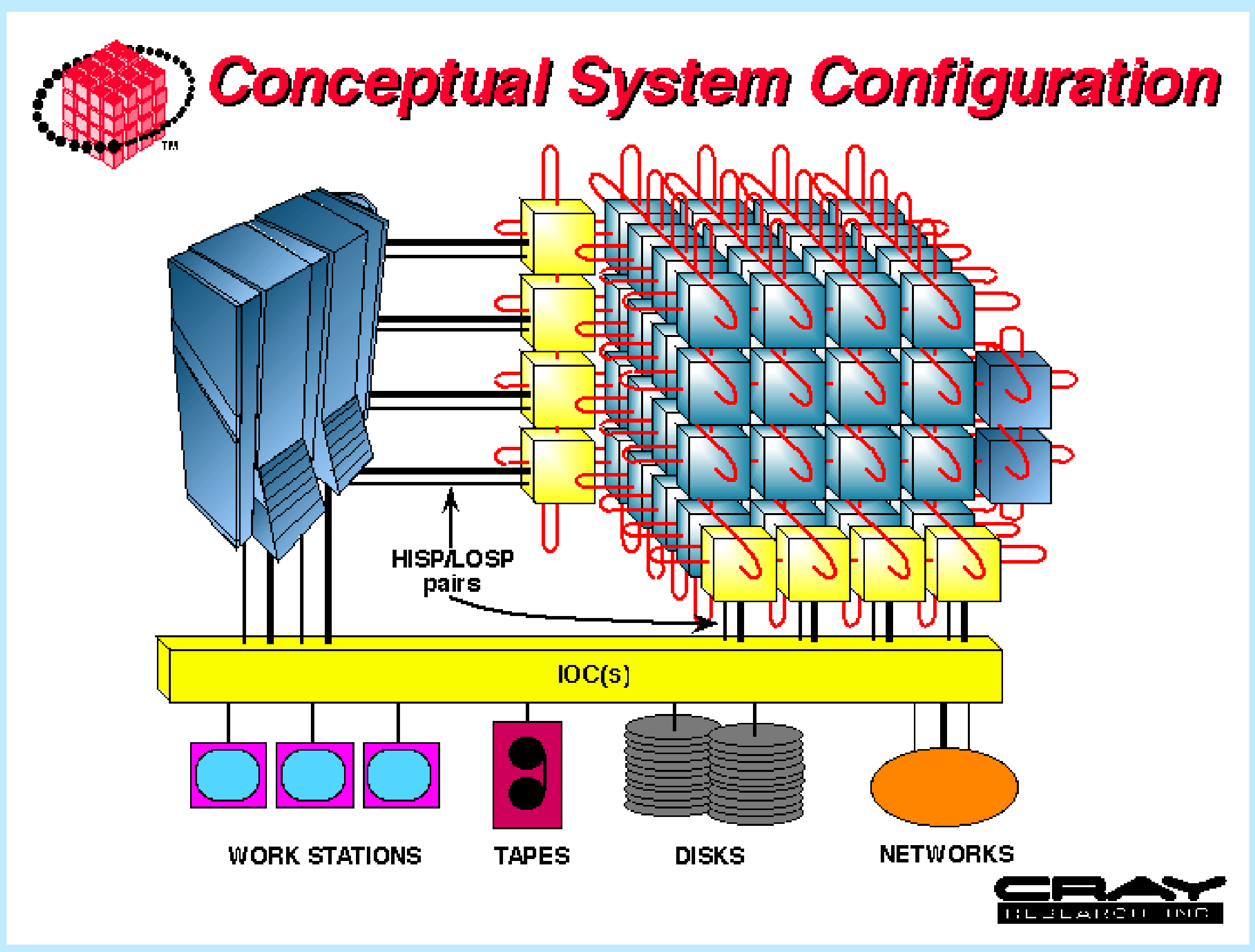
Parallele Rechnerarchitekturen und parallele Programmierung



Günther Fröhlich

Bedeutung der Vorlesung *Paralleldatenverarbeitung*

- **Massiv** parallele Rechner werden in erster Linie nur in **Forschungsinstituten** und im privatwirtschaftlichen Bereich nur in der **Luft- und Raumfahrtindustrie** sowie in der **Automobil- und Rüstungsindustrie** eingesetzt – relativ exotische Stellung.
- In **allen Klassen** von Computern wird die Parallelisierung der Programmausführung als Mittel der Leistungssteigerung vorangetrieben. Die Vorlesung will die verschiedenen Ansätze erläutern.
- Das einige Milliarden Dollar umfassende Forschungsförderungs-programm **Grand Challenges** wurde Anfang der neunziger Jahre von der Regierung der USA begonnen. Ziel ist die Weiterentwicklung des Supercomputing, die als notwendig erachtet wird, um die “großen Probleme unserer Zeit” zu lösen (Project ASCI siehe http://www.top500.org/list/2001/11/)

Themen der Vorlesung

- Das Kapitel **aktuelle Pressemeldungen** liefert Material zur Veranschaulichung, welche Firmen Supercomputer entwickeln, in welchen Bereichen und von welchen Auftraggebern Supercomputer eingesetzt werden.

- Das Kapitel **Strukturen von Parallelrechner-Architekturen** beschreibt wesentliche Eigenschaften der Parallerechner vom **Standpunkt der Hardware**. Es wird gezeigt, wie die verschiedenen Architekturen mit der Organisation von Parallelität auf verschiedenen **Ebenen** korreliert.

Themen der Vorlesung

- **Konzepte der Parallelarbeit** behandelt die theoretischen Grundlagen der Parallelarbeit.
- Lernziel: verstehen, daß Parallelarbeit auf unterschiedlichen Ebenen realisiert werden kann, (z.B. Anweisungs- und Prozeßebene).
- Es wird ausgeführt,
  - mit welchen Mitteln Anweisungen parallelisiert werden können ,
  - welche Schwierigkeiten beim Erkennen von Parallelität auf Anweisungsebene auftreten,
  - welche Bedeutung den Compilern in diesem Zusammenhang zukommt,
  - wie Parallelarbeit auf Prozeßebene organisiert werden kann
  - wie die verschiedenen Prozesse synchronisiert werden .

Themen der Vorlesung

- Im Kapitel **Superskalare Prozessoren** und **Very Long Instruction Word (VLIW)-**Maschinen werden zwei aktuelle Konzepte der Parallelarbeit auf Anweisungsebene vorgestellt und diskutiert.
- Beiden gemeinsam ist der **Einsatz mehrerer Funktionseinhei-ten** wie zum Beispiel Integer-Unit, Floatingpoint-Unit, Branch-Unit, Load/Store-Unit u.ä. Obwohl die VLIW-Maschinen **mehrere Instruktionen auf einmal in den Prozessor laden** und verarbeiten können, wird gezeigt, weshalb die Performance von superskalaren Prozessoren zur Zeit die der VLIW-Maschinen übertrifft und weshalb beide Architekturen mehr als alle anderen so gut oder so schlecht sind wie ihre Compiler!

Themen der Vorlesung

- Im ersten Kapitel wird die Variable bereits dadurch kritisiert, daß beim Versuch der Parallelisierung eines Algorithmus Datenkonflikte erkannt und sehr aufwendig vermieden werden müssen. Einen ganz anderen Weg beschreiten hier die **Datenflußarchitekturen**, die mit der **Zündregel-Semantik** in Verbindung mit dem **Prinzip der einmaligen Zuweisung** in der Lage sind, jede Möglichkeit der Parallelarbeit **selbst** zu erkennen und mit dem Ziel der bestmöglichen Ausnutzung der Betriebsmittel auszuführen. Obwohl Datenflußarchitekturen kaum über das Stadium von Prototypen hinaus gelangten, sollen sie wegen des interessanten parallelen Ansatzes und der konsequentesten Kritik der von Neumann Variable in einem gesonderten Kapitel gewürdigt werden.

Themen der Vorlesung

- Multiprozessorsysteme
  - mit gemeinsamem Speicher
  - mit verteiltem Speicher.
- Die zukünftigen Supercomputer im Teraflops-Leistungsbereich können nur als massiv-parallele Rechner realisiert werden. Die unterschiedlichen Ansätze für massiv-parallele Systeme werden in diesem Kapitel vorgestellt und die jeweiligen Vor- und Nachteile bezüglich Programmierung, Skalierbarkeit, Synchro-nisation usw. herausgearbeitet

Themen der Vorlesung

- Auf dem Gebiet der Superrechner dominierten bis Mitte der neunziger Jahre die **Vektorrechner**. Diese Architektur setzt auf der **Mikroarchitekturebene** - neben mehreren Funktionseinhei-ten mit **arithmetischen Pipelines** - spezielle **Vektorregister-bänke** ein. Damit läßt sich die Verarbeitung von Aufgabentypen mit **feldartigen Datenstrukturen** beschleunigen. Um die parallelen Eigenschaften von Vektorrechnern durch Software nutzen zu können, wird die Befehlssatzarchitektur durch **spezielle Vektorbefehle** ergänzt. Dies wird eingehend im Kapitel *Vektormaschinen* erläutert. Dabei wird auch der Unterschied zu den **RE-Arrays** herausgearbeitet, die auf der Software-Seite wie Vektormaschinen erscheinen, sich aber auf der Mikroarchitekturebene durch den Einsatz einer Vielzahl von ALUs unterscheiden und auch daraus ihre besondere Rechenleistung beziehen.

Hitachi SR 8000-F im Leibniz Rechenzentrum (Rechenzentrum für Münchner Hochschulen)

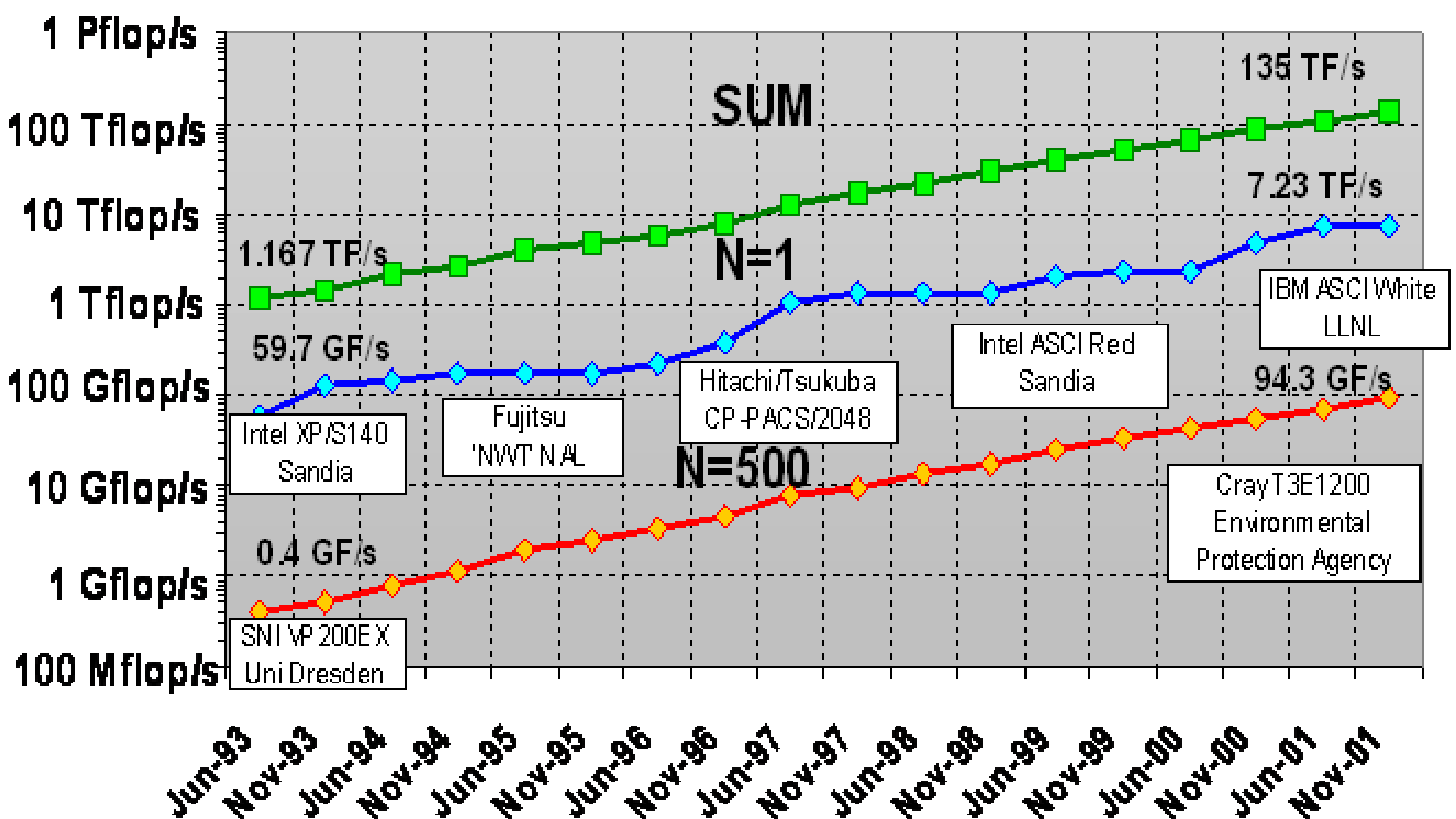
	Anfangskonfiguration 2000	**Endausbau 2002**
Anzahl der SMP-Knoten	112	168
Prozessoren pro Knoten	8	8
Anzahl Prozessoren	896	1344
Spitzenrechenleistung **eines** Prozessors	1.5 GFlop/s	1.5 GFlop/s
Spitzenrechenleistung **eines** Knotens	12 GFlop/s	12 GFlop/s
Spitzenrechenleistung des Gesamtsystems	1344 GFlop/s	2016 GFlop/s

AlphaServer SC ES45/1 GHz im Pittsburg Supercomputing Center

The TCSini is the PSC's latest supercomputing system. Currently comprised of 64 Compaq Alphaserver nodes, it will grow to over 600 nodes providing six teraflops of computing power at peak performance by the end of 2001.

TOP500

Performance Development



Customer Type

500
400
300
200
100
0

Vendor
Classified
Academic
Industry
Research

Jun-93
Nov-93
Jun-94
Nov-94
Jun-95
Nov-95
Jun-96
Nov-96
Jun-97
Nov-97
Jun-98
Nov-98
Jun-99
Nov-99
Jun-00
Nov-00
Jun-01
Nov-01

Chip Technology

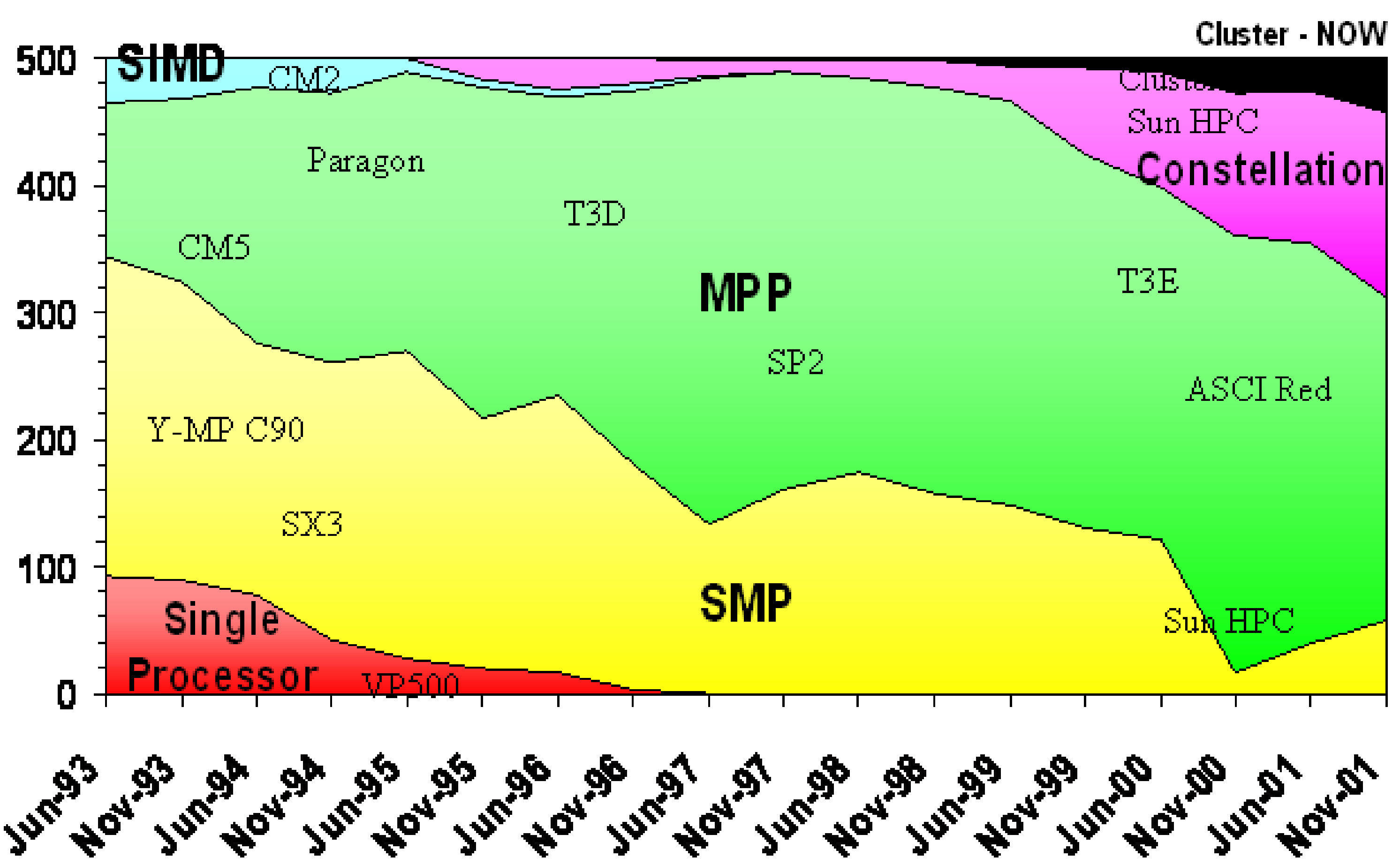
proprietary
other COTS
Sparc
MIPS
PA
Power
intel
Alpha

500
400
300
200
100
0

Jun-93 Nov-93 Jun-94 Nov-94 Jun-95 Nov-95 Jun-96 Nov-96 Jun-97 Nov-97 Jun-98 Nov-98 Jun-99 Nov-99 Jun-00 Nov-00 Jun-01 Nov-01

TOP500

Architectures



Parallele Anwendungen - Finite Elemente Programme - Temperaturverteilung

30 30 30 30 30 30 30 30

30 30 30 30 30 30 30 30 30

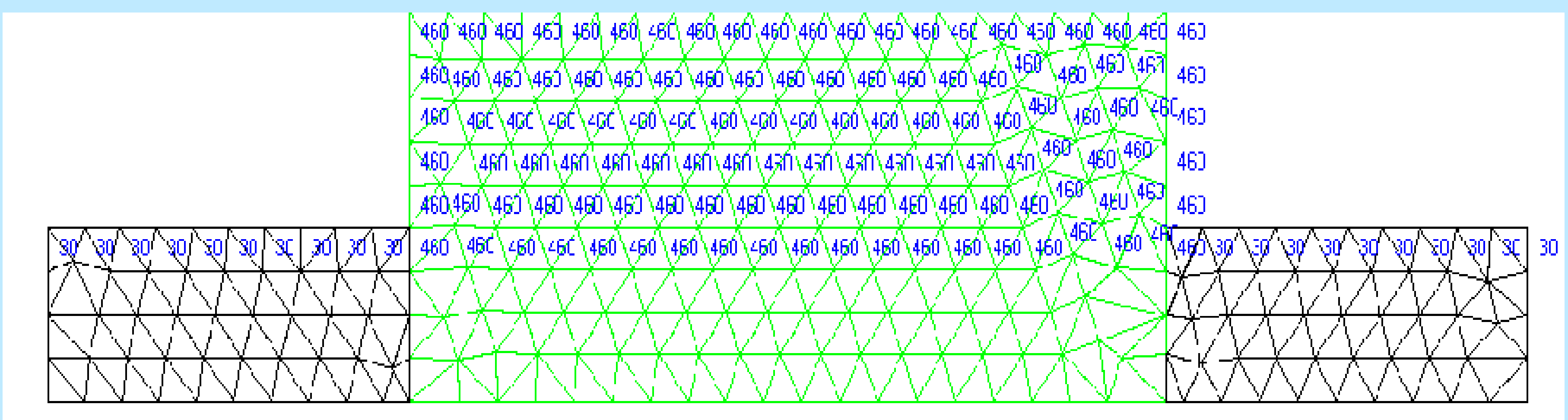
Starttemperatur 460ºC an folgenden Stellen:

460 460 460 460 460 460 460 460 460 460 460

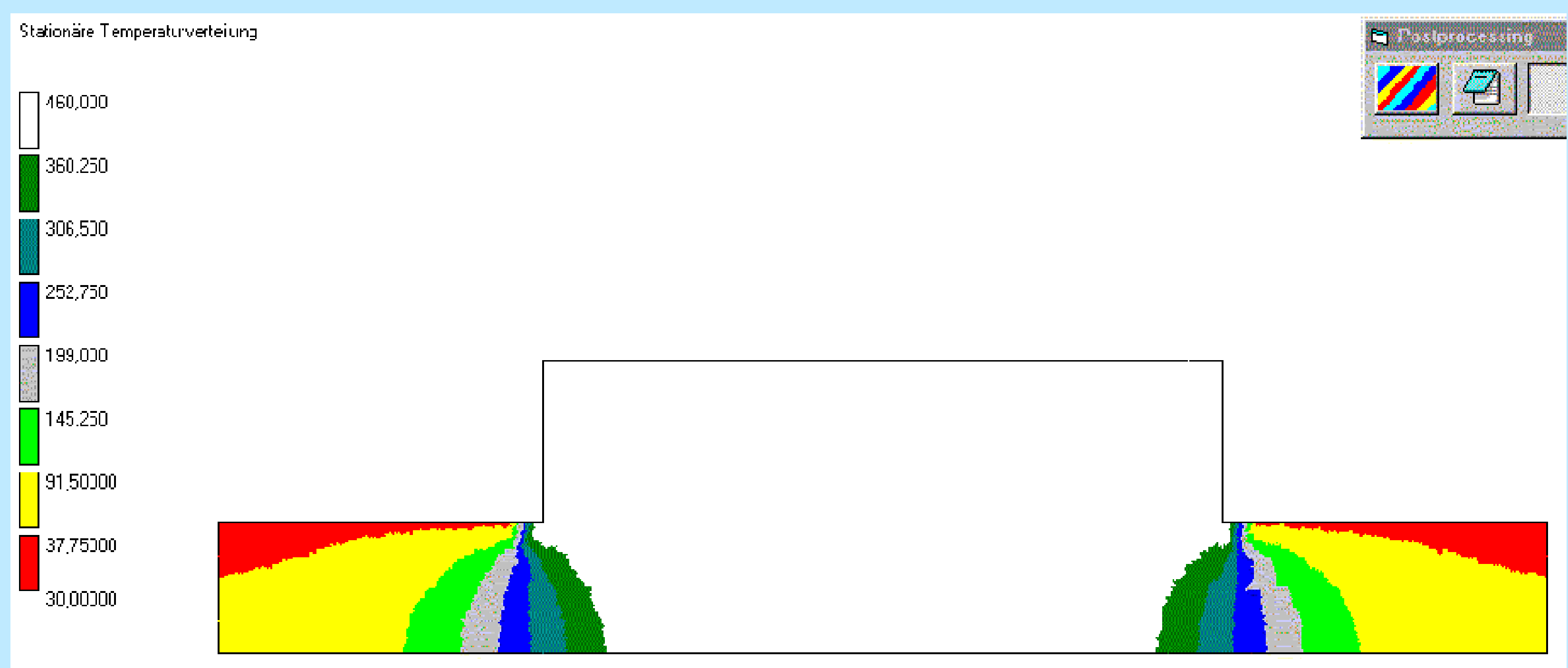
460 460 460 460 460 460 460 460 460 460 460

460 460 460 460 460 460 460 460 460 460 460

Parallele Anwendungen - Finite Elemente Programme - Temperaturverteilung



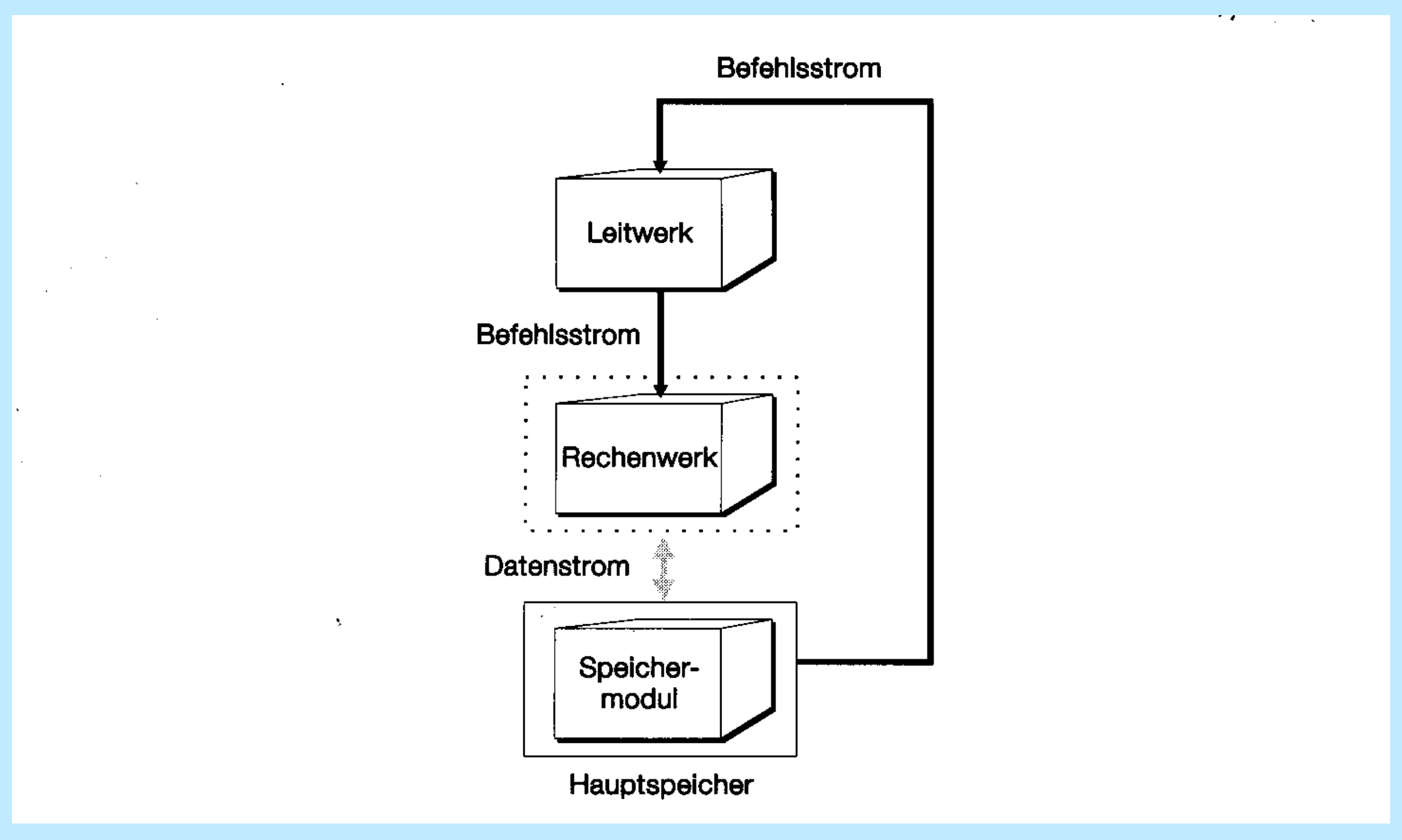
Parallele Anwendungen - Finite Elemente Programme - Temperaturverteilung



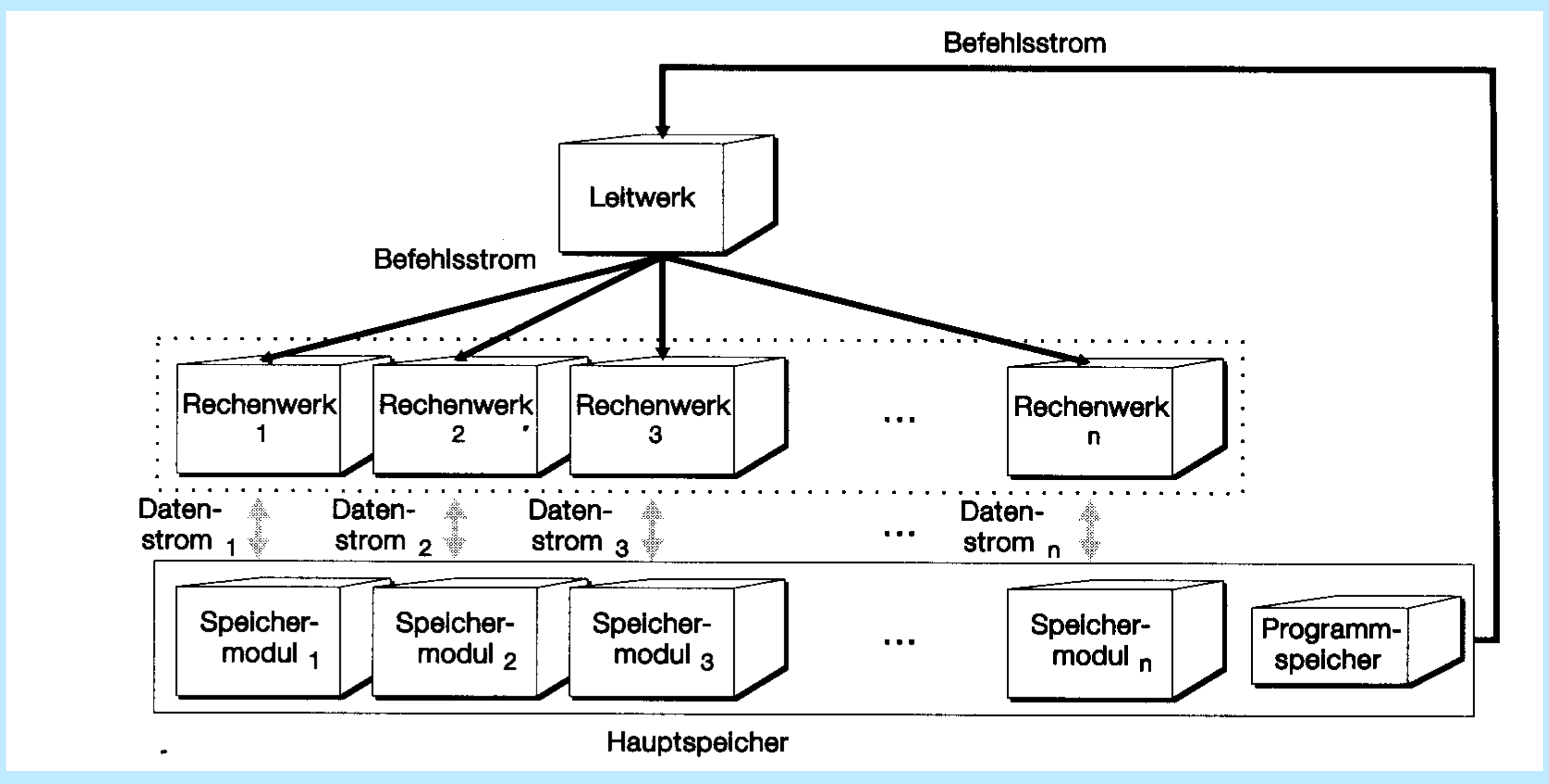
Einordnung von Parallelrechnern - Flynnsche Klassifikation

- **SISD** - von-Neumann-Architekturen (Einprozessorrechner)
- **SIMD** - Feldrechner und Vektorrechner
- **MIMD** - die Multiprozessorsysteme

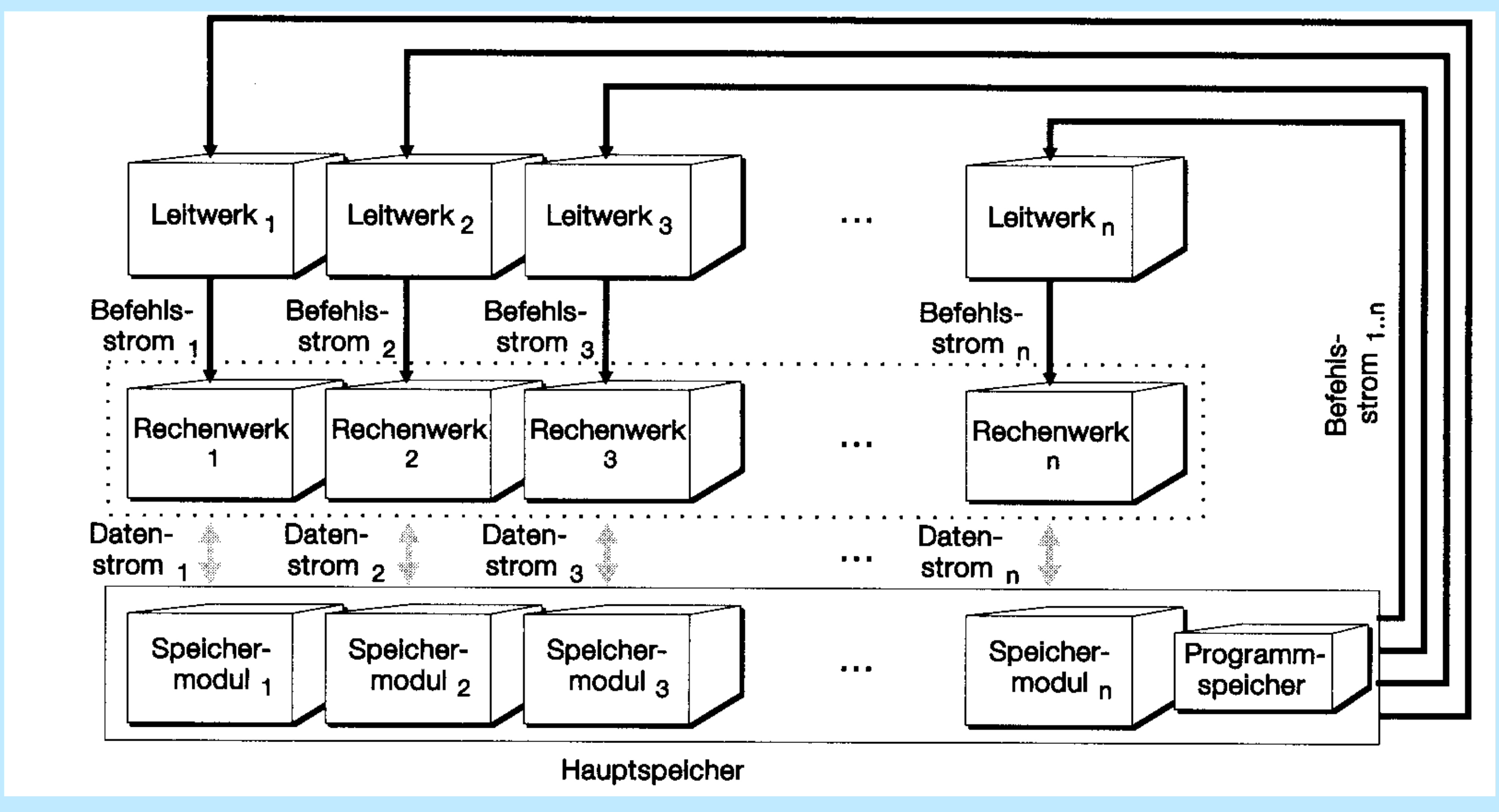
Single Instruction, Single Data



Single Instruction, Multiple Data



Multiple Intruction, Multiple Data



Multiprozessorsysteme

- gehören in die Kategorie MIMD
- bestehen aus mehr als einem Prozessor
- sind *homogen,* wenn alle Prozessoren hardwaremäßig gleich sind
- sind *symetrisch,* wenn die Prozessoren bezüglich ihrer Rolle im System austauschbar sind
- sind *asymetrisch,* wenn die Prozessoren unterschiedliche Rollen im System spielen

Speichergekoppelte Multiprozessorsysteme

- Es gibt einen zentralen Speicher für alle Prozessoren oder zumindest einen allen zugänglichen Kommunikationsspeicher. Man nennt sie deshalb auch *Systeme mit gemeinsamem Speicher.* Alle Prozessoren besitzen einen gemeinsamen Adreßraum. Kommunikation und Synchronisation geschehen über gemeinsame Variablen.. Man unterscheidet:
- Symmetrischer Multiprozessor ***SMP***: ein globalen Speicher
- Distributed-shared-memory-System ***DSM:*** gemeinsamer Adreßraum trotz physikalisch verteilter Speichermodule

Nachrichtengekoppelte Multiprozessorsysteme

- Hier gibt es nur **private** Speicher . Die Prozessoren können daher nicht über den Speicher kommunizieren, sondern müssen sich explizit Nachrichten (*messages*) über eine Verbindungsstruktur zusenden. Man spricht deshalb auch *von* **Systemen mit verteiltem Speicher***.* Jeder Prozessor verfügt über seinen eigenen Adressraum .

Verteiltes Rechnen in einem **Workstation-Cluster**

- Kopplungsgrad nimmt ab, Programme müssen immer grobkörniger sein

Metacomputer: Zusammenschluß weit entfernter Rechner

- Skalierbarkeit der Hardware nimmt zu.

