VERTEILTE UND PARALLELE PROGRAMMIERUNG

PD Stefan Bosse

Universität Koblenz-Landau, Fak. Informatik
Universität Bremen, FB Mathematik & Informatik

SS 2018

Version 2018-07-16

sbosse@uni-koblenz.de

INHALT

1. Inhalt
2. Überblick
3. Einführung in die verteilte und parallele Datenverarbeitung
4. Das Prozessmodell
5. Zustands-Raum Diagramme
6. Petri Netze
7. Parallelisierung
8. Parallele Programmierung
9. Synchronisation und Kommunikation
10. Parallelisierung und Metriken
11. Plattformen und Netzwerke
12. Verteilte Programmierung und Systeme
13. Referenzen

ÜBERBLICK

SCHWERPUNKTE IN DIESEM KURS

- Grundlagen von parallelen und verteilten Systemen
- Konzepte der parallelen und verteilten Programmierung
- Praktische Relevanz und Anwendung
- Plattformen und Technologien
- Netzwerke, Nachrichten, und Protokolle (MPI,..)

Begleitet von Übungen um obige Techniken konkret anzuwenden

Vorlesung
2 SWS mit Grundlagen und Live Programming

Übung
2 SWS mit Programmierung und angewandter Vertiefung

Voraussetzungen
Grundlegende Programmierfähigkeiten, Grundkenntnisse in Rechnerarchitektur und Netzwerken

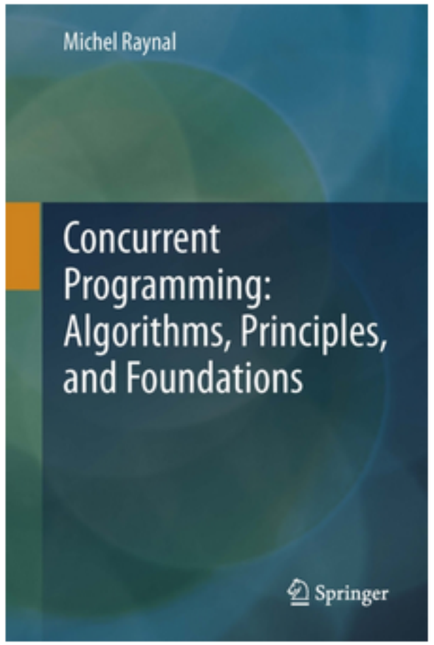
LITERATUR

Vorlesungsskript

Die Inhalte der Vorlesung werden sukzessive bereitgestellt

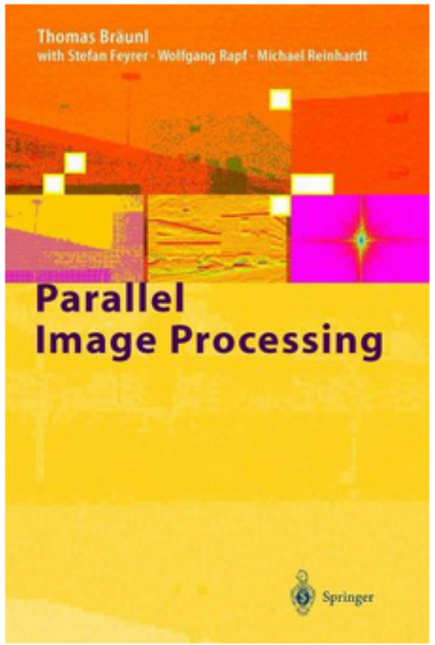
Concurrent Programming: algorithms, Principles, and Foundations

Michel Raynal, Springer 2013,
ISBN 978-3-642-320626-2



Parallel Image Processing

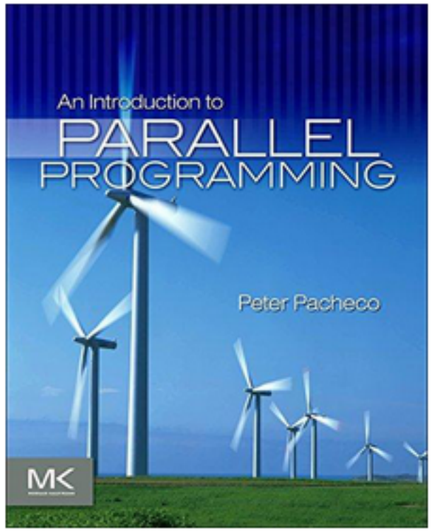
T. Bräunl, Springer Berlin, 2001



LITERATUR

An introduction to parallel programming

P. S. Pacheco, Elsevier, MK, 2011.



SOFTWARE

Verwendete Software (Vorlesung und Übung):

threads.js/parallel.js

sun45.informatik.uni-bremen.de

- Programmierung in JavaScript
- Bibliothek für Parallele und Verteilte Systeme
- Einfach zu Erlernen
- Auf jeder JavaScript VM ausführbar

```
T=require('./threads');
S={ev:T.Event()};
T.Thread(function (id) {
  ev.await();
  print('Hey got the event!');
},'mytask1',S);
```

jxcore

sun45.informatik.uni-bremen.de

- JavaScript VM mit Multithreading
- Ausführung von der Kommandozeile
- Wird auch für Live Programming genutzt
- Einsatz auf verschiedenen Hostplattformen : PC, Smartphone, Embedded PC, Server, ..
- Einsatz auf Betriebssystemen: Windows, Linux, Solaris, MacOS, Android

```
> jx threads_demo.js
```

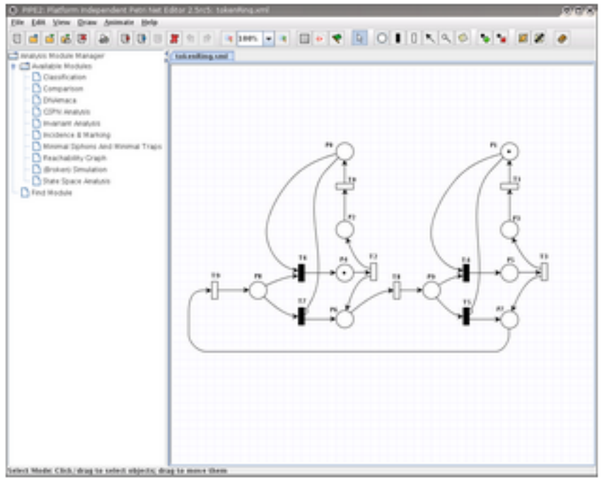
SOFTWARE

Verwendete Software (Vorlesung und Übung):

pipe

pipe2.sourceforge.net

- GUI basierte Simulation und Analyse von Petri Netzen
- Modellierung von parallelen Systemen mit Diagrammen



ZIELE

1. Verständnis der Grundprinzipien und Architekturen verteilter (VS) und paralleler Systeme (PS) und Fähigkeit zum Transfer auf technische Systeme
2. Verständnis und Fähigkeit der programmatischen Anwendung von Synchronisation und Kommunikation in VS und PS
3. Verständnis der Probleme und dem Betrieb von parallelen Systemen im Vergleich zu sequenziellen Systemen (Effizienz, Blockierung, Skalierung, Ressourcenbedarf)
4. Praktische Kenntnisse der Programmierung von PS und VS anhand von Programmierübungen mit JavaScript und jxcore (node.js)
5. Erkenntnisse von Grenzen und Möglichkeiten der Parallelisierung und Verteilung und die Fähigkeit effiziente Systeme zu entwickeln
6. Vorbereitung für Methoden und zukunftige Trends im Cloud Computing und Internet der Dinge

INHALTE

A. Parallele und Verteilte Systeme
B. Sequenzielle und Parallele Datenverarbeitung
C. Funktionale, Sequenzielle, und Parallele Komposition
D. Prozessmodelle, Petri-Netze
E. Kommunizierende Prozesse
F. Synchronisation und Kommunikation (Primitiven)
G. Praktische Parallele Programmierung mit JavaScript und threads.js/parallel.js
H. Parallelisierung: Methoden und ALgorithmen
I. Netzwerke und Nachrichrenaustausch
J. Praktische Verteilte Programmierung mit JavaScript und parallel.js

EINFÜHRUNG IN DIE VERTEILTE UND PARALLELE DATENVERARBEITUNG

VERTEILTE VS. PARALLELE SYSTEME

Verteiltes System

Ein verteiltes System ist eine Sammlung von **lose gekoppelten** Prozessoren oder Computern, die über ein Kommunikationsnetzwerk miteinander verbunden sind (**Multicomputer**).

- **Speichermodell** : Verteilter Speicher → Jeder Prozessor verfügt über privaten Speicher
- **Kommunikation** : Nachrichtenbasiert über Netzwerke
- **Ressourcen** : Nicht direkt geteilt

Paralleles System

Ein paralleles System ist eine Sammlung von **stark gekoppelten** Prozessoren (**Multiprozessoren**)

- **Speichermodell** : Gemeinsamer Speicher
- **Kommunikation** : Direkt über elektrische Signale → Switched Network (Kreuzschiene) | Bus → Punkt-zu-Punkt | Punkt-zu-N-Netzwerke
- **Ressourcen** : Gemeinsam genutzt (Bus, Speicher, IO)

VERTEILTE VS. PARALLELE SYSTEME

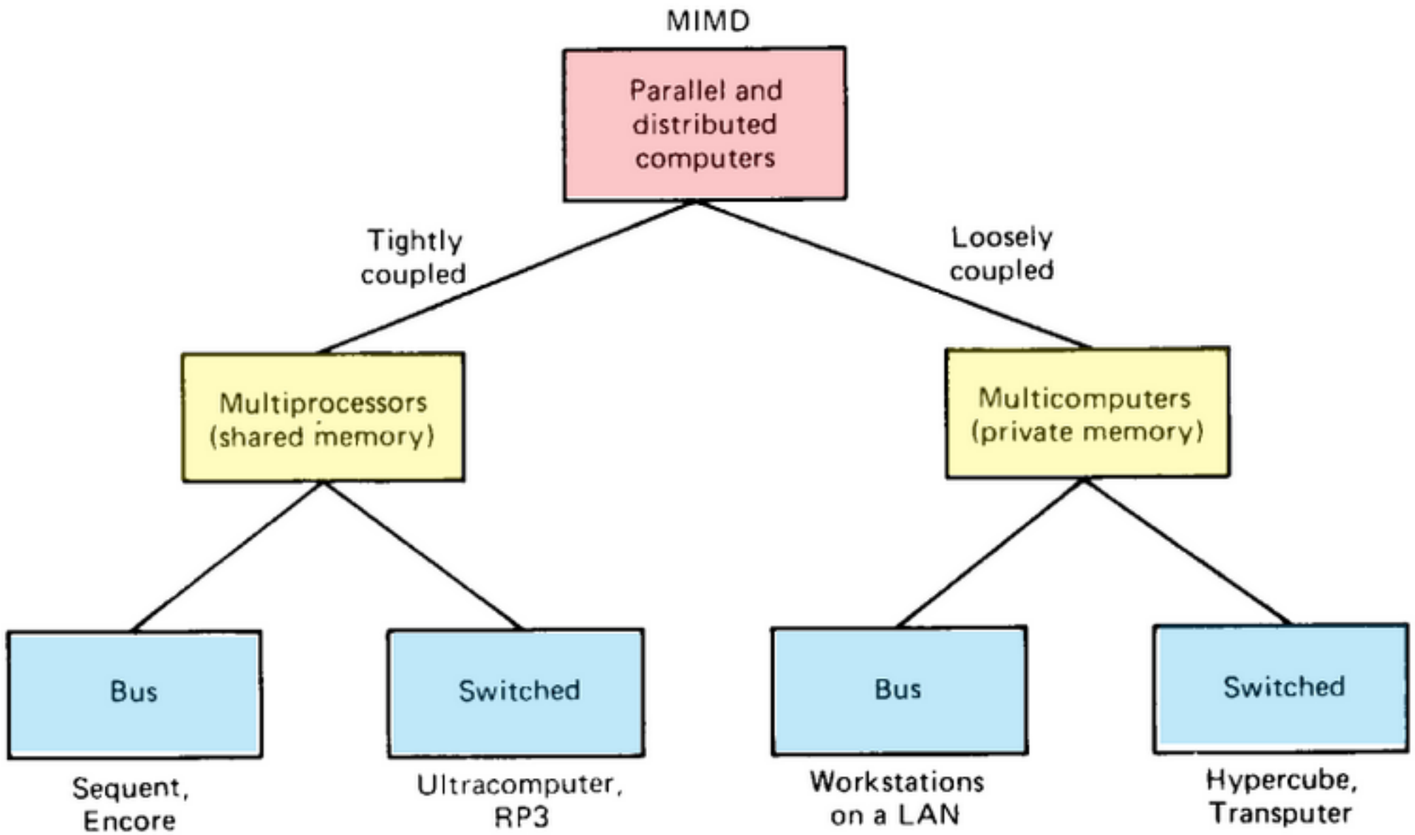


Abb. 1. Taxonomie von verteilten und parallelen Systemen [1]

VERTEILTE VS. PARALLELE SYSTEME

Verteilter Speicher

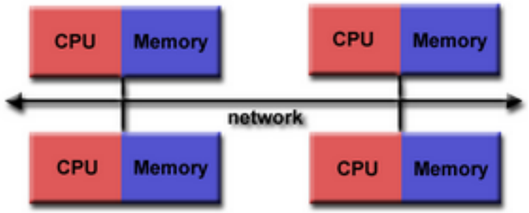
- Zugriff auf Speicher erfordert Netzwerkkommunikation
- Vorteil: Speicher ist skalierbar mit Anzahl der Prozessoren
- Nachteil: Langsamer Speicherzugriff zwischen Prozessen

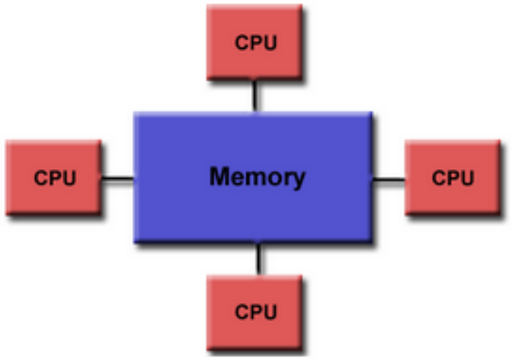
Unified Memory Architecture

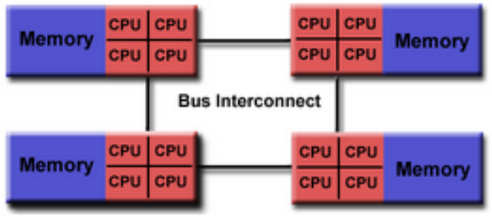
- Symmetrisches Multiprocessing (SMP)
- Vorteil: Konstante Zugriffszeit auf Speicher
- Vorteil: Schneller Speicherzugriff zwischen Prozessen

Non Unified Memory Architecture

- Vorteil: Clustering von SMPs
- Nachteil: Ungleiche Zugriffszeiten auf Speicher



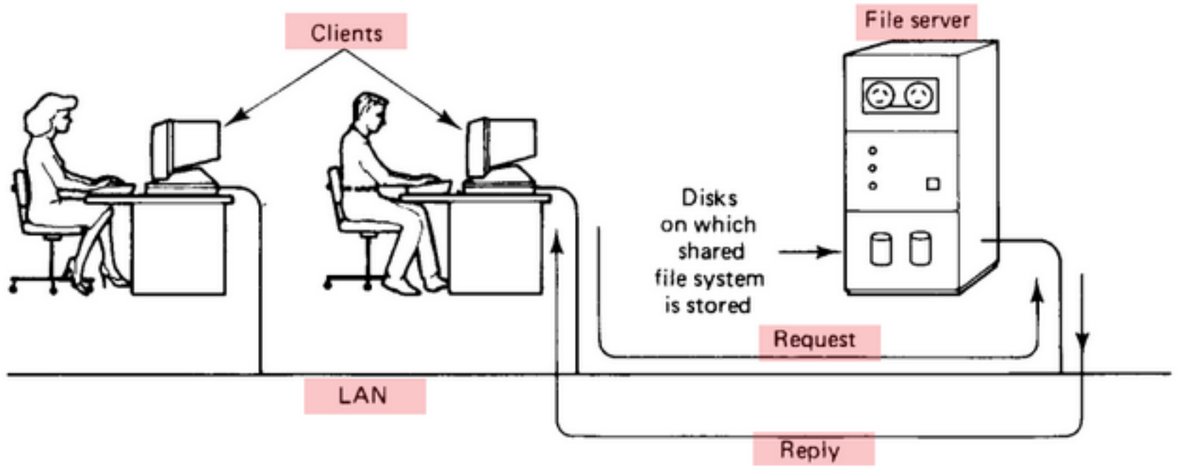




[computing.llnl.gov]

VERTEILTE SYSTEME

- Historisch basierend auf **Netzwerk-Betriebssystemen** (z. B. Linux-Clustern): *Benutzer sind sich der Vielzahl von Maschinen bewusst!*
  - Tools: Remote Login (`telnet`, `ssh`), Remote Desktop (Windows), Dateitransfer (`FTP`, `SSH`), Netzwerk Dateisystem (`NFS`)
- Ein **Verteiltes Betriebssystem** verbirgt die einzelnen Maschinen: *Benutzer sind sich der Vielzahl von Maschinen nicht bewusst!*
  - Zugriff auf entfernte Ressourcen ähnlich wie der Zugriff auf lokale
  - Übertragung von Prozessen (Berechnung) und Programmcode (anstatt Daten)



[Tanenbaum]

VERTEILTE SYSTEME

Geschichte der Verteilten Betriebssysteme (DOS)

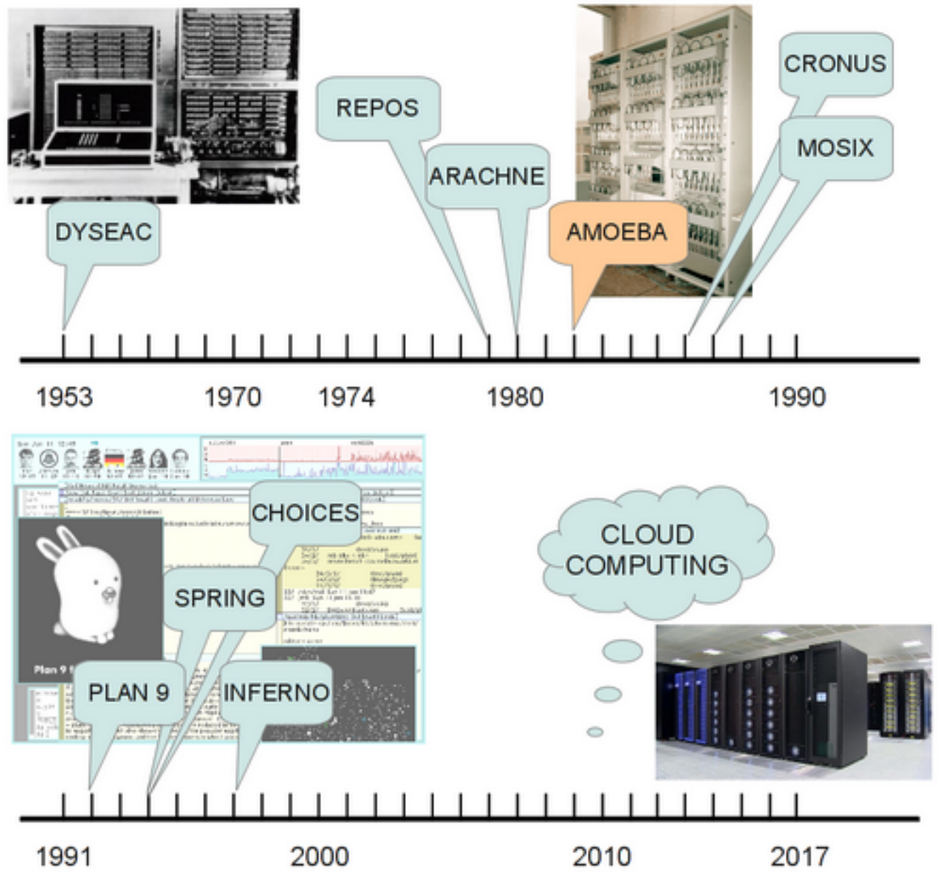


Abb. 2. Zeitleiste einiger ausgewählter DOS. Goldenes Zeitalter der DOS-Entwicklung war in den 80er und 90er Jahren!

VERTEILTE SYSTEME

Entwurfskriterien und Eigenschaften

Namensgebung
Wie können wir ein Objekt benennen, das weit entfernt an einem unbekanntem Ort ist?

Robustheit
Was passiert, wenn eine Maschine oder ein Netzwerk ausfällt?

Sicherheit
Wie können wir unser System vor Versagen, Betrug, Eindringen, Diebstahl von Daten, … schützen?

Performance
Langsamer als je zuvor?

Konsistenz
Ich machte eine Banktransaktion, die Bestätigung der Transaktion ging verloren, und die Transaktion wurde wiederholt. Mein Konto wurde zweimal belastet?

Skalierbarkeit
Was passiert mit diesen Kriterien, wenn wir die Anzahl der Maschinen um das Zehnfache erhöhen?

PARALLELE SYSTEME

Definition

- Zerlegung (Partitionierung) eines sequenziellen Algorithmus oder eines Programms in **parallele Tasks** (Prozesse) → **Parallele Komposition**
- Ausführung der Prozesse parallel (nebenläufig und ggfs. konkurrierend) auf mehreren Verarbeitungseinheiten (u. A. generische programmgesteuerte Prozessoren)

Motivation für parallele Datenverarbeitung

- Verkleinerung der Berechnungslatenz

Def. Latenz: Gesamte oder Teilbearbeitungszeit eines Datensatzes

- Erhöhung des Datendurchsatzes

Def. Datendurchsatz: Anzahl der verarbeiteten Datensätze pro Zeiteinheit

PARALLELE SYSTEME

- Latenz und Bandbreite sind zunächst unabhängig!
- Pipelining kann die Bandbreite erhöhen (nur Sinnvoll bei Datenströmen)
- Parallele Tasks können die Latenz verringern

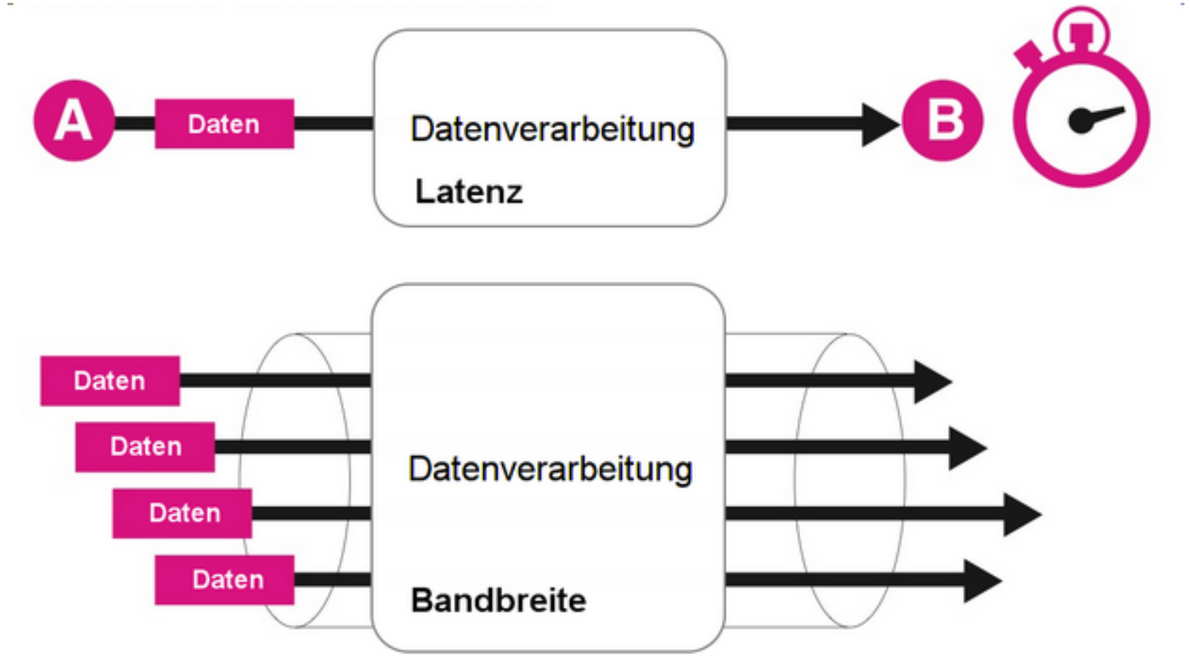


Abb. 3. Unterschied Latenz zu Bandbreite

PARALLELE SYSTEME

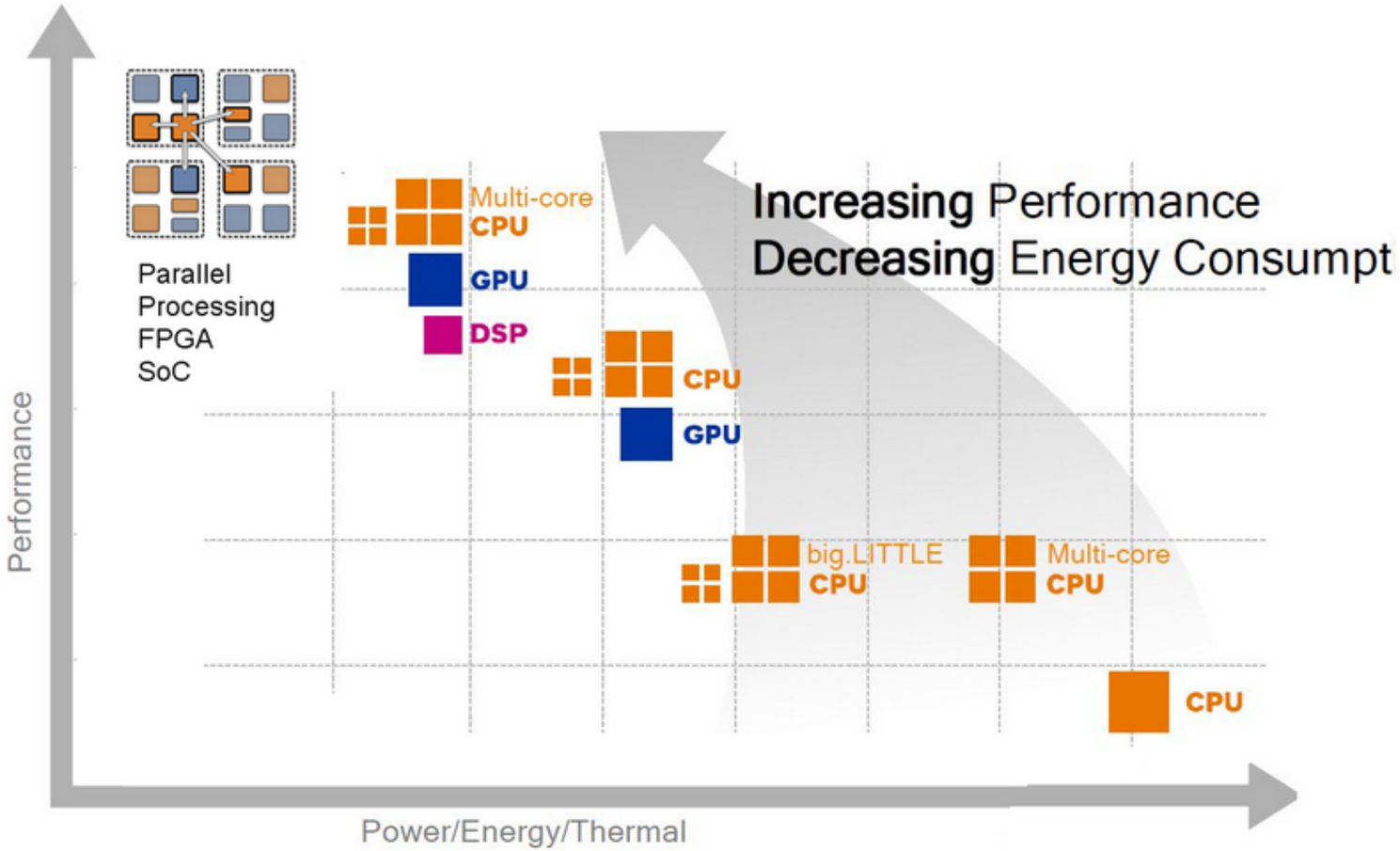
Man unterscheidet: Parallele Rechnerarchitektur und parallele Datenverarbeitung

