Programmierung Paralleler Systeme

Kapitel 1: Einführung

Claus-Uwe Linster

IMMD IV, Lehrstuhl für Betriebssysteme
Universität Erlangen-Nürnberg

linster@informatik.uni-erlangen.de

Inhaltsangabe: Programmierung Paralleler Systeme (z. Zt. 2 + 2)

Inhaltsangabe: Programmierung Paralleler Systeme (z. Zt. 2 + 2)

1 Parallelität, wo?

auf Prozessorebene

Speichersystemebene

Systemebene, Verbindungsnetzwerkebene

Netzebene

E/A-Ebene (wird nicht behandelt im Rahmen dieser Vorlesung)

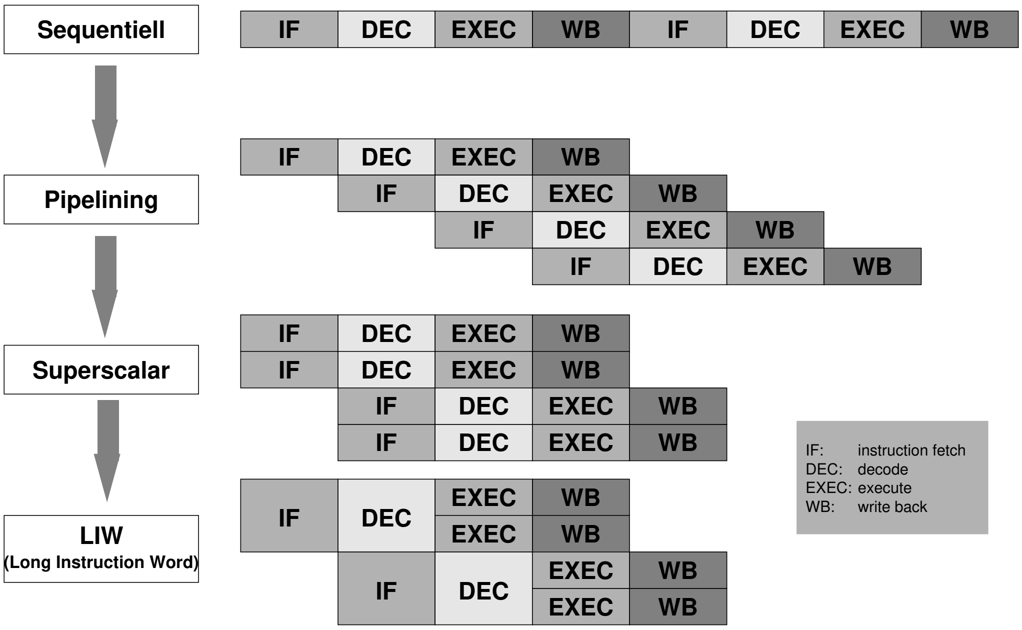
1.1 ... auf Prozessorebene

- Parallelisierung des Befehlsstroms
  - Pipelining
  - Superskalare Prozessoren (mehrere Functional Units)
  - VLIW (Very Large Instruction Word)
- Parallelisierung des Datenstroms
  - Vektoreinheiten
- allen Hardware-Eigenschaften gemeinsam:
  Nutzung ist abhängig von der “Compilerleistung” und der inhärent vorhandenen Möglichkeiten des Programmcodes, die HW-Eigenschaften optimal zu nutzen; die potentiell mögliche Parallelität wird stark durch die gegenseitige Abhängigkeit der Daten bestimmt.

1.1.1 Pipelining, Superscalar, VLIW

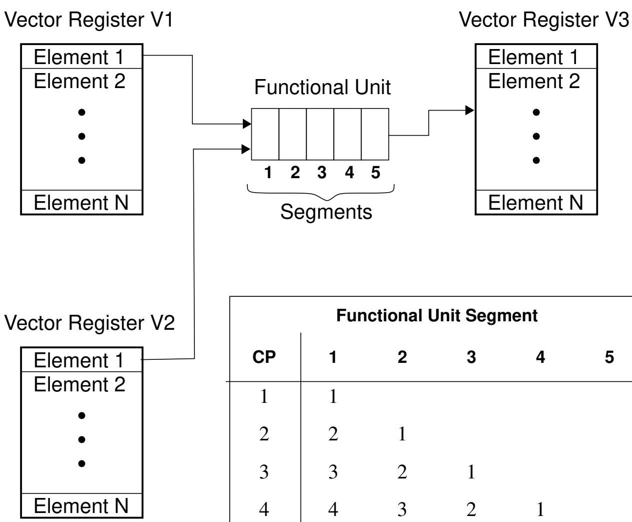
- Sequentiell:
  Die einzelnen Phasen einer Befehlsausführung werden sequentiell abgearbeitet. Ein Instruktionsstrom.
- Pipelinig:
  verschiedene Phasen verschiedener Instruktionen werden parallel verarbeitet. Ein Instruktionsstrom.
- Superscalar:
  die Möglichkeit mehrere Funktionen parallel zu verarbeiten
  z. B. Load / Store : Add / Multiply : Branch
  Paralleler Instruktionsstrom ausgewählter Instruktionstypen.
- VLIW:
  “Breite” Instruktions-Verarbeitung
  mehrere Befehle werden parallel bearbeitet.
  Paralleler Instruktionsstrom.

Pipelining, Superscalar, (V)LIW



1.1.2 Vektorprozessoren

- “Breite” Daten-Verarbeitung
  ein Vektor von Daten wird syncron in gleicher Weise bearbeitet



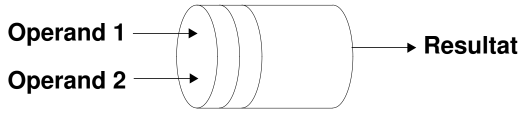
Functional Unit Segment

CP	1	2	3	4	5
1	1				
2	2	1			
3	3	2	1		
4	4	3	2	1	
5	5	4	3	2	1
6	6	5	4	3	2

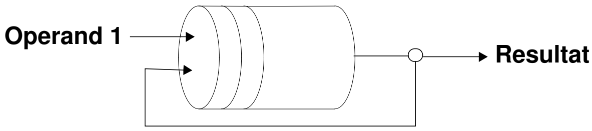
Elements in each segment during successive CPs

1.1.2.1 Vektor-Pipeline-Betriebsarten

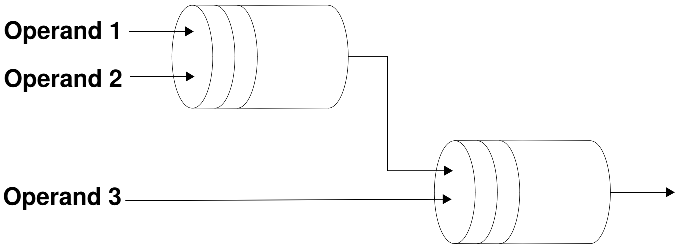
- Vektorverarbeitung in Kombination mit “Instruktionsparallelität”
  - normale Betriebsart



