→ Synchronisation zwischen Caches (und HS)

### 6.4.3.5 Cache-Kohärenzprotokolle (III)

Häufig genutzte Technik: „**Snooping**“

- Speicherzugriffe laufen über ein gemeinsames Medium (Bus oder Verbindungsnetzwerk)
- Alle angeschlossenen Cache-Controller „beschnüffeln“ dieses Medium und identifizieren Zugriffe auf Blöcke, die sie jeweils selbst zwischengespeichert haben
- Sie regieren dann mit einem Protokoll

Methoden dieser Protokolle:

- „**write through**“: Durchschreiben, d.h. jedes Schreiben in einen Cache führt sofort auch zum Schreiben dieser Daten in den Hauptspeicher; für Multiprozessorsystem wegen zu hoher Busbelastung nicht akzeptabel
- „**deferred write**“: verzögertes Rückschreiben der Daten aus dem Cache in den HS erst dann, wenn die Daten aus dem Cache verdrängt werden

### 6.4.3.5 Cache-Kohärenzprotokolle (IV)

Protokollfamilie für deferred-write: Write-Back-Invalidate-Protocol  
Write Invalidate Snoopy Cache Prot.

- Dabei Unterscheidung der Protokolle nach den verschiedenen Zuständen, die eine Cache-Zeile annehmen kann (Kennzeichnung der Cachezeile durch Zusatzbits)
  - **MSI:** Modified-Shared-Invalid-Protocol (veraltet)
  - **MESI:** Modified-Exclusive-Shared-Invalid-Protocol
  - **MOESI:** Modified-Owner-Exclusive-Shared-Invalid-Protocol

Alternative Kohärenzprotokolle zum Snooping:

- Verzeichnisbasierte Protokolle (siehe CC-NUMA)

Beispiele in realen Systemen:

- MESI: Intel Prozessoren, IBM Power PC
- MOESI: Sun UltraSPARC II, Athlon MP
- Verzeichnisbasiert: SGI Origin

#### 6.4.4 NORMA-Architekturen

##### 6.4.4.1 allgemeine Merkmale von NORMA (I)

NORMA = No Remote Memory Access

kein direkter Zugriff auf Speicher anderer Prozessoren

- Prozessoren verfügen jeweils über private (lokale) Speichermodule
- kein physikalisch gemeinsamer Speicher
- und auch kein virtuell gemeinsamer Adressraum!
- leistungsfähiges Verbindungsnetzwerk zwischen den Knoten nötig

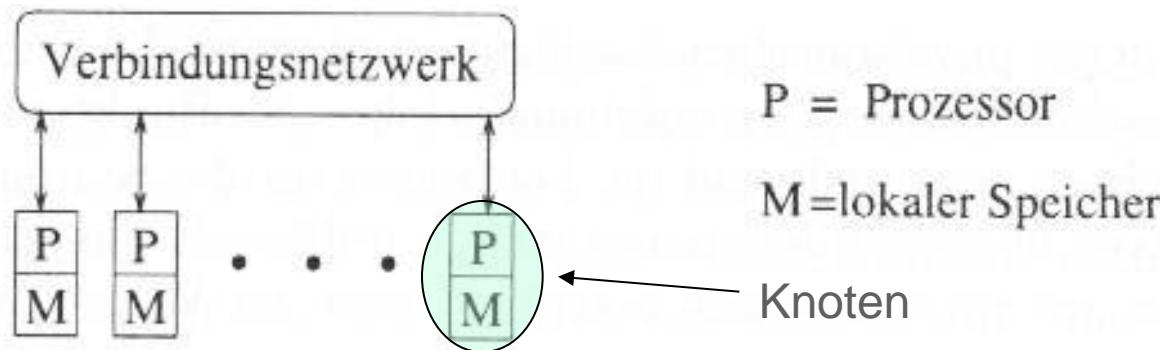


Bild-Quelle: Rauber/Rünger:  
Parallele und verteilte  
Programmierung.  
Berlin: Springer, 2000  
(ergänzt)

- Distributed Memory System / Distributed Memory Machine (DMM)
- Mehrrechnersysteme (Multicomputersysteme)

#### 6.4.4.1 allgemeine Merkmale von NORMA (II)

##### Detailansicht

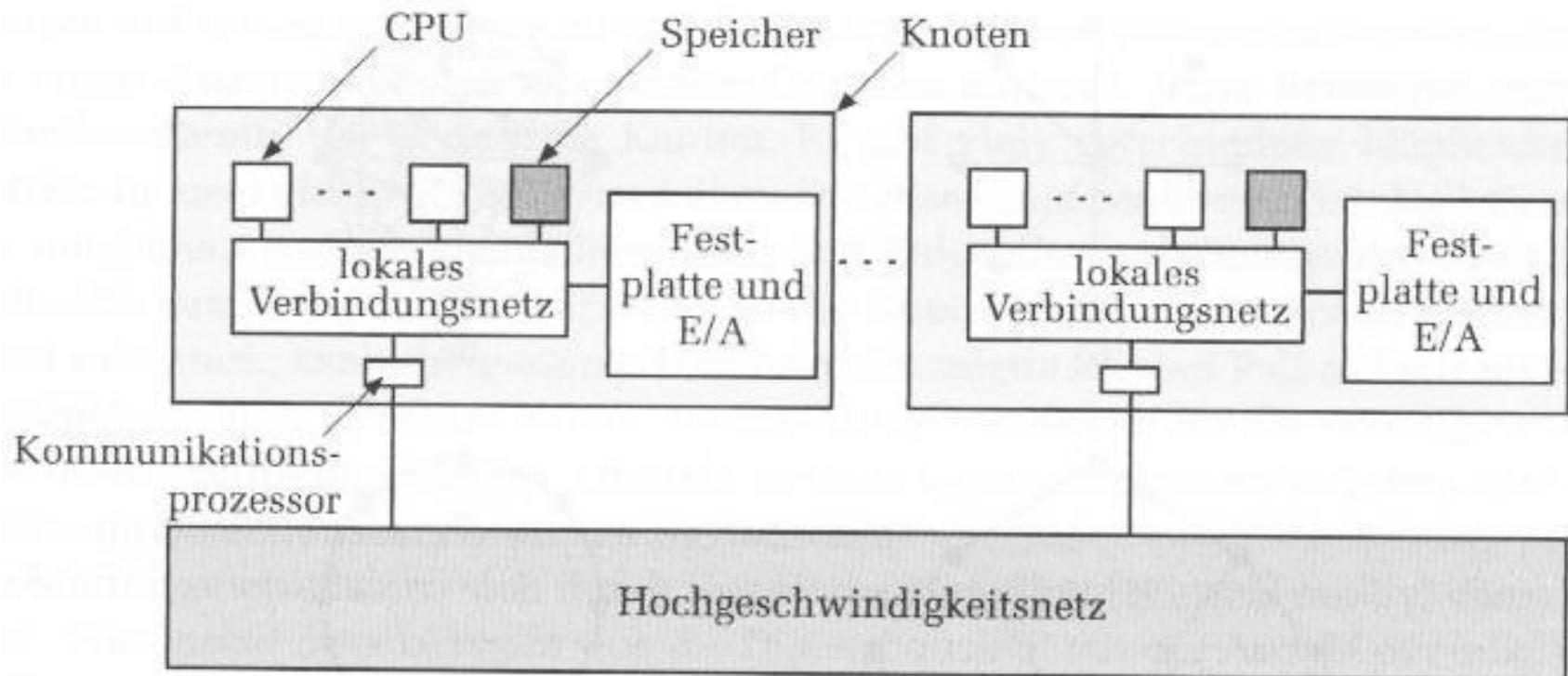


Bild-Quelle: Tanenbaum: Computerarchitektur. München: Pearson-Studium, 2006

#### 6.4.4.1 allgemeine Merkmale von NORMA (III)

NORMA behebt die Nachteile von Multiprozessorssystemen:

- keine Konkurrenz bei Zugriffen zum gemeinsamen Speicher über Bus oder Verbindungsnetzwerk
- höhere Skalierbarkeit (Prozessoranzahl bis über 10 000)

Aber: nur Zugriffe zum jeweils privaten Speicher möglich,  
daher Kommunikation im Gesamtsystem nur mittels **Nachrichtenaustausch (message passing)** über das Verbindungsnetzwerk,  
d.h. keine load/store-Operationen zu fremden Speichern!

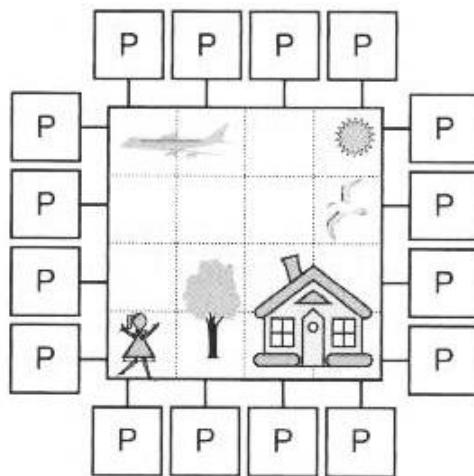
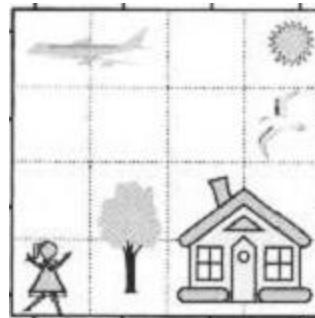
Zu NORMA gibt es viele verschiedene Formen, vor allem:

- massiv-parallele Prozessorsysteme MPP
- Cluster
- Grid

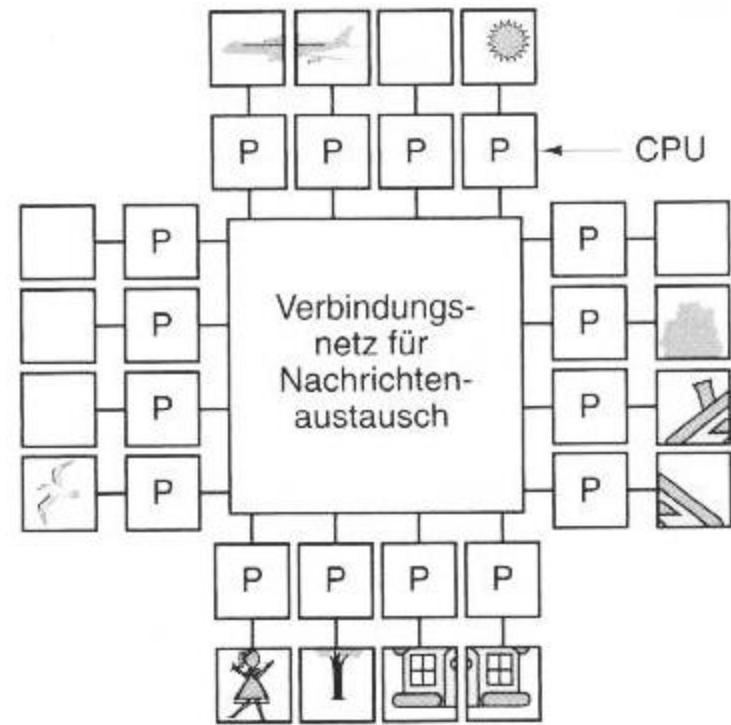
#### 6.4.4.1 allgemeine Merkmale von NORMA (IV)

Bild-Quelle:  
Tanenbaum/Goodman:  
Computerarchitektur.  
München: Pearson-  
Studium, 2001

Ein Bild ...



... in einem System mit  
gemeinsamem Speicher  
(Multiprozessorsystem)



... in einem System mit verteiltem  
Speicher (Mehrrechnersystem)

#### 6.4.4.2 Massiv-parallele Prozessorsysteme (MPP) (I)

Merkmale:

- i.allg. große Supercomputer mit Tausenden Knoten
- In den Knoten meist Standard-Prozessoren (Intel, AMD, Sun UltraSPARC, IBM PowerPC)
- Meist proprietäres Hochleistungsverbundungsnetzwerk zur Verbindung aller Knoten untereinander (mit kleiner Latenz und großer Bandbreite)
- Große E/A- bzw. Plattenspeicherkapazität wegen großer Datenmengen (Terabyte-Bereich)
- Spezielle Hard- und Software zur Sicherung von Fehlertoleranz (vgl. Ausfallwahrscheinlichkeit bei Tausenden CPUs!)

### 6.4.4.2 Massiv-parallele Prozessorsysteme (MPP) (II)

Anwendung:

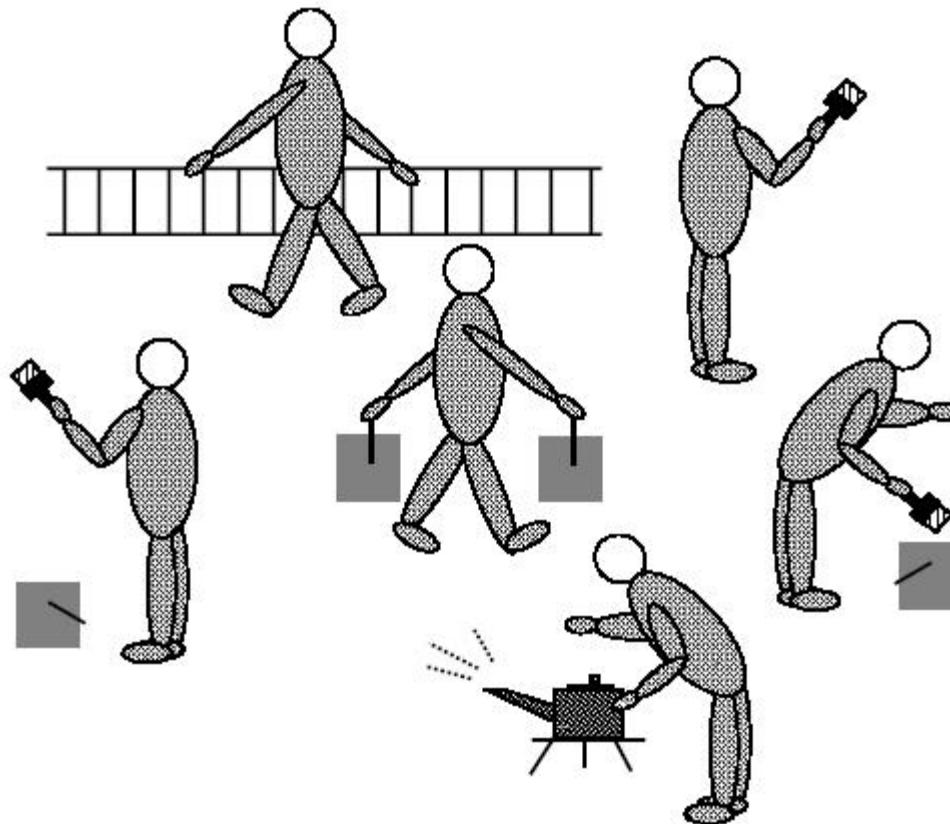
- Höhepunkt in den 80/90er Jahren
- oft Ablösung von teuren SIMD-Systemen
- Umfangreiche (meist wiss.-techn.) Berechnungen
- Transaktionsverarbeitung
- Data Warehousing/Data Mining
- Inzwischen auch im kommerziellen/industriellen Umfeld

Beispiel-Systeme:

- IBM Blue Gene-Reihe (mit PowerPC) seit 2001
- Cray Red Storm (mit AMD Opteron) 2004
- Cray T3E

### 6.4.4.2 Massiv-parallele Prozessorsysteme (MPP) (III)

„MPP in Action“:



Quelle: epcc (Edinburgh)

### 6.4.4.3 Cluster (I)

Allg. Merkmale:

- Multicomputer aus (meist via Ethernet) vernetzten PCs/Workstations (ggf. auch aus Servern oder sogar Supercomputern)
- NOW Network of Workstations bzw. COW Cluster of Workstations
- (COTS Commodity of the shelf: Cluster aus billigen Standard-PCs)
- Leistungsfähigkeit fast beliebig steigerbar, nur abhängig von
  - Leistung der einzelnen Knoten
  - Max. DÜ-Rate des verwendeten Netzwerks
  - Wartungs- und Administrationsaufwand
- [Pfister1998]: „A cluster is a type of parallel system that consists of interconnected whole computers and is used as a single, unified computing resource.“
  - Knoten = eigenständige Computer
  - Cluster verhält sich wie ein einheitlicher Uniprozessor  
(= Konzept der parallelen Transparenz: Single System Image)
- Cluster i.e.S.: wenn die Knoten **exklusiv** für den Cluster arbeiten

### 6.4.4.3 Cluster (II)

Vorteile:

- bei Knoten aus Standard-HW einfacher Ersatz möglich
- Finanziell günstiger als entspr. Großrechner
- Flexibilität und gute Erweiterbarkeit

Nachteile:

- Größerer Administrationsaufwand (und damit Personalaufwand)  
(Tools zur Visualisierung des Zustands von 1000 Knoten??)
- Größerer Aufwand zur Verteilung und Kontrolle der Anwendungen

Histor. Entwicklung der Cluster:

- 1983 VAX-11 von Digital Equipment, 1986 VAX 8978 (4/8 Knoten)
- 90er Jahre: UNIX-Workstation-Cluster + PVM/MPI-Middleware
- Ab Mitte 90er Jahre: PC-Cluster
- Heute oft Knoten als „Pizzabox“ bzw. Blades

### 6.4.4.3 Cluster (III)

Vernetzung im Cluster:

- bei wenigen Knoten: Standard-Netzwerktechnologien wie Fast-Ethernet, Gigabit-Ethernet
- bei vielen Knoten: spezielle Hochgeschwindigkeitsnetzwerke wie Myrinet, Infiniband, Scalable Coherent Interconnect SCI