Motivation für parallele Datenverarbeitung

- Steigerung der Energieeffizienz von mikroelektronischen Systemen:
  - Komplexe generische Einprozessoranlagen besitzen ungünstige Energieeffizienz, aber:
  - Parallelisierung kann zu verbesserter Energieeffizienz des Gesamtsystems führen!
- Skalierung von parallelen Rechnern auf reine Digitallogiksysteme für anwendungsspezifische Lösungen kann deutliche Reduktion der Hardware-Komplexität und der elektrischen Leistungsaufnahme bedeuten!
  - System-on-Chip Entwurf
  - Leistungsaufnahme eine CMOS Digitalschaltkreises (f: Frequenz, N: Schaltende Elemente, U: Spannung, C: Technologie):

$$P(f, N, U, C) \sim fNU^2C \tag{1}$$

PARALLELE SYSTEME



[Qualcomm]

Abb. 4. Erhöhung der Performanz und der Energieeffizienz durch Parallelisierung

PARALLELE SYSTEME

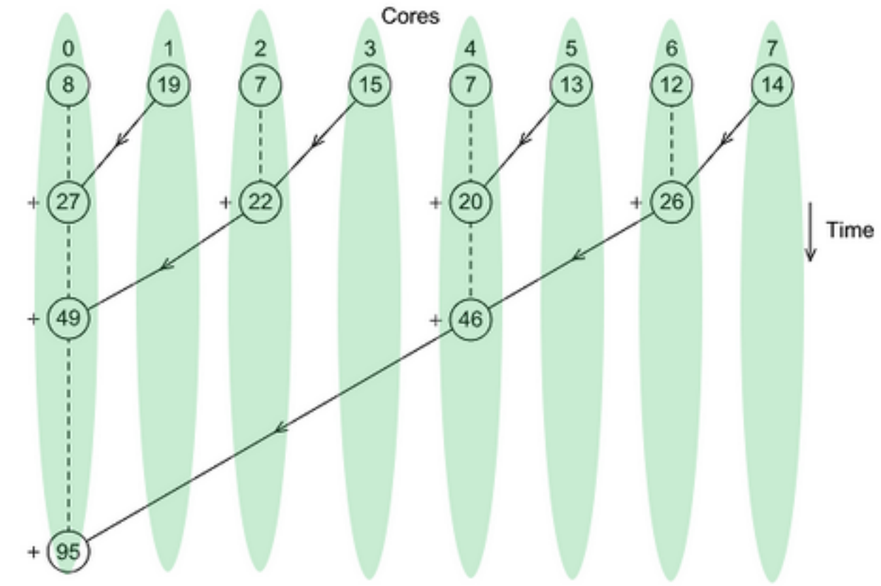
Anwendung paralleler Datenverarbeitung

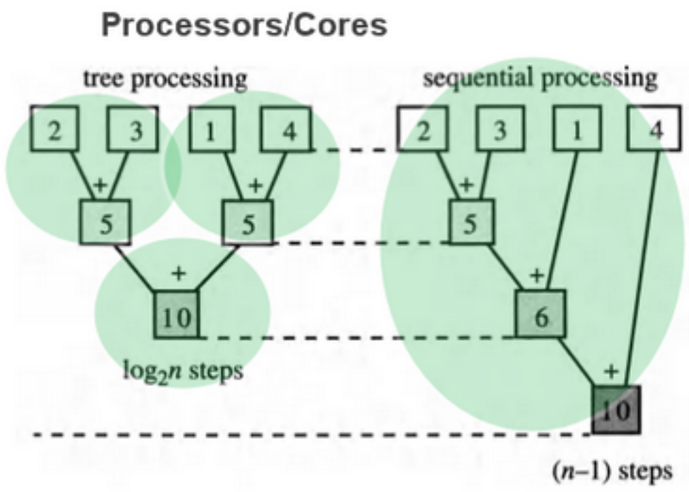
- Digitale Bildverarbeitung und automatische Bildinterpretation (Vision)
- Datenkompression (Video, mpeg)
- Komplexe Steuerungssysteme mit großer Anzahl von Freiheitsgraden, wie z.B. Positionierungssteuerung von Robotergelenken und Maschinen
- Kommunikation, z. B. nachrichten-basiertes Routing
- Kryptoverfahren
- Parallele numerische Verfahren, wie z. B. Lösung von Differentialgleichungen (Strömungsmechanik, Elektromagnetische Wellenausbreitung, Wetter- und Klimamodelle)

PARALLELE SYSTEME

Beispiel Numerik und Digitale Bildverarbeitung

- Welche Probleme und Nachteile gibt es?



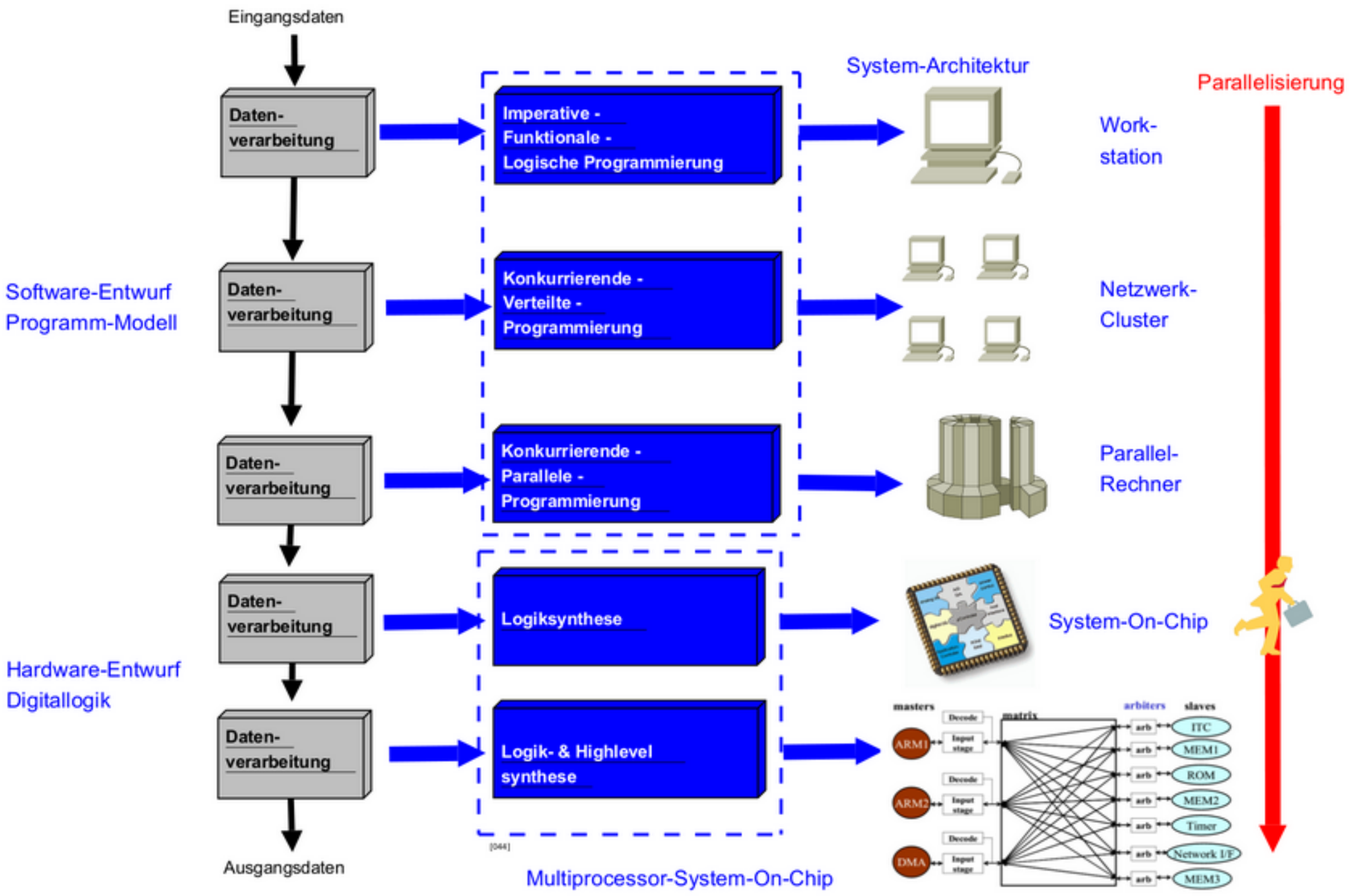


[1,2]

Abb. 5. Parallele Berechnung der Summe von Bildpixeln mit Baumstruktur

PARALLELE SYSTEME

Ebenen der Datenverarbeitung in Abhängigkeit vom Parallelisierungsgrad



PARALLELE SYSTEME

Rechnerarchitekturen in der Datenverarbeitung

- Für sequenzielle Datenverarbeitung existieren generische programmgesteuerte Rechneranlagen (von-Neumann Architektur), die die meisten sequenziellen Algorithmen effizient ausführen können.
- Das Programm besteht dabei aus einer Vielzahl elementarer einfacher Maschinenbefehle → **Sequenzielle Komposition**

Es gibt keine generischen Rechneranlagen und Architekturen für parallele Datenverarbeitung

- Parallele Algorithmen kann man klassifizieren.
- Die Klasse bestimmt die Rechnerarchitektur, die diese Algorithmen effizient verarbeiten können.

PARALLELE SYSTEME

Beispiel Sensorisches Material

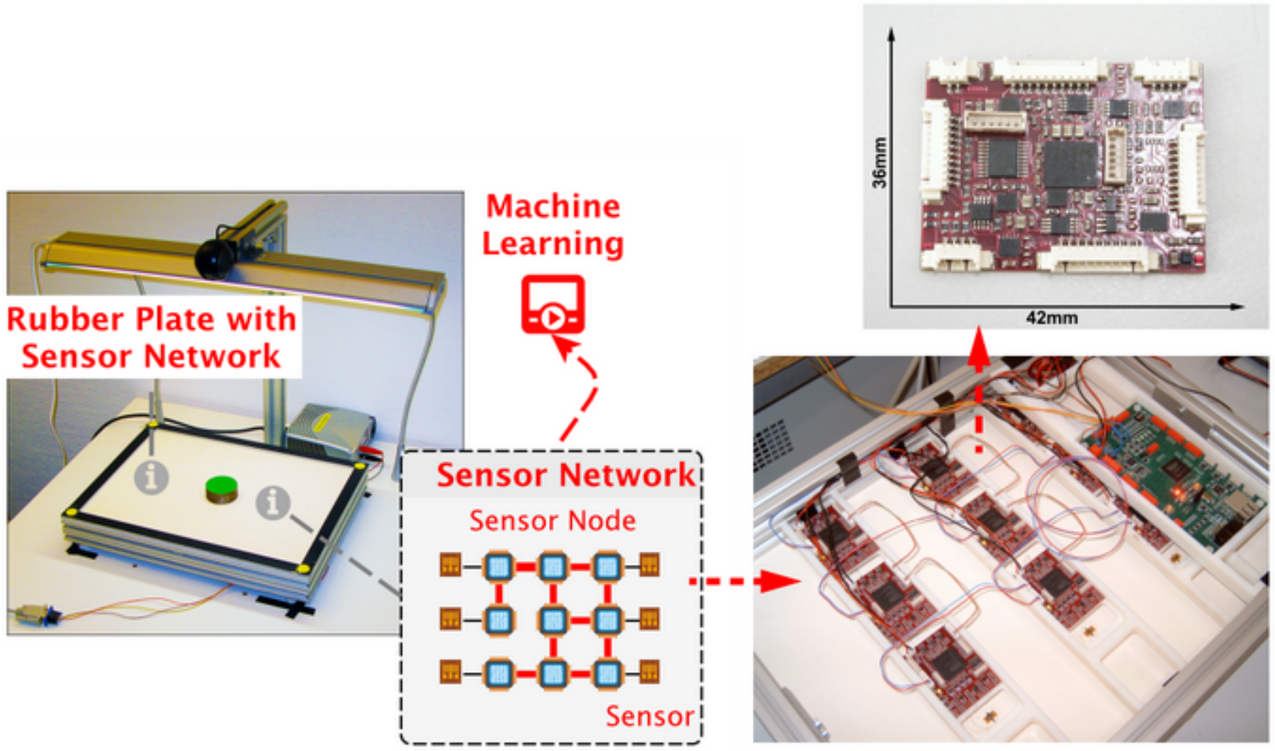


Abb. 6. Beispiel eines verteilten und parallelen Systems: Sensornetzwerk (unten rechts) mit Maschen Topologie mit System-on-Chip Sensorknoten (oben rechts) mit massiv paralleler Datenverarbeitung. Sensorik: Dehnungsmessstreifen auf Gummiplatte (links)

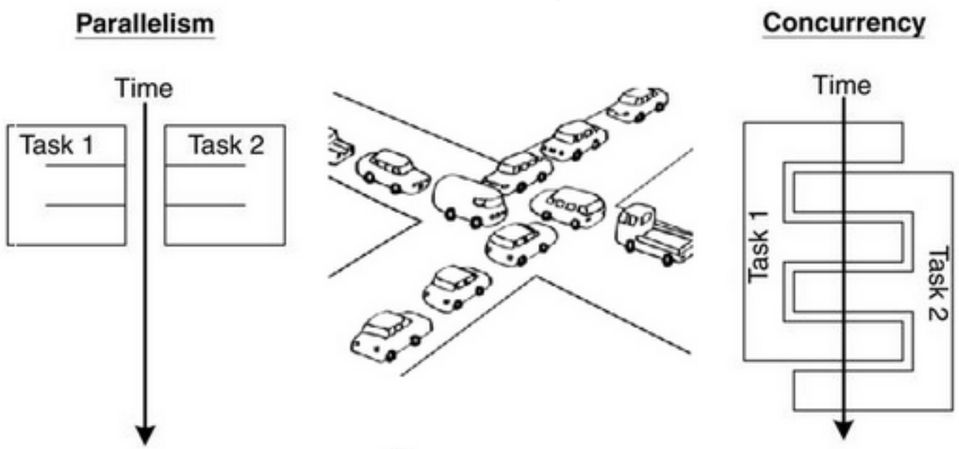
PARALLELE SYSTEME

Parallelität und Nebenläufigkeit

- Parallelität und Nebenläufigkeit ist ein zeitliches Ablaufmodell
- Beschreibt eine zeitliche Überlappung oder Gleichzeitigkeit bei der Ausführung von parallelen Prozessen
- Nebenläufigkeit kann ohne Synchronisation auskommen!

Konkurrenz

- Concurrent → übereinstimmend!
- Konkurrenz beschreibt den Wettbewerb um geteilte Ressourcen!
- Wettbewerb bedeutet Konflikt welcher aufgelöst werden muss!
- Synchronisation zw. Prozessen!
- Konsens Programmiermodell!

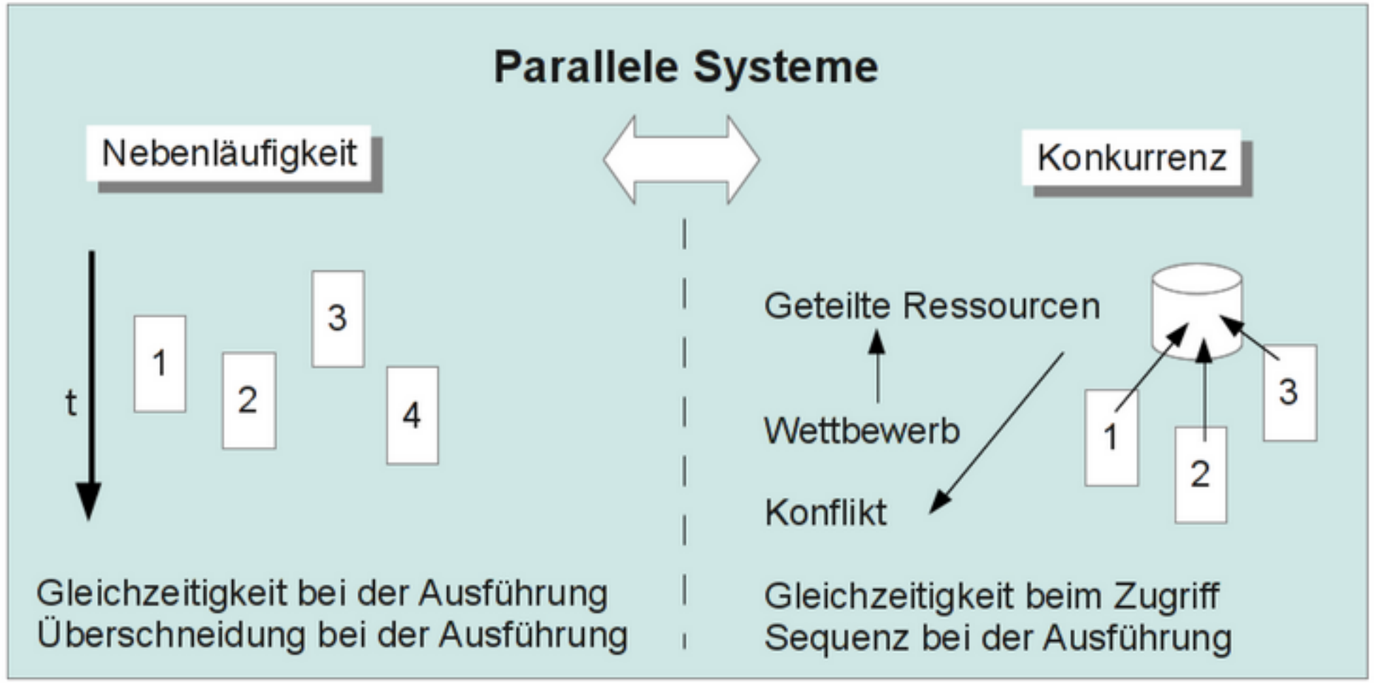


[blog.golang.org]

PARALLELE SYSTEME

Parallele und verteilte Datenverarbeitungssysteme vereinen Nebenläufigkeit und Konkurrenz

- Nebenläufigkeit durch Ausführungsplattform
- Konkurrenz muss durch Programmierung gelöst werden



SEQUENZIELLE VS. PARALLELE DATENVERARBEITUNG

- Bei der sequenziellen Datenverarbeitung gibt es einen Algorithmhus, der durch ein Program implementiert wird, das durch einen Prozess auf einem Prozessor ausgeführt wird.
- Das Programm besteht aus einer Sequenz von Operationen die exakt in der Reihenfolge hintereinander ausgeführt werden.
- Bei einem Parallen Programm findet eine Partitionierung in Subprogramme statt, die von mehreren Prozessen i.A. auf verschiedenen Prozessoren ausgeführt werden.
- Drei Phasen einer parallelen Programmausführung:
  1. **Verteilungsphase** der Eingabedaten
  2. **Ausführungsphase** mit Paralleler Verarbeitung
  3. **Zusammenführungsphase** der Ausgabedaten

Was könnte noch fehlen?

SEQUENZIELLE VS. PARALLELE DATENVERARBEITUNG

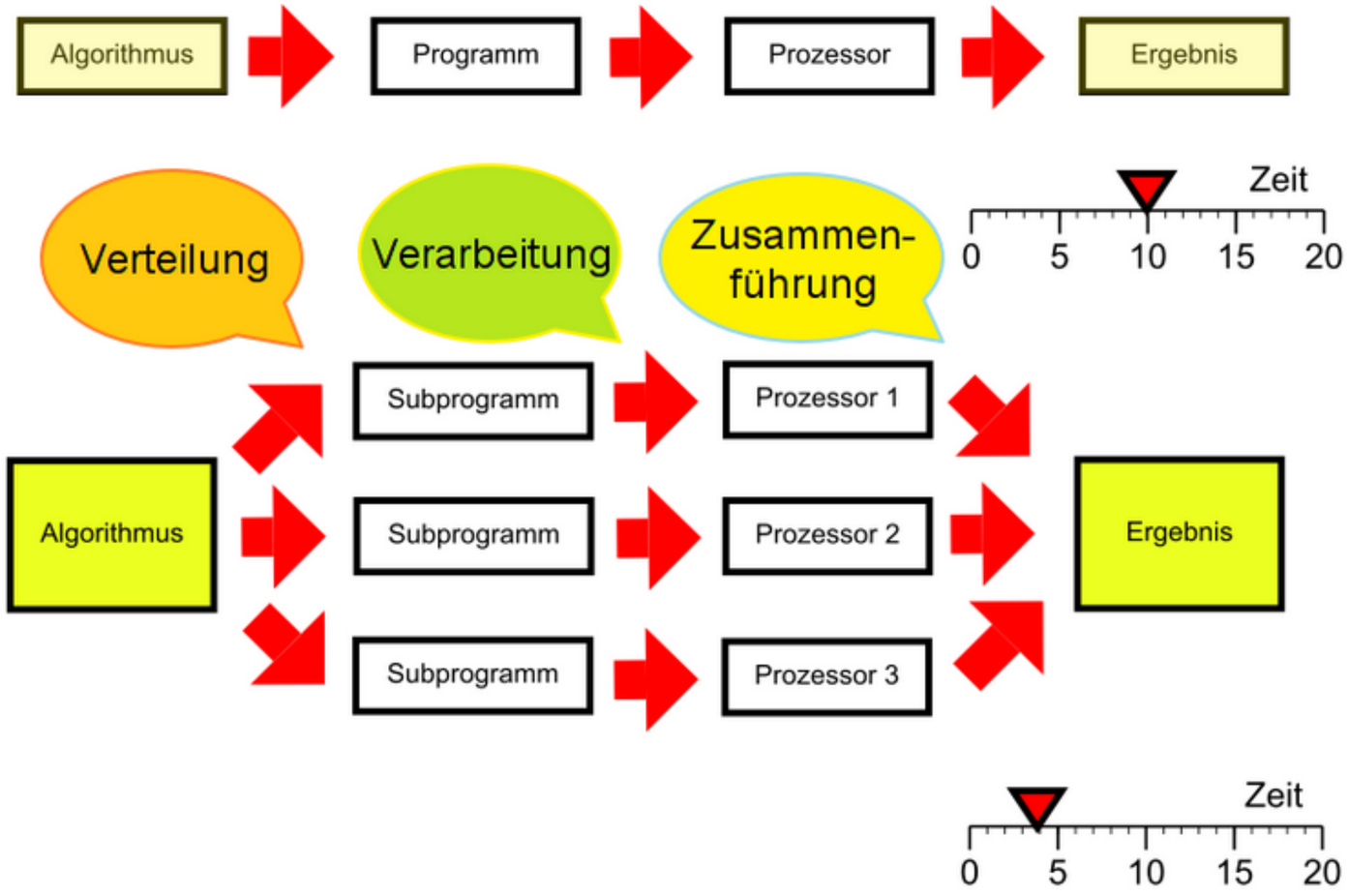


Abb. 7. Vergleich klassiche sequenzielle zur parellelen Datenverarbeitung

DAS PROZESSMODELL

FUNKTIONALE KOMPOSITION

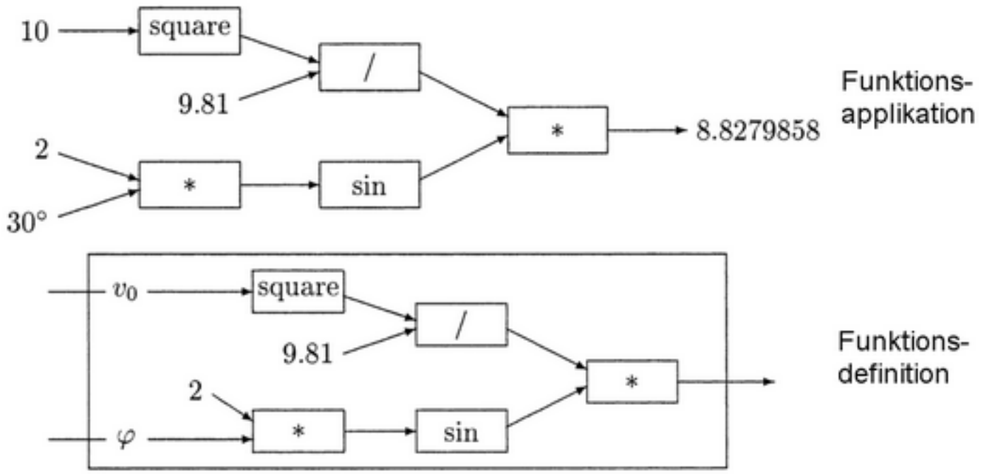
Eine Berechnung setzt sich aus der verschachtelten und verketteten Applikation von Funktionen zusammen, die jeweils aus elementaren Ausdrücken $E=\{\varepsilon_1,\varepsilon_2,..\}$ bestehen → ***Funktionale Komposition*** $F(X)=f_1(f_2(f_3..(X)))$

- Jede Funktion f_i beschreibt einen Teil der Berechnung.
- Ausdrücke bestehen aus:

1. Konstante und variable Werte `1,2.0,'c',"hello",x,y,z`
2. Einfache arithmetische Ausdrücke ε=`x+y`
3. Zusammengesetzte Ausdrücke (arithmetische Komposition) ε=`(x+1)*(y-1)*z`
4. Relationale Ausdrücke `x < 0`
5. Boolesche Ausdrücke `a and b or c`
6. Bedingte Ausdrücke liefern Werte `if x < 0 then x+1 else x-1`
7. Funktionsapplikation `f(x,y,z)`

FUNKTIONALE KOMPOSITION

- Entspricht dem mathematischen Modell was auf Funktionen und Ausdrücken gründet mit den Methoden
  - **Definition** $f(x) = \lambda.x \rightarrow \varepsilon(x)$
  - **Applikation** $f(\varepsilon)$, bei mehreren Funktionsargumenten $f(\varepsilon_1,\varepsilon_2,..)$
  - **Komposition** $f \circ g \circ h \circ .. = f(g(h(..(x)))$
  - **Rekursion** $f(x) = \lambda.x \rightarrow \varepsilon(f,x)$
  - und **Substitution** $a=\varepsilon$
- Zeitliches Modell: unbestimmt ($t \mapsto 0$), *Auswertereihenfolge nicht festgelegt*



SEQUENZIELLE KOMPOSITION

Eine imperative Berechnung setzt sich aus einer Sequenz $\mathbf{I}=(i_1, i_2, ..)$ *von elementaren Anweisungen* $\mathbf{A}=\{a_1,a_2,..\}$ *zusammen →*
Sequenzielle Komposition $i_1; i_2; ...$

- Die Anweisungen werden sequentiell in *exakt* der angegebenen Reihenfolge ausgeführt
- Die Menge der Anweisungen **A** lässt sich in folgende Klassen unterteilen:

1. Arithmetische, relationale, und boolesche **Ausdrücke** (in Verbindung mit 2./3.) `x+1, a*(b-c)*3, x<(a+b), ...`
2. **Datenanweisungen**: Wertzuweisungen an Variablen `x := a+b`
3. **Kontrollanweisungen** wie bedingte Verzweigungen und Sprünge (Schleifen)

```
if a < b then x := 0
label: x:=x+1; if x < 100 then loop label
```

SEQUENZIELLE KOMPOSITION

- Man fasst die Sequenz **I** als ein imperatives **Programm X** == Ausführungsvorschrift zusammen.
- Die **Ausführung** des Programms (statisch) bezeichnet man als **Prozess** (dynamisch)
- Ein Prozess befindet sich aktuell immer in einem **Zustand σ** einer endlichen Menge von Zuständen $\Sigma=\{\sigma_1, \sigma_2, ..\}$.
- Der gesamte Zustandsraum **Σ** des Prozesses setzt sich aus dem
  - **Kontrollzustand** $s \in \mathbf{S}=\{s_1, s_2, ..\}$, und dem
  - **Datenzustand** mit der Menge aller Speicherzustände (Daten) $d \in \mathbf{D}=\{d_1, d_2, ..\}$ zusammen.
- Der Fortschritt eines sequenziellen Programms bedeutet ein Änderung des Kontroll- und Datenzustandes → **Zustandsübergänge**
- Die Zustandsübergänge können durch ein Zustandsübergangsdiagramm dargestellt werden.