Rechnerkopplung - Metacomputer Beispiel Wetterberechnung

Parallelitätsebene:
- Programm
- Prozess

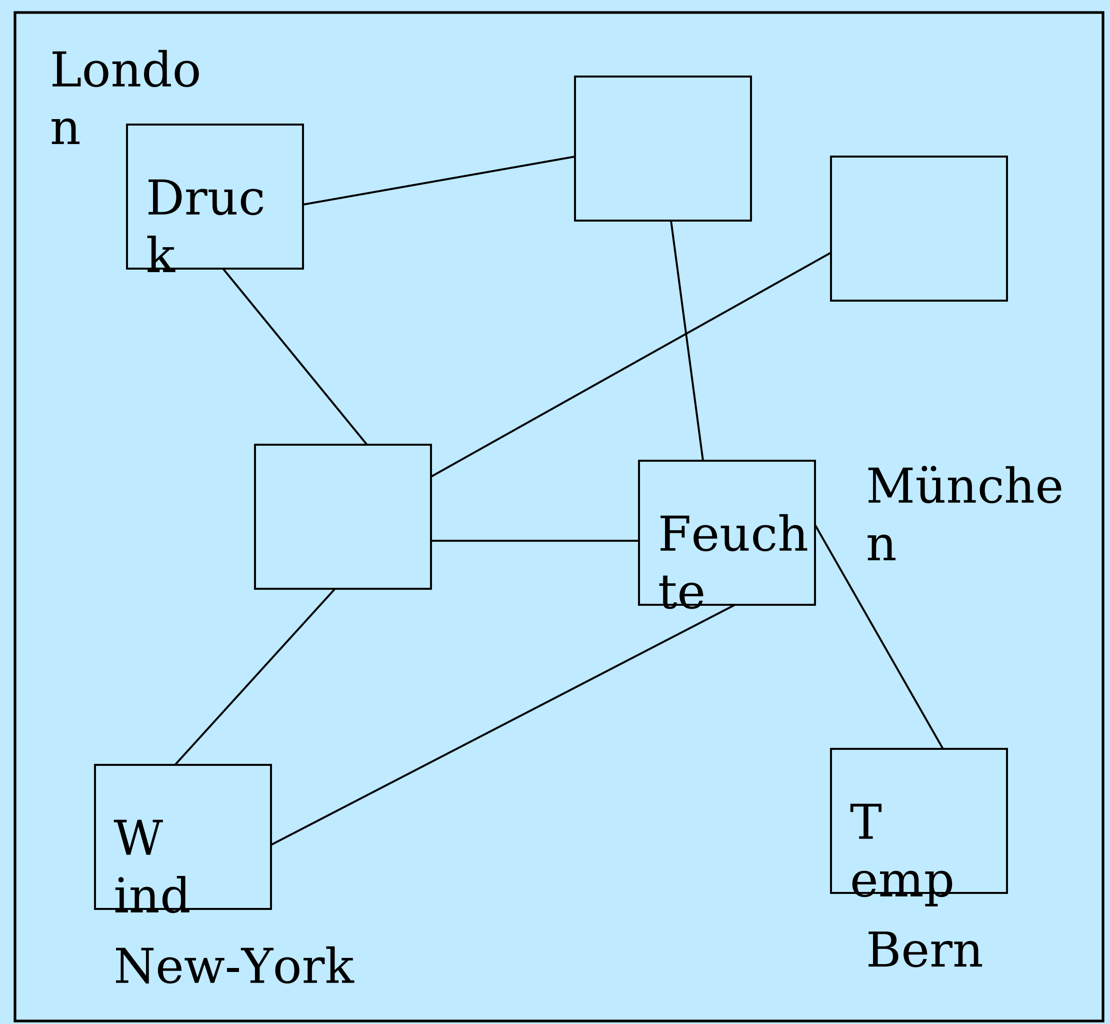
Entfernung:
unbegrenzt

Granularität : grob

Problem: Bandbreite des Netzes

Netz: Internet, Standleitungen
Lohnt bei hohem Rechenaufwand mit minimalem Kommu-nikationsaufwand

Status: Entwicklung



Workstation-Cluster, PC-Cluster
Beispiel Wetterberechnung

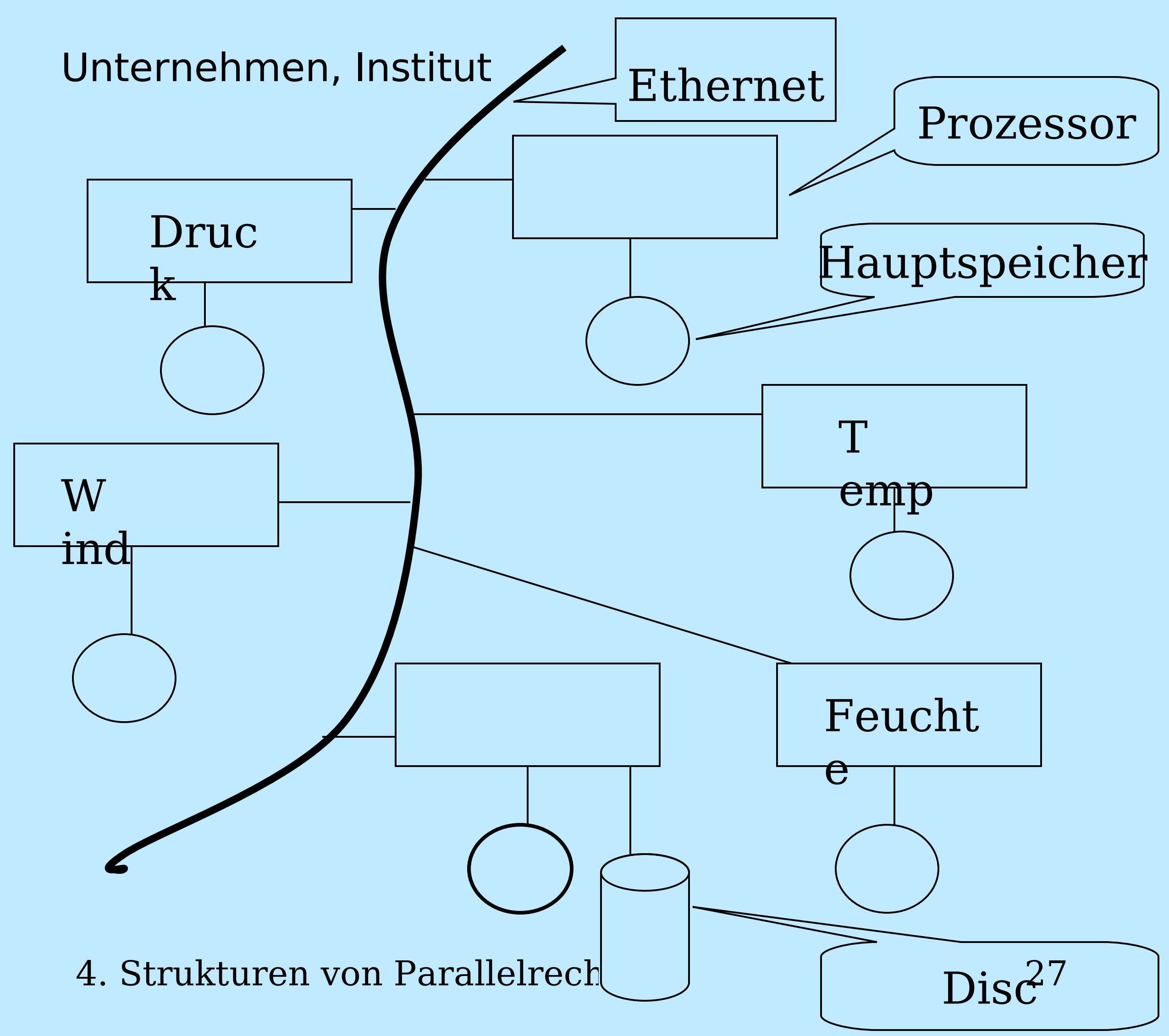
Parallelitätsebene
- Programm
- Prozess

Entfernung: Gebäude
Granularität : grob
Problem: Bandbreite des Netzes
Netz: z.B.Ethernet Lohnt bei hohem Rechenaufwand mit geringem Kommunikationsaufwand

Status: realisiert

Kommunikation: Nachrichten

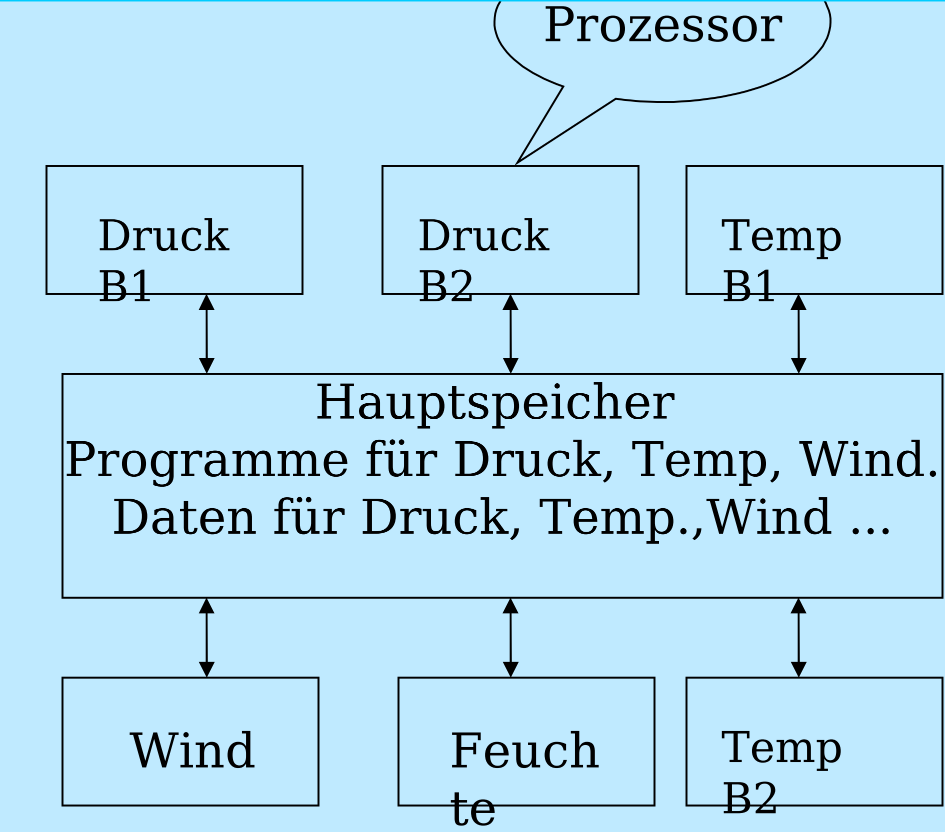
Adressraum: getrennt



Prozessorkopplung - Speicherkopplung Beispiel Wetterberechnung

Parallelitätsebene:
- Programm,
- Prozess
- ***Block***

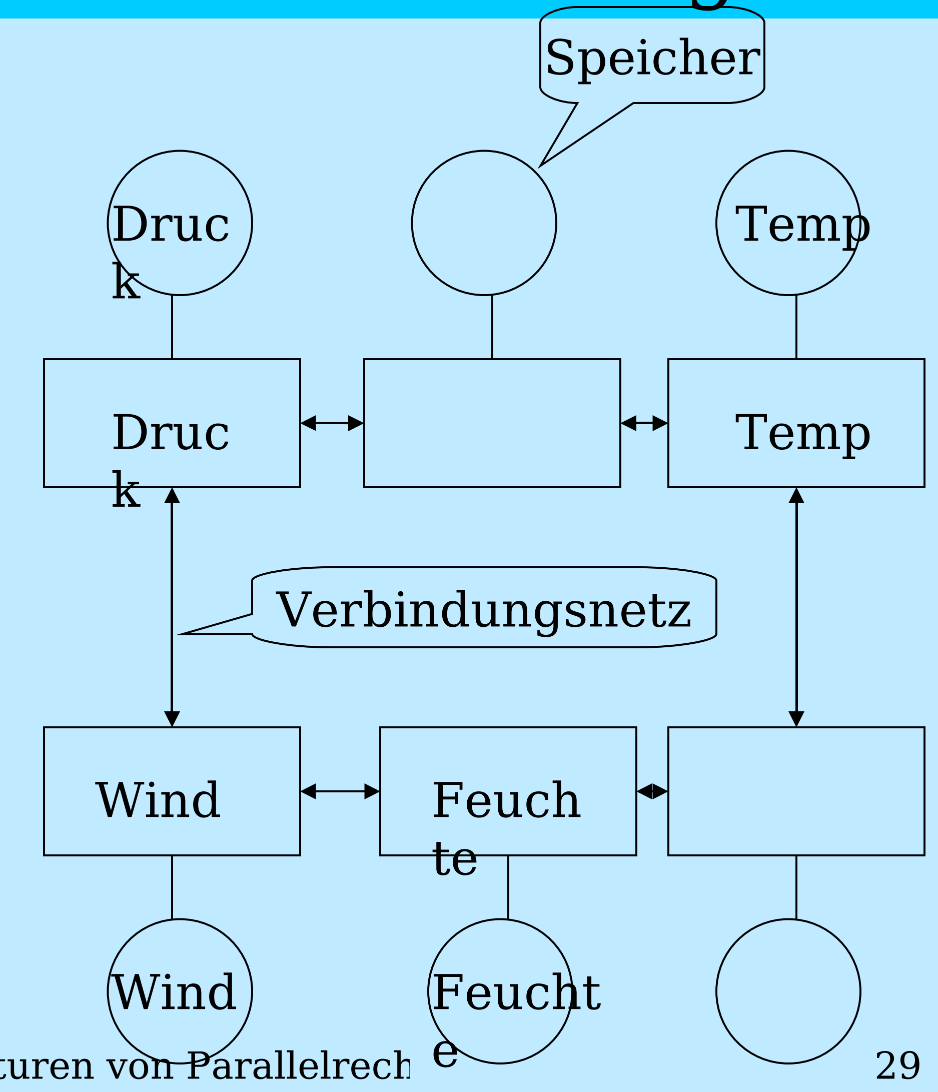
Entfernung: gering, alle Prozessoren bilden einen Computer
Granularität: fein
Netz: keins
Kommunikation: Speicher
Adressraum: gemeinsam



Prozessorkopplung - Nachrichtenorientiert Beispiel Wetterberechnung

Parallelitätsebene:
- Programm,
- Prozess
- ***Block***

Entfernung: gering, alle Prozessoren bilden einen Computer
Granularität: fein
Netz: sehr schnelle Verbindungswege, hohe Bandbreite und kurze Wege
Kommunikation: Nachrichten
Adressraum: getrennt

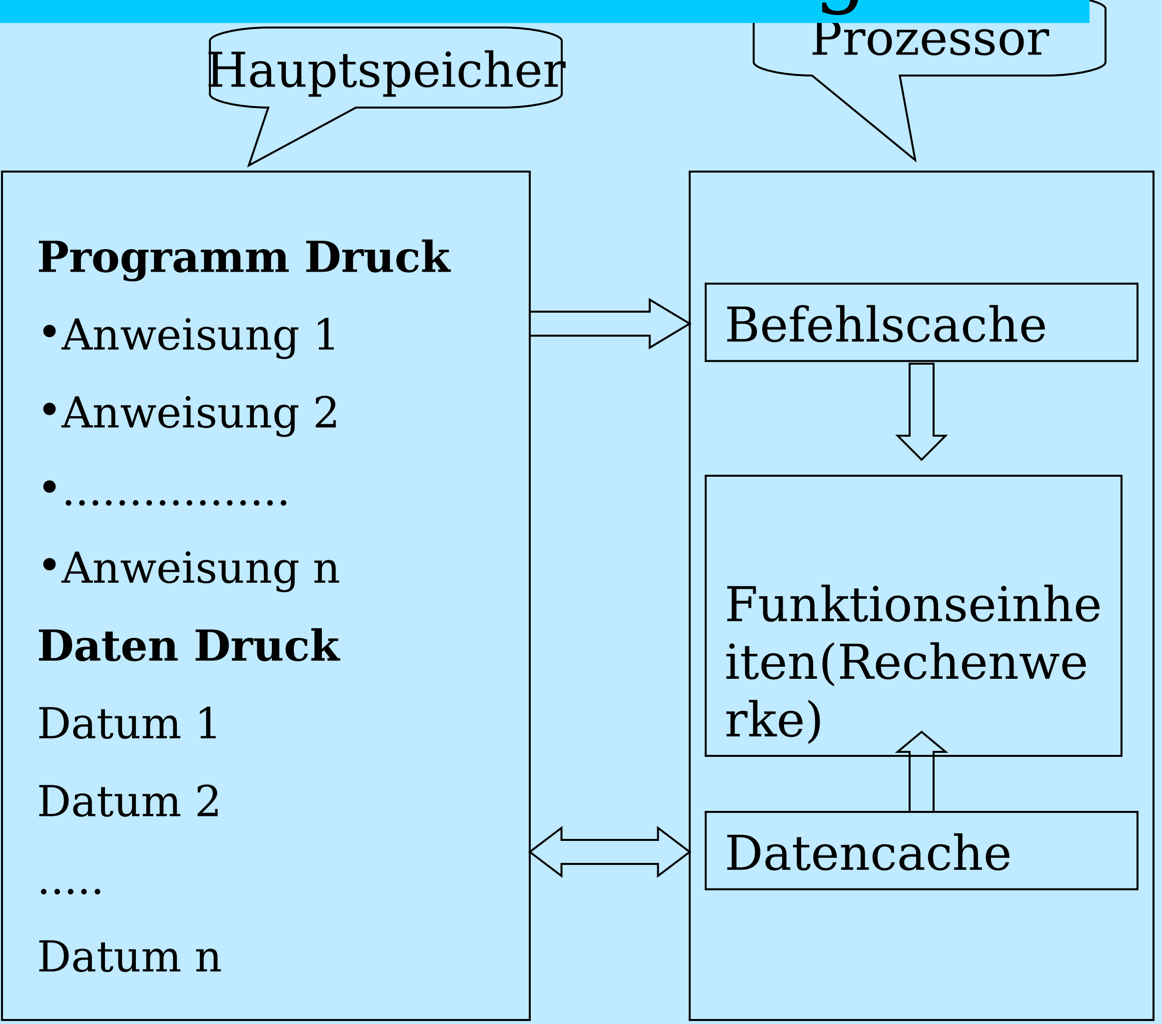


Prozessorarchitektur - Anweisungsebene
Beispiel Wetterberechnung

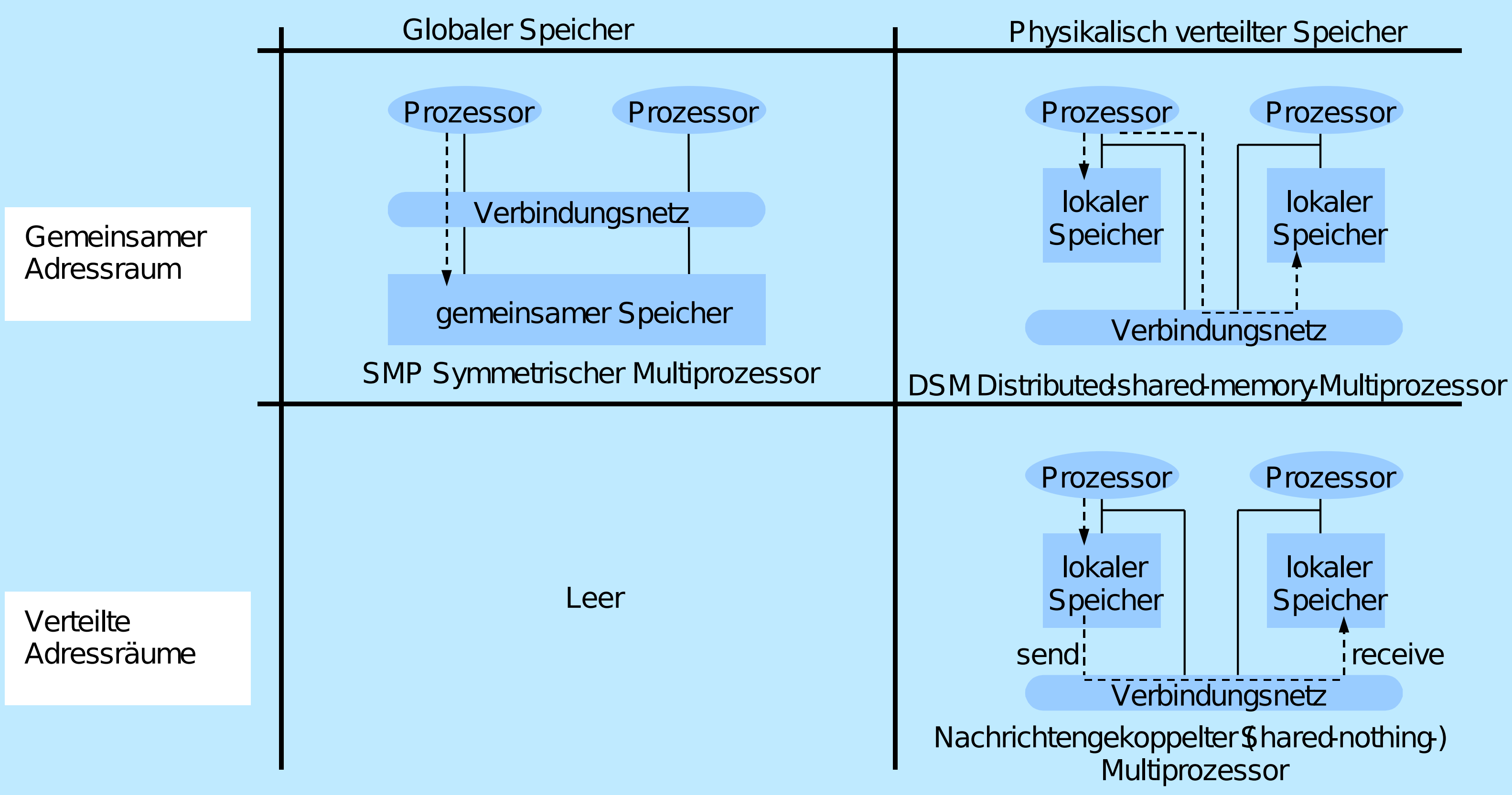
Parallelitätsebene:
- Anweisungen

Designmethode:
- Phasenpipelining
- Funktionspipe-lining
- RISC
- Superscalare Architektur
- Very Long Instruction Word (VLIW)

Problem:
Befehlsabhängig-keiten

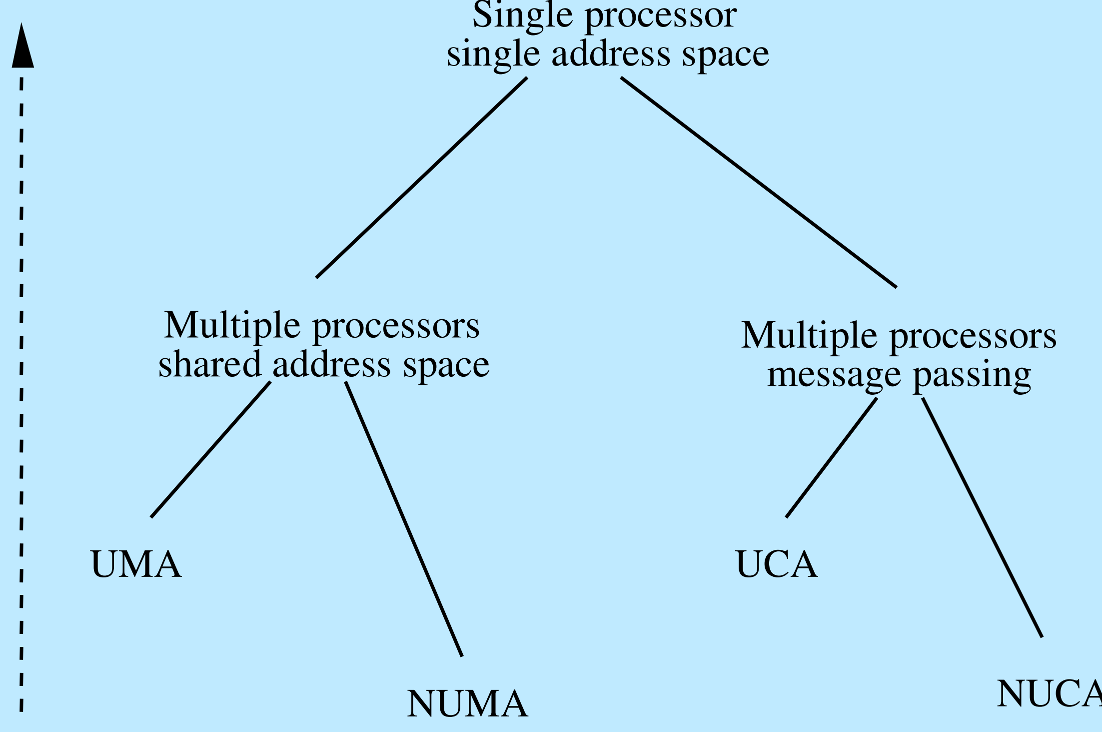


Konfiguration von Multiprozessoren



Zugriffszeit/Übertragungszeit-Modelle

Eichfachheit der Programmierung



Die Ebenen der Parallelarbeit - ***Programmebene***

Diese Ebene wird durch die parallele Verarbeitung verschiedener Programme charakterisiert, die vollständig voneinander unabhängige Einheiten ohne gemeinsame Daten und mit wenig oder keinerlei Kommunikations- und Synchronisationsbedarf sind. Parallelverarbeitung auf dieser Ebene wird vom Betriebssystem organisiert.

Die Ebenen der Parallelarbeit - ***Prozessebene***

Parallelität auf dieser Ebene tritt auf, wenn ein Programm in eine Anzahl parallel auszuführender Prozesse (hier im Sinne **schwergewichtiger Prozesse***)* zerlegt wird. Jeder Prozeß besteht aus vielen sequentiell geordneten Befehlen und umfasst eigene Datenbereiche. Typische Beispiele sind UNIX-Prozesse und Tasks in Ada. Da die einzelnen Prozesse innerhalb eines Programms ablaufen, müssen sie synchronisiert werden, und in der Regel kommunizieren sie miteinan-der. Von Betriebssystemseite findet eine Unterstützung dieser Parallelitätsebene durch Betriebssystem-Primitive zur Prozeßverwaltung, Prozeßsynchronisation und Prozeßkommunikation statt.

Die Ebenen der Parallelarbeit - ***Blockebene***

Diese Ebene betrifft Anweisungsblöcke oder leichtgewichtige **Prozesse** *(threads, light-weighted processes)*. Die leichtgewichtigen Prozesse unterscheiden sich von den schwergewichtigen darin, dass sie aus weniger Befehlen bestehen und mit anderen leichtgewichtigen Prozessen einen gemeinsamen Adressbereich teilen. Dieser gemeinsame Adressraum ist in der Regel der Adressraum eines umfassenden, schwergewichtigen Prozesses. Typische Beispiele für leichtgewichtige Prozesse sind die Threads gemäß POSIX-1003.lc-1995-Standard, wie sie in mehrfädigen Betriebssystemen Verwendung finden. Die Kommunikation geschieht über die gemeinsamen Daten, die

Die Ebenen der Parallelarbeit - ***Blockebene***

Da für die Ausführung eines solchen leichtgewichtigen Prozesses kein eigener Adreßbereich geschaffen wird, ist im Vergleich zu einem schwergewichtigen Prozeß der Aufwand für die Prozeßerzeugung, -beendigung oder einen Prozeßwechsel gering. Zur Blockebene der Parallelität gehören unter anderem innere oder äußere parallele Schleifen in FORTRAN-Dialekten Für viele, meist numerische Programme liegt auf der Blockebene durch parallel ausführbare Schleifeniterationen die potentiell größte Parallelität vor.

Die Ebenen der Parallelarbeit – ***Anweisungsebene***

Auf dieser Ebene können einzelne Maschinenbefehle oder elementare Anweisungen (in der Sprache nicht weiter zerlegbare Datenoperationen) parallel zueinander ausgeführt werden. Optimierende Compiler für superskalare Prozessoren und für VLIW-Prozessoren sind in der Lage, Parallelität auf der Befehlsebene durch die Analyse der sequentiellen Befehlsfolge, die durch Übersetzung eines imperativen, sequentiellen Programms entsteht, zu bestimmen und durch eventuelle Umordnungen der Befehle für den Prozessor nutzbar zu machen. In Datenflußsprachen und in manchen funktionalen Programmiersprachen wird Parallelität auf der Anweisungsebene sogar

Die Ebenen der Parallelarbeit – ***Suboperationsebene***

Eine elementare Anweisung wird durch den Compiler oder in der Maschine in Suboperationen aufgebrochen, die parallel ausgeführt werden. Typische Beispiele dafür sind Vektor- oder Feldoperationen, die von einem Vektorrechner ,,datenparallel" ausgeführt werden. Um Parallelität auf der Suboperationsebene auszudrücken, müssen komplexe Datenstrukturen und Datenoperationen entweder in der höheren Programmiersprache verfügbar sein oder von einem vektorisierenden oder parallelisierenden Compiler aus einer sequentiellen Programmiersprache für die Maschinensprache erzeugt werden. Beispiele für Programmiersprachen mit komplexen Datenstrukturen sind APL, PASCAL-SC, FORTRAN-90, HPF, C*, Parallaxis und Modula-2*. Suboperationsparallelität bietet in Verbindung mit komplexen Operationen wie beispielsweise Vektor- oder Matrixoperationen die Möglichkeit, einen hohen Parallehtätsgrad zu erreichen. Was bei konventionellen Sprachen auf der Blockebene durch mehrere geschachtelte Schleifen programmiert werden muß, kann dann oft durch eine einzige elementare Anweisung ausgedrückt und auf der Maschine parallel ausgeführt werden. Falls ein Parallelrechner so entworfen ist, daß Parallelarbeit auf der Suboperationsebene für einen Satz komplexer Maschinenbefehle ausgeführt wird, entfällt ein Großteil der Schleifen und damit ein wesentlicher Anteil der sonst hohen Parallelarbeit auf der Blockebene.

Die Ebenen der Parallelarbeit

Parallelitätsebene	Architekturen
Prozesse	**Multiprozessorsysteme** mit verteiltem Speicher **Nachrichtenorientierte MIMD-Systeme**
Datenstrukturen	**Vektorrechner** **Rechenelemente-Arrays** **Multiprozessorsysteme** mit verteiltem Speicher
Blockebene	**Multiprozessesorsysteme** mit gemeinsamem Speicher und Schleifen-Parallelisierierung ***Vielfädige Architekturen*** feiner Granularität

Die Ebenen der Parallelarbeit

Maschinenbefehle (Anweisungen inklusive Elementaroperationen)	(**RISC** Prozessoren mit Phasenpipelining) **Superskalare** *Prozessoren* mit parallel arbeitenden Funktionseinheiten **VLIW** Maschinen mit parallel arbeitenden Funktionseinheiten **feinkörnige Datenflußrechner** mit parallel arbeitenden Funktionseinheiten

Techniken der Parallelarbeit

Parallelarbeitstechniken	Programmebene	Prozeßebene		Anweisungsebene	Suboperationsebene
Techniken der Parallelarbeit durch Rechnerkopplung					
Hyper- und Metacomputer	X	X			
Workstation-Cluster	X	X			
Techniken der Parallelarbeit durch Prozessorkopplung					
Nachrichtenkopplung	X	X			
Speicherkopplung (SMP)	X	X	X		
Speicherkopplung (DSM)	X	X	X		
Grobkörniges Datenflußprinzip		X	X		
Techniken der Parallelarbeit in der Prozessorarchitektur					
Befehlspipelining				X	
Superskalar				X	
VLIW				X	
Überlappung von E/A- mit CPU-Operationen				X	
Feinkörniges Datenflußprinzip				X	
SIMD-Techniken					
Vektorrechnerprinzip					X
Feldrechnerprinzip					X

Beschleunigung und Effizienz von parallelen Systemen
Fragestellung

Beispiel: Ein Einprozessorsystem benötige für 1000 Operationen 1000 Schritte. Ein System mit 4 Prozessoren benötige dafür 1200 Operationen, die in 400 Schritten ausgeführt werden können.

Frage: Wie groß ist die Beschleunigung und wie ist Aufwand und Nutzen zu *bewerten* ?

Die folgenden Formeln beschreiben Beschleunigung und Effizienz von parallelen Systemen.

.

Beschleunigung und Effizienz von parallelen Systemen

Definitionen

P(1): Anzahl der auszuführenden (Einheits-) Operationen des Programms auf einem Einprozessorsystem.

P(n): Anzahl der auszuführenden (Einheits-) Operationen des Programms auf einem Multiprozessorsystem mit *n* Prozessoren.

T(1): Ausführungszeit auf einem Finprozessorsystem in Schritten

T(n): Ausführungszeit auf einem Multiprozessorsystem mit *n* Prozessoren in Schritten (oder Takten).

Vereinfachenden Voraussetzungen:

T(l) = P(1), da in einem Einprozessorsystem jede (Einheits-)Ope-ration in genau einem Schritt

Beschleunigung und Effizienz von parallelen Systemen

Speed-up

Die Beschleunigung (Leistungssteigerung, *Speed-up) S(n)* ist definiert als

$$S(n) = \frac{T(1)}{T(n)} \qquad z.B.: S(4) = \frac{1000\,Schritte}{400\,Schritte} = 2.5$$

Üblicherweise gilt $1 \leq S(n) \leq n$

Danach arbeitet also im **schlechtesten Fall** das Multiprozessorsystem gerade so schnell wie das Einprozessorsystem und im **besten Fall** ist die Leistungs-steigerung linear zur Anzahl der eingesetzten Prozessoren.

Beschleunigung und Effizienz von parallelen Systemen

Effizienz

Ob der Aufwand zur **Leistungssteigerung** in einem günstigen Verhältnis steht, zeigt die

Effizienz *E(n) (efflciency):* $E(n) = \frac{S(n)}{n}$

$$z.B.: E(n) = \frac{10}{20} = 0.5 \qquad z.B.: E(4) = \frac{2.5}{4} = 0.625$$

Allgemein gilt: $\frac{1}{n} \leq E(n) \leq 1$

Beschleunigung und Effizienz von parallelen Systemen

Skalierbarkeit

Von **Skalierbarkeit eines Parallelrechners** spricht man, wenn das Hinzufügen von weiteren Verarbeitungselementen zu einer kürzeren Gesamtausführungszeit führt (**lineare Steigerung der Beschleunigung)** .

Wichtig ist eine **angemessene Problemgröße**. Bei fester Problemgröße und steigender Prozessorzahl wird irgendwann eine Sättigung eintreten. Die **Skalierbarkeit ist beschränkt**

Beschleunigung und Effizienz von parallelen Systemen
Mehraufwand, Parallelindex

Mehraufwand für die Parallelisierung **R(n) :**

$$R(n) = \frac{P(n)}{P(1)}$$

R(n) beschreibt den bei einem Multiprozessorsystem erforder-lichen Mehraufwand für die Organisation, Synchronisation und Kommunikation der Prozessoren.

$$R(4) = \frac{1200}{1000} = 1.2$$

Es gilt: $1 \le R(n)$

Der **Parallelindex** ***I(n)*** *(parallel index)* ist defi-niert als:

$$I(n) = \frac{P(n)}{T(n)}$$

$$I(4) = \frac{1200}{400} = 3$$

Er gibt den mittleren Grad an Parallelität bzw. die Anzahl der parallelen Operationen pro Zeiteinheit an.

Beschleunigung und Effizienz von parallelen Systemen
Auslastung

Die **Auslastung** *U(n) (utilization)* ist definiert als:

$$U(n) = \frac{I(n)}{n}$$

$$U(n) = R(n) \cdot E(n)$$

$$U(n) = \frac{P(n)}{n \cdot T(n)}$$

$$U(4) = \frac{1200}{4 \cdot 400} = 0.75$$

Sie entspricht dem **normierten Parallelindex**. Sie gibt an, wieviele Operationen jeder Prozessor im Durchschnitt pro Zeiteinheit ausgeführt hat.

Beschleunigung und Effizienz von parallelen Systemen

Das Beispiel

Ein Einprozessorsystem benötige für 1000 Operationen 1000 Schritte. Ein System mit 4 Prozessoren benötige dafür 1200 Operationen, die in 400 Schritten ausgeführt werden können. Damit gilt:

$P(1) = T(1) = 1000$, $P(4) = 1200$ und $T(4) = 400$

Daraus ergibt sich: $S(4) = 2.5$ und $E(4) = 0.625$

$1(4) = 3$ und $U(4) = 0.75$

Es sind im Mittel also drei Prozessoren gleichzeitig tätig, d.h., jeder Prozessor ist nur zu 75% der Zeit aktiv.

$R(4) = 1.2$

Auf einem Multiprozessorsystem sind 20% mehr Operationen als auf einem Einprozessorsystem

Beschleunigung und Effizienz von parallelen Systemen

Amdahls Gesetz

$$T(n) = T(1) \cdot \frac{1-a}{n} + T(1) \cdot a$$

a Summe der Ausführungszeit des nur sequentiell ausführbaren Programmteils

$$S(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{T(1) \cdot \frac{1-a}{n} + T(1) \cdot a}$$

$$= \frac{1}{\frac{1-a}{n} + a} = \frac{n}{(1-a) + n \cdot a}$$

$$S(n) \leq \frac{1}{a}$$

Beschleunigung und Effizienz von parallelen Systemen **Beispiel** zu **Amdahls Gesetz**

Beispiel mit a=0.1 (d.h. 90 % des Programmes können parallel, 10 % müssen sequentiell ausgeführt werden

Der Speedup bei n =2

$$\frac{2}{(1-0.1)+2\cdot 0.1} = \frac{2}{0.9+0.2} = \frac{2}{1.1} = 1.8182$$

5

$$\frac{5}{(1-0.1)+5\cdot 0.1} = \frac{5}{0.9+0.5} = \frac{5}{1.4} = 3.5714$$

10

$$\frac{10}{(1-0.1)+10\cdot 0.1} = \frac{10}{0.9+1.0} = \frac{10}{1.9} = 5.2632$$

100

$$\frac{100}{(1-0.1)+100\cdot 0.1} = \frac{100}{0.9+10} = \frac{100}{10.9} = 9.1743$$

$$\frac{1000}{(1-0.1)+1000\cdot 0.1} = \frac{1000}{0.9+100} = \frac{1000}{100.9} = 9.9108$$

Beschleunigung und Effizienz von parallelen Systemen – **maximales Speed-up**

Beispiel mit a=0.1 (d.h. 90 % des Programmes können parallel, 10 % müssen sequentiell ausgeführt werden

Der maximale Speedup in Abhängigkeit vom Anteil sequentiellen Codes:

$$S(n) = \frac{1}{a}$$

Auch wenn man noch soviele Prozes-soren einsetzt, ein Speedup von zehn ist bei 10% sequentiellen Codes nicht zu übertreffen

$$S(n) = \frac{1}{0.1} = 10$$

Implizite Prgrammparallelität

```
1  DO WHILE (X .NE. Stop)
2     Y = S(X)
3     IF  (X+L(X).EQ. Y)  THEN
4        L(X) = L(X) + L(Y)
5        S(X) = S(Y)
6     ELSE
7        X = Y
8     ENDIF
9  ENDDO
```

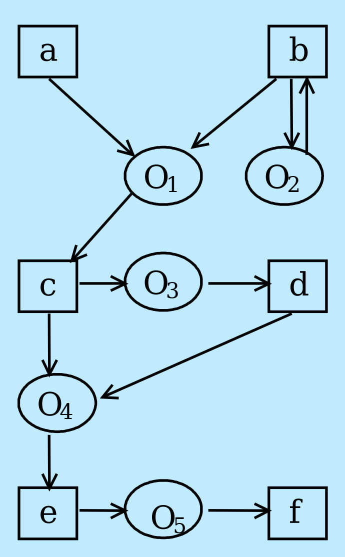
Diese beiden Befehle sind parallel ausführbar

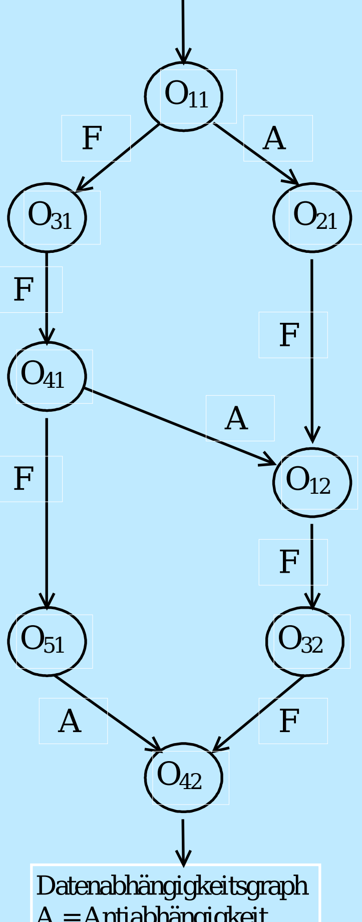
Datenabhängigkeiten und Datenabhängigkeitsanalysen

O_0	**DO I = Anfang, Ende**
O_1	**c = a + b**
O_2	**b = b * b**
O_3	**d = 5 * c**
O_4	**e = c * d**
O_5	**f = e –SQRT(ABS(e))**
O_6	**END DO**

Berechnungsgraph und Datenabhängigkeitsgraph

```
O0 DO I = Anfang, Ende
O1    c = a + b
O2    b = b * b
O3    d = 5 * c
O4    e = c * d
O5    f = e –SQRT(ABS(e))
O6 END DO
```





Berechnungsgraph

Datenabhängigkeitsgraph
A = Antiabhängigkeit
F = Flußabhängigkeit

Datenabhängigkeiten

Definition: Benutzung:	X = A + B ↘ Y = X	Datenfluß-Abhängigkeit - **RAW** (eine Variable darf nicht benutzt werden, wenn sie noch nicht definiert ist)
Definition: Benutzung:	Y = X ↗ X = A+B	Anti-Datenabhängigkeit - **WAR** (eine Variable darf noch nicht geändert werden, wenn eine Operation noch den vorherigen Wert verlangt)
Definition: Benutzung:	X = A+B ... X = C+D	Ausgangs-Datenabhängigkeit - **WAW** (vor dem Wiederbeschreiben einer Variablen muß sicher gestellt werden, daß nicht eine andere Anweisung noch nicht abgeschlossen ist und den vorherigen Wert verlangt.)