Aufstellkonzepte von Clustern:

- „Glass House“, d.h. gesamtes Cluster steht in speziellem Raum
  - Einfachere Wartung
  - Besserer Schutz gegen Sabotage, aber empfindlicher bei Brand
  - Meist schnelleres Netzwerk
- Campus-Wide, d.h. Cluster über verschiedene Gebäude verteilt
  - NOW bzw. „Feierabend-Cluster“
  - Unempfindlicher bei Brand u.ä.
  - Meist kein Hochgeschwindigkeitsnetzwerk
  - Wartung schwieriger

#### 6.4.4.3 Cluster (IV)

Einsatzgebiete von Clustern:

1. Hochverfügbarkeit: **High Availability Cluster (HA-Cluster)**
    - Ziel: Große Ausfallsicherheit / hohe Verfügbarkeit durch Redundanz, USV u.a.
    - Verfügbarkeit  $V = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}}$
- MTBF
- $\frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}}$
- MTBF=Mean Time Between Failures, MTTR=Mean Time To Repair
- $V=100\% \rightarrow 24 \text{ Std./Tag} \times 7 \text{ Tage/Woche} \rightarrow 8760 \text{ Std./Jahr}$
  - $V= 99\% \rightarrow \text{max. } 3,7 \text{ Ausfalltage erlaubt!}$
  - $V= 99,9\% \rightarrow \text{max. } 9 \text{ Std. Ausfallzeit erlaubt!}$
  - Verhalten bei Knotenausfall:
    - Active/Passive-Cluster (Hot Standby): mind. 1 Knoten im Betrieb passiv (als Reserve zur Fkt.-übernahme „Fail-Over“)
    - Active/Active-Cluster: Alle Knoten im Betrieb aktiv, bei Ausfall übernehmen andere Knoten die Arbeit zusätzlich

#### 6.4.4.3 Cluster (V)

##### 2. Hochleistung: High Performance Cluster (HPC)

- Ziel: möglichst hohe Rechenleistung
- Gut erweiterbar
- Im Vgl. zu Großrechnern preisgünstiger und herstellerunabhängig
- Im Vergleich zu Großrechnern höherer Administrations- und Wartungsaufwand
- Typ. Anwendung: Forschung, Analyse, Simulation u.a. im naturwiss.-techn. Bereich, bei Banken, Automobilbau, Militär
- „Beowulf-Cluster“: oft verwendet für HPC mit Linux
- „Wolfpack“: oft verwendet für Windows-basierte HPC (Windows Compute Cluster Server)

#### 6.4.4.3 Cluster (VI)

### 3. Hoher Datendurchsatz: High Throughput Cluster (HTC)

- Ziel: Max. Datendurchsatz, d.h. max. Anzahl von Jobs/Zeit
- Ist eine Variante des HPC
- Für sehr viele, aber eher kleine Aufträge geeignet
- Typ. Anwendung: Web-Server / Suchmaschinen (Google), Mail-Server, ...
- Lastverbund (Lastverteiler ist oft ein Name-Server),
- Verteilung der vielen Aufträge an jeweils einen anderen Knoten

### 6.4.4.3 Cluster (VII)

#### 4. Skalierbare Cluster

- Kompromiss zwischen Hochleistung und Hochverfügbarkeit
- Einige oder alle Knoten sind redundant ausgelegt
- Alle Knoten erhalten Aufträge vom Lastverteiler
- Alle Knoten überwachen sich gegenseitig bzgl. Ausfall
- Typ. Anwendung: Server (Web-, Mail-, ...)

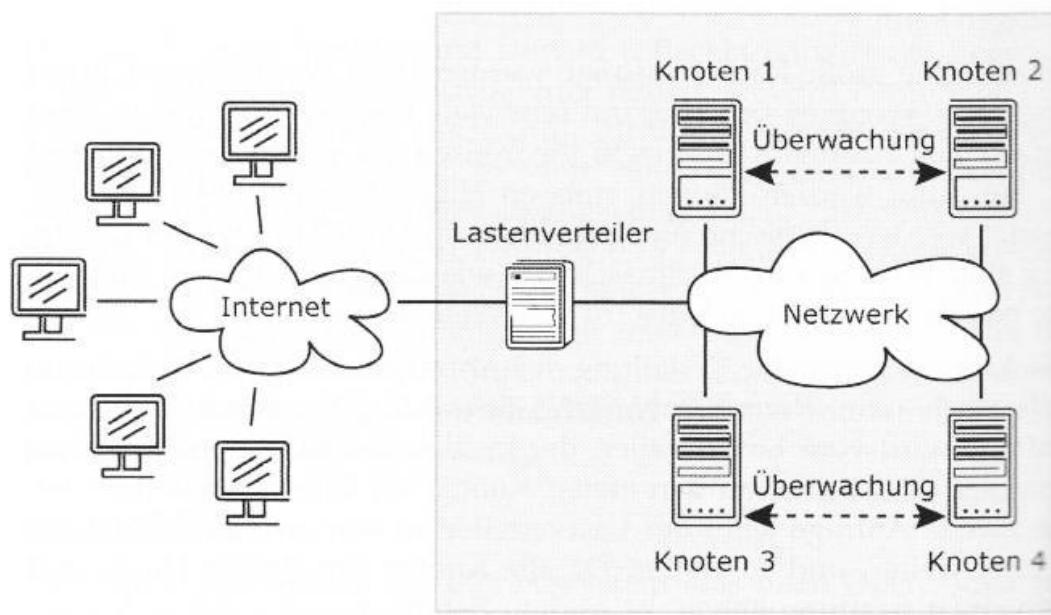


Bild-Quelle: Bengel:  
Masterkurs parallele und  
verteilte Systeme  
Vieweg/Teubner: 2008

#### 6.4.4.4 Grid Computing (I)

- Quasi Weiterentwicklung von Clustering in Richtung Ortsungebundenheit und institutionsübergreifender Nutzung
  - Umfang und Komplexität von Daten steigt
  - Bedarf an Rechenleistung und Speicherkapazität wächst (besonders in Wiss./Technik, Medizin)
  - Arbeitswelt wird immer standortübergreifender/verteilt
  - Oft kollaborative Arbeitsumgebungen für mehrere Partner nötig
- **Grid Computing** = Technik zur Integration und gemeinsamen, domänenübergreifenden, ortsunabhängigen Nutzung verteilter Ressourcen auf Basis bestehender Kommunikationsinfrastrukturen wie z.B. dem Internet
- Vergleich mit Stromnetz (einfacher Bezug/Abrechnung, Quelle egal)
- Nutzung im Rahmen virtueller Organisationen, z.B.:
  - Internat. Koordinierung nach Katastrophen
  - Internat. tätiges Industriekonsortium für Großprojekte

## 6.4.4.4 Grid Computing (II)

Anforderungen:

- Dynamische Zuteilung von Ressourcen an Benutzer des Grid
- Verbergen der Heterogenität der Ressourcen des Grids für seine Benutzer
- Stabilität, Verfügbarkeit
- Datensicherheit, Datenschutz über Grenzen einer Einrichtung hinaus
- Single Sign-On
- Abrechnung? Überwachung/Monitoring?

Lösungsansätze:

- Offene, standardisierte Protokolle und Schnittstellen
- Beachtung der Multilateralität  
(im Gegensatz zu Client-Server oder Peer-to-Peer)

## 6.4.4.4 Grid Computing (III)

Typisches Modell: 4 Grid-Ebenen

Ebene	Funktion
Application	Anwendungen, die verwaltete Ressourcen in kontrollierter Form gemeinsam nutzen
Collective	Ermittlung, Zuteilung, Überwachung und Steuerung von Ressourcengruppen
Resource	Sicherer, verwalteter Zugriff auf individuelle Ressourcen
Fabric	Physische Ressourcen: Computer, Speicher, Netzwerke, Sensoren, Programme, Daten, ggf. Spezial-HW/SW

#### 6.4.4.4 Grid Computing (IV)

Standardisierung

→ OGF Open Grid Forum: Open Grid Services Architecture (OGSA)

Realisierung zur Zeit i.allg. via Web-Services

→ Web Service Resource Framework (WSRF) legt Standards bzgl.  
Protokollen und Schnittstellen fest

Beispiele für akt. Middleware zum Grid-Computing:

- Globus Toolkit (Globus Alliance)
- gLITE (Leightweight Middleware for Grid Computing; CERN, EU)
- Unicore (Uniformes Interface für Computerressourcen; FZ Jülich)
- Sowie weitere SW-Pakete/Toolkits zur Verwaltung von Grids  
(z.B. GridSphere, Shibboleth, VOMS, SRB, GAT ...)

## 6.5. Mischform GPU

### 6.5.1 Überblick (I)

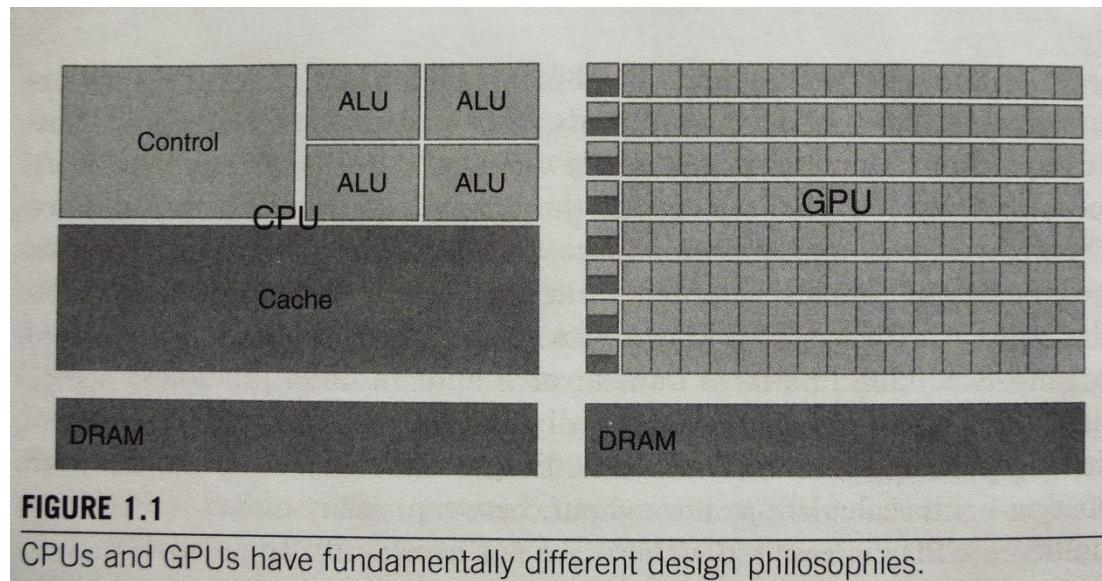
GPU = Graphics Processing Unit (oder [früher] einfach: „Grafikkarte“ -? -)

- früher primär als „graf. Coprozessor“ (Beschleuniger, accelerator)
- in Abgrenzung zu Multicores entwickelte sich hier eine Form der „Manycore-“ bzw. „Many-Thread-Architektur“, d.h.
  - parallele Ausführung sehr vieler Threads auf sehr vielen Cores
  - mit typ. „Rechen-Befehlen“ z.B. floating point Operationen
  - über vielen gleichartigen Daten (vgl. SIMD)
- Entwicklung getrieben u.a. durch Video Games Industrie

### 6.5.1 Überblick GPU (II)

- Vorteile im Vgl. zu Multicore:
  - einfachere Befehlssteuerung (vgl. SIMD),
  - daher mehr Platz on Chip für noch mehr parallele execution units,
  - höhere Bandbreite zum Speicher, geringere Latenz
- aber: Anwendung muss viele parallele Threads für die HW „liefern“

CPU-  
Architektur  
eher  
„Latenz-  
orientiert“



GPU-  
Architektur  
eher  
„Durchsatz-  
orientiert“

### 6.5.1 Überblick GPU (III)

- GPUs können heute nicht nur die typ. Grafikberechnungen ausführen, sondern auch allg. Berechnungen, das trifft auf viele wiss.-techn. (v.a. numerische) Berechnungen zu
- einige frühere Nachteile der GPU wurden inzw. ausgemerzt, z.B.
  - Unterstützung IEEE Std. f. Floating Point Operationen
  - Massen(markt)-Produkt
  - kleiner Formfaktor
- insbes. heute leichtere Programmierung wegen besserer Progr.-modelle (general purpose parallel programming interface), die an besondere Architektur der GPUs angepasst sind:
  - CUDA
  - OpenCL
- wegen breiterer Anwendungsbereiche (z.B. Medizin, Finanzwesen, ...) und verbesserter Programmiermöglichkeiten -> allg. Bezeichnung:  
**→ General-Purpose Graphics Processing Unit (GPGPU)**
- aktuelle HPC-Systeme: oft Kombination aus vielen CPUs und GPUs

## 6.5.2 Beispiel: NVIDIA-GPU

### Vergleich (Analogie) SIMD - GPU

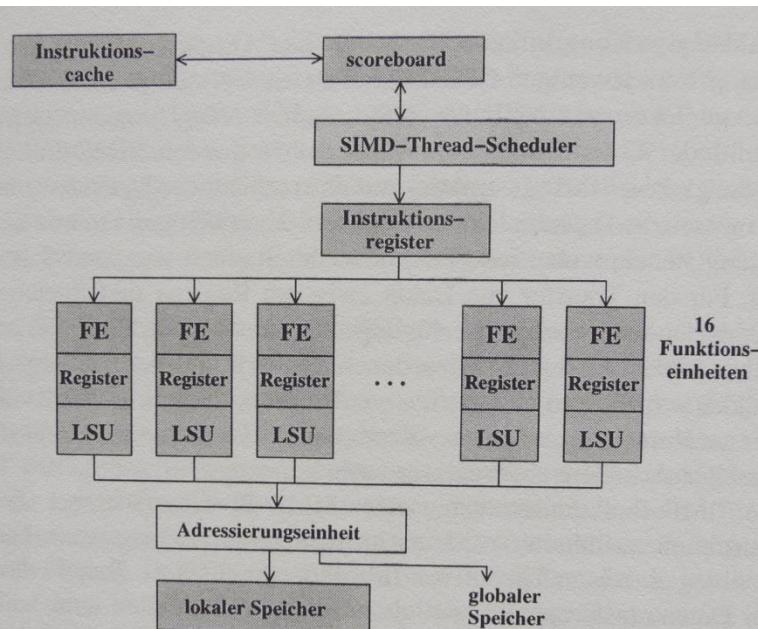
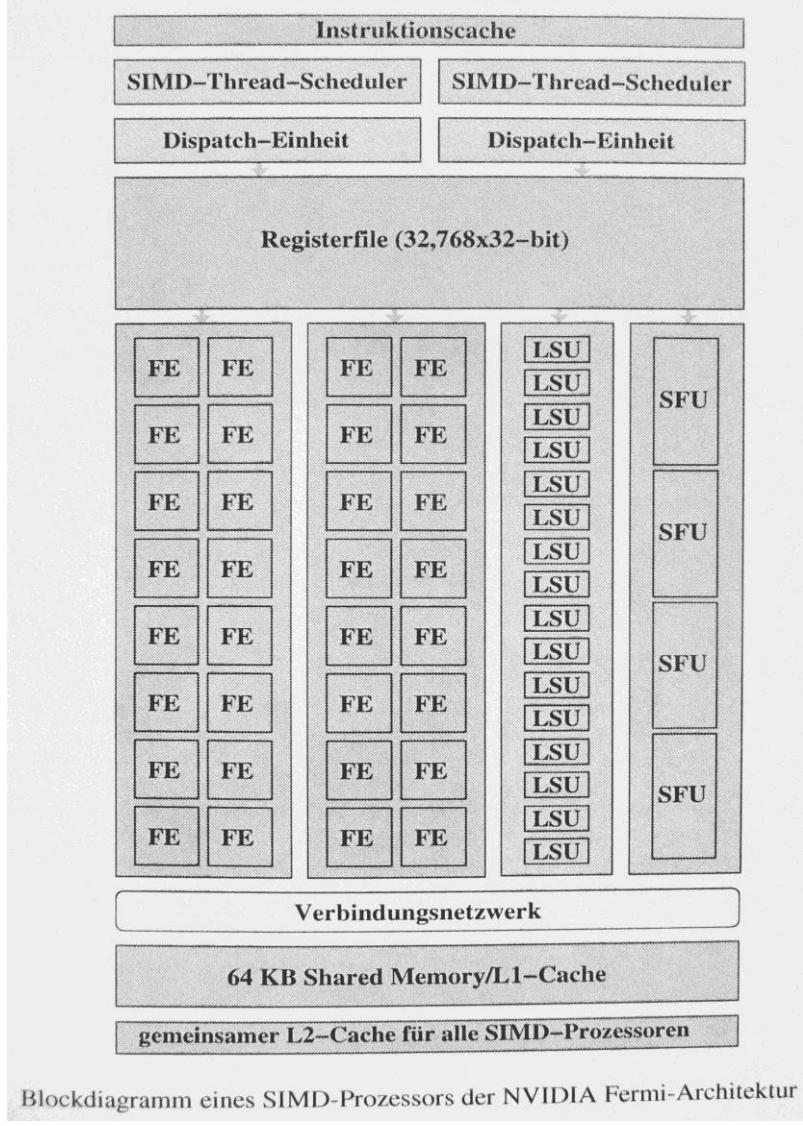


Abb. 7.1 Blockdiagramm eines SIMD-Prozessors nach [76] mit 16 Funktionseinheiten (FE), von denen jede eine separate Menge von Registern und eine eigene Transfereinheit (engl. Load-Store-Unit, LSU) hat