ZUSTÄNDE UND ZUSTANDSDIAGRAMM

- Jede Anweisung i_n wird durch wenigstens einen Kontrollzustand $s_m \in \mathbf{S}$ repräsentiert: $i_n \in \mathbf{I} \in \mathbf{A} \Rightarrow s_m \in \mathbf{S}$

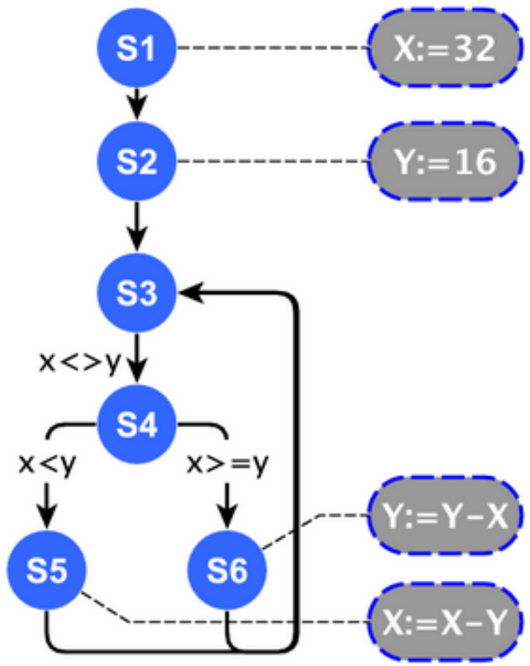
Beispiel Berechnung GGT

A={*assign* $^{:=}$, *loop* ^{WHILE}, *branch* ^{IF}}

S={s1,s2,s3,s4,s5,s6}

```
x^i1→s1 := 32;
y^i2→s2 := 16;
WHILEi^i3→s3 x <> y DO
  IF^i4→s4 x < y THEN
    x^i5→s5 := x - y;
  ELSE
    y^i6→s6 := y - x;
```

Kontrollzustandsdiagramm



SEQUENZIELLER PROZESS

Ausführungszustände

- Ein Prozess *P* besitzt einen "makroskopischen" **Ausführungszustand** *ps*. Die Basismenge der Ausführungszustände **PS** ist ("Milestones"):
  - Start und bereite aber noch nicht rechnend (START)
  - Rechnend (RUN)
  - Terminiert und nicht mehr rechnend (END)
- Dazu kann es eine erweiterte Menge **PS***an Ausführungszuständen geben:
  - Auf ein Ereignis wartend (AWAIT)
  - Blockiert (BLOCKED)
  - Rechenbereit aber nicht rechnend (nach Ereignis) (READY)

$$PS = \{\texttt{START}, \texttt{RUN}, \texttt{END}\}$$
$$PS^* = \{\texttt{AWAIT}, \texttt{BLOCKED}, \texttt{READY}\}$$
$$ps(P) \in PS \cup PS^*$$

SEQUENZIELLER PROZESS

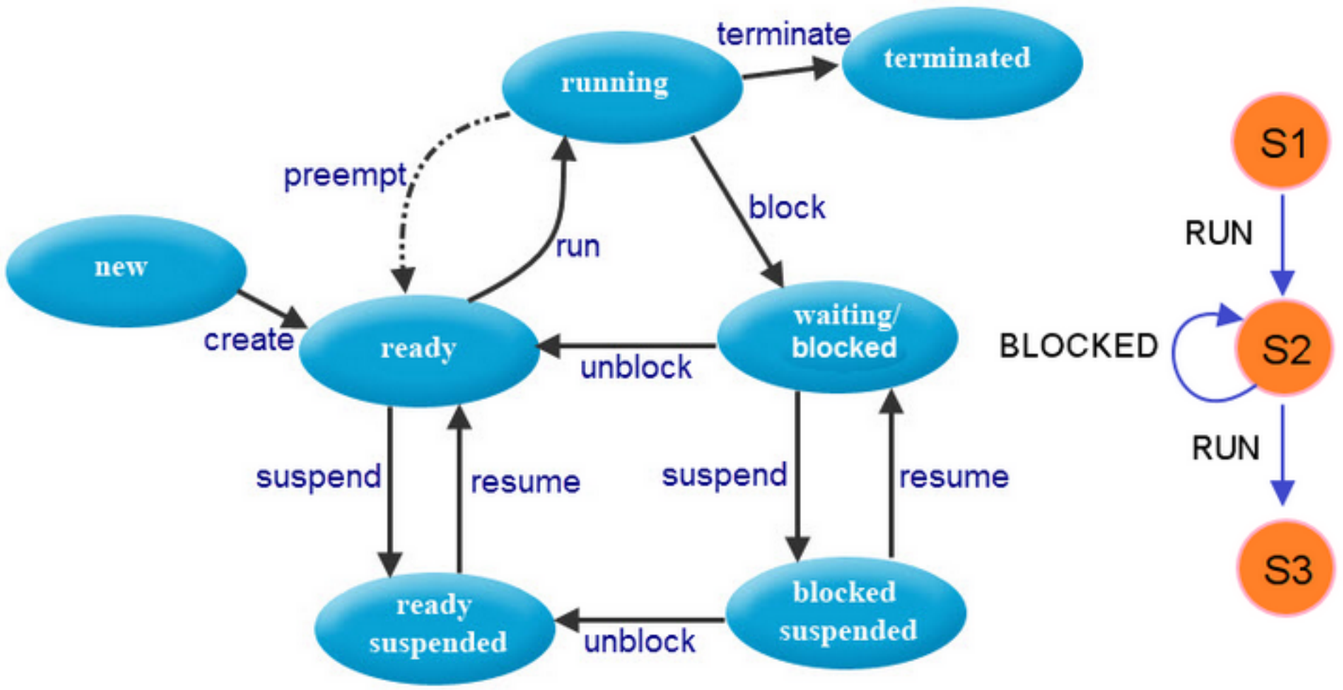
Prozessblockierung

- Das Konzept der Prozessblockierung ist elementar für kommunizierende Prozesse
- Kommunikation: Lesen und Schreiben von Daten, Ein- und Ausgabe, Netzwerknachrichten, usw.
- Befindet sich ein Prozess im Ausführungszustand `BLOCKED` wartet dieser auf ein Ereignis:
  - Daten sind verfügbar
  - Zeitüberschreitung
  - Geteilte Ressource ist frei
  - Synchronisation mit anderen Prozessen
- Bei einem blockierter Prozess schreitet der Kontrollfluss nicht weiter voran
- Die ausführung einer Operation (Instruktion) kann verzögert werden bis das zu erwartende Ereignis eintritt

SEQUENZIELLER PROZESS

- Ein Prozess kann mittels der Operation `suspend` und `resume` blockiert oder wieder lauffähig gemacht werden
- In einem Kontrollflussdiagramm würde es einen Kantenübergang auf den blockierten Zustand geben

Prozesszustandsübergangsdiagramm

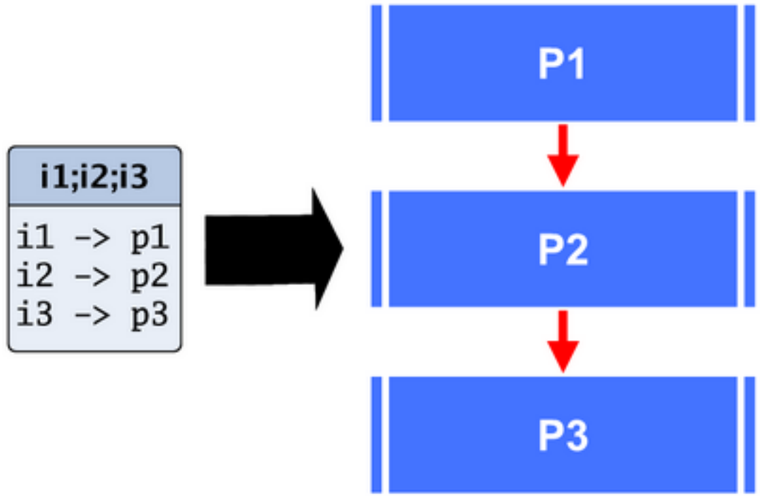


[bitsofcomputer.blogspot.com]

SEQUENZIELLER PROZESS

Instruktionen sind Prozesse

- Jede Instruktion *i* kann als ein elementarer Prozess *P* aufgefasst werden.
- Zu jeder Instruktion gehört eine **Aktion**
- Terminierung eines Prozesses entspricht ausgeführter Instruktion.
- Ein folgender Prozess wird erst dann aktiviert (gestartet) wenn der vorherige terminiert ist → Synchronisation.



SEQUENZIELLER PROZESS

Zeitliches Modell

- Ein Prozess führt die Anweisungen der Reihe nach aus. Das symbolische Semikolon teilt den Anweisungen eigene Ausführungsschritte (Zeitpunkte t_i) zu.

$$
\begin{aligned}
& i_1 : t_1 \mapsto i_2 : t_2 \mapsto i_3 : t_3 \mapsto .. \mapsto i_n : t_n \text{ mit} \\
& t_1 < t_2 < t_3 < .. < t_n \\
& T = \sum_i t_i - t_{i-1}
\end{aligned}
$$

- Die gesamte Ausführungszeit *T(p)* eines sequenziellen Prozesses *p* ist die Summe der Ausführungszeiten der Elementarprozesse

SEQUENZIELLER PROZESS

Prozesskonstruktor

$$P = (P_1 \; ; \; P_2 \; ; \; .. \; ; \; P_n) \;\; \Leftrightarrow$$

```
SEQ
  i1
  i2
  ..
  in
```

PARALLELE KOMPOSITION

- Ausgangspunkt sei das Ausführungsmodell eines sequenziellen Prozesses *P*

Ein sequenzieller Prozess Ps besteht aus einer Sequenz von Anweisungen $I=(i_1; i_2;..)$ *(Berechnungen), die schrittweise und nacheinander ausgeführt werden:* $Ps=P(i_1; i_2; ..; i_n) \Rightarrow p_1 \mapsto p_2 \mapsto .. \mapsto p_n$

- Ein Prozess *P* kann sich in verschiedenen Ausführungszuständen $ps \in$ **RS** $\subset$ **PS** $\cup$ **PS*** befinden:

```
RS={START, RUN, BLOCKED(i),END}
ps(P) ∈ RS
```

- Dabei kann ein Prozess in einer Instruktion *i* **blockieren**, d.h. die folgende Instruktion *i*+1 wird noch nicht ausgeführt und der Prozess verbleibt aktionslos in der aktuellen Instruktion

PARALLELE KOMPOSITION

Prozesse als steuerbare Objekte

- Ein Prozess *P* kann mittels Operationen gestartet und gestoppt werden:

Definition 1.

```
op start(P):   ps(P)=START|END ↦ ps(P)=RUN
op stop(P):    ps(P)=RUN|BLOCKED|START ↦ ps(P)=END
op suspend(P): ps(P)=RUN ↦ ps(P)=BLOCKED
op resume(P):  ps(P)=BLOCKED ↦ ps(P)=RUN
```

Parallele Prozesse

Ein Datenverarbeitungssystem als paralleler Prozess kann aus einer Vielzahl nebenläufig ausgeführter Prozesse $P=\{P_1, P_2, P_3,\ldots\}$ *bestehen →*
Parallele Komposition $P=P_1 \parallel P_2 \parallel P_3 \parallel ..$

PARALLELER PROZESS

Prozesskonstruktor

$$P = (P_1 \parallel P_2 \parallel .. \parallel P_n) \quad \Leftrightarrow$$

```
PAR
  i1
  i2
  ..
  in
```

Ausführungsreihenfolge

- Achtung! Wenn keine Kommunikation (Synchronisation) zwischen den Prozessen stattfindet ist die Ausführung und das Endergebnis (der Berechnung) gleich der sequenziellen Ausführung der Prozesse:

$$P_1 \parallel P_2 \parallel .. \parallel P_n \Rightarrow P_1; P_2; ..; P_n$$
$$P_1 \parallel P_2 \Rightarrow P_1; P_2 \Rightarrow P_2; P_1!$$

PARALLELER PROZESS

Prozesserzeugung und Synchronisation

- Ein neuer paralleler Prozess kann durch den entsprechenden Prozess-konstruktor erzeugt werden
- Geschieht dies innerhalb eines Prozesses P_i so wird dessen Prozessausführung blockiert *bis* alle Teilprozesses des parallelen Prozessen terminiert sind!

Forking

- Ein neuer Prozess P_j kann mittels Forking erzeugt werden (Aufspaltung des Kontrollflusses, so auch von einem anderen Prozess P_i) ohne dass ein ausführender Prozess auf die Terminierung des Parallelprozesses warten muss:

$$P_1; /\!/ P_2; P_3 = P_1; P_3 \parallel P_2$$
$$/\!/ P_1; /\!/ P_2; P_3 = P_1 \parallel P_2 \parallel P_3$$

PARALLELER PROZESS

Prozesskonstruktor

$$P = P_1; /\!/ P_2 \quad \Leftrightarrow$$

```
FORK
  i1
  i2
  ..
  in
```

PARALLELER PROZESS

Zeitliches Modell

$$P = P_1 \parallel P_2 \parallel .. \parallel P_n$$
$$P_1 = (i_{1,1} : t_{1,1}; i_{1,2}, : t_{1,2}; ..) \cdots P_n = (i_{n,1} : t_{n,1}; i_{n,2} : t_{n,1}; ..), \text{ mit}$$
$$t_{i,1} < t_{i,2} < .. < t_{i,m} \text{ fü r jeden Prozess mit } m \text{ Instr. und}$$
$$t(P) = [t_1, t_2] = [t_{1,1}, t_{1,q}] \cup [t_{2,1}, t_{2,r}] \cup .. \cup [t_{n,1}, t_{n,s}]$$
$$t_1 = \min\{t_{1,1}, t_{2,1}, .., t_{n,1}\}$$
$$t_2 = \max\{t_{1,q}, t_{2,r}, .., t_{n,s}\}$$
$$T = t_2 - t_1$$

PROZESSFLUSS

- Der Prozessfluss in einem parallelen System kann durch Übergangs- und Terminierungsdiagramme dargestellt werden. (wie bereits im seq. Fall gezeigt wurde).
- Eingehende Kanten (Pfeile) des Graphen beschreiben den Start eines Prozesses, ausgehende Kanten die Terminierung.

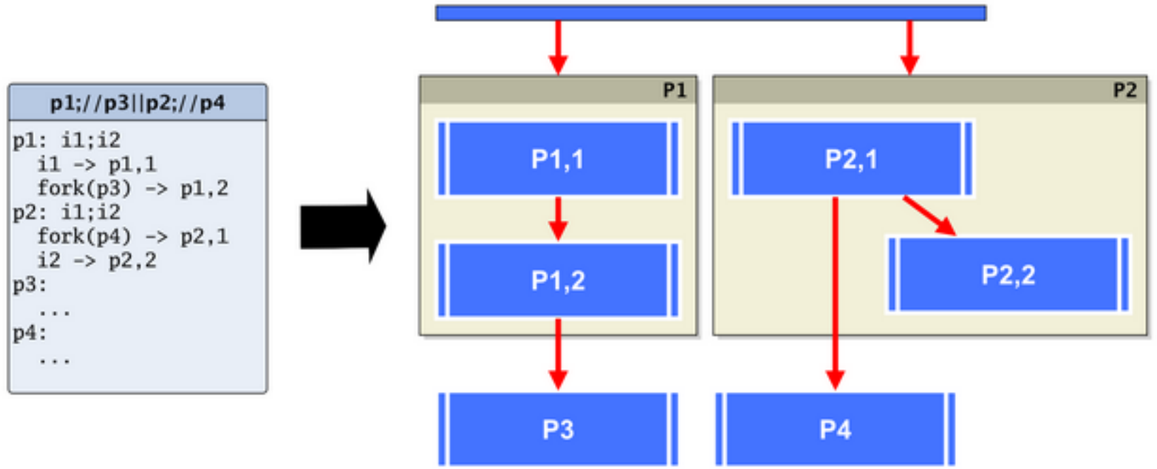


Abb. 8. Beispiel eines Prozessflussdiagramms für vier Prozesse. P_1 startet P_3, und P_2 spaltet P_4 ab. (mit $P_{_,i}$ = *Instruktion*(P,i))

PROZESSFLUSS

Beispiel

```
Seq([
  function (id) { log('P1: I am '+id) },
  function (id) { log('P2: I am '+id) },
])
Par([
  function (id) { log('P3: I am '+id) },
  function (id) { log('P4: I am '+id) },
])
log('after')
```

JS Processes

CLEAR GET RESET Par

METAPROZESSE

- Die Menge aller Prozesse $\{P_1, P_2, ..\}$ eines Systems bilden ein Meta-Prozess P. Dieser terminiert wenn alle enthaltenen (Sub-) Prozesse terminiert sind.

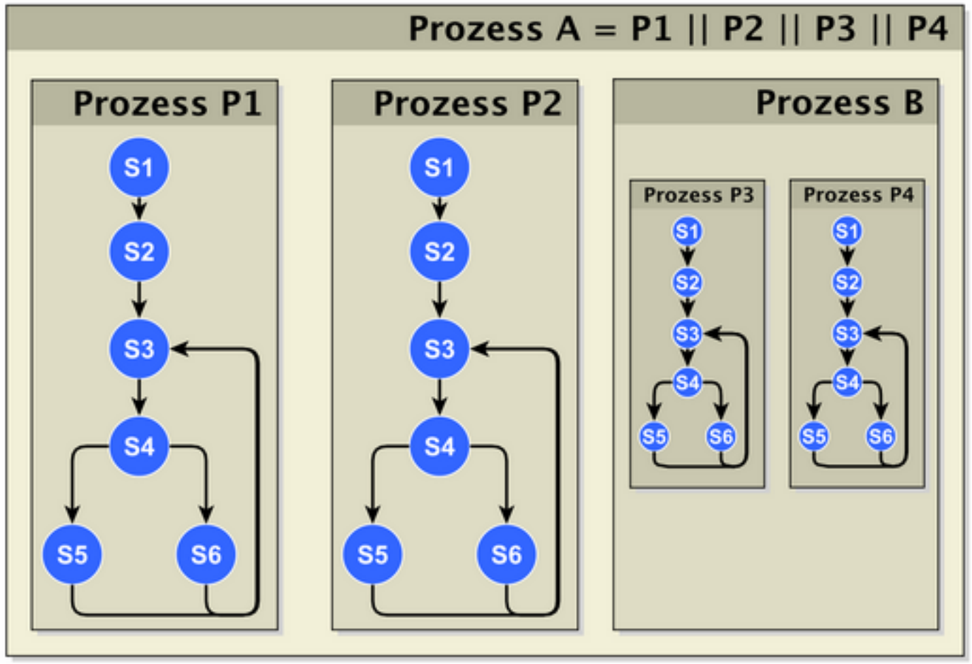


Abb. 9. Vier parallele Prozesse P_1, P_2, P_3, und P_4. Die Prozesse bilden hier eine Prozesshierarchie, d.h z.B. P_3 und P_4 gehören zum Meta-Prozess P_B.

KOMMUNIZIERENDE SEQUENZIELLE PROZESSE

Definition 2.
Prozesskommunikation bedeutet Synchronisation zwischen Prozessen

Prozess Synchronisation tritt auf, wenn der Berechnungsfortschritt eines oder mehrerer Prozesse von dem Verhalten der anderen Prozesse abhängt.

- Zwei Arten der Prozess Interaktion benötigen Synchronisation:
  - **Wettbewerb** (z.B. um geteilte Ressourcen)
  - **Kooperation**
- Parallele Datenverarbeitung bedeutet:
  - **Distribution**: Partitionierung und Verteilung von Eingabedaten auf verschiedene Prozesse, die jeweils ein Teilergebnis berechenen → **Kooperation**
  - **Zusammenführung**: Zusammenführen von Teilergebnissen → **Kooperation**

KOMMUNIZIERENDE SEQUENZIELLE PROZESSE

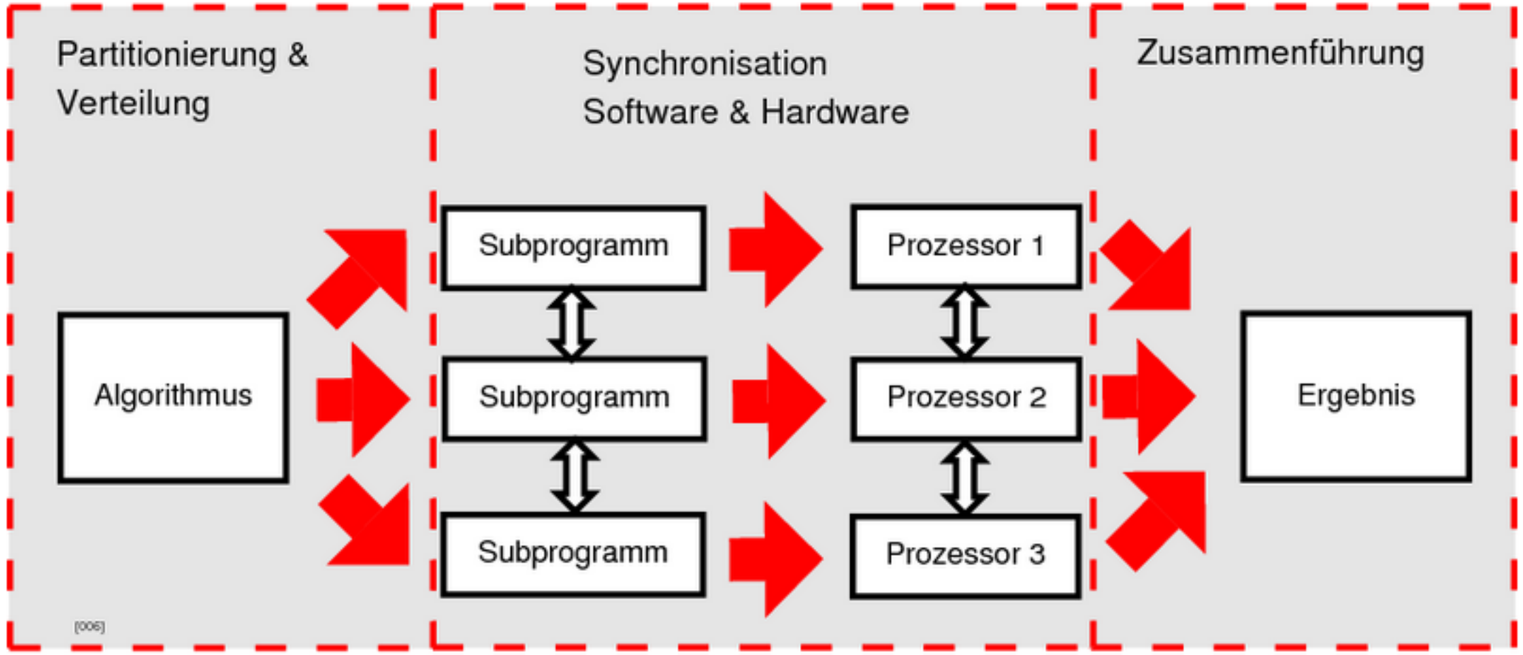


Abb. 10. Kommunikation von parallelen Prozessen in der parallelen Datenverarbeitung, z.B. um Rechenergebnisse auszutauschen oder während der Verteilungs- und Zusammenführungsphase

KOMMUNIZIERENDE SEQUENZIELLE PROZESSE

Paralleler Datenzugriff

- Daten werden gespeichert und befinden sich in Speicherelementen: einzelne Register oder adressierbare RAM Speicher.
- Mehrere Prozesse müssen je nach Datenabhängigkeit parallel auf geteilte Ressource Speicher zugreifen, um
  - » Eingabedaten zu lesen und
  - » Zwischenergebnisse auszutauschen und
  - » Ausgabedaten schreiben zu können.

⇔ **Wettbewerb**

- Nebenläufiger Zugriff auf geteilte Ressourcen ist ein Konflikt der aufgelöst werden muss → Mutual Exclusion Problem

GETEILTE RESSOURCEN UND KOMMUNIKATION

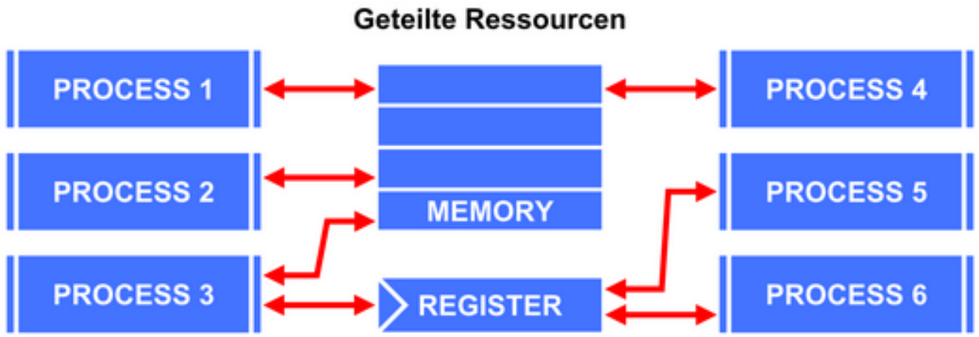


Abb. 11. Paralleler Zugriff auf Speicherresourcen von Prozessen

Queue

- Ressourcen-Konflikt und Synchronisation kann mit Daten-Queues/Channels aufgelöst werden.
- Eine **Queue** hat im wesentlichen zwei Operationen die synchronisierten Zugriff ermöglicht (FIFO Speicher):

Definition 3.

```
op write(Q,x): Q[y,z,...],x → Q[x,y,z,...]
op read(Q):    Q[x,y,z] → Q[x,y],z
```

GETEILTE RESSOURCEN UND KOMMUNIKATION

- Eine Queue hat einen Prozess PW der Daten in die Queue schreiben kann und einen Prozess PR der Daten aus der Queue lesen kann.

Synchronisation

```
Queue Q=[]      ⇔ state=EMPTY: Lesender Prozess wird blockiert bis Q ≠ []
Queue Q=[x,y,z] ⇔ state=FULL:  Schreibender Prozess wird blockiert bis state(Q) ≠ FULL
```

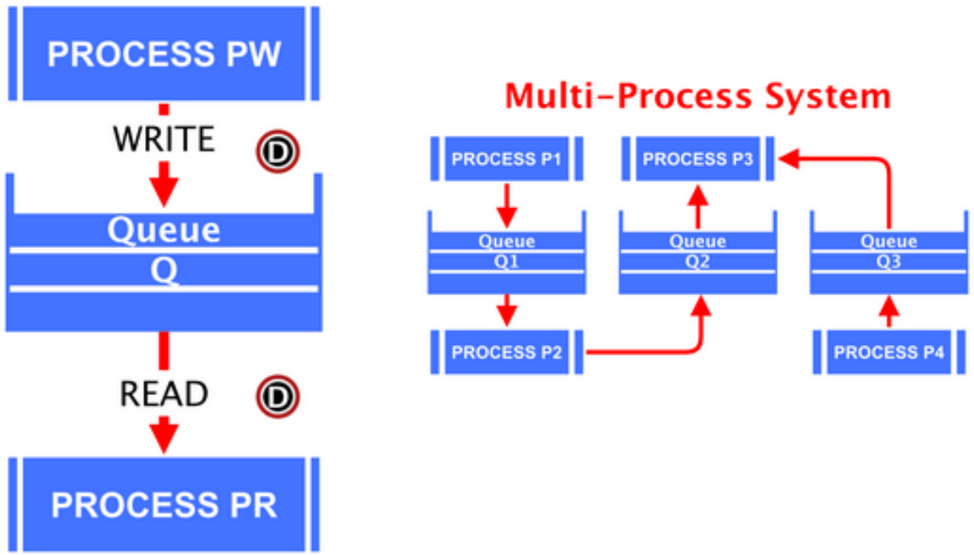


Abb. 12. Verbindung zweier Prozesse durch eine Queue

GETEILTE RESSOURCEN UND KOMMUNIKATION

Channel

- Ähnlich wie eine Queue eventuell mit Einschränkungen:
  - Puffergröße: 0 (!!) → *Kein Wettbewerb*! / 1
  - Nur zwei Prozesse zugelassen: Ein *Leser* und ein *Schreiber*
  - Rendezvous Synchronisation: Schreibender Prozess wird blockiert bis lesender Prozess *read* Operation ausführt

Beispiel

```
Par([
  function () {
    var x=ch.read();
    log('hello '+x);
  },
  function () {
    ch.write('world');
  }
],{
  ch:Channel()
})
```

GETEILTE RESSOURCEN UND KOMMUNIKATION

JS Processes

CLEAR GET RESET Chann

PROZESSMEHRFACHAUSWAHL

- Ein Prozess kann in seinem Kontrollfluss nur *eine* blockierende Anweisung gleichzeitig aufrufen, d.h. in unserem Prozessmodell nur einen folgenden elementaren Prozess aktivieren.
- Häufig ist es aber notwendig dass der Prozessfluss in mehrere Elementarprozesse mit einer blockierenden Operation aufspaltet (z.B. Lesen eines Channels) von dem dann nur ein Prozess zur Ausführung kommt (der erste der bereit wurde) → *Konditionaler Fork*
- Der aufgespaltete konditonale Prozessfluss wird dann wieder zusammengeführt (*join*) → Klassische IO Selektion

Auswahlprozesskonstruktor:

$$P = (p_{1,cond} \rightarrow p_{1,do} | p_{2,cond} \rightarrow p_{2,do} | ..)$$

```
ALT
  i1,cond?i1,do
  i1,cond?i1,do
  ..
```

PROZESSMEHRFACHAUSWAHL

Beispiel ALT

```
Par([
  function () {
    var x;
    Alt([
      [function () { x=ch1.read() },  Konditonaler Prozess 1
       function () { log('Got channel 1') }],  Ereignis Prozess 1
      [function () { x=ch2.read() },  Konditonaler Prozess 2
       function () { log('Got channel 2') }],  Ereignis Prozess 2
      [function () { x=ch3.read() },  Konditonaler Prozess 3
       function () { log('Got channel 3') }],  Ereignis Prozess 3
    ]);
    log('hello '+x);
  },
  function () {
    ch1.write('world');
  }
],{
  ch1:Channel(),
  ch2:Channel(),
  ch3:Channel()
})
```

PROZESSMEHRFACHAUSWAHL

JS Processes

CLEAR GET RESET Chann

CSP MODELL UND PROZESSALGEBRA

- Prozessalgebra nach dem “Communicating Sequential Processes” (CSP) Modell von Hoare:
- Beschreibung des *Verhaltens* von Prozessen als Reaktion auf *Ereignisse* → Synchronisierter *Prozessfluss*
- *Kommunikation* sind Ereignisse
- Algebraisch wird die Entwicklung von Prozessen durch Ereignisse *x*,*y*, .. beschrieben

Definition 4.