Die Parallelität einschränkenden Abhängigkeiten

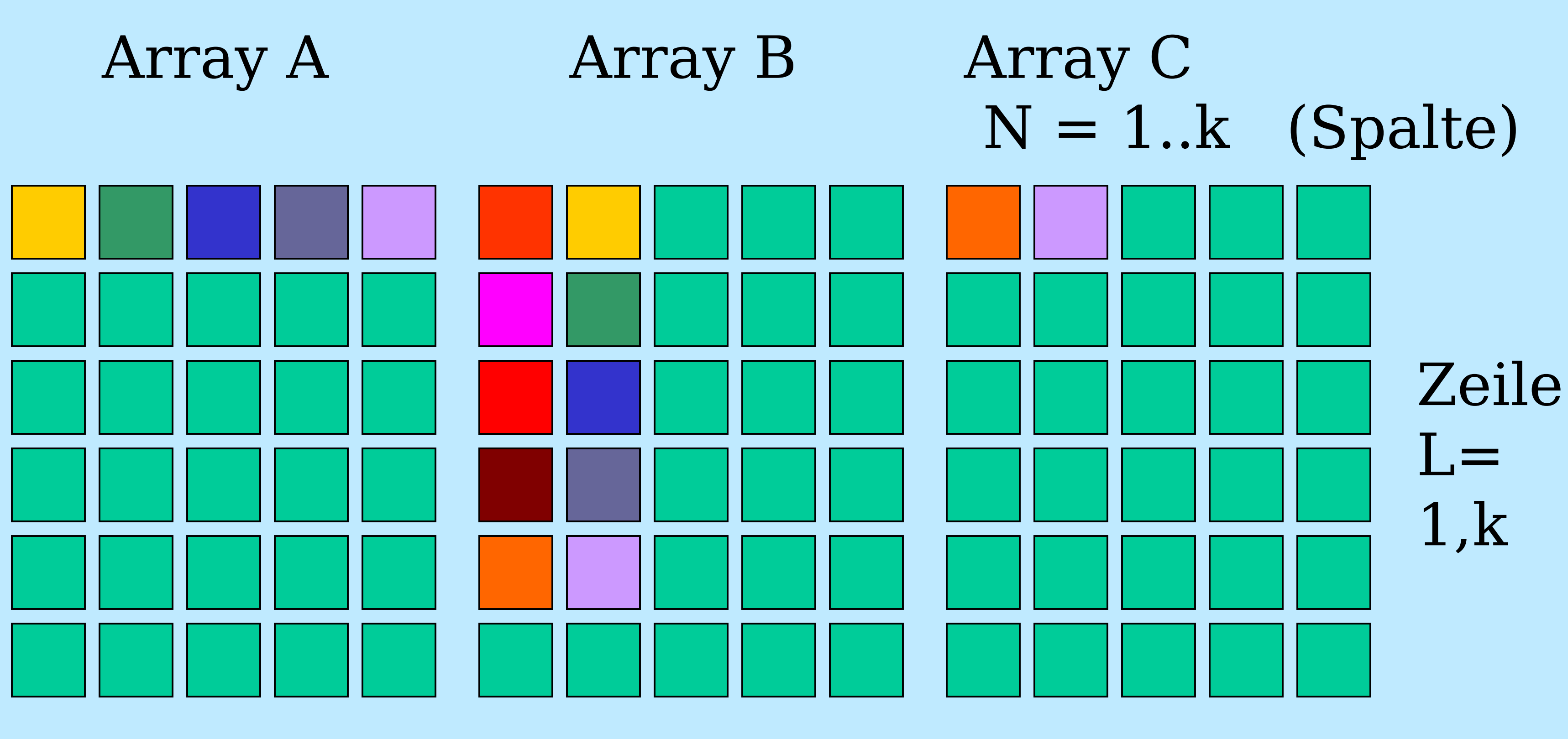
- Datenabhängigkeiten
  - Datenflußabhängigkeit
  - Antiabhängigkeit
  - Ausgabeabhängigkeit
- Prozedurale Abhängigkeiten
  - Fallunterscheidungen
  - Iterationen
    Das bedeutet, daß die Anweisungen, die auf eine Programmverzweigung folgen, erst dann ausgeführt werden können, wenn entschieden ist, wohin verzweigt werden soll.
- Operationale Abhängigkeiten – Abhängigkeit von der Anzahl zur Verfügung steheder Funktionseinheiten

Explizite Datenparallelität

```
procedure MMULT(I,J,K, A,B,C);
value I,J,K;
integer I,J,K;
real array A,B,C;
begin
    integer L,M,N;
    for L := 1 step 1 until K do
        for  N := 1 step 1 until k do
            begin
            C[L,N] :=  0;
            for M := 1 step 1 until J do
                C[L,N] := C[L,N] +
                           A[L,M] * B[M,N]
            end
        end
    end
```

Wieviel einfacher wäre es dagegen, für die als Matrizen deklarierten Variablen A,B und C zu schreiben: **C := A*B**

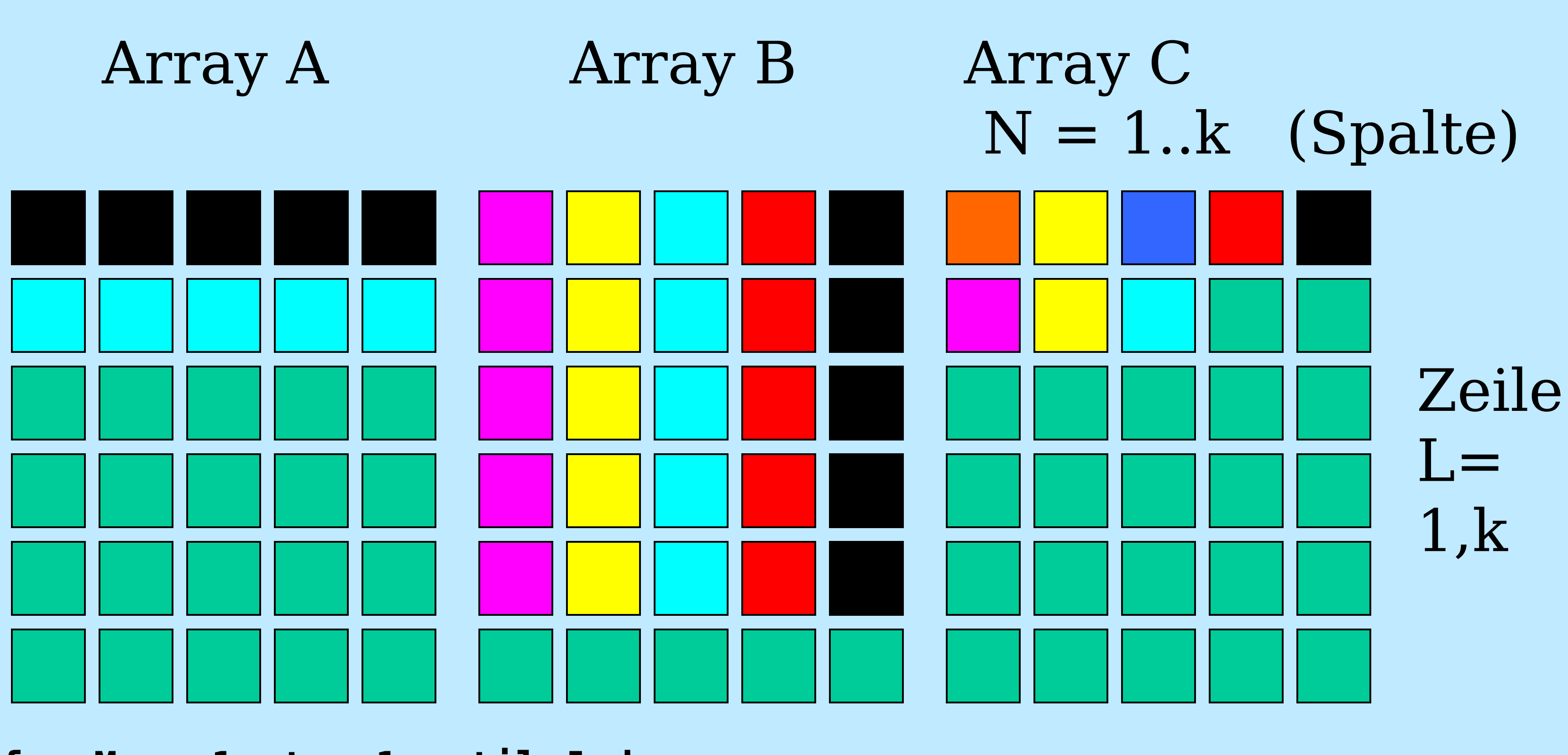
Explizite Datenparallelität




```
for M := 1 step 1 until J do
      C[L,N] := C[L,N]+ A[L,M] * B[M,N]
end
```

A läuft in der Spalte,
B läuft in der Zeile
L läuft langsamer als N

Explizite Datenparallelität




```
for M := 1 step 1 until J do
      C[L,N] := C[L,N]+ A[L,M] * B[M,N]
end
```

A läuft in der Spalte,
B läuft in der Zeile
L läuft langsamer als N

Die Anweisungsebene - Compileroptimierungen

- Mehrere gleichzeitig arbeitsfähige Funktionseinheiten zur Ausführung elemantarer Operationen müssen mit einem Befehlsstrom versorgt werden, der möglichst alle Einheiten gleichzeitig benutzt. Zur entsprechenden Aufbereitung des Codes setzen Compiler Optimierungsstrategien ein:
  - Register Optimierung
    - wird von jedem guten Compiler unabhängig von den weiteren, spezielleren Anforderungen der gegebenen Architekturform ausgeführt.
    - sorgt dafür, dass Zwischenergebnisse im Registerfile so lange gehalten werden, wie sie benötigt werden um langsame Hauptspeicherzugriffe zu vermeiden.

Compileroptimierungen Loop Unrolling

– Loop Unrolling ist eine Programmtransforma-tion des Compilers, bei der mehrere Schleifeniterationen als lineares Programmstück hintereinander geschrieben werden.
  - Nutzen: Operationen (Sprungbefehle) können eingespart werden.

```
DO I = 1, 10
   c = a/I + b/I
   b = I/b*b
   d = I * c
   e = c * d
   f = e - 1*ABS(SQRT(e))
   print *, c,b,d,e,f
END DO
```

Durch vierfaches Aufrollen der Schleife wird der Schleifenkörper fünfmal aneinandergefügt.

Compileroptimierungen - LOOP unrolling

Var1

```
DO I =1, 10,5
c =a/I +b/I
b =I/b * b
d =I • c
e =c • d
f =e - 1 *ABS(SQRT(e))
print *, c,b,d,e,f
I =I+1
c =a/I +b/I
b =I/b * b
d =I • c
e =c • d
f =e - 1*ABS(SQRT(e))
I =I+1
print *, c,b,d,e,f
c =a/I +b/I
b =I/b * b
d =I • c
e =c • d
f =e - 1*ABS(SQRT(e))
print *, c,b,d,e,f
I =I+1
c =a/I +b/I
b =I/b * b
d =I • c
e =c • d
f =e - 1*ABS(SQRT(e))
print *, c,b,d,e,f
I =I+1
c =a/I +b/I
b =I/b * b
d =I • c
e =c • d
f =e - 1*ABS(SQRT(e))
print *, c,b,d,e,f
END DO
```

Var2

```
DO I =1, 10,5
c =a/I +b/I
b =I/b * b
d =I • c
e =c • d
f =e - 1*ABS(SQRT(e))
print *, c,b,d,e,f
I =I+1
c =a/I +b/I
b =I/b * b
d1=I • c
e1=c • d1
f =e1 - 1*ABS(SQRT(e))
I =I+1
print *, c,b,d1,e1,f
c =a/I +b/I
b =I/b * b
d2 =I • c
e2 =c • d2
f =e2 - 1*ABS(SQRT(e))
print *, c,b,d2,e2,f
I =I+1
c =a/I +b/I
b =I/b * b
d3=I • c
e3=c • d3
f =e3 - 1*ABS(SQRT(e))
print *, c,b,d3,e3,f
I =I+1
c =a/I +b/I
b =I/b * b
d4=I • c
e4=c • d4
f =e - 1*ABS(SQRT(e))
print *, c,b,d4,e4,f
END DO
```

Compileroptimierungen - LOOP Jamming

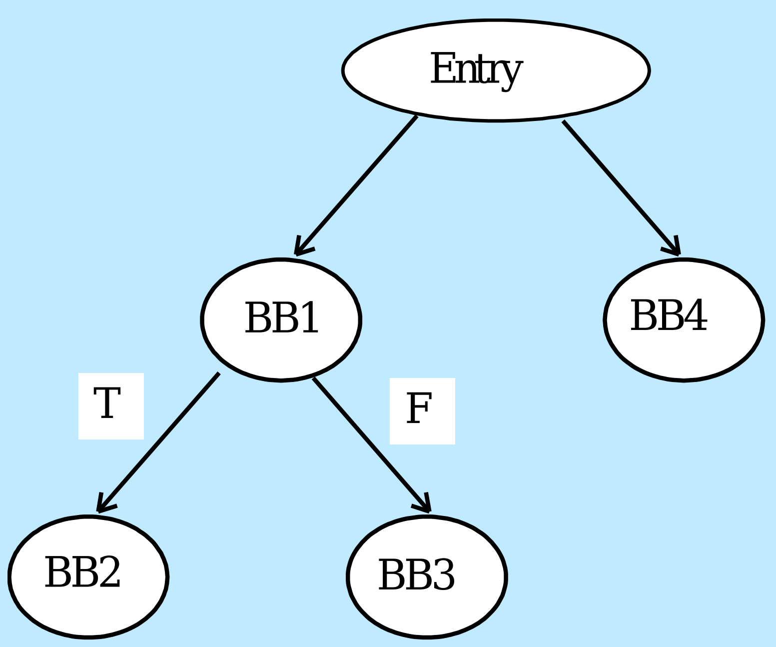
```
DO N = K, J
    S(N) = S(K+N) * S(J-N)
ENDDO
X = X**2
DO N = K, J
     T(N) = K * J / N
ENDDO

DO N = K, J
    S(N) = S(K+N) * S(J-N)
    T(N) = K * J / N
ENDDO
X = X**2
```

Die Vorteile zur Parallelisierung der Anweisungen liegen auf der Hand.

Compileroptimierungen: Erkennen der Parallelität über Basisblöcke hinweg

```
BB1
IF Bedingung THEN
        BB2
ELSE
        BB3
ENDIF
BB4
```



Erkennen der Parallelität über Prozedurgrenzen hinweg

- Der notwendige **Aufwand** für eine interprozedurale Daten-abhängigkeitsanalyse steht bisher aber in keinem günstigen Verhältnis zum erzielbaren Gewinn an Parallelität.
- Auf Grundlage der vom Compiler erkannten Parallelität können so viele Operationen gleichzeitig ausgeführt werden, wie Betriebsmittel zur Verfügung stehen (maschinenabhängig). Da unterschiedliche Operationen unterschiedliche Bearbeitungszeiten (Taktzyklen) benötigen, ergibt sich aus der parallelen Ausführung das Problem der Synchronisation, das später noch betrachtet wird.

Parallelarbeit auf Prozessebene

Prozeßebene: mehrere Prozesse (Programme) erledigen getrennt aber nicht unabhängig voneinander verschiedene Aspekte einer Aufgabe.

Problem: Wie wird die Ausführung von Teilaufgaben im Sinne einer geordneten Ausführung der Gesamtaufgabe synchronisiert?.

Nachrichtenorientiert: Bei MIMD-Architekturen mit verteiltem Speicher müssen die Prozesse über sogenannte **messages** kommunizieren. Dabei entsteht das Problem der **Kommunikationslatenz**, das **Effizienzprobleme v**erursacht **.** Was nützt ein Multiprozessor-System, wenn wenige Prozessoren arbeiten und die meisten nur auf deren Ergebnisse (eine Nachricht) warten, bevor sie weiterarbeiten können.

Die Kommunikationslatenz sinkt, wenn die kooperierenden Prozesse möglichst komplexe Teilaufgaben übernehmen und damit die Kommunikationshäufigkeit und der Kommunkations-Overhead sinkt. Einen Compiler, der ein Anwendungsprogramm in eine Vielzahl von kommunizierenden Prozessen zerlegt, gibt es zur Zeit noch nicht. Die Realisierbarkeit steht noch in Frage. Nach wie vor ist es Aufgabe des Programmierers, diese Zerlegung durchzuführen und die Kommunikation sowie die Synchronisation zu organisieren. Die **Synchronisationsstrategien** werden in einem gesonderten Abschnitt besprochen.

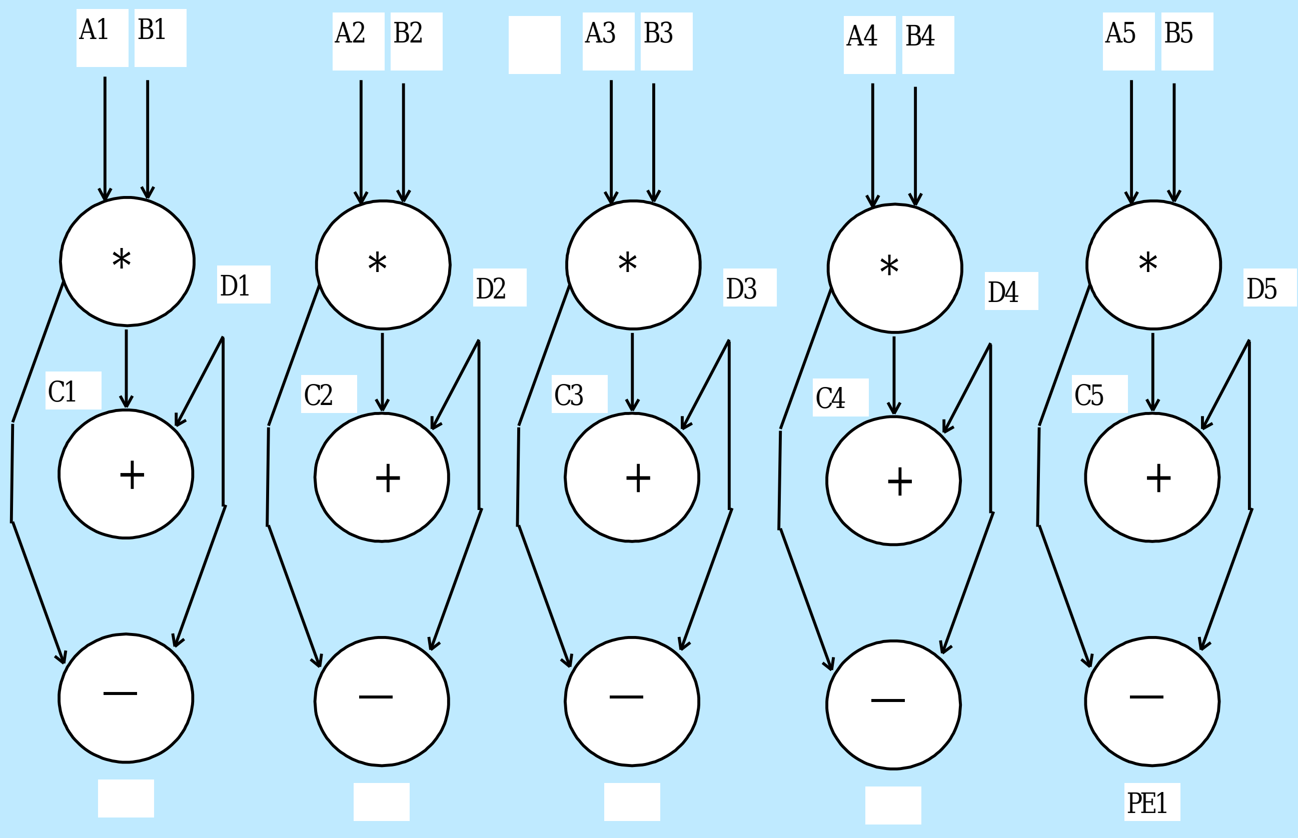
Nutzung der Anweisungsparallelität

- **Compiler** für superskalare Prozessoren und VLIW-Maschinen unterstützen die Parallelität auf Anweisungsebene. Diese Art der Parallelarbeit muß somit nicht programmiert werden
- Konstrukte (**Compiler-Direktiven**), die Programmiersprachen wie ALGOL bereits in den sechziger Jahren anboten, damit bei vom Programmierer erkannter Datenunabhängigkeit der Compiler parallelen Code erzeugen konnte
- (z.B. PARBEGIN A1;A2;A3 END)
- Bei der **Nutzung der Schleifen-Parallelität** findet man sowohl die explizite Programmierung in einer Sprache, die Anweisungen für die parallele Schleifenausführung hat (z.B. FORTRAN 95) als auch die automatische Erkennung durch den Compiler. Compiler für *Vektormaschinen* haben heute immer auch einen **automatischen Vektorisierer**, der Schleifen in skalaren Programmiersprachen (z.B. FORTRAN 77) erkennen und automatisch für die Verarbeitung aufbereiten kann. An der **Schleifen-Parallelisierung** kann man prinzipielle Unterschiede zwischen verschiedenen parallelen Architekturen deutlich machen

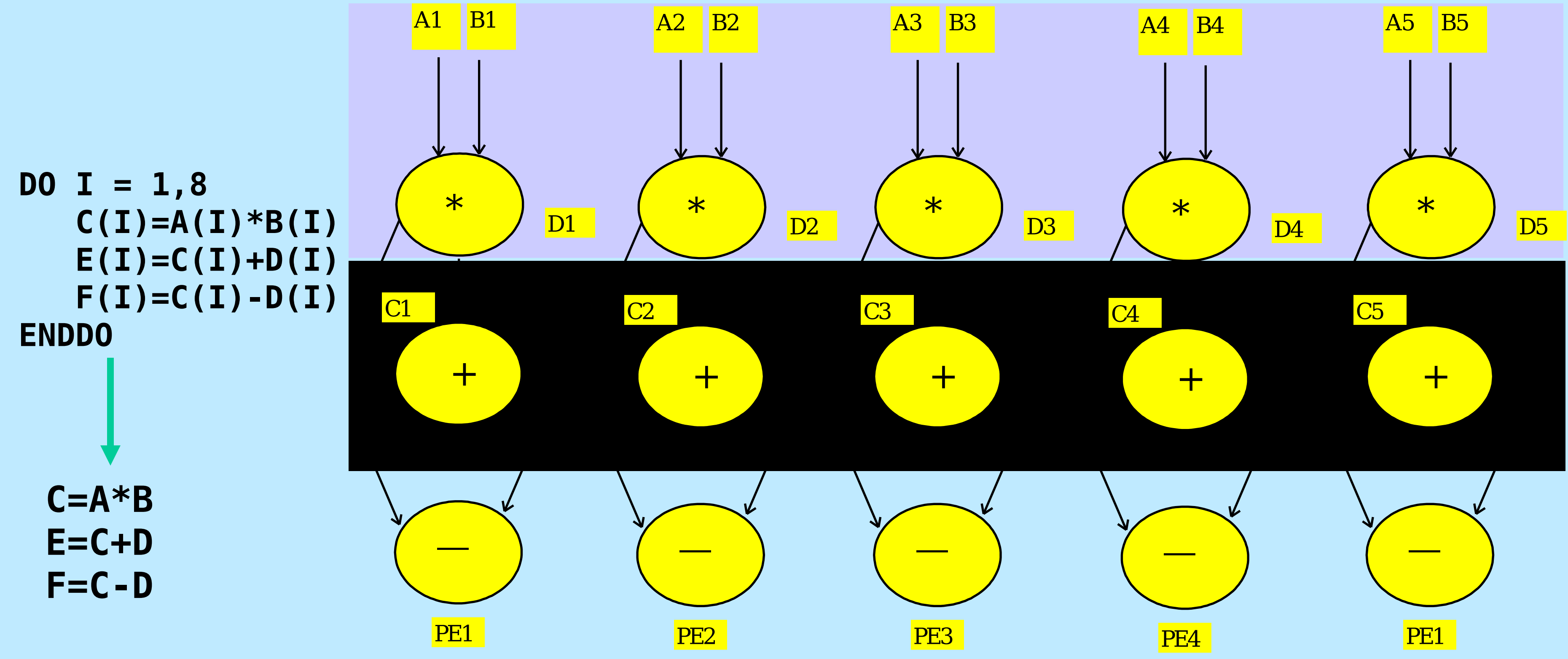
Schleifen-Parallelisierung

- Vektorisierung
- Interation

Beispiel:
DO I = 1,8
C(I) = A(I) * B(I)
E(I) = C(I) + D(I)
F(I) = C(I) - D(I)
ENDDO

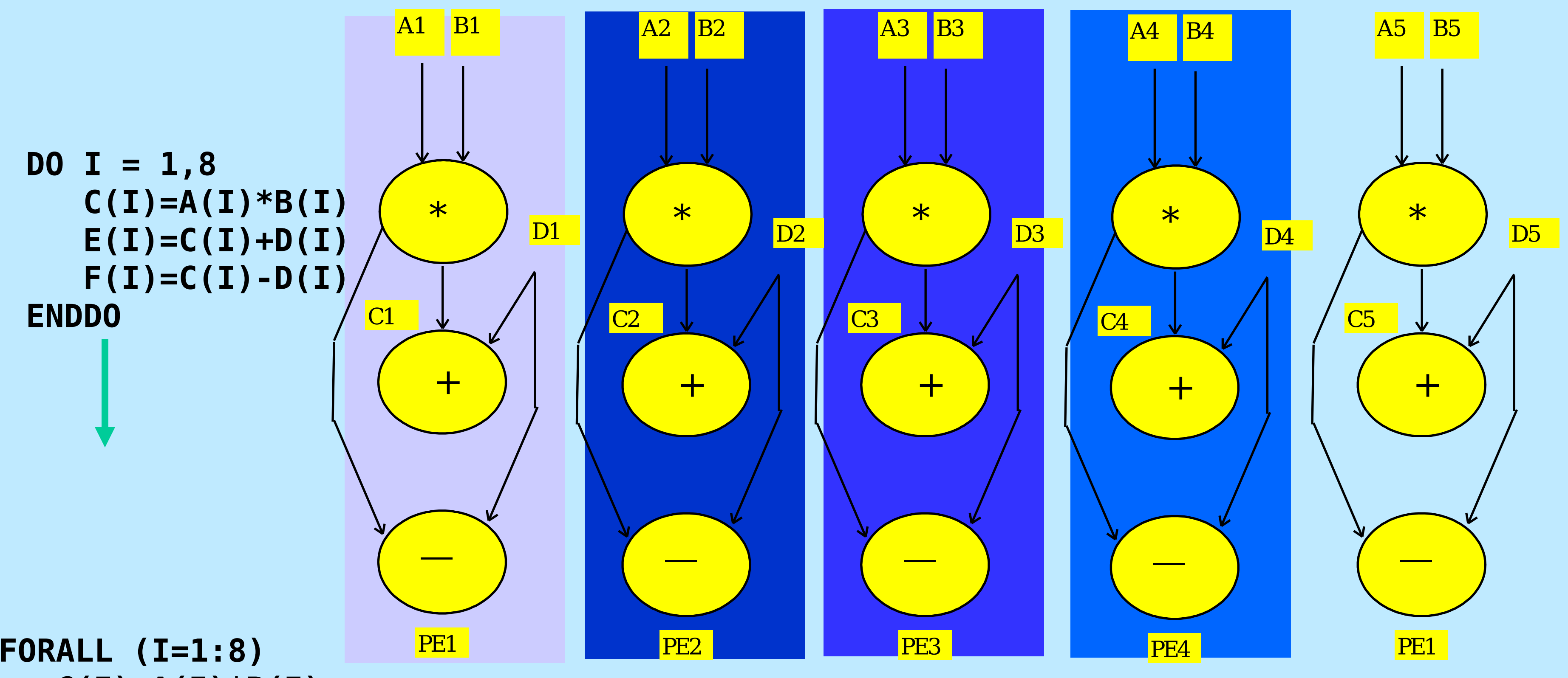


Schleifen-Parallelisierung
Vektorisierung



Schleifen-Parallelisierung
Iteration

```
DO I = 1,8
   C(I)=A(I)*B(I)
   E(I)=C(I)+D(I)
   F(I)=C(I)-D(I)
ENDDO
```



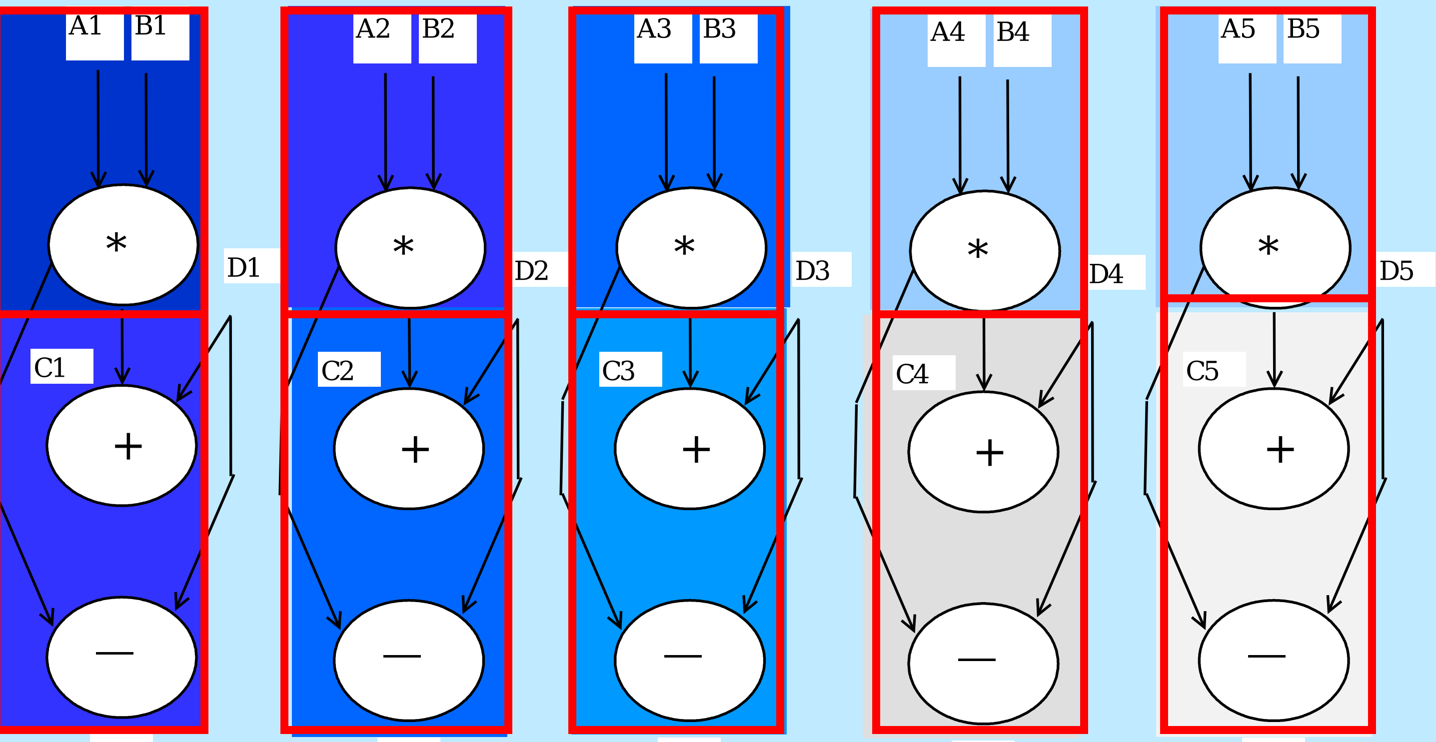

```
FORALL (I=1:8)
   C(I)=A(I)*B(I);
   E(I)=C(I)+D(I);
   F(I)=C(I)-D(I)
ENDPAR
```

explizite Anweisungen für die parallele Schleifenausführung in HPF, Fortran 95

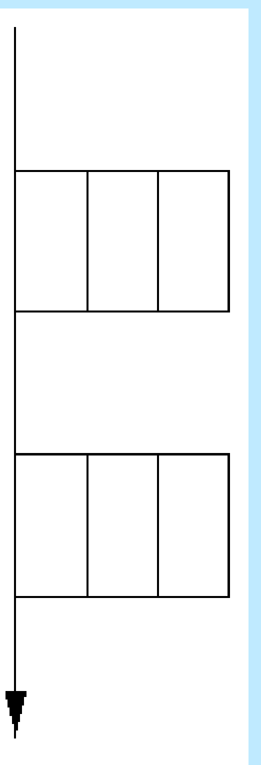
Schleifen-Parallelisierung Pipeline-Vektorprozessor

C=A*B
E=C+D
F=C-D

ein Multiplizierwerk und 2 Addierer /Sub-trahierer werden überlappend (pipe-lined) eingesetzt.



Synchronisation von Parallelarbeit Fork-Join Synchronisation



Sequentieller Code zur Initialisierung

Paralleler Code

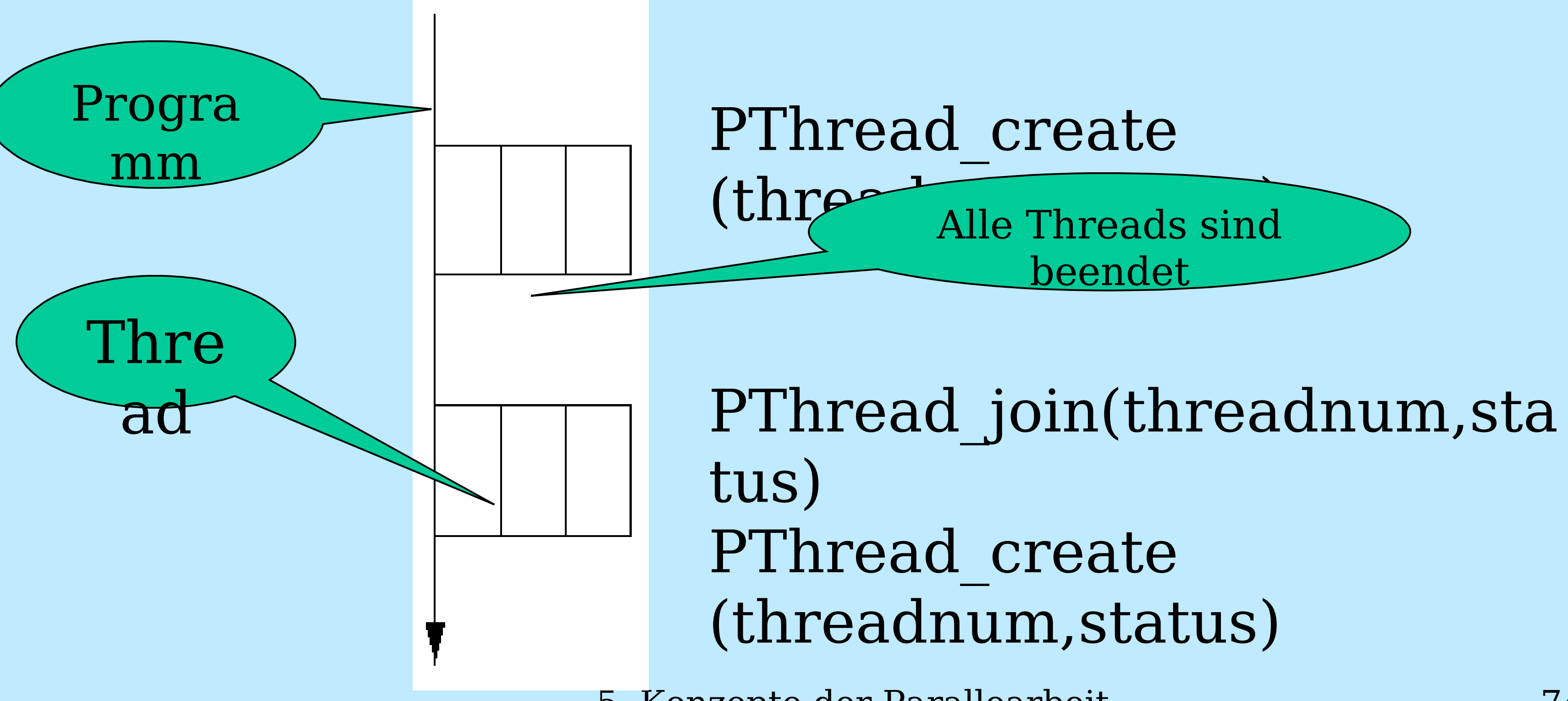
Sequentieller Code

Paralleler Code

Sequentieller Code zum Abschluß

Synchronisation von Parallelarbeit
Fork-Join Synchronisation bei SMP

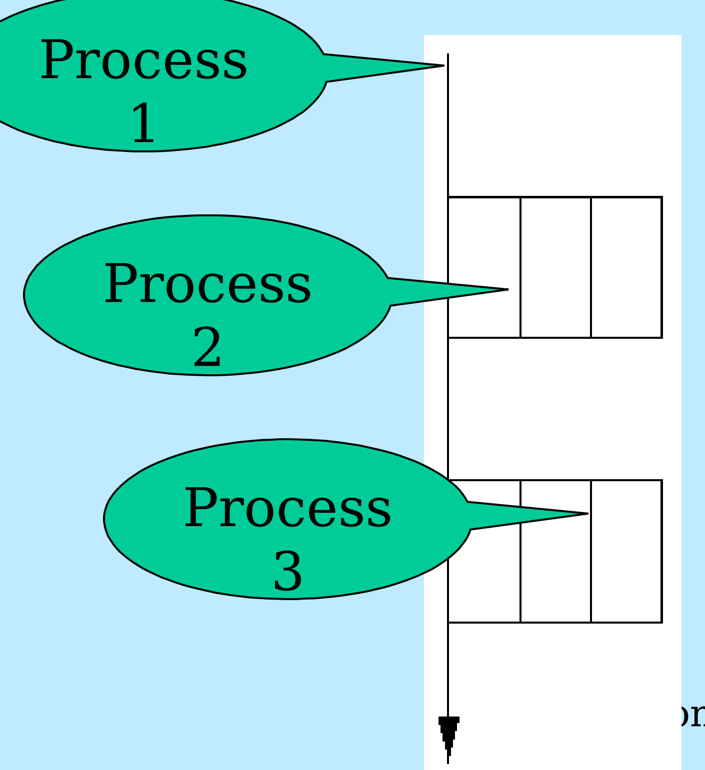
- Fork-Join Synchronisation bei SMP ist bei der Verwaltung von Threads (leichtgewichtigen Prozessen) realisiert (siehe Posix-Threads)



PThread_join(threadnum,sta

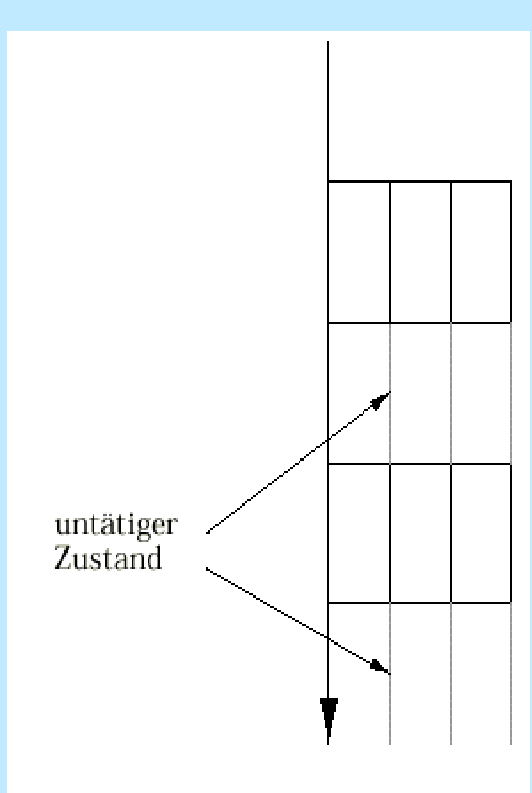
Synchronisation von Parallelarbeit Fork-Join Synchronisation nachrichtenorientiert

- Fork-Join Synchronisation bei nachrichtenorientierten Systemen wird mit ***send*** und ***receive*** einer Kommunikationsbibliothek realisiert (siehe MPI). Der parallele Teil wird von n Prozessen ausgeführt.



Send_message(process,message,status)
Receive_message(...anfangen ...)
Send_message (fertig)
Receive_message (...fertig...)

