Vorlesung

# Programmierung Paralleler Systeme

Wintersemester 1998/99 (10317)




## A.1 Inhalt

- Vorlesung
  - Architekturprinzipien und Programmiermodelle von Parallelrechnern
  - Effiziente Nutzung von Monoprozessoren (Cache, Speicher, E/A)
  - Threads und Koordinierung in Multiprozessoren mit gemeinsamem Speicher
  - Architektur und Programmierung von Rechnern mit verteiltem Speicher
  - Architektur und Programmierung von Vektorrechnern
- Übungen
  - Einsatz von Cache-, TLB- und Speicher-angepassten Datenstrukturen
  - Speicherabgebildete Dateien
  - **P**osix Threads (Pthreads)
  - **M**essage **P**assing **I**nterface(MPI)
  - Unterstützung der Vektorisierung durch Programmgestaltung und Direktiven




## A Organisatorisches

- Dozenten
  - Dr. F. Bellosa, IMMD IV, Frank.Bellosa@informatik.uni-erlangen.de
  - Dr. G. Wellein, RRZE, Gerhard.Wellein@rrze.uni-erlangen.de
- Vorlesung und Übungen
  - für Studierende der Fachrichtung Informatik im Hauptstudium, für Studierende der Ingenieur- und Naturwissenschaften im Hauptstudium
  - 4 anrechenbare Semesterwochenstunden 2 SWS Vorlesung, 2 SWS Übungen




## A.2 Vorlesung, Übung, Schein, Folien, URL

- Vorlesung: 15.3. - 19.3. und 22.3. - 26.3. von 9:00 bis 12:15 im 0.031
- Übungen: 15.3. - 19.3. und 22.3. - 25.3. von 13:00 bis 16:00 im 01.153
- Schein: nach einer mündliche Prüfung am 26.3.
- Folien: werden im WWW zur Verfügung gestellt.
- URL zur Vorlesung:
  - http://www4.informatik.uni-erlangen.de/Lehre/WS98/V_PPS/
  - hier findet man Folien zum Ausdrucken und Zusatzinformationen

# B Einführung

## B.1 Programmierung paralleler Systeme, warum?

- Nutzung von *mehr* Hardware zur Lösung eines Problems
  - ◆ Nutzung brachliegender Ressourcen
  - ◆ Anschaffung von spezieller Hardware für das parallele Rechnen
- Potentielle Erweiterbarkeit der Hardware bei wachsender Problemgröße
- Abbildung der nebenläufigen Problemstellung auf eine parallele Lösung (z.B. verteilte Anlagensteuerung, Telefonnetz)
- Verknüpfen bestehender sequentieller Komponenten zu einem parallelen System
- Zuverlässigkeit durch Replikation




## B.2 Ebenen der Parallelität

- Prozessorebene
- Speicherebene
- Systemebene, Verbindungsnetz
- Netzwerkebene
- E/A-Ebene (nicht Teil dieser Vorlesung)

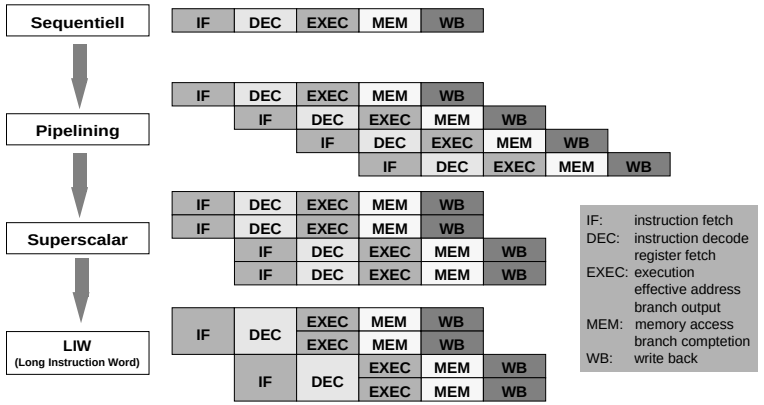



# 1 Prozessorebene

- Parallelisierung des Befehlsstroms
  - ◆ Sequentiell:
    Die einzelnen Phasen einer Befehlsausführung werden sequentiell abgearbeitet. Ein Instruktionsstrom.
  - ◆ Pipelining:
    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 (Very Large Instruction Word):
    “Breite” Instruktions-Verarbeitung:
    Mehrere Befehle eine parallelen Instruktionsstroms werden bearbeitet.