Blockdiagramm eines SIMD-Prozessors der NVIDIA Fermi-Architektur

LSU Load/Store Unit SFU Special Function Unit

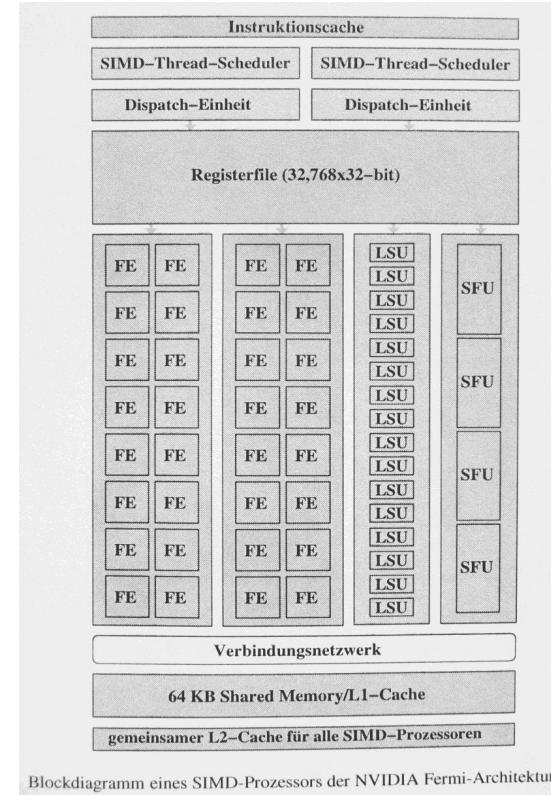
### 6.5.2 Beispiel: NVIDIA-GPU (II)

GPU intern:

→ viele multithreaded SIMD-Prozessoren  
(vgl.-bar als unabh. MIMD Prozessorkerne)

Bsp. NVIDIA GTX (*Fermi*)

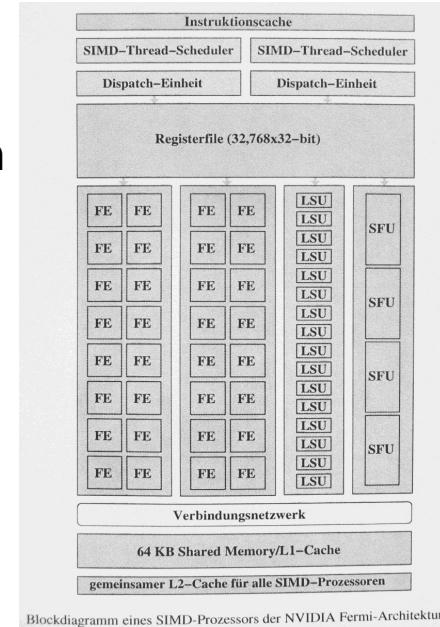
- jeder SIMD-Prozessor hat 16 physikalische FEs  
(FE = „*SIMD-Kern*“)
- jede FE besteht aus 1 int- und 1 floatingpoint-Einheit
- jeder FE sind 2048 32-bit-Register logisch zugeteilt
- Register werden auf Threads aufgeteilt,  
z.B. je 64 Reg. bei 32 Threads pro FE verfügbar  
Reg. werden den Threads bei Erzeug. dyn. zugeteilt
- bei Fermi: 2 SIMD-Thread-Scheduler, daher 2 x 16 FEs
  - + 16 Datentransfereinheiten zw. Reg. und Speicher (LSU)
  - + 4 Spezialeinheiten (SFU, für spez. Fkt. wie sin(), cos(), Wurzel, Reziproke)
- Speicherhierarchie (L1 je SIMD-Proz., L2 f. alle SIMD-Proz. einer GPU)



## 6.5.2 Beispiel: NVIDIA-GPU (III)

Eine Fermi-GPU enthält 7, 11, 14 oder 15 SIMD-Prozessoren

- Zweistufiges Thread-Scheduling:
  1. ein Thread-Block-Scheduler ordnet den SIMD-Prozessoren Blöcke aus mehreren Threads zu
  2. je SIMD-Prozessor ordnet ein SIMD-Thread-Scheduler den FEs einzelne SIMD-Threads zu



Bsp.:

NVIDIA Fermi GTX 480 hat Peak-Performance von ca. 1300 GFLOPs für Werte einfacher Genauigkeit,  
(512 GFLOPs für doppelte Genauigkeit: ca. 10-fach schneller als vgl.-bare CPU)

## 6.5.2 Beispiel: NVIDIA-GPU (IV)

neue energieeffizientere Architektur  
von NVIDIA: **Kepler**

Bsp.: GeForce GTX 680

- 8 SIMD Prozessoren  
(*streaming multiprocessors SMX*)
- Jeder SIMD-Prozessor SMX enthält: →
  - 4 SIMD-Thread-Scheduler
  - Registerfile
  - $2 \times 96 = 192$  FEs (SIMD-Kerne)
  - $2 \times 16 = 32$  LSUs
  - 16 SFUs
  - L1 und L2 Cache

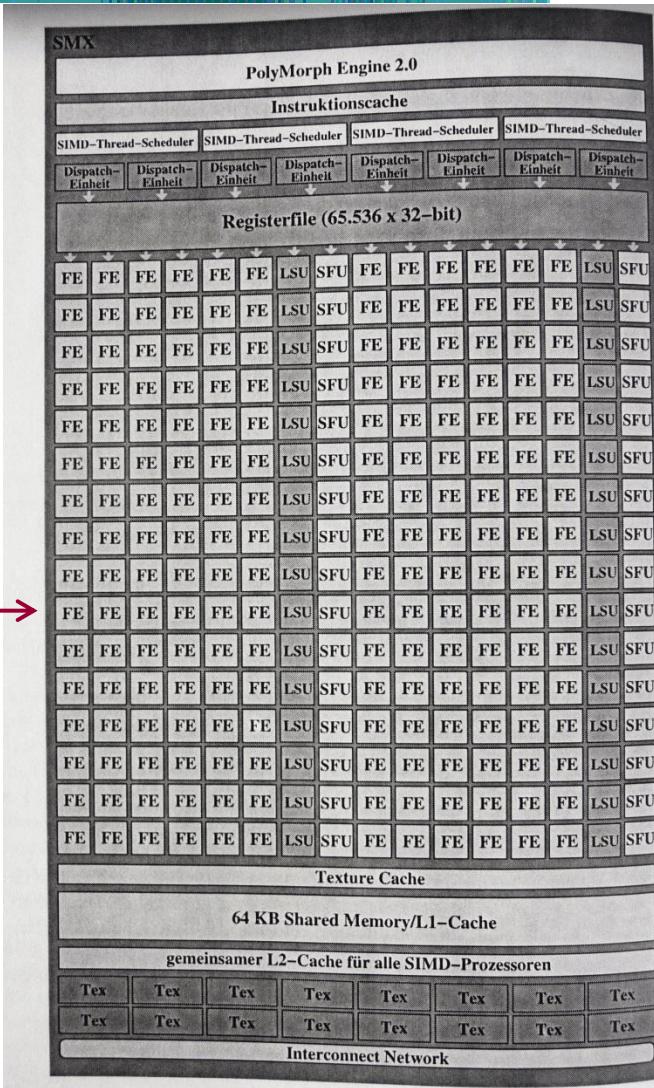


Abb. 7.3 Blockdiagramm eines SIMD-Prozessors der Kepler-Architektur

Bild-Quelle: Rauber/Rünger, 2013

## 6.5.2 Beispiel: NVIDIA-GPU (V)

Bsp.: Kepler  
GeForce GTX 680

Zusammenfassung von  
je 2 SMX-SIMD-Prozessoren  
zu einem *Graphics Processing Cluster (GPC)*, d.h.  
jede GPU besteht aus 4 GPCs  
= 8 SMX  
=  $8 \times 192 = 1536$  SIMD-Kerne!

Zusätzlich je GPC noch  
eine *Raster Engine*  
f. schnelle Grafikverarbeitung

Peak Performance:  
bei einfacher Genauigkeit  
ca. 3000 GFLOPs!

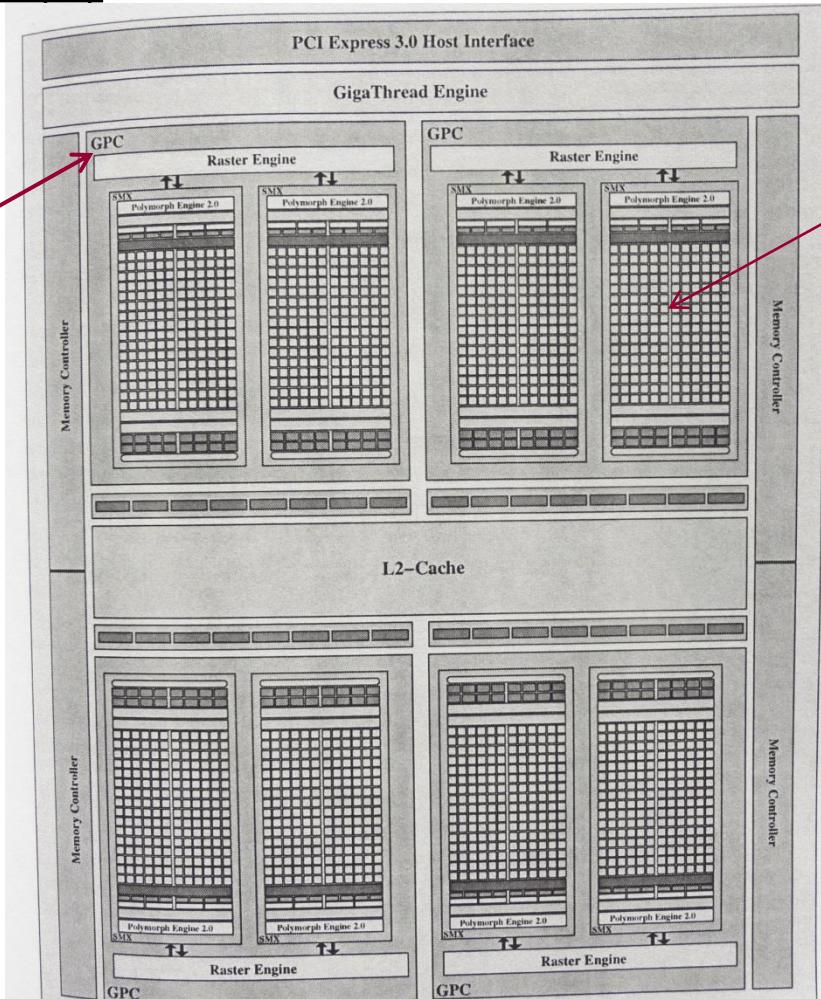


Abb. 7.4 Blockdiagramm der GeForce GTX 680 mit Kepler-Architektur

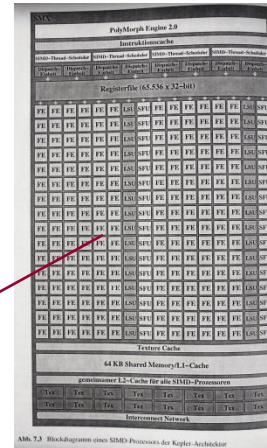


Abb. 7.5 Blockdiagramm eines SIMD-Prozessors der Kepler-Architektur

Bild-Quelle:  
Rauber/Rünger, 2013

## 6.5.2 Beispiel: NVIDIA-GPU (VI)

### Leistungsvergleich NVIDIA-GPUs

GPU	GTX 285 (Tesla)	GTX 480 (Fermi)	GTX 580 (Fermi)	GTX 680 (Kepler)	
Transistoren	$1,40 \cdot 10^9$	$3,2 \cdot 10^9$	$3,0 \cdot 10^9$	$3,54 \cdot 10^9$	
SIMD-Prozessoren	30	15	16	8	
SIMD-Kerne pro SIMD-Prozessor	8	32	32	192	↑
SIMD-Kerne insgesamt	240	480	512	1536	↑
L2-Cache	/	768 KB	768 KB	512 KB	
Performance	1063 GF	1344 GF	1581 GF	3090 GF	↑
Bandbreite Speicher	159 GB/sec	177 GB/sec	192 GB/sec	192 GB/sec	
Taktrate Speicher	2484 MHz	3696 MHz	4008 MHz	6008 MHz	↗
Verbrauch	204 W	250 W	244 W	195 W	→

## 6.6. Verbindungsnetzwerke von Parallelrechnern

### 6.6.1 Überblick (I)

#### Verbindungsnetzwerk (interconnection network)

- Aufgabe: Transport einer Information (z.B. Ergebnisse, oder zur Synchronisation) von einem Prozessor zu einem anderen Prozessor oder zum Speicher, möglichst korrekt und in möglichst kurzer Zeit
- Anwendungsbeispiele bei Parallelrechnern:
  - SIMD: zur Kopplung der PEs
  - Datenflussrechner: zur Verbindung zwischen Speicher und PEs
  - MIMD: Kommunikation zwischen Prozessoren bzw. zum Speicher
- hat großen Einfluss auf das Leistungsverhalten des gesamten Rechnersystems, z.B. hins. Skalierung

## 6.6. Verbindungsnetzwerke von Parallelrechnern

### 6.6.1 Überblick (II)

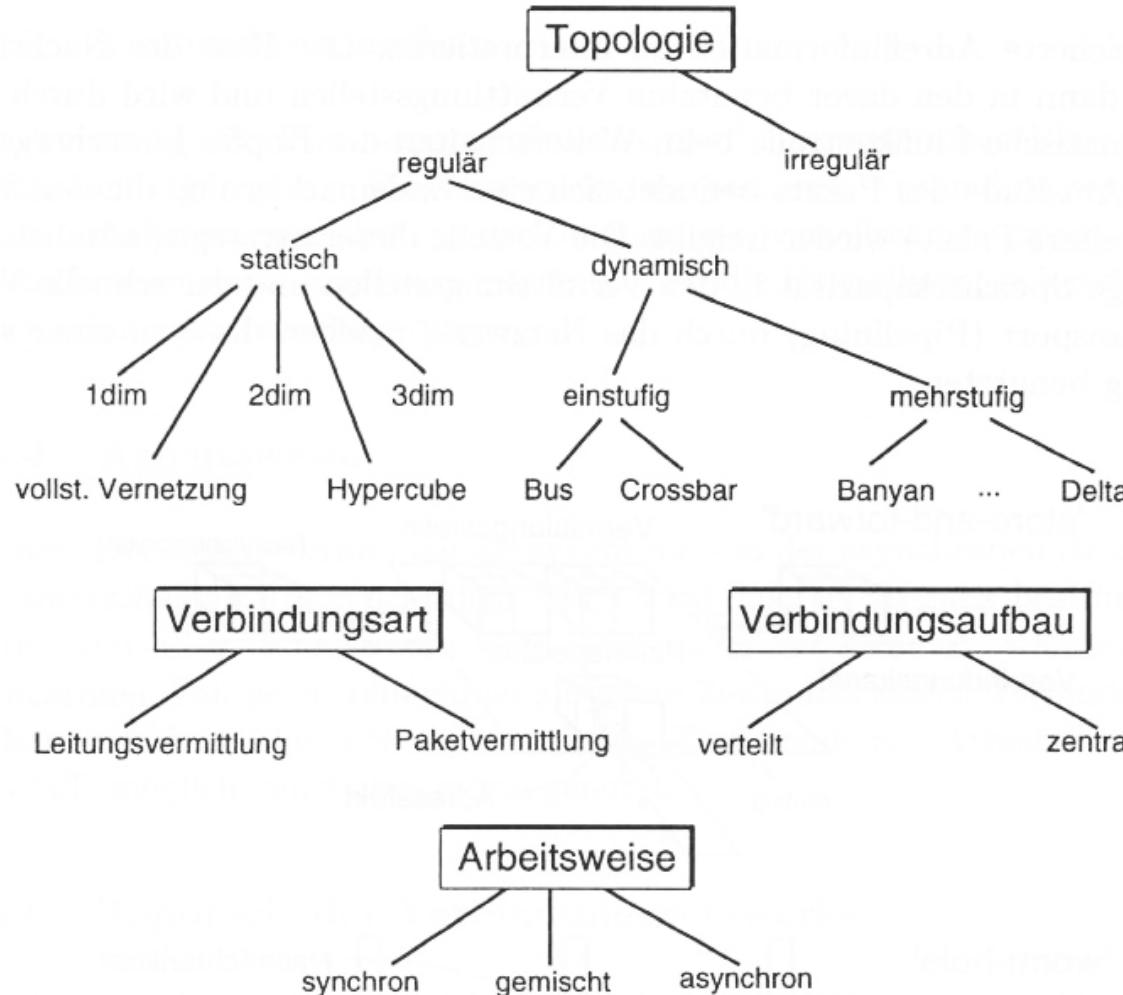


Bild-Quelle: Waldschmidt:  
Parallelrechner. Teubner, 1995

## 6.6.2 Topologie von Verbindungsnetzwerken

### 6.6.2.1 Begriffe (I):

- **Topologie** = Form bzw. geometrische/räumliche Struktur der Verknüpfung/Verschaltung der Elemente, auch Art der Anordnung von Schaltern und Leitungen;  
i.allg. als Graph beschrieben,
- **Grad (degree)** bzw. **Verzweigungsgrad eines Knotens**  
= Anzahl der Verbindungen/Links (ein-/auslaufende Kanten)
- **Grad des Netzwerkes** = max. Grad eines Knotens im Netz  
→ Je höher der Verzweigungsgrad, desto höher die Fehlertoleranz (d.h. bei Ausfall einer Verbindung: Einrichtung einer Umleitung)