Synchronisation von Parallelarbeit Reusable-Thread-Pool-Modell nach Pancake



Sequentieller Code zur Initialisierung

Paralleler Code

Sequentieller Code aktiv, Threads sind suspendiert, nicht beendet
Threads wurden wieder aktiviert

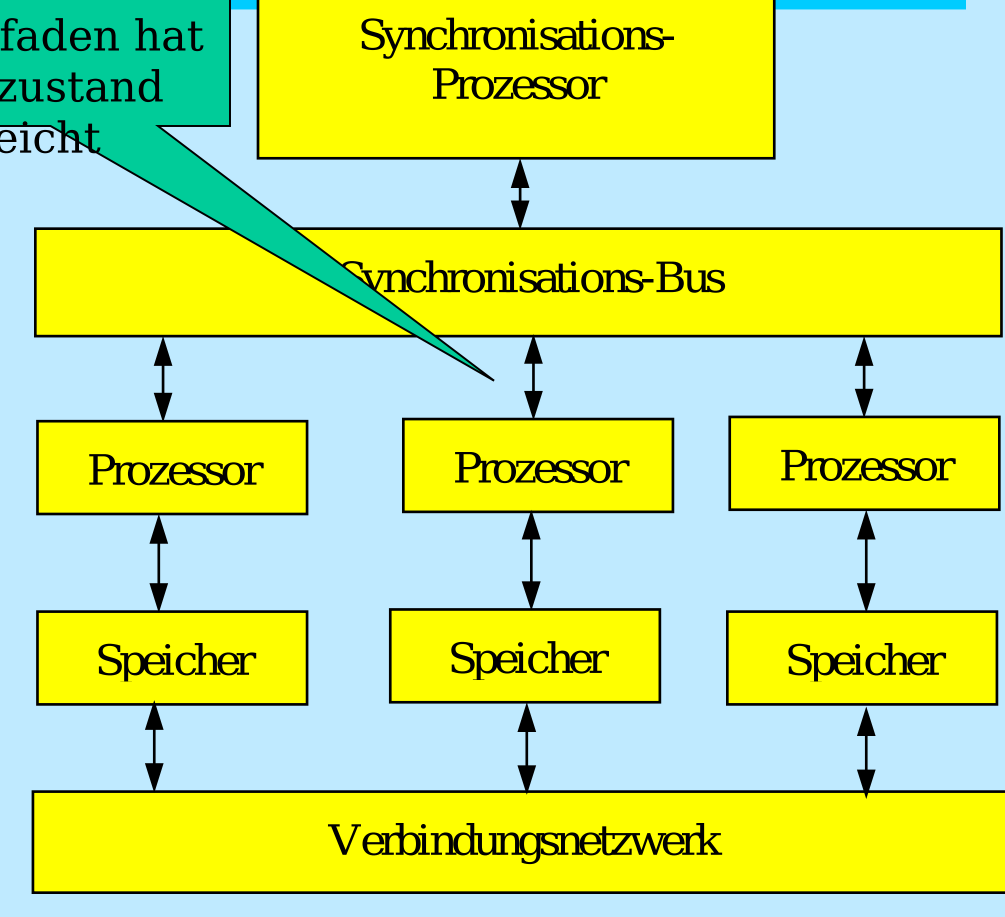
Sequentieller Code
zum Abschluß

Synchronisation von Parallelarbeit Reusable-Thread-Pool-Modell - Bewertung

- Vorteil
  - Aufwand zur Thread-Erzeugung wird minimiert. Jeder Thread wird nur einmal erzeugt und bei Bedarf geweckt.
- Nachteile
  - Längere Startzeit des Programmes
  - Ressourcenverschwendung, falls zur Laufzeit des Programmes einige Threads nicht benötigt werden

Synchronisation von Parallelarbeit Lock-Step Betrieb, Barrieren Synchronisation

- Spezielle Hardware (Synchronisationsbus) gestattet sehr schnelle Synchronisation
  - UND Verknüpfung
  - ODER Verknüpfung (paralleles Suchen)
- Step umfaßt alle Operationen eines Rechenschritts
- Lock umfaßt den Datenaustausch nach einem Rechenschritt



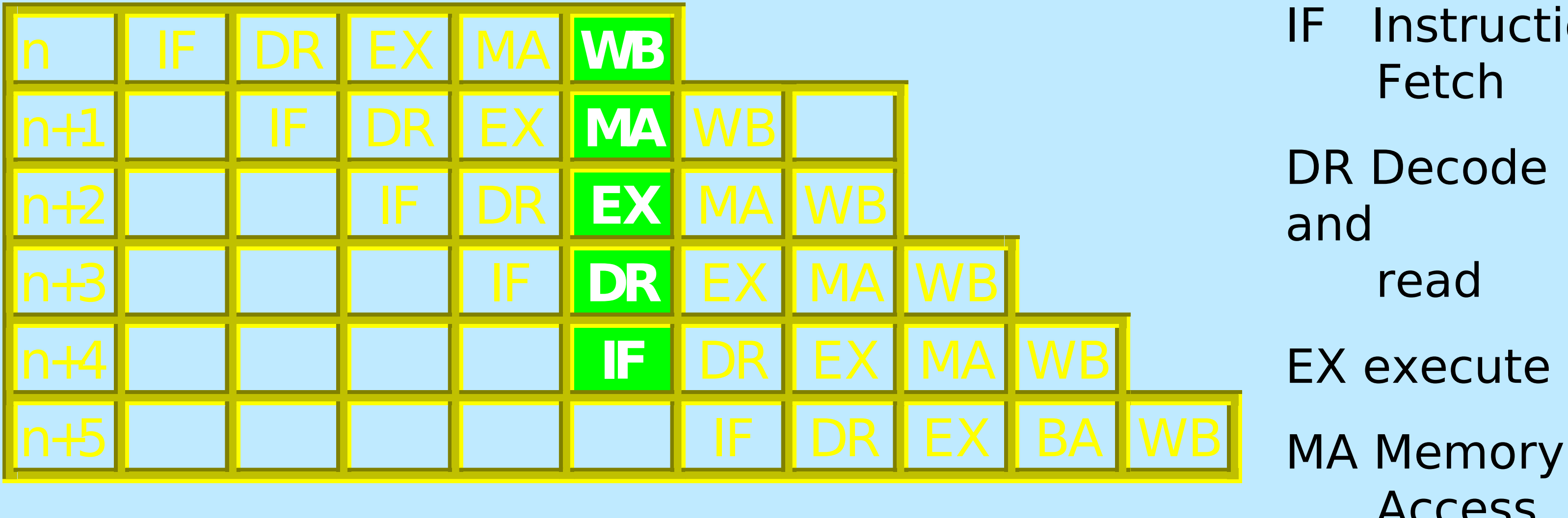
Synchronisation kooperierender Prozesse Allgemeine Regeln

- Die Kommunikationskanäle sind die einzigen Wege, über die Knoten kommunizieren können.
- Ein Kommunikationskanal überträgt Informationen innerhalb einer endlichen Zeit, im übrigen aber nicht beschränkten Zeit. Diese Zeit heißt *Kommunikationslatenz*.
- Ein Knoten ist zu jeder Zeit entweder mit der Ausführung einer Operation beschäftigt, oder er wartet auf eine Eingabe über einen seiner Eingabekanäle.
- Jeder Knoten verfügt über einen gewissen Speicherbereich.
- Jeder Knoten führt einen bestimmten Prozeß (single processing) oder eine Anzahl von Prozessen (multi processing) aus.

Prozeß-Kommunikationsmechanismen

- **Remote Process Invocation** — Ein Prozeß startet einen anderen Prozeß und fährt dann mit seiner Arbeit fort, ohne daß eine weitere Kommunikation mit dem initiierten Prozeß besteht. Kommunikationsart: Strenggenommen *keine Kommunikation*
- **Remote Procedure Call** — Der sendende Prozeß geht nach dem Aussenden einer Nachricht in den Wartezustand, bis der Empfänger den Vollzug des in der Botschaft enthaltenen Auftrags meldet. Kommunikationsart: *synchron*
- **Rendezvous** — Der Senderprozeß stellt an den Empfängerprozeß den Antrag auf ein Rendezvous und geht unmittelbar danach in den Wartezustand, in dem er verbleibt, bis der Empfänger seine Bereitschaft zur Kommunikation durch eine entsprechende Rückantwort gezeigt hat. Der Antrag enthält bereits eine Beschreibung der Objekte, die der Senderprozeß dem Empfänger übermitteln möchte. Dies setzt den Empfängerprozeß in die Lage, zu prüfen, ob er zum Empfang der Nachricht bereit ist. Erst dann meldet er sich zurück. Nachdem auf diese Weise eine Synchronisation zwischen Sender und Empfänger hergestellt worden ist (in ADA Rendezvous genannt), kann die eigentliche Kommunikation zwischen den Prozessen beginnen. Z.B. können Datenobjekte vom Speicherraum des Senders in den Speicherraum des Empfängers kopiert werden.Kommunikationsart: *synchron*
- **No-Wait-Send** — Der sendende Prozeß fährt nach dem Aussenden der Botschaft in seiner Arbeit fort, bis er an die Stelle kommt, wo er ohne Erhalt einer Rückantwort nicht weiterarbeiten kann und gegebenenfalls darauf warten muß.Kommunikationsart: *asynchron*

Phasenpipelining - das Prinzip



IF Instruction Fetch

DR Decode and read

EX execute

MA Memory Access

WB Write back

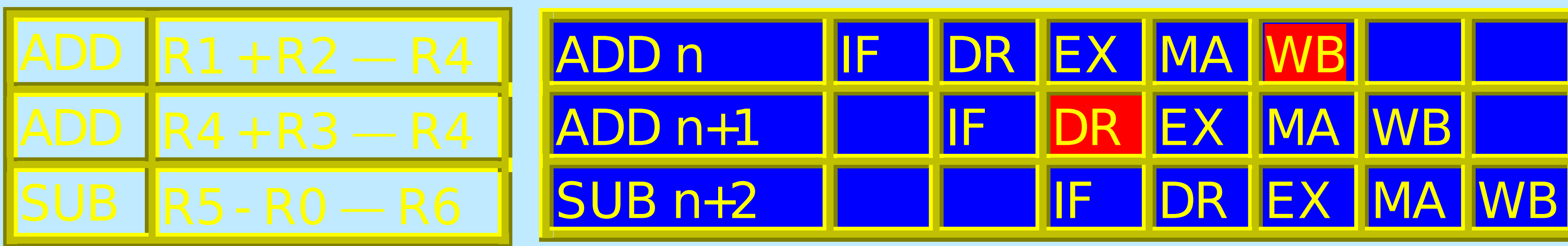
- Ist eine Implementierungstechnik, die zeitlich überlappendes Bearbeiten von Befehlen ermöglicht.
- Keine Verringerung der Latenzzeit eines einzelnen Befehls
- Erhöhter Durchsatz von abgearbeiteten Befehlen pro Zeiteinheit

Phasenpipeline - Ladekonflikte

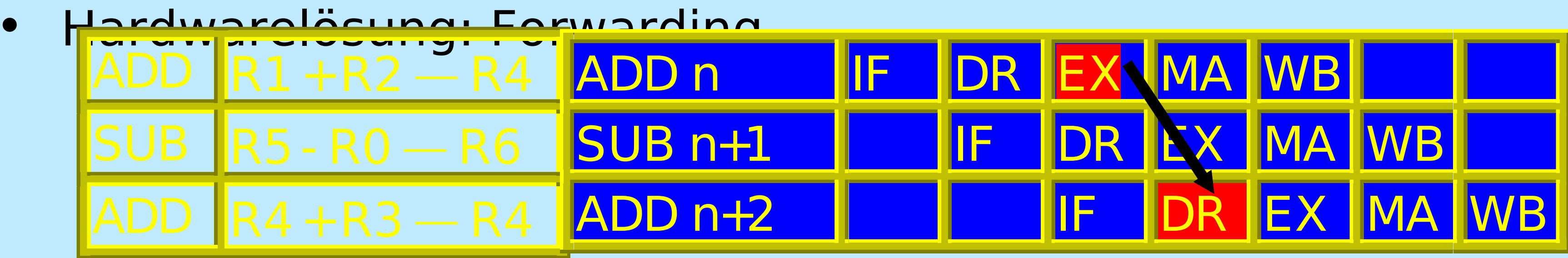
- Benötigt ein Befehl das Ergebnis eines vorherigen LOAD, müssen entsprechend NOPs eingefügt werden !
- Muß das Datum aus dem Hauptspeicher geholt werden, wird die ganze Pipeline entsprechend viele Taktzyklen angehalten !
- LOAD Befehle machen ~ 18% des Befehlsstroms aus !

LOAD R1,(R2)	IF	DR	EX	MA	WB				
NOP		IF	DR	EX	MA	WB			
NOP			IF	DR	EX	MA	WB		
NOP				IF	DR	EX	MA	WB	
ADD R1, R3					IF	DR	EX	MA	WB

Phasenpipeline - Datenflußkonflikt



- Softwarelösung : Compiler sortiert Code um oder fügt NOPs ein
- Hardwarelösung: Forwarding

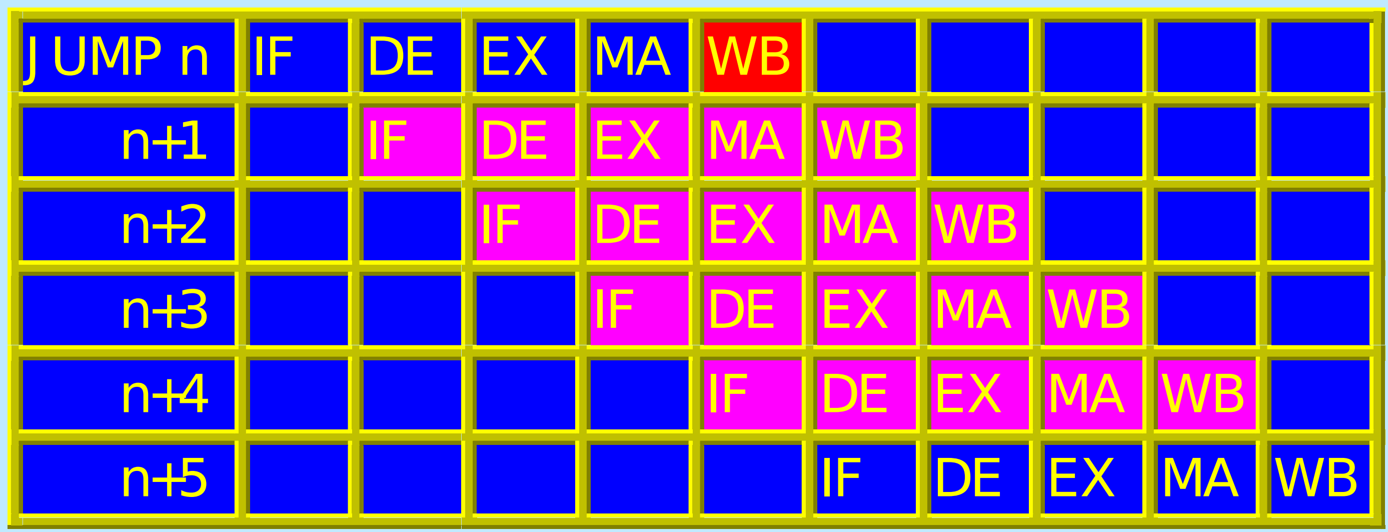


Phasenpipeline - Strukturelle Konflikte

LOAD n	IF	DR	EX	MA		WB					
STORE n+1		IF	DR	EX		MA		WB			
ADD n+2			IF	DR		EX		MA	WB		
ADD n+3					IF	DR		EX	MA	WB	
ADD n+4							IF	DR	EX	MA	WB

- Memory Access und Instruction Fetch benötigen Datenbus
- Getrennte Befehls- und Datenspeicher vermeiden strukturelle Konflikte (Harvard Architektur)

Phasenpipeline - Steuerkonflikte



JUMP n	IF	DE	EX	MA	WB					
n+1		IF	DE	EX	MA	WB				
n+2			IF	DE	EX	MA	WB			
n+3				IF	DE	EX	MA	WB		
n+4					IF	DE	EX	MA	WB	
n+5						IF	DE	EX	MA	WB

- Die Befehle n+1 bis n+4 müssen durch NOPs ersetzt werden
- Sprünge machen ~ 15 % des Befehlsstroms aus !
- deshalb: Branch Delay Slots - Branch Prediction - Sprungvermeidung (Alpha: Conditional Move Befehle)

Phasenpipeline - Branch Delay Slots

Adresse	normaler Sprung	verzögerter Sprung	optimierter Sprung
100	LOAD X,A	LOAD X,A	LOAD X,A
101	ADD 1,A	ADD 1,A	JUMP 105
102	JUMP 105	JUMP 106	ADD 1,A
103	ADD A,B	NOP	ADD A,B
104	SUB C,B	ADD A,B	SUB C,B
105	STORE A,Z	SUB C,B	STORE A,Z
106		STORE A,Z	

Phasenpipeline - Häufigkeit von Befehlsklassen

Befehl	Pipeline-Konflikte	dynamische Häufigkeit in %
bedingte Sprünge	Steuerkonflikt	11
unbedingte Sprünge	Steuerkonflikt	4
Load	Ladekonflikt/ Struktureller Konflikt	18
Store	Struktureller Konflikt	8
Arithmetik	Datenkonflikt	39
Rest		20

Phasenpipeline - Pipelinekonflikte

Die Vermeidung und Auflösung der viel-fachen Konflikte nimmt eine zentrale Be-deutung in der Prozessorarchitektur ein

- Forwarding
- Sprungvermeidende Befehle
- Harvard Architektur
- große L1 und L2 Caches
- Branch Delay Slots
- Branch Prediction
- Branch History Table
- Branch Target Buffer
- optimierende, codesortierende Compiler !

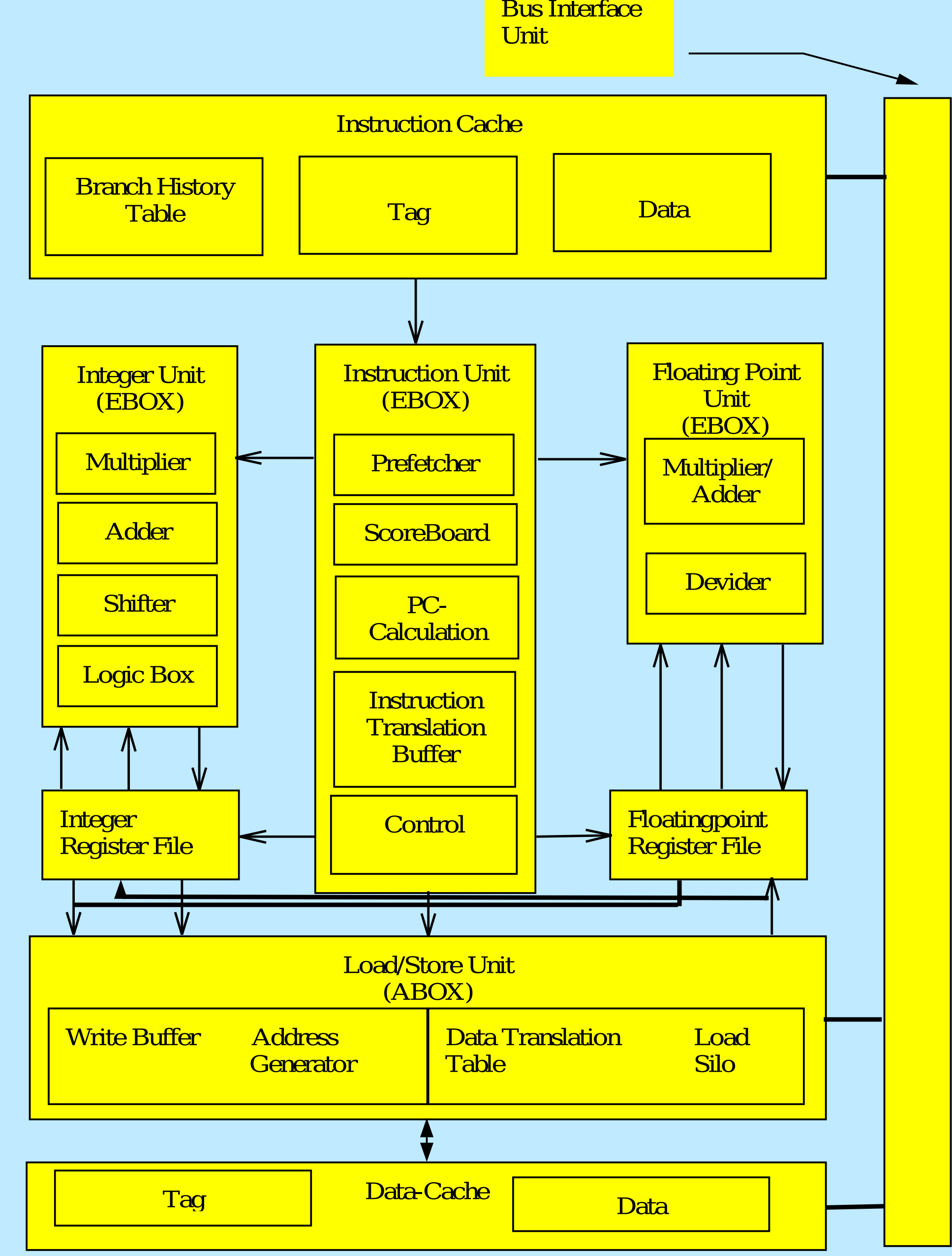
Je tiefer die Pipeline, desto schlimmer wirken sich Pipelinekonflikte aus!

	ALU. .Load /Store
Pentium	5
P6	12/17
NexGen	7
MIPS R4600	5
MIPS R10000	5/7
PowerPC 604	6
PowerPC 620	5
Alpha 21x64	7
SuperSPARC	4
UltraSPARC	5
PA-RISC 8000	7 / 9
Pentium III	12 /17
Pentium 4	20

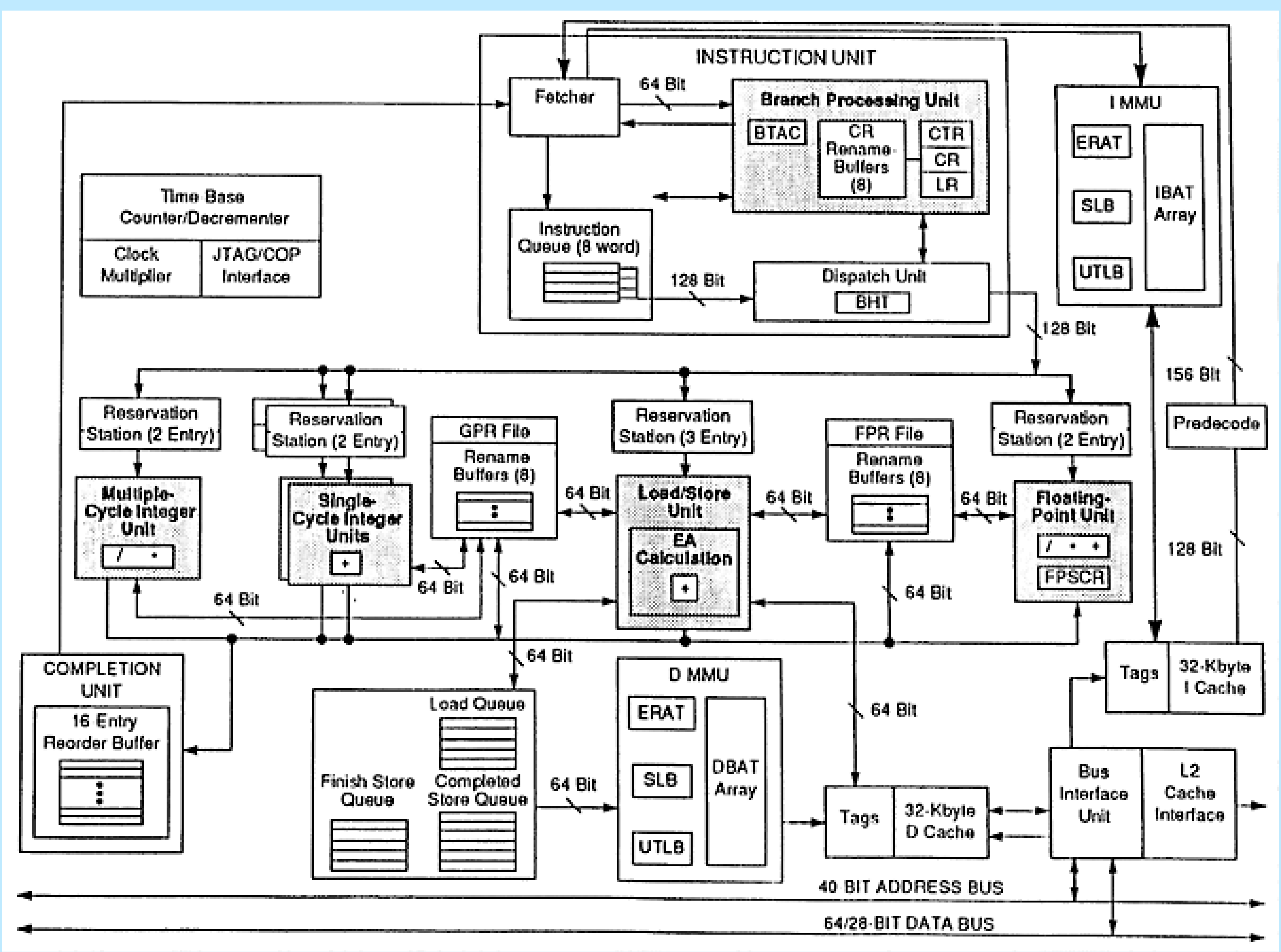
Funktionspipeline – Superskalare Prozessoren

- Die Schranke des Phasen-Pipelinings liegt bei einem CPI (Cycles per Intruction – Taktzyklen pro Befehl) von bestenfalls eins.
- Eine weitere Verbesserung, also Instruction per Cycle, erfordert die gleichzeitige Verarbeitung von mehr als einem Befehl in mehr als einer Ausführungseinheit.
- Diese Superskalar-Technik ist eine Implementierungstechnik, die aus einem **sequentiellen Befehlsstrom** pro Takt **mehr als einen Befehl** den Ausführungseinheiten eines Prozessors zuordnen kann
- Funktions-Pipelining

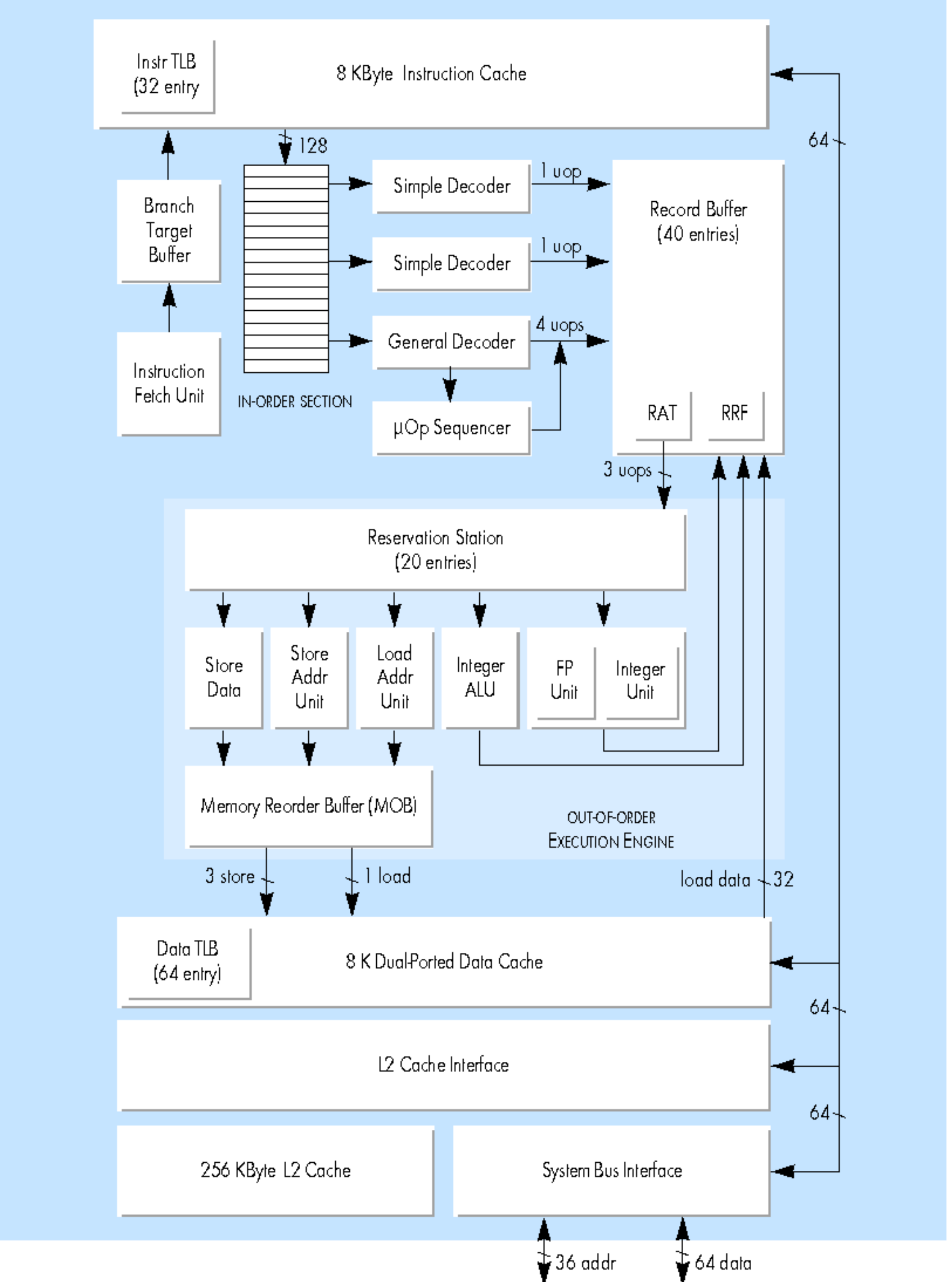
Superskalare Prozessoren: Blockschaltbild des Alpha 21064



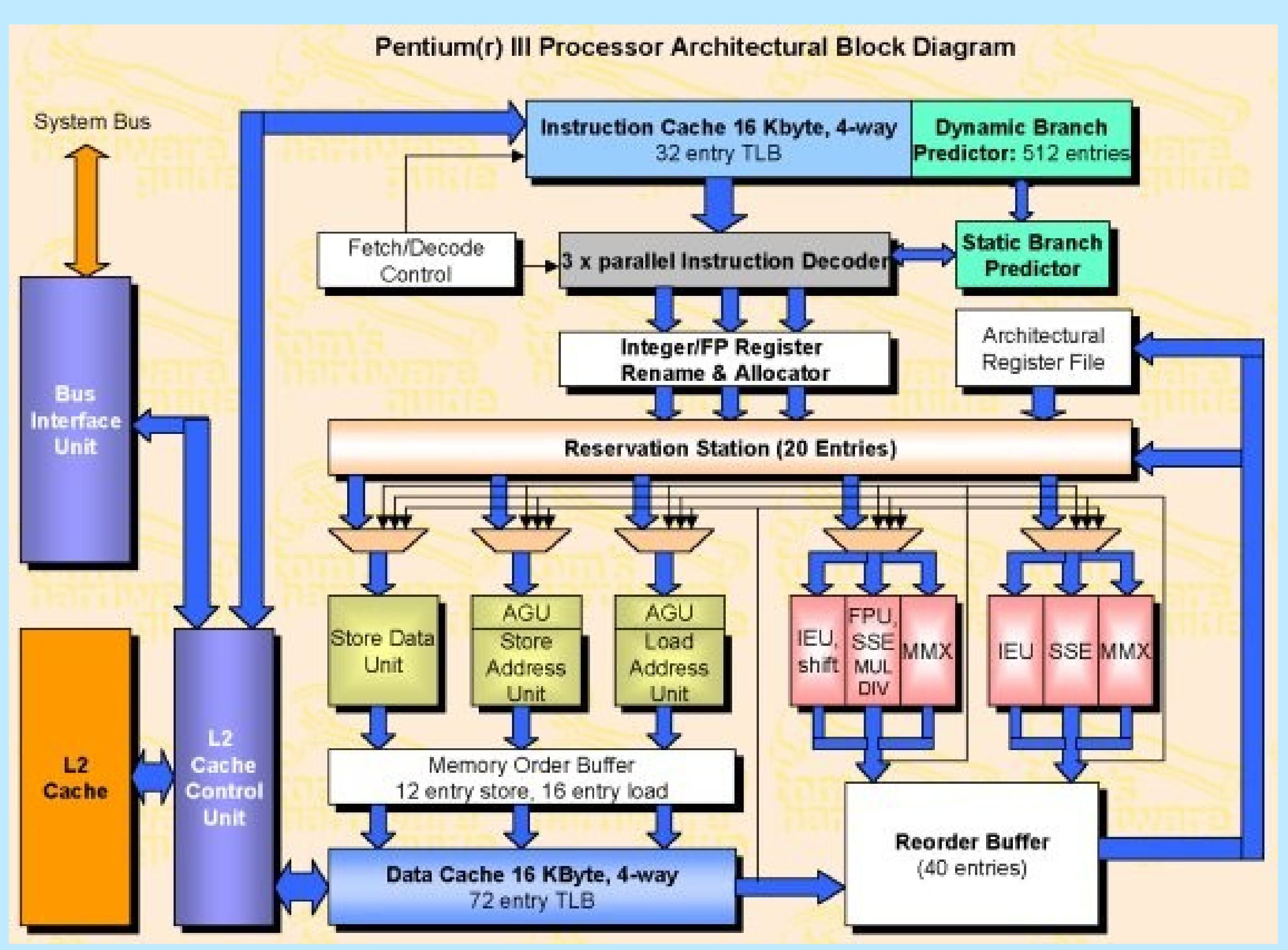
Blockschaltbild des PowerPC 620



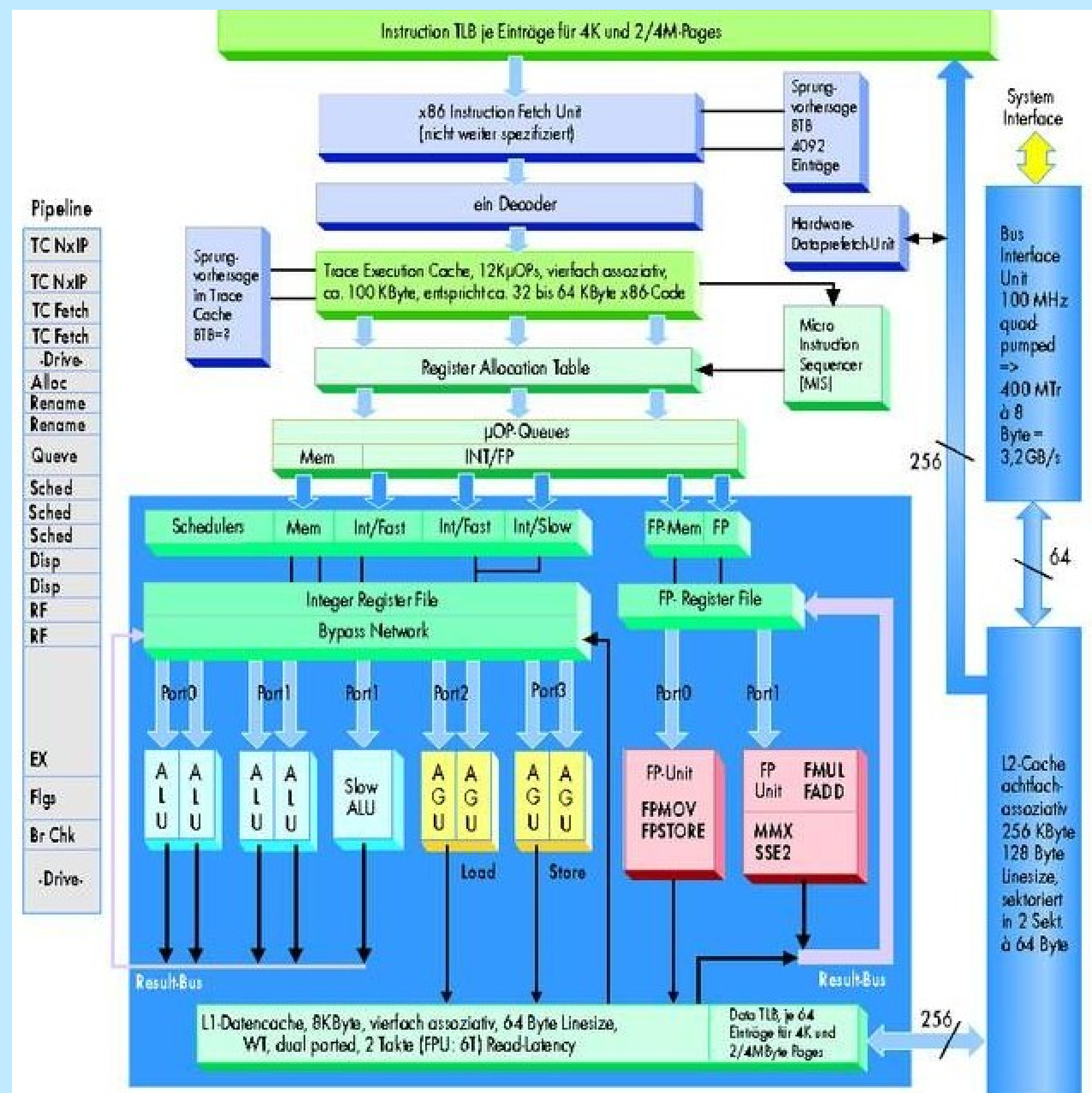
Blockschaltbild des Pentium II



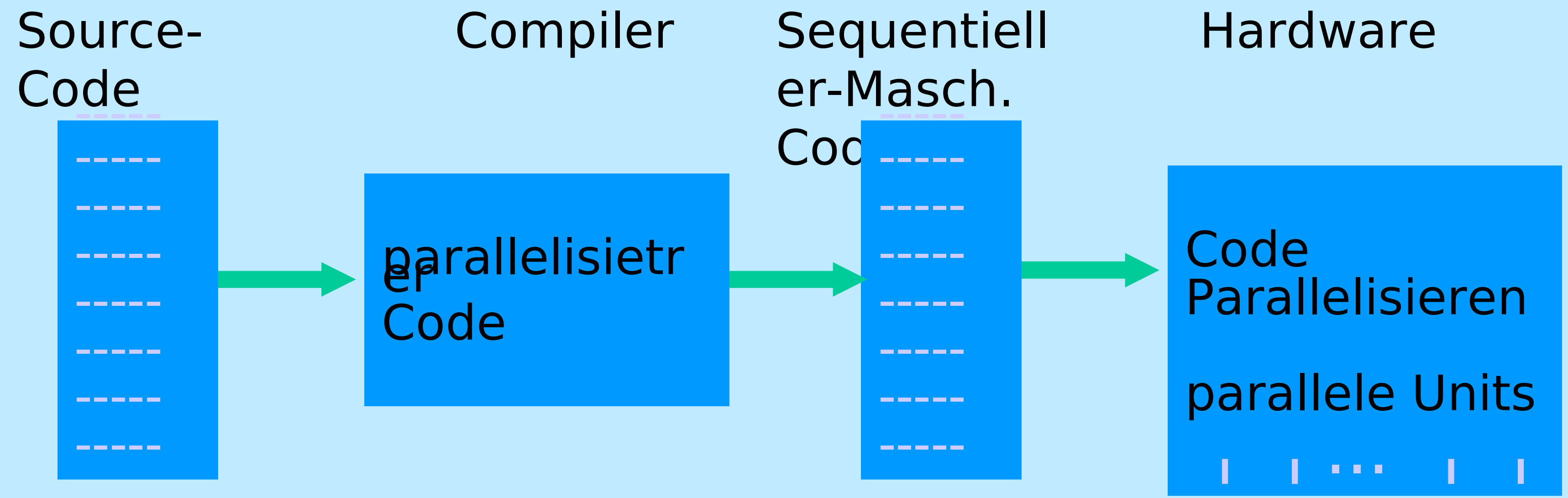
Blockschaltbild des Pentium III



Block-schalt-bild des Pentium 4



Probleme superskalarer Befehlsausführung



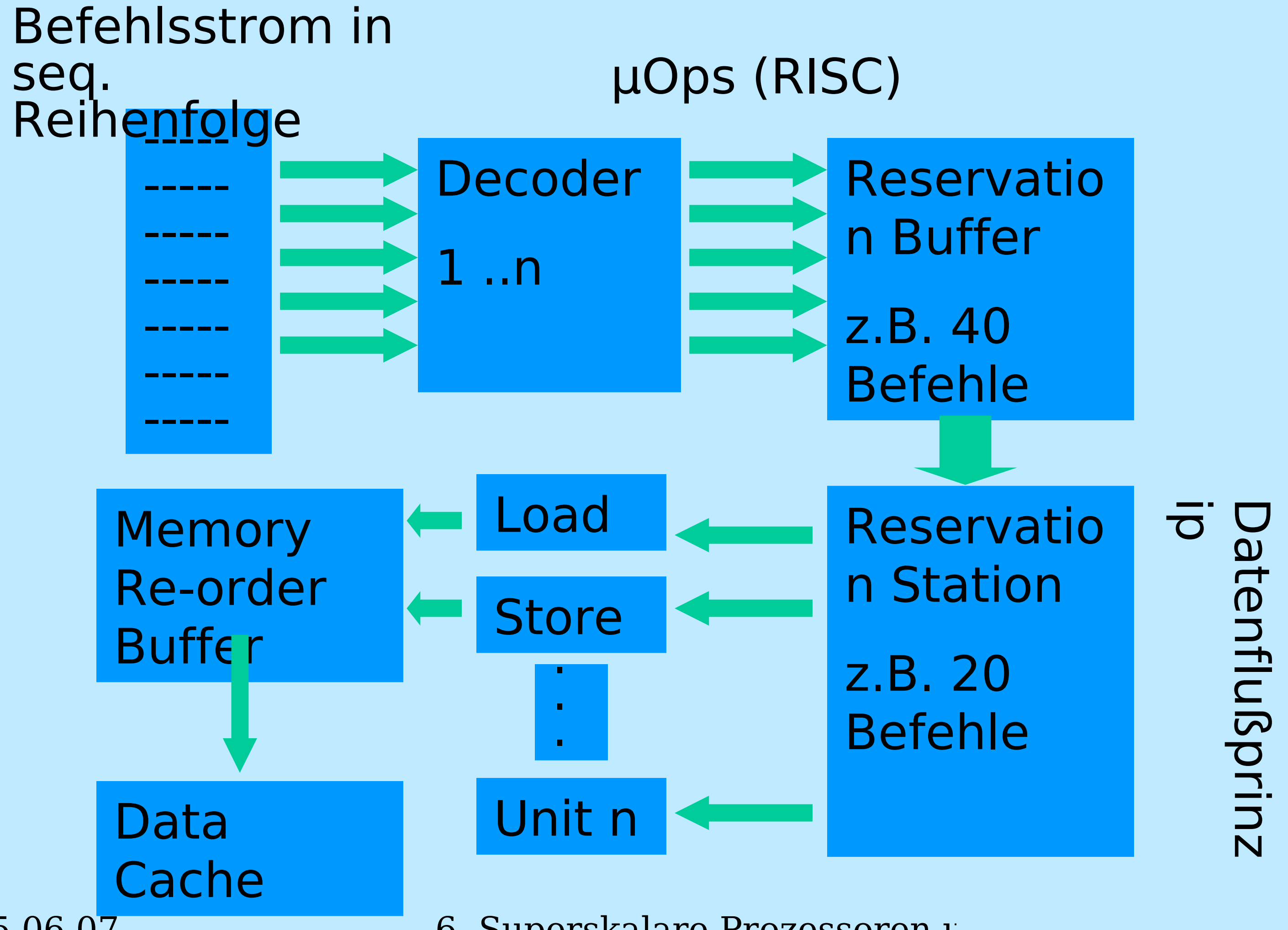
Der vom Compiler erzeugte **sequentielle** Befehlsstrom soll zur Laufzeit **parallel** von den Verarbeitungseinheiten ausgeführt werden.