Ereignis : x

Prozess : P

Ein Prozess der nach einem Ereignis x sich wie Prozess P verhält: $x \rightarrow P$

Folge von Ereignissen: $x \rightarrow (y \rightarrow P)$

Parallele Ausführung: $P \parallel Q$

Alphabet eines Prozesses: $\alpha P = x_1, y_2, ..$

CSP MODELL UND PROZESSALGEBRA

- Ein *Alphabet* eines Prozesses beschreibt die möglichen Ereignisse die als Vorbedingung eines Prozesses auftreten können: $\alpha P = x_1, y_2, ..$
- Rekursive Definition von Prozessen: $\mu X : A \bullet F(X)$
  Mit A: Alphabet des Prozesses (Ereignismenge)

Beispiel

- Ein Taktprozess:

$$A = \alpha CLOCK = \{tick\}$$
$$CLOCK = \mu X : \{tick\} \bullet (tick \rightarrow X)$$

- Ein Snackautomat:

$$A = \alpha SNACK = \{coin, choco\}$$
$$SNACK = \mu X : \{coin, choco\} \bullet (coin \rightarrow (choc \rightarrow X))$$

CSP MODELL UND PROZESSALGEBRA

- Deterministische oder nichtdeterministische Auswahl von Ereignissen und Prozessentwicklungen sind typisch für parallele Systeme!

Definition 5.
Auswahl: $(x \rightarrow P | y \rightarrow Q)$

→ D.h. entweder tritt Ereignis x ein und der Auswahlprozess verhält sich wie P, oder es tritt das Ereignis y ein und der Prozess verhält sich wie Q.

Es gilt: $\alpha(x \rightarrow P | y \rightarrow Q) = \alpha P = \alpha Q$

→ D.h. die Ereignismenge ist immer x, y.

CSP MODELL UND PROZESSALGEBRA

Beispiel Snackautomat mit Choco und Kaffee

$$VMCT = \mu X \bullet coin \rightarrow (choc \rightarrow X | coffee \rightarrow X)$$

Beispiel Alt Prozessauswahlkonstruktor

$$ALT = \mu X : \{in_1.x, in_2.x, \cdots, out_1.x, out_2.x, \cdots\} \bullet ($$
$$in_1.x \rightarrow (P_1 \rightarrow (out_1.x \rightarrow X))|$$
$$in_2.x \rightarrow (P_2 \rightarrow (out_2.x \rightarrow X))| \cdots)$$

CSP MODELL UND PROZESSALGEBRA

Spur

- Eine Spur (Trace) des Verhaltens eines Prozesses beschreibt die zeitliche Abfolge von Ereignissen in die ein Prozess verwickelt war.
- Spruen werden von Beobachtern aufgenommen (Monitor).

Definition 6.
Spur: $< x, y, .. >$

Leere Spur: $<>$

Spurbindung: $< a, b > \hat{} < c, d > \rightarrow < a, b, c, d >$

Beispiele

$$< coin, choc, coin, choc, coin, coin, coffee >$$
$$< tick, tick, tick >$$

CSP MODELL UND PROZESSALGEBRA

Spurenmenge

- Es gibt eine *Vielzahl* von möglichen Spuren in einem Einzel- und Multiprozesssystem!
- Jede Spur kann zudem in Teilspuren zerlegt werden beginned mit der leeren Spur.
- Spurenmenge ⇔ Zustandsraumdiagramm!

Beispiele

$$
\begin{aligned}
&traces(STOP) = \{<>\} \\
&traces(coin \rightarrow STOP) = \{<>, < coin >\} \\
&traces(\mu X \bullet tick \rightarrow X) = \{<>, < tick >, < tick, tick >, ..\} = \{tick\}* \\
&traces(\mu X \bullet coin \rightarrow choc \rightarrow X) = \{<>, < coin >, < coin, choc >, ..\} \\
&= \{s | \exists n \bullet s \leqslant < coin, choc >^n\}
\end{aligned}
$$

CSP MODELL UND PROZESSALGEBRA

Kommunikation

Definition 7.

$\alpha c(P) = \{v | c.v \in \alpha P\}$: Menge aller Nachrichten eines Prozeses *P*

$(c!v \rightarrow P) = (c.v \rightarrow P)$: Ausgabe eines Wertes über den Kanal *c* (Schreiben)

$(c?x \rightarrow P)$: Eingabe eines Wertes über einen Kanal *c* (Lesen)

ZUSTANDS-RAUM DIAGRAMME

Für die Visualisierung und Analyse von nebenläufigen Prozessen und Aktionen

NEBENLÄUFIGE PROGRAMMIERUNG: ZUSTANDS-RAUM DIAGRAMME

Zustands-Raum Diagramme *beschreiben die möglichen Zustands-Entwicklungen - das Verhalten - von parallelen Programmen.*

- Computer sind endliche Zustandsautomaten. Der Zustandsübergang wird durch die Programminstruktionen hervorgerufen.
- Der Zustand **S** eines parallelen Programms welches aus N Prozessen P_i besteht setzt sich zusammen aus folgenden Tupeln:
  - » Globale Variablen des Programms:
  - » Lokale Variablen und Instruktionszeiger von Prozess 1:
  - » Lokale Variablen und Instruktionszeiger von Prozess 2: usw.
- Die Berechnung ändert globale und lokale Variablen (Datenfluss) sowie Instruktionszeiger (Kontrollfluss) und führt zu einer Zustandsänderung $comp: s_i \rightarrow s_j$.

NEBENLÄUFIGE PROGRAMMIERUNG: ZUSTANDS-RAUM DIAGRAMME

- Wesentliche Zustandsänderung ist die Änderung von lokalen und globalen Speicher.
- Variablen sind Bestandteil von Ausdrücken und erlauben zwei Operationen: {READ,WRITE}

```
var x: read(x) ⇔ RHS value(x) → ε(x)
       write(x,v) ⇔ LHS reference(x) → x := v
```

Beschreibung eines parallelen Programms auf Programmierebene

- Der Zugriff auf globale und somit geteilte Variablen muss atomar sein, d.h. immer nur ein Prozess kann den Wert einer Variable lesen oder einen neuen Wert schreiben.
- Der parallele Zugriff auf eine geteilte Ressource muss aufgelöst werden (*Konflikt*): i.A. mutualer Ausschluss durch *Sequenzialisierung* der parallelen Zugriffe!

NEBENLÄUFIGE PROGRAMMIERUNG: ZUSTANDS-RAUM DIAGRAMME

- Beispiel

```
write(X,v1) || write (X,v2) || read(X,x3) →
write(X,v1);read(X,x3);write(X,v2)
```

- Algebraisch ausgedrückt ergibt sich die Transformation:

$$\frac{\begin{array}{c}(write(x,v_1) \rightarrow p_1) \parallel \\ (write(x,v_2) \rightarrow p_2)\end{array}}{\begin{array}{l}(write(x,v_1) \rightarrow (write(x,v_2) \rightarrow (p_1 \parallel p_2)))| \\ (write(x,v_2) \rightarrow (write(x,v_1) \rightarrow (p_1 \parallel p_2)))| \\ ..\end{array}}$$

NEBENLÄUFIGE PROGRAMMIERUNG: ZUSTANDS-RAUM DIAGRAMME

- Instruktionen von Prozessen werden sequenziell ausgeführt. Instruktionen können auf Prozess-lokale und Programm-globale Variablen zugreifen.
- Definition eines parallelen Programms: globale Variablen (*V*) und Prozesse mit lokalen Variablen (*v*)

```
var V1,V2,...
process p1(a1,a2,...)
  var v1,v2,....
  <Instruktion1 i1>
  <Instruktion2 i2>
  Vi := ε(ai,vi,Vi)
  vi := ε(ai,vi,Vi)
end
```

```
process p2(a1,a2,...)
  var v1,v2,....
  <Instruktion1 i1>
  <Instruktion1 i2>
  Vi := ε(ai,vi,Vi)
  vi := ε(ai,vi,Vi)
end
```

NEBENLÄUFIGE PROGRAMMIERUNG: ZUSTANDS-RAUM DIAGRAMME

- Bei einem sequenziellen Programm ist das Ergebnis einer Berechnung (d.h. die Werte aller Variablen) deterministisch allein durch die Anweisungssequenz und die Eingabedaten gegeben, und kann beliebig oft wiederholt werden - immer mit dem gleichen Ergebnis → Reihenfolge aller Anweisungen der Berechnung vorgegeben und fest
- Bei einem parallelen Programm können mehrere Anweisungen verschiedener Prozesse gleichzeitig ausgeführt werden bzw. konkurrieren.
- Die Reihenfolge parallel ausgeführter Anweisungen von einzelnen Prozessen kann hingegen undeterministisch = zufällig sein!!
- Jeder der Prozesse kann als nächster den Zustand des Programms ändern.
- Ein Zustands-Raum Diagramm beschreibt die Änderung des Programmzustandes als sequenzielle Auswertung aller möglichen Prozessaktivitäten.

NEBENLÄUFIGE PROGRAMMIERUNG: ZUSTANDS-RAUM DIAGRAMME

- Das Diagramm ist ein gerichteter Graph, dessen Knoten den aktuellen Programmzustand $s \in \mathbf{S}$ beschreiben, und dessen Kanten die möglichen Zustandsübergänge beschreiben.
- Es gibt einen ausgewiesen Startzustand (Initialisierung) und einen oder mehrere Endzustände.
- Gibt es mehrere Endzustände liegt wohl möglich ein Entwurfsfehler vor, da das Programm unterschiedliche Endergebnisse liefern kann.

Algorithmus: Entwicklung des Zustands-Raum Diagramms

- Parallelen Programms welches aus N Prozessen $\mathbf{P}=\{p_1,p_2,..,p_N\}$ besteht.

1. Initialisiere den Startzustand $s_x=s_0$ und erzeuge Wurzel-Knoten s_0 im Diagramm.
2. Aktueller Zustand: s_x. Setze $\mathbf{P}^*:=\mathbf{P}$ mit $\mathbf{P}=\{p_1,p_2,..,p_N\}$ als Prozessliste und $\mathbf{S}^*=\{\}$

NEBENLÄUFIGE PROGRAMMIERUNG: ZUSTANDS-RAUM DIAGRAMME

3. Wähle (entferne) einen beliebigen Prozess p_x aus der Prozessliste $\mathbf{P}^*$ und setze $\mathbf{P}^*:=\{p_n | p_n \in \mathbf{P}^* \wedge p_n \notin p_x\}$
4. Evaluiere die nächste Instruktion $i_{x,n}$ von p_x und berechne die Wirkung auf lokale und globale Variablen sowie den Instruktionszeiger $I_{x,n}$.
5. Erzeuge einen neuen Zustandsknoten $s_{j/=x}$, füge ihn zur Liste $\mathbf{S}^*:=\mathbf{S}^* \cup \{s_j\}$ hinzu und verbinde ihn mittels einer Kante $t_{x\rightarrow j}$ mit dem aktuellen Ausgangszustand s_x
6. Wiederhole Schritt 3 bis 5 für alle anderen verbleibenden Prozesse $p \in \mathbf{P}$ bis $\mathbf{P}^*=\{\}$
7. Setze $\mathbf{S}^{**}:=\mathbf{S}^*$. Für jeden Knoten $s_x \in \mathbf{S}^{**}$ wiederhole die Schritte 2 bis 6. Iteriere Schritt 7 bis alle Prozesse terminiert sind oder keine Zustandsänderung mehr auftritt.

NEBENLÄUFIGE PROGRAMMIERUNG: ZUSTANDS-RAUM DIAGRAMME

Beispiel

```
// Sequenzielles Programm
for i := 1 to 3 do i1
```

```
// Paralleles Programm
process p1
   for i := 1 to 3 do i1
end
process p2
   for j := 1 to 3 do i2
end
```

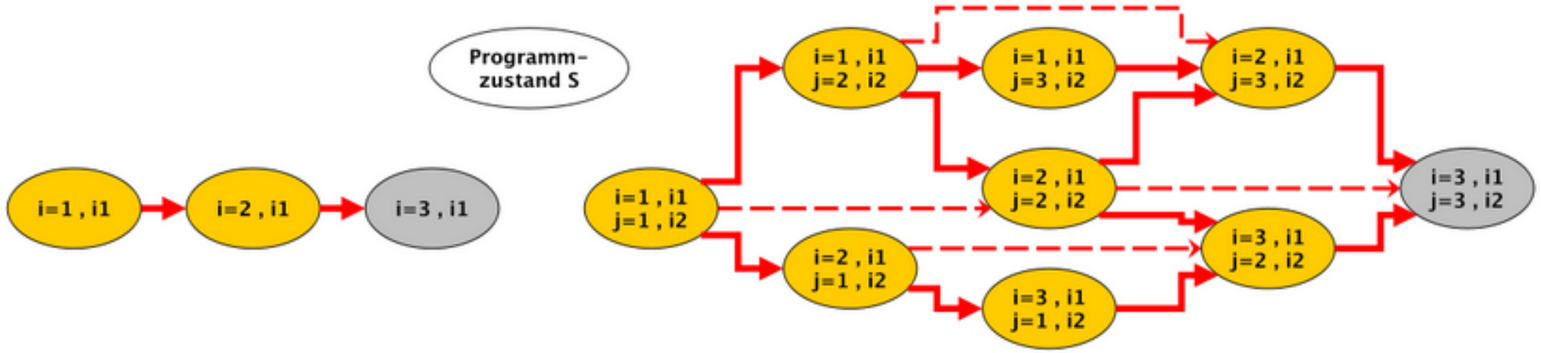


Abb. 13. Beispiel und mögliche Entwicklung des Programmzustands (lokale Var. i,j)

GLOBALE RESSOURCEN UND SYNCHRONISATION

Globale Ressourcen

- In dem bisherigen Programmmodell gibt es nur globale Variablen als globale geteilte Ressourcen, die konkurrierend von Prozessen gelesen und beschrieben werden können.
- Die globalen Variablen dienen dem Datenaustausch = Kommunikation zwischen den Prozessen (Shared Memory Model!).

Atomare Aktionen

- *Atomare Operationen*, Schreibweise $|A|$ werden in einem Schritt ausgeführt und können nicht durch andere Prozesse unterbrochen werden (keine Interferenz).
- Einzelne Lese- und Schreibe-Operationen mit globalen Variablen sind atomar. Nebenläufigkeit wird durch Sequenzialisierung aufgelöst.

```
t1: |X := v|
```

GLOBALE RESSOURCEN UND SYNCHRONISATION

Nicht atomare Aktionen und Synchronisation

Zusammengesetzte Ausdrücke können aus mehreren Berechnungsschritten bestehen, so ist z. B. `X := X + 1` als Ganzes nicht mehr atomar!

```
t1: |temp=X|
t2: |X:=temp+1|
```

Beispiel einer unzureichend geschützten nicht atomaren Aktion

```
var X := 1
// INCR(X)        // DECR(X)
proc p1           proc p2
  var t:=0          var t:=0
  t := X            t := X
  X := t + 1        X := t - 1
end               end
```

GLOBALE RESSOURCEN UND SYNCHRONISATION

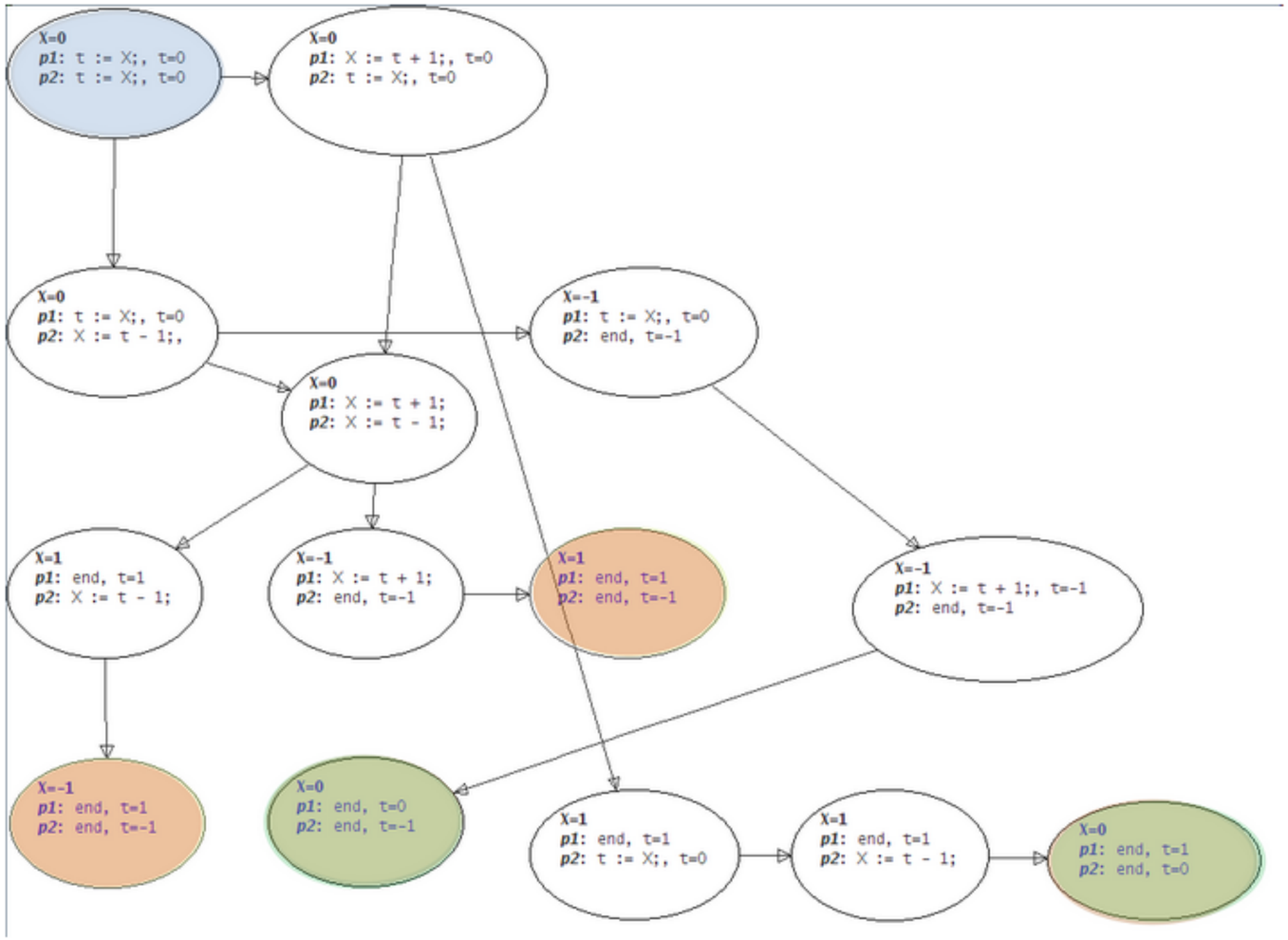


Abb. 14. Zustands-Raum Diagramm mit unterschiedlichen Endergebnissen (X=0,-1,1)

GLOBALE RESSOURCEN UND SYNCHRONISATION

Nicht atomare Aktionen (Sequenz von Aktionen) mit globalen Ressourcen müssen durch Mutuale Ausschlusssynchronisation geschützt werden → ***Schutz von kritischen Codebereichen ⇒ Datenkonsistenz!***

Semaphore

- Eine Semaphore S ist ein Zähler $s>0$ der niemals negative Werte annehmen kann.
- Es gibt zwei wesentliche atomare Operationen die von einer Menge von Prozessen $P=\{p_1,p_2,\ldots\}$ auf einer Semaphore S angewendet werden können:

Definition 8.

```
op down(S) | p:
  |IF s > 0 THEN s := s - 1 else WAITERS+(S,p),BLOCK(p)|
op up(S) | p:
  |IF s = 0 AND WAITERS(S) ≠ [] THEN pi=WAITERS-(S),UNBLOCK(pi)
   ELSE s := s + 1|
```

GLOBALE RESSOURCEN UND SYNCHRONISATION

- Eine Semaphore mit einem Startwert s=1 entspricht einer Mutex (binäre Semaphore). Ein *down(S)* leitet einen kritischen Bereich in einer Sequenz $i_1, i_2, i_3, \ldots$ ein, und ein *up(S)* gibt ihn wieder frei:

```
... down(S) |i1,i2,i3,…| up(S) ...
```

Beispiel einer mit einer Semaphore geschützten nicht atomaren Aktion

```
var X := 1
sema S := 1
// INCR(X)      // DECR(X)
proc p1         proc p2
  var t:=0        var t:=0
  down(S)         down(S)
  t := X          t := X
  X := t + 1      X := t - 1
  up(S)           up(S)
end             end
```

GLOBALE RESSOURCEN UND SYNCHRONISATION

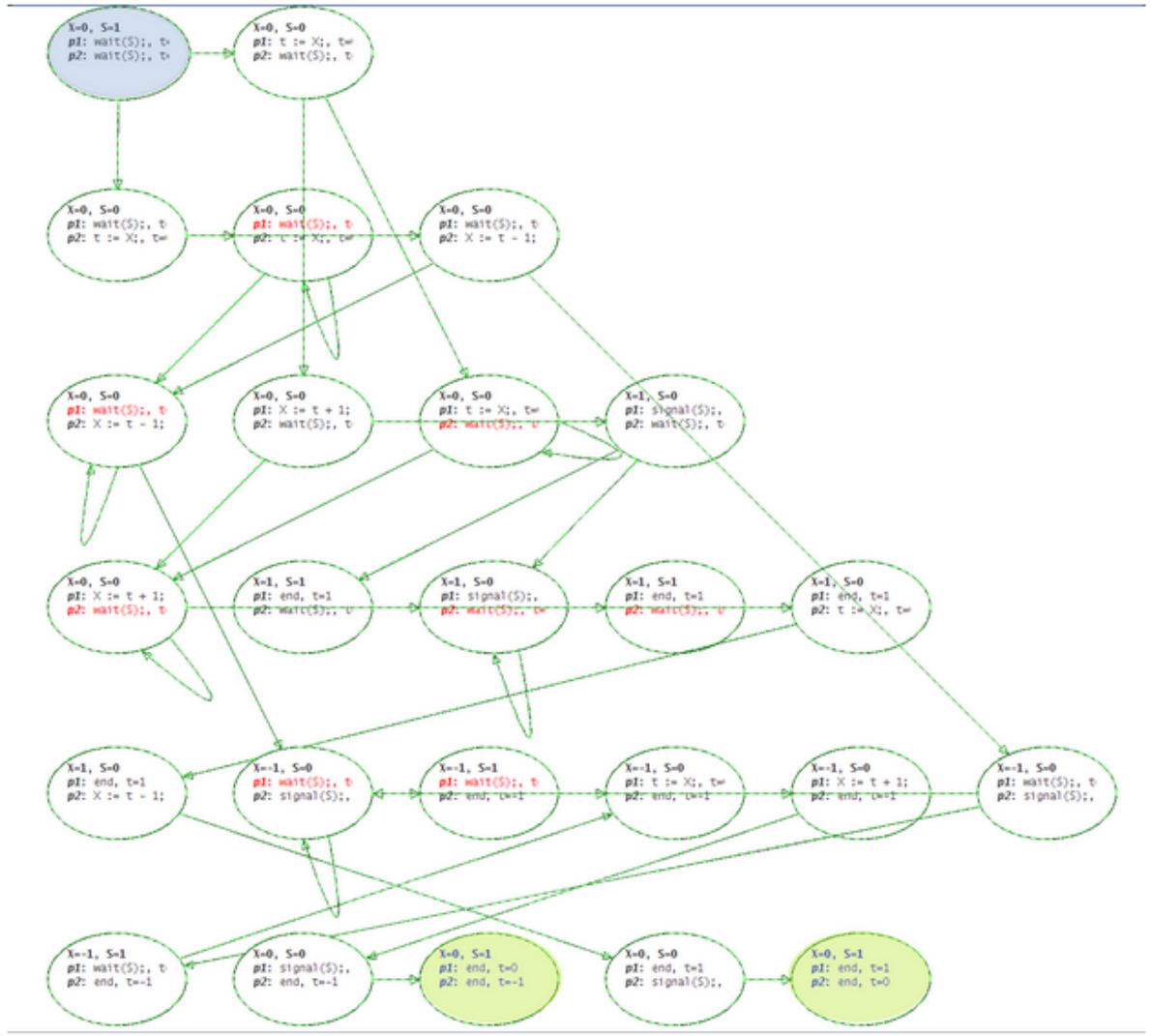


Abb. 15. Zustands-Raum Diagramm mit einem Endergebnis (X=1)

PETRI NETZE

Petri-Netze für die Modellierung, Analyse, und Implementierung von Parallelen Systemen und Digitallogik

PETRI-NETZE: EINFÜHRUNG UND GRUNDLAGEN

Petri-Netze eignen sich für die Darstellung paralleler Prozesse und deren Analyse des dynamischen Verhaltens

Petri-Netz (State-Transition)

- Ein Petri-Netz ist ein Quadrupel PN = $<S,T,E,A>$ mit
  - S: Menge von Stellen (oder auch Plätzen sowie Zuständen/States)
  - T: Menge von Übergängen (oder auch Transitions)
  - E: Menge der Eingabekanten, gegeben durch die Relation $E \subseteq S \times T$
  - A: Menge der Ausgabekanten, gegeben durch die Relation $A \subseteq T \times S$

Markierungen M

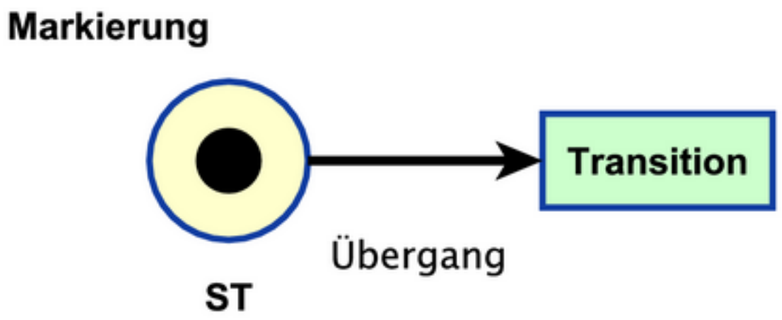
- Das **dynamische Verhalten** der Petri-Netze wird mittels von **Markierungen** beschrieben (Tokens).
- Es gibt eine **Markierungsfunktion** $M: S \rightarrow N$ mit $N=\{0,1,2..\}$, die die Anzahl der Markierungen $M(s)$, $s \in S$ einer Stelle s angibt.

PETRI-NETZE: EINFÜHRUNG UND GRUNDLAGEN

- Eine Stelle kann keine, eine, oder mehrere Markierungen besitzen (aufnehmen). Es kann eine maximale Kapazität k einer Stelle geben, die Übergänge beeinflusst (k-begrenzte Stellen, im Gegensatz zu unbegrenzten mit $k \to \infty$).
- Eine Markierung kann durch Übergänge $t \in T$ von eine Stelle $s_1 \in S$ zu einer anderen $s_2 \in S$ übertragen werden.