### 6.6.2.1 Begriffe (II):

- **Durchmesser (diameter)** = maximale Entfernung zwischen zwei Knoten, [Anzahl der Kanten, „hops“]  
→ Wert für die Verzögerung der Übertragung im ungünstigsten Fall
- **Dimension** = Anzahl der möglichen Wege von Quelle zum Ziel
  - 0-dimensional: keine Wahlmöglichkeit für Weg, d.h. nur 1 Weg
  - 1-dimensional: eine Entscheidungsdimension für Wegewahl
  - 2-dimensional: zwei „Entscheidungsachsen“ zur Wegewahl, usw.
- **Bisektionsbandbreite** = minimale Anzahl von Kanten, die aus einem Graphen entfernt werden müssen, um das Netz in zwei gleichgroße Teilnetze zu zerlegen  
→ Wert für Sättigung des Netzes bei Hochlast, d.h. für Belastbarkeit bei gleichzeitiger Übertragung von Informationen

### 6.6.2.1 Begriffe (III):

- **Knoten- und Kanten-Konnektivität** = minimale Anzahl von Knoten/Kanten, die ausfallen müssten, um das Netz in zwei nicht mehr verbundene Teile zu zerlegen, es also zu unterbrechen
- **Symmetrie:** falls Sicht des Netzwerkes von jedem Knoten aus gleich → *Anforderungen an ein Verbindungsnetzwerk für Parallelrechner:*
  - Kleiner Durchmesser für kurze Entfernung im Netz
  - Kleiner Grad jedes Knotens zur Reduzierung des HW-Aufwandes
  - Hohe Bisektionsbandbreite für hohen Durchsatz
  - Hohe Konnektivität für hohe Zuverlässigkeit
  - Einfache Erweiterbarkeit (auf mehr Prozessoren) für gute Skalierung

### 6.6.2.2 Direkte (statische) Verbindungsnetzwerke (I)

Merkmale:

- regelmäßige Netzwerkstruktur (= regelmäßiger Graph)
- *direkte* Verbindung (Link) zwischen den Knoten (Prozessoren) durch eine physikalische Leitung
- feste Punkt-zu-Punkt-Verbindung, kann nach Aufbau nicht mehr geändert werden
- Netz hat nur Verbindungsfunction, Vermittlung erfolgt indirekt in den PEs selbst
- Anzahl der Verbindungen für einen Knoten variiert
- Anwendung meist für PR mit verteiltem Speicher, wobei Knoten = Prozessor + zugehöriger Speicher

### 6.6.2.2 Direkte (statische) Verbindungsnetzwerke (II)

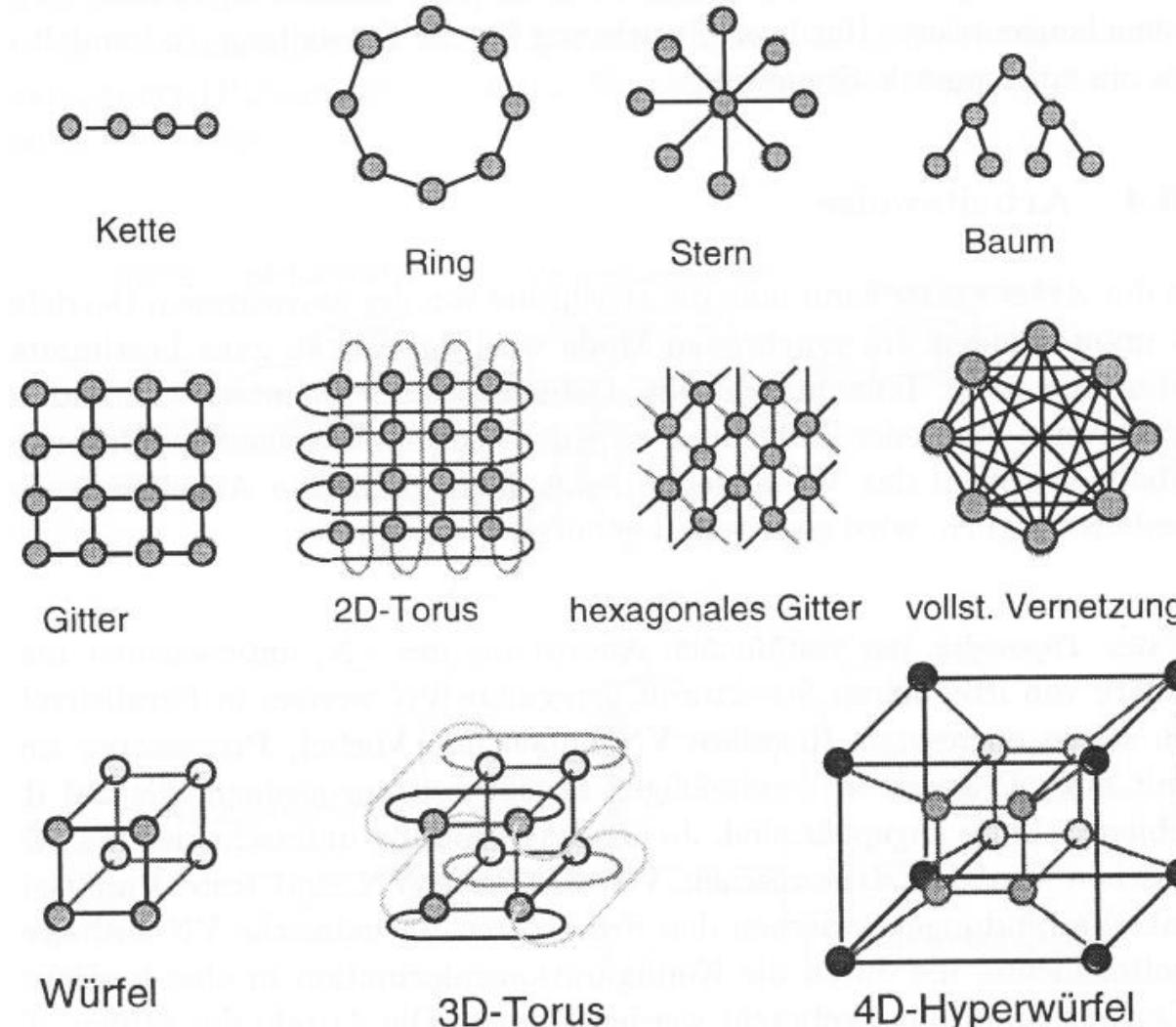


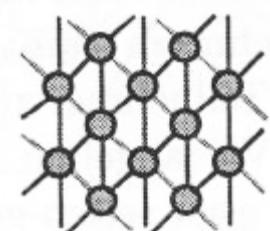
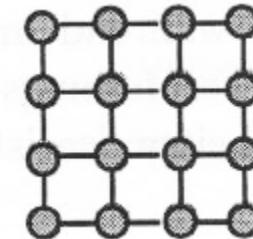
Bild-Quelle: Waldschmidt:  
Parallelrechner. Teubner, 1995

### 6.6.2.2 Direkte (statische) Verbindungsnetzwerke (III)

Häufig angewendete Formen:

- **Gitter**

- (auch nearest neighbor mesh bzw. grid)
- oft für zweidimensionale Probleme  
(2-dimensionales Gitter)
- dabei nur max. 4 Verbindungskanäle pro Knoten (=Grad) bei beliebiger Gittergröße
- Bsp.: Transputer (Prozessor mit 4 Links, leider „ausgestorben“)  
Intel Paragon (ebenfalls 2-dimensionales Gitter)
- Allg.: d-dimensionales Gitter: Grad eines Knotens =  $2d$
- Hexagonales Gitter:  
entspricht einer 2D-Darstellung eines 3D-Gitters  
Anwendung z.B. beim systolischen Array  
(siehe SIMD/ Feldrechner) und bei Signal-  
prozessoren, vor allem bei Problemen  
im 3-dimensionalen Raum



### 6.6.2.2 Direkte (statische) Verbindungsnetzwerke (IV)

„Historisches Bsp.“: Transputer (**Transfer + Computer**)

- HW-Unterstützung für Kommunikation *im* Prozessor: 4 Links/CPU
- Typ. Vertreter: T800 von INMOS
- Transputer-Cluster-Systeme von Parsytec (z.B. 16 x T800 pro Knoten)
- Spezielles BS (HELIOS), spez. Progr.-sprache (OCCAM)

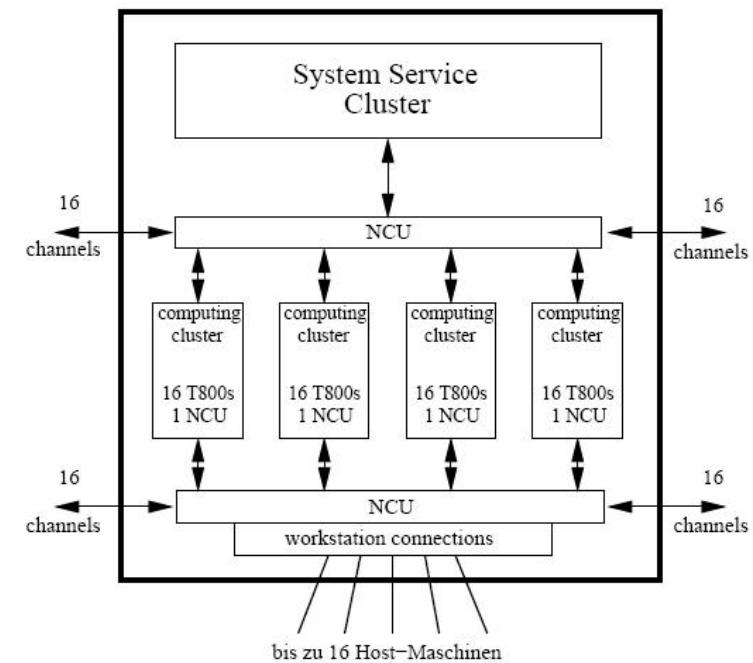
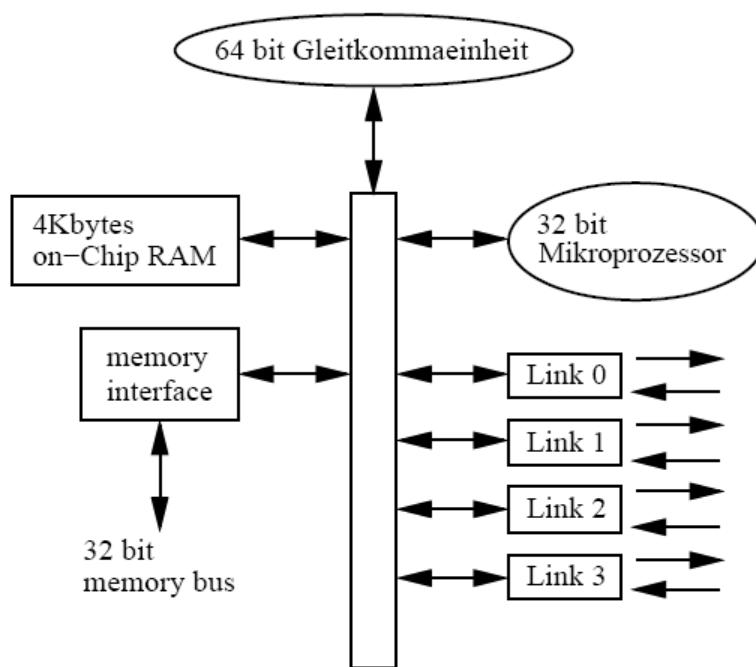
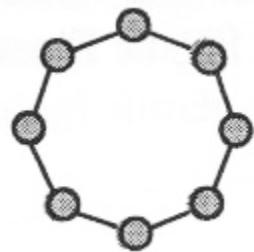


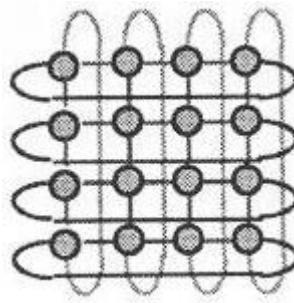
Bild-Quelle: Petri: VO Paralleles Rechnen Uni Potsdam WS2003/04

### 6.6.2.2 Direkte (statische) Verbindungsnetzwerke (V)

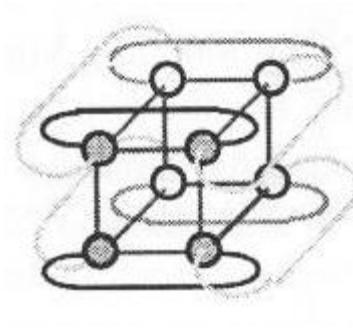
- **Torus**
  - d-dimensionaler Torus = Variante des d-dimensionalen Gitters durch zusätzl. Verbindung der Randknoten
  - Grad für alle Knoten =  $2d$
  - Spezialfall: Ring = 1D-Torus



1D-Torus



2D-Torus



3D-Torus

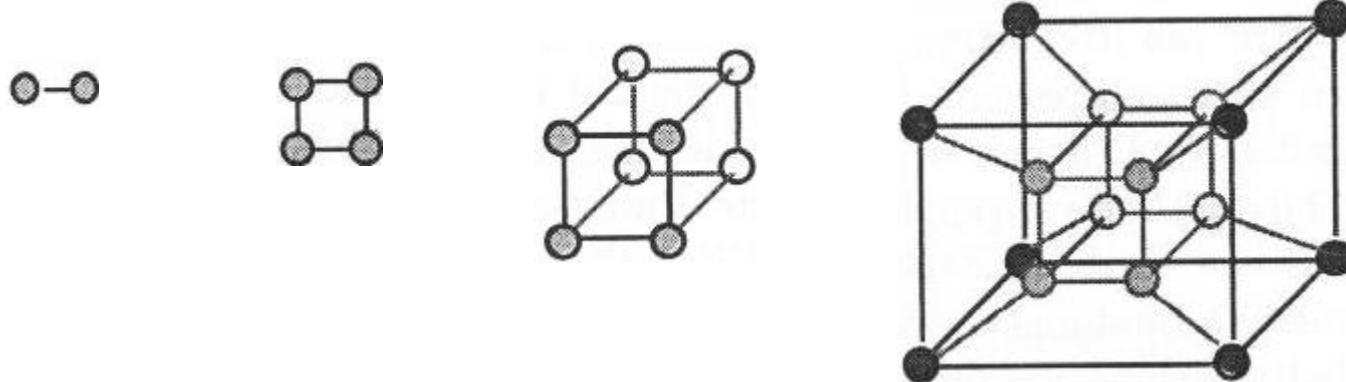
- Anwendung z.B. bei Cray T3D, T3E;  
Ring z.B. bei KSR-1 und Convex SPP2000 (4 parallele Ringe),  
3D-Torus bei IBM Blue/GeneL und Red Storm

### 6.6.2.2 Direkte (statische) Verbindungsnetzwerke (VI)

- Würfel (Cube), Hyperwürfel (Hypercube)

- Allg.: k-dimensionaler Würfel hat  $N=2^k$  Knoten

1D-Würfel    2D-Würfel    3D-Würfel    4D-Würfel (4D-Hypercube)

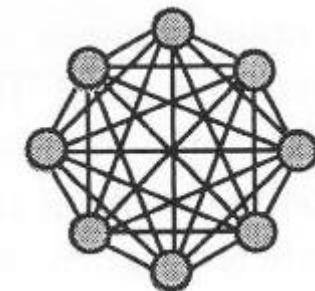


- Durchmesser eines k-dimens. Würfels = k (d.h. zwischen zwei beliebigen Knoten gibt es einen Pfad aus höchstens k Kanten)
- Grad eines k-dimens. Würfels = k (Anz. Nachbarknoten)
- wegen Abhängigkeit von der Dimension schlecht skalierbar
- Anwendung z.B. Intel iPSC/860, SGI Origin 2000

## 6.6.2.2 Direkte (statische) Verbindungsnetzwerke (VII)

- **Vollständige Vernetzung (completely interconnected)**

- Direkte Verbindung aller Knoten untereinander mit je 1 Verbindung, also Durchmesser = 1
- verbindungsreichste Topologie:  $\text{Grad} = N-1$  (Kanäle je Knoten)
- Hoher Aufwand schon bei kleinen Systemen
- daher bei PR kaum verwendet, (obwohl idealer Durchmesser!) stattdessen „elektronische Simulation“ der vollständigen Vernetzung durch Switch (Stern)



## 6.6.2.2 Direkte (statische) Verbindungsnetzwerke (VIII)

N = Anzahl der Knoten im Verbindungsnetzwerk

Topologie	Grad	Durchmesser	Anzahl Verbindungen	Skalierbar	Symmetrisch
1D-Gitter (Kette)	2	$N - 1$	$N - 1$	ja	nein
1D-Torus (Ring)	2	$\frac{1}{2} \cdot (N - 1)$	$N$	ja	ja
2D-Gitter	4	$2 \cdot (\sqrt{N} - 1)$	$2N - 2 \cdot \sqrt{N}$	ja	nein
2D-Torus	4	$\sqrt{N} - 1$	$2N$	ja	ja
3D-Gitter	6	$3 \cdot (\sqrt[3]{N} - 1)$	$3N - 3 \cdot \sqrt[3]{N}$	ja	nein
3D-Torus	6	$\frac{3}{2} \cdot (\sqrt[3]{N} - 1)$	$3N$	ja	ja
Hypercube	$\log_2 N$	$\log_2 N$	$N \cdot \log_2(N/2)$ ?	nein	ja
binärer Baum	3	$2 \cdot (\log_2 N - 1)$	$N - 1$	ja	ja
vollst. Vernetzung	$N - 1$	1	$\frac{N \cdot (N - 1)}{2}$	nein	ja