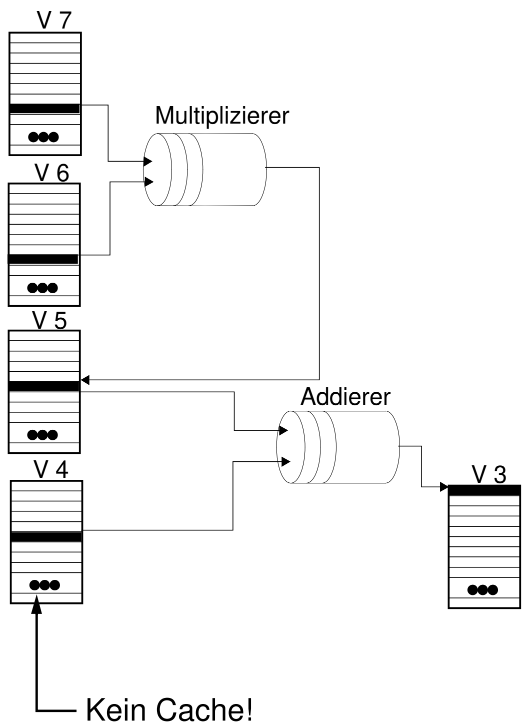
  - Rückkopplung



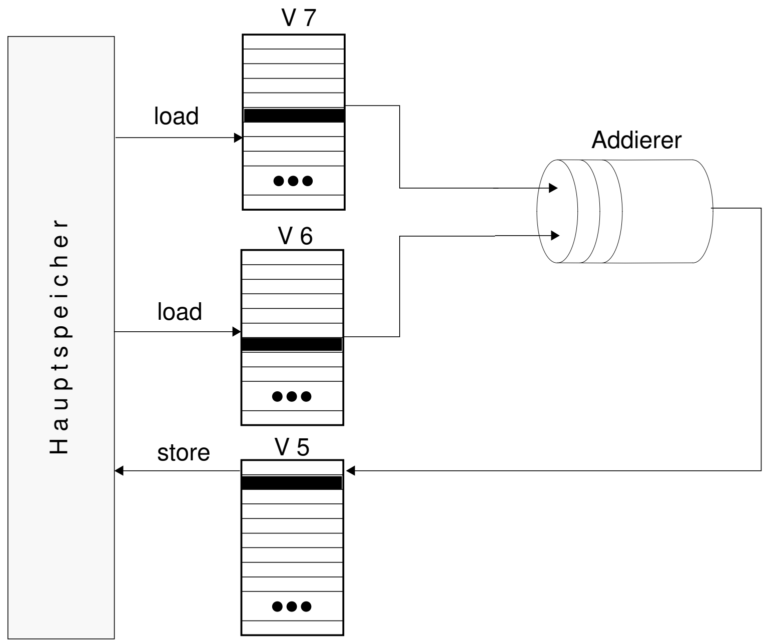
  - Verkettung (Chaining)



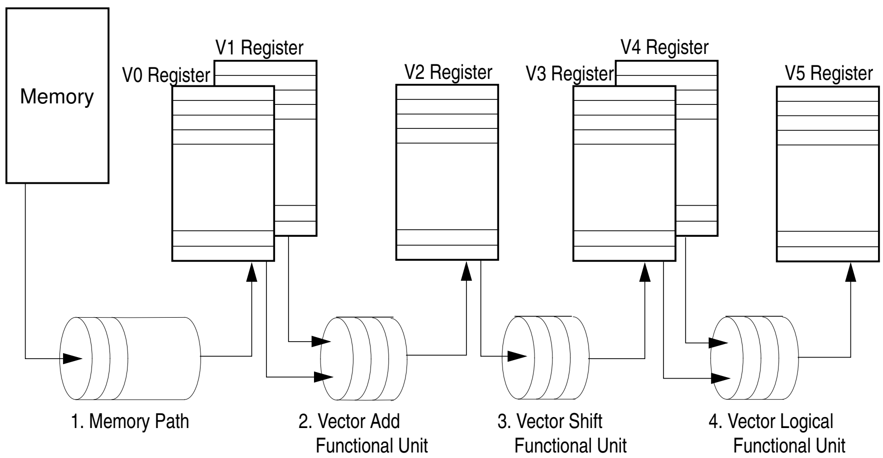
1.1.2.2 Chaining zwischen Registern



1.1.2.3 Chaining zum Speicher

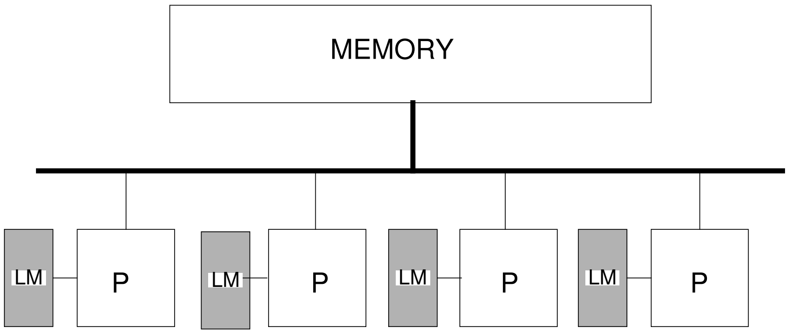


1.1.2.4 Vektor Chaining Beispiel



1.2 ...auf Speicherebene

- Shared-Memory-Systeme



- Durch die techn. Entwicklung bedingt, stieg die Verarbeitungsgeschwindigkeit der Prozessoren stärker, als die der (zu einem vertretbaren Preis herzustellenden) Arbeitsspeicher.
- Seit Mitte der 70er Jahre: Prozessoren (Flops) ca. 1 : 10.000
  Speicherzyklus ca. 1 : 10

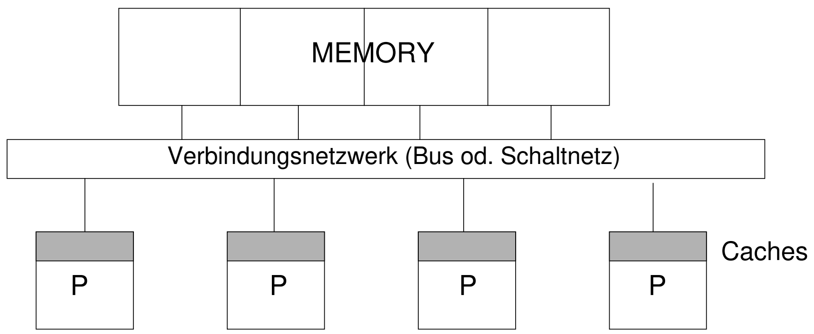
...auf Speicherebene

- Abhilfe: Erweiterung der Speicherbandbreite durch
  - Parallelisierung des Zugriffs
    - Speicherbänke
    - Interleaving (verschränkter Speicherzugriff)
    - Daten-Transfer in “Burst Mode” (einmal Adresse, viele Daten)
    - Verbreiterung der Datenpfade (mit Memory Banking)
    - Einführung von Schaltnetzwerken und damit potentiell vielen parallelen Zugriffspfaden (z. B. Crossbar; hier. Crossbar)
  - Erhöhtem techn. Einsatz
    - Erhöhung der Übertragungsleistung durch höhere Taktraten auf dem Bus; Problem: Je schneller der Bus, desto “kürzer” der phys. Aufbau;
    - Einführung von Speicherhierachien (Cache: on-node; on-board; on-chip)
    - Verbesserte Cache-Kohärenz-Protokolle (write through; copy back)
    - Vergrößerung der Cache-Speicher (kann kontraproduktiv wirken).