Interpretation

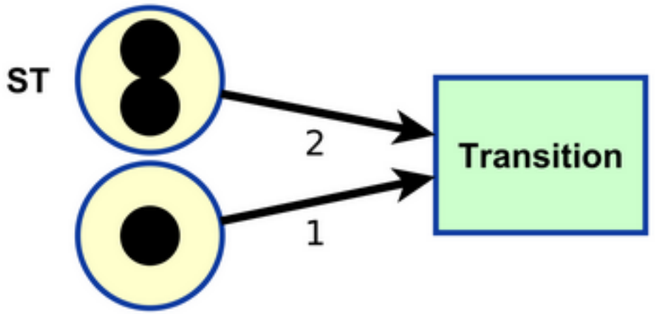
- Petri-Netze können auch verschiedene Weise interpretiert werden, d.h. die Abbildung von Systemen auf PN durch Assoziation von Stellen und Übergängen!



PETRI-NETZE: EINFÜHRUNG UND GRUNDLAGEN

Regeln für den Übergang

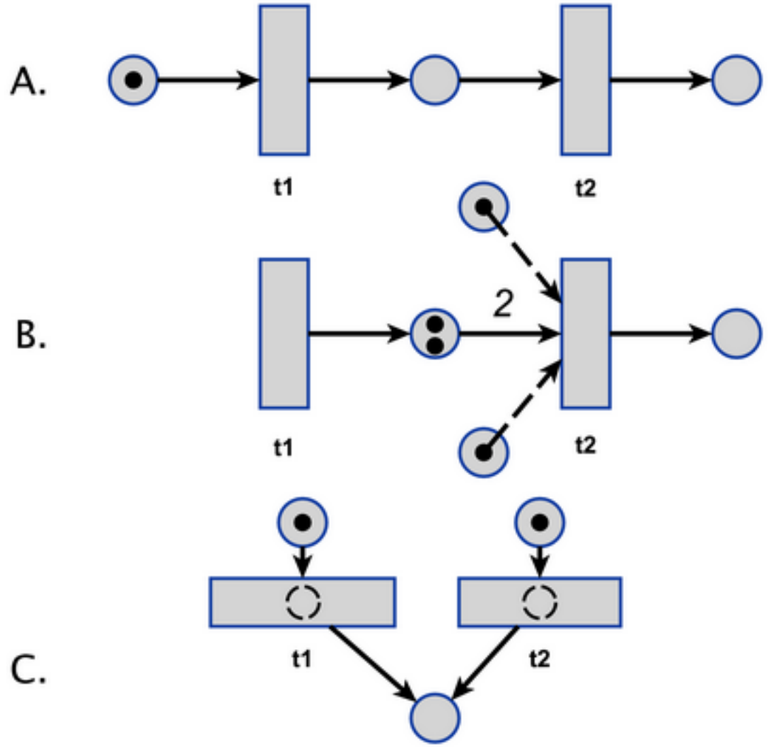
- Jedem Übergang $t \in T$ ist eine Menge von Eingangsstellen $S_E \in S$ zugeordnet (mindestens eine Stelle).
- Jedem Übergang ist eine Menge von Ausgangsstellen $S_A \in S$ zugeordnet (mindestens eine Stelle).
- Ein Übergang kann schalten, d.h. Markierungen von den Eingangs- zu den Ausgangsstellen übertragen, wenn alle Eingangsstellen wenigstens eine Markierung besitzen.
- Eine Kante kann eine Gewichtung g besitzen, die die Anzahl der gleichzeitig zu übertragenden Markierungen angibt.



PETRI-NETZE: EINFÜHRUNG UND GRUNDLAGEN

Dynamik: Kausalität

Verschiedene Grundsituationen und Vorbedingungen (Guards/Aktivierung) für Übergänge



A. Der Übergang t_{1+} muss vor dem Übergang $_{t}2$~ geschaltet haben → Kausalität, Ereignisreihenfolge

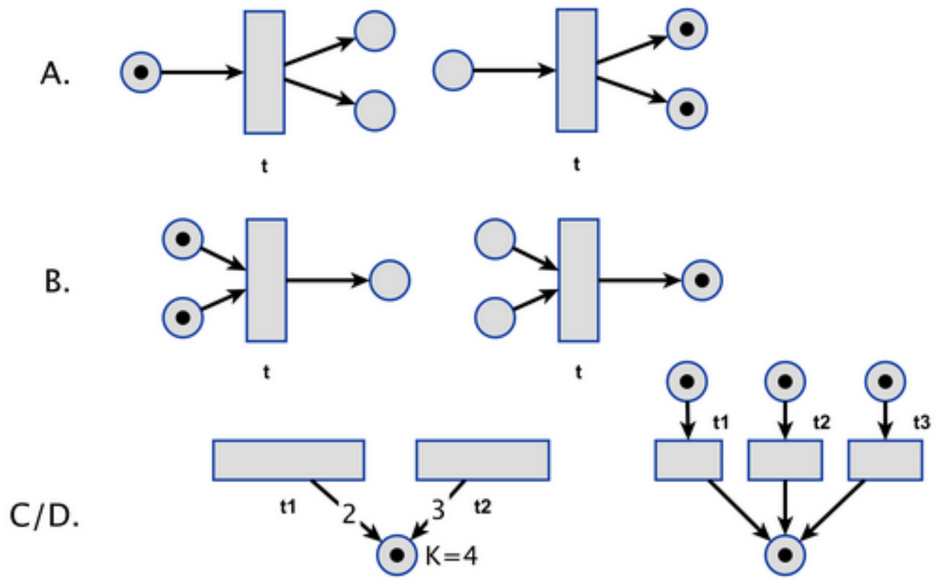
B. Hier muss t_1 zweimal geschaltet haben, um die Kante zu t_2 mit dem Gewicht g=2 zu aktivieren. Zusätzlich müssen zwei weitere Stellen Markierungen besitzen, um den Übergang t_2 mit insgesamt vier Markierungen aktivieren zu können.

C. Konflikt: Entweder Übergang t_1 oder t_2 wird aktiviert (nicht deterministisch!)

PETRI-NETZE: EINFÜHRUNG UND GRUNDLAGEN

Dynamik: Parallelität und Synchronisation

Verschiedene Grundsituationen für die Synchronisation bei paralleler Aktivierung von Teilnetzen



A. Fork Situation: Ein Übergang t produziert bei seiner Aktivierung mehrere Markierungen (entsprechend seiner Ausgangskanten).

B. Join Situation: Mehrere Markierungen entsprechend der Eingangskanten werden konsumiert und aktivieren den Übergang t.

C. D. Kontakt: nur bei Stellen mit endlicher Kapazität k möglich. Ähnlich dem Konflikt, wo ein ein Übergang (Kante x) einen anderen deaktiviert (Kante y). Jedoch auch D: Nebenläufigkeit der Übergänge t1, t2, und t3.

PETRI-NETZE: EINFÜHRUNG UND GRUNDLAGEN

Beispiel: Dining Philosophers

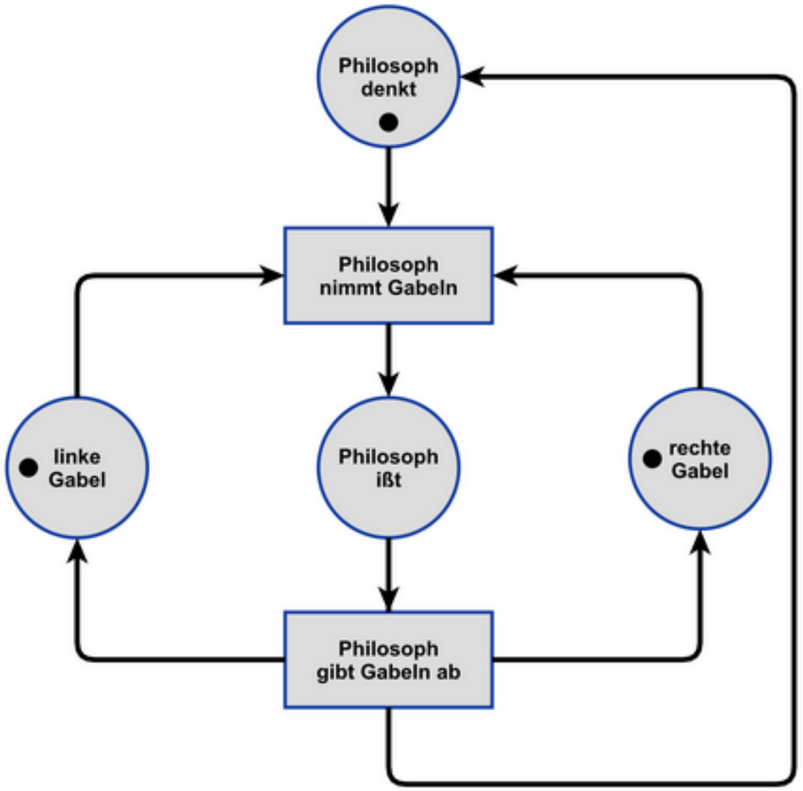


Abb. 16. Dinierende Philosophen (Ausschnitt für einen Philosophen) mit einer möglichen Situation (linke und rechte Gabel Stellen sind Ressourcen, und die beiden anderen Stellen bilden den Zustand des Philosophens ab). Es gilt k=1!

PETRI-NETZE: EINFÜHRUNG UND GRUNDLAGEN

Petri-Netze mit Ressourcen und Prozessen

- Stellen S können Lager/Kontainer für Daten/Ressourcen (Markierungen/Tokens) repräsentieren, meistens mit einer Kapazität k > 1!
- FIFO-Queues wären Implementierungen solcher Kontainer!
- Übergänge T können mit Prozessen oder Operationen verknüpft sein, die Daten/Ressourcen verarbeiten.
- Datenflussgraphen können in solche ST Netze abgebildet werden.

Petri-Netze und Digitallogik

- ST Petri-Netze können ebenfalls zur Darstellung kombinatorischer und sequenzieller Digitallogiksysteme verwendet werden.
- Dabei werden Stellen Signalwerten, und Übergängen Signaländerungen zugeordnet.

PETRI-NETZE: EINFÜHRUNG UND GRUNDLAGEN

- Ein Signal (hier x und y) wird mit zwei Stellen verknüpft (zweiwertige Logik).
- Die Übergänge werden mit einer Änderung des gegebenen Signals verknüpft (Aktivierungsbedingung): $A = \{a+, a-, a\sim\}$, wobei $a+$ den Übergang $0 \rightarrow 1$, $a-$: $1 \rightarrow 0$, und $a\sim$ entweder $0 \rightarrow 1$ oder $1 \rightarrow 0$ des Signals a bedeutet.
- Der bidirektionale Pfeil (Schleife) bedeutet hier einen "nur-lese" Transfer der Stellen der Eingangssignale!

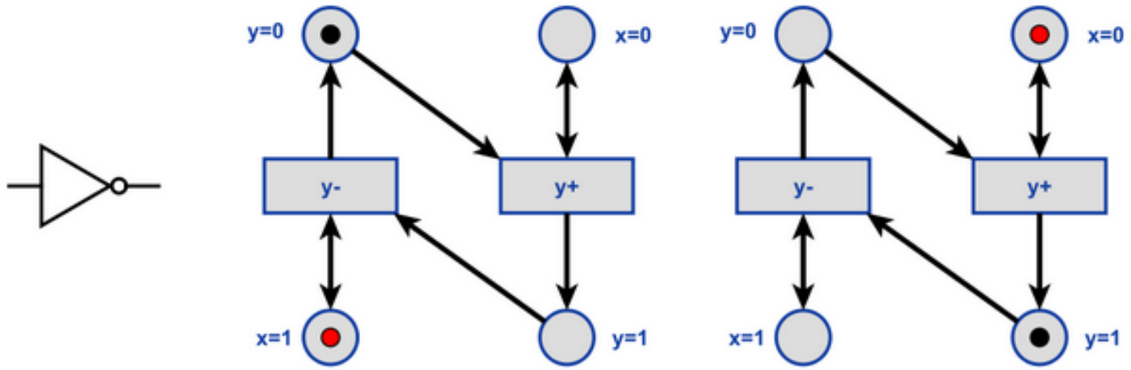


Abb. 17. Beispiel: (Circuit) PN für die Beschreibung des Verhaltens eines Inverters aus der Digitallogik (kombinatorische Logik) mit erweiterten Regelsatz für Übergänge

PETRI-NETZE: EINFÜHRUNG UND GRUNDLAGEN

- Die Plätze der Eingangssignale werden explizit mit Markierungen versehen.
- Anmerkung: Das PN bildet einen Zustandsübergangsgraphen (STG) ab!

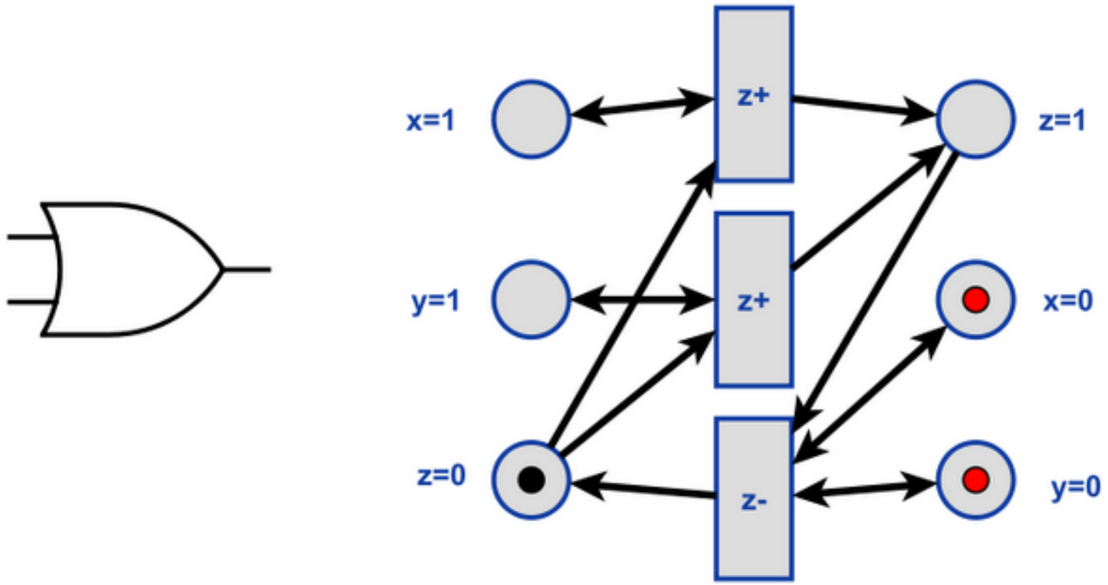


Abb. 18. Weiteres Beispiel: (Circuit) PN für die Beschreibung des Verhaltens eines Oder-Gatters

KOMMUNIZIERENDE SEQ. PROZESSE UND PETRI-NETZE

- Petri-Netze können zur Beschreibung von Multi-Prozess (MP) Systemen verwendet werden.
- Petri-Netze können aus Signalflussdiagrammen oder Datenflussgraphen abgeleitet werden.
- Sie können dann zur Synthese von MP Systemen verwendet werden.

Kontroll-Prozess-Modell

Prozesse P

Prozesse $p \in P$ führen (sequenziell) Anweisungen durch.
Dabei werden Prozesse auf Übergänge $t \in T$ abgebildet, die mit der sequenziellen Ausführung von Anweisungen bei deren Aktivierung verknüpft sind.

Prozesssteuerung

Markierungen aktivieren Prozesse und dienen der Synchronisation.
Parallelisierung durch spezielle Fork-Übergänge und Synchronisation durch spezielle Join-Übergänge.

KOMMUNIZIERENDE SEQ. PROZESSE UND PETRI-NETZE

Kommunikation

Interprozess-Kommunikation (synchronisierter Datenaustausch) findet durch Kontainer (Queues) statt, die durch spezielle Übergänge $t \in T$ repräsentiert und abgebildet werden.

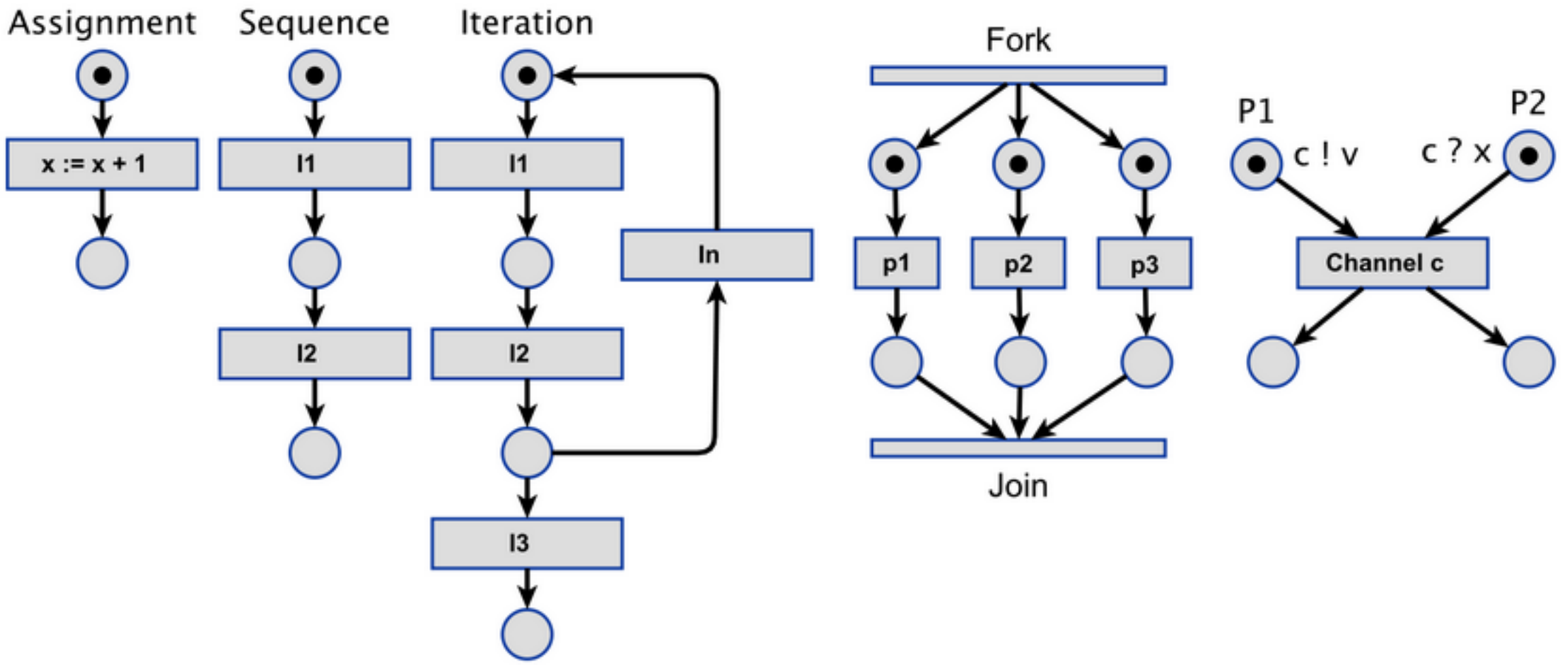


Abb. 19. Abbildungen von Anweisungen und Prozesskonstruktoren (z. B. :=, Seq, while, Par, ?!) auf Kontroll-Prozess-Petri-Netze

KOMMUNIZIERENDE SEQ. PROZESSE UND PETRI-NETZE

Daten-Prozess-Modell

Prozesse P

Prozesse $p \in P$ (bzw. funktionale Blöcke $f \in F$ deren Berechnung durch Prozesse gekapselt werden können) führen Berechnungen durch.

- » Dabei werden Prozesse auf Übergänge $t \in T$ abgebildet, die mit der Berechnung als Aktion verknüpft sind.
- » Direkte Abbildung aus Signalflussdiagrammen möglich!
- » Pipeline-Architektur

Daten

Daten werden durch Markierungen repräsentiert.

- » Sie dienen der Aktivierung von Prozessen = Übergängen, die die Daten verarbeiten sollen.

KOMMUNIZIERENDE SEQ. PROZESSE UND PETRI-NETZE

Kommunikation

Interprozess-Kommunikation (synchronisierter Datenaustausch) findet durch Kontainer (Queues) statt, die durch Stellen sÎS repräsentiert und abgebildet werden.

» Die Daten (Markierungen) werden durch die Stellen zwischen Übergangen übertragen.

KOMMUNIZIERENDE SEQ. PROZESSE UND PETRI-NETZE

Beispiel aus der Digitalen Signalverarbeitung

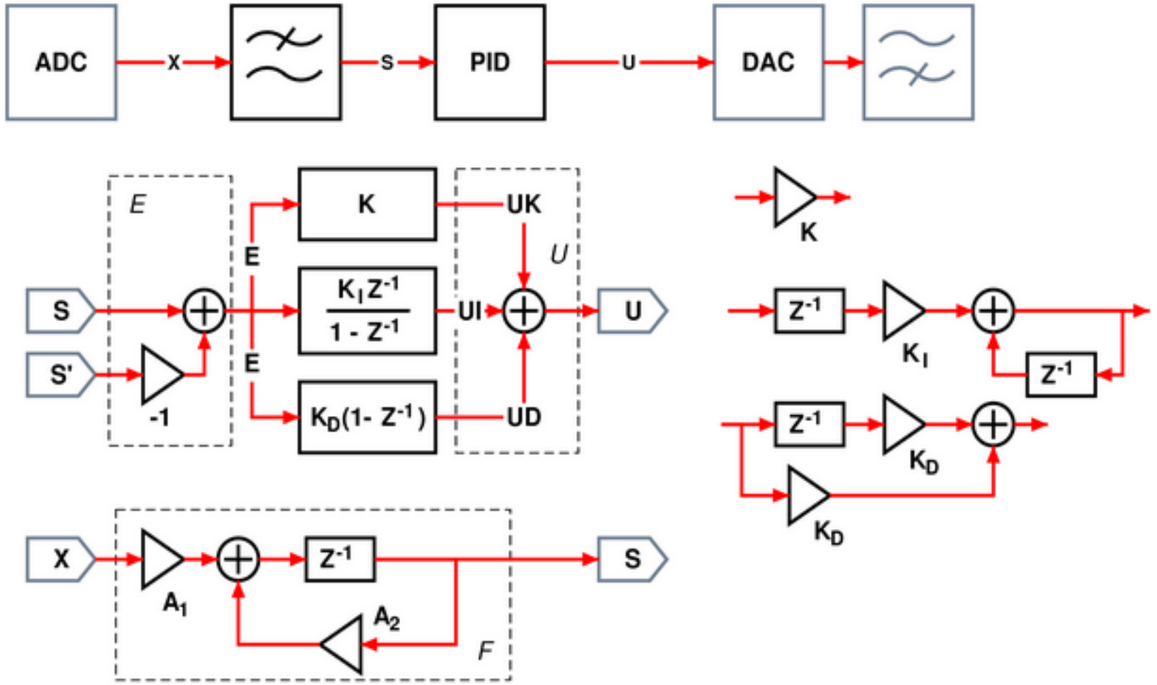


Abb. 20. Beispiel eines Signalflussdiagramms für einen PID-Regler Algorithmus mit Grundelementen Addierer , Multiplizierer , und Verzögerungs-/Speicherelement Z^{-1}

KOMMUNIZIERENDE SEQ. PROZESSE UND PETRI-NETZE

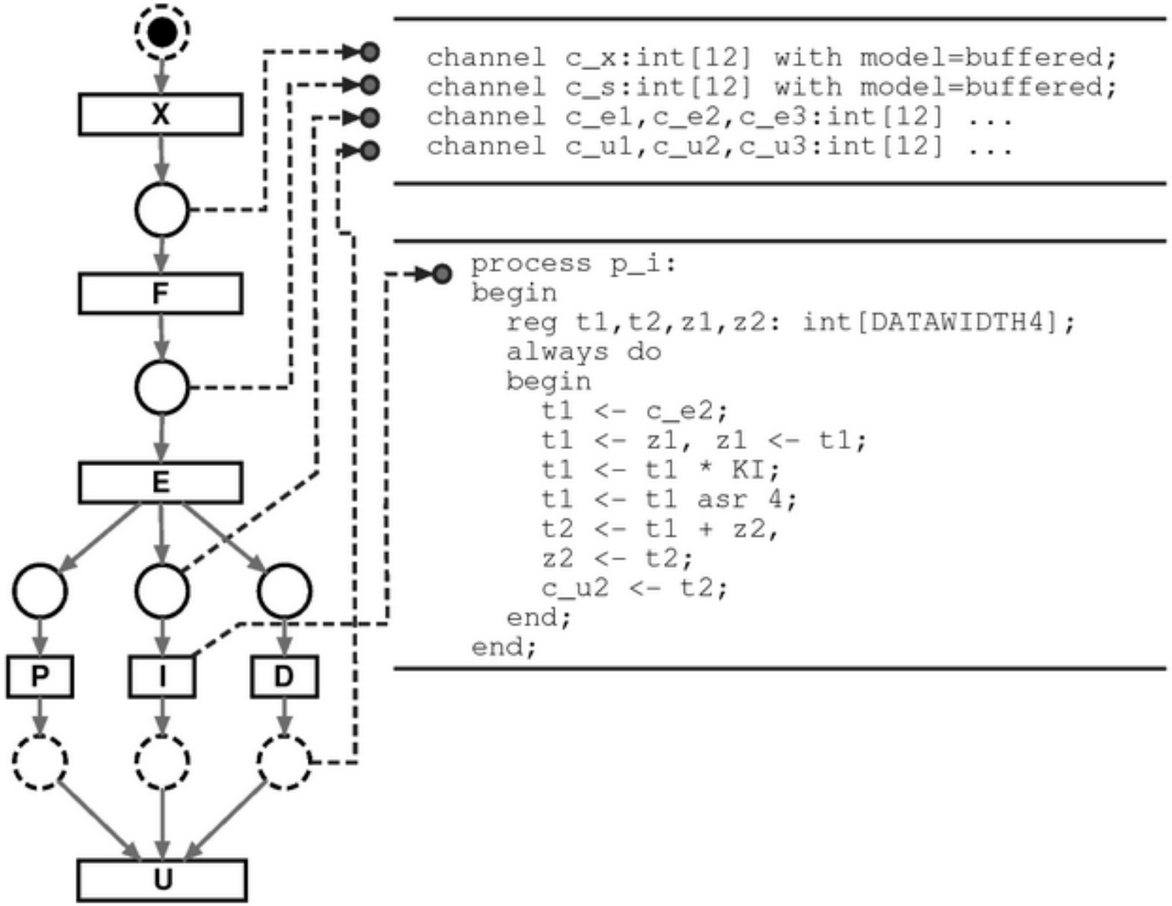


Abb. 21. Abbildung auf Petri-Netz (Funktionsblöcke/Operationen auf Übergänge, Daten auf Stellen) und hier umgekehrt schließlich auf CSP (process) mit Kommunikationskanälen (channel) und ConPro Programmiersprache.

DEADLOCKS IN PETRI-NETZEN

- Deadlocks in Petri-Netzen entstehen durch Konflikte oder Mehfachauswahl bei Übergängen (Nicht-Determinismus), wo die Reihenfolge der Aktivierungen maßgeblich ist.
  1. PN mit genau einer Kante zu und von jeder Stelle sind deadlock/konfliktfrei!
  2. PN wo alle Übergänge höchstens einen Eingangs- und Ausgangsplatz besitzen sind deadlock/konfliktfrei!