? richtig:  $\frac{N(N-1)}{2} * \log_2 N$

Bild-Quelle: Waldschmidt: Parallelrechner. Teubner, 1995;  
aktualisiert durch VO Uni Mannheim WS 2003/04

### 6.6.2.3 Indirekte (dynamische) Verbindungsnetzwerke (I)

Merkmale:

- regelmäßige Netzwerkstruktur (= regelmäßiger Graph)
- Verbindung zwischen den Knoten (Prozessoren/Speicher) **indirekt** durch mehrere Leitungen und dazwischen liegende **Schalter (bzw. aktive Koppelemente)**
- also *keine* festen Punkt-zu-Punkt-Verbindungen
- Schalter können je nach Konfiguration unterschiedliche Leitungsanordnung bewirken, damit übt das Netz auch die Vermittlungsfunktion aus
- Anwendung sowohl
  - für PR mit verteiltem Speicher (Verbindung zwischen Prozessoren)
  - als auch für PR mit gemeinsamem Speicher (Verbindung zwischen Prozessoren und Speicher)

### 6.6.2.3 Indirekte (dynamische) Verbindungsnetzwerke (II)

Typische Unterscheidung indirekter Netze:

- Anzahl der Stufen zwischen Eingang und Ausgang
  - Einstufige Netze
  - Mehrstufige Netze

Weitere Eigenschaften:

- Anzahl der Wege zu den Ausgängen:
  - Einpfadnetze
  - Mehrpfadnetze
- Anzahl der ausführbaren Permutationen bgl. der Wege:
  - blockierend
  - blockierungsfrei

### 6.6.2.3 Indirekte (dynamische) Verbindungsnetzwerke (III)

Allgemeines Modell eines indirekten Netwerkes

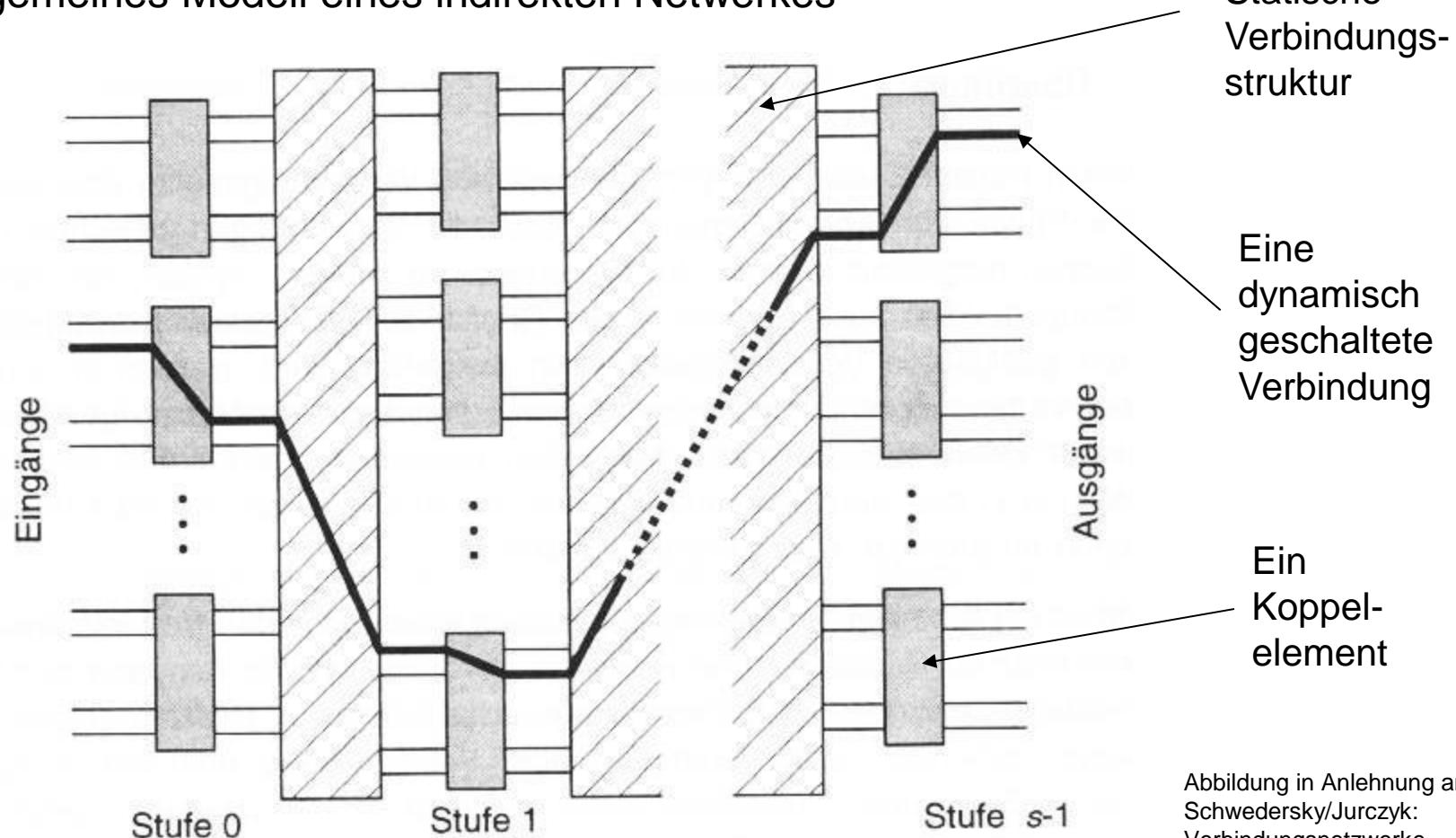
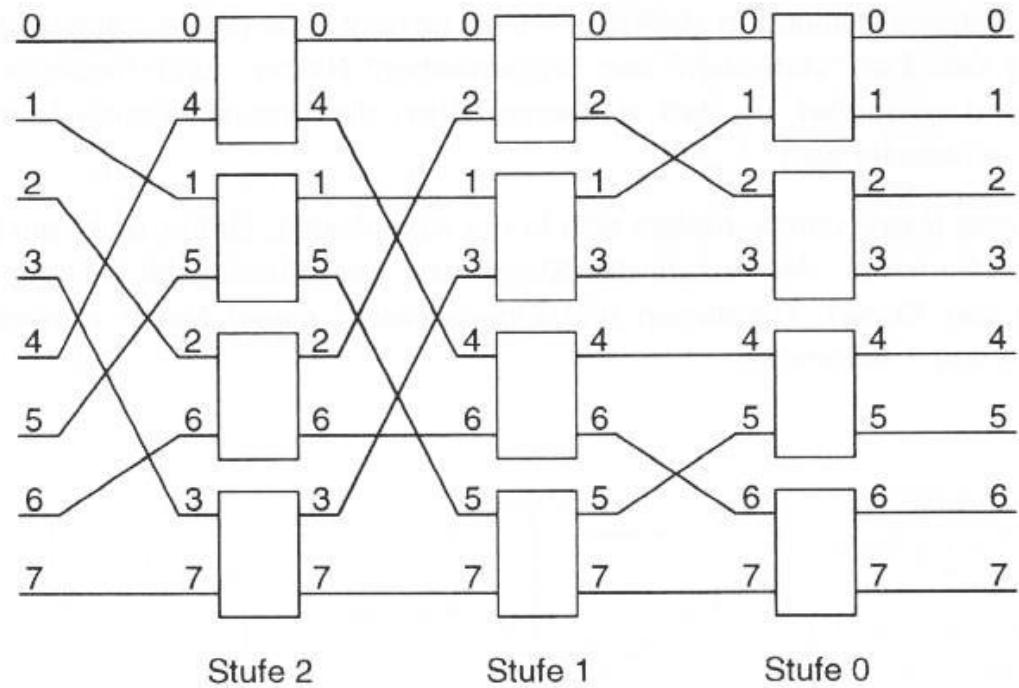
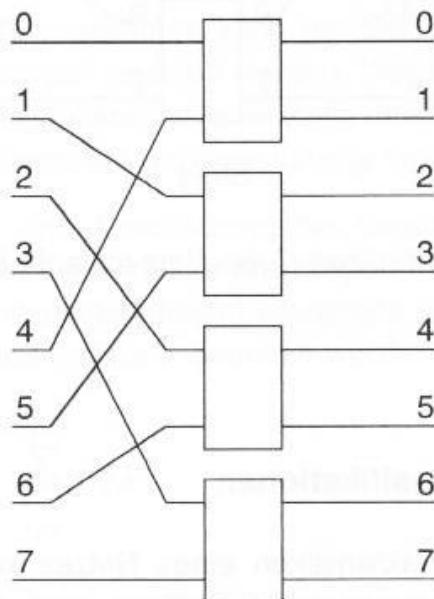


Abbildung in Anlehnung an:  
Schwidersky/Jurczyk:  
Verbindungsnetzwerke.  
Stuttgart: BG Teubner, 1996

### 6.6.2.3 Indirekte (dynamische) Verbindungsnetzwerke (IV)

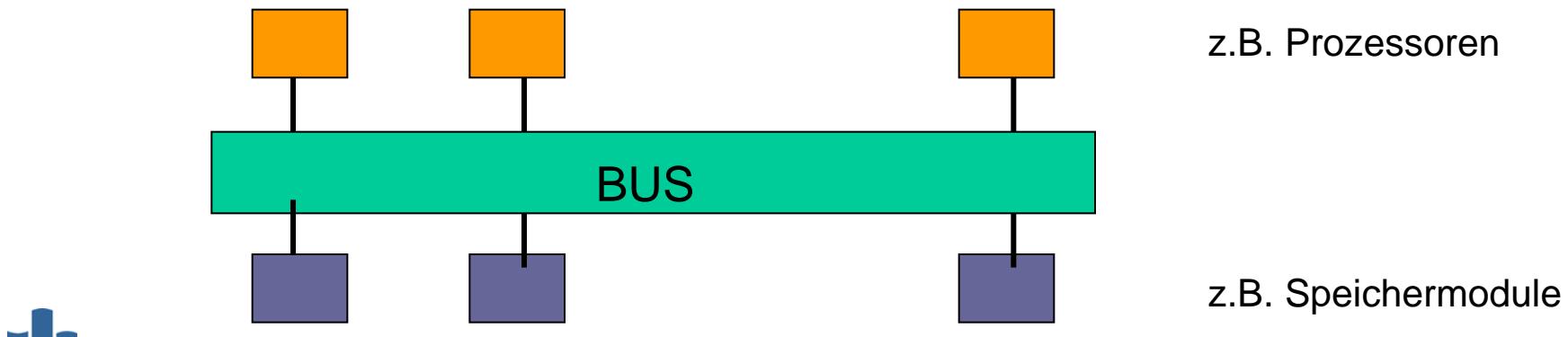
Beispiele für ein einstufiges und ein mehrstufiges indirektes Netzwerk



### 6.6.2.3 Indirekte (dynamische) Verbindungsnetzwerke (V)

Typische einstufige indirekte Verbindungsnetze:

- **Bus**
  - „Leitungsbündel“
  - Teilnehmer haben meist nur 1 Anschluss an den Bus
  - zu einem Zeitpunkt ist immer nur 1 Datentransport möglich
  - bei gleichzeitigem Transportwunsch mehrerer Teilnehmer ist Koordinierung des Buszugriffs nötig (Arbitrierung)
  - wegen begrenzter Bandbreite und schlechter Skalierung nur für wenige Teilnehmer geeignet
  - Anwendung in PR: Speicherbus, I/O-Bus, Peripheriebus



### 6.6.2.3 Indirekte (dynamische) Verbindungsnetzwerke (VI)

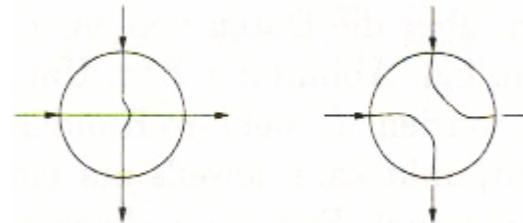
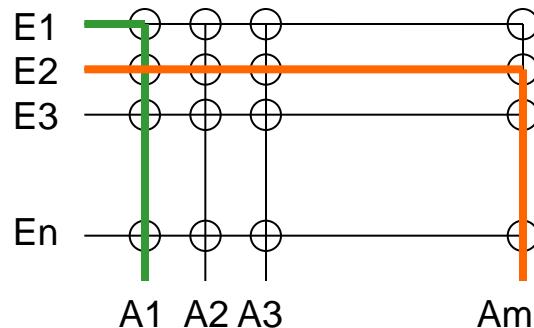
Typische einstufige indirekte Verbindungsnetze:

- **n x m Crossbar (Kreuzschienenverteiler, Koppelfeld) (I)**
  - das universellste dynam. Verbindungsnetz
  - verbindet in 1 Stufe jeweils beliebige Paare von Ein-/Ausgängen
  - Verteiler besteht aus horizontalen und vertikalen Busleitungen
  - an jedem Kreuzungspunkt (Koppelpunkt) ist ein Schalter (switch)
  - daher Größe des Netzes technisch begrenzt (Pin-Anzahl!!)
  - n x m Crossbar: n Eingänge, m Ausgänge, n x m Schalter
  - daher n x m gleichzeitige Verbindungen möglich, jede beliebige Permutation ist möglich; daher blockierungsfrei

Bsp.-  
Zuordnung:

$E_i$  = CPUs

$A_j$  = Speicher



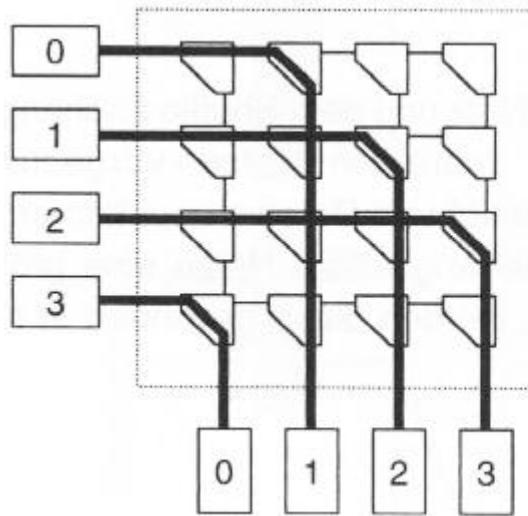
mögliche Schalterstellungen am  
Kreuzungspunkt (cross point)

Bild-Quelle: Rauber/Rünger:  
Parallele und verteilte  
Programmierung.  
Berlin: Springer, 2000  
(angepasst)

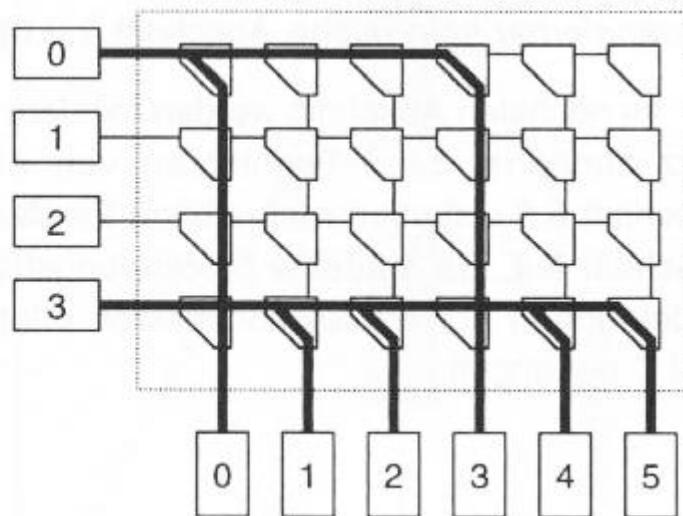
### 6.6.2.3 Indirekte (dynamische) Verbindungsnetzwerke (VII)

Typische einstufige indirekte Verbindungsnetze:

- **n x m Crossbar (Kreuzschienenverteiler, Koppelfeld) (II)**
  - Schalter kann ggf. auch Multicast-Verbindungen herstellen



Bsp. 4x4-Crossbar mit folg. Verbindungen:  
E0-A1, E1-A2, E2-A3, E3-A0 (Permutation)



Bsp. 4x6-Crossbar mit folg. Multicast-  
Verbindungen: E0 – A0 und A3, sowie  
E3 – A1, A2, A4, A5

## 6.6.2.3 Indirekte (dynamische) Verbindungsnetzwerke (VIII)

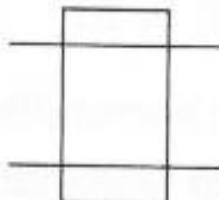
Typische einstufige indirekte Verbindungsnetze:

- **n x m Crossbar (Kreuzschienenverteiler, Koppelfeld) (III)**
  - Anwendung bei PR:
    - Verbindungsnetz zwischen Prozessoren und Speicher  
(bei physikalisch gemeinsamem Speicher: UMA),  
Bsp.: Multicore-Prozessoren AMD Opteron
    - Verbindungsnetz zwischen verteilten Prozessoren/Knoten  
untereinander (NUMA/COMA, NORMA)  
Bsp.: IBM RS600/SP, Sun Fire E25K (18 x 18 Crossbar)  
Bsp. für rel. großen Crossbar: Fujitsu VPP500  
(Vektorrechner) mit 224 x 224 Crossbar
    - als Grundbaustein für mehrstufige Netzwerke
  - Größe eines Crossbars leider techn. begrenzt  
(Bsp.  $1000 \times 1000 = 1.000\,000$  Koppelpunkte konstruktiv unmöglich)  
daher andere Lösung nötig ...