- Datenabhängigkeiten,die eine sequentielle Ausführung erfordern, verhindern zeitweise die superskalare Bearbeitung
- Prozessoren werden mit zusätzlicher Logik ausgestattet, um die Datenabhängigkeiten möglichst aufzulösen

Techniken Superskalarer Befehlsausführung: Dynamic (Out of Order) - Execution

- Ziel: optimale Ausnutzung der Verarbeitungseinheiten
- Idee: Die Instruktionen liegen aufbereitet in einem Zwischenspeicher (reservation station). Diejenigen, deren Operanden bereitstehen und für die eine Verarbeitungseinheit frei ist, werden im nächsten Zyklus ausgeführt. (Datenflußprinzip)
- Problem: Die Ergebnisse der Operationen müssen nachträglich in der eigentlich erwünschten Reihenfolge ausgegeben werden. Dafür gibt es einen zusätzlichen Reorder Buffer

Dynamic Execution beim Intel Pentium II



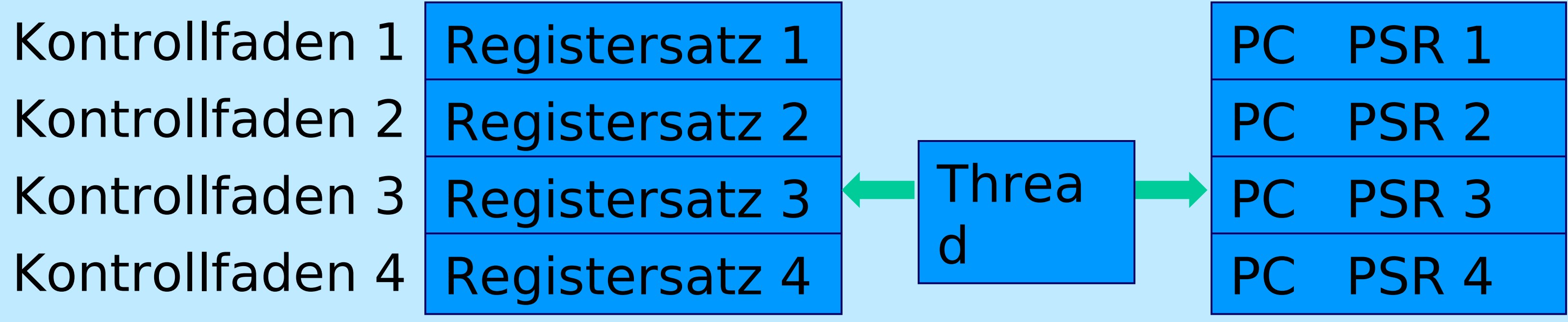
Probleme der Superskalartechnik: Datenabhängigkeiten

- Datenflußabhängigkeit
- Antiabhängigkeit
- Ausgabeabhängigkeit

I1	R3 = R3 op R5
I2	R4 = R3 + 1
I3	R3 = R5 + 1
I4	R7 = R3 op R4

Datenflußabhängigkeiten (I1,I2) sind nicht auflösbar
Antiabhängigkeiten (I3, I2) und Ausgabeabhängigkeiten (I1,I3) sind mit **Register Renaming** auflösbar.

Erweiterung der Superskalarität
Vielfädige superskalare Prozessoren



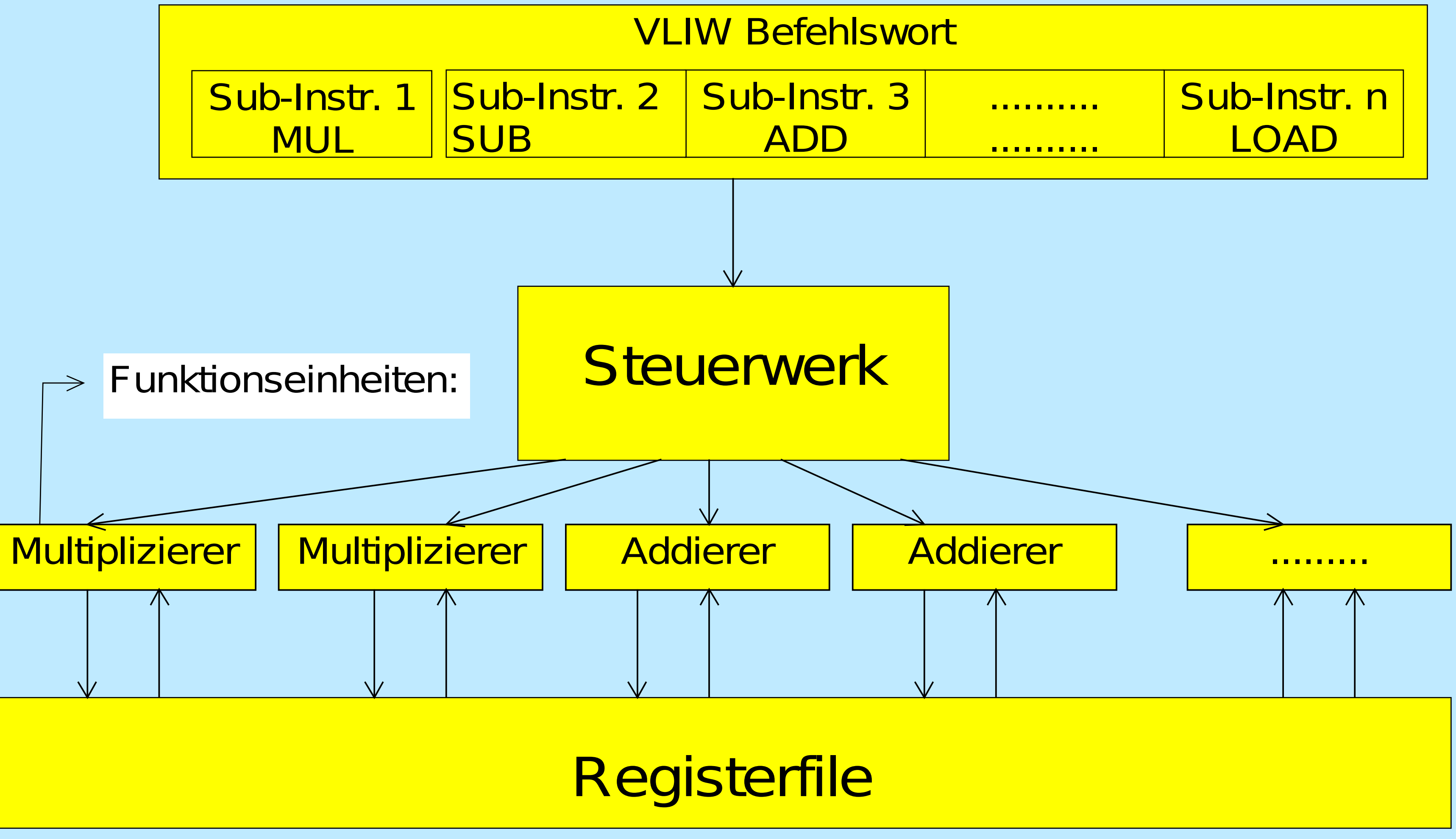
Ziel: Wartezeiten überbrücken, Ausführungseinheiten ständig beschäftigen

Lösung: Mehrere Kontrollfäden (Threads) sind in verschiedenen Registersätzen auf dem Prozessor geladen. Ein schneller Kontextwechsel geschieht durch Umschalten auf einen anderen Registersatz

Grenzen der Superskalarität

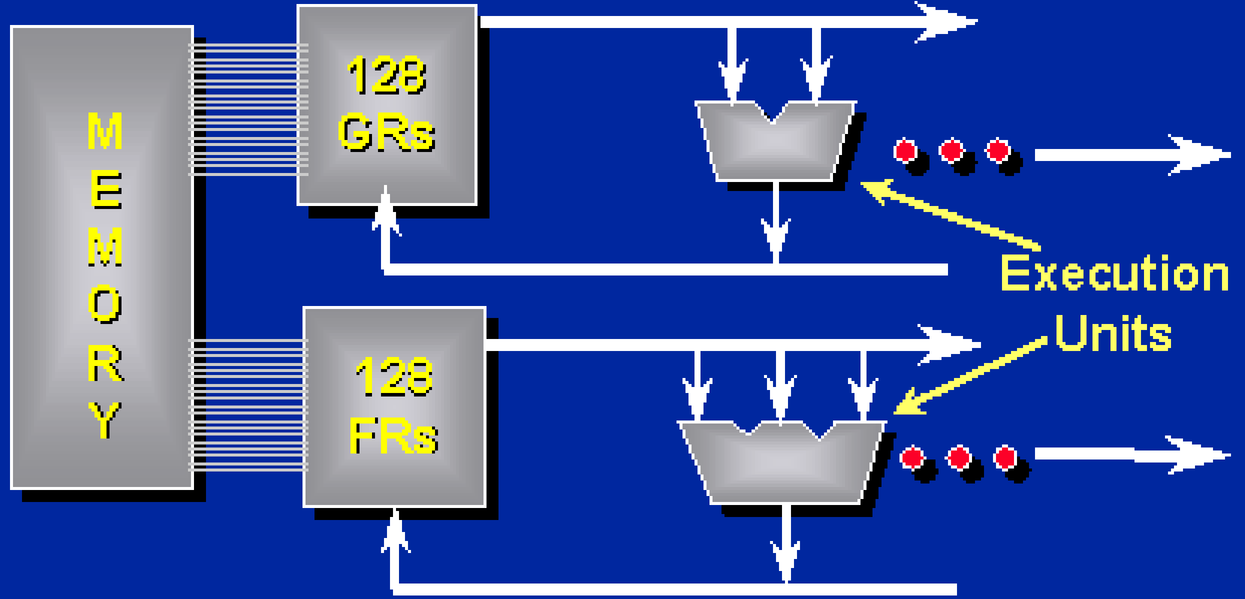
- Je mehr Funktionseinheiten eingesetzt werden, desto wirkungsvollere Compiler-Optimierungen sind notwendig, um parallel ausführbaren Code zu erzeugen. Die Parallelisierbarkeit steigt aber nicht proportional mit der Anzahl von Funktionseinheiten
- Superskalare Pipeline- Prozessoren sind nur so gut wie ihre Compiler, die genaue Kenntnis der Hardware benötigen !
- Ein Lösungsansatz heißt:
  - VLIW - Very Large Instruction Word
  - EPIC - Explicit Parallel Instruction Computing
- EPIC ist die Intel-Neuauflage des alten VLIW Prinzips

VLIW / EPIC - Konsequent Superskalar



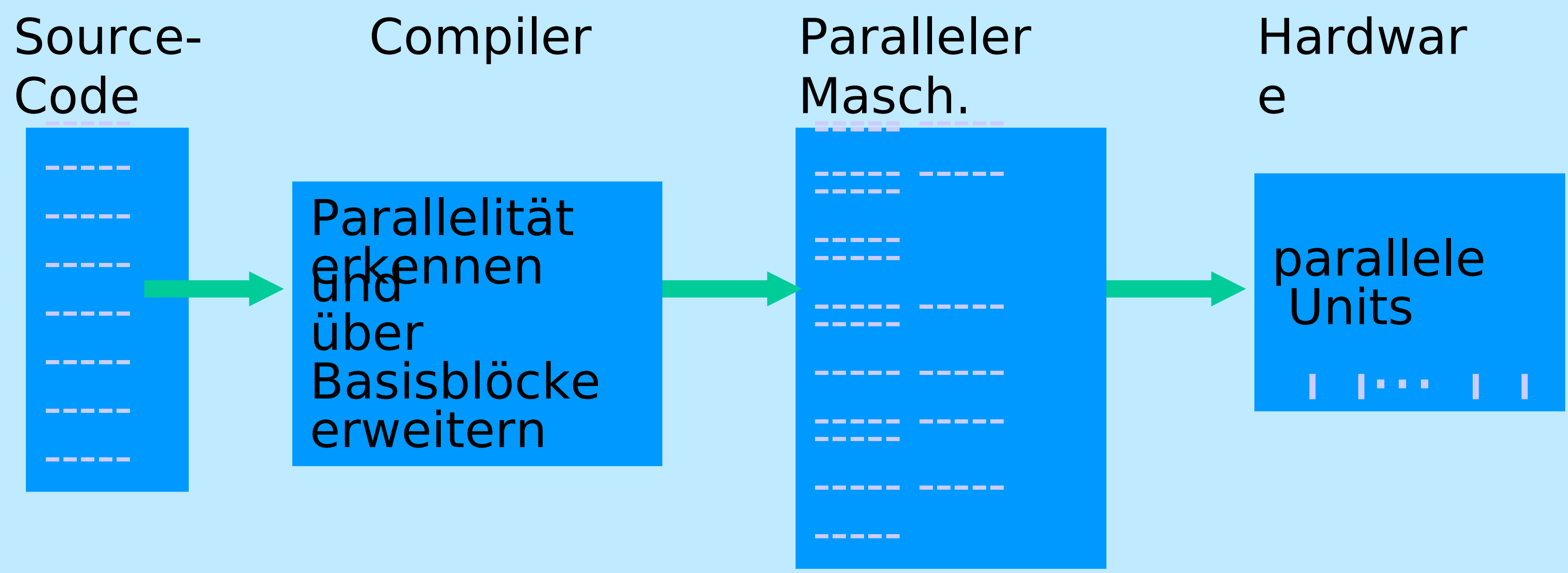
EPIC - Intels IA-64 Architektur

Architecture Resources Provide for Parallel Execution & Scalability



- Massively resourced - large register files
  - Traditional architectures are forced to rename registers
- Inherently scalable - replicated function units
- Explictly parallel - transistors used more effectively

EPIC - Explizit parallele Strategie



Der Compiler erkennt die Parallelität im Source-Code, analysiert sie über die Basisblöcke hinaus und bereitet die Befehle explizit als parallelen Code für die Hardware auf

.

EPIC - Explizit parallele Strategie

Nachteile

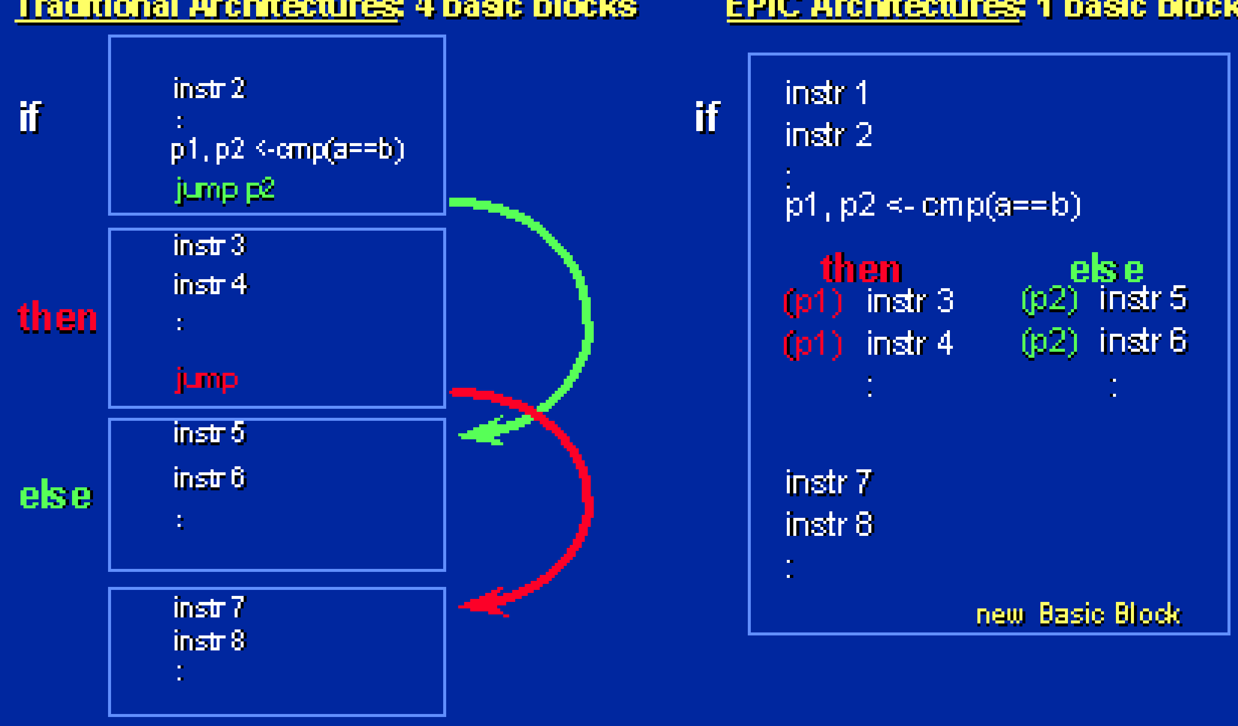
- Verzicht auf Phasenpipelining
- Geringere Taktfrequenz wegen Multiport Register File
- Bei schlecht parallelisierbarem Code sinkt die Performance auf die eines relativ langsam getakteten skalaren Prozessors (fehlendes Phasenpipelining)

Vorteile

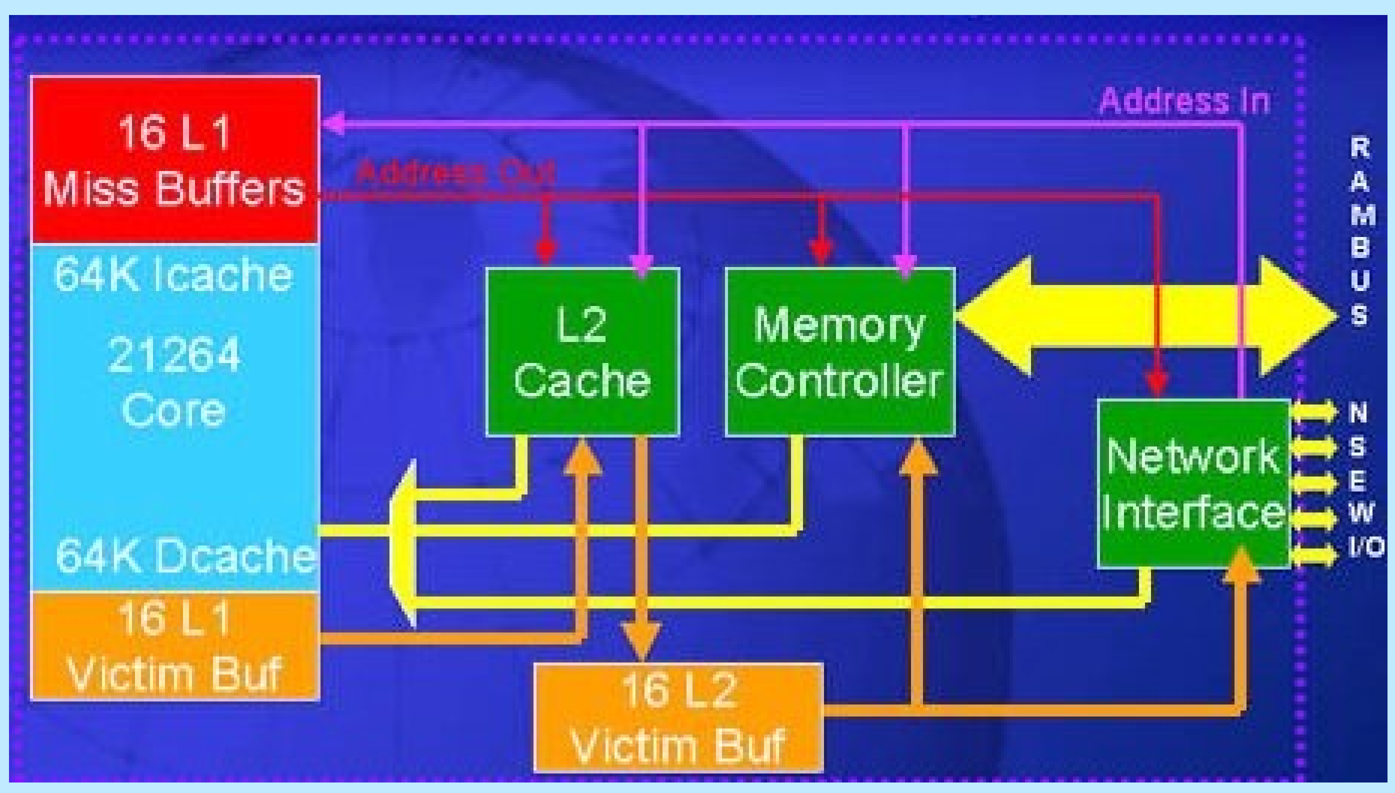
- einfacheres Prozessordesign wegen fehlender Dynamic Execution
- einfacheres Prozessordesign, weil Pipelinekonflikte nicht aufgelöst werden müssen
- dadurch zusätzliche Resourcen für Branch Predication

EPIC - Intels IA-64 Architektur

- Branch Predication Den Befehlen eines Basis-Blocks wird ein Prädikat vom Compiler verliehen, zu welchem Zweig sie gehören: IF- als auch ELSE Zweig können so gleichzeitig ausgeführt werden. Ist die Bedingung geprüft, kann der falsche Zweig eliminiert und die Ergebnisse des



Alpha 21364 Chip-Blockdiagramm



- Der integrierte **1.5 MB große L2-Cache** und der **integrierte Memo-ry Controller** unterstützen die Ein-Prozessor Performance
- Durch das **integrierte Network-Interface** eignet sich der 21364 gut für Hochleistungs-Multiprozessorsysteme

Zusammenfassung: Superskalare Prozessoren und VLIW-Maschinen

Senken von CPI

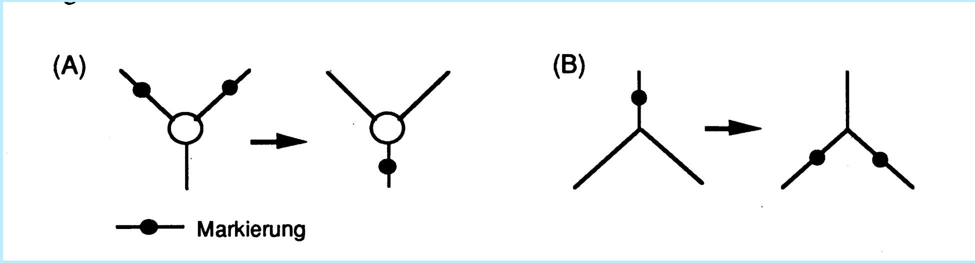
- Einführung von **Phasenpipelining**
  Beginn der RISC Ära
  Ziel: CPI auf eins senken
  Problem: Pipelinekonflikte
  Lösung: Spezielle Hardware
  Optimierende Compiler
- Einführung von **Funktionspipelining**
  Einsatz mehrerer parallel arbeitender
  Units - **Superskalare Prozessoren**
  Register-Renaming
  Out of Order Execution
  Ziel: CPI soll deutlich unter eins
  liegen oder: IPC > 1
  Problem: sequentieller Code schränkt

- **Vielfädig superskalare Prozes-soren**
  Auslastungslücken der Units sol-len durch Mischen von Befehlen
  aus mehreren Befehlsströmen
  geschlossen werden.
  Ziel: Steigern von IPC auf >4
  Problem: zusätzlicher Hardware-aufwand für die Organisation des
  Befehlsstromes im Prozessor
- **Single Chip Multiprocessor**
  Ziel: Steigern von IPC auf >4
  Problem: Vervielfachung der
  Superskalarprobleme

Datenflußrechner

- Verkörpern eine Architektur, bei der die Maschine jede Möglichkeit der Parallelarbeit - Datenparallelität wie Programmparallelität – selbst erkennen kann
- ***Grundidee:*** die Ausführung eines Algorithmus wird nicht durch einen vorgegebenen Kontrollfluß gesteuert, sondern die Maschine entscheidet selbst zur Laufzeit, welche Operationen in welchen Zeitpunkten ausgeführt werden.
- ***Grundlage:*** ein Datenflußgraph, der Operationen des Algorithmus und die zwischen ihnen bestehenden Datenabhängigkeiten angibt.
  - Jeder **Knoten** des Datenflußgraphen repräsentiert eine **Operation**.
  - Die **Operanden** der Operationen werden durch **Marken (token**) repräsentiert, mit denen die Eingänge des Knotens belegt werden können.
  - ***Zündregel-Semantik:*** Erst wenn jeder Eingang mit einer Marke belegt ist, d.h. wenn alle Operanden vorhanden sind, kann die Operation ausgeführt, **gezündet** werden.

Datenflußrechner



Zündregel für einen Akteur (A) und einen Verbindungsknoten (B)

Datenflußrechner

```
input u, v, w;
x = u + v*w;
y = u — v*w;
output x, y;
```

Bild 9-3 zeigt das zugehörige DFS. Dieses besteht aus den 8 Akteuren S1 bis S8 und den 6 Verbindungsknoten L1 bis L6. Die Akteure Sl, S2, S3 bzw. S7, S8 übernehmen die Eingabe bzw. die Ausgabe.

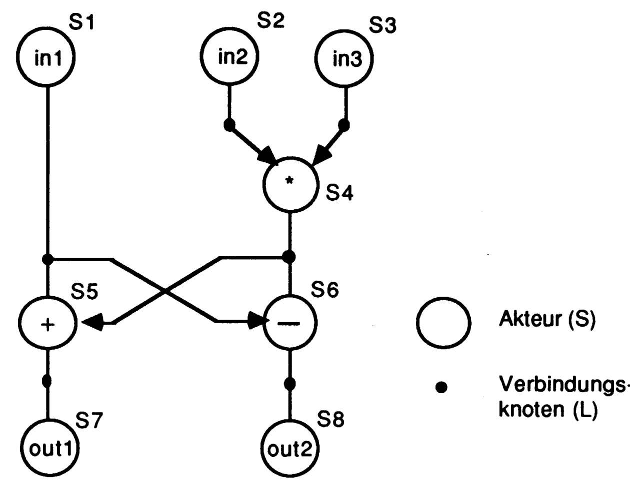


Bild 9-3 Datenflußschema für eine *FFT-Butterfly*-Operation (ohne Markierungen)

Eine Aktivität kann gezündet werden, wenn alle Eingabewerte verfügbar geworden sind. Daten sind mehr als die zu manipulierenden Objekte. Sie sind die treibende Kraft bei der Programmaus-führung (***datengetrieben***). Dies erfordert, daß ein Akteur seine Eingabedaten ***konsumieren*** muß, damit diese nicht einen weiteren Akteur zünden können. Gleichzeitig produziert die Ausführung des Akteurs Ausgabewerte, die ihrerseits Eingabewerte nachfolgender Akteure sein können. Damit werden ***alte Werte*** konsumiert und ***neue Werte*** erzeugt, aber

Datenflußrechner

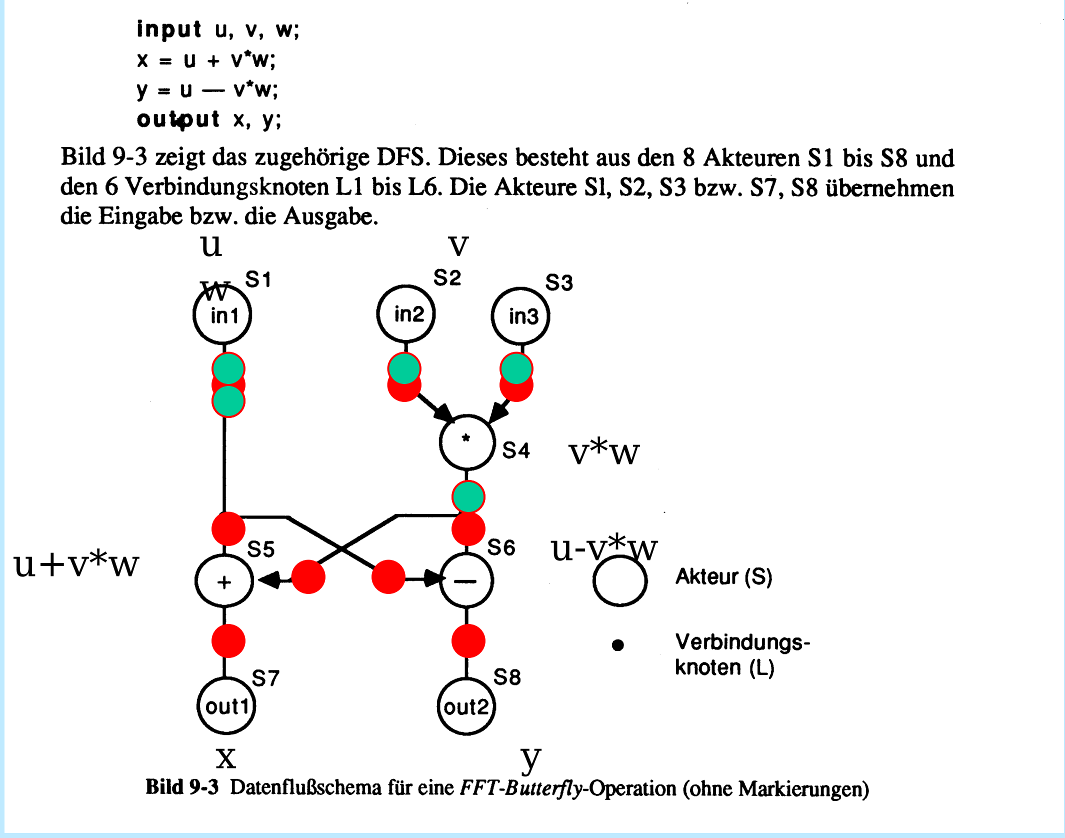


Bild 9-3 Datenflußschema für eine *FFT-Butterfly*-Operation (ohne Markierungen)

Datenflußrechner

- Das Datenflußprinzip hat in Bezug auf Parallelarbeit den Vorteil der Nähe zum **funktionalen Programmierstiel**, da Variablen nur **einmal** beschrieben werden. Variablen im eigentlichen Sinne gibt es nicht, ein Datum wird einmal **erzeugt** und wieder **verbraucht**.
- Die **Zündregel-Semantik** in Verbindung mit dem Prinzip der **einmaligen Zuweisung** führt dazu, daß die Maschine jede Möglichkeit der Parallelität erkennen und mit dem Ziel der optimalen Ausnutzung der vorhandenen Betriebsmittel nutzen kann. Diese faszinierende potentielle Eigenschaft hat im Laufe der Zeit immer wieder zu Forschungsprojekten geführt, wie man das Prinzip **kosteneffektiv** realisieren kann. Allerdings gibt es bis heute keine Produkte dieser Art auf dem Markt.

Datenflußrechner - Grundlagen

- Im Datenflußrechner wird von der **sequentiellen Programm-ausführung** völlig abgegangen. Die Operationen des Programms werden nach dem Datenflußprinzip grundsätzlich in dem Augenblick ausgeführt, in dem

∀ • sie mit allen ihren Operandenwerten versehen sind und

∀ • ein Prozessor zu ihrer Ausführung verfügbar ist.

- Einen **Befehlszähler** wie in der von-Neumann-Maschine gibt es damit nicht.

Datenflußrechner - Grundlagen

- **Idealfall: es** sind immer so viele ausführbare Aktionen vorhanden, wie Prozessoren zur Verfügung stehen (**optimale Auslastung** der Prozes-soren).
- Das Prinzip der einmaligen Zuweisungen reduziert die Datenabhängig-keiten auf die ***Datenflußabhängigkeit*** , weil es die im von-Neumann-Rechner übliche Mehrfachbelegung von Speicherplätzen nicht gibt.
- Die **Synchronisation** zwischen den datenabhängigen Operationen wird von der Maschine selbst, d.h. zur Laufzeit auf Grundlage der Zündregel-Semantik vorgenommen. Deshalb sagt man auch, daß die Ausführung eines Datenflußprogramms ***datengetrieben*** ist.
- Das ***Datenflußprinzip*** ist das ***Operationsprinzip*** *einer Rechner-architektur*, bei der die Steuerung des Ablaufs eines Algorithmus nicht explizit durch einen Kontrollfluß bestimmt wird, sondern implizit aus dem Datenfluß abgeleitet wird, der mit der Berechnung verbunden ist.

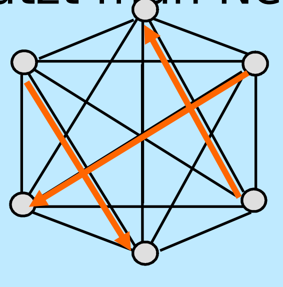
Datenflußrechner - Nachteile

- Das **Prinzip der einmaligen Zuweisung** bringt es mit sich, daß bei mehrfacher Verwendung eines Datenwertes entsprechend ***viele Kopien*** erzeugt werden. Dem Vorteil, daß damit der Zugriff eines Akteurs auf einen Datenwert völlig vom Zugriff anderer Akteure auf denselben Datenwert entkoppelt ist, ist andererseits mit dem Nachteil verbunden, daß der **Speicherbedarf entsprechend vervielfacht** wird. Daher gehen praktisch alle bisher realisierten Datenflußmaschinen hier vom reinen Datenflußprinzip ab.
- Es gibt keine Konstrukte für **Verzweigungen** und **Iterationen.** Reine Datenflußrechner eignen sich nur für spezielle Problemklassen, bei denen ein Satz von Operationen repetierend auf eine Folge von Werten angewandt wird.

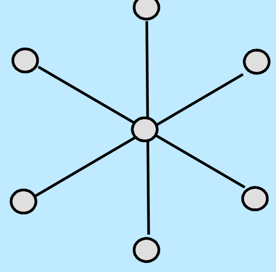
Netze zur Prozessorkopplung und Prozessor-Speicherkopplung

Bei nachrichtenorientierten Systemen müssen die Prozessoren miteinander ver-bunden werden. Bei Systemen mit gemeinsamem Speicher müssen die Prozesso-ren mit den Speichermodulen verbunden werden. Dafür benutzt man Netze.

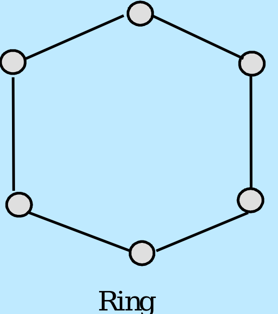
Grösster Hard-wareaufwand Höchster Paralleli-tätsgrad



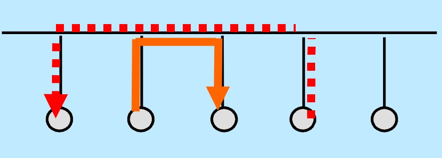
vollständige Vernetzung



Sternverbindung



Ring

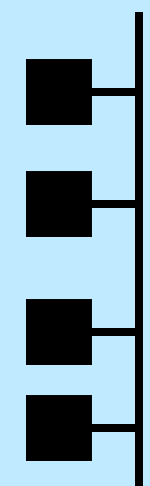


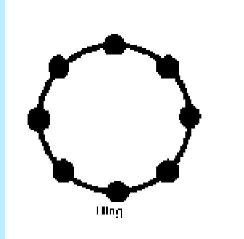
Bus

geringster Parallelittätsgrad geringster Hardwareaufwand

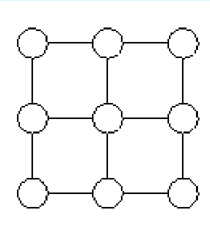
Netze zur Prozessorkopplung und Prozessor-Speicherkopplung

- **Bus**: Der Bus ist die einfachste Möglichkeit, ein Netz zu realisieren. Nachteil: Kapazität ist auf eine kleine Anzahl von Prozessoren beschränkt und es ist nur eine Übertragung pro Zeiteinheit möglich.
- **Ring** (uni- oder bidirektional): Der Ring ist eine weitere einfache Möglichkeit, ein Netz zu realisieren. Es treten aber dieselben Kapazitätsprobleme wie beim Bus auf.

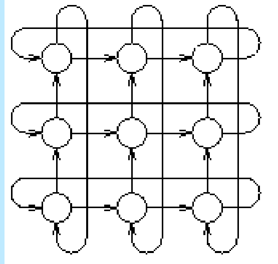




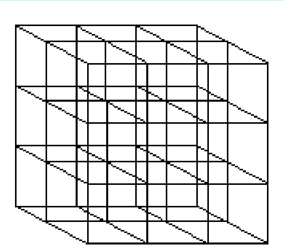
Netze zur Prozessorkopplung und Prozessor-Speicherkopplung



- **2D-Gitter und -Torus**: Eine sehr verbreitete Topologie ist die des zweidimensionalen Gitters.



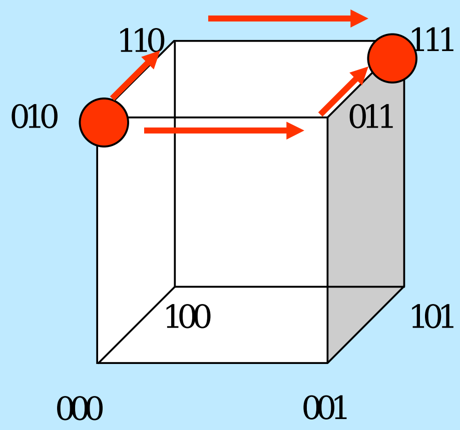
- Der **Torus** stellt eine Erweiterung des Gitters dar, bei welchem zusätzliche Leitungen in beiden Dimensionen hinzugefügt werden, indem man die Seiten des Gitters koppelt.



- **3D-Gitter und -Torus**: Erweitert man das 2D-Gitter um eine Dimension, so erhält man das 3D-Gitter.

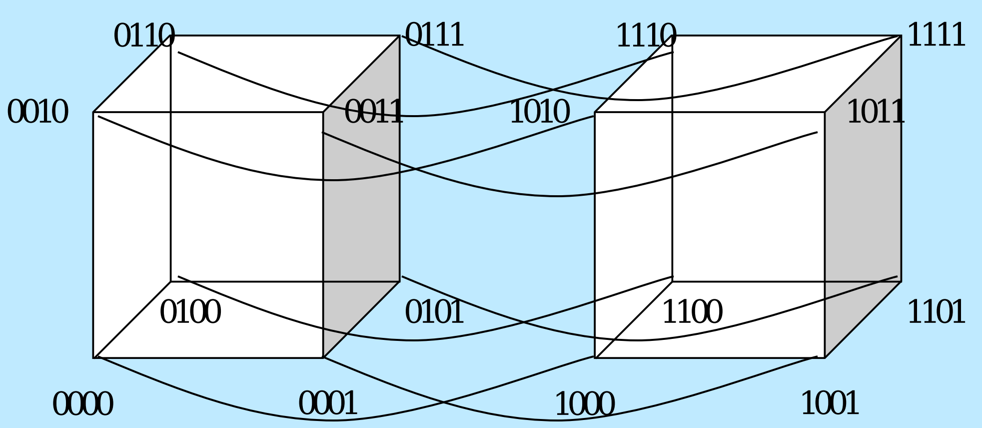
Netze zur Prozessorkopplung und Prozessor-Speicherkopplung

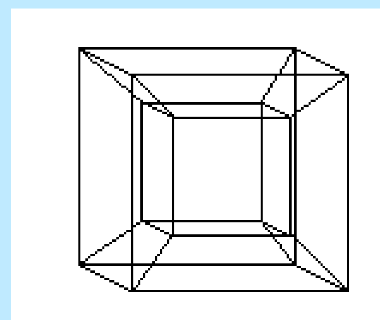
- **Würfel**. Zur Wegberechnung müssen unter-schiedliche Bits in Start- und Zieladresse untersucht werden. Dazu dient die **XOR**-Ver-knüpfung. **Beispiel:** Gegeben seien die Kno-ten A = 010 und B = 111. Eine Wegbeschrei-bung erhält man mit **W= A XOR B = 101**. W gibt nun an, welche Bits geändert werden müssen, um vom Anfangsknoten zum Endknoten zu gelangen. Hier ergeben sich die beiden Möglichkeiten 010, 011,111 und 010,110,111. Es ergeben sich bei k gesetzten Bits in W k! verschiedene Wegmöglichkeiten, um vom Startknoten A zum Endknoten B zu gelangen.