...auf Speicherebene

- Der technologische Aufwand bedingt wiederum komplexere "Speicheranschaltungen", die sich aufgrund räumlicher Ausdehnung nicht beliebig hoch takten lassen (Einschwingverhalten) und deren Funktionalität sich nicht mit "einem" Takt realisieren läßt.
- Die Zeit, die zwischen dem Anstoß eines Zugriffs auf den Speicher und dem Eintreffen des 'ersten" Datums vergeht, nennt man Latenzzeit.
  - Diese Latenzzeit ist in der Regel höher als die Zykluszeit des Speichers.
  - Heute typische Latenzzeiten auf den Zugriff lokaler Speicher beträgt etwa 50 Prozessorzyklen (etwa: 250 - 400 ns!!!)
- Maßnahmen zur Verdeckung der Latenzzeit *(Latencyhiding)*
  - *Prefetching*; d.h. rechtzeitiges Anstoßen eines Zugriffs; also nicht *"on demand".*
  - Die Möglichkeit mehrere *Prefetchings* anstoßen zu können: *oustanding loads*
  - Die Möglichkeit einer dynamischen Umordnung der Befehlsausführungs-reihenfolge *(Instuctionreordering)*; d.h. es wird das ausgeführt, was gerade ausführbar ist.

...auf Speicherebene



- Max. erreichbare (noch sinnvolle) Prozessorzahl bei nicht hierachischen Shared-Memory-Architekturen (UMA): 20 - 30;
  Typische Vertreter: Sun-Sparc-Multiprozessoren, SGI-Challenge
- Zur Erreichung größerer Prozessorzahlen bleibt nur die Verteilung des Speichersystems; Hierachisierung; Clusterung und damit NUMA-Architekturen.
  Typische Vertreter: Convex SPP1600, KSR, SGI Origin
- Da es aus Kostengründen auch Shared-Memory-Systeme ohne Cache-Kohärenz gibt (Cray T3E), spricht man bei denen mit Kohärenz auch von CCUMA-Systemen. (Cache-Coherend...)

1.3 Verbindungsnetzwerkebene

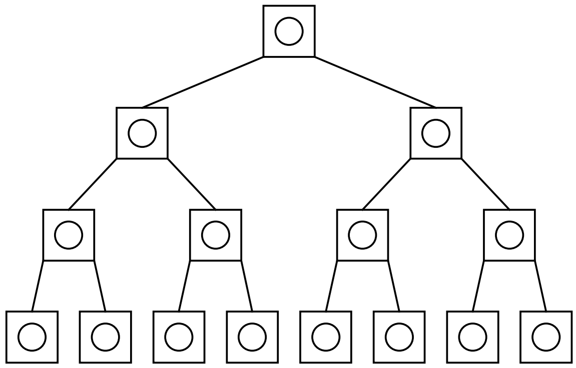
- Aufgrund der technisch sehr aufwendigen (und kostenintensiven) Speichersysteme, insbesondere für Rechner mit sehr großen Prozessorzahlen, favorisierte man - insbesondere in den 80er Jahren - Lösungen mit verteilten lokalen Speichern.
- Verbindungsnetzwerke
  - statische Verbindungsnetzwerke
  - Schaltnetzwerke
- Wunsch: Vollständiges Verbindungsnetzwerk; Komplexität steigt mit K!
  - Abhilfe:
    - Man beschränkt sich auf eine “gewisse” Nachbarschaft, die man direkt erreichen kann, wobei man sich auf Eigenschaften numerischer Lösungsverfahren stützt.
      Problem: globale Kommunikation / Synchronisation
    - globale Kommunikation wird mehrstufig mit - möglichst autarken, nicht die Rechenleistung einzelner Knoten belastenden - Routern realisiert.

Verbindungsnetzwerkebene

- Kriterien der Bewertung von stat. Verbindungsnetzwerken
  - Anzahl der direkt erreichbaren Nachbarknoten; Anzahl der Links pro Knoten
  - Anzahl der Zwischenstufen die erforderlich sind um den am weitesten entfernten Knoten zu erreichen (Durchmesser).
  - Anzahl der Verbindungssegmente insgesamt (Komplexität)

Verbindungsnetzwerkebene

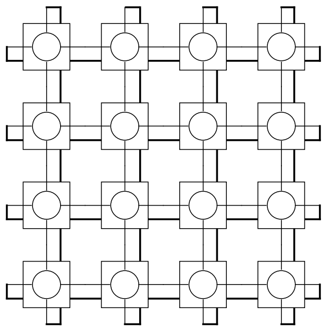
- Binärbaum



- Links pro Knoten: 3;
- Anzahl der Knoten: $K = 2^d - 1$; d = Ebenenzahl
- Ausbauinkrement: $I = 2 * K + 1$;
- Max. Routingstufen: $D = 2(ld (K+1)-1) = 2(d - 1)$; ~ ld(K);
- Komplexität des Netzes: $A = K -1$; ~ K
- Vermeidung des Kommunikationsengpasses zur Wurzel hin: Fat-Tree
- Typischer Vertreter: Thinking Maschines CM5

Verbindungsnetzwerkebene

- Gitter (2D - Mash)

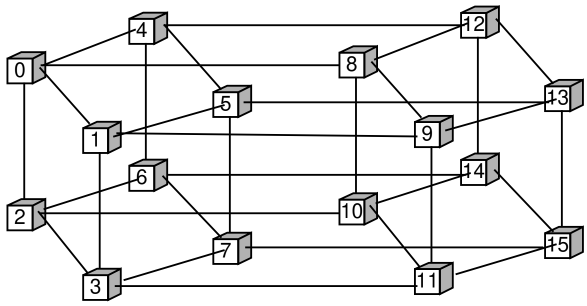


hier: zum Torus geschlossen; quadratisch

- Links pro Knoten: 4
- Anzahl der Knoten: $K = q^2$; q = Kantenlänge
- Ausbauinkrement: $I = 2(q + 1) - 1$
- Max. Routingstufen: $D = (K^{-2} -1)$
- Komplexität des Netzes: $A = 2 * K$
- Typische Vertreter: INTEL Paragon XP/S, Transputersysteme

Verbindungsnetzwerkebene

- Hypercube



- Links pro Knoten: n; n = Dimension
- Anzahl der Knoten: $K = 2^n$;
- Ausbauinkrement: I = 2 * K
- Max. Routingstufen: D = n
- Komplexität des Netzes: A = K * n/2; ~ ld(K)
- Typische Vertreter: nCUBE; Max. n = 13 (8.192 Knoten)