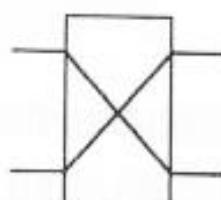
### 6.6.2.3 Indirekte (dynamische) Verbindungsnetzwerke (IX)

Typische einstufige indirekte Verbindungsnetze:

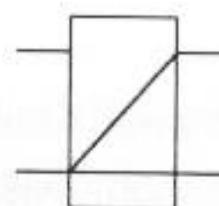
- Einstufiges „Shuffle“ (Shuffle-Exchange-Netz) (I)
  - Grundbaustein ist hier ein 2x2-Schalter mit 2 Schaltfunktionen:
    - *straight/forward*: beide Eingänge werden durchgeschaltet
    - *exchange*: beide Leitungen werden gekreuzt geschaltet
    - mitunter erweitert um zwei Broadcast-Funktionen:  
*lower broadcast* und *upper broadcast*



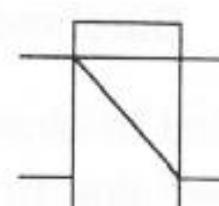
Straight



Exchange



Lower  
Broadcast

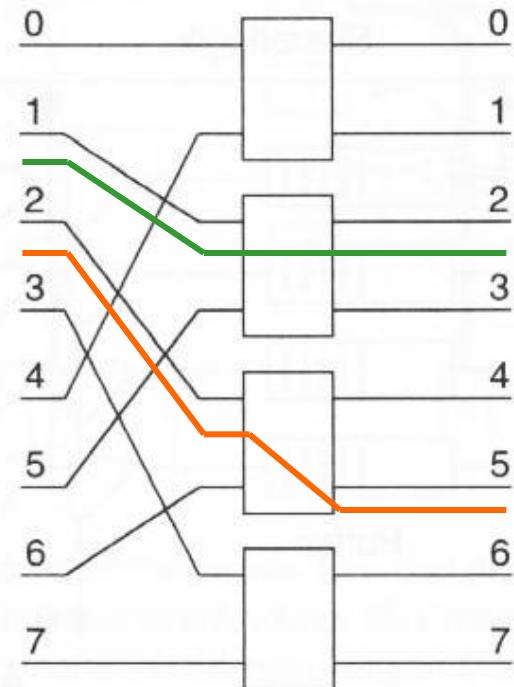
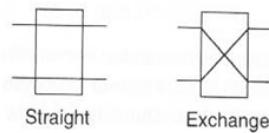


Upper  
Broadcast

### 6.6.2.3 Indirekte (dynamische) Verbindungsnetzwerke (X)

Typische einstufige indirekte Verbindungsnetze:

- Einstufiges „Shuffle“ (Shuffle-Exchange-Netz) (II)
  - Für beliebige Verbindungen von Knoten sind meist mehrere Durchläufe durch das Netz nötig
  - Bsp.-Netz mit 8 Knoten:  
Weg von K1 zu K5 ?
    - K1 → straight → K2
    - K4 → exchange → K5



### 6.6.2.3 Indirekte (dynamische) Verbindungsnetzwerke (XI)

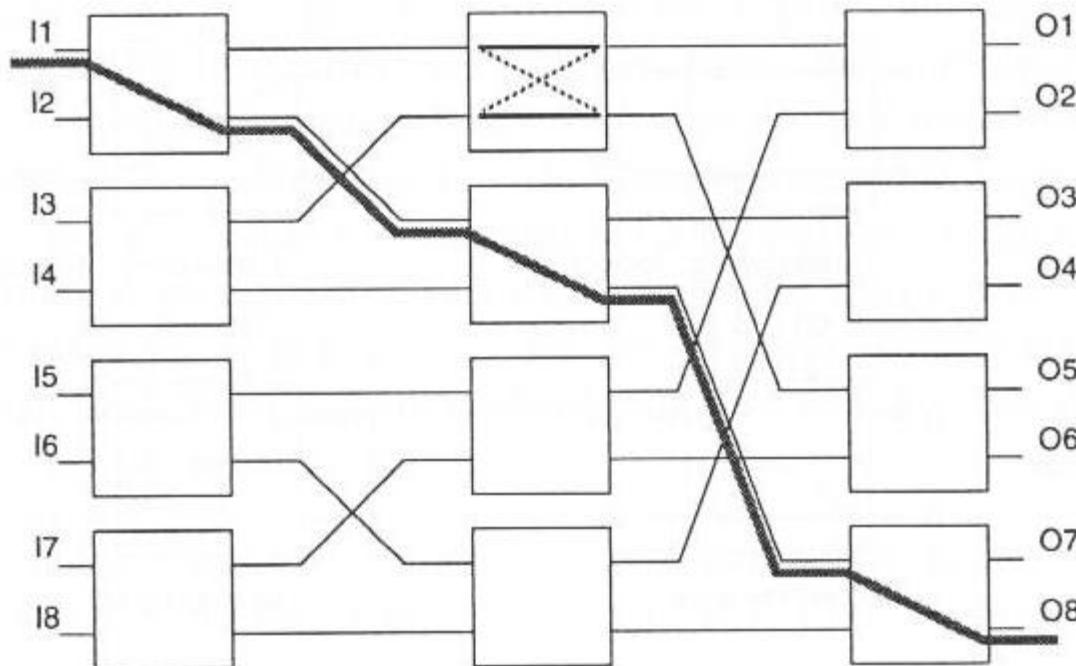
Typische mehrstufige indirekte Verbindungsnetze:

- multistage switching networks
- bestehen aus mehreren Schichten/Lagen von Schaltern und Leitungen, zumeist auf Shuffle- oder Crossbar-Basis
- Ziel: für größere Zahl von zu verbindenden Einheiten einen geringeren Abstand zu erreichen als mit direkten Netzen
- meist regelmäßiger Aufbau und gleichgroßer Grad der Ein-/Ausgänge bei den Schaltern
- für den Pfad durch das Gesamtnetz müssen die Schalter in den einzelnen Stufen dynamisch passend gesetzt werden
- Bsp.: nicht blockierungsfrei:  
Banyan (Butterfly)  
Delta  
Baseline  
Omega
- blockierungsfrei:  
Benes  
CLOS

### 6.6.2.3 Indirekte (dynamische) Verbindungsnetzwerke (XII)

Typische mehrstufige indirekte Verbindungsnetze:

- **Banyan- (Butterfly-) Netz k-dimensional**
  - $n = 2^{k+1}$  Eingänge  $\rightarrow m = 2^{k+1}$  Ausgänge,  $k+1$  Stufen mit je  $2^k$  Knoten aus  $2 \times 2$  Crossbar-Schaltern
  - von jedem Eingang gibt es genau einen Weg zu jedem Ausgang



3 stufiges  
Banyan Netz mit  
einem Pfad, der  
weitere Pfade  
blockiert

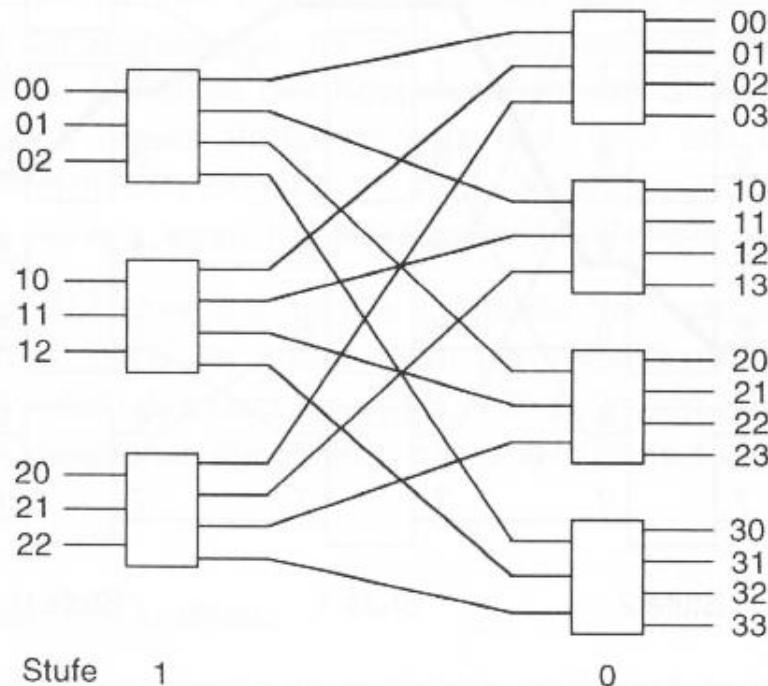
Bild-Quelle: Waldschmidt:  
Parallelrechner. Teubner, 1995;

### 6.6.2.3 Indirekte (dynamische) Verbindungsnetzwerke (XIII)

Typische mehrstufige indirekte Verbindungsnetze:

- **Delta-Netz**

- Schalter beliebiger Größe, oft jedoch 2x2 Crossbar-Schalter  
(typ. Vertreter: Generalized Cube GC-Netze)
- von jedem Eingang gibt es genau einen Weg zu jedem Ausgang



Zweistufiges Delta-Netz  
mit 9 Eingängen und 16  
Ausgängen

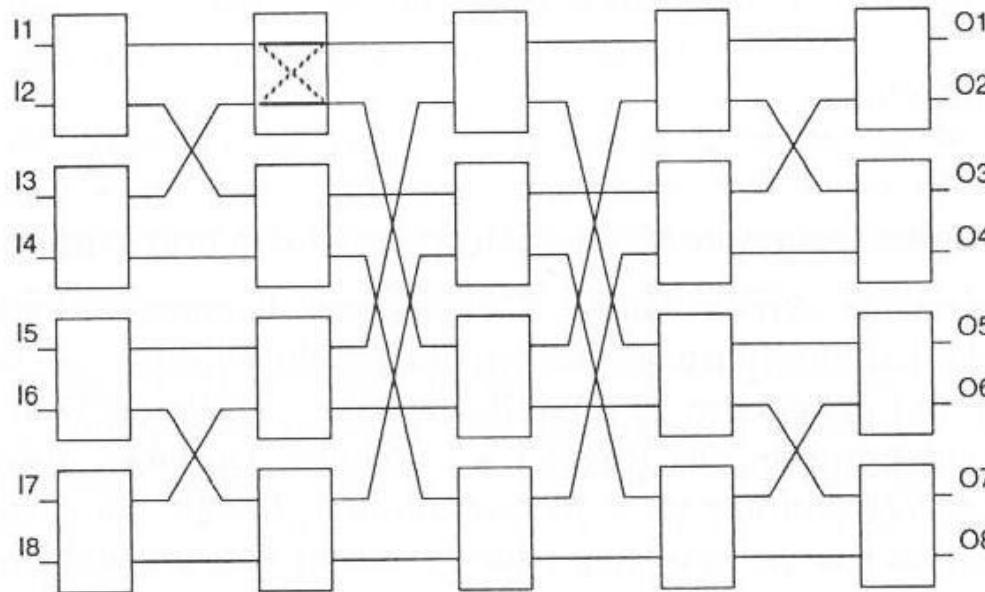
Bild-Quelle: Schwedersky/Jurczyk:  
Verbindungsnetzwerke. Stuttgart:  
BG Teubner, 1996

### 6.6.2.3 Indirekte (dynamische) Verbindungsnetzwerke (XIV)

Typische mehrstufige indirekte Verbindungsnetze:

- **Benes-Netz k-dimensional**

- besteht aus zwei hintereinandergeschalteten Banyan-Netzen
- $2k+1$  Stufen mit je  $n = 2^k$  Schaltern pro Stufe
- verbessertes Blockierungsverhalten im Vgl. zu Banyan, weil mehrere Kommunikationen im Netz parallel möglich sind



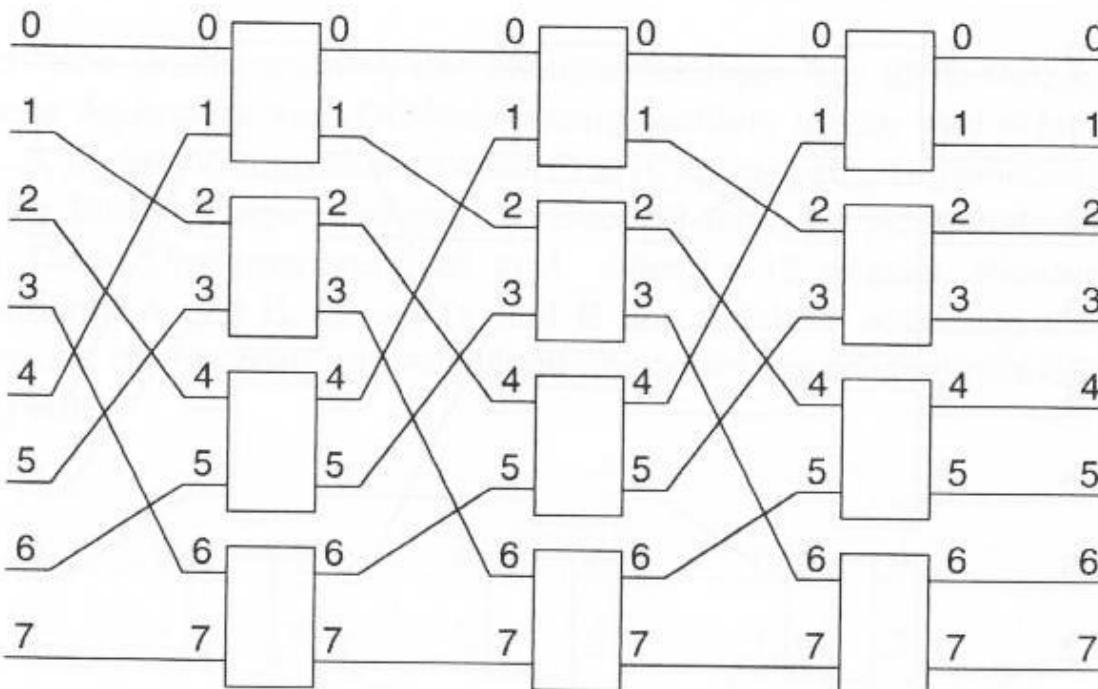
Fünfstufiges Benes-Netz mit 8 Eingängen und 8 Ausgängen  
(als Verbindung von 2 dreistufigen Banyan-Netzen)

Bild-Quelle: Waldschmidt:  
Parallelrechner. Teubner, 1995;

### 6.6.2.3 Indirekte (dynamische) Verbindungsnetzwerke (XV)

Typische mehrstufige indirekte Verbindungsnetze:

- **$n \times n$  Omega-Netz (auch „Perfect Shuffle“)**
  - $\log_2 n$  Stufen mit je  $n/2$  Schaltern (2x2 Schalter)
  - insgesamt  $(n/2) \log_2 n$  Schalter (vgl.  $n^2$  bei Kreuzschienenvert.)
  - topologisch äquivalent zum GC-Netz



8x8 Omega-Netz  
mit 3 Stufen  
= 12 2x2-Schalter  
(z.B. für 8 CPUs  
und 8 Speicher-  
module)

Bild-Quelle: Schwedersky/Jurczyk:  
Verbindungsnetzwerke. Stuttgart:  
BG Teubner, 1996

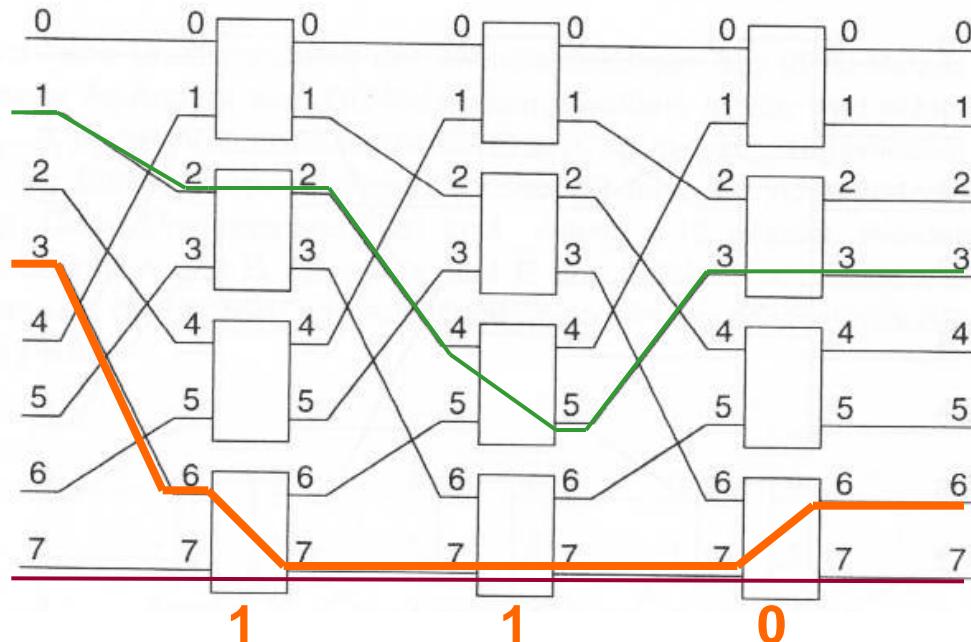
### 6.6.2.3 Indirekte (dynamische) Verbindungsnetzwerke (XVI)

Typische mehrstufige indirekte Verbindungsnetze:

- **$n \times n$  Omega-Netz**

*Beispiel: Eingang = Prozessor, Ausgang = Speicher*

- Prozessor #3 will Daten aus Speichermodul #6 (binär: 110) lesen
- Jeder Schalter der Stufe i interpretiert das i-te Bit der Adresse und schaltet bei 0 den oberen und bei 1 den unteren Ausgang durch



Es sind parallel weitere Zugriffspfade möglich, (z.B. von Prozessor #1 nach Speichermodul #3), aber auch Blockierungen, (z.B. Pfad von Prozessor #7 nach Speichermodul #7 klappt wegen Schalter in Stufe 1 nicht - Konkurrenz).