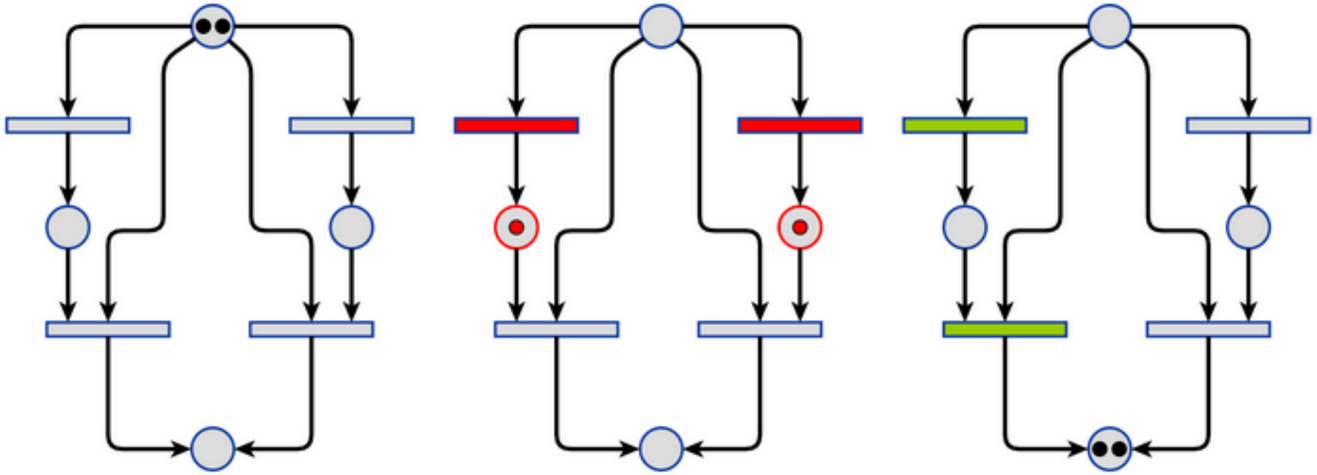


Abb. 22. Beispiel einer Deadlock Situation die in einem Petri-Netz auftreten kann, in Abhängigkeit von der Reihenfolge der Aktivierungen der Übergänge

VERHALTENSMODELLE

Zustands-basiert (Verhalten)

Reaktive Systeme:

- Finite State Machines
- Petri-Netze
- Hierarchische nebenläufige FSMs

Aktions-basiert (Verhalten)

Transformatorische Systeme:

- Datenflussgraph
- Kontrollflussgraph

Heterogene Modelle

- Programmflussgraph: Kontrollflussgraph + Datenflussgraph
- Programmiersprachliche Paradigmen

VERHALTENSMODELLE

Daten-, Kontroll-, und Programmfluss

- Ein Programm wird durch einen Programmfluss beschrieben
- Der Programmfluss lässt sich in reine Daten- und Kontrollanweisungen zerlegen → Daten- und Kontrollfluss
- Der Datenfluss wird durch den Datenpfad implementiert (Verhaltensbeschreibung des Datenpfades, mit Datenabhängigkeitsgraphen) → Register, Funktionale Operatoren, Datenpfadselektoren (Multiplexer)
- Der Kontrollfluss wird durch den Kontrollpfad implementiert (Verhaltensbeschreibung des Kontrollpfades mit Zustandsübergangsgraphen) → Zustandsautomat
- Datenanweisungen verwenden arithmetische, relationale, und Boolesche Ausdrücke

DATENABHÄNGIGKEITSGRAPHEN

- Eine Datenabhängigkeit in der Informatik ist eine Situation, in der sich eine Programmanweisung (Anweisung) auf die Daten einer vorhergehenden Anweisung bezieht.
- Ein Datenabhängigkeitsgraph G = (I, E) besteht aus einer Menge von Instruktionen I und einer Menge transitiver Relationen R = I x I, mit (a, b) in R falls die Instruktion a vor b bewertet werden muss.
- Mit anderen Worten, es gibt eine Kante von a nach b:

wenn a vor b ausgewertet werden muss.

Scheduling

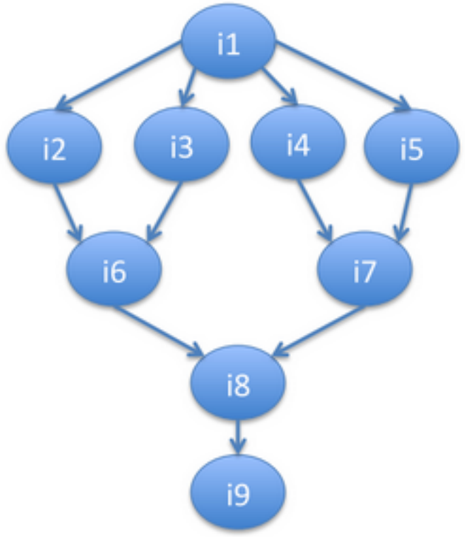
- Jeder Knoten (Anweisung) auf derselben Ebene ist unabhängig von der anderen → Parallelisierung
- Um die Anweisungen in Level *i* auszuführen, müssen die Anweisungen in Level *i*-1 beendet sein

DATENABHÄNGIGKEITSGRAPHEN

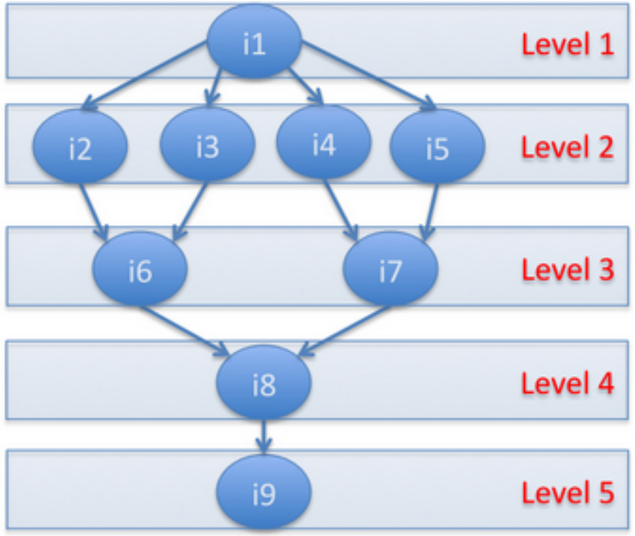
Program

i1) a1 = f(x) ;
i2) a2 = a1 + 3 ;
i3) a3 = a1 * 5;
i4) a4 = a1 – 6 ;
i5) a5 = a1 / 2;
i6) a6 = a2 * a3;
i7) a7 = a4 – a5 ;
i8) a8 = a6 * a7 ;
i9) crt(a8) ;

Data dependency graph



Scheduling



[Ortiz-Ubarri, José]

Abb. 23. Beispiel eines DAG für eine Instruktionssequenz (nur Datenanweisungen)

PARALLELISIERUNG

PARALLELISIERUNGSKLASSEN

Datenfluss

Der Datenfluss beschreibt den Fluss von Daten durch Verarbeitungs- und Speichereinheiten (Register).
Die Verarbeitungseinheiten sind Knoten eines gerichteten Graphs, die Kanten beschreiben den Datenfluss und bilden die Datenpfade. Die Verarbeitungseinheiten müssen aktiviert werden.

Kontrollfluss

Der Kontrollfluss beschreibt die temporale schrittweise Verarbeitung von Daten im Datenpfad durch zustandsbasierte selektive Aktivierung von Verarbeitungseinheiten.
Der Kontrollfluss kann durch Zustandsübergangsdiagramme beschrieben werden.

Programmfluss

Der Programmfluss setzt sich kombiniert aus Daten- und Kontrollfluss zusammen.

PARALLELISIERUNGSKLASSEN

- Programmanweisungen werden Zuständen S1,S2,.. zugeordnet.
- Einfache Anweisungen (Berechnungen) werden jeweils einem Zustand, komplexe Anweisungen i.A. mehreren Unterzuständen zugeordnet.

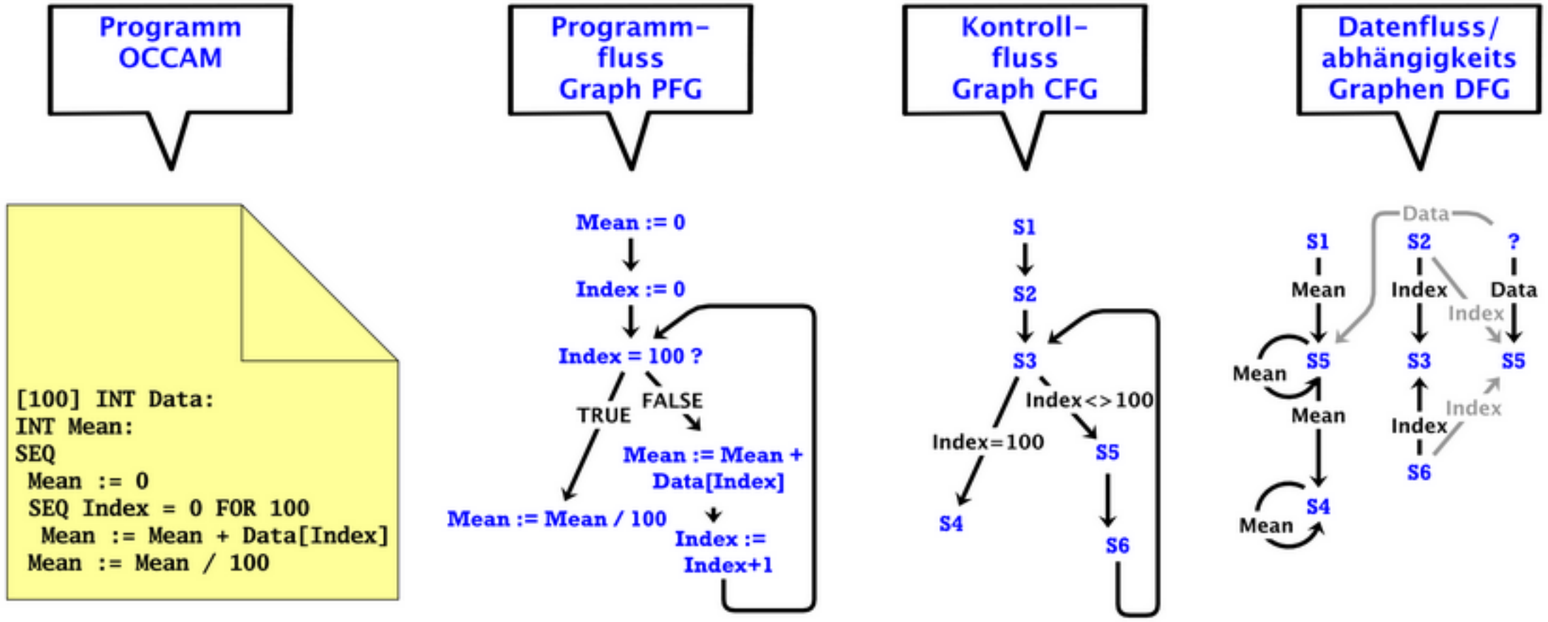


Abb. 24. Programm-, Kontroll-, und Datenflussgraphen

PARALLELISIERUNGSKLASSEN

Datenparallelität

In vielen Programmen werden dieselben Operationen auf unterschiedliche Elemente einer Datenstruktur angewendet. Im einfachsten Fall sind dies die Elemente eines Feldes.

- Wenn die angewendeten Operationen unabhängig voneinander sind, kann diese verfügbare Parallelität dadurch ausgenutzt werden, um die zu manipulierenden Elemente der Datenstruktur auf verschiedene Prozessoren zu verteilen, so dass jeder Prozessor die Operation auf den ihm zugeordneten Elementen ausführt.
- Parallelität im Datenpfad → Feine Granularität
- Bei der Datenparallelität wird unterschieden zwischen:
  - » Parallele Ausführung der gleichen Instruktion auf verschiedenen Daten (Vektorparallelität) → Vektoranweisung
  - » Ausführung verschiedener Instruktionen die nur Daten verarbeiten (reine Datenanweisungen)

PARALLELISIERUNGSKLASSEN

- Zur Ausnutzung der Datenparallelität wurden sequentielle Programmiersprachen zu datenparallelen Programmiersprachen erweitert. Diese verwenden wie sequentielle Programmiersprachen einen Kontrollfluss, der aber auch datenparallele Operationen ausführen kann.
  - » Z.B. µRTL: Bindung von mehreren Datenpfadanweisungen durch Kommasyntax: $x \leftarrow \varepsilon, y \leftarrow \varepsilon, .., z \leftarrow \varepsilon;$
- Häufig werden Vektoranweisungen deklarativ und nicht prozedural imperativ beschrieben.
- Beispiel für eine deklarative Vektoranweisung und die dazugehörige imperative Anweisungssequenz in Schleifenform (Fortran 90):

```
a(1 : n) = b(0 : n − 1) + c(1 : n) ⇔
FOR (i=1:n)
  a(i) = b(i-1) + c(i)
ENDFOR
```

PARALLELISIERUNGSKLASSEN

- Datenabhängigkeiten können zwischen der linken und rechten Seite einer Datenzuweisung bestehen, so dass
  - » zuerst auf alle auf der rechten Seite auftretenden Felder zugegriffen wird und die auf der rechten Seite spezifizierten Berechnungen durchgeführt werden,
  - » bevor die Zuweisung an das Feld auf der linken Seite der Vektoranweisung erfolgt!
- Daher ist folgende Vektoranweisung nicht in die folgende Schleife transformierbar:

```
a(1 : n) = a(0 : n − 1) + a(2 : n + 1) ≠
FOR (i=1:n)
  a(i) = a(i-1) + a(i+1)
ENDFOR
```

- Datenparallelität benötigt i.A. keine weitere Synchronisation

PARALLELISIERUNGSKLASSEN

Instruktionsparallelität

- Parallelität im Kontrollpfad (Zustandsautomat) auf Instruktions- und Prozessebene → Grobe Granularität je nach Anzahl der Instruktionen pro Prozess (Task)
- Gebundene Instruktionsblöcke sind Parallelprozesse (*Par* Prozesskonstruktor)
- Bekannt als Multithreading Programmier- und Ausführungsmodell (z.B. *pthreads*)
- Instruktionsparallelität benötigt i.A. Synchronisation zwischen den einzelnen Prozessen

PARALLELISIERUNGSKLASSEN

Möglichkeiten der Parallelisierung

1. Parallelität auf Bitebene (jegliche funktionale Operation)
2. Parallelität durch Pipelining (nur Datenströme)
3. Parallelität durch mehrere Funktionseinheiten:
   - » Superskalare Prozessoren
   - » Very Large Instruction Word (VLIW) Prozessoren
4. Funktionsparallelität (Evaluierung der Funktionsargumente und Rekursion)
5. Parallelität auf Prozess- bzw. Threadebene (Kontrollpfadebene)

PARALLELISIERUNGSMETHODEN

Abrollung von Schleifen

- Das Abrollen von Schleifen ist eine der gängigsten Parallelisierungsmethoden bei der Schleifen durch Replikation der Schleifeninstruktionen (Schleifenkörper) in eine paralellisierbare Anweisungssequenz
  - » teilweise
  - » oder vollständig transformiert werden.

Definition 9.

```
FOR i = i1 to i2 DO
  FOR j = j1 to j2 DO
    ..
    IB(i,j,x(i,j),x(i+1,j),..,);
  END
END ⇒
  IB(i1,j1,x(i1,j1),x(i1+1,j1),..) ‖
  IB(i1+1,j1,x(i1+1,j1),x(i1+2,j1),..) ‖
  IB(i1+n,ij+m,x(i1+n,ij+m),x(i1+n+1,j1+m),..) ‖
  ..
```

PARALLELISIERUNGSMETHODEN

- Zu Beachten sind Datenabhängigkeiten zwischen einzelnen Anweisungsiterationen und Kosten durch die Abrollung (Overhead)
- Beispiel: Nicht abrollbar *und* parallelisierbar durch Datenabhängigkeit $IS_b(IS_{b-1}(IS_{b-2}(..))$!

```
X := 1;
FOR i = a to b DO
  IS: X := X*i;
END
```

PARALLELISIERUNGSMETHODEN

Deklarative Beschreibung von Vektoroperationen

- In der Bildverarbeitung werden Vektoroperationen (und Matrixoperationen) sehr häufig verwendet. Dabei werden häufig die gleichen Operationen auf alle Pixel eines Bildes (Feldelemente) angewendet.
- Anstelle der iterativen Berechnung mittels Schleifen definiert man deklarativ die Berechnung eines Pixels (**Punktoperator**) oder eines Bereiches des Bildes (**Lokaloperator**):

Definition 10.
$img' = F(img) ::= \{f(x)|x \in img\} \Leftrightarrow$

```
PROCEDURE F (img: VECTOR OF type):
BEGIN
  RETURN f(img)
END
```

PARALLELISIERUNGSMETHODEN

Beispiel Punktoperator

```
1   PROCEDURE rgb2color(g_red,g_green,g_blue: VECTOR OF gray):
2             VECTOR OF color;
3   (* transform gray image to color image *)
4   VAR res: VECTOR OF color;
5   BEGIN
6     res.red   := g_red;
7     res.green := g_green;
8     res.blue  := g_blue;
9     RETURN res;
10  END rgb2color;
```

```
1   PROCEDURE gray2color(img: VECTOR OF gray): VECTOR OF color;
2   (* transform gray image to color image *)
3   BEGIN
4     RETURN rgb2color(img,img,img)
5   END gray2color;
```

```
1   PROCEDURE color2gray(img: VECTOR OF color): VECTOR OF gray;
2   (* transform color image to gray image *)
3   BEGIN
4     RETURN (img.red + img.green + img.blue) DIV 3
5   END color2gray;
```

Abb. 25. Algorithmus: Transformation von Farb- in Grauwertbilder [2]

PARALLELISIERUNGSMETHODEN

JavaScript

- JavaScript (und andere funktionalen Programmiersprachen) arbeiten intensiv mit Listen und Arrays
- Listen und Arrays lassen sich mittels eines Abbildungsoperators und einer Elementfunktion (Punktoperator) transformieren (map-and-reduce)

JavaScript

CLEAR HELP Point

PARALLELISIERUNGSMETHODEN

Lokaloperatoren

- Lokale Operatoren sind rechenintensiver als Punktoperatoren. Um ein neues Pixel zu berechnen wird der alte Wert des Pixels sowie die alten räumlich benachbarten Werte der Pixel innerhalb einer definierten maximalen Entfernung verwendet.
- Für eine 3x3-Nachbarschaftsberechnung werden zum Beispiel Neun Datenwerte für jedes Pixel benötigt.
- Für die Berechnung lokaler Operatoren wird eine bestimmte Anzahl von parallelen Datenaustauschvorgängen benötigt → Kommunikation.
- Wenn jedes Pixel auf einem anderen Prozessor gespeichert wird ist Kommunikation teuer!

PARALLELISIERUNGSMETHODEN

Beispiel Lokaloperator

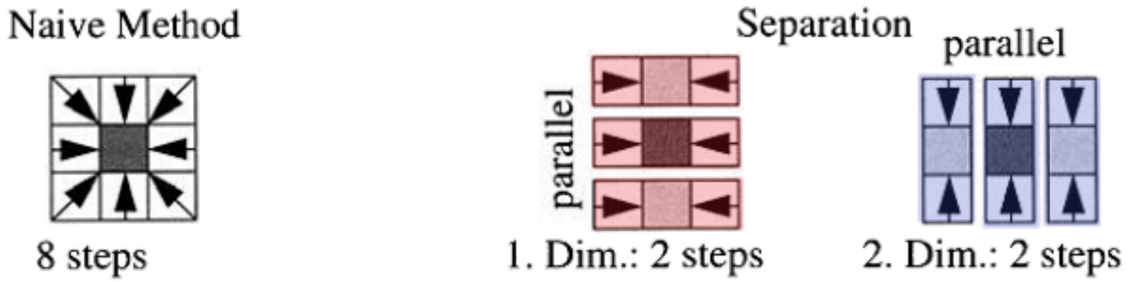


Abb. 26. Komplexitätsreduktion durch Separation der Operatoren: (1) Alle Zeilenoperationen werden parallel ausgeführt (2) Alle Spaltenoperationen werden parallel ausgeführt

```
1   PROCEDURE sum_3x3(img: grid OF gray): grid OF INTEGER;
2   (* returns sum of local 3x3 neighborhood area *)
3   VAR res: grid OF INTEGER;
4   BEGIN
5     res:= img + MOVE.right(img) + MOVE.left(img);(*horizontal*)
6     res:= res + MOVE.down(res)  + MOVE.up(res);  (*vertical  *)
7     RETURN res;
8   END sum_3x3;
```

Abb. 27. Algorithmus: Parallele Nachbarschaftsberechnung

PARALLELISIERUNGSMETHODEN

Basisblock Optimierer und Scheduler

- Nebenläufigkeits- und Datenabhängigkeitsanalyse
- Ziel: Reduktion von Zeitschritten, Minimierung der Latenz, Parallelisierung

1. Basisblöcke sind Bereiche im Programmflussgraphen, die nur einen Kontrollzugang am Kopf und nur einen Kontrollausgang am Ende haben, und keine weiteren Seiteneingänge.
2. Ein Basisblock der nur aus Datenanweisungen besteht ist ein **Major-Block**. Dieser wird in elementare **Minor-Blöcke** zerlegt, die wenigstens eine Anweisung enthalten (bei einem gebunden Block der ganze Block)
3. Es werden Datenabhängigkeitsgraphen für jeden Major-Block erstellt.
4. Nicht abhängige Anweisungen werden durch einen Scheduler mit ASAP-Verhalten in einen **gebundenen Block** (paralleler Prozess) zusammengefasst.

FUNKTIONALE PROGRAMMIERUNG

- Determinismus macht paralleles Auswerten der Argumente und Parallelisierung bei Rekursion möglich
- Parameter von Funktionen sind datenunabhängig → Parallele Evaluierung der Funktionsargumente bei der Funktionsapplikation
- Funktionsaufrufe sind gänzlich oder partiell datenunabhängig → Parallelisierung der Funktionsaufrufe (Endrekursion, Kopfrekursion bringt kein Vorteil)
- Probleme:
  - » Argumente haben teils kein ausreichendes Potential (Ausdrücke zu einfach und zu flach)
  - » Große Rekursionstiefen können zu einer zu feinen Aufteilung führen → nicht gut auf Prozessoren aufzuteilen

ASYNCHRONE EREIGNISVERARBEITUNG

- Neben der Parallelisierung der reinen Datenverarbeitung (Berechnung) kann auch eine Partitionierung von Ein- und Ausgabe bzw. der Ereignisverarbeitung erfolgen
- Bekanntes Beispiel: Geräteinterrupts mit einfachen Foreground-Background System mit zwei Prozessen:
  - P1: Hohe Priorität (Ereignisverarbeitung → Interrupthandler → Foreground Prozess)
  - P2: Niedrige Priorität (Berechnung → Hauptprogramm → Background Prozess) mit Preemption durch Ereignisverarbeitung
- Ein- und Ausgabeoperationen eines Programms können **synchron** (blockierend) oder **asynchron** (im Hintergrund verarbeitet und nicht blockierend) ausgeführt werden.

ASYNCHRONE EREIGNISVERARBEITUNG

Beispiel JavaScript

- Synchrone Operationen liefern das Ergebnis in einer Datenanweisung zurück die solange blockiert wird bis das Ereignis eintritt(Ergebnisdaten verfügbar sind). Dabei werden zwei aufeinander folgende Operationen sequenziell ausgeführt, und eine folgende Berechnung (nicht ereignisabhängig) erste nach den Ereignissen ausgeführt:

```
x = IO1(arg1,arg2,..);
y = IO2(arg1,arg2,..);
z = f(x,y);
```

- Bei der asynchronen (nebenläufigen) Ausführung von ereignisabhängigen Operationen wird das Ergebnis über eine Callback Funktion verarbeitet. Die IO Operation blockiert nicht. Vorteil: Folgende Berechnungen können unmittelbar ausgeführt werden.

```
IO1(arg1,arg2,..,function (res) { x=res; });
IO2(arg1,arg2,..,function (res) { y=res; });
z = f(x,y); // Problem?
```

ASYNCHRONE EREIGNISVERARBEITUNG

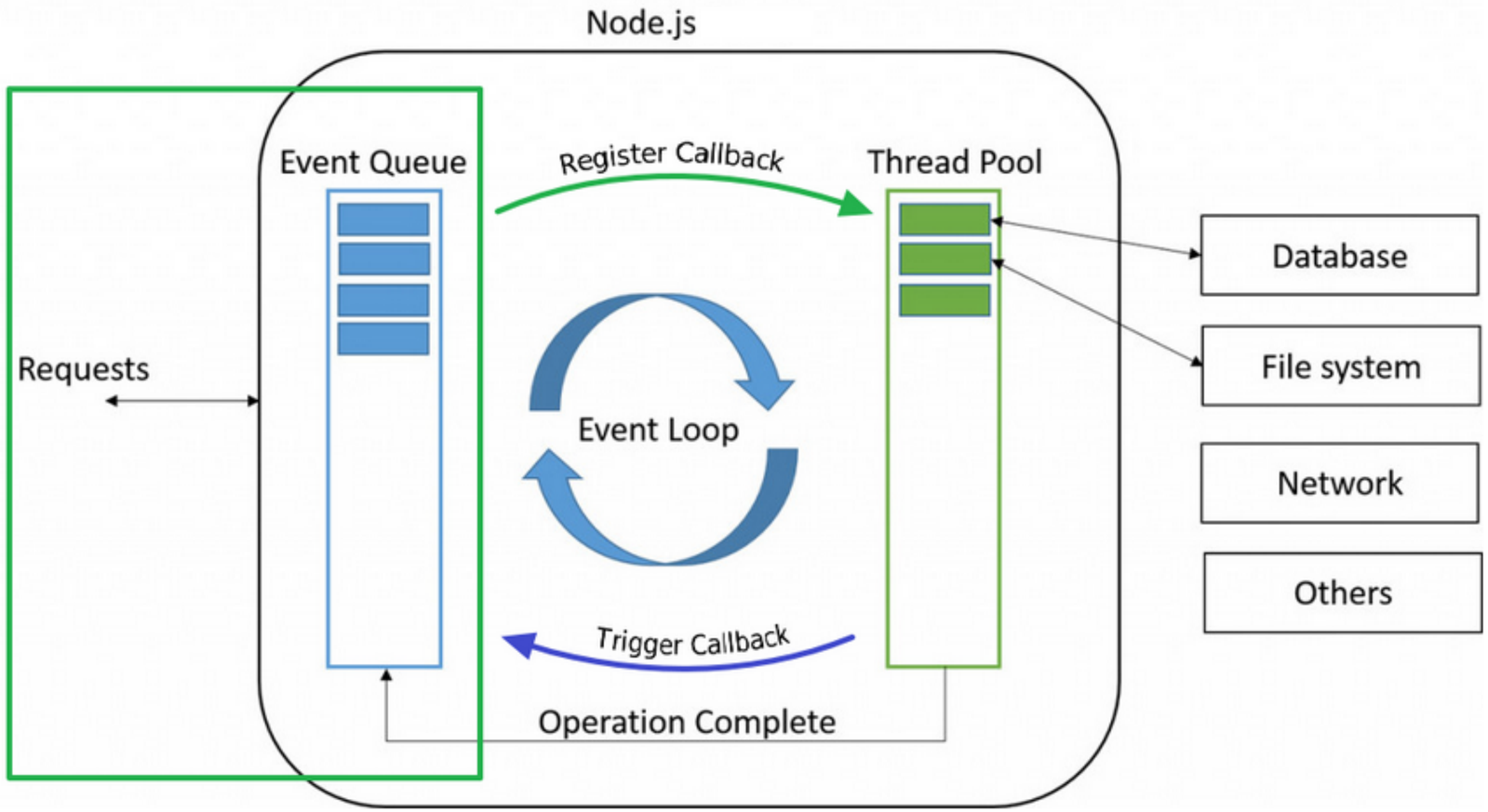


Abb. 28. Ereignisbasierte Verarbeitung von asynchronen Operationen in node.js/jxcore: Ein JS Thread verbunden über die Eventloop mit N IO Threads

ASYNCHRONE EREIGNISVERARBEITUNG

Synchronisation

- Asynchoene Ereignisverarbeitung mit preemptiven Verhalten benötigt *explizite* Synchronisation (Locks...) zur Atomarisierung von kritischen Bereichen
- Asynchrine Ereignisverarbeitung spaltet den Datenfluss auf und benötigt Daten- und Ergebnissynchronisation über Prädikatfunktionen oder explizite Synchronisation:

```
var x,y,z;
function P(f,x,y,z) {
  if (x!=undefined && y!=undefined)
    return f(x,y);
  else return z;
}
IO1(arg1,arg2,..,function (res) { x=res; z=P(f,x,y,z)});
IO2(arg1,arg2,..,function (res) { y=res; z=P(f,x,y,z)});
```

PARALLELE PROGRAMMIERUNG

ÜBERBLICK

JAVASCRIPT :: DATEN UND VARIABLEN

- Variablen werden mit dem Schlüsselwort `var` definiert → Erzeugung eines Datencontainers!
- Es gibt keine Typendeklaration in JS! Kerntypen:

  T_{core}={`number, boolean, object, array, string, function`}
- Alle Variablen sind **polymorph** und können alle Werttypen aufnehmen.
- Bei der Variabledefinition kann ein Ausdruckswert zugewiesen werden

```
var v = ε,..;   v = ε;
```

JavaScript

CLEAR HELP var assign

JAVASCRIPT :: FUNKTIONEN

- Funktionen können mit einem Namen oder anonym definiert werden
- Funktionen sind Werte 1. Ordnung → Funktionen können Variablen oder Funktionsargumenten zugewiesen werden
- Eine Funktion kann einen Wert mit der `return` Anweisung zurückgeben. Ohne explizite Wertrückgabe → `undefined`
- Es wird nur Call-by-value Aufruf unterstützt - jedoch werden Objekte, Funktionen und Arrays als Referenz übergeben; Parameter p_i sind an Funktionsblock gebunden

```
function name (p1,p2,..) { statements ; return ε } name(ε1,ε2,..)
```

JavaScript

CLEAR HELP fun

JAVASCRIPT :: FUNKTIONEN

- Da in JavaScript Funktionen Werte erster Ordnung sind können
  - » Funktionen an Funktionen übergeben werden und
  - » Funktionen neue Funktionen zurückgeben (als Ergebnis mit `return`)
- Es können daher **anonyme** Funktionen `function (..) {..}` definiert werden die entweder einer Variablen als Wert oder als Funktionsargument übergeben werden.

```
var x = function (pi) { ε(pi) }
array.forEach(function(elem,index) { ε(pi) }
```

JAVASCRIPT :: DATENSTRUKTUREN

In JavaScript sind Objekte universelle Datenstrukturen (sowohl Datenstrukturen als auch Objekte) die mit Hashtabellen implementiert werden. Arrays werden in JavaScript ebenfalls als Hashtabelle implementiert!. D.h. Objekte == Datenstrukturen == Arrays == Hashtabellen.