Netze zur Prozessorkopplung und Prozessor-Speicherkopplung

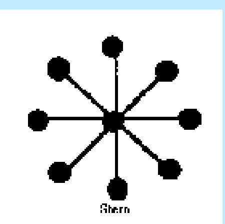
- **Hyperkubus:** Das Konstruktionsprinzip eines n-dim. Hyperkubus zu einem (n+1)-dim. Hyper-kubus lautet, dass man zwei n-dim. Hyperkuben nebeneinanderlegt und korrespondierende Knoten verbindet.



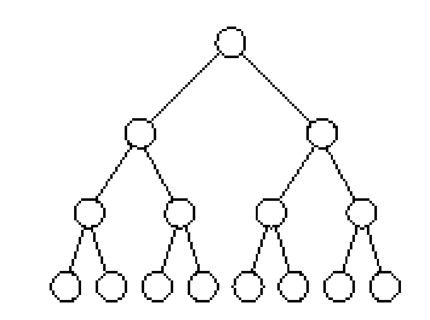


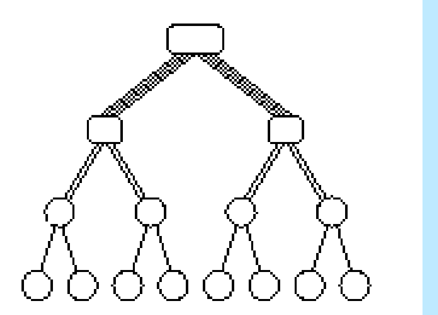
Netze zur Prozessorkopplung und Prozessor-Speicherkopplung

- **Stern**
  Der Stern ist eine weiteres einfaches Ver-knüpfungsnetz. Der zentrale Knoten bildet einen Engpass.



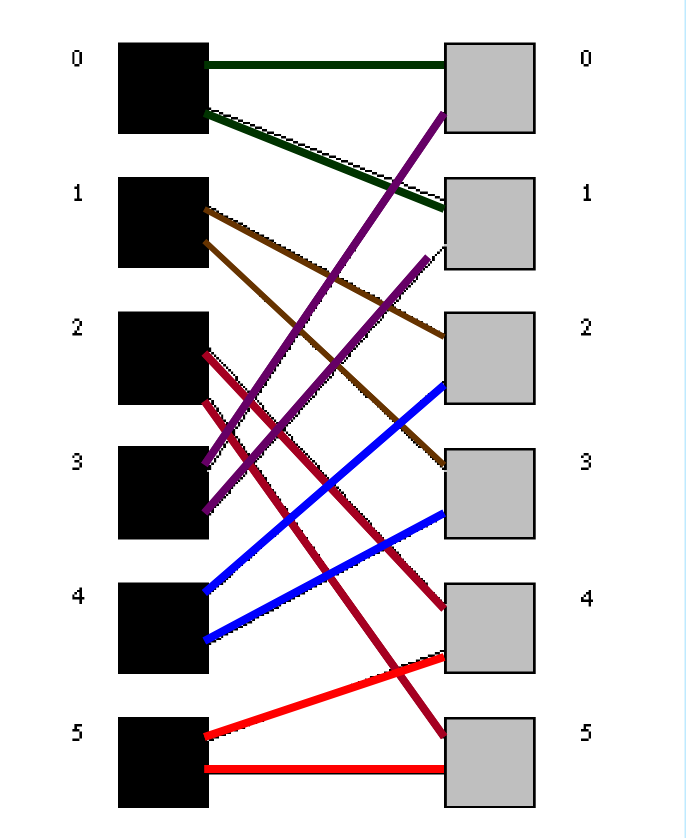
- **Baum:** Der Baum hat einen Engpass bei seinem Wurzelknoten, wenn Daten aus einer Hälfte des Baumes in die andere Hälfte transportiert werden sollen. Um diesem Engpass entgegenzuwirken, fügt man zusätzliche Pfade dem Baum hinzu, je weiter man sich der Wurze nähert. Damit erhält man einen **Fat Tree.**





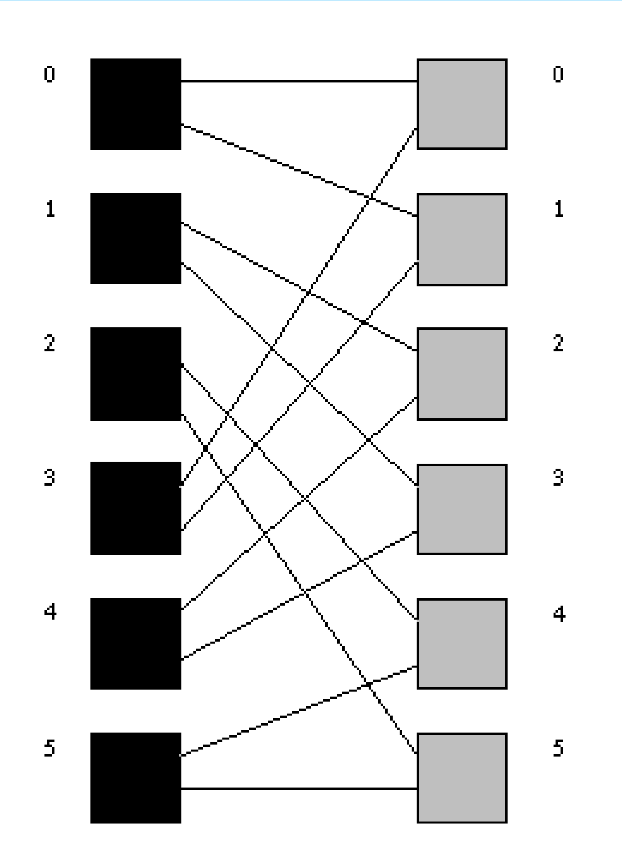
Netze zur Prozessorkopplung und Prozessor-Speicherkopplung

- **Mischpermutation** (Perfect Shuffle): B
  der Mischpermutation ist jeder Knoten p
  mit den Knoten (2p) mod N und
  (2p+1)mod N verbunden.Im Beispiel:
  2*0 mod 6 = 0 mod 6 = 0
  2*0+1 mod 6 = 1 mod 6 = 1
  2*1 mod 6 = 2 mod 6 = 2
  2*1+1 mod 6 = 3 mod 6 = 3
  2*2 mod 6 = 4 mod 6 = 4
  2*2+1 mod 6 = 5 mod 6 = 5
  2*3 mod 6 = 6 mod 6 = 0
  2*3+1 mod 6 = 7 mod 6 = 1
  2*4 mod 6 = 8 mod 6 = 2
  2*4+1 mod 6 = 9 mod 6 = 3
  2*5 mod 6 =10 mod 6 = 4
  2*5+1 mod 6 =11 mod 6 = 5



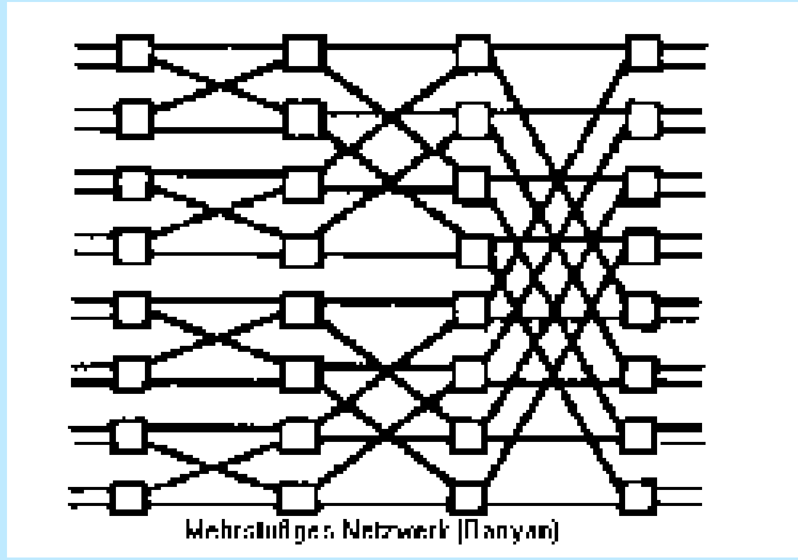
Netze zur Prozessorkopplung und Prozessor-Speicherkopplung

- **Mischpermutation** : Bei mehrmaligem durchlaufen des Netzwerkes lassen sich alle Permutationen (Verbindungen) herstellen. Obere Schranke: $3*(\log_2 N)-1$
  (N =16: 3*4 -1=11
  N = 8 : 3*3 -1= 8
  N= 6 : 3*2.5-1= 7
  N = 4 : 3*2 -1= 5)
- Als Verbindungsnetz für Hochleistungsrechner nicht geeignet
- Statt das Netzwerk mehrmals zu durchlaufen, kann man auch entsprechend viele Netzwerke hintereinander schalten

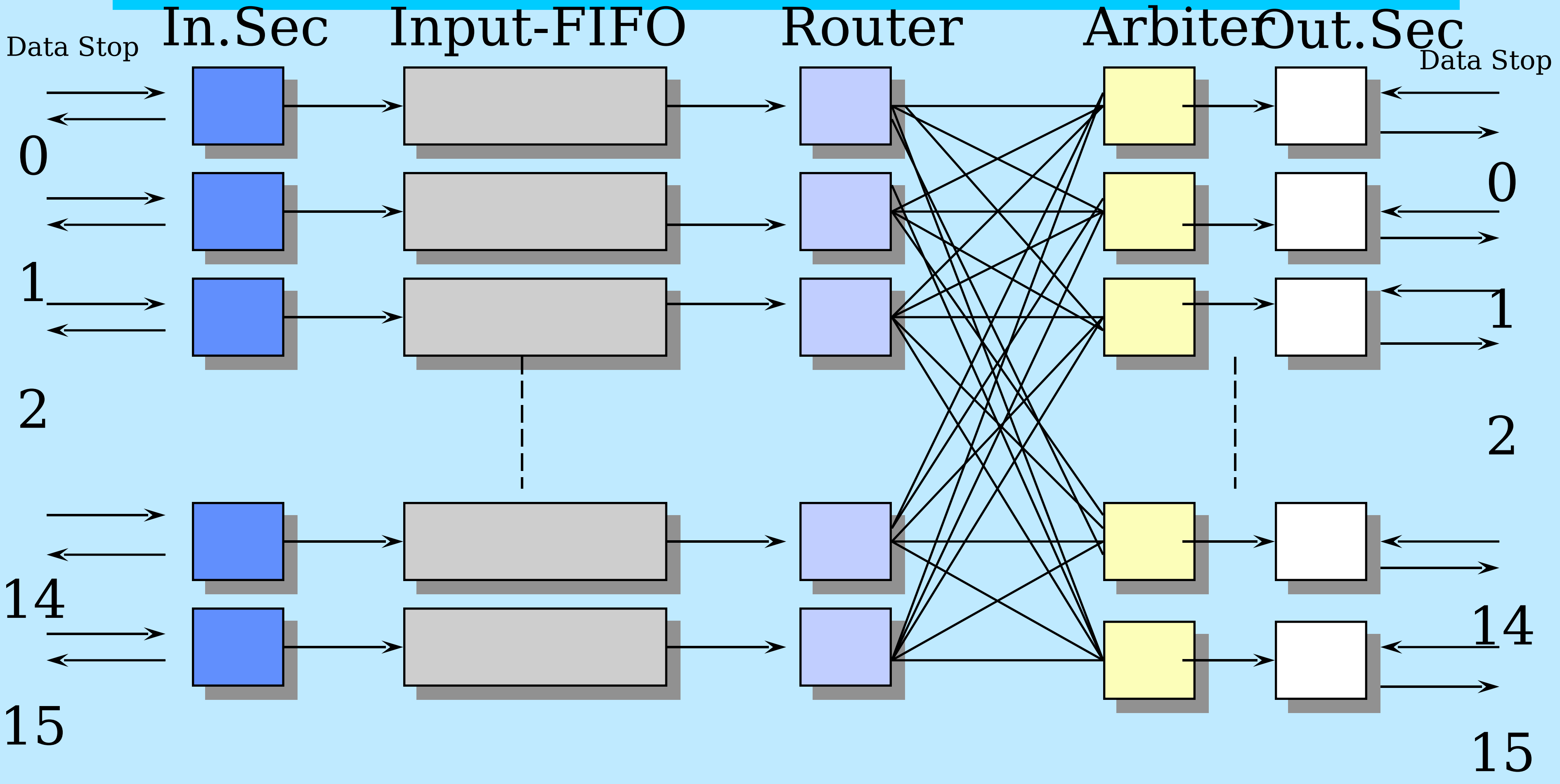


Netze zur Prozessorkopplung und Prozessor-Speicherkopplung

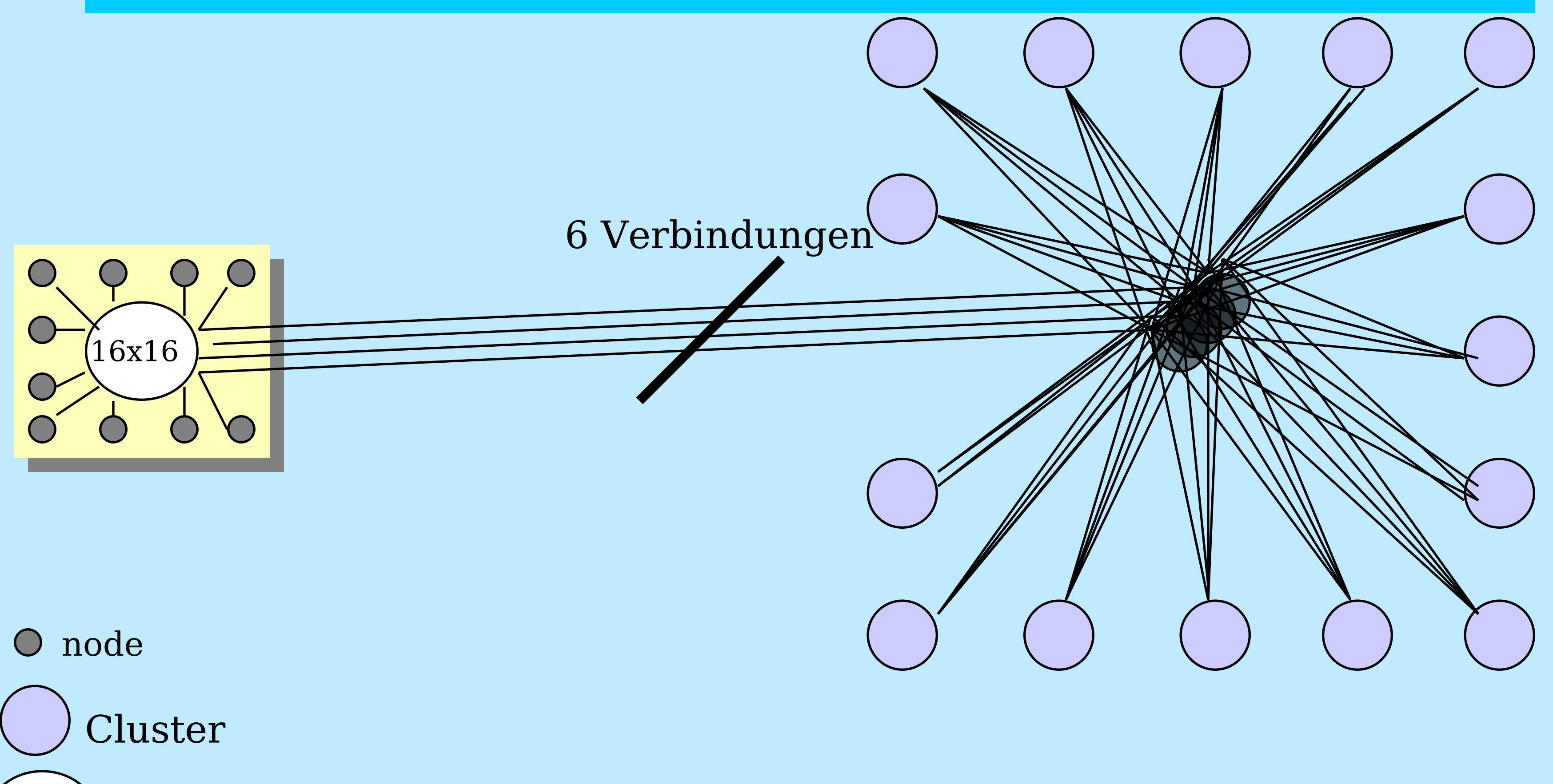
- **Banyan Netzwerk**
  Das Netzwerk ist ein mehrstufiges Verbindungsnetzt



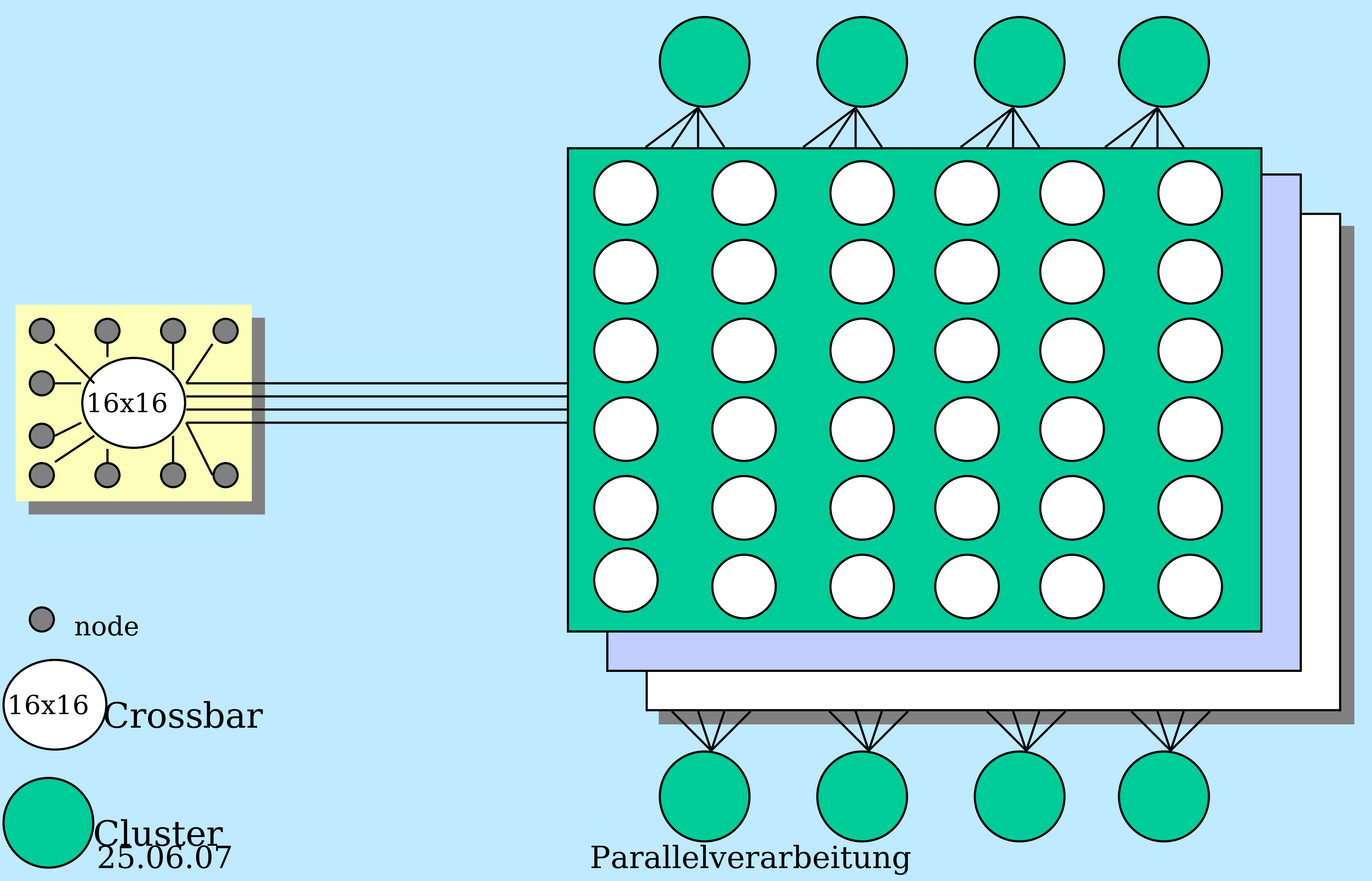
Netze zur Prozessorkopplung und Prozessor-Speicherkopplung : Der Crossbar



Dreistufige Crossbar-Verbindungstopologie für 160 Knoten



Supercluster von Crossbars



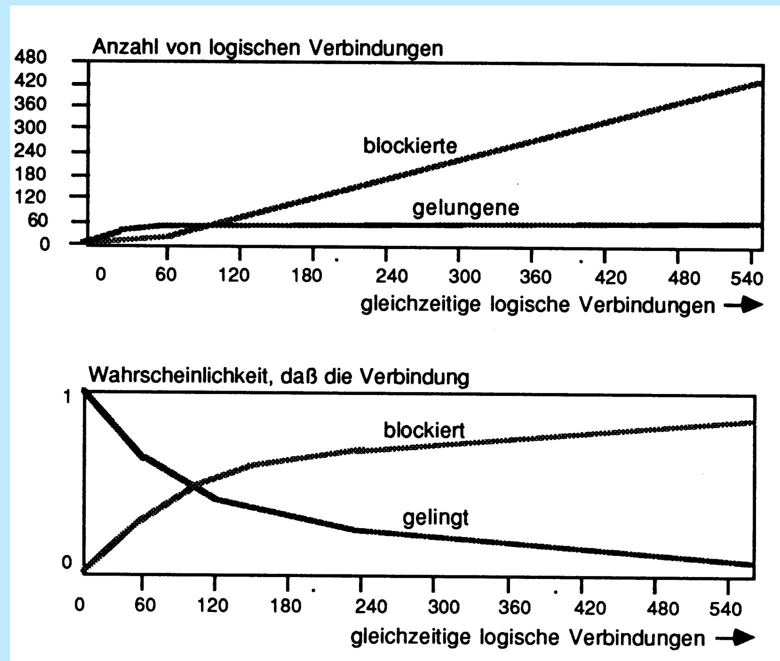
Qualitative Bewertung der Netze

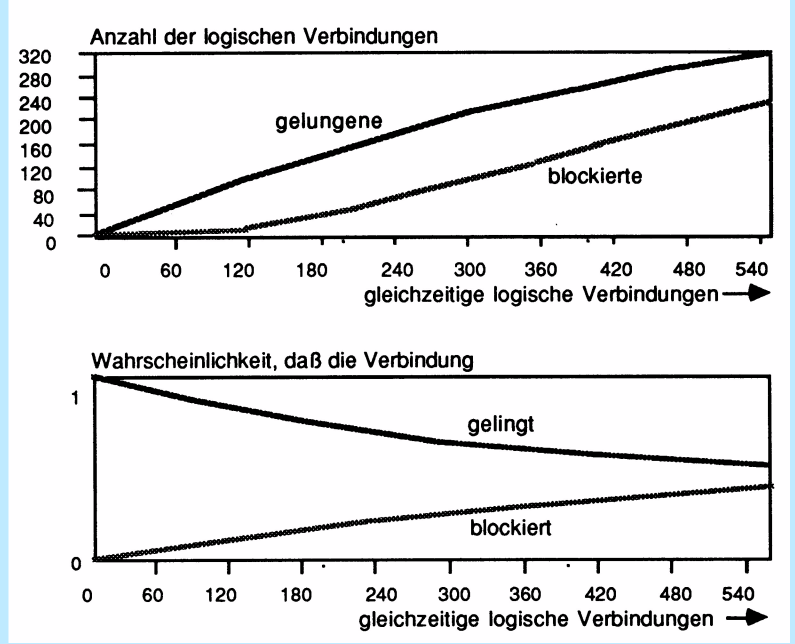
- Niedrige Kosten (beschränkter Verbindungsgrad jedes Knotens)
- Kleiner Durchmesser (maximale Pfadlänge zwischen zwei Knoten)
- Effizienter Datenaustausch aller Knoten gleichzeitig.
- Die Pfadberechnung sollte einfach sein.
- Redundanz (mehr als ein Pfad von Knoten A zum Knoten B)
- Kommunikation sollte frei von statischen und dynamischen Verklemmungen sein
- Das Verdrahtungsmuster sollte regelmäßig sein.
- Die Leitungen sollten kurz sein.

Charakteristika der Netze

- Topologie
- Dynamik
  - Statische Netze: es existieren feste Verbindungen zwischen Paaren von Netzknoten (Ring,Stern, Torus…)
  - Dynamische Netze: enthalten eine Komponente “Schaltnetz”, an die alle Knoten über Ein- und Ausgänge angeschlossen sind (Crossbar, mehrstufige Permutationsnetze)
- Blockierung
  - Ein Verbindungsnetz heist blockierungsfrei, falls jede gewünschte Verbindung zwischen zwei Modulen unabhängig von bestehenden Verbindungen hergestellt werden kann

Blockierungsverhalten der 2D-Gitterverbindung



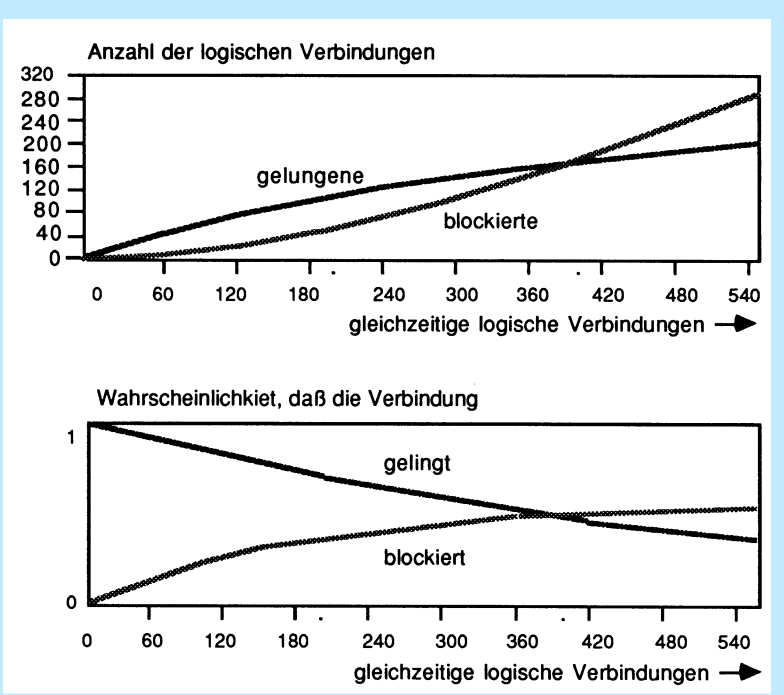


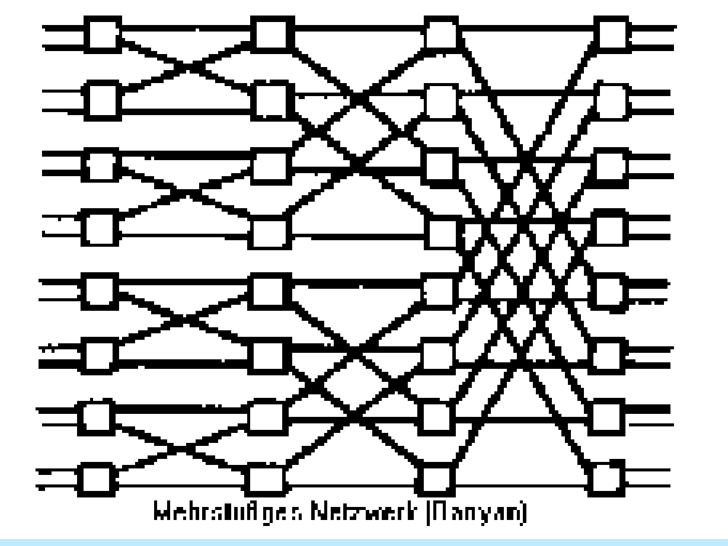
Ohne Lokalität

Mit Lokalität (häufige Kommunikation mit direkten Nachbarknoten – partielle Differenzialgleichungen)

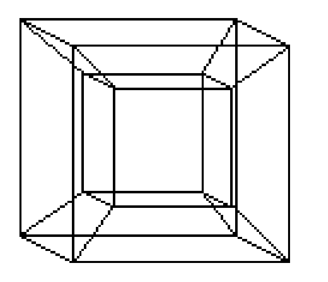
Das Blockierungsverhalten hängt von der Anwendung ab !

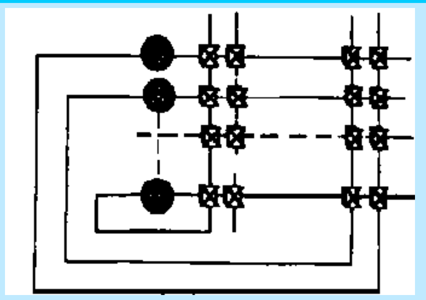
Blockierungsverhalten eines mehrstufigen Verbindungsnetzes (Banyan)

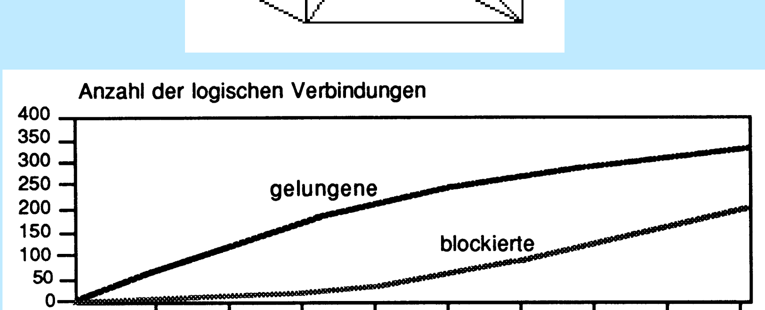


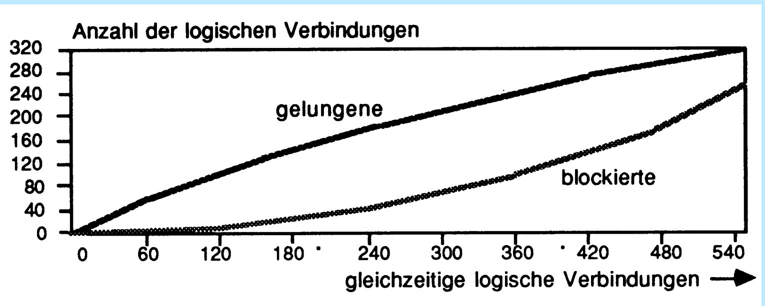


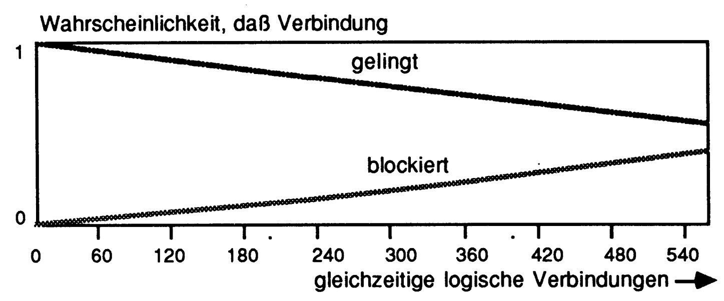
Blockierungsverhalten 10D-Hyperwürfel Crossbar

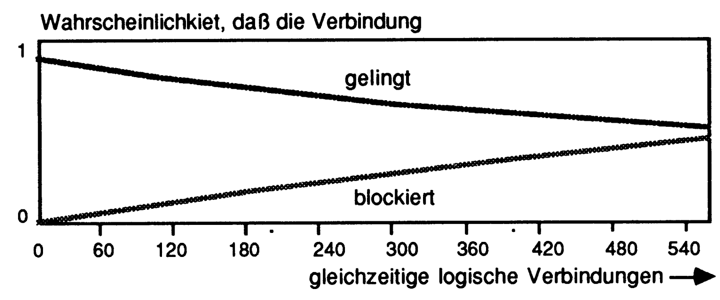


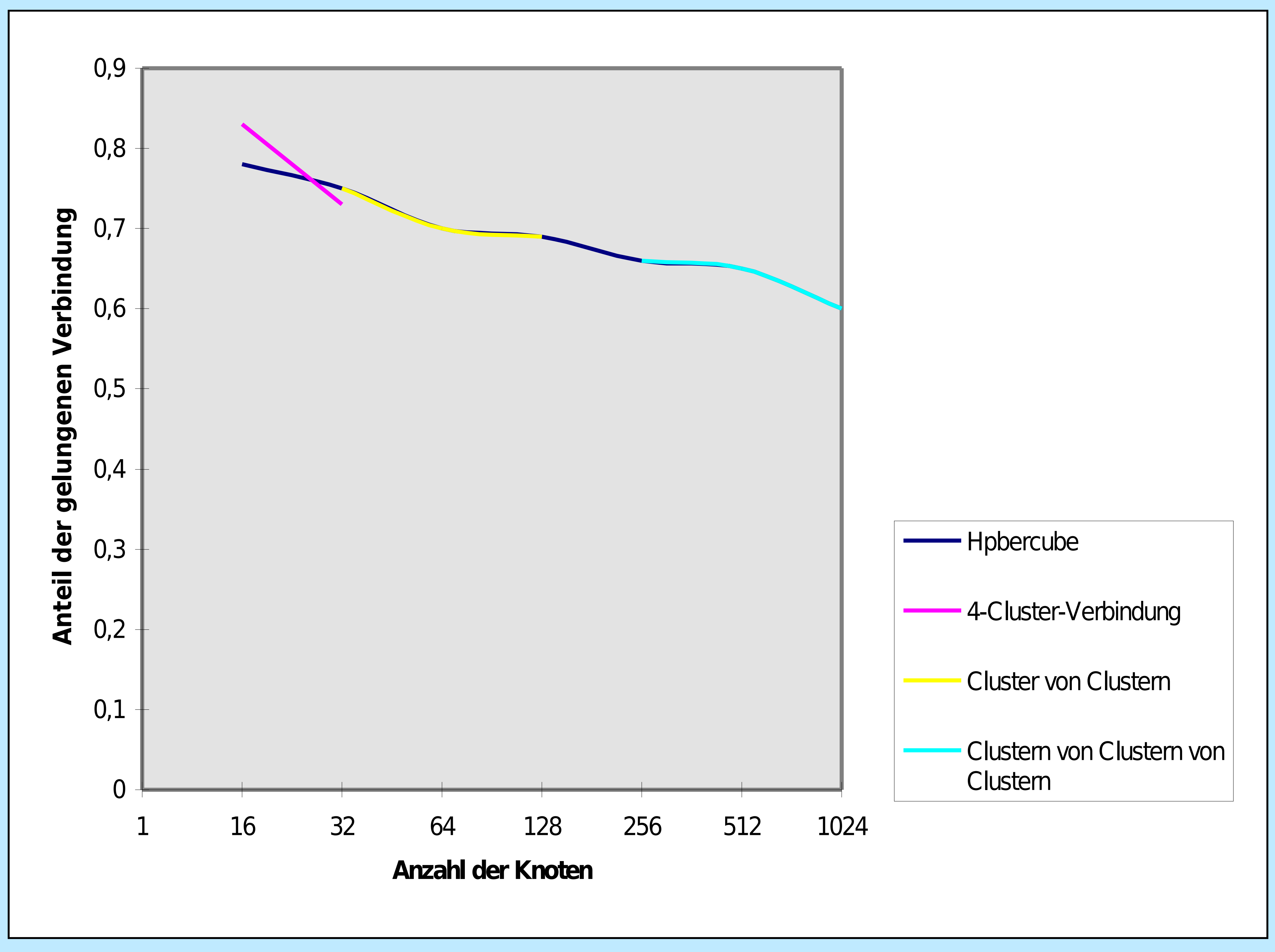










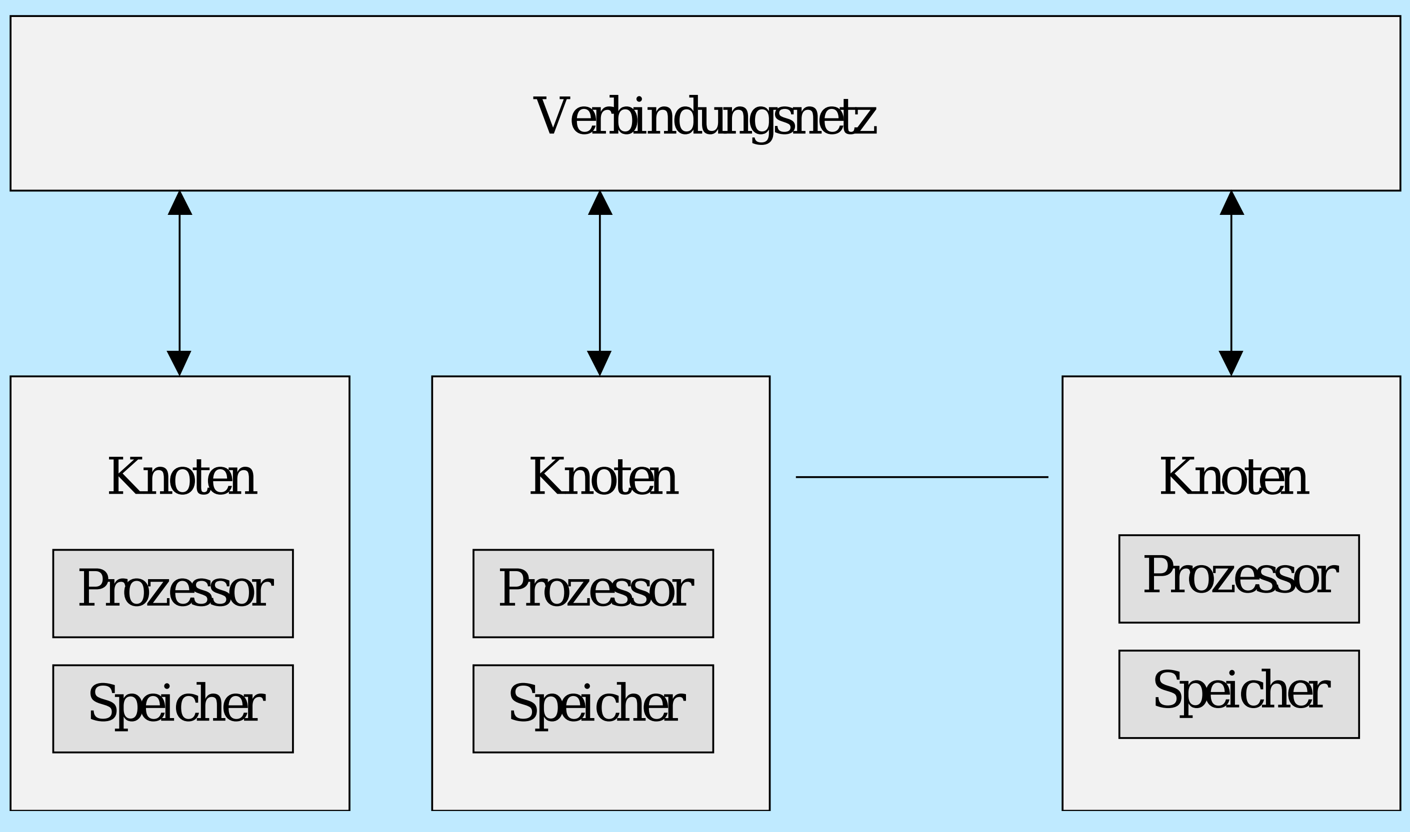


Anteil der gelungenen Verbindung
0,9
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0
1
16
32
64
128
256
512
1024
Anzahl der Knoten
Hpbercube
4-Cluster-Verbindung
Cluster von Clustern
Clustern von Clustern von Clustern