Bild-Quelle: Schwedersky/Jurczyk:  
Verbindungsnetzwerke. Stuttgart:  
BG Teubner, 1996 (ergänzt)

### 6.6.2.3 Indirekte (dynamische) Verbindungsnetzwerke (XVII)

Mischform aus direkten und indirekten Verbindungsnetzen:

- mehrfache Verwendung des gleichen (oder verschiedener) Netzwerke zum Aufbau eines komplexeren Verbindungsnetzwerks
- Beispiele: Crossbar-Hierarchien,

Fat Tree

Stern/Switch

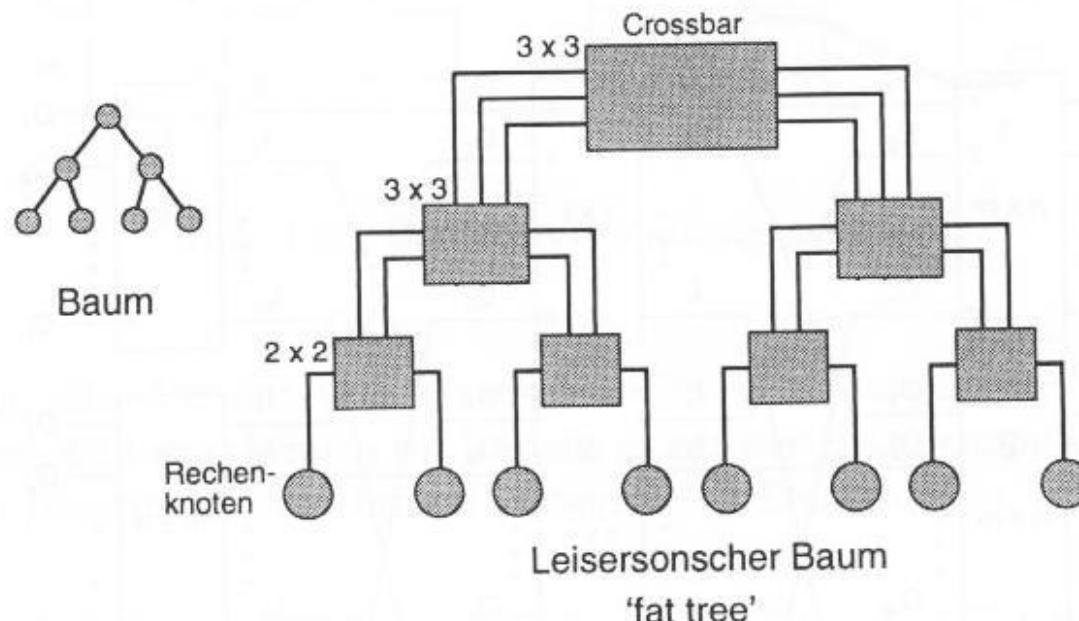
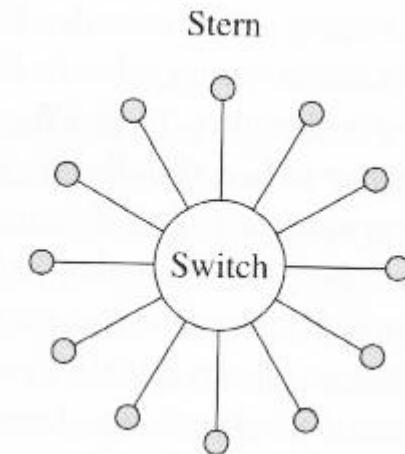


Bild-Quelle: Waldschmidt:  
Parallelrechner. Teubner, 1995;

### 6.6.2.3 Indirekte (dynamische) Verbindungsnetzwerke (XVIII)

#### Switch

- „elektron. Simulation“ einer Vollvernetzung
- Switch ist mittels Ports direkt mit jedem Knoten verbunden
- ankommende Nachrichten werden geprüft und an den Zielknoten weitergeleitet
- mehrere disjunkte Knotenpaare können gleichzeitig mit voller Bandbreite kommunizieren
- Verringerung von Kollisionen durch kurzzeitiges Zurückhalten von Paketen im Switch
- mehrfache „Überbuchung“ der Ports möglich, trotzdem noch gute Lösung, weil selten gleichzeitige Benutzung vieler Ports nötig ist
- typischer Vertreter: switched Ethernet (vor allem bei Clustern)

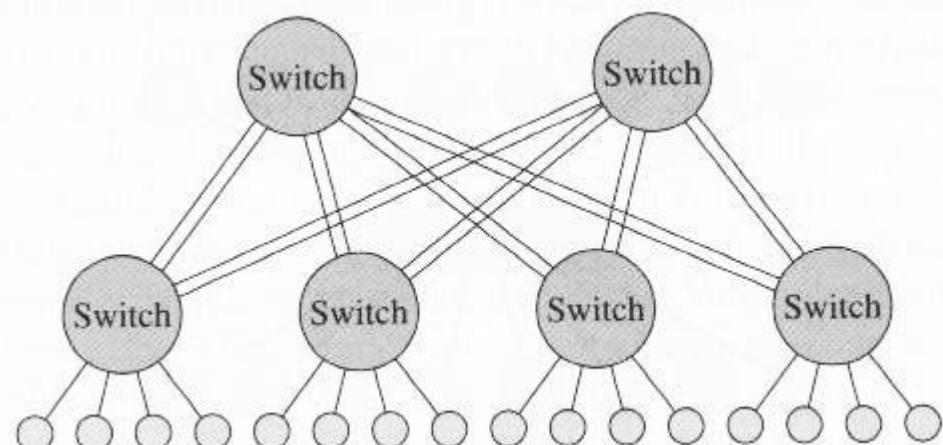
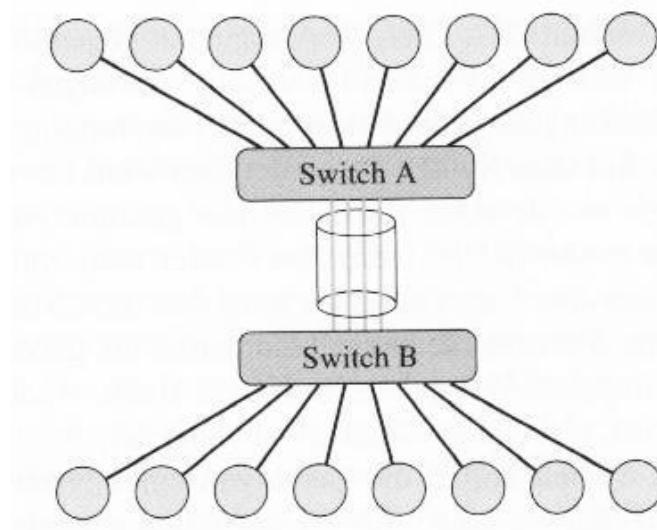


### 6.6.2.3 Indirekte (dynamische) Verbindungsnetzwerke (XIX)

Beispiele für Techniken zur Vergrößerung der Bandbreite:

zwei Switches mit gebündelter  
Verbindung (trunking)

fat tree mit mehreren Switches



### 6.6.3 Verbindungsart

#### 6.6.3.1 Überblick

Zwei Verbindungsarten:

- **Leitungsvermittlung:** Belegung einer Leitung für die gesamte Dauer der Datenübermittlung
  - günstig für viele schnelle Übertragungen zwischen denselben Partnern
  - aber vorübergehende Blockierung der benutzten Leitungen und Schalter
- **Paketvermittlung:** Aufteilung der Datenmenge in Pakete und Transport der einzelnen Pakete im Netz
  - dazu sind unterwegs Adressangaben auszuwerten (Routing)
  - Leitungen können kurz nacheinander von verschiedenen Paketen mit unterschiedlichen Zielen durchlaufen werden
  - wird wegen größerer Bedeutung für PR weiter betrachtet ...

### 6.6.3.2 Paketvermittlung (I)

#### Routing

- entscheidet, **welchen** Pfad eine Nachricht im Netz gehen soll
- die Netz-Topologie bestimmt die prinzipiell möglichen Pfade
- Beachtung von Konkurrenz: Verzögerungen, falls mehrere Nachrichten zur gleichen Zeit über die selbe Verbindung geschickt werden sollen
- Vermeidung von Staus, z.B. bzgl. Puffer, Gefahr von Paketverlusten
- Optimierung bzgl. Verbindungskosten (Länge und Belastung des Pfades)

### 6.6.3.2 Paketvermittlung (II)

#### Klassifizierungsmöglichkeiten von Routingalgorithmen

- bzgl. Pfadlänge:
  - Minimale Routingalgorithmen:  
wählen stets den kürzesten Pfad, Staugefahr
  - Nichtminimale Routingalgorithmen:  
weichen bei Überlastung auch von minimalen Paden ab
- bzgl. Deterministik:
  - Deterministische Routingalgorithmen:  
Pfad ist eindeutig - nur in Abhängigkeit von Sender und Empfänger, kann zu ungleichmäßiger Netzauslastung führen
  - Adaptive Routingalgorithmen:  
bietet mehrere mögliche Pfade bei Beachtung der Netzauslastung und damit auch höhere Fehlertoleranz

### 6.6.3.2 Paketvermittlung (III)

Beispiele von Routingalgorithmen:

- XY-Routing für zweidimensionale Gitter:
  - Paket geht zuerst horizontal bis zur entspr. Zielkoordinate und dann vertikal bis zum Ziel;
  - ist deterministisch und minimal
- Quellenbasiertes Routing
  - Sender legt gesamten Pfad fest: für jeden Knoten auf dem Pfad wird vorab der zu wählende Ausgang bestimmt;
  - die Folge der nacheinander zu wählenden Ausgänge wird als Adressinformation im Kopf der Nachricht gespeichert;
  - beim Passieren eines Knotens wird die entspr. Eintragung aus dem Kopf entfernt
  - Bsp.: Myrinet

### 6.6.3.2 Paketvermittlung (IV)

Beispiele von Routingalgorithmen (Forts.):

- Tabellenorientiertes Routing
  - Jeder Knoten enthält eine Routingtabelle, die für jede Zieladresse den zu wählenden Ausgang bzw. den nächsten Knoten angibt
  - Bsp.: Ethernet
- Virtuelle Kanäle
  - vor allem für minimale adaptive Kanäle
  - mehrere virt. Kanäle teilen sich eine physikal. Verbindung
  - für jeden virt. Kanal wird ein eigener Puffer bereitgestellt
  - Realisierung z.B. mittels Zerlegung in log. Teilnetze

### 6.6.3.2 Paketvermittlung (V)

#### Switching

- legt fest, **wie** eine Nachricht den vom Routing gewählten Pfad durchläuft, d.h.
  - ob und wie eine Nachricht in Pakete zerlegt wird
  - ob der Pfad zwischen Quelle und Ziel vollständig oder nur teilweise allokiert wird
  - wie Nachrichten vom Eingang eines Schalters auf seinen Ausgang gelegt werden (Routing legt fest, welcher Ausgang zu wählen ist!)
- Lösung hat großen Einfluss auf Zeitbedarf für Datentransport

### 6.6.3.2 Paketvermittlung (VI)

#### Switching-Strategien:

- Circuit-Switching
  - Es wird der gesamte benötigte Pfad exklusiv für eine Nachricht geschaltet, bis sie das Ziel erreicht hat
- Packet-Switching
  - Nachricht wird in Pakete zerlegt, die unabhängig voneinander durch das Netz gesendet werden
  - Paket enthält i.allg. Header, Daten und Prüfsumme
  - verschiedene Realisierungsformen:
    - store-and-forward
    - cut-through
    - wormhole (HW-Unterstützung durch Router)

### 6.6.3.2 Paketvermittlung (VII)

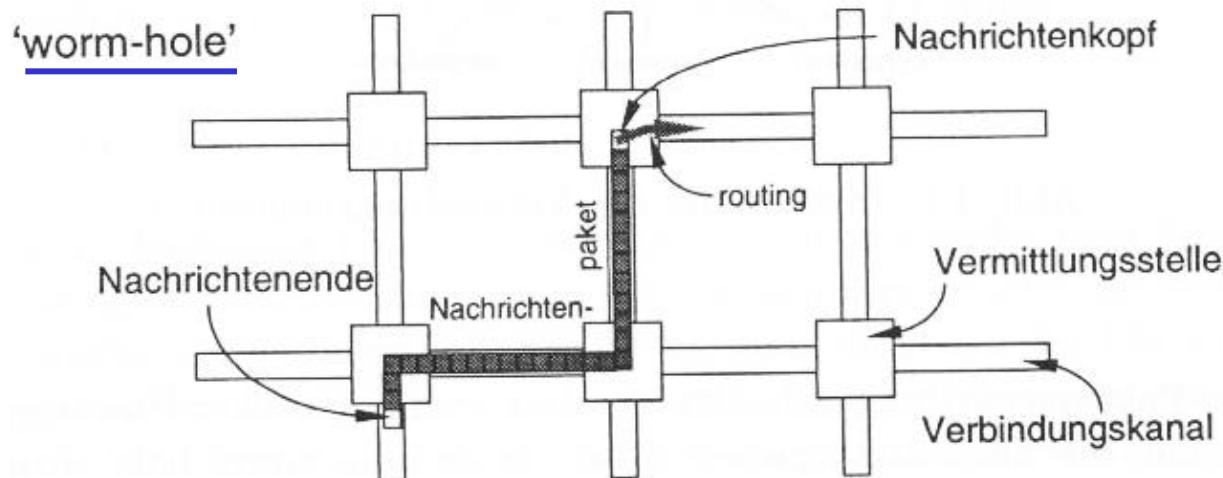
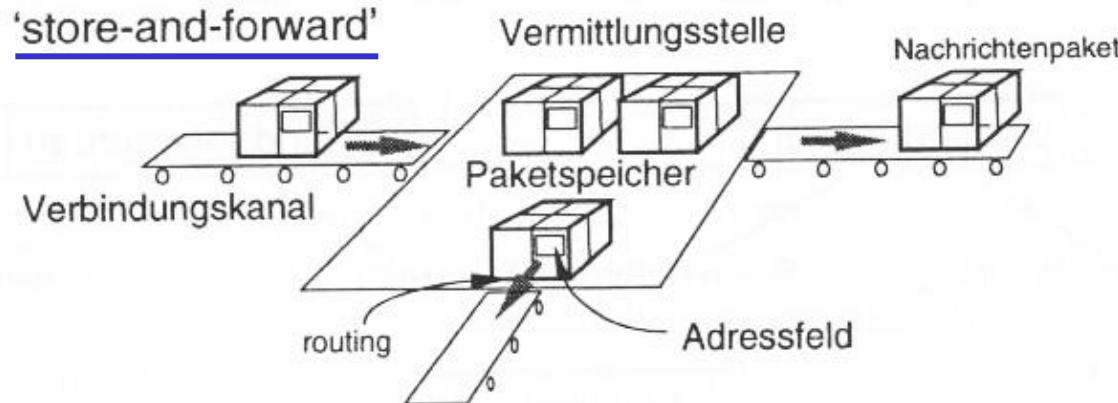


Bild-Quelle: Waldschmidt:  
Parallelrechner. Teubner, 1995;

### 6.6.3.2 Paketvermittlung (VIII)

#### Flusskontrolle

- nötig, wenn sich mehrere Nachrichten im Netzwerk befinden
- regelt, welchem Paket eine Verbindung oder ein Puffer zugeteilt wird
- falls eine Ressource bereits belegt ist, ist zu entscheiden, ob die Nachricht/das Paket
  - blockiert
  - in einem Puffer zwischengespeichert
  - über alternativen Pfad geleitet
  - einfach verworfenwerden soll
- bei PR besondere Anforderungen weil sich sehr viele Nachrichten auf engem Raum im Netzwerk befinden, daher besondere Zuverlässigkeit, hohe Geschwindigkeit und Stauvermeidung nötig

## 6.6.4 Beispiele für Hochgeschwindigkeitsnetzwerke (I)

### Gigabit Ethernet

- IEEE 802 für lokale Netze (PC-Bereich)
- CSMA/CD Verfahren, Kollisionserkennung
- Glasfaser- oder Kupferleitungen
- zuverlässig, robust, preiswert
- Switches

### Myrinet (Fa. Myricom)

- proprietäres GM-Protokoll
- Glasfaserkabel, Switches mit internem fat-tree-Topologie
- leistungsfähiger, aber teurer als Ethernet

### 6.6.4 Beispiele für Hochgeschwindigkeitsnetzwerke (II)

#### **SCI Scalable Coherent Interface**

- unidirektionale Punkt-zu-Punkt-Verbindungen (kein Switch)
- Topologie: Ring, 2D-Torus, 3D-Torus
- besonders geringe Latenzzeit zwischen benachbarten Knoten wächst allerdings mit Entfernung rasch an
- Kupfer- oder Glasfaserleitungen
- rel. teuer

#### **InfiniBand (InfiniBand Trade Assoc., Firmenverband)**

- rel. komplexe, universelle und zuverlässige Architektur (IBA)
- Netzwerk aus geswitchten bidirektionalen Verbindungen  
→ „fabric“ („Gewebe“)
- Kupfer- oder Glasfaserkabel, computerintern via Backplane