Verbindungsnetzwerkebene

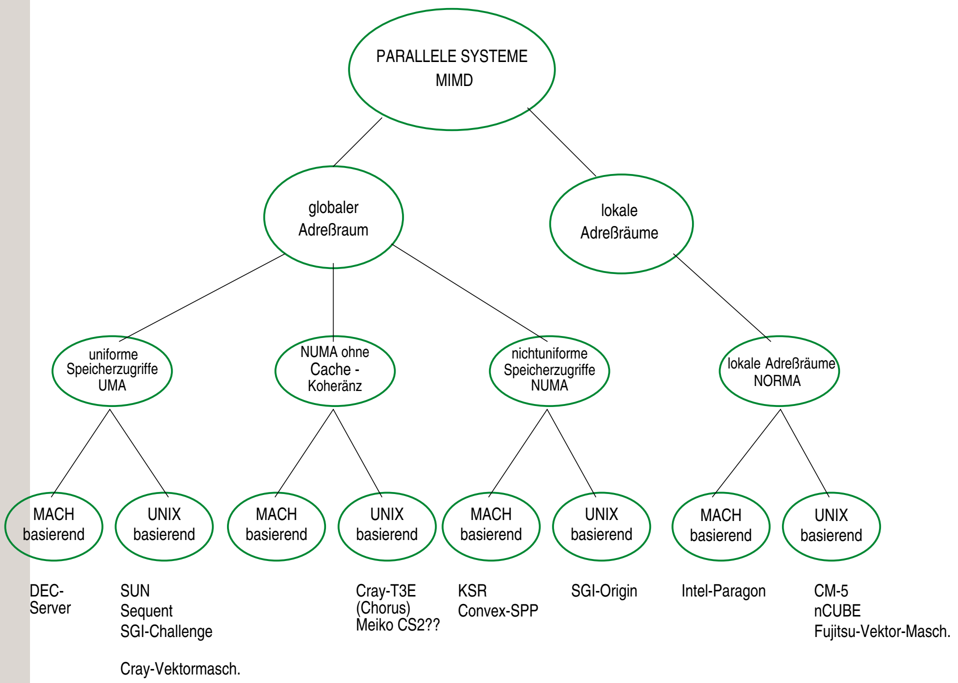
- Workstationcluster
  - gebildet mit Hochleistungs-Vernetzungskomponenten; spezielle Switches; z. B. IBM SP-2
  - gebildet auf der Basis von Standardvernetzungskomponenten; 100MBit Ethernet, ATM.
- Prognose für Systeme mit verteilten lokalen Speichern:
  - aufgrund der immer schneller werden Standard-Netzwerk-Komponenten, werden Spezialrechner aus Entwicklungskostengründen ihre Bedeutung verlieren; universell einsetzbare "Workstationcluster" werden sich hier durchsetzen.
  - Typische Vertreter: IBM -SP2
  - Problem: Geeignete Systemsoftware zur Administration und Betrieb solcher Systeme: z. B. NQS

1.4 Klassifikation Paralleler Systeme

- Flynn's Klassifikation

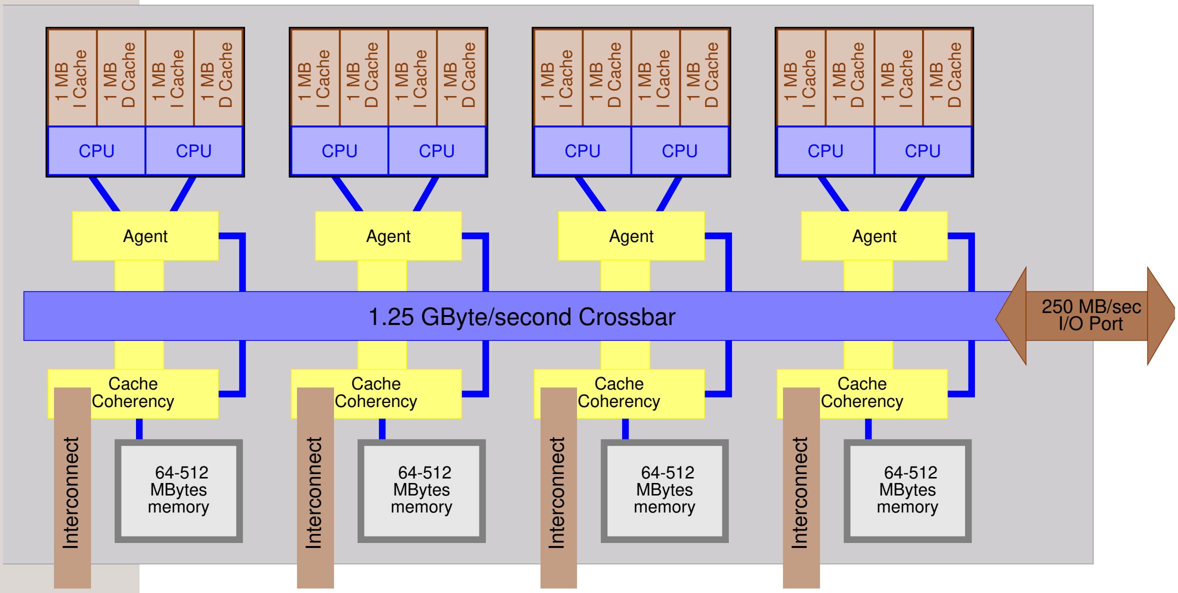
Data - Stream / Instruction - Stream	SINGLE	MULTIPLE
SINGLE	**SISD** Klass. von Neumann-Architektur	**SIMD** Vektor - Array - Prozessoren
MULTIPLE	**MISD** nicht sinnvoll	**MIMD** Parallelrechner; bestehend aus SISD - oder SIMD - Prozessoren