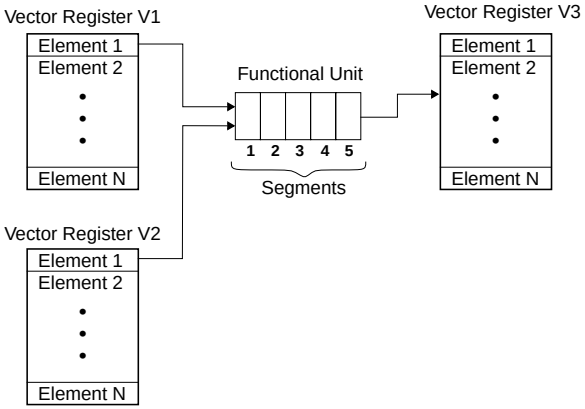



# 1 Prozessorebene (2)



# 1 Prozessorebene (3)

- Parallelisierung des Datenstroms
  - Vektoreinheiten

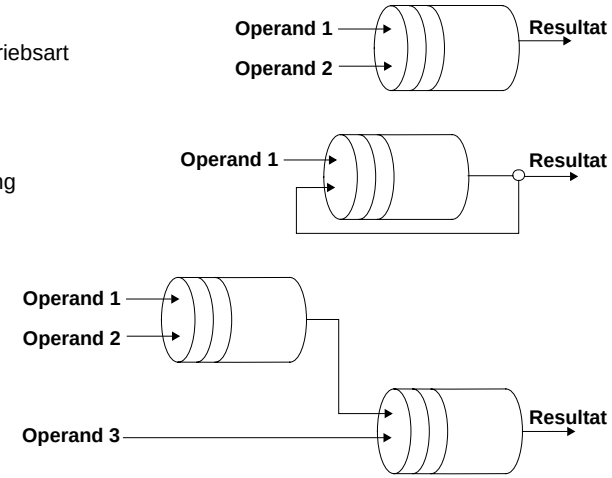






# 1 Prozessorebene (4)

- Vektorverarbeitung in Kombination mit “Instruktionsparallelität”
  - normale Betriebsart
  - Rückkopplung
  - Verkettung (Chaining)

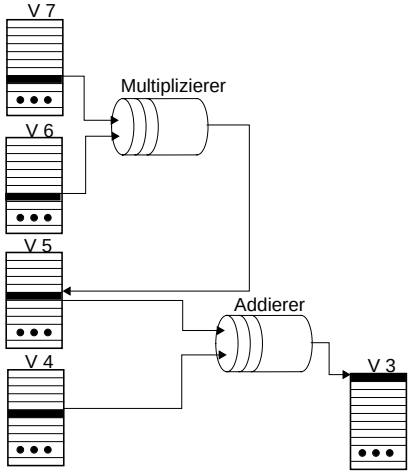






# 1 Prozessorebene (5)

- Chaining zwischen Registern

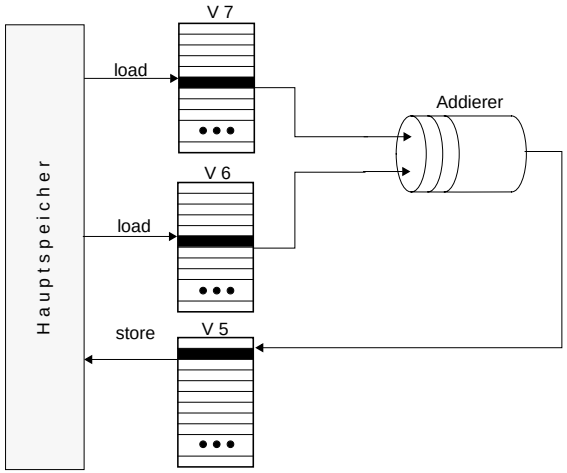






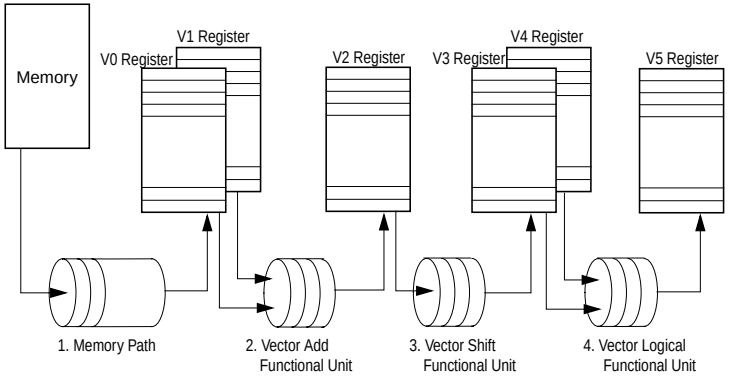
# 1 Prozessorebene (6)

- Chaining zum Speicher



# 1 Prozessorebene (7)

◆ Beispiel:

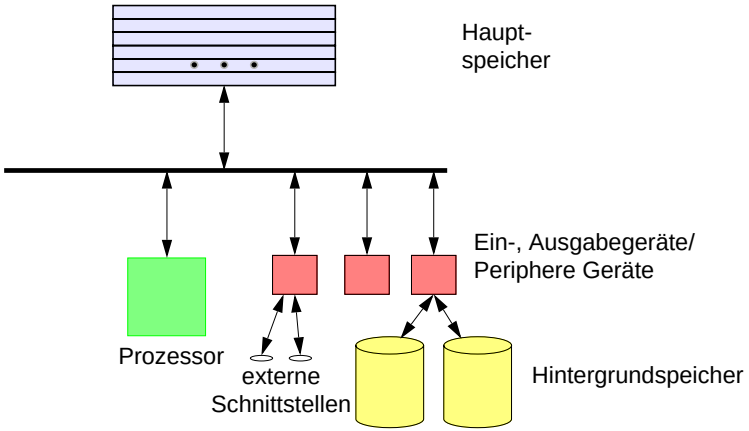






# 2 Speicherebene

■ Hauptspeicher mit einer Speicherbank

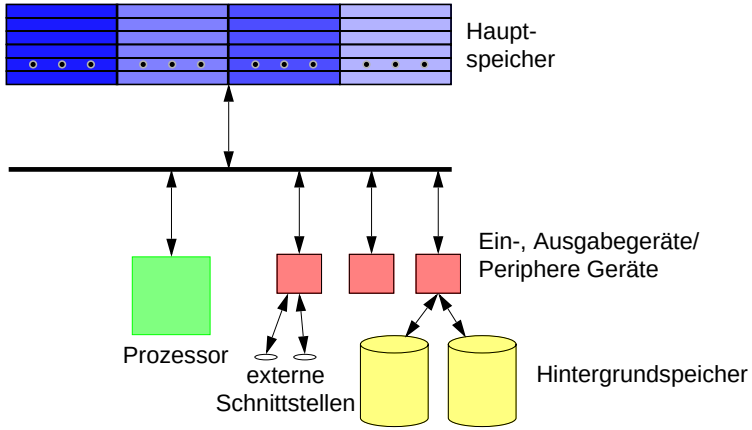






# 2 Speicherebene (2)

■ Parallelisierung des Zugriffs durch Speicherbänke

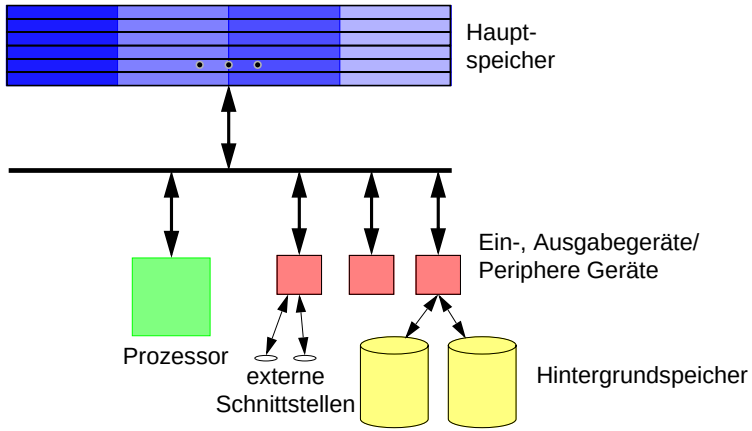






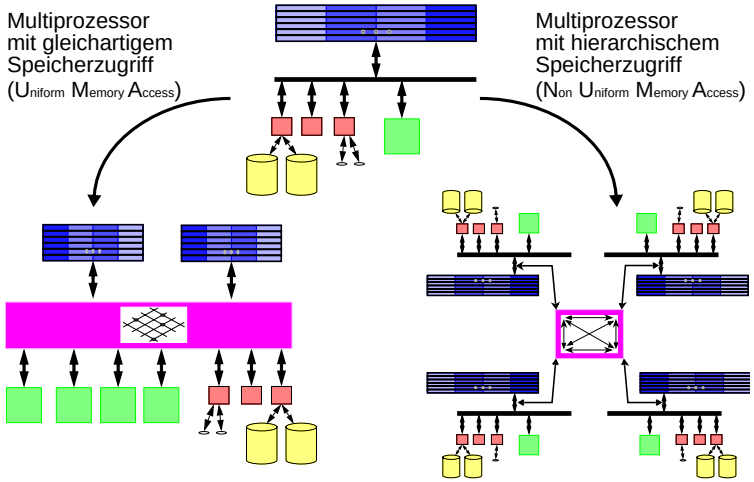
# 2 Speicherebene (3)

■ Parallelisierung des sequentiellen Zugriffs durch Interleaving



# 3 Systemebene

- Multiprozessoren mit globalem Adreßraum

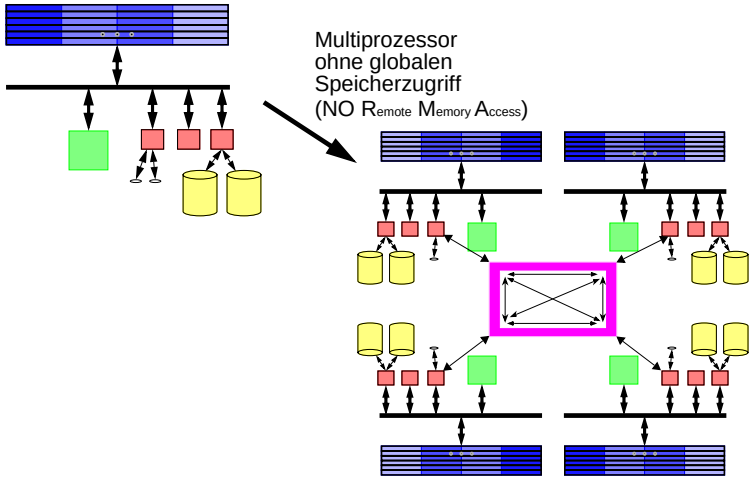






# 3 Systemebene (2)

- Multiprozessor mit lokalem Adreßraum







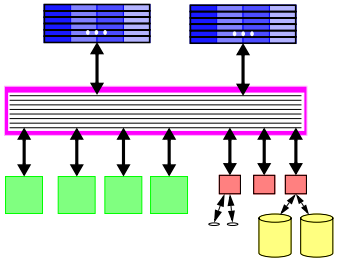
# 3 Systemebene (3)

- Verbindungsnetzwerke für Multiprozessoren
  - ◆ Wunsch: Vollständiges Verbindungsnetzwerk; Komplexität steigt mit K!
  - ◆ Abhilfe:
    - Einschränkung der Nachbarschaftsbeziehung
    - globale Kommunikation wird mehrstufig mit - möglichst autarken, nicht die Rechenleistung einzelner Knoten belastenden - Routern realisiert.
  - ◆ Kriterien der Bewertung von statischen Verbindungsnetzwerken
    - Anzahl der direkt erreichbaren Nachbarknoten (Links)
    - Anzahl der Zwischenstufen die erforderlich sind um den am weitesten entfernten Knoten zu erreichen (Durchmesser).
    - Anzahl der Knoten in der nächsten Ausbaustufe (Ausbauinkrement)
    - Anzahl der Verbindungssegmente (Komplexität)