#### 6.6.4 Beispiele für Hochgeschwindigkeitsnetzwerke (III)

##### [QsNet, QsNet II \(Fa. Quadrics\)](#)

- hohe Bandbreite, geringe Latenz, gute Skalierung
- insbes. zur Verwendung bei SMP-Systemen
- Kupfer- oder Glasfaserkabel
- Switches mit interner fat-tree-Topologie

## 6.7 Realisierungsbeispiele

Aktuell: siehe [TOP500.org](http://TOP500.org) ☺

„prominente“ Beispiele:

### a) Sun Fire E25 K

- NUMA, gemeinsamer Speicher
- 72 UltraSPARC-IV-Prozessoren (jeweils 2 Cores UltraSparc-III-Cu)
- 18 BoardSets (mit je 8 CPUs + 32 GB RAM + 4 PCI)
- insges. 144 CPUs, 576 GB RAM, 72 PCI-Slots
- Zentralplatine mit 3 18x18 Kreuzschienenverteilern (für Adressen, Rückverbindung und Daten)
- komplexes Speichersystem (weil entspr. großer Kreuzschienenverteilер nicht realisierbar)

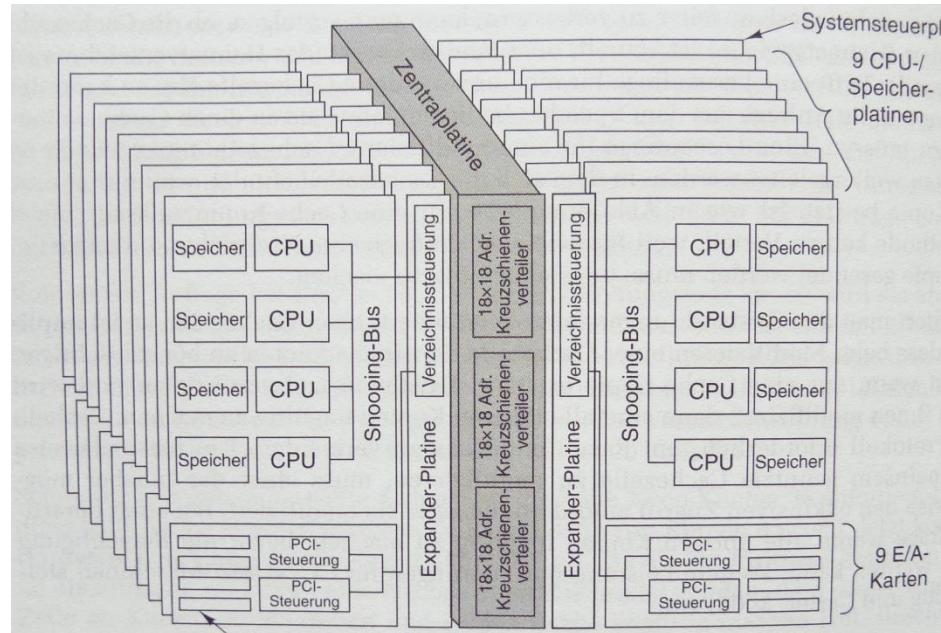


Abbildung 8.30: Der E25K-Multiprozessor von Sun Microsystems

## 6.7 Realisierungsbeispiele

### a) Sun Fire E25 K (II)

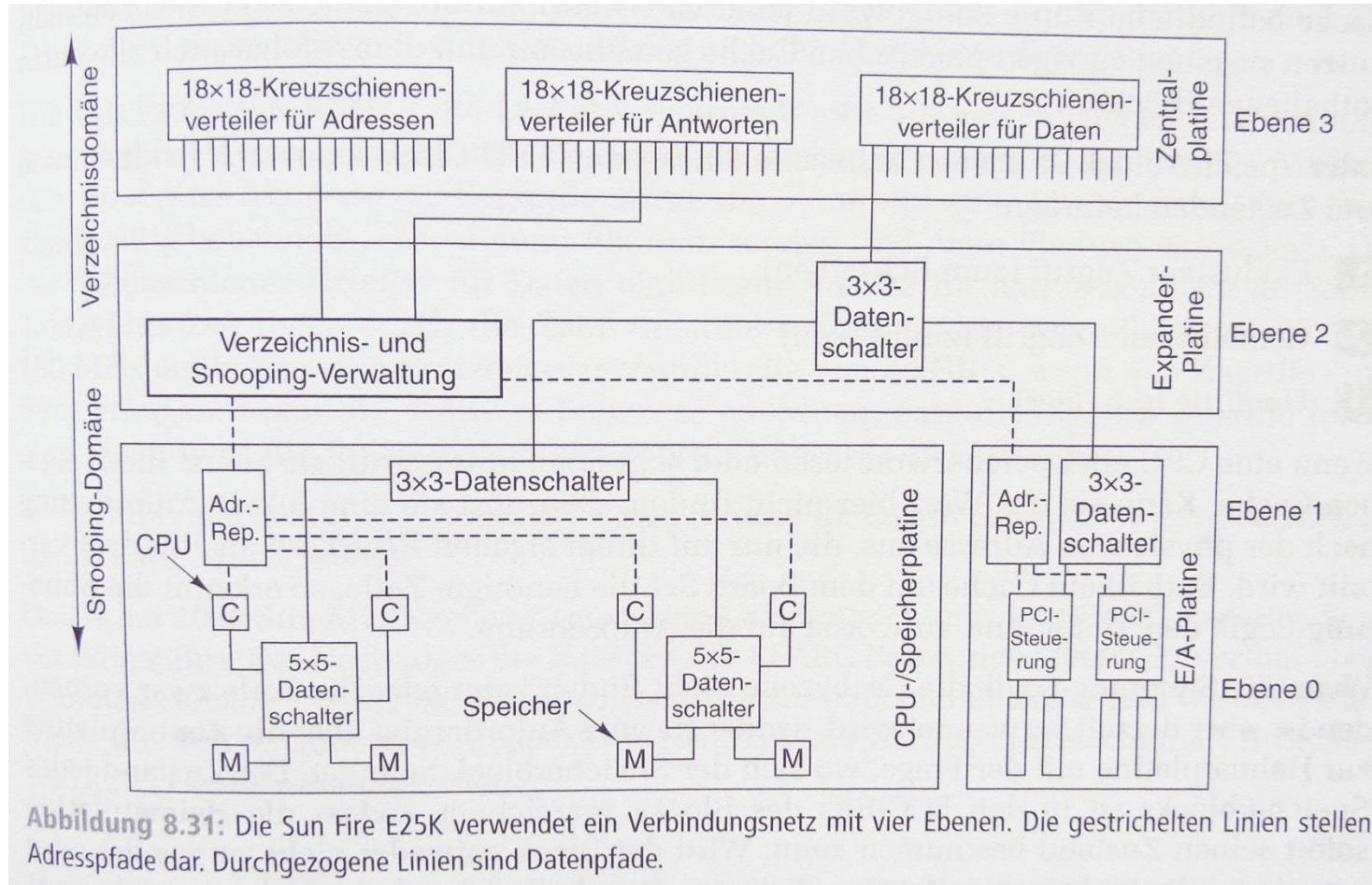


Abbildung 8.31: Die Sun Fire E25K verwendet ein Verbindungsnetz mit vier Ebenen. Die gestrichelten Linien stellen Adresspfade dar. Durchgezogene Linien sind Datenpfade.

## 6.7 Realisierungsbeispiele

### b) IBM BlueGene

- Massiv Paralleles System (MPP)
- 1. Generation /L (2001), 2. Generation /P (2007)
- Designziele: Effizienz bzgl. FLOPs/\$, FLOPs/Watt, FLOPs/m<sup>3</sup>
- kundenspezif. Lösung auf einem Chip als Basis für MPP-System
  - 4 PowerPC-450 Kerne
  - Je Kern 2 FP-Einheiten
  - 3 Cache Level
  - Cache-Kohärenz zw. Kernen und L1-Cache
  - schneller Bus zum VN (= 3D Torus, 72 x 32 x 32) daher pro Chip 6 Verbindungen (oben, unten, N, O, S, W)
  - schnelles Netzwerk für Daten-Broadcast zu allen Prozessoren

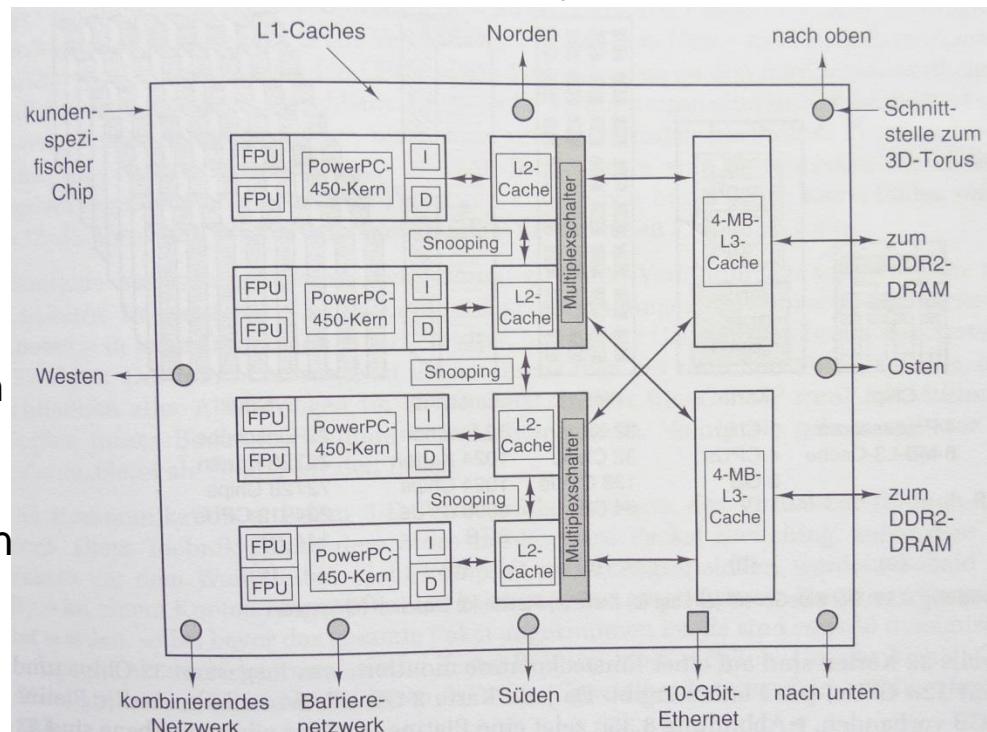


Abbildung 8.34: Der kundenspezifische Prozessorchip BlueGene/P

## 6.7 Realisierungsbeispiele

### b) IBM BlueGene (II): Gesamtsystem

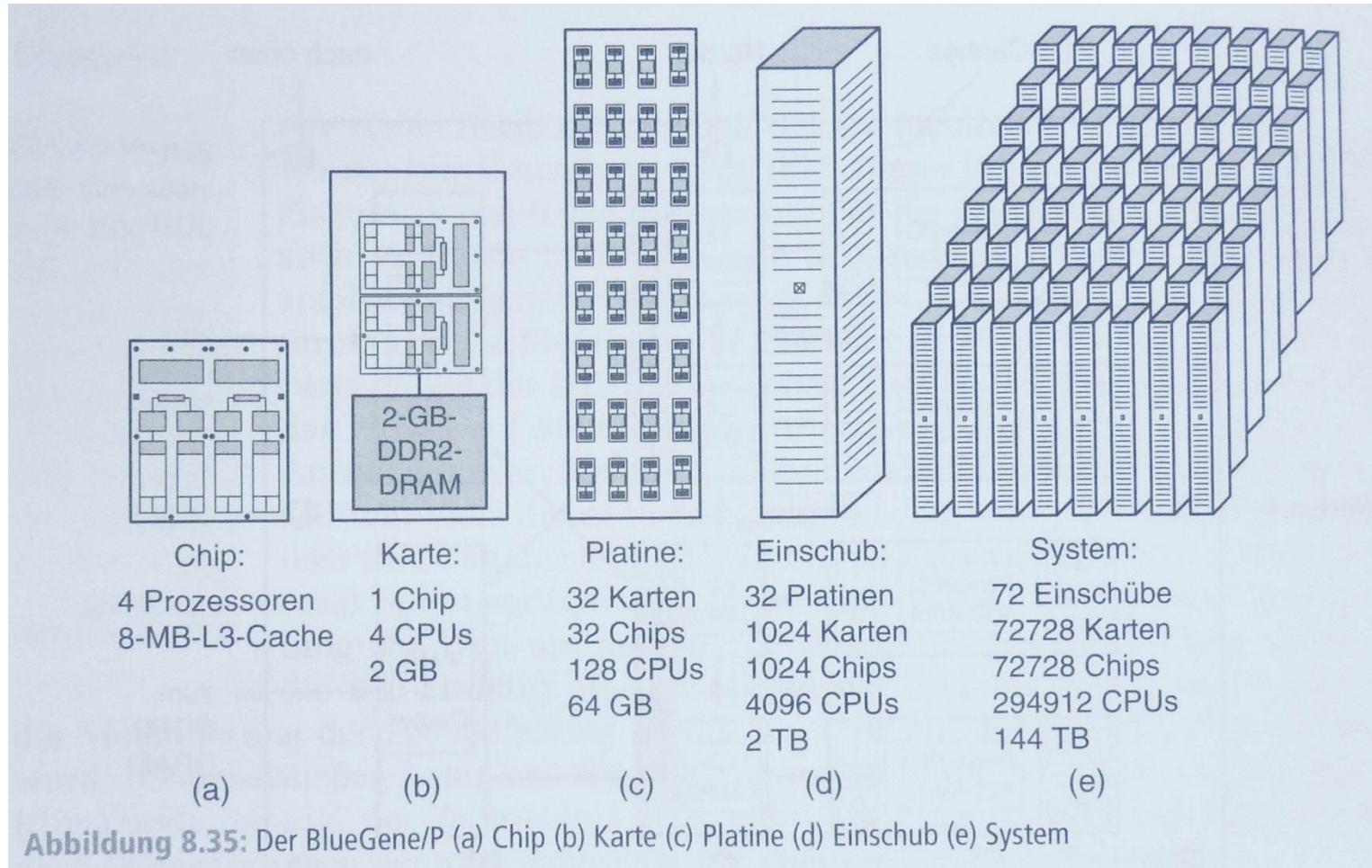


Abbildung 8.35: Der BlueGene/P (a) Chip (b) Karte (c) Platine (d) Einschub (e) System

## 6.7 Realisierungsbeispiele

### c) Google Cluster:

- Ziel: optimales Preis-Leistungsverhältnis
- daher keine High-End-Server, sondern sehr viele Standard-PCs
- zwar vgl.-weise höhere Ausfallrate von ca. 2% pro Jahr, aber fehlertolerante SW-Lösungen
- Grundbaustein „Google-PC“:
  - Intel CPU, 4GB RAM, 2TB Platte, + spez. Ethernet-Chip
  - Gehäuse ca. 5 cm Höhe
  - bis zu 40 PCs in einem 19“ Einschub
  - bis zu 80 PCs in einem Gestellrahmen, verbunden via Ethernet Switch
  - insges. z.B. 5120 PCs in einem typ. Datenzentrum
  - hohe Energiedichte 3000W/m<sup>3</sup> bei Platzbedarf von ca. 3m<sup>2</sup> pro Gestellrahmen

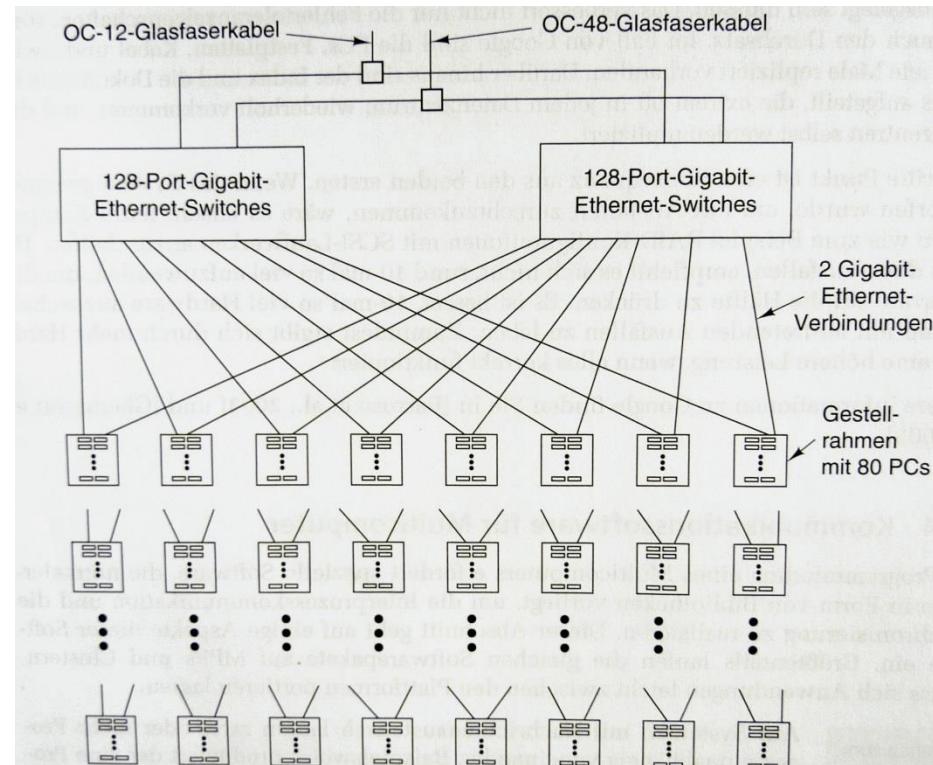


Abbildung 8.39: Ein typischer Google-Cluster