1.5 Parallele MIMD - Systeme



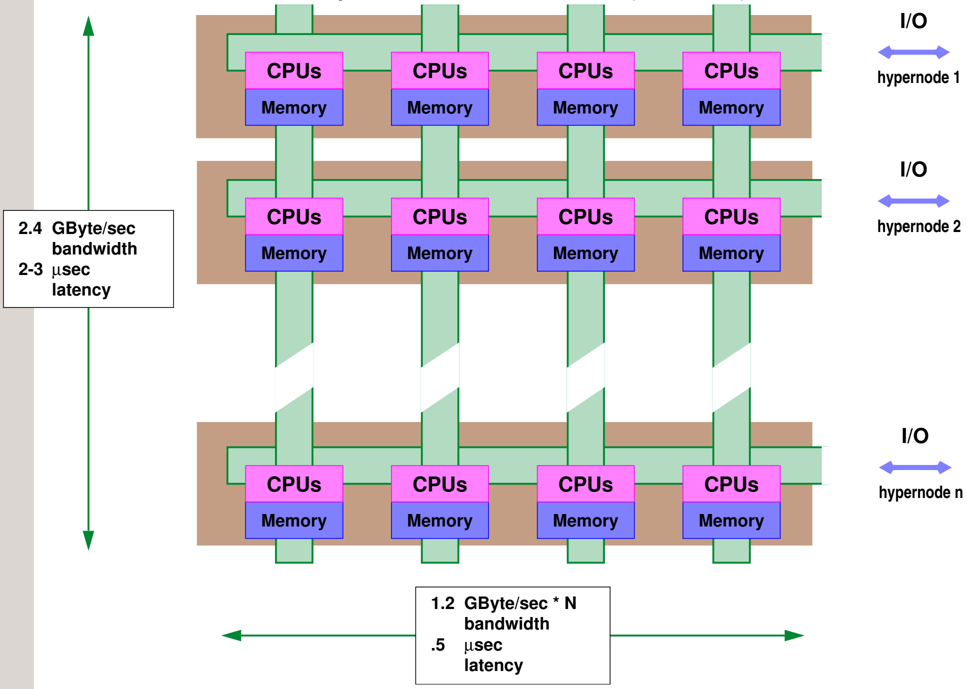
Parallele MIMD - Systeme

- Convex SPP1600 - Hypernode Architektur (RRZE)

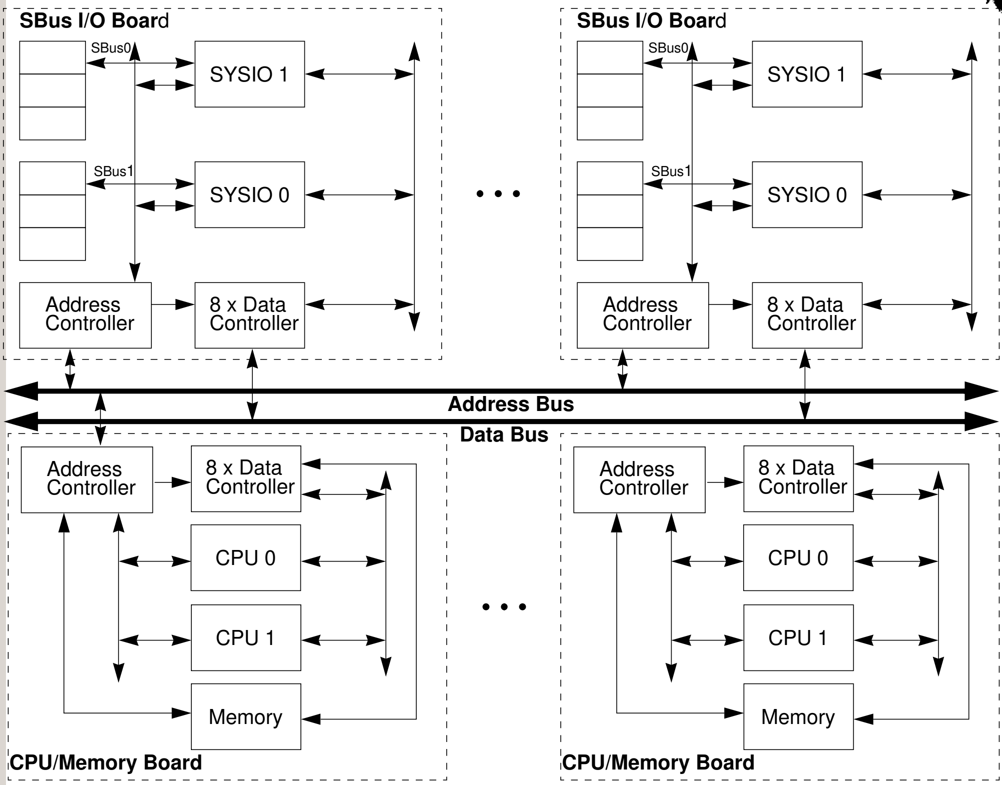


Parallele MIMD - Systeme

- Convex SPP1 - Speicherhierachie (RRZE)

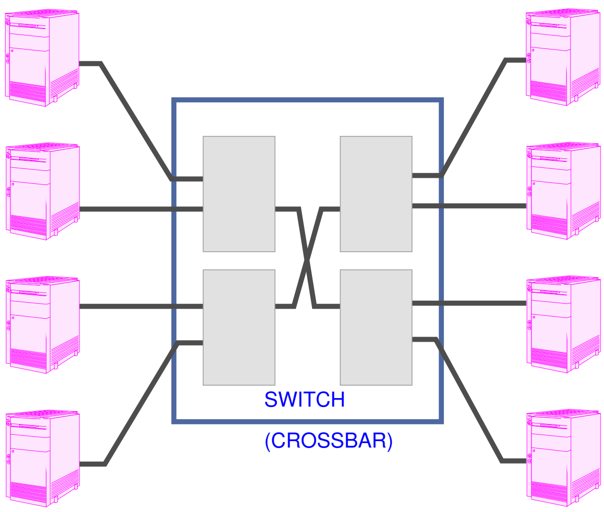


■ Sun-Enterprise-Server (IMMD, RRZE)



Parallele MIMD - Systeme

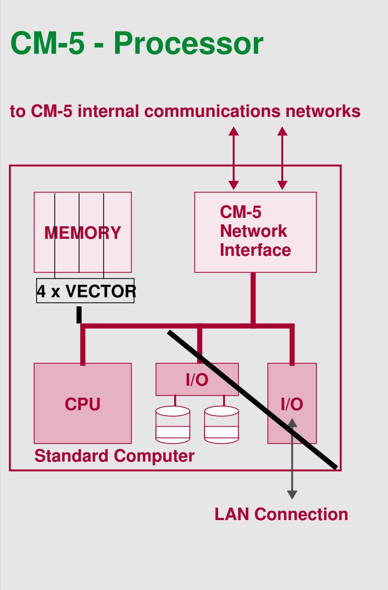
■ IBM SP-2 (max. 128 Prozessoren) am LRZ

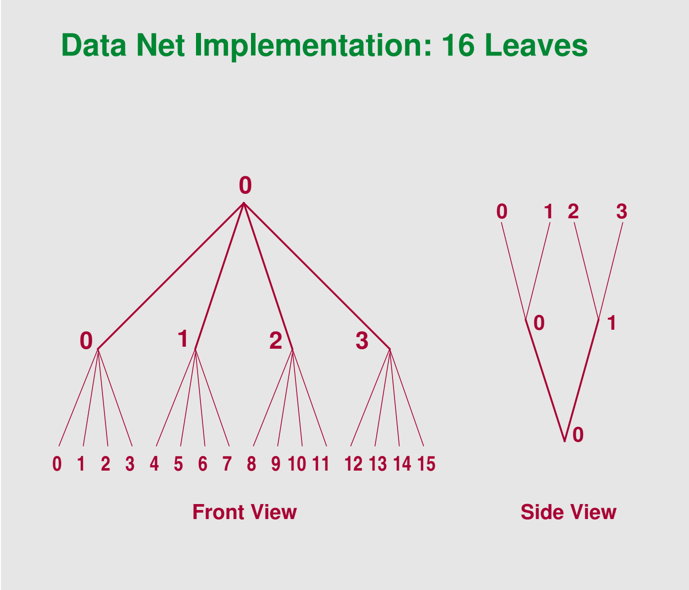


- 70 Knoten (9000/590)
- 14 dicke, 256MByte
- 56 dünne, 128MB, geringere Speicherbandbreite, kleinerer E/A-Ausbau, gleiche Peak-Leistung