# 3 Systemebene (4)

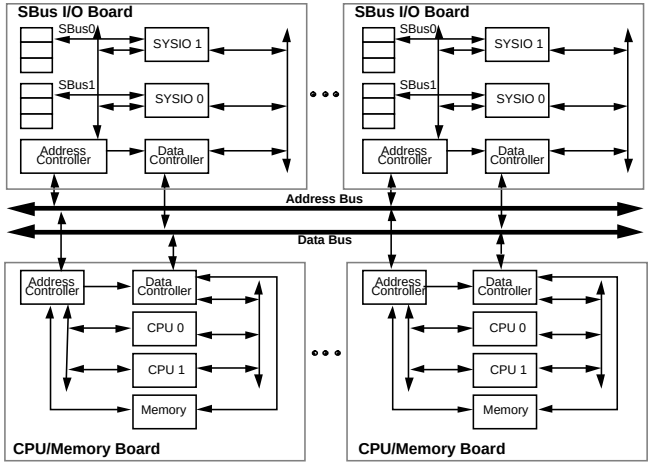
- Bussysteme (meist in UMA Multiprozessoren).



  - Maximal erreichbare Prozessorzahl liegt heute bei 24 Prozessoren.
  - Parallelisierung durch Aufspaltung in Address- und Datenbus
  - Latenzverbergung durch Transaktionskonzept
  - Bustakt kann nur durch Verkürzung des Busses gesteigert werden (-> Centerplane Ansatz, maximaler Bustakt derzeit bei ca. 100 Mhz).
  - Beispiel: SUN Enterprise 3000/4000/6000 Server

# 3 Systemebene (5)

- Architektur SUN Enterprise X000

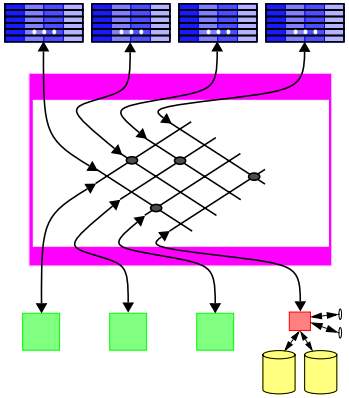






# 3 Systemebene (6)

- Kreuzschiene=Crossbar in UMA Multiprozessoren

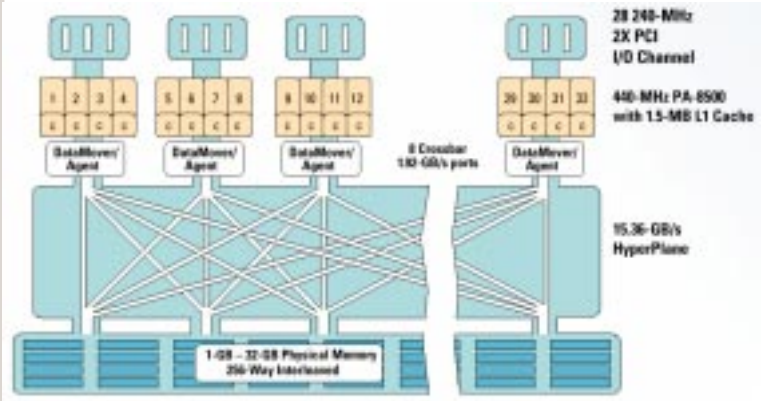


- Nichtblockierende Verbindung zum Hauptspeicher
- Hohe Bandbreite
- Hardware-Komplexität steigt quadratisch
- Maximal erreichbare Prozessorzahl liegt heute bei 32 Prozessoren
- Beispiel: HP V2500 Klasse




# 3 Systemebene (7)

- Architektur HP V2500 Class



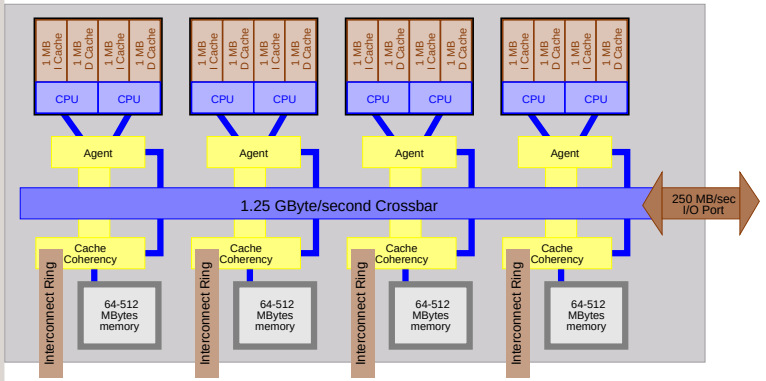




# 3 Systemebene (8)

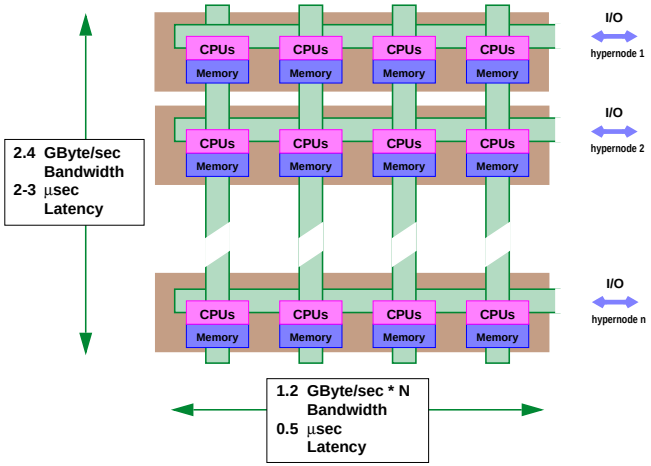
- Multiprozessoren mit hierarchischem Speicherzugriff (NUMA)