- Es gibt *kein* nutzerdefinierbares Typensystem in JavaScript.
- Eine Datenstruktur kann jederzeit definiert und verändert werden (d.h. Attribute hinzugefügt werden)

```
var dataobject = {
  a:ε,
  b:ε, ..
  f:function () { .. }
}
..
dataobject.c = ε
```

JAVASCRIPT :: DATENSTRUKTUREN

- Dadurch dass Objekte und Arrays mit Hashtabellen implementiert (d.h. Elemente werden durch eine Textzeichenkette referenziert) werden gibt es verschiedene Möglichkeiten auf Datenstrukturen und Objektattribute zuzugreifen:

```
dataobject.attribute
dataobject["attribute"]
array[index]
array["attribute"]
```

JAVASCRIPT :: OBJEKTE

- Objekte zeichnen sich in der objektorientierten Programmierung durch Methoden aus mit der ein Zugriff auf die privaten Daten (Variablen) eines Objekts möglich wird.
- In JavaScript kann auf Variablen eines Objekts (die Attribute) immer direkt zugegriffen werden.
- Attribute können Funktionen sein - jedoch können die Funktionen nicht wie Methoden direkt auf die Daten des Objektes zugreifen.
- Daher definiert man Methoden über Prototypenerweiterung in JavaScript.
- Die Methoden können über die `this` Variable direkt auf das zugehörige Objekt zugreifen (also auch auf die Variablen/Attribute)
- Es gibt eine Konstruktionsfunktion für solche Objekte mit Prototypendefinition der Methoden
- Objekte werden mit dem `new` Operator und der Konstruktionsfunktion erzeugt.

JAVASCRIPT :: OBJEKTE

```
function constructor (p_i) {
  this.x=ε
  ..
}
constructor.prototype.method_i = function (..) {
  this.x=ε;
  ..
}
...

var obj = new constructor(..);
```

JAVASCRIPT :: THREADS.JS

- Die `threads.js` Bibliothek implementiert das CCSP Multiprozessmodell mittels Multithreading von `jxcore`
- Die Bibliothek muss mittels `require('./threads').init(`*`options`*`)` geladen und initialisiert werden

Ausführungsmodell

- Es werden Prozesse auf Threads abgebildet die bestenfalls 1:1 auf den CPUs / Cores ausgeführt werden. Es gibt eine Obergrenze an Threads (max. 64) die statisch bei der Initialisierung des threads Moduls über die Optionen festgelegt werden müssen(`numthr:number`).
- Prozesse können synchronisiert auf geteilte Objekte (konkurrierend) zugreifen. Der Zugriff erfolgt durch Nachrichtenaustausch mit dem Hauptthread (Hauptprozess).

JAVASCRIPT :: THREADS.JS

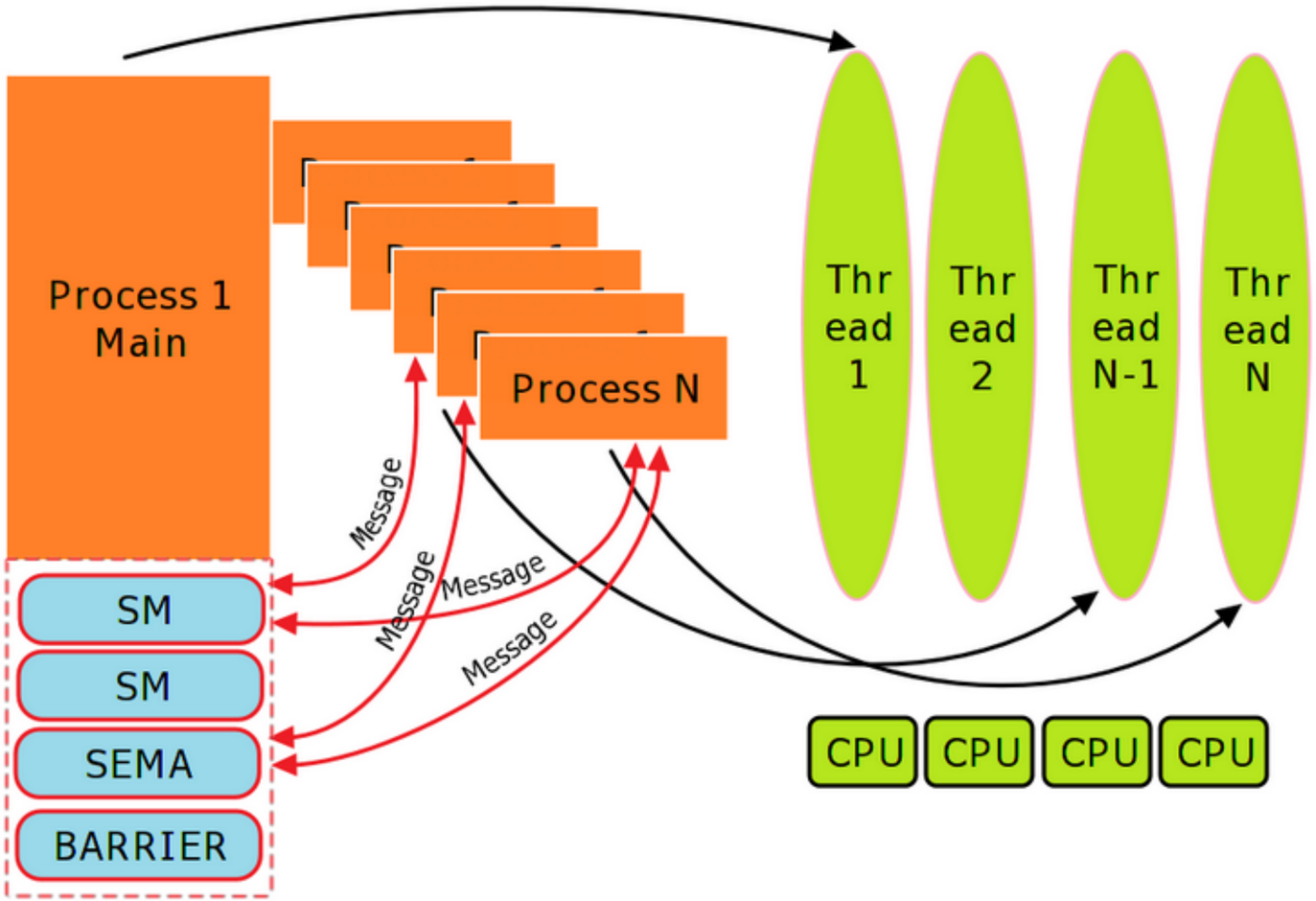


Abb. 29. Threadmodell und Kommunikation der Prozesse mit geteilten Objekten

JAVASCRIPT :: THREADS.JS

Prozesskonstruktoren

Seq([f_1,f_2,..])

Der *Seq* Konstruktor erzeugt eine sequenzielle Komposition von Prozessen (über einer Array von Funktionen definiert). Da JavsScript inhärent sequenziell ist gleichbedeutend mit: f_1();f_2();..

Par([f_1,f_2,..],{*shared*})

Der *Par* Konstruktor erzeugt eine parallele Komposition von Prozessen (über einer Array von Funktionen definiert). Alle Subprozesse werden nach Erzeugung automatisch gestartet. Der aufrufende Prozess wird blockiert bis alle Subprozesse terminiert sind. Ein *shared environment* kann mit geteilten (IPC) Objekten übergeben werden.

JAVASCRIPT :: THREADS.JS

Fork([f_1,f_2,..],{*shared*})

Der *Fork* Konstruktor erzeugt eine parallele Komposition von Prozessen (über einer Array von Funktionen definiert). Alle Subprozesse werden nach Erzeugung automatisch gestartet. Der aufrufende Prozess wird nicht blockiert! Ein *shared environment* kann mit geteilten (IPC) Objekten übergeben werden.

Geteilte Ressourcen

- Prozesse können sich Ressourcen teilen:
  - » Interprozesskommunikation (Channel, Semaphore, ..)
  - » Arrays (Matrizen)
- Geteilte Ressourcen müssen explizit erzeugt werden und als *zweites* Argument an die Prozesskonstruktoren übergeben werden.

JAVASCRIPT :: THREADS.JS

```
Par([
  function process1 () {
    var x;
    x=ch.read();
    log(x);
  },
  function process2 () {
    var x=Math.random();
    ch.write(x);
    log(x);
  }
],{
  ch:Channel()
})
```

JAVASCRIPT :: THREADS.JS

Weitere Prozesskonstruktoren:

Alt([[$f_{1,cond}$,$f_{1,do}$],[$f_{2,cond}$,$f_{2,do}$,..],{*shared*})

Der *Alt* Konstruktor erzeugt eine parallele Komposition von Prozessen (über ein Array von Funktionsarrays definiert) von denen aber nur ein blockierter konditionaler Prozess selektiert (aktiviert) wird. Jeder konditionale Prozess kann *eine* blockierende Operation ausführen. Der erste bereite Prozess wird ausgeführt mit nachfolgender Aktivierung (eines optionalen) *do* Prozesses. Der aufrufende Prozess wird blockiert bis ein Subprozess terminiert ist.

- Alternative Prozesskonstruktoren werden eingesetzt um
  - Auf eine Menge von synchronisierenden Kommunikationskanälen selektiv auswählend zugreifen zu können

JAVASCRIPT :: THREADS.JS

```
Alt([
  [
    function process1,cond () {
      x=ch1.read();
    },
    function process1,do () {
      log(x);
    },
  ],
  [
    function process2,cond () {
      x=ch2.read();
    },
    function process2,do () {
      log(x);
    },
  ],
  ..
],{
  ch1:Channel(),
  ch2:Channel(),
  ..
})
```

JAVASCRIPT :: THREADS.JS

ProcessArray

Der *ProcessArray* Konstruktor erzeugt eine parallele Komposition von einem Array von *N* Prozessen (über eine einzige Funktion definiert). Alle Subprozesse werden nach Erzeugung automatisch gestartet. Den Prozessen wird beim Start ein Index als Argument übergen. Der aufrufende Prozess wird blockiert wenn *wait* auf *True* gesetzt wird (Verhalten wie *Par* Konstruktor), ansonsten wird er nicht blockiert (Verhalten wie *Fork* Konstruktor)!

```
ProcessArray(function (index) {
  ..
  },
  N,,
  {environment}
  wait?
)
```

IPC Objekte

Channel(*depth*)

Ein Channel stellt synchronisierten Datenaustausch zwischen Prozessen her. Ein Channel mit *depth*=0 (oder kein Argument) implementiert das klassische Rendezvous System: Schreiber wird blockiert bis Leser zugreift und umgekehrt. Ansonsten gibt *depth* die Anzahl der Speicherzellen des FIFOs an. Ein Channel ist polymorph und kann gelesen oder beschrieben werden mit folgenden Operationen:

```
x = channel.read()
channel.write(ε)
```

Semaphore(*init*)

Eine Sempahore stellt Synchronisation in Produzenten-Konsumenten Systemen über einen geschützten Zähler her. Der Startwert *init* des Zählers muss angegeben werden. Der Zähler kann um eins erhöht oder erniedrigt werden.

JAVASCRIPT :: THREADS.JS :: IPC

Es stehen folgende Operationen zur Verfügung:

```
semaphore.up()
semaphore.down()
```

... weitere folgen ...

JAVASCRIPT :: THREADS.JS

Shared Memory

- Es stehen zwei verschiedene geteilte Speicherobjekte zur Verfügung:

Matrix(dims: number [],init?:number|function)

Eine n-dimensionale Float32 Matrix die zwischen parallelen Prozessen geteilt werden kann.
Der Zugriff auf die Zellen der Matrix erfolgt mit den Operationen `m.read(index`$_1$`:number, index`$_2$`:number, ..)` → `number` und `m.write(val:number,index`$_1$`:number, index`$_2$`, ..)`. Es gibt die Möglichkeit die Float32 Matrix mittels der `m.slice(range`$_1$`:(number|number []|null), range`$_2$`, ..)` Operation in ein (nicht teilbares) natives Array zu wandeln. Der Platzhalter `null` oder `undefined` wählt den gesamten Indexbereich der jeweiligen Dimension der Matrix aus.

JAVASCRIPT :: THREADS.JS

Memory(length:number,init?:number|function)

Ein lineares Byte8 Buffer Array das zwischen parallelen Prozessen geteilt werden kann. Der Zugriff auf die Zellen erfolgt wie bei nativen Arrays `b[index:number]`.

- Achtung bei threads.js Version < 1.1.22: Der erzeugende Prozess (main) muss `b.buffer` verwenden (ausgenommen Prozesse in einem `Seq` Konstruktor wo der Buffer über das environment übergeben wird). Alle inneren Prozesse die den Buffer als geteilte Ressource (über das shared environment) nutzen verwenden direkt `b`!

JAVASCRIPT :: THREADS.JS

Beispiele

```
var m = Matrix([1000,1000],Math.random);
var b = Memory(1000000,0);
b[0]=1;
m.write(0,[99,123]);
var msub = m.slice([null,0]); // returns first row as Array
Par([
  function () {
   for(var i=1;i<100;i++)
     mat1.write(mat1.read([i,2])-buf1[i],[i-1,2]);
  },
  function () {
   for(var i=1;i<100;i++)
     mat1.write(mat1.read([i,3])-buf1[i],[i-1,3]);
  }
],{
  mat1:m,
  buf1:b
});
```

SYNCHRONISATION UND KOMMUNIKATION

NEBENLÄUFIGKEIT, WETTBEWERB, UND SYNCHRONISATION

Definition 11.
Nebenläufigkeit mit Konkurrenz. Operationen in verschiedenen Prozessen eines Programms konkurrieren z. B. um gemeinsame Ressourcen → Wettbewerb. Konkurrenz bedeutet optionale Parallelität und kann auch sequenziell ausgeführt werden!

Definition 12.
Parallelität. Parallele Ausführung auf Verarbeitungs- und Systemebene mit zeitlicher Überschneidung von Ereignissen und Operationen ohne zwangsläufig mit Wettbewerb.

- **Konkurrierender Zugriff auf geteilte Ressourcen** wie z. B. Speicher oder Kommunikationskanäle müssen synchronisiert werden, z. B. mit mutualen Ausschluss (Mutex) und zeitlicher Sequenzialisierung → Schutz kritischer Ausführungsbereiche

NEBENLÄUFIGKEIT, WETTBEWERB, UND SYNCHRONISATION

- **Zeitliche ereignisbasierte Synchronisation** z.B. mit Semaphoren oder einfacher mit Events werden in Produzenten-Konsumenten Anwendungen benötigt, z.B.
  - » Bei der Partitionierung und Verteilung von Eingabedaten auf mehrere Prozesse (Produzenten);
  - » Der Annahme und Verarbeitung durch die einzelnen Prozesse (Konsumenten); sowie
  - » Bei der Einsammlung von Ergebnis-/Ausgabedaten.
- **Synchronisation ist Kommunikation** und dient der Bewahrung von Invarianten der Datenverarbeitung (Datenkonsistenz)!
- **Synchronisationsobjekte sind ebenfalls geteilte Ressourcen**!!!!!

DER MUTUALE AUSSCHLUSS: EIN PROBLEM UND SEINE DEFINITION

Definition 13.
Kritischer Bereich. Sequenzielle Ausführung einer Anweisungssequenz $K=\{I_1,I_2,..\}$ in einem parallelen Programm darf nur von einem einzigen Prozess zur gleichen Zeit ausgeführt werden. Ein kritischer Bereich K ist daher eine geschützte Ressource.

Definition 14.
Mutualer Ausschluss. Es muss einen Algorithmus geben der sicher stellt dass immer und maximal nur ein Prozess p_i aus einer Menge von Prozessen $P=\{p_1,p_2,..\}$ den kritischen Bereich K ausführen darf (die Ressource zugeteilt bekommt).

Definition 15.
Wettbewerb. Konkurrieren mehrere Prozesse um die Ressource K, so gewinnt maximal einer, die anderen verlieren den Wettbewerb.

EIGENSCHAFTEN VON PARALLELEN SYSTEMEN

*Ein solches **Wettbewerbsproblem** ist gekennzeichnet durch die Eigenschaften **Sicherheit** und **Lebendigkeit**.*

Sicherheit bedeutet: Bedingung Anzahl der Prozesse im kritischen Bereich K muss $|\{p \mid I(p) \in K\}| \leq 1$ *immer erfüllt sein.*

EIGENSCHAFTEN VON PARALLELEN SYSTEMEN

Eigenschaft (Liveness) Lebendigkeit → Starvation Freiheit

- Starvation Freiheit bedeutet dass ein Prozess p_i der die Ressource/den kritischen Bereich anfragt maximal eine endliche Anzahl von Zeiteinheiten / Wettbewerben $n \neq \infty$ durch jeden anderen Prozess umgangen werden kann (Bypass, haben den Wettbewerb gewonnen).

Eigenschaft Deadlock Freiheit

- Wenn bis zu einem beliebigen Zeitpunkt t eine Reihe von Prozessen versucht haben den kritischen Bereich zu erlangen, ihn aber nicht erhalten haben, zu wird es in der Zukunft eine Zeit $t' > t$ geben, wo sie erfolgreich sind (d.h. die Anfrage-Operation terminiert).
- D.h. eine Anfrage der Ressource terminiert mit einem gewonnen Wettbewerb garantiert irgendwann.
  - » Starvation Freiheit impliziert Deadlock Freiheit!
  - » Worst case eines Deadlocks: alle Prozesse verlieren den Wettbewerb, keiner kann die Ressource mehr nutzen/zugeteilt bekommen!

EIGENSCHAFTEN VON PARALLELEN SYSTEMEN

Eigenschaft Begrenztes (Bounded) Bypass Kriterium

- Es gibt eine Funktion $f(N)$ für N Prozesse die die maximale Anzahl von verlorenen Wettbewerben (bei Konkurrenz) angibt.

Service gegen Klienten Sichtweise

Der Service ist die Ressource, der kritische Bereich. Der Klient ist der Prozess, der die Ressource anfragt.

- Aus Sicht des Service (Ressource) ist die Deadlock Freiheit wichtigstes Kriterium. Der konkurrierende Wettbewerb führt immer dazu, dass irgendein Prozess die Ressource nutzen kann → *die Ressource gewinnt immer.*
- Aus Sicht des Klienten (Prozesses) ist die Starvation Freiheit wichtigstes Kriterium. Wenn immer ein Prozess die Ressource anfragt, wird er sie (eventuell) zugeteilt bekommen → *der Prozess gewinnt immer.*

EIGENSCHAFTEN VON PARALLELEN SYSTEMEN

- Es gibt daher eine Hierarchie der Lebendigkeitseigenschaften (entsprechend ihrer Stärke):

---

1. Bounded Bypass $f(N)=X \Rightarrow$
2. Starvation Freiheit $\equiv$ Endlicher Bypass $f(N) \neq \infty \Rightarrow$
3. Deadlock Freiheit

Deadlock

- Tritt häufig ein wenn zwei (oder mehrere) Prozesse gegenseitig eine Ressource (Lock) beanspruchen die aber jeweils vom anderen bereits belegt ist.
  - » Prozesse, die bereits Sperren halten, fordern neue Sperren an.
  - » Die Anforderungen für neue Sperren werden gleichzeitig gestellt.
  - » Zwei oder mehr Prozesse bilden eine kreisförmige Kette, in der jeder Prozesse auf eine Sperre wartet, die vom nächsten Prozess in der Kette gehalten wird → Dining Philosopher Problem.

EIGENSCHAFTEN VON PARALLELEN SYSTEMEN

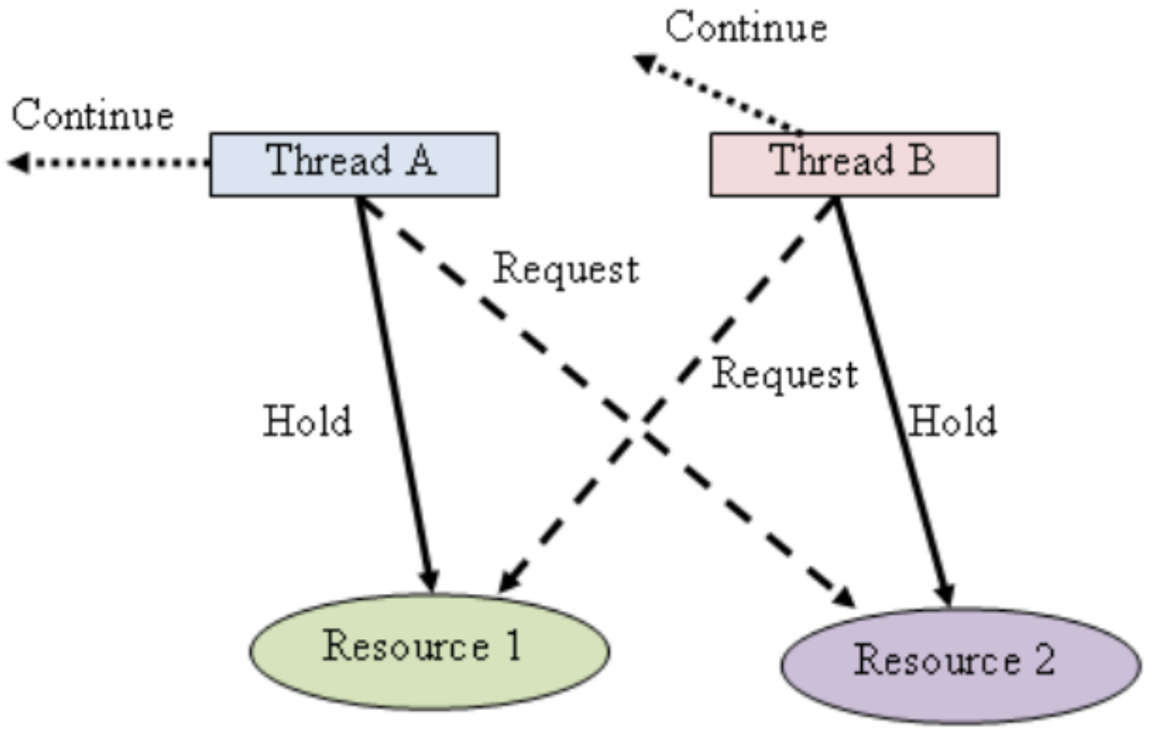


Abb. 30. Deadlocksituation zweier Prozesse die wechselseitig gesperrte Ressourcen anfordern

LOCK UND ATOMARE OPERATIONEN

Mutualer Ausschluss mit LOCK Objekt

- Ein LOCK Objekt ist ein geteiltes Objekt dass sich in zwei verschiedenen Zuständen S_{LOCK}={*FREE*,*LOCKED*} befinden kann und eine Ressource schützt:

  State FREE
  Die Ressource ist nicht in Verwendung

  State LOCKED
  Die Ressource wird bereits von einem Prozess verwendet, andere Prozesse müssen warten
- Ein LOCK Objekt stellt Prozessen zwei Operationen zur Verfügung:

Operation ACQUIRE / LOCK

Ein Prozess kann eine Ressource anfordern. Ist der LOCK im Zustand S_{LOCK}=*FREE*, wird dem anfordernden Prozess die Ressource gewährt (Zustandsübergang S_{LOCK}: *FREE* ⇒ *LOCKED*), andernfalls wird er blockiert bis die Ressource (vom Eigentümer) freigegeben wird.

LOCK UND ATOMARE OPERATIONEN

Operation RELEASE / UNLOCK

Ein Prozess gibt eine zuvor mit ACQUIRE beanspruchte Ressource wieder frei (Zustandsübergang S_{LOCK}: *LOCKED* ⇒ *FREE*).

> *Ein Prozess muss die Operation ACQUIRE und RELEASE immer paarweise verwenden!*

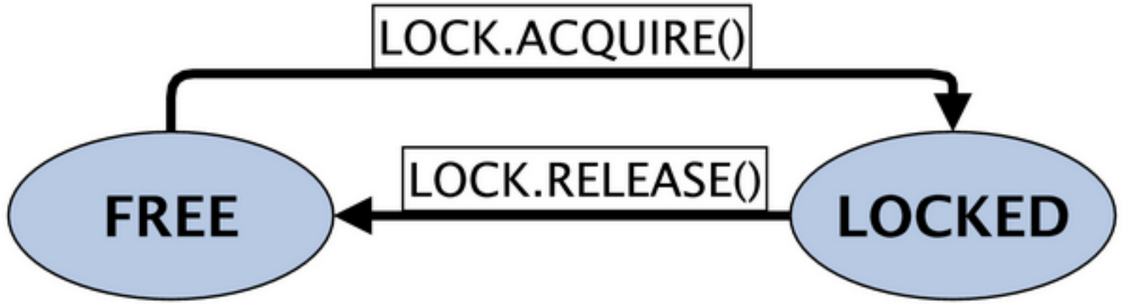


Abb. 31. Zustandsdiagramm des LOCK Objekts

LOCK UND ATOMARE OPERATIONEN

```
OBJECT L: LOCK
VAR V
PAR
  PROCESS P1 IS                          PROCESS P2 IS
  WHILE (True)                           WHILE (True)
    non-critical section                   non-critical section
    L.ACQUIRE()                            L.ACQUIRE()
     critical section(V)                    critical section(V)
    L.RELEASE()                            L.RELEASE()
    non-critical section                   non-critical section
  DONE                                    DONE
```

Abb. 32. Beispiel der Nutzung eines LOCK Objekts für den Schutz von kritischen Bereichen beim Zugriff auf geteilte Ressourcen

- Das LOCK Objekt löst das mutuale Ausschlussproblem (mutual exclusion), und wird daher auch als Mutex bezeichnet.
- Mehr als zwei Prozesse können mit einem LOCK synchronisiert werden.

LOCK UND ATOMARE OPERATIONEN

Synchronisation durch Blockierung

- Ereignisbasierte Synchronisation blockiert die Ausführung von Prozessen bis das Ereignis auf das gewartet wurde eingetreten ist.
  - » Hier die Freigabe eines belegten LOCKs

Atomare Ressourcen: Register

- Ein Lese/Schreib Register ist die einfachste Form einer geteilten Ressource und wird für die Implementierung eine LOCK Objektes benötigt.

Definition 16.
Atomares Register. Der Zugriff auf ein atomares Register R erfolgt mittels zweier Operationen $OP=\{READ,WRITE\}$: $R.READ()$, welches den aktuellen Wert von R zurück liefert, und $R.WRITE(v)$, welche einen neuen Wert in das Register schreibt. Dabei ist ein Konflikt durch konkurrierende Schreib- und Lesezugriffe von einer Menge von Prozessen durch mutualen Ausschluss aufgelöst!