Charakteristika der Netze

- Kosten
  - Sie werden durch die Anzahl Schalter, die Anzahl der Verbindungskanäle per Schalter, die Übertragungsbreite und die Länge der Verbindungskanäle bestimmt.
- Leistung
  - Die totale Bandbreite ergibt sich als Bandbreite der Verbindungskanäle multipliziert mit deren Anzahl
- Skalierbarkeit
  - Skalierbare Multiprozessorsysteme benötigen skalierbare Netze, d.h. die Bandbreite des Netztes wächst mit der Systemgröße.
- Verbindungsart
  - Leitungs- oder Paketverbindung.
- Arbeitsweise

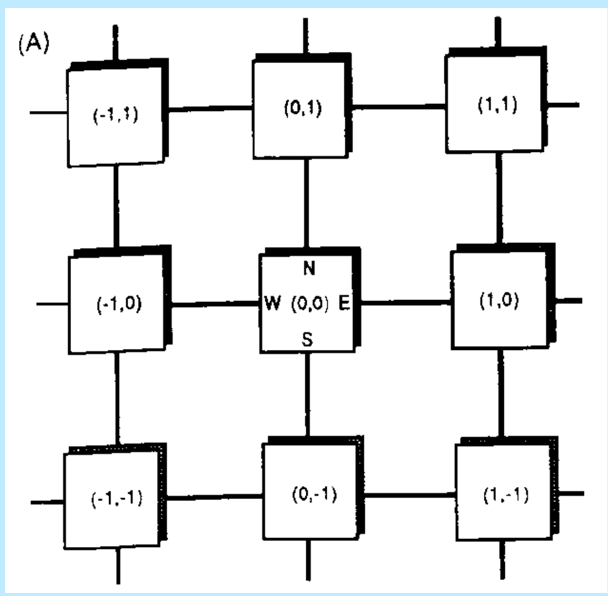
Nachrichtengekoppelte Multiprozessorsysteme



Generationen nachrichtengekoppelter Multiprozessorsysteme

Erste Generation 1983-1987

- Statische Verbindungsnetze
- Softwaregesteuerter Nachrichtenaustausch
- Store and forward Verfahren
- langsame Kommunikation - nur grobkörnige paralle Algorithmen möglich
- Beispiele: Cosmic Cube, Intel iPSC/1



Generationen nachrichtengekoppelter Multiprozessorsysteme

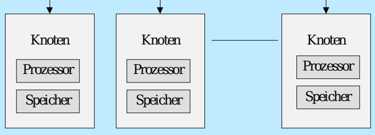
Zweite Generation 1988-1992

- hardwaregestütze Nachrichtenübermittlung - Prozessoren der Zwischenknoten werden nicht von der Übertragung belastet
- Höhere Kommunikationsgeschwindigkeit
- Worm Hole Übermittlungsverfahren
- Meist hyperkubus- oder gitterartige Verbindungsstrukturen
- Beispiele: Intel iPSC/2, iPSC/860, Intel Paragon, IBM RS/6000 SP

Generationen nachrichtengekoppelter Multiprozessorsysteme

Dritte Generation ab 1993

- Kommunikationseinrichtungen auf den Prozessorchips
- Schnelle Kommunikation - feinkörnige paralle Algorithmen möglich
- Dynamische Verbindungsstrukturen (Crossbar)
- Kommunikations- und Synchronisationsbefehle werden in die Befehlssatz der Prozessoren integeriert



- Aber: Wenig Verbreitung weil Tendenz zu Standard-µProzessoren
- Distributed Shared Memory Systeme

Alternativen zu nachrichtengekoppelten Systemen

Verteilte Systeme:

- Anzahl von kooperierenden vernetzten Rechnern
- Möglich seit Verbreitung schneller Netzwerke
- Vorteile:

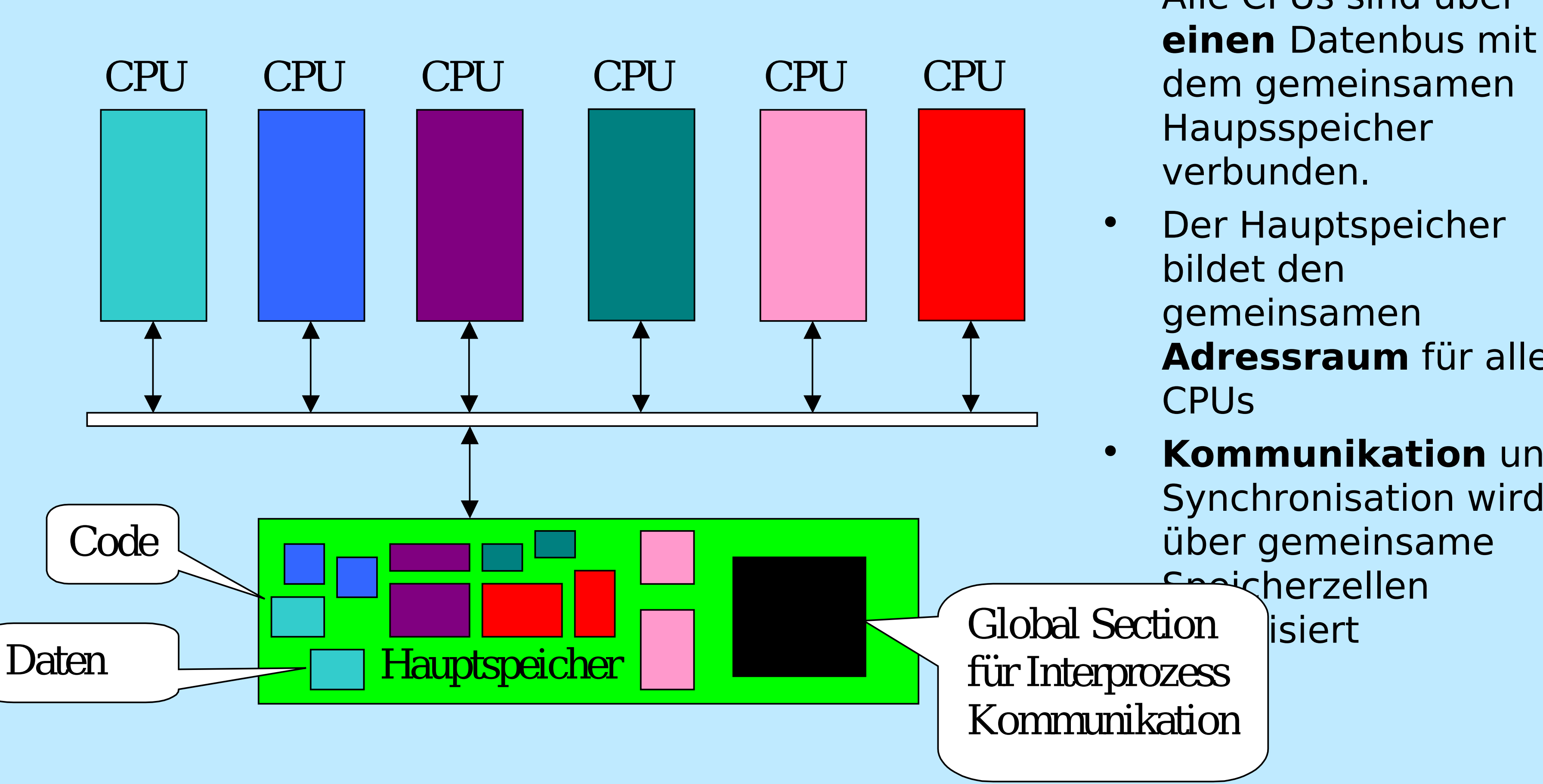
  Workstations sind oft nicht ausgelastet.

  Die neuesten Prozessoren sind meist zuerst in Workstations verfügbar, erst mit einer Zeitverzögerung in Multiprozessoren. Derzeit verdoppelt sich die Prozessorleistung alle 18 Monate -> Cluster von Workstations ist oft den Multiprozessoren leistungsmäßig überlegen.

  Ein solcher **virtueller Parallelrechner** kann leicht um weitere Rechner erweitert werden.

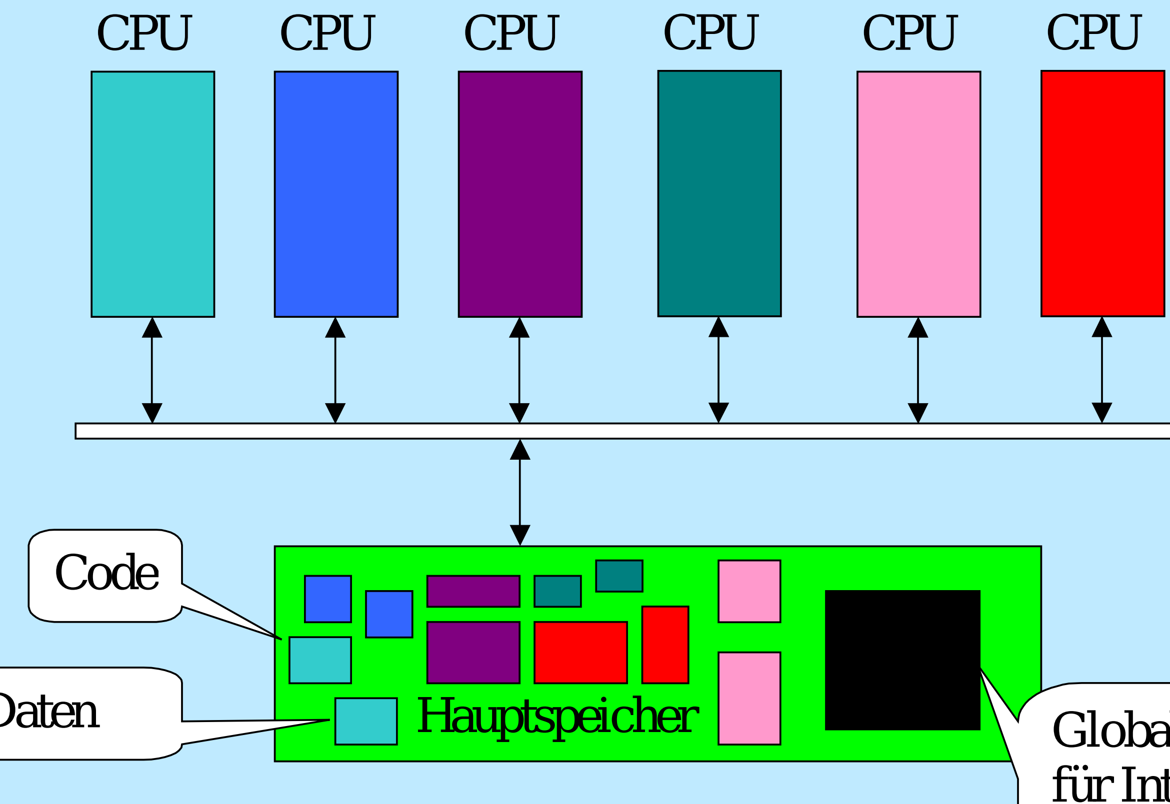
  Nachrichtengekoppelte Programmierschnittstellen PVM und MPI arbeiten sowohl bei Multiprozessoren als auch bei Workstation - Clustern

Speichergekoppelte Multiprozessorsysteme



- Alle CPUs sind über **einen** Datenbus mit dem gemeinsamen Haupsspeicher verbunden.
- Der Hauptspeicher bildet den gemeinsamen **Adressraum** für alle CPUs
- **Kommunikation** und Synchronisation wird über gemeinsame Speicherzellen ...isiert

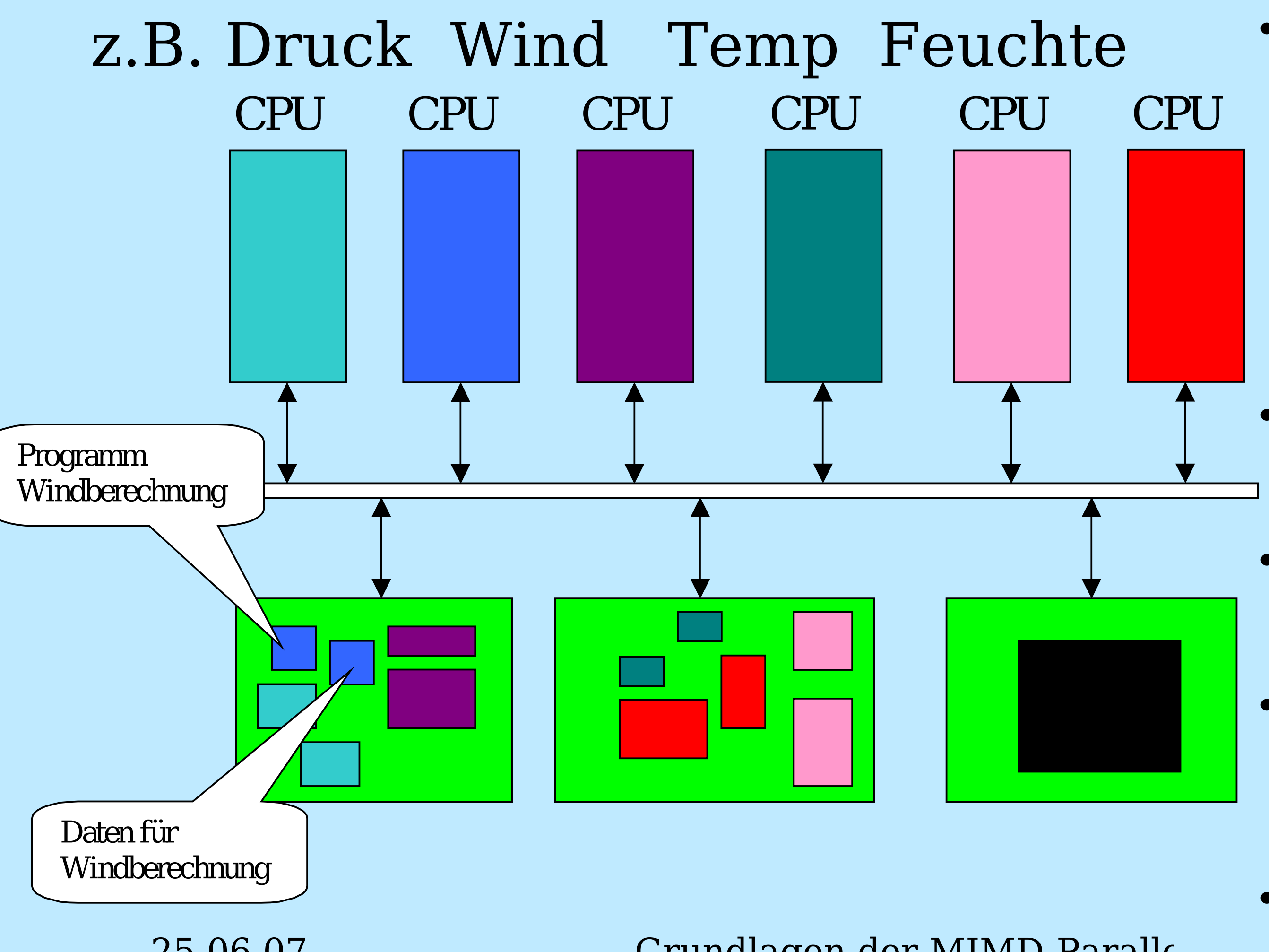
Effizienzprobleme speichergekoppelter Multiprozessorsysteme



- Programmierung mit ge-meinsamem Adressraum ist sehr effizient (einfach, schnell) ***aber:***
- Der von-*Neumann Flaschenhals* multipliziert sich mit der Anzahl angeschlossener CPUs.
- Jede CPU greift bei je-dem Befehl auf den Code-Teil und sehr häufig zusätzlich auf den Daten-Teil im HS

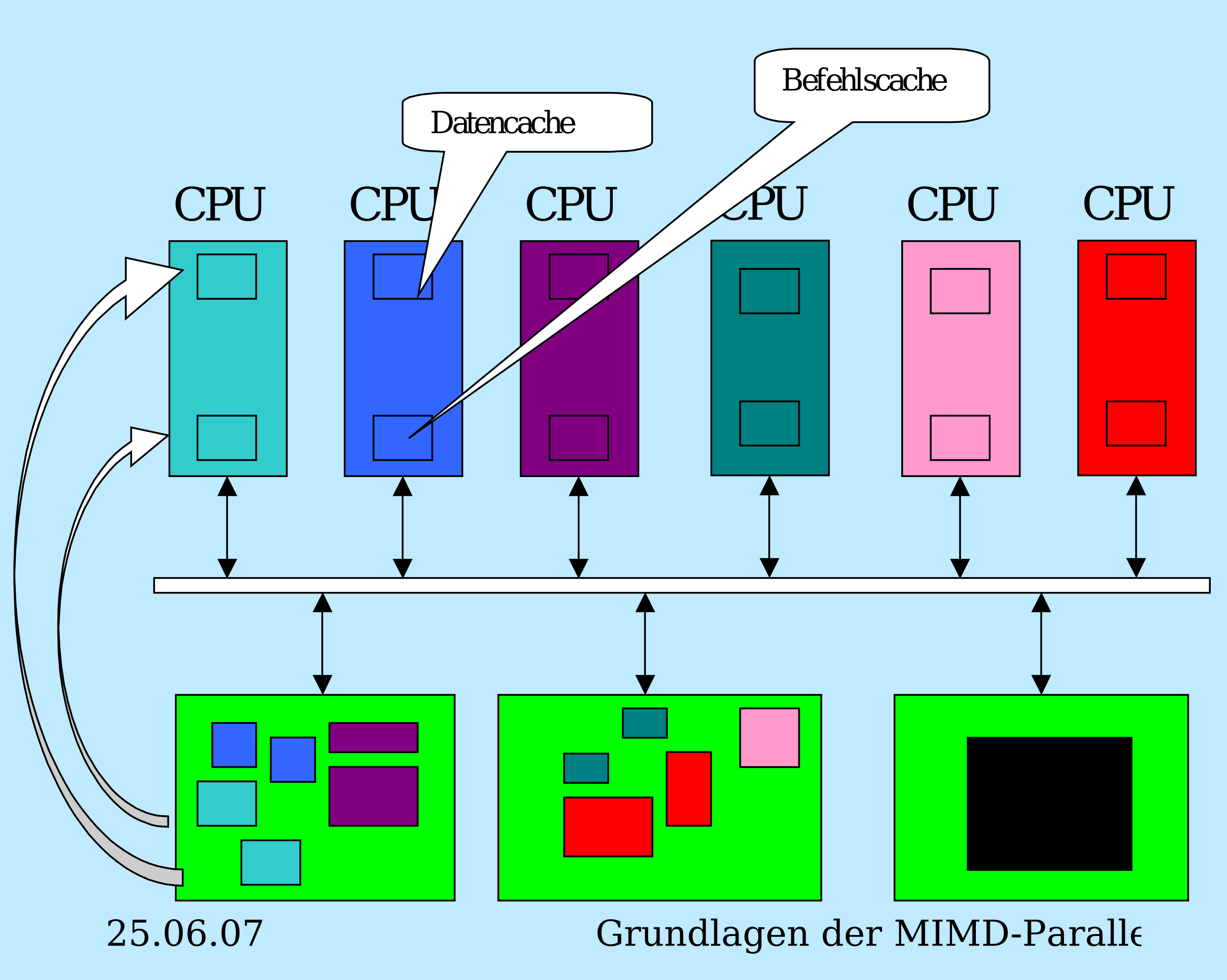
lem der cherbandbreite

Maßnahmen zur Steigerung der Effizienz



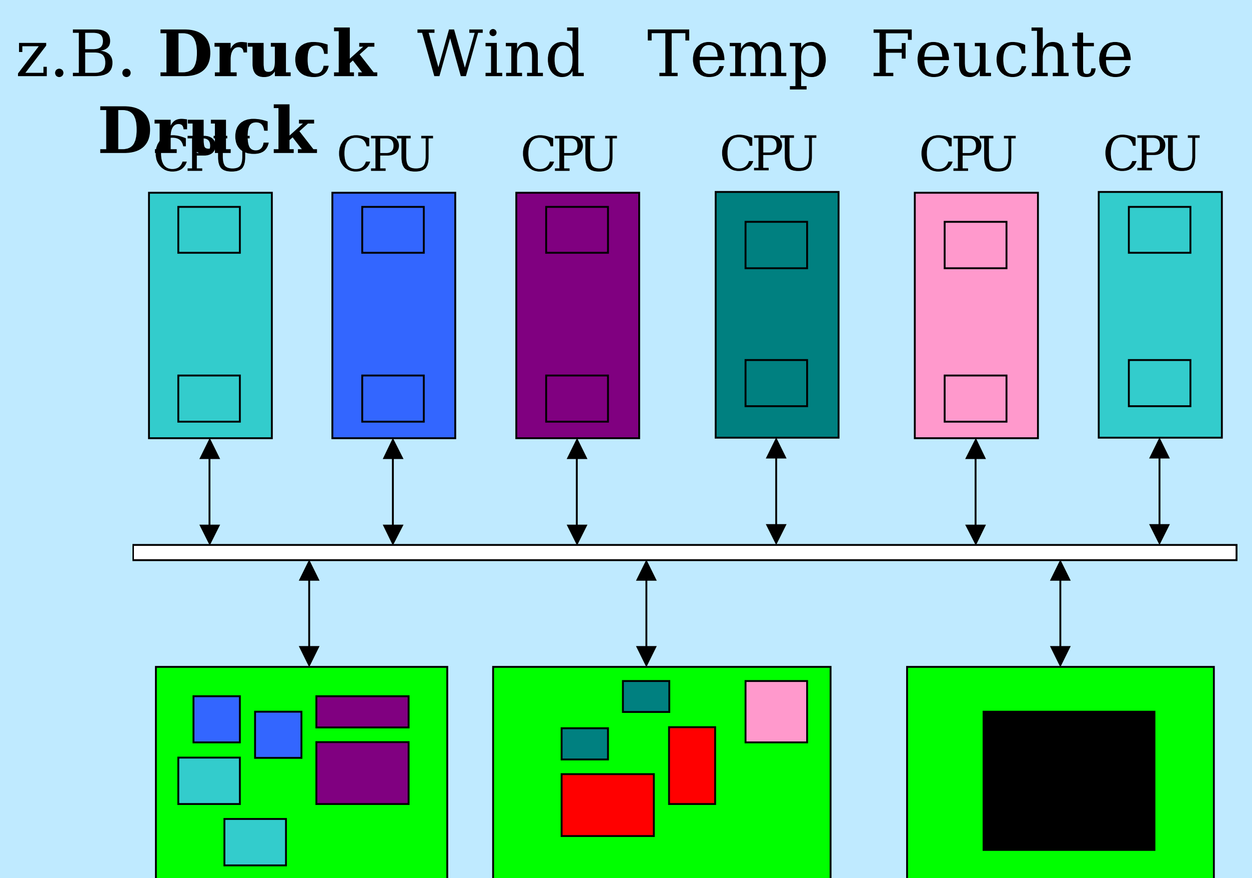
- **Speicherzugriffe minimieren**. Je kleiner die Programme, desto besser (CISC Philosophie) Jedoch: RISC Prozessoren benö-tigen bis zu 4 mal mehr Speicherzugriffe (Befehle).
- Speicher auf mehrere **Speicherbänke** verteilen.
- Speicherzugriffe überlappt ausführen (pipelined)
- **Burst Mode** (benachbarte Daten des angeforderten Datums mitliefern)
- Kombination von allem.

Maßnahmen zur Steigerung der Effizienz – lokale Caches



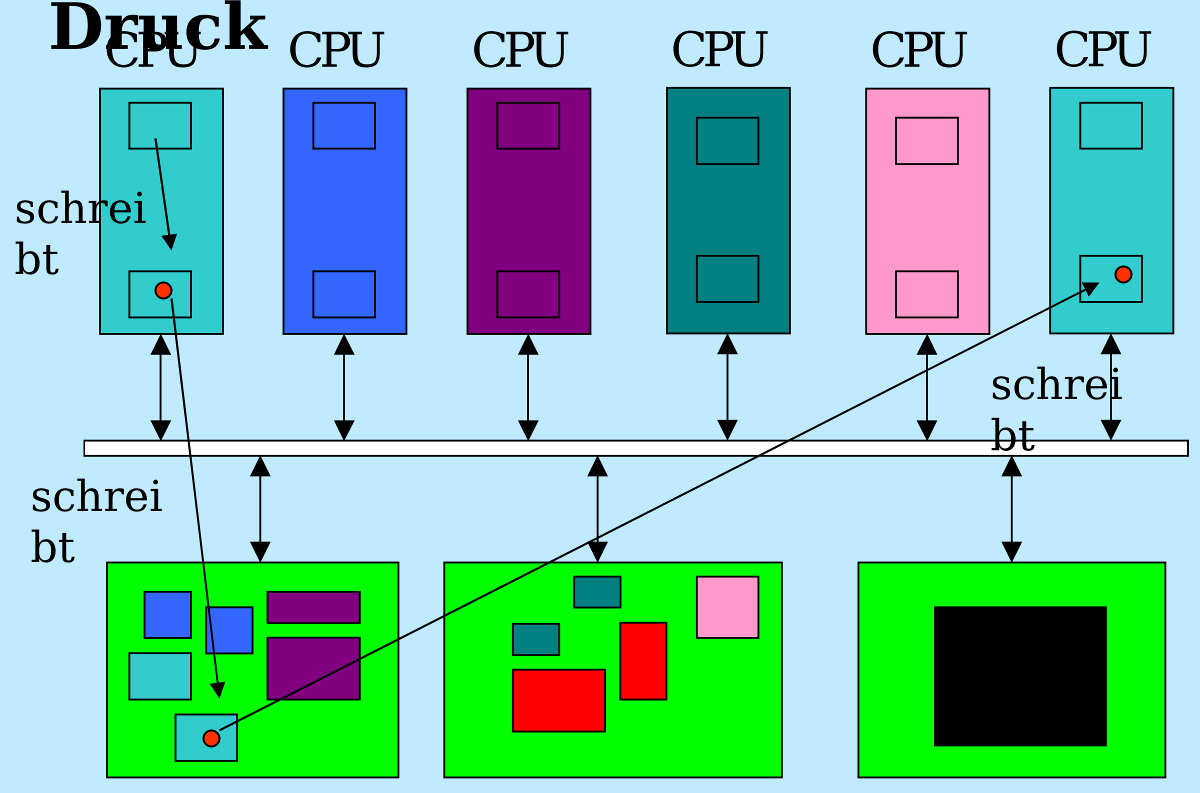
- Der gemeinsame Speicher wird ergänzt mit lokalen schnellen Caches (Annäherung an System mit verteiltem Speicher). Der gemeinsame Adressraum bleibt erhalten. Lokale Speicher bilden keine Speichererweiterng, son-dern präsentieren einen Ausschnitt des Haupt-speichers. Je weniger Überschneidungen im Datenbereich der einzelnen Knoten existieren, desto

Lokale Caches und Cachekonsistenz



- Arbeiten mehrere CPUs auf dem gleichen Datenbestand (im Beispiel: 2 Prozessoren berechnen jeweils Wind, Druck..), existieren mehrere Kopien eines Datums des HS in den Datencaches. Was bedeutet es, wenn ein Datum von einer CPU im Cache verändert wird?

Lokale Caches und Cachekonsistenz



- Es muss sichergestellt sein, dass jeder Knoten das zuletzt geschriebene Datum einer Speicherzelle liest. Das erfordert: Eine Verän-derung in einem Cache führt zum update aller Caches und des Haupt-speichers. Das ist strenge Speicherkonsistenz. Das bedeutet hohen technischen Aufwand, der sich nicht lohnt, wenn ein verändertes Datum nie mehr von

Cachekohärenzprotokoll

z.B. **Druck** Wind Temp Feuchte **Druck**

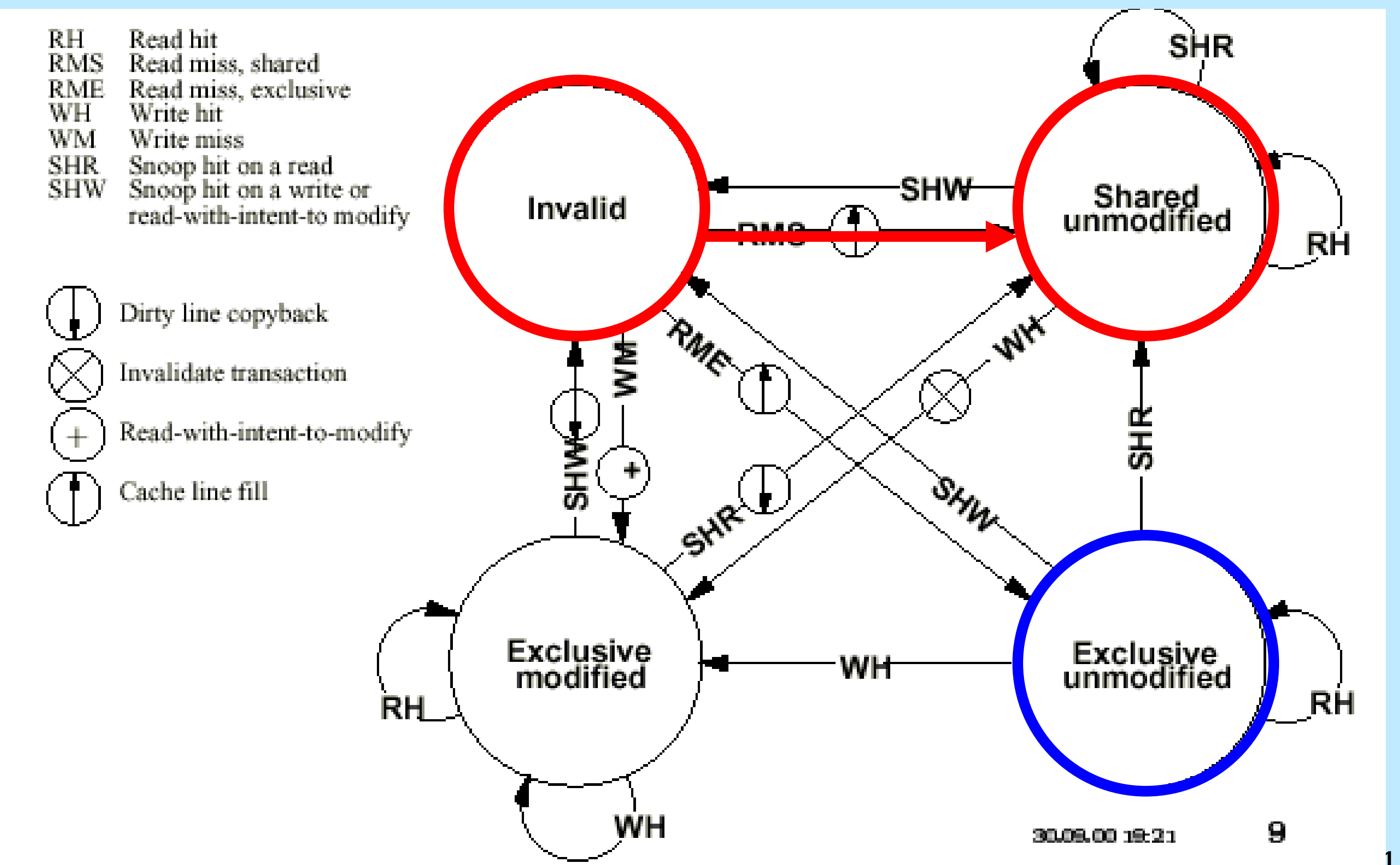
CPU CPU CPU CPU CPU CPU

schreibt

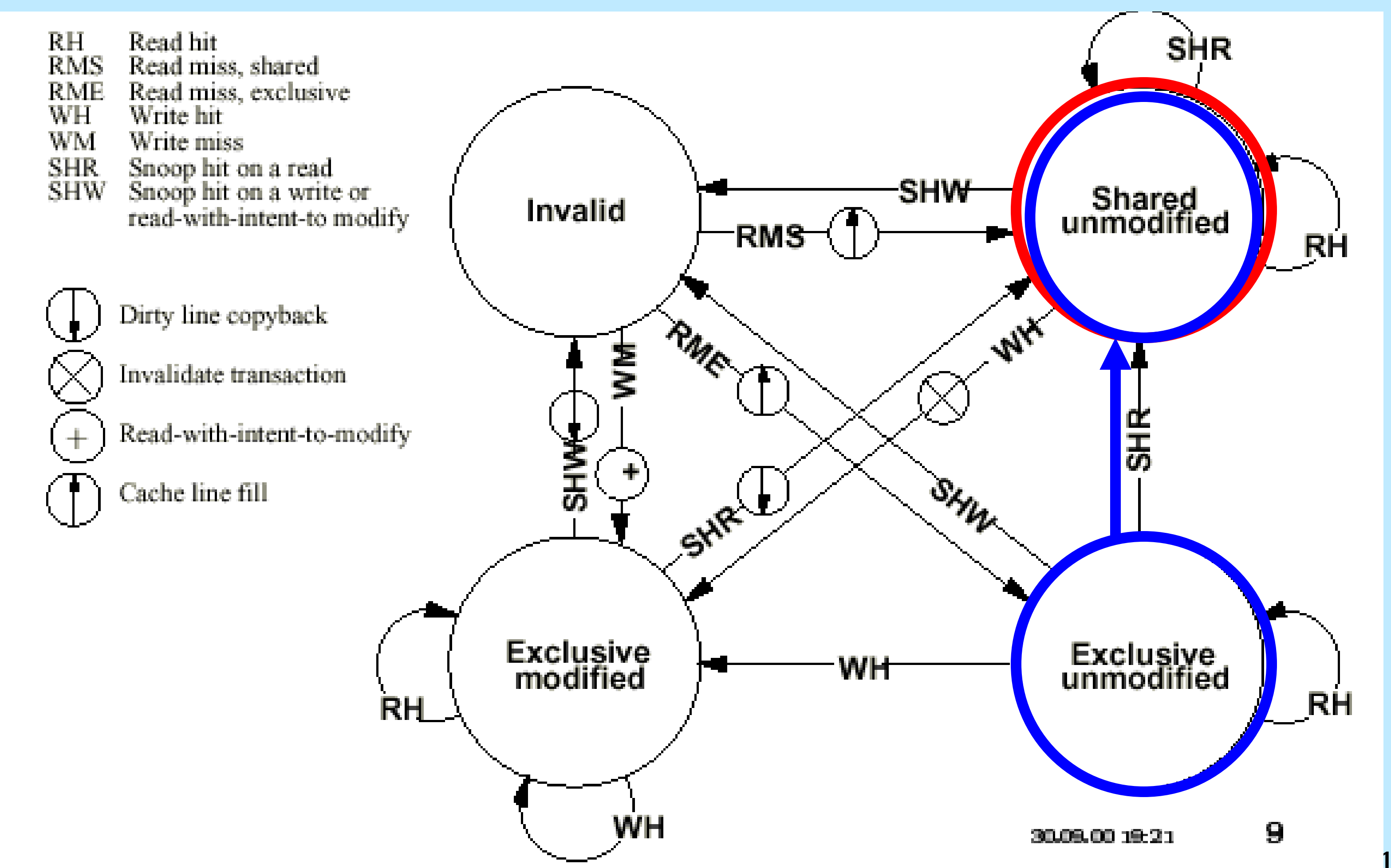
ungültig

- Kohärenz ist **eingeschränkte Konsistenz**
- Ein Kohärenzprotokoll sorgt ‘nur’ dafür, dass immer die aktuellen, nicht veraltete Daten verwendet werden.
- **Write Update Protokoll** Änderung eines Datum -> Änderung aller Kopien, wenn auch verzögert
- **Write Invalidate Protokol** Änderung eines Datum -> alle

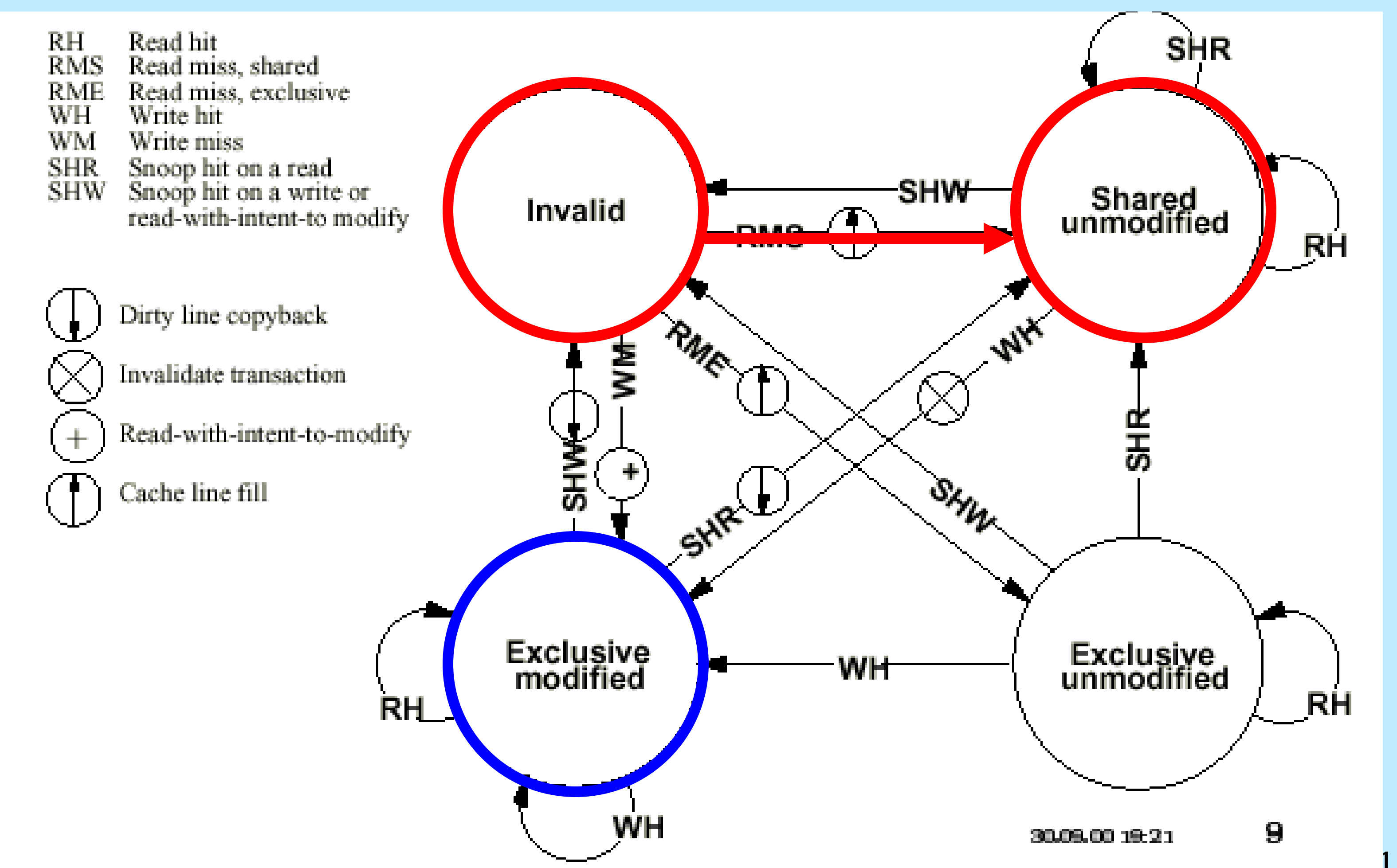
MESI - Lesen eines Datums bei ungültigem Cacheblock



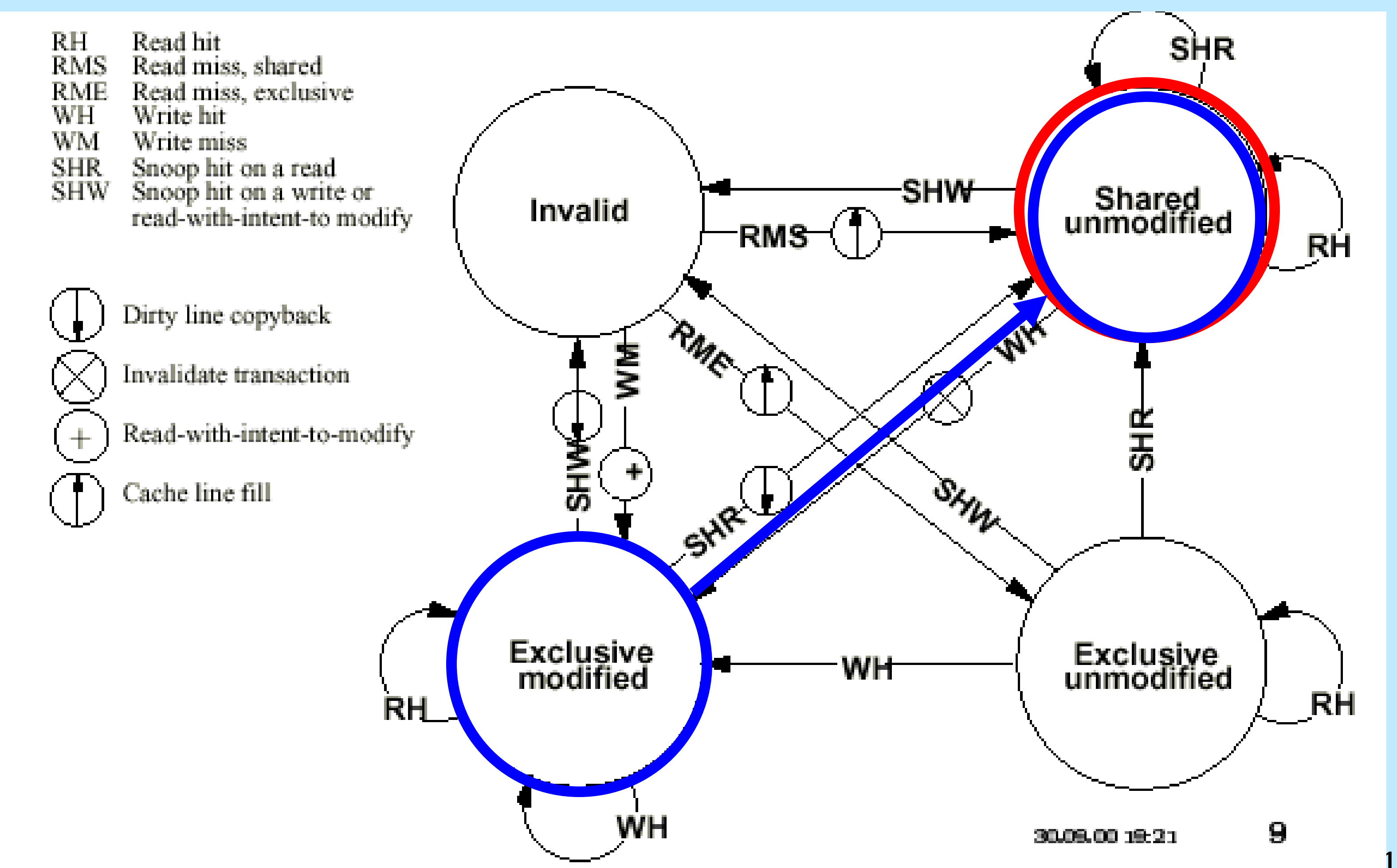
MESI - Lesen eines Datums bei ungültigem Cacheblock



MESI - Lesen eines Datums bei ungültigem Cacheblock



MESI - Lesen eines Datums bei ungültigem Cacheblock



Konzepte der parallelen Programmierung

- Die historische Entwicklung zeigte immer wieder den **Gegensatz von Effizienz und Portierbarkeit** paralleler Software.
- Früher: für jede neue Architektur wurde eine neue Sprache entwickelt. Z.B. **OCCAM** für transputerbasierte Systeme. Problem:
  - Programmierer mussten für jeden Rechner eine **neue Sprache** lernen
  - Programme waren **kaum portabel**
- Lösungsansätze:
  - **Einbindung paralleler Konstrukte** in herkömmliche Programmiersprachen. Aber: Compiler waren auf Architekturen spezialisiert, SW nicht portierbar.
  - Ab 1990 Entwicklung von standardisierten **Message Passing Libraries**, die auf verschieden Architekturen implementiert werden. Die Prozeduren werden über ein Interface in Standardprogrammiersprachen eingebunden.
  - Ab 1992: Wiederaufgreifen der Idee in die Sprache integrierten Parallelkonstrukte: High Performance Fortran

Konzepte der parallelen Programmierung

- Unabhängig vom benutzten Konzept ist die parallele Programmierung problematisch, weil:
  - **Neue Fehlerarten** entstehen (Synchronisations/Kommunikationsfehler)
  - Das **Verhalten** paralleler Programme schwerer nachvollziebar und oft **nicht reproduzierbar** ist.
  - Die **Aufteilung des Problems** und die Zuordnung der Prozesse zu den Prozessoren einen starken Einfluss auf die **Performance** des Programms haben

Hauptfragen der parallelen Programmierung

- Kann ein gegebenes Problem in Teilaufgaben zerlegt werden, die ohne Datenzugriffskonsflikte bearbeitet werden können ?
- Erkennen von Datenzugriffskonflikten: Wird in einer Gruppe von Anweisungen auf bestimmte Daten lesend und schreibend zugegriffen?
- Können die Datenzugriffskonflikte durch Änderung der Reihenfolge aufgelöst werden?
- Ist die Änderung der Reihenfolge algorithmisch zulässig?