Convex SPP1600 - Hypernode Architektur (RRZE)



# 3 Systemebene (9)

- Convex SPP1 - Speicherhierachie

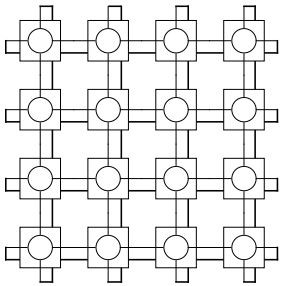






# 3 Systemebene (10)

- 2D (*3D*) Torus

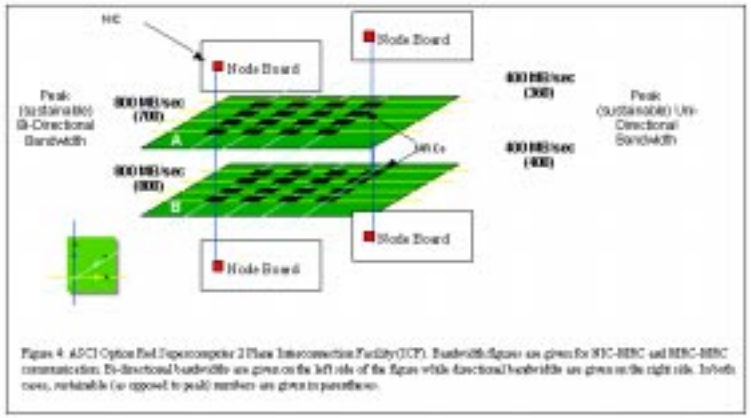


- ◆ Links pro Knoten: 4 (*6*)
- ◆ Anzahl der Knoten: K = $q^{2(3)}$; q = Kantenlänge ; Komplexität: A= $2*q^2$ ($3*q^3$)
- ◆ Ausbauinkrement: I = 2(q + 1) - 1 ($3q^2 + 3q + 1$)
- ◆ Max. Routingstufen: D = $2 \cdot \lfloor q/2 \rfloor$ ; ($3 \cdot \lfloor q/2 \rfloor$)
- ◆ Typische Vertreter: Intel ASCI Red (9152 PPro), *Cray T3E (2048 Alpha)*




# 3 Systemebene (11)

- Beispiel NORMA MP: Intel ASCI (**A**ccelerated **S**trategic **C**omputing **I**nitiative) Red

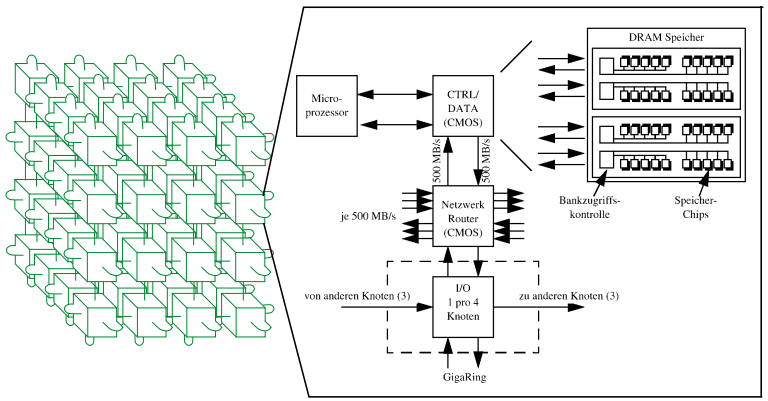






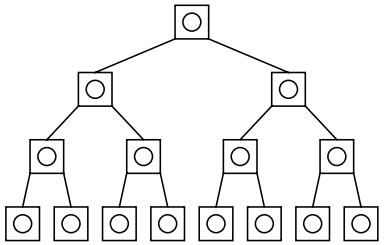
# 3 Systemebene (12)

- Beispiel NUMA MP: Cray T3E mit DEC Alpha 21164 CPU (600 MHz)



## 3 Systemebene (13)

- Binärbaum

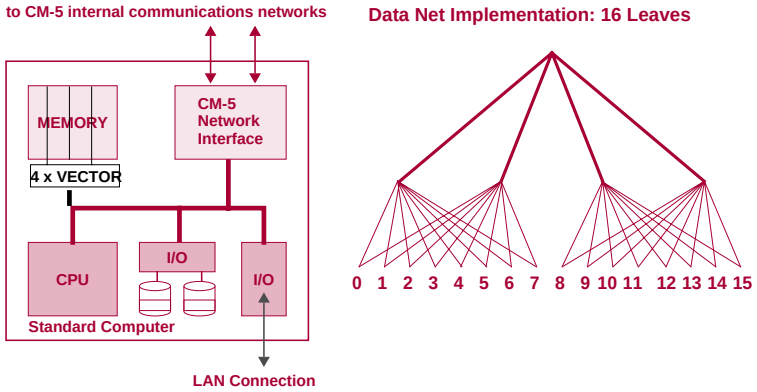


  - Links pro Knoten: 3;
  - Anzahl der Knoten: $K = 2^d - 1$; d = Ebenenzahl
  - Ausbauinkrement: I = K + 1;
  - Max. Routingstufen: D = 2(ld (K+1)-1) = 2(d - 1)
  - Vermeidung des Kommunikationsengpasses zur Wurzel hin: Fat-Tree
  - Typischer Vertreter: Thinking Maschines CM5