Parallele MIMD - Systeme

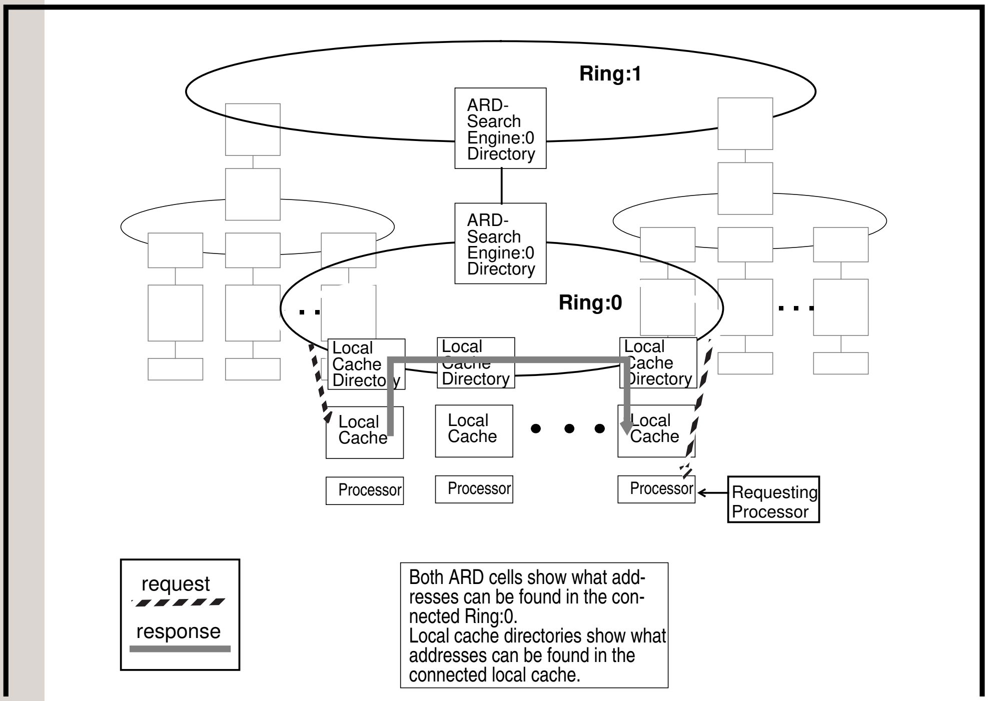
- Thinking Machines CM-5 (IMMD)





Parallele MIMD - Systeme

- KSR (IMMD)



1.6 Message Passing vs. Shared Memory

- Message Passing Architekturen
  - Programmiermodell/Kommunikationsmodell ist 'message passing'
  - Kein globaler Adreßraum; also NORMA-Architekturen
    NORMA = NO Random Memory Access
  - Das Verhältnis Befehlsausführungszeit/Latenzzeit zwischen Aktivitätsträgern(Threads, Prozesse) auf verschiedene Prozessoren liegt bestenfalls bei 1 : 10.000 (IBM SP-2);
    Hinzu kommt ein erheblicher Protokoll - Overhead
  - Eine 'shared memory'-Programmiermodell kann als 'shared distributed memory' simuliert werden, ist aber extrem ineffizient. ('paging' über 'message passing'); Beispiel: Intel Paragon

Message Passing vs. Shared Memory

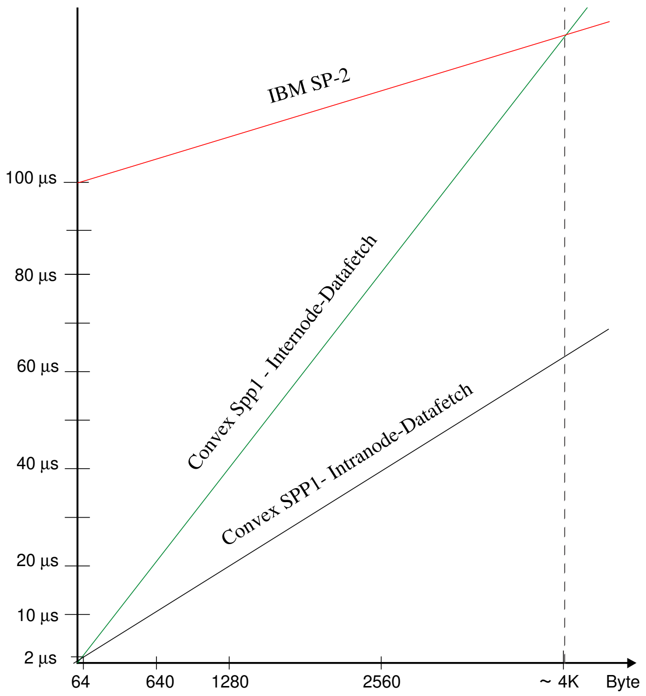
- Shared Memory Architekturen
  - Programmiermodell/Kommunikationsmodell ist 'shared memory'
  - Globaler Adreßraum; also UMA/NUMA-Architekturen
    UMA = Uniform Memory Access
    NUMA = Non Uniform Memory Access
  - Das Verhältnis Befehlsausführungszeit/Latenzzeit zwischen Aktivitätsträgern(Threads, Prozesse) auf verschiedene Prozessoren reduziert sich auf Speicherzugriffe, bei NUMA-Architekturen hängt die Zeit von der "Entfernung" ab.
    Beispiel Convex SPP1600: 1 : 1 Cache
    1 : 40 Nodespeicher
    1 : 200 Internodespeicher
  - Ein 'message passing'-Programmiermodell läßt sich effizient auf einem 'shared memory'-System simulieren. ('message passing' über 'shared data'-Segmente)

Message Passing vs. Shared Memory

- Schlußfolgerung:
  - Grobgranulare Parallelität ist geeignet für beide Architekturklassen
  - Feingranulare Parallelität ist nur geeignet für 'shared memory'-Systeme
- Wann ist ein Programm grob-/feingranular?
  - Versuch einer Quantifizierung am Beispiel
    IBM SP-2/Convex SPP1600:

Message Passing vs. Shared Memory

- Qualitativer Vergleich
- IBM SP-2[1]
  per Prozessor:
  40 µs Latenz +
  60 µs Protokoll-Overhead
  Transfereinheit "beliebig" groß
- Convex SPP1600
  per Node = 8 Prozessoren:
  2 µs Latenz +
  0 µs Protokoll-Overhead
  Transferblock 64 Byte



1.Abschätzung, nicht verifiziert

1.7 Historisches

- Viele grundliegende Techniken sind älter als heute im allgemeinen angenommen wird.
  Hier einige Beispiele:
  - 1953 Bitparallele Rechnenwerke (IBM 701)
  - 1958 CPU/Channel Overlap (Autarke E/A-Kanäle) (IBM 704)
  - 1959 Instruction Look Ahead (IBM Stretch)
  - 1959 Memory Interleaving (IBM Stretch)
  - 1961 Pipelining (Feranti Atlas)
  - 1961 Cache - Speicher (Feranti Atlas)
  - 1961 Virtueller Speicher (Feranti Atlas)
  - ....