Ansätze zur Parallelen Programmierung Funktionale Zerlegung

- Kann ein Problem in Teilaufgaben zerlegt werden, die ohne Datenzugriffskonsflikte bearbeitet werden können, bietet sich die **funktionale Zerlegung** an.
  - Bei funktionaler Zerlegung wird die Gesamtarbeit in eine Reihe von Teilaufgaben aufgeteilt und jede Teilarbeit wird von einem anderen Prozessor bearbeitet. Der am häufigsten auftretende Fall ist der einer **Pipeline** .

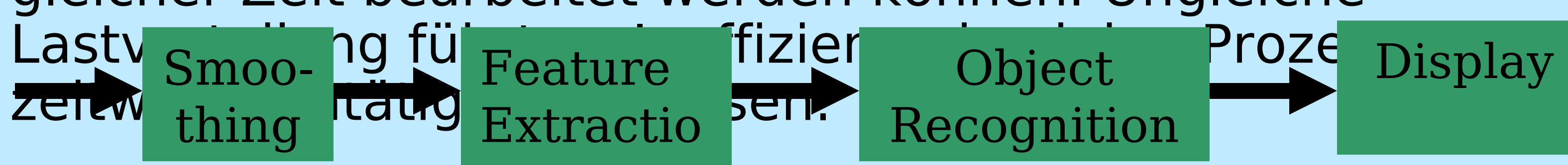
Image → Smoothing → Feature Extraction → Object Recognition → Display

Image | Geglättetes Bild | Feature Beschreibungen | Objektbeschrei-bungen

- Die Bilddaten werden von Prozessor zu Prozessor weitergegeben

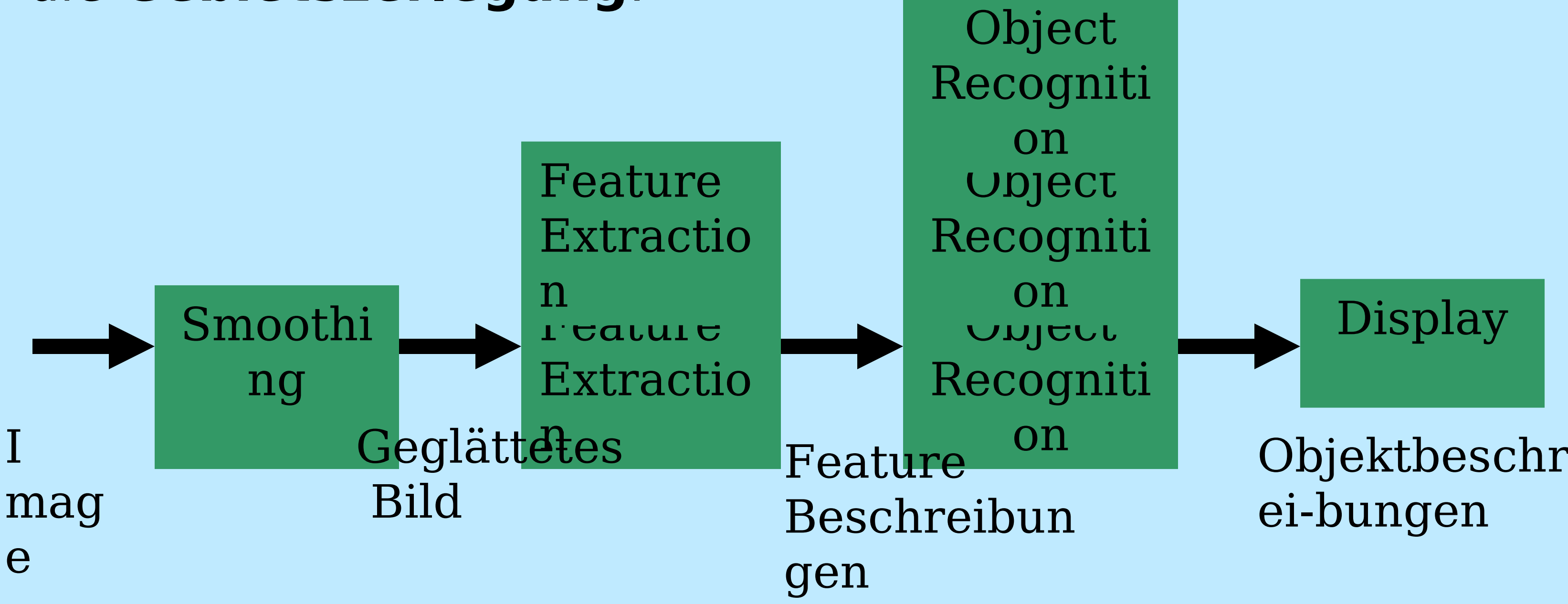
Ansätze zur Parallelen Programmierung Probleme der Funktionale Zerlegung

- **Startup/Shutdown**
  Leerlauf von Prozessoren am Anfang (Füllen) und Ende der Bearbeitung
- **Skalierbarkeit**
  Mit der Anzahl Teilaufgaben ist auch die Anzahl verwendbarer Prozessoren festgelegt. Die Skalierbarkeit ist hierdurch begrenzt.
- **Lastverteilung**
  Bei der Implementierung der Pipeline muss darauf geachtet werden, dass die Teilaufgaben in nahezu gleicher Zeit bearbeitet werden können. Ungleiche

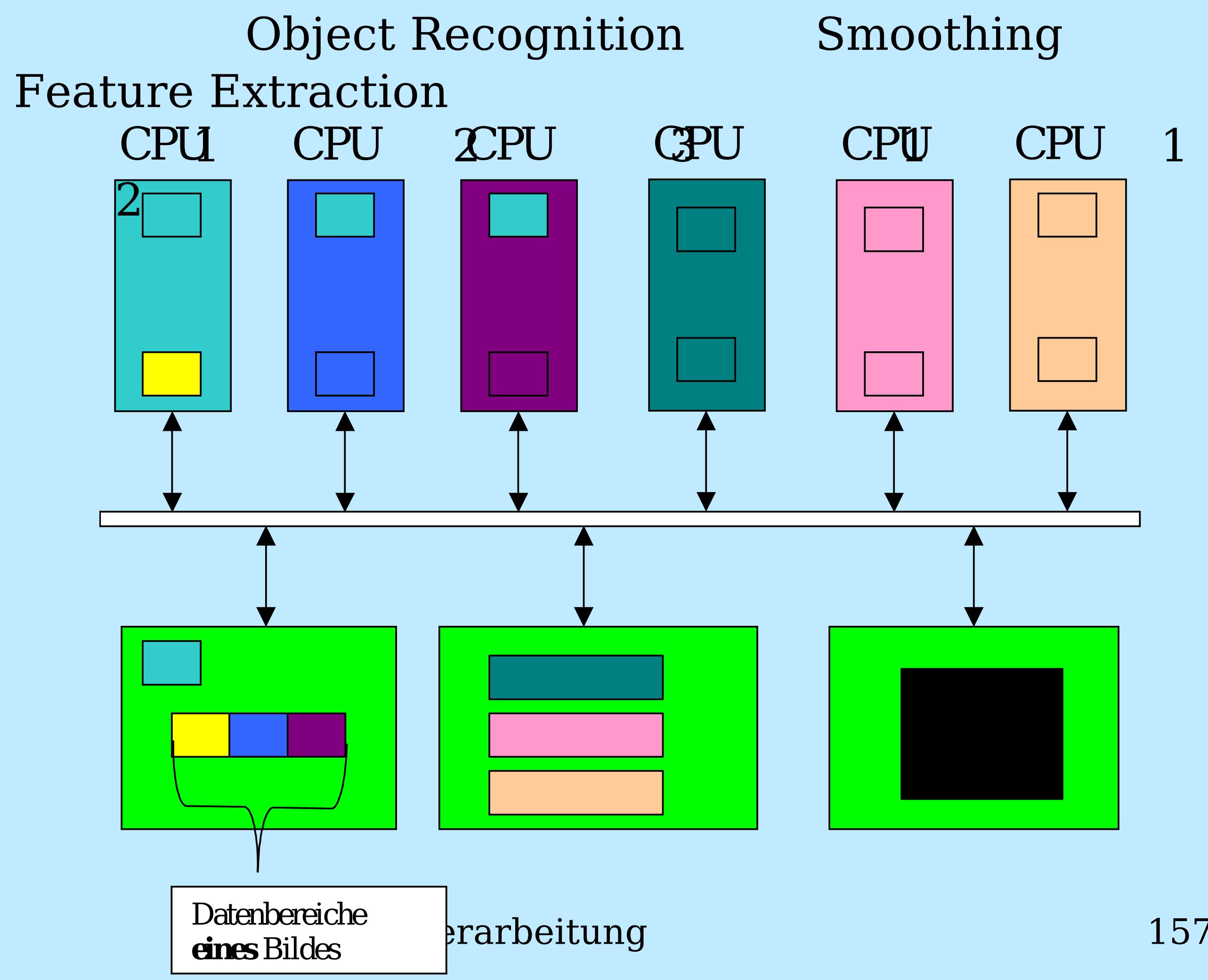


Ansätze zur Parallelen Programmierung Probleme der Funktionale Zerlegung

- Lassen sich die längeren Pipelinestufen weiter parallelisieren, lässt sich die Lastverteilung verbessern und die Skalierbarkeit erhöhen. Eine Methode hierfür ist die **Gebietszerlegung**:



Ansätze zur Parallelen Programmierung Gebietszerlegung



Nachrichtenorientierte Programmierung am Beispiel Intels Paragon XP/S

- Weiterentwicklung des Intel iPSC/860 innerhalb des Touchstone Rechnerarchitektur-Projektes des amerikanischen Verteidigungsministeriums
- 1992 mit 2048 Prozessoren installiert
- Zweidimensionale Gitterverbindung mit 1GBit pro Sekunde
- Spitzenleistung: 300 GFLOPS

Nachrichtenorientierte Programmierung am Beispiel Intels Paragon XP/S

Das NX Betriebssystem

- verteiltes Betriebssystem
- BS-Kern auf jedem Prozessorknoten
- ca. 20.000 C-Codezeilen, 90 kByte Objektcode
- pro Knoten maximal 20 Prozesse mit je 1 GByte virtuellem Adreßraum
- Verwaltung der numerischen Coprozessoren
- Organisation der Nachrichtenübertragung
  - für kurze Nachrichten (bis 100 Bytes ein „One Trip Protocol“
  - für lange Nachrichten ein „Three Trip Protocol“

NX - Synchrone (blockierende) send-/receive-Befehle

csend(type, buf, len, node, pid)

`long type;`	Nachrichtennummer
`char *buf;`	Zeiger auf Nachrichtenpuffer
`long len;`	Nachrichtenlänge
`long node, pid;`	Nummer desEmpfängerknotens und des Empfängerprozesses

blockiert nur, bis der Sendepuffer wieder frei ist und nicht, bis die Nachricht empfangen wurde!!

crecv(typesel, buf, len)

`long typesel;`	Nachrichtennummer, -maske oder
	-1 für jede beliebige Nachricht
`char *buf;`	Zeiger auf Nachrichtenpuffer
`long len;`	Länge des Puffers

blockiert, bis die Nachricht vollständig im Empfangspuffer steht

NX - Asynchroner send-Befehl

long isend(type, buf, len, node, pid)

`long type;`	Nachrichtennummer
`char *buf;`	Zeiger auf Nachrichtenpuffer
`long len, node, pid;`	Nachrichtenlänge, Nummer des Empfängerknotens und des Empfängerprozesses

Returnwert: interne Nachrichtennummer (mid)

- nicht blockierend, d.h. das Absenden der Nachricht wird angestoßen und, unabhängig davon, ob der Sendepuffer wieder frei ist, mit dem Programm fortgefahren

typischerweise kommt danach:

```
if (!msgdone(mid)) msgwait(mid);
```

NX - Asynchroner receive-Befehl

`long irecv(typesel, buf, len)`

`long typesel;`	Nachrichtennummer, -maske oder -1 für jede beliebige Nachricht
`char *buf;`	Zeiger auf Nachrichtenpuffer
`long len;`	Länge des Puffers
Returnwert:	interne Nachrichtennummer

- Empfang der Nachricht wird vorbereitet
- eventuell davor:
  `cprobe(typesel);` Abfrage, ob Nachricht angekommen ist

- und typischerweise danach:
  `msgwait(mid);` warte, bis asynchrone Operation beendet ist

NX - Informationsbefehle über eine empfangene Nachricht

`long infocount()` Länge der Nachricht (len)

`long infonode()` Prozessornummer des Senders (node)

`long infopid()` Prozeßnummer des Senders (pid)

`long infotype()` Nachrichtennummer (type)

allgemeine Informationsbefehle

`long mynode()` Knotennummer

`long mypid()` Prozeßnummer

`long numnodes()` Anzahl Knoten des Subcubes

`long nodedim()` Dimension des Subcubes

`unsigned long mclock()` Zeit in Millisekunden

NX - Wichtige globale Operationen

`gsync` Synchronisation aller Knotenprozesse mit der gleichen Prozeßnummer,

`gdsum` globales Aufsummieren (nach dem `g` steht ein `d` für *Double-precision*-, `i` für Integer- und `s` für Floatzahlen),

`gdprod` globale Multiplikation (nach dem `g` steht ein `d` für *Double-precision*-, `i` für Integer- und `s` für Gleitpunktzahlen),

`giand` globale, bitweise AND-Operation (entsprechend für OR und XOR),

`gland` globale, logische AND-Operation (entsprechend für OR und XOR),

`gopf` führt eine globale, vom Benutzer definierbare, assoziative
und kommutative Operation aus.

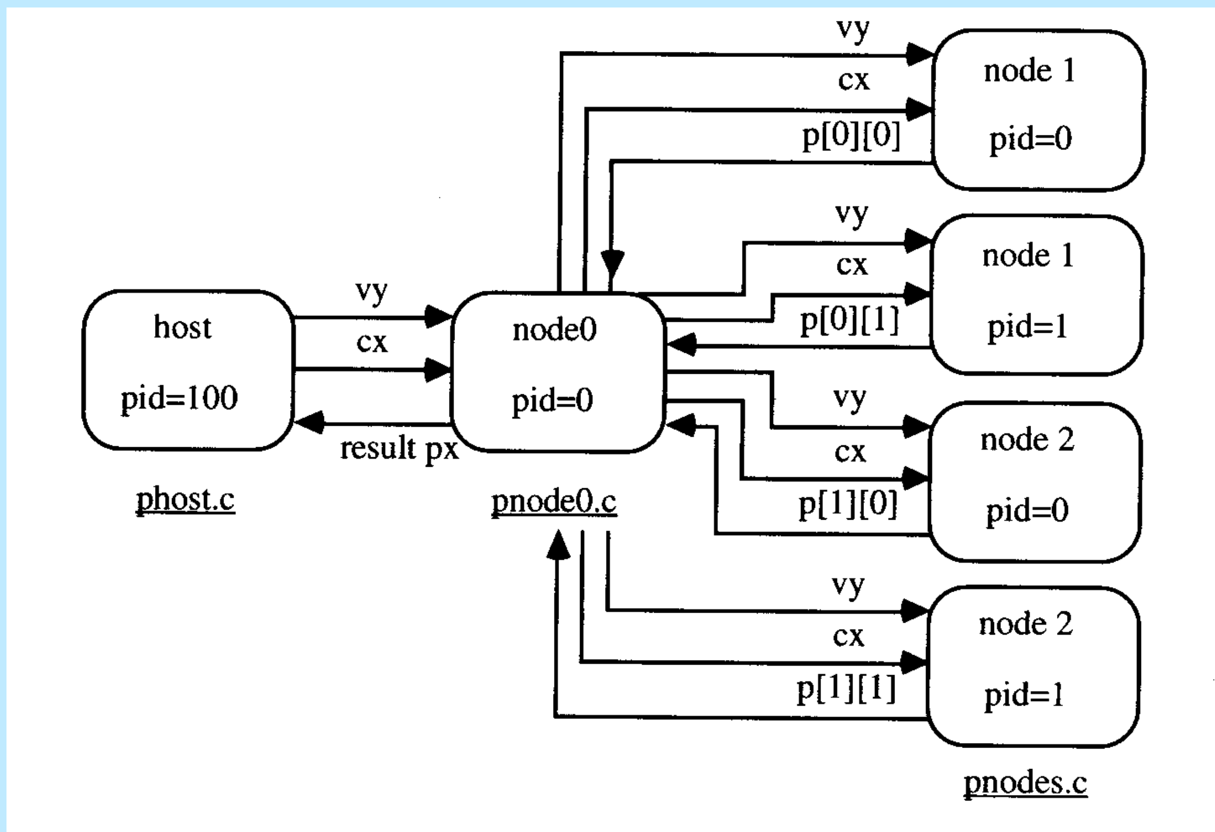
NX - Weitere Operationen

`csendrecv`	Senden einer Nachricht und Erzeugen einer Empfangsoperation für die Antwort mit Warten auf Fertigstellung
`gsendx`	Senden eine Nachricht an eine Liste von Knoten mit Warten auf Fertigstellung
`hsend`	Senden einer Nachricht und Erzeugen einer Operation, die nach Abschluß des Sendens aufgerufen wird
`hrecv`	Empfangen einer Nachricht und Erzeugen einer Operation (*handler procedure*), die nach Abschluß des Empfangens aufgerufen wird
`hsendrecv`	Senden einer Nachricht und Erzeugen einer Empfangsoperation als *Handler*-Prozedur für die Antwort, die nach Ankunft der Antwort aufgerufen wird

Nachrichtenorientierte Programmierung

- Programmbeispiel Matrixmultiplikation
  Das Programm multipliziert 2x2 Matrizen derart, daß jedes der vier Knotenprogramme genau ein Element der Resultatmatrix beisteuert. Das Programm ist auf einen Hypercubusrechner Intel iPSC/2 oder iPSC/860 zugeschnitten. Es werden der Einfachheit halber nur drei Knoten benutzt, doch läßt sich das Programm leicht skalieren. Das host-Programm läuft auf dem *System resource manager.* Es lädt das pnode0-Programm auf den Hypercube-Knoten 0 und das pnodes-Programm auf die Knoten 1 und 2, jeweils Prozeß 0 und 1, eines zweidimensionalen Hyperkubus. Die Kommunikationsstruktur ist in folgender Abbildung dargestellt (**pid** bedeutet Prozeßnummer, **vy** und **cx** sind die zu multiplizierenden Matrizen und px ist die Resultatmatrix).

Nachrichtenorientierte Programmierung am Beispiel Intels Paragon XP/S



- Programmbeispiel Matrixmultiplikation

Nachrichtenorientierte Programmierung am Beispiel Intels Paragon XP/S

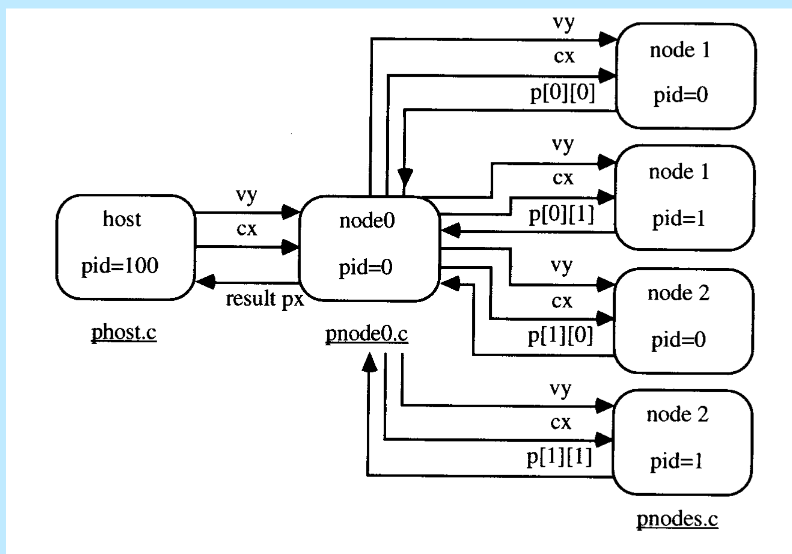
```
//  host program
#include <cube.h>
main()
int k,i,j;
double px[2][2],vy[2][2],cx[2][2],inbuf;
long outmsg_id, inmsg_id;
vy[0][0]=l.0; vy[0][1]=3.0; // Vorbelegung der
vy[1][0]=2.0; vy[1][1]=4.0; // Matrix
cx[0][0]=5.0; cx[0] [l]=7.0;
cx[1][0]=6.0; cx[1] [11=8.0;
setpid(100)
load("pnode0",0,0); // Verteilen der
load("pnodes",1,0); //Programme auf die
load("pnodes",1,1); //einzelnen Knoten
load("pnodes",2,0);
load("pnodes',2,1);
// use of an asynchronous send Operation
outmsg_id=isend(0,vy, 4*sizeof(double) ,0,0);
if ( !msgdone (outmsg_id)) msgwait (outlmsg_id);
csend( l,cx,4*sizeof(double) ,0,0); // use of a synchronous send Operation
crecv(-1,px,4*sizeof(double)); // synchronous receive operation
                               // recv result matrix from node 0
```

Nachrichtenorientierte Programmierung am Beispiel Intels Paragon XP/S

```
// pnode0 program for node 0
// receive 2 matrices from phost,
// distribute work to nodes,
// collect results from nodes, compress
// and send resultmatrix to phost
#include <cube.h>
main()
{
int k,i,j;
double px[2][2],vy[2][2],cx[2][2],inbuf;
long outnsg_id, inmsg_id;
crecv(0,vy,4*sizeOf(double)); // synchronous receive Operation)
crecv(1,cx,4*sizeOf(double)); // Matrizen vy und cx von host abholen
//  synchronous send Operations
k=2;
for(i=l;i<3;i++)              //        Matrizen an die Rechenknoten node 1 & 2
    for(j=0;j<2;j++)          //        an die Prozesse 0 & 1 senden
{csend(k,vy,4*sizeof(double) ,i,j);
 k++;                        //Incrementieren der Nachrichtennummer
 csend(k, cx,4*sizeOf (double) ,i,j);
 k++}
```

Nachrichtenorientierte Programmierung am Beispiel Intels Paragon XP/S

```
i=0;
while (i< 4)
{    cprobe(-1);    // wait until a message of any type arrives
// asyncbronous receive Operation
// receive message of any type
    inmsg_id=irecv(-l, &inbuf, sizeof (double));
    i++
// wait until receive Operation is completed
    msgwait(inmsg~id);
// infotype returns the type of the message
    switch ( infotype ())
       {
    case  10 : px[0][0]=inbuf; break;
    case  11 : px[0][l]=inbuf; break;
    case  20 : px[l][0]=inbuf; break;
    case  21 : px[l][l]=inbuf; break;
       }
}
// send result matrix to phost
csend(99,px,4*sizeOf(double) myhost() ,100);
}
```



Nachrichtenorientierte Programmierung am Beispiel Intels Paragon XP/S

```
// pnodes program for node 1&2 process 0 & 1

#include <oube.h>
main()
{
int k;
double res,vy[2] [2] ,cx[2] [2];
res=0. 0;

crecv(-l,vy,4*sizeof(double));
creov(-1,cx,4*sizeOf(double));

for ( k=0 ; k<2 ; k++)
   res +=  vy[mynode()-1] [k] * cx[k] [mypid()];

csend(10*mynode()+ mypid() ,&res,sizeof(double) ,0,0);
}
```

vy(0,0)*cx(0,0) + vy(0,1)*cx(1,0)

vy(0,0)*cx(0,1) + vy(0,1)*cx(1,1)

vy(1,0)*cx(0,0) + vy(1,1)*cx(1,0)

vy(1,0)*cx(0,1) + vy(1,1)*cx(1,1)

Datenparallele Programmierung
High Performance Fortran

Datenparallele Programmierung OpenMP

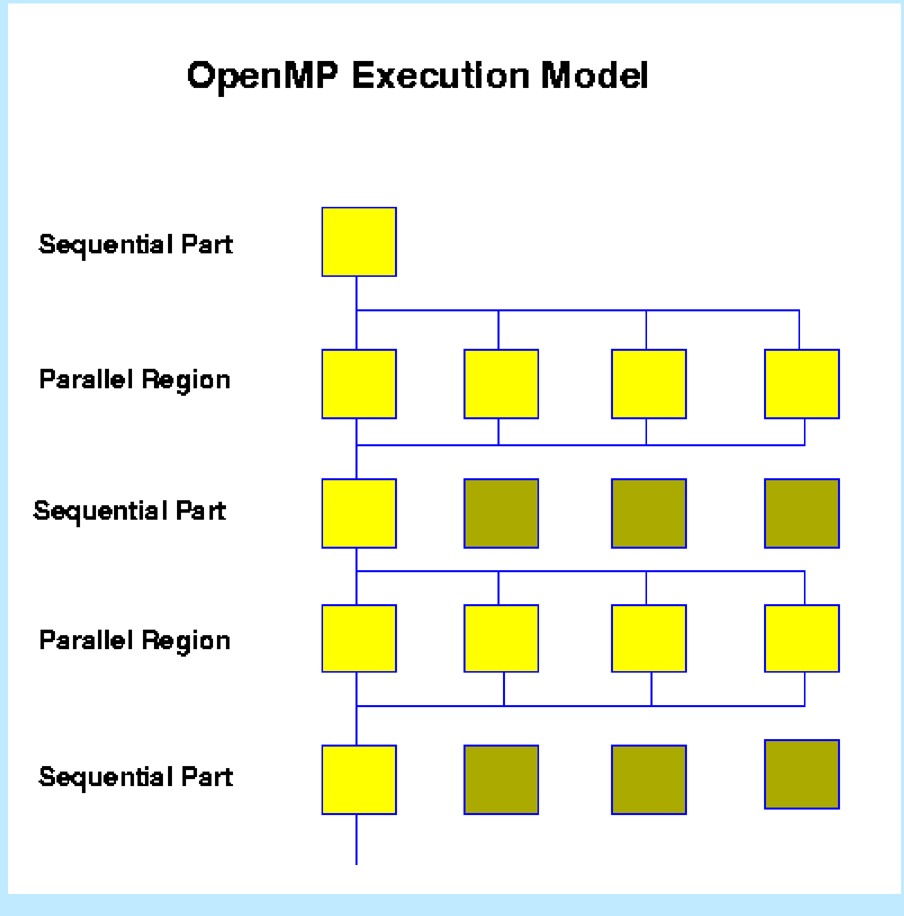
OpenMP umfaßt eine Sammlung von Compiler-Direktiven, Bibliotheks-Funktionen und definierten Umgebungsvariablen zur Ausnutzung der parallelen Eigenschaften von **Shared-Memor**y-Maschinen. OpenMP ist für Fortran sowie C / C++ definiert.
Ziel: Bereitstellung eines portablen, herstellerunabhängigen paral. Programmiermodells.
Synchronisation: Verwendet das sogen. **fork-join**-Modell für die parallele Ausführung. Ein Programm startet als ein einzelner **Thread** *,* der **Master Thread**. Beim Erreichen des ersten parallelen Konstrukts erzeugt der Master Thread eine Gruppe von Theads und wird selbst Master der Gruppe. Nach Beendigung des parallelen Konstrukts synchronisiert sich die Gruppe, und nur der Master Thread arbeitet weiter. Die Parallelisierung erfolgt auf Schleifenebene.
Beispiel: Fortran-Unterprogramm, wie es häufig bei der numerischen Berechnung diskreter zweidimensionaler Probleme auftritt.
Es initialisiert zwei Felder für die diskrete numerische Lösung u(i,j) und die exakte Lösung f(i,j). Die Doppelschleife wird von den verfügbaren Prozessoren parallel verarbeitet ***(parallel do),*** wobei die Indexbereiche für i und j verteilt werden. Die beiden Felder sowie die Größen n, m, dx, dy und alpha stehen allen Prozessen global zur Verfügung (werden ***geshared),*** die Größen i,j,xx,yy sind pro Prozeß lokal ***(private).***

OpenMP - Beispielprogramm

```
subroutine initialize (n,m,dx,dy,u,f,alpha)
! Initializes data. Assumes exact solution is u(x,y) = (1-x^2)*(1-y^2)
      implicit none
      integer n,m
      real u(n,m),f(n,m),dx,dy,alpha
      integer i,j, xx,yy
      dx = 2.0 / (n-1)
      dy = 2.0 / (m-1)
!$omp parallel do
!$omp& shared(n,m,dx,dy,u,f,alpha)
!$omp& private(i,j,xx,yy)
      do j = 1,m
            do i = 1,n
                  xx = -1.0 + dx * (i-1)     ! -1 < x < 1
                  yy = -1.0 + dy * (j-1)     ! -1 < y < 1
                  u(i,j) = 0.0
                  f(i,j) = -alpha *(1.0-xx*xx)*(1.0-yy*yy)- 2.0*(1.0-xx*xx)-
                  2.0*(1.0-yy*yy)
            enddo
      enddo
!$omp end parallel do
      return
      end
```

Die Quelle des seriellen Programms muss lediglich um Kommentarzeilen mit den OpenMP-Direktiven erweitert werden. Weiterer Vorteil : die Parallelisierung kann schrittweise durchgeführt werden.
(n,m,dx,dy,u,f,alpha)

OpenMP – Fork-Join Execution Model



OpenMP – Vergleich mit anderen Parallelisierungsmethoden

	MPI (shared und di-stribute memory Systeme)	**OpenMP (shared memory Systeme)**	**proprietäre Direktiven**	**HPF**
portabel	Ja	Ja	Nein	Ja
skalierbar	Ja	Teilweise	Teilweise	Ja
unterstützt Datenparallelität	Nein	Ja	Ja	Ja
unterstützt inkrementelle Parallelisierung	Nein	Ja	Ja	Teilweise
Serielle Funtionalität bleibt erhalten	Nein	Ja	Ja	Ja
Korrektheit verifizierbar	Nein	Ja	?	?

Generelle Aussage: Mit OpenMP erhält man - im Gegensatz zu MPI - sehr schnell ein funktionsfähiges paralleles Programm; um aber auch gute parallele Performance zu erhalten, muss man noch einiges an Optimierungsarbeit hineinstecken und kommt dann doch nicht ganz an die von MPI erreichbare Skalierungsfähigkeit des parallelen Codes heran.

OpenMP – Vergleich mit anderen Parallelisierungsmethoden

Angesichts der von gegenwärtigen Implementierungen vorgegebenen Einschränkung des Einsatzes auf shared-memory Systeme ist es auf Hochleistungsrechnern mit entsprechender Architektur sinnvoll, MPI und OpenMP *komplementär* zueinander zu verwenden. Schematisch ergibt sich dann folgende Hierarchisierung:

- Die Arbeit wird in möglichst grosse Untereinheiten zerlegt, die mittels MPI auf die einzelnen leistungsfähigen Knoten verteilt wird.
- Die Untereinheiten werden mit OpenMP-Direktiven versehen; jeder Knoten generiert eine passende Anzahl von Threads, die dann die Untereinheiten bearbeiten.
- Auf der tiefsten Stufe (innerste Schleifen) verbleiben dann noch die Optimierungs-möglichkeiten, die schon bisher die Compiler bieten: Optimale Cache-Nutzung (RISC-basierte Prozessoren), Vektorisierung (Cray T90), Pseudo-Vektorisierung (SR8000-F1) etc.

Message Passing Interface MPI

- Entwickelt seit Januar 1993 vom MPI-Forum unter Mitwirkung von ca. 60 Personen aus über 40 Organisationen aus Industrie, Forschungs-instituten und Universitäten aus Amerika und Europa.
- MPI dient der Programmierung von parallelen Systemen mit verteiltem Speicher.
- MPI ist eine standardisierte Bibliothek, die Funktionen zur Inter-Prozessommunikation und Verteilung der Prozesse auf die Prozessoren beinhaltet.
- Ziel von MPI ist es, für Programme und Bibliotheken Portabilität zu gewährleisten.
- MPI soll langfristig Kontinuität der parallelen Programmierschnittstelle gewährleisten, damit Programmentwickler nicht in wenigen Jahren auf Grund eines neuen Standards oder einer neuen Rechnergeneration noch einmal von vorne anfangen müssen

Pi Berechnung - Das Näherungsverfahren

compute pi by integrating f(x) = 4/(1 + x**2)

```
program calcpi
implicit none
integer * 4 I
integer * 4 N           !Anzahl Iterationen
REAL*8 Pi               !Ergebnis
REAL*8 H                !Interval x1-x2
REAL*8 x                !Wertebereich von 0..1

READ   (UNIT = *, FMT = *) N
Pi = 0.
H   = 1./N
DO i = 1,N
    x = H * (FLOAT(I) - 0.5)
    pi = pi + 4./(1.+(x*x))
ENDDO
print *, 'pi', i, pi * h
end
```

Pi Berechnung - Beispiele

compute pi by integrating f(x) = 4/(1 + x**2)

1.X	Letztes X	Pi	Anzahl Iterationen	Interval
0.05	0.95	3.1424260148158 2	10	0.100
0.025	0.97	3.1418010166985 2	20	0.050
0.004	0.99	3.1416009422194 5	100	0.009
0.0005	0.99	3.1415928319180 3	1000	0.0001

MPI – Pi Berechnung

```
!******************************************************************************
!    fpi.f90 - compute pi by integrating f(x) = 4/(1 + x**2)
!
!   Variables:
!
!     pi          the calculated result
!     n           number of points of integration.
!     x            midpoint of each rectangle's interval
!     f            function to integrate
!     sum,pi   area of rectangles
!     tmp        temporary scratch space for global summation
!     i            do loop index
!****************************************************************************
!
program Main
  !
  ! use MPI
  !.. Implicit Declarations ..
  implicit none
  !
! #include "mpif.h"
   include 'mpif.h'
  !
  !.. Local Scalars ..
  integer, parameter :: master = 0
  integer :: i,n,myid,numprocs,ierr,rc
  double precision, parameter :: pi25dt = 3.141592653589793238462643d0
  double precision :: a,h,mypi,pi,sum,x
  !
  !.. Intrinsic Functions ..
  intrinsic ABS, DBLE
```

MPI – Pi Berechnung

```
  !.. Statement Function to integrate
  double precision :: f
  f(a) = 4.d0 / (1.d0+a*a)
     call MPI_INIT( ierr )                                            ! Initialisierung der MPI-
Umgebung
     call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )     ! Prozesskennung erfragen
     call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr ) ! Zahl der Geschwister
erfragen
  !
  ! ... Executable Statements ...
  do
    if ( myid .eq. 0 ) then
       write (UNIT=*, FMT=‘(‘’Enter the number of intervals: (0 quits)’’)’)
       read (UNIT = *, FMT = ‘( I10)’)  n
   end if
   call MPI_BCAST(n,1,MPI_INTEGER,master,MPI_COMM_WORLD,ierr) ! N wird an alle
geschickt
    if (n <= 0) exit      ! check for quit signal
    h = 1.0d0 / n         ! calculate the interval size
    sum = 0.0d0
    do i = myid+1, n, numprocs           !z.B. n=2, h = 0.5
       x = h * (DBLE(i)-0.5d0)           ! X1 = 0.5*0.5 = 0.25,  x2=0.5*1.5=0.75
       sum = sum + f(x)                   !sum1 = 4/1+0.25*0.25)               = 3.7647,
     end do                                 !sum2 = 3.7647+4/(1+(0.75*0.75)) = 6.3247
   mypi = h * sum                          !mypi = 0.5* 6.3247  = 3.1624
   ! Für alle  mypi wird das arithmetische Mittel gebildet
   call
MPI_REDUCE(mypi,pi,1,MPI_DOUBLE_PRECISION,MPI_SUM,master,MPI_COMM_WORLD,ierr
)
   if (myid .eq. 0) then
       print*, ‘Pi ist :’,  pi,’Fehler ist:’,  ABS(pi-pi25dt)
   endif
 end do
```