## 3 Systemebene (14)

- Beispiel NORMA MP: Thinking Machines CM-5 am IMMD



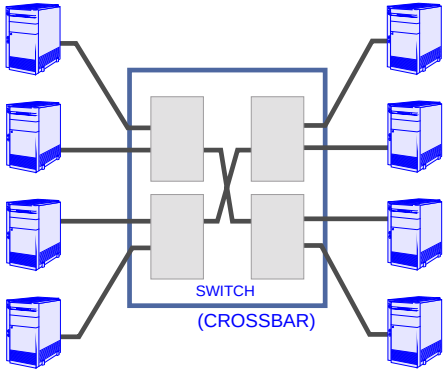







## 3 Systemebene (15)

- Verbindungnetz mit zentraler Paketvermittlung
  - Beispiel NORMA MP: IBM SP-2 (max. 128 Prozessoren) am LRZ München




- 70 Knoten IBM 9000/590 (66.7 MHz, 267 MFlop/s)
- 4 dicke, 256MByte
- 56 dünne, 128MB, geringere Speicherbandbreite, kleinerer E/A-Ausbau, gleiche Peak-Leistung
- 300 µsec Latenz (IP)
- 40 µsec Latenz(IBM Prot.)
- 12 MB/s Pt. - Pt. Bandbr. (IP)
- 35 MB/s Pt. - Pt. Bandbr.




## 4 Netzwerkebene

- Verbindung von mehreren Rechnern mit IP-Technologie zu einem Cluster
- Keine Spezialhardware erforderlich
- Die LAN-Technologie nähert sich immer mehr dem Spezialnetzwerk an
  - Myri-Net ca. 300 MBit/s
  - ATM 622 MBit/s
  - Ethernet 1 Gbit/s

# B.3 Klassifikation

- nach Flynn

| Data - Stream / Instruction - Stream | SINGLE | MULTIPLE |
|---|---|---|
| SINGLE | **SISD**<br>Klass. von Neumann-Architektur (Monoprozessor) | **SIMD**<br>Vektor - Array - Prozessoren |
| MULTIPLE | **MISD**<br>nicht sinnvoll | **MIMD**<br>Parallelrechner<br>- aus SISD - Prozessoren<br>- aus SIMD - Prozessoren |



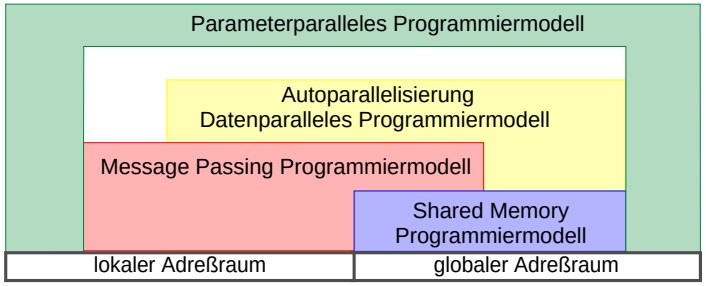

# B.3 Klassifikation (2)

- **Typische MIMD Vertreter**

| | | |
|---|---|---|
| verteilte Speicherorganisation | NORMA:<br>• Workstation-/Server-Cluster<br>• IBM SP2<br>• Intel ASCI Red<br>• Fujitsu VPP (?) | ccNUMA:<br>• Convex SPP<br>• SGI Origin<br>• Sequent NumaQ<br>NUMA:<br>• CRAY T3E |
| zentrale Speicherorganisation | | UMA:<br>• SUN Enterprise X000<br>• HP V-Class<br>• DEC AlphaServer<br>• IBM RS/6000 S70<br>• NEX SX/4 / CRAY T90 |
| | lokaler Adreßraum | globaler Adreßraum |




# B.4 Programmiermodelle



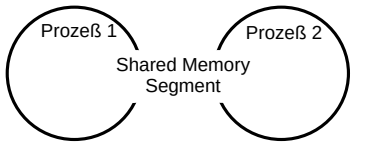

- Unabhängig vom Programmiermodell muß man beim Programmieren paralleler Systeme ein Optimum folgender Eigenschaften erreichen:
  - ◆ Minimierungen der Kommunikation
  - ◆ Maximierung des “lokalen” Verhaltens
  - ◆ Verteilung der Rechenlast auf alle Prozessoren (loadbalancing)




# B.4 Programmiermodelle

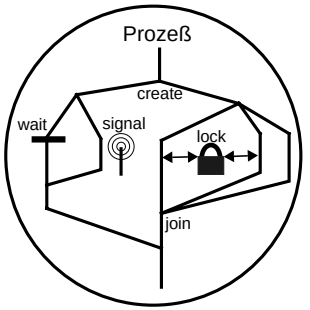
- Shared Memory Programmiermodell
  - ◆ Geeignet für UMA- und NUMA-Architekturen
  - ◆ Kommunikation über gemeinsame Speicherbereiche
  - ◆ Explizite Koordinierung
  - ◆ Aktivitätsträger sind Prozesse und/oder Threads

Parallele Prozesse




```
shm_id = shm_get(Schlüssel,Größe,Zugriffsrechte)
ptr = shm_at(shm_id)          shm_dt(prt)
semaphore = sema_create(Schlüssel,Zugriffsrechte)
wait(semaphore)               signal(semaphore)
```