1.8 Leistungszahlen

- nirgends wird mehr mit fragwürdigen Zahlen operiert
- Peak-Leistung ist die theoretisch obere Schranke an Prozessorleistung und ist bestenfalls ein Leistungsmaß neben so wichtigen Daten wie:
  - Reale Parallelität der FUs des Prozessors selbst (Instruction Scheduling)
  - Cachegrößen, -technik und -breite
  - Speicherbandbreiten
  - Speicherlatenzzeiten
  - Kommunikationsoverhead; entfernter Zugriff
  - Platten-I/O-Leistungsdaten
- Generell gilt: Je weiter Taktraten und Peakleistung auseinanderklaffen, desto höher ist die theoretisch nutzbare “innere” Parallelität, aber auch desto geringer ist die tatsächliche “Ausbeute” an Leistung.

Leistungszahlen

- Weitere Leistungsmaße versuchen mehr eine Gesamteindruck zu vermitteln, sie geben jedoch immer nur spezifische Teilaspekte wieder. Beispiele:
  - Wheatstone
  - Drystone
  - LINPACK
  - LAPACK
  - Specmarks
  - Xmarks
  - Transaction_marks?
- Amerikaner lieben 'rankinglists'; egal welchen Aussagewert sie haben; beliebt ist die TOP 500 Liste
- Es soll Firmen geben, die Prozessoren (und Systeme) z. B. auf den Linpack-Test optimieren.

1.9 Leistungsausbeute

- **Gute Leistungsausbeute bedeutet bei Vektorrechnern > 50% und bei Parallelrechnern (auf der Basis von Riskprozessoren) > 10% der Peakleistung.**
- Die schlechte Leistungsausbeute bei Risk-Parallelrechnern ist
  - in der schlechten Ausnutzung der “Prozessorparallelität” (Abschnitt 1.1) und “Speichersystemparallelität” (Abschnitt 1.2) durch
    - Datenabhängigkeit,
    - mangelnde Compilerleistung, inhärente Parallelität zu erkennen und auf die komplexen Hardwaremechnismen hin zu optimieren. (parallele Datenströme, Prefetching, Instruction Reordering)
    - einen nicht adäquaten(den Compiler unterstützenden) Programmierstil,
    - mangelnde inhärente Parallelität des zu lösenden Problems und
    - mangelnde Optimierung der Betriebssystemstrategien für numerische Anwendungen.

Leistungsausbeute

- Die schlechte Leistungsausbeute bei Parallelrechnern ist insbesondere auch im mangelnden Parallelitätsgrad der Anwendung selbst begründet.
- *Speedup* (im Vergleich zum Ablauf des gleichen Programms auf einem Prozessor)

$$s(p) = \frac{t(1)}{t(p)}$$

- Effizienz (im Vergleich zum Ablauf des gleichen Programms auf einem Prozessor)

$$e(p) = \frac{t(1) \times 100}{p \times t(p)}[Prozent]$$

- Eine andere Variante der Betrachtung wäre den *Speedup* bzw. die Effizienz durch "Vergrößerung" des Problems zu bestimmen.

Leistungsausbeute

- Amdahls Gesetz

$$t(1) = t_s + (1 - t_s) = t_s + t_p$$

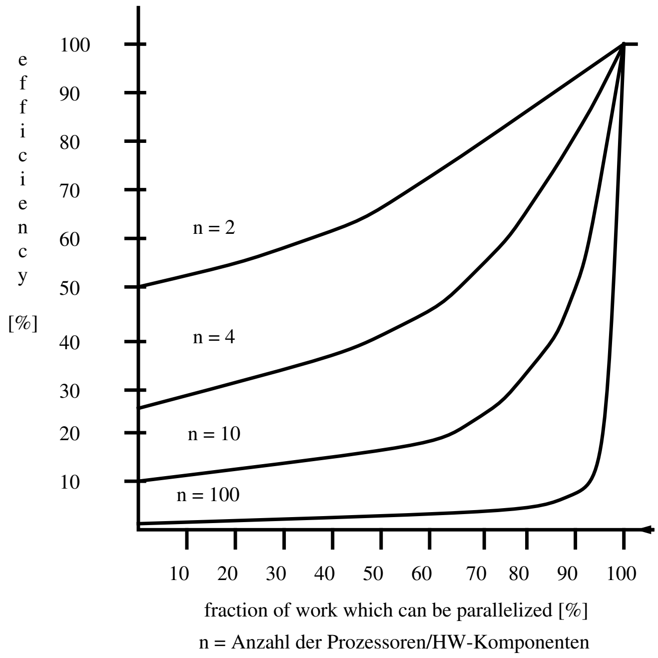
$$t(p) = t_s + \frac{1 - t_s}{p}$$

$$s(p) = \frac{1}{t_s + \frac{1 - t_s}{p}} \leq \frac{1}{t_s}$$

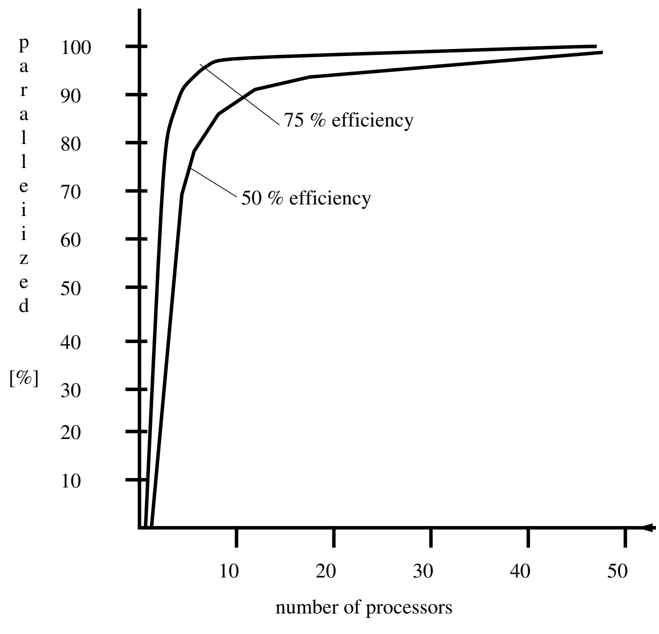
d.h. der maximale Speedup bzw. die max. Effiziens sind durch den sequentiellen Anteil (wesentlich) beschränkt.

- Der algorithmische Mehr-/Minderaufwand der durch die Parallelisierung des Problems selbst entsteht, wird in der Regel nicht berücksichtigt.

Leistungsausbeute



Leistungsausbeute



1.10 Programmiermodelle

- Shared Memory Programmiermodell
- Message Passing Programmiermodell
- Datenparalleles Programmiermodell oder
  Single Programm Multiple Date - SPMD
  - Shared Memory
  - Message Passing
- Parameterparalleles Programmiermodell
- Unabhängig vom Programmiermodell muß man beim Programmieren Paralleler Systeme ein Optimum folgender Eigenschaften erreichen:
  - Minimierungen der Kommunikation
  - Maximierung des “lokalen” Verhaltens
  - Verteilung der Daten so, daß die Rechenlast auf allen Prozessoren gleich ist (loadbalancing).

1.10.1 Shared Memory Programmiermodell

- Kommunikation über gemeinsame Speicherbereiche
- Explizite Koordinierung
- Aktivitätsträger sind Prozesse und/oder Threads
- Keine Unterstützung durch spezielle Sprachkonstrukte
- Keine “automatische” Parallelisierung
- Unterstützung durch Thread-Libraries
- Erzwingung von Prozessorzuordnungen nicht immer gegeben
- Scheduling des Betriebssystems unterstützt in der Regel Lokalitätsverhalten
- Geeignet insbesondere für UMA- und NUMA-Architekturen
- Feingranulare Parallelität möglich
- Programmiersprachen: C; C++
- Schwierige Fehlersuche
- Wenig Werkzeuge für paralleles Debuggen
- Geeignet, hoch effiziente (architekturspezifische) Programme zu schreiben

1.10.2 Message Passing Programmiermodell

- Kommunikation über *sent/recieve*-Mechanismen
- Explizite Koordinierung; implizit realisiert über *sent/recieve*-Mechanismen
- Aktivitätsträger sind Prozesse (lokal sind Threads möglich)
- Keine Unterstützung durch spezielle Sprachkonstrukte
- Keine “automatische” Parallelisierung
- Unterstützung durch Message Passing-Libraries: (PVM; MPI)
- Feste Zuordnung Prozeß/Prozessor, keine Migration von Prozessen möglich
- Geeignet für alle Rechnerarchitekturklassen
- Geeignet für grobgranulare Parallelität; Grad ist abhängig von verwendeteter Rechnerarchitektur (*Latency* des Verbindungsnetzwerks).
- Programmiersprachen: C; C++; Fortran
- Schwierige Fehlersuche, wenig Werkzeuge für paralleles Debuggen
- Geeignet portable Programme für alle denkbaren Architekturen zu schreiben, wobei die Effizienz stark von der rechnerarchitektur-spezifischen, sinnvollen parallelen Granularität abhängt.

1.10.3 Datenparalleles Programmiermodell

- Shared Memory - Single Programm Multiple Date - SPMD
  - Implizite Kommunikation, realisiert über gemeinsame Speicherbereiche
  - Implizite Koordinierung
  - Aktivitätsträger sind Prozesse mit identischer Kontrollflußstruktur; dynamisch verschiedener Kontrollfluß möglich.
  - Unterstützung der parallelen Programmierung durch spezielle Sprachkonstrukte; typischer Weise “Feldverarbeitung”.
  - Automatische Parallelisierung; Unterstützungsmöglichkeit durch Compilerdirektiven
  - Erzwingung von Prozessorzuordnungen nicht immer gegeben
  - Scheduling des Betriebssystems unterstützt in der Regel Lokalitätsverhalten.

Datenparalleles Programmiermodell

- Shared Memory - Single Programm Multiple Date - SPMD (cont)
  - Geeignet insbesondere für UMA- und NUMA-Architekturen
  - Feingranulare Parallelität möglich
  - Programmiersprachen: Fortran90/95;(HPF)
  - Einfachere Fehlersuche
  - Parallele Debugger einfacher realisierbar; da nur **ein** Kontrollfluß
  - Geeignet, portable Programme für UMA - und NUMA - Architekturen und potentiell auch für Vektorrechner zu schreiben, wobei die Effizienz vom maschinenspezifischen Granularitätsgrad und dem Lokalitätsverhalten, insbesondere bei NUMA-Maschinen, abhängt.

Datenparalleles Programmiermodell

- Message Passing - Single Programm Multiple Date - SPMD
  - Implizite Kommunikation, realisiert über *sent/recieve*-Mechanismen
  - Implizite Koordinierung; realisiert über *sent/recieve*-Mechanismen
  - Aktivitätsträger sind Prozesse mit identischer Kontrollflußstruktur; dynamisch verschiedener Kontrollfluß möglich
  - Unterstützung der parallelen Programmierung durch spezielle Sprachkonstrukte; typischer Weise “Feldverarbeitung”.
  - Automatische Parallelisierung; Unterstützungsmöglichkeit durch Compilerdirektiven
  - Feste Zuordnung Prozeß/Prozessor
  - Keine Migration von Prozessen möglich (und sinnvoll).

Datenparalleles Programmiermodell

- Message Passing - Single Programm Multiple Date - SPMD (cont)
  - Geeignet für alle Rechnerarchitekturklassen
  - Geeignet für grobgranulare Parallelität; Grad ist abhängig von verwendeter Rechnerarchitektur.
  - Programmiersprachen: High Performance Fortran - HPF
  - Einfachere Fehlersuche
  - Parallele Debugger einfacher realisierbar; da nur **ein** Kontrollfluß
  - Geeignet portable Programme für alle denkbaren Architekturen zu schreiben, wobei die Effizienz stark von der rechnerarchitektur-spezifischen, sinnvollen parallelen Granularität abhängt.

1.10.4 Parameterparalleles Programmiermodell

- Keine dynamische Kommunikation zwischen den Komponenten des parallelen Systems.
- Master/Slave - Konfiguration, wobei der Masterprozeß an die sonst unabhängigen Slaveprozesse Parametersätze verteilt und ggf. die Ergebnisse wieder einsammelt und auswertet.
- Im einfachsten Fall ist der Masterprozeß eine Shellprozedur.

1.10.5 Zusammenfassung

	nutzbare Parallelität	Programmier -sprachen	Bibliotheken	Compiler-unterstütz-ung	Synchroni-sation	Daten-austausch
Vektor-rechner	Datenparallel	C, (C++) Fortran		automatische Vektorisierung	implizit	keiner
MIMD-NORMA	Datenparallel	HPF XOCL (Fujitsu)		Datenvertei-lung Feldverarbei-tung	implizit	implizit (sent/receive)
MIMD-NORMA	Datenparallel und Programm-parallel	C, (C++) Fortran	PVM MPI	keine	explizit	explizit (sent/receive)
MIMD-UMA/ NUMA	Datenparallel	Fortran90/95		Feldverarbei-tung	implizit	implizit
MIMD-UMA/ NUMA	Datenparallel und Programm-parallel	C, C++	p-threads (PVM) (MPI)	keine	explizit	explizit