1 Prozeß + Parallele Threads

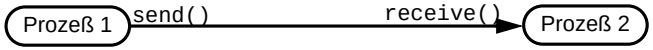


- Shared Memory Programmiermodell (2)
  - Keine Unterstützung durch spezielle Sprachkonstrukte
  - Keine “automatische” Parallelisierung
  - Unterstützung durch Thread-Libraries (Pthreads-Library)
  - Prozessorzuordnungen ist meist beinflußbar
  - Betriebssystem/Thread-Library sorgt für Lastverteilung
  - Feingranulare Parallelität möglich
  - Programmiersprachen: C; C++
  - Geeignet, hocheffiziente Programme zu schreiben






- Message Passing Programmiermodell
  - Geeignet für  alle MIMD Architekturen
  - Kommunikation über *sent/receive*-Mechanismen




  - Explizite Koordinierung; implizit realisiert über *sent/receive*-Mechanismen
  - Aktivitätsträger sind Prozesse (lokal sind Threads möglich)
  - Keine Unterstützung durch spezielle Sprachkonstrukte
  - Keine “automatische” Parallelisierung
  - Unterstützung durch Message Passing-Libraries: (PVM; MPI)
  - NORMA: Feste Zuordnung Prozeß/Prozessor; Prozeßmigration unmöglich
  - Geeignet für grobgranulare Parallelität; Grad ist abhängig von verwendeter Rechnerarchitektur (*Latenz/Bandbreite* des Verbindungsnetzwerks).
  - Programmiersprachen: C; C++; Fortran
  - Geeignet portable Programme für alle denkbaren Architekturen zu schreiben






- Shared Memory - Autoparallelisierung/Datenparallele Programmierung
  - Geeignet für alle UMA/NUMA Architekturen
  - Implizite Kommunikation, realisiert über gemeinsame Speicherbereiche
  - Implizite Koordinierung
  - Aktivitätsträger sind Prozesse/Threads mit identischer Kontrollflußstruktur; dynamisch verschiedener Kontrollfluß möglich.
  - Unterstützung der parallelen Programmierung durch spezielle Sprachkonstrukte; typischer Weise “Feldverarbeitung”
  - Automatische Parallelisierung potentiell nebenläufiger Programmteile (z.B. Schleifen); Unterstützungsmöglichkeit durch Compilerdirektiven
  - Feingranulare Parallelität möglich
  - Nur ein Teil der Nebenläufigkeit des Lösungsverfahrens ist nutzbar.
  - Programmiersprachen: Fortran90/95, C, C++
  - Einfachere Fehlersuche, da Koordinierungsfehler nicht möglich sind.






- Message Passing - Autoparallelisierung/Datenparallele Programmierung
  - Geeignet für alle MIMD Architekturen
  - Implizite Kommunikation/Koordination über *sent/receive* Mechanismen
  - Aktivitätsträger sind Prozesse mit identischer Kontrollflußstruktur; dynamisch verschiedener Kontrollfluß möglich
  - Unterstützung der parallelen Programmierung durch spezielle Sprachkonstrukte; typischer Weise “Feldverarbeitung”.
  - Automatische Parallelisierung potentiell nebenläufiger Programmteile (z.B. Schleifen); Unterstützungsmöglichkeit durch Compilerdirektiven
  - Geeignet für grobgranulare Parallelität; Grad ist abhängig von verwendeter Rechnerarchitektur (*Latenz/Bandbreite* des Verbindungsnetzwerks).
  - Nur ein Teil der Nebenläufigkeit des Lösungsverfahrens ist nutzbar.
  - Programmiersprachen: High Performance Fortran - HPF
  - Einfachere Fehlersuche durch implizite Kommunikation/Koordination, jedoch komplexer Debugger zur Überwachung einer Vielzahl von Rechnern

## B.4 Programmiermodelle

- 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.




## B.5 Lokaler vs. Globaler Adreßraum

- NORMA Architekturen
  - Basis-Programmiermodel ist ’Message Passing’.
  - 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’ vom Betriebssystem über die Seitenadressierung simuliert werden, ist aber extrem ineffizient. (’paging’ über ’message passing’); Beispiel: Intel Paragon




## B.5 Lokaler vs. Globaler Adreßraum

- UMA/NUMA Architekturen
  - Basis-Programmiermodel ist ’Shared Memory’.
  - 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:

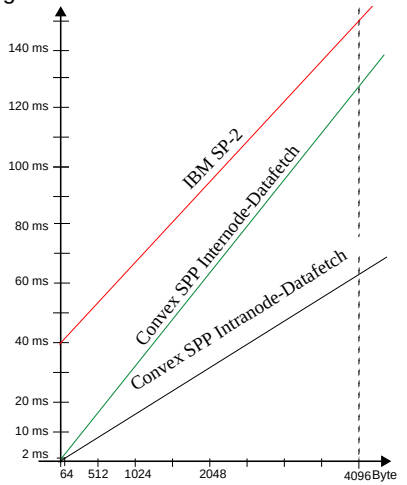
    | | |
    |---|---|
    | 1 : 1 | Register/Cache |
    | 1 : 40 | Nodespeicher (UMA) |
    | 1 : 200 | Internodespeicher (NUMA) |

  - Ein ’message passing’-Programmiermodell läßt sich effizient auf einem ’shared memory’-System simulieren.
    (’message passing’ über ’shared data’-Segmente)
- Schlußfolgerung:
  - Grobgranulare Parallelität ist geeignet für beide Architekturklassen
  - Feingranulare Parallelität ist nur geeignet für ’shared memory’-Systeme




## B.5 Lokaler vs. Globaler Adreßraum

- Wann ist ein Programm grob-/feingranular?
  - IBM SP-2
    - 40 µs Latenz
    - 35MB/s
    - Transfereinheit “beliebig”
  - Convex SPP1600
    - 0.5 µs Latenz (Intranode)
      1.2 GB/s Bandbreite
      Transferblock 32 Byte
    - 2 µs Latenz (Internode)
      2.4 GB/s Bandbreite
      Transferblock 64 Byte



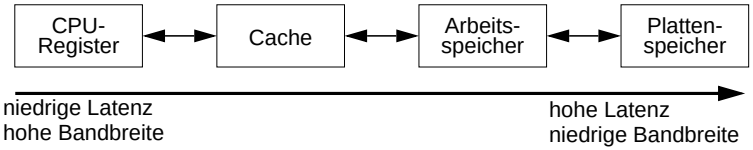
# B.6 Leistungsausbeute

- Peak-Leistung ist die theoretisch obere Schranke
- Viele Leistungsmaße geben jedoch immer nur spezifische Teilaspekte wieder. Beispiele:
  - SPECmarks
  - LINPACK
  - LAPACK
  - TPC
  - Beliebt sind 'rankinglists' (TOP 500) egal welchen Aussagewert sie haben.
- Sequentielle Leistung:
  - Wie gut wird die Leistung des Prozessors im sequentiellen Fall ausgenutzt?
- Parallele Leistung
  - Welche Leistung erhält man durch die Parallelisierung?




# 1 Grundlagen für Prozessoreffizienz

- Schneller Speicherzugriff durch Ausnutzung der Speicherhierarchie




  - Zeitliche Lokalität:
    Wenn ein Speicherort angesprochen wird, dann wird er innerhalb eines kleinen Zeitintervalls mehrfach angesprochen
  - Örtliche Lokalität:
    Wenn ein Speicherort angesprochen wird, dann werden innerhalb eines kleinen Zeitintervalls überwiegend Speicherorte mit geringem Adreßabstand angesprochen




# 1 Grundlagen für Prozessoreffizienz (2)

- Beispiel

```
int M = 10000;
int main()
{
 int i, j, k;
 int a[M][M], b[M][M], c[M][M];
 for (i = 0; i < M; i++)
    for (j = 0; j < M; j++) {
      a[i][j] = i; b[i][j] = j; c[i][j] = 0;
    }
 for (i = 0; i < M; i++)
     for (j = 0; j < M; j++)
         for (k = 0; k < M; k++)
             c[i][j] = c[i][j] + a[i][k] * b[k][j];
}
```

  - `c[i][j]` weist zeitliche Lokalität auf
  - `a[i][k]` weist örtliche Lokalität auf
  - `b[k][j]` weist keines von beidem auf




# 1 Grundlagen für Prozessoreffizienz (3)

- Ausnutzung der Prozessorparallelität
  - architekturspezifische Codegenerierung (Prozessortyp, Cachegröße)
  - geringe Datenabhängigkeit im Code
    (➜instruction reordering, ➜outstanding loads, ➜instruction scheduling)
  - verhersagbare Sprünge (➜branch prediction, ➜loop unrolling)
  - vorhersagbare Speicherzugriffe (➜prefetch operations)
- Ausnutzung optimierter Bibiotheken und Algorithmen
  - libc (allgemeine Bibliotheken z.B. **`memcpy`**(), **`bsort()`** )
  - NAG, BLAS, LAPACK (numerische Bibliotheken)
  - *Numerical Recipies*
  - ...

## 2 Leistungsausbeute durch Parallelisierung

- *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)

$$e(p) = \frac{t(1) \cdot 100}{t(p) \cdot p}\ \%$$

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

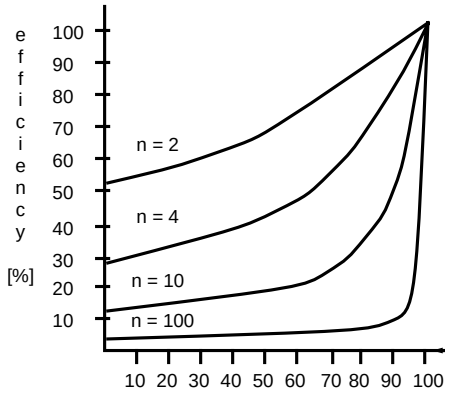



## 2 Leistungsausbeute durch Parallelisierung (2)

- Amdahls Gesetz
  - $(1) = t(1) \cdot (f_s + (1 - f_s)) = t_s + t_l$ $f_s$ = sequentieller Programmanteil $f_p$ = parallelisierbarer Programmanteil
  - $t(p) = t_s + \frac{t_p}{p}$
  - $(p) = \frac{1}{f_s + \frac{1 - f_s}{p}} \leq \frac{1}{f}$ d.h. der maximale Speedup bzw. die max. Effizienz sind durch den sequentiellen Anteil (wesentlich) beschränkt.
  - Der algorithmische Mehr-/Minderaufwand der durch die Parallelisierung des Problems selbst entsteht, wird bei diesen einfachen Abschätzungen nicht berücksichtigt, kann aber erheblich sein!
  - Bei der Parallelisierung kann die effektive Speicherbandbreite der Prozessoren durch die Ausnutzung der Caches ansteigen (Parallelität auf Speicherebene). Damit wird ein superlinearer Speedup möglich.




## 2 Leistungsausbeute durch Parallelisierung (3)




fraction of work which can be parallelized
n = number of processors/HW-components




## 2 Leistungsausbeute durch Parallelisierung (4)