LOCK UND ATOMARE OPERATIONEN

Besipielssituation

```
VAR GLOBAL SHARED: R          (VAR LOCAL NOTSHARED: x)
P1: VAR x     P2: VAR x       P3: VAR x
|x := ε(R)|   |R := ε(x)|     |R := ε(x)|
```

Eigenschaften eines atomaren Registers

Ein atomares geteiltes Register erfüllt dabei folgende Eigenschaften:

- Zugriffe (*op=READ/WRITE*) erscheinen als wären sie zu einem einzigen Zeitpunkt $\tau(op)$ ausgeführt worden.
- Der Zeitpunkt der Ausführung der Operation liegt garantiert innerhalb des Intervalls:
  $\tau(op_S) \leq \tau(op) \leq \tau(op_E)$, mit
  - *S*: Start und *E*: Ende der Operation (gemeint ist: Aktivierung durch Prozess)
- Für zwei verschiedene Operationen op_1 und op_2 gilt:
  $\tau(op_1) \neq \tau(op_2)$

LOCK UND ATOMARE OPERATIONEN

- Jede Lese-Operation gibt den Wert zurück der von der am nächsten (in der Vergangenheit) liegenden Schreibe-Operation resultiert.

> *Das bedeutet: ein atomares Register verhält sich so als ob die Ausführung aller Operationen sequenziell ausgeführt worden wäre.*

Komposition mit atomaren Objekten

- Atomarität erlaubt die einfache Komposition von geteilten (atomaren) Objekten zu neuen geteilten und atomaren Objekten ohne zusätzlichen (Synchronisations-) Aufwand.
- D.h. wenn eine *Menge von für sich atomaren Registern* $\{R_1, R_2, ..\}$ mit den Operationen $\{R_1.READ, R_1.WRITE\}$, $\{R_2.READ, R_3.WRITE\}$, .. zu einem neuen Objekt $(R_1, R_2, ...)$ zusammengefasst (komponiert) werden, so ist dieses ebenfalls *atomar*.

LOCK UND ATOMARE OPERATIONEN

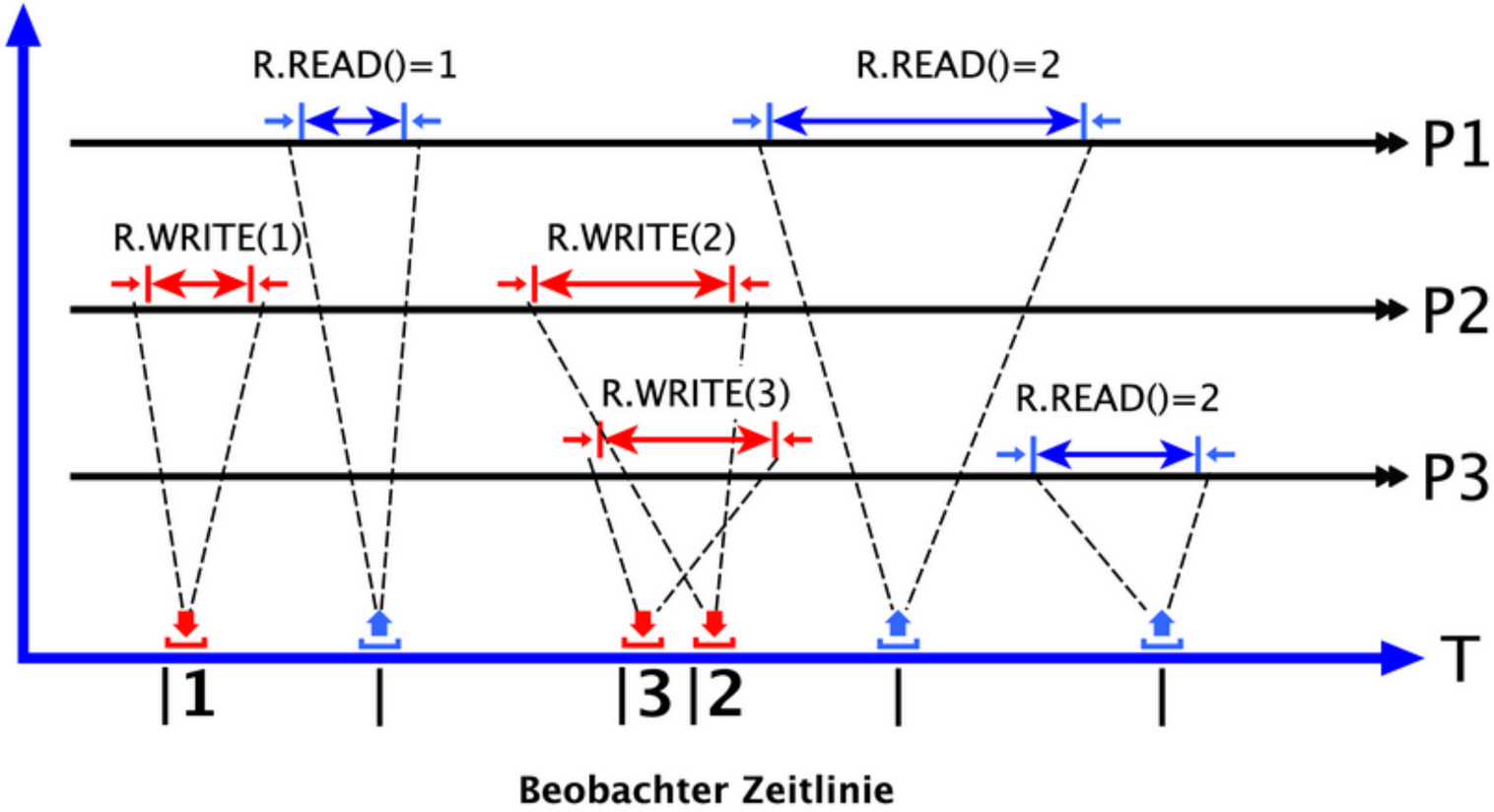


Abb. 33. Beispiel von konkurrierenden und zeitlich überlappenden Schreib/Lese Zugriffen auf ein atomares Register

MUTEX

- Eine Mutex garantiert mittels eines LOCK Objekts den mutualen Ausschluss durch Erfüllung einer **Invariante**, d.h. einer logischen Bedingung die niemals verletzt werden darf.
- Angenommen eine Menge von Prozessen $\{P_1, P_2, ..\}$ wollen durch Mutualen Ausschluss sicher stellen dass immer nur ein Prozess sich im kritischen Bereich CS befindet (p_i=True). Dann gilt für die **Invariante der Mutex**:

$$Mutex: \quad (p_1 \oplus p_2 \oplus .. \oplus p_n \oplus (\neg p_1 \wedge \neg p_2 .. \wedge \neg p_n))$$

- Das Prädikat *Mutex* muss eine globale Invariante sein, die mit atomaren Registern implementiert werden kann.
- Die Blockierung eines oder mehrere Prozesse beruht prinzipiell auf einer **atomaren Schleife** die auf das Eintreten einer booleschen Bedingung wartet, z.B. in der Form
  |`await` (*cond*) `then` (*action*)|
  - » Die Anweisung blockiert den Prozess solange *cond* nicht wahr ist und dann unmittelbar (atomar) die Anweisung *action* ausführt (Datenzuweisung) → **Test-and-set**

MUTEX

Protokolle

- Es muss in jedem Prozess der einen kritischen Bereich ausführen muss ein Eingangs- und Ausgangsprotokoll eingehalten werden damit der mutuale Ausschluss garantiert werden kann.
- Für die Implementierung dieses Protokolls soll gelten:

1. Mutualer Ausschluss und Einhaltung der LOCK Invariante (logische Bedingung): Höchstens ein Prozess zur Zeit führt seinen kritischen Abschnitt aus.
2. Keine Deadlocks und Livelocks: Wenn zwei oder mehr Prozesse versuchen, ihre kritischen Abschnitte zu betreten, wird mindestens einer erfolgreich sein.
3. Niedrige Latenz (niedrige Verzögerung): Wenn ein Prozess versucht, in seinen kritischen Abschnitt einzutreten, und die anderen Prozesse ihre nicht kritischen Abschnitte ausführen oder beendet haben, wird der erste Prozess nicht daran gehindert, in seinen kritischen Abschnitt einzutreten.
4. Garantierter Eintritt: Ein Prozess, der versucht, in seinen kritischen Abschnitt einzutreten, wird schließlich erfolgreich sein.

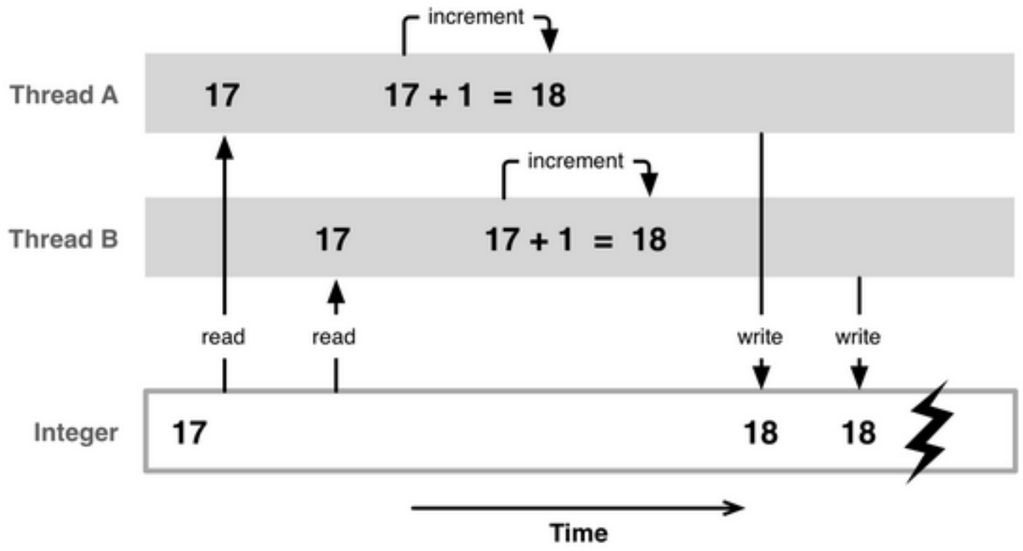
MUTEX

Kritisches Abschnittsproblem: Grobe Lösung A.

```
bool in1 = false, in2 = false;
## MUTEX: not(inl and in2) -- global invariant
process CSl {
  while (true) {
    |await (!in2) then in1 = true|; /* entry */
    critical section
    |in1 = false|;
    noncritical section
    /* exit */
  }
}
process CS2 {
  while (true) {
    |await (!in1) then in2 = true|; /* entry */
    critical section
    |in2 = false|;
    noncritical section
    /* exit */
  }
}
```

MUTEX

- Die Lösung A. garantiert den mutualen Ausschluss aber nicht die Lebendigkeit einzelner Prozesse.
  - » Es können Wettlaufbedingungen eintreten (race conditions), wo immer nur einer oder einige wenige Prozesse den Wettbewerb gewinnen



- Ein fairer Scheduler ist erforderlich um Wettlaufbedingungen zu vermeiden!

MUTEX

Kritisches Abschnittsproblem: Grobe Lösung B. mit Spin Locks

- Benötigt nur noch eine Variable → geringere Latenz!

```
bool lock = false;
process CS1 {
  while (true) {
    |await (!lock) then lock = true|;  /* entry */
    critical section
    lock = false;
    noncritical section;
    /* exit */
  }
}
process CS2 {
  while (true) {
    |await (!lock) then lock = true|; /* entry */
    critical section
    lock = false;
    noncritical section
    /* exit */
  }
}
```

MUTEX

- Da es nur zwei Zustände der Mutex gibt reicht eine Variable für die die Invariante gilt:

$$lock = in_1 \vee in_2$$

- Weiterer Vorteil der Spin-Lock Lösung: Sie kann für beliebige N>2 Prozesssysteme angewendet werden.

Test-and-set Operation

- Die atomare *await-then* Anweisung ist prinzipiell eine atomare *Test-and-set* (*TS*) Operation, wie sie von vielen Mikroprozessoren als eigenständiger Maschinenbefehl zur Verfügung gestellt wird.

```
bool TS (bool lock) {
  |bool initial = lock;
   lock = true;
   return initial|
}
```

MUTEX

- Die bedingten *atomaren Aktionen* werdend ann durch Schleifen ersetzt, die erst dann enden, wenn die Sperre falsch ist, und TS gibt daher false zurück.
- Da alle Prozesse dieselben Protokolle ausführen, funktioniert die gezeigte Lösung für eine beliebige Anzahl von Prozessen.
- Wenn eine Sperrvariable als ein Spin Lock verwendet wird, heisst das, dass die Prozesse während des Wartens auf das Löschen der Sperre eine Schleife durchlaufen (rotieren).
- Gegenseitiger Ausschluss ist sichergestellt:
  - » Wenn zwei oder mehr Prozesse versuchen, in ihren kritischen Abschnitt einzutreten, wird nur einer erfolgreich sein, der erste zu sein, der den Wert von lock von falsch auf wahr ändert
  - » Daher beendet nur einer sein Eintrittsprotokoll.

MUTEX

- Die vorherigen Algorithmen können dann für *N* Prozesse mit einer *TS* Operation und einer bedingten Schleife implementiert werden, in der Art:

```
bool lock = false;
process CS[i = 1 to n] {
  while (true) {
    while (TS(lock)) skip;
    critical section
    lock = false;
    noncritical section
  }
}
```

- Es wird *keine atomare Schleife* mehr benötigt!
- Aber auch hier ist der Eintritt (Lebendigkeit einzelner Prozesse) nicht garantiert. Warum?
- Weiterhin zeigt obiger Algorithmus bei Multiprozessorsystemen eine schlechte Performanz → Speicherzugriffe.

MUTEX

Test-Test-and-Set Lösung

- Besser eine geschachtelte und verkettete Kombination aus Test und Test-and-Set:

```
bool lock = false;
process CS[i = 1 to n] {
  while (true) {
    while (lock) skip;
    while (TS(lock)) { while (lock) skip};
    critical section
    lock = false;
    noncritical section
  }
}
```

- Diese Variante verbesserte den Speicherzugriff über die Cacheebene (probalistisch!)

MUTEX

Faire Lösung mit dem Tie-Breaker Algorithmus

- Faires Scheduling durch "Fahrstuhlalgorithmus" und Iteration über mehrere (n-1) Ebenen:

```
int in[1:n] = ([n] 0), last[1:n] = ([n] 0);
process CS[i = 1 to n] {
while (true) {
  for [j = 1 to n] { /* entry protocol */
    /* remember process i is in stage j and is last */
    last[j] = i; in[i] = j;
    for [k = 1 to n st i != k] {
      /* wait if process k is in higher numbered stage and
         process i was the last to enter stage j */
      while (in[k] >= in[i] and last[j] == i) skip;
    }
  }
  critical section
  in[i] = 0;
  noncritical section
 }
}
```

MUTEX

Peterson-Algorithmus für das LOCK Objekt und zwei Prozessen

- Algorithmus für die Operationen {*ACQUIRE*,*RELEASE*} des LOCK Objekts, der mutualen Ausschluss garantiert und Wettbewerb um eine geschützte Ressource ermöglicht.
- Der Peterson Algorithmus besteht aus zwei Komponenten, die unterschiedliche Eigenschaften des LOCK erfüllen.
- Erfüllung des mutualen Ausschlusses

```
OPERATION ACQUIREA(i) IS
  AFTER_YOU := i
  WAIT UNTIL AFTER_YOU /= i
  RETURN
END OPERATION
```

```
OPERATION RELEASEA(i) IS
   RETURN
 END OPERATION
```

MUTEX

- Dieser Teil garantiert den mutualen Ausschluss und löst das Wettbewerbsproblem, d. h. maximal ein Prozess *x* von zwei Prozessen erlangt die Ressource und die Operation *ACQUIRE*(*x*) terminiert, der Operation *ACQUIRE*(*y*) des anderen Prozesses *y* terminiert nicht.
- Nachteil: Rendezvous ist erforderlich. Beide Prozesse müssen die Ressource anfordern, d.h. ein einzelner Prozess verliert seinen eigenen Wettbewerb immer.
- Erfüllung des Lebendigkeitskriteriums

```
OPERATION ACQUIREB(i) IS
  FLAG[i] := UP
  WAIT UNTIL FLAG[j] = DOWN
  RETURN
END OPERATION
```

```
OPERATION RELEASEB(i) IS
   FLAG[i] := DOWN
END OPERATION
```

- Dieser Teil löst das Problem, wenn nur ein Prozess zu gleichen Zeit an der Ressource interessiert ist, und die *ACQUIRE*(*i*) Operation terminieren kann.

MUTEX

- Nachteil: Deadlock wenn beide Prozesse gleichzeitig *ACQUIRE* Operation durchführen. (beide setzen gleichzeitig ihr FLAG auf *UP*).
- Zusammenführung beider Teilalgorithmen 1. und 2. führt zum Peterson Algorithmus für die Implementierung der Operationen eines LOCK Objekts (Zwei Prozesse *i* und *j*).

```
OBJECT LOCK IS
REG FLAG[2],AFTER_YOU
OPERATION ACQUIRE(i) IS
  FLAG[i] := UP
  AFTER_YOU := i
  WAIT UNTIL FLAG[j] = DOWN OR
       AFTER_YOU != i
  RETURN
END OPERATION
```

```
OPERATION RELEASE(i) IS
   FLAG[i] := DOWN
   RETURN
END OPERATION
```

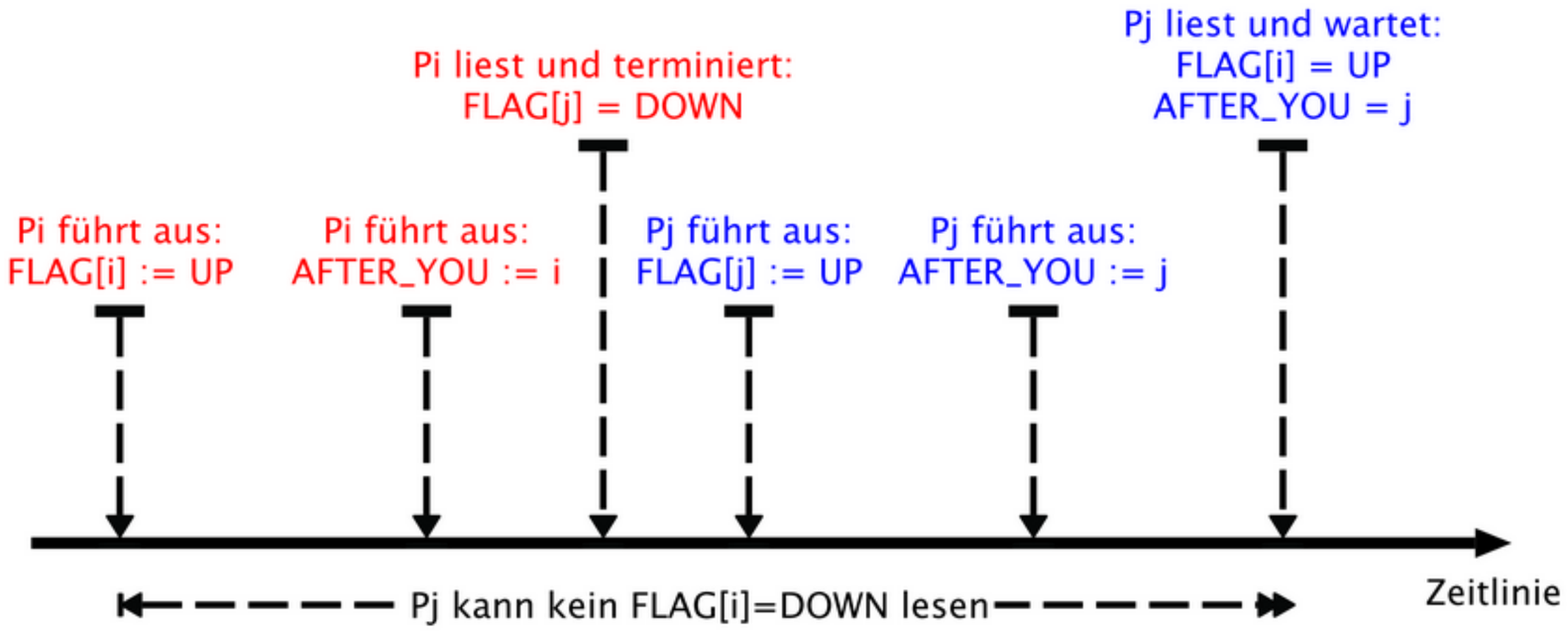
MUTEX

- FLAG zeigt das Interesse an einer Ressource an mit Wertemenge {*DOWN*, *UP*}.
- AFTER_YOU besitzt die Wertemenge {1,2} und bezeichnet die Prozess-nummer.
- FLAG und AFTER_YOU sind atomare Register (Single Write Multiple Read Verhalten).
- Dieser Algorithmus erfüllt
  - Mutuale Exklusion (max. ein Prozess erhält die Ressource)
  - Bounded Bypass (ein Prozess gewinnt den Wettbewerb) mit $f(n)=1$

MUTEX

Mutual Exklusion Eigenschaft (I)

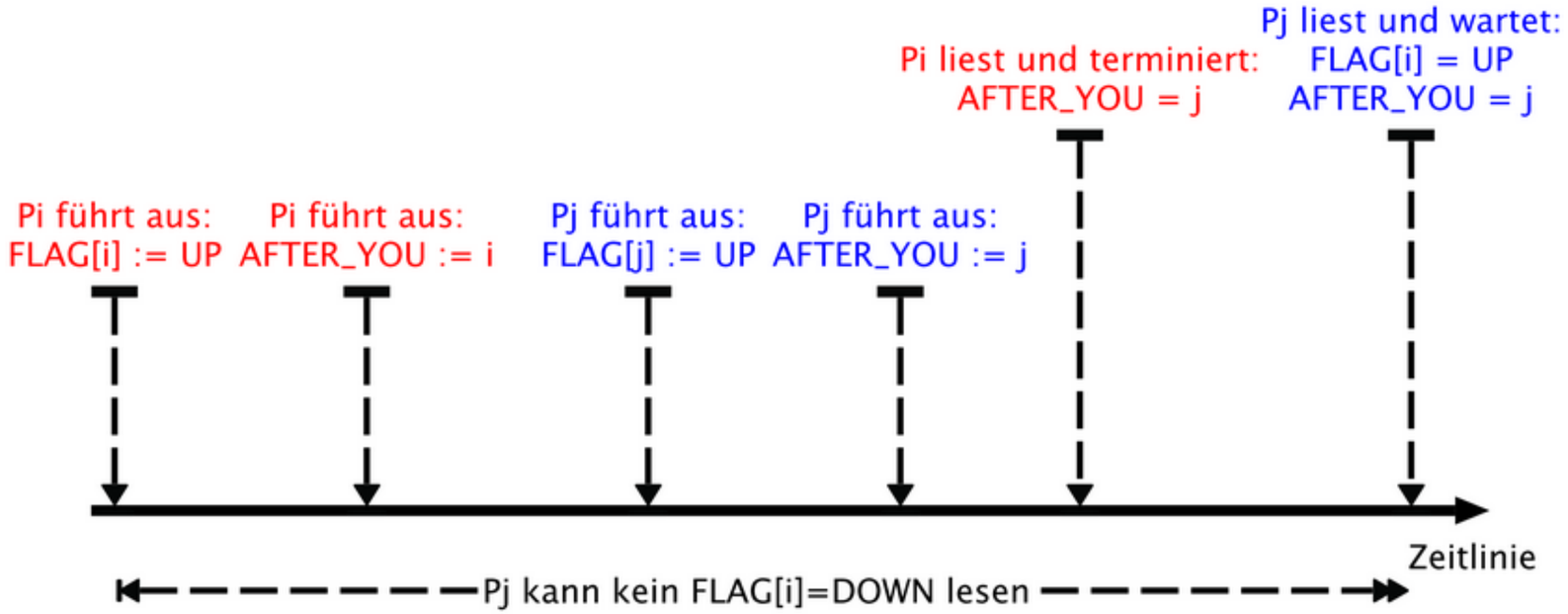
- Der erste Prozess (*i*) führt FLAG[*i*] := *UP* und AFTER_YOU := *i aus
- Der erste Prozess (*i*) terminiert durch FLAG[*j*]=*DOWN*
- Der zweite Prozess (*j*) führt FLAG[*j*] := *UP* und AFTER_YOU := *j* aus
- Der zweite Prozess (*j*) wartet



MUTEX

Mutual Exklusion Eigenschaft (II)

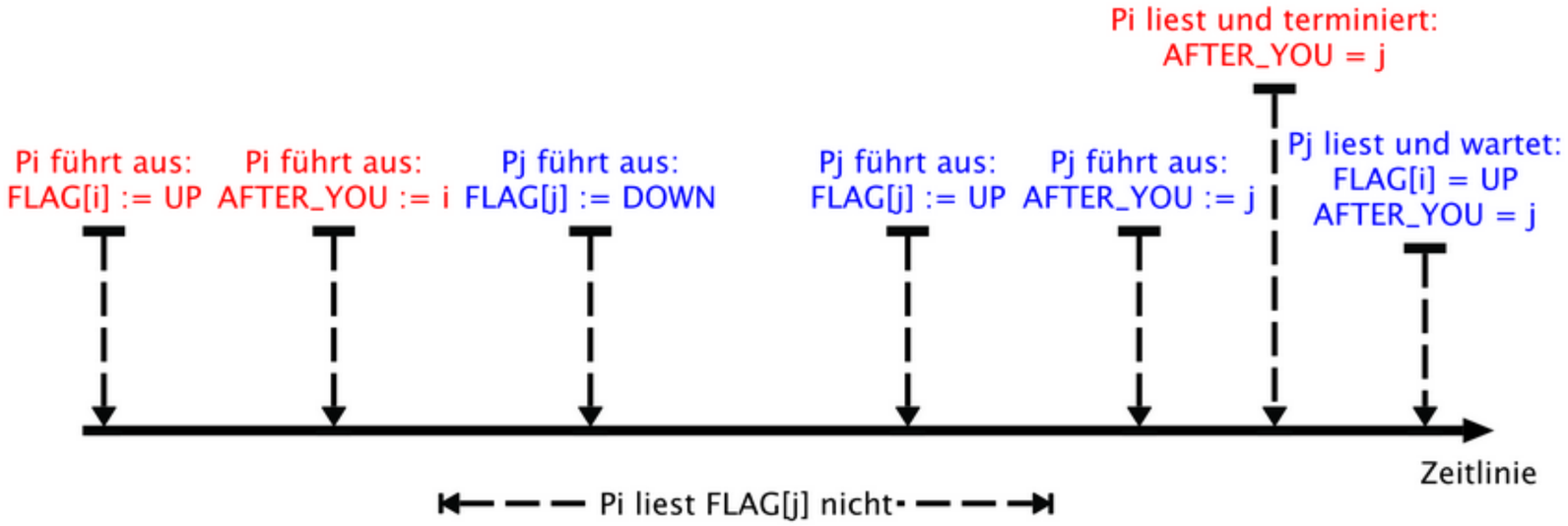
- Der erste Prozess (*i*) führt FLAG[*i*] := UP und AFTER_YOU := *i* aus
- Der zweite Prozess (*j*) führt FLAG[*j*] := UP und AFTER_YOU := *j* aus
- Der erste Prozess (*i*) terminiert durch AFTER_YOU = *j*
- Der zweite Prozess (*j*) wartet



MUTEX

Bounded Bypass

- Ein Prozess (*j*) besitzt die Ressource (FLAG[j] := UP)
- Der andere Prozess (*i*) führt FLAG[i] := UP und AFTER_YOU := *i* aus
- Eine zeit lang liest Prozess (*i*) FLAG[*j*] nicht.
- Während dessen gibt Prozess (*j*) die Ressource frei und führt FLAG[*j*] := *DOWN* aus.
- Danach beantragt Prozess (*j*) die Ressource erneut, und führt FLAG[*j*] := *UP* und AFTER_YOU := *j* aus.
- Der Prozess (*i*) liest AFTER_YOU=*j* und terminiert, Prozess (*j*) liest FLAG[*i*] = *UP* und wartet, d.h. $f(n)=1$!



MUTEX

Peterson Algorithmus für N Prozesse

```
OBJECT LOCK IS
REG FLAG_LEVEL[N],AFTER_YOU[N]
OPERATION ACQUIRE(i) IS
  FOR l = 1 TO (N-1) DO
    FLAG_LEVEL[i] := l
    AFTER_YOU[l] := i
    WAIT (∀ k != i:
       FLAG_LEVEL[k] < l) OR
       AFTER_YOU[l] != i
  RETURN
END OPERATION
```

```
OPERATION RELEASE(i) IS
  FLAG_LEVEL[i] := 0
  RETURN
END OPERATION
```

- Der N=2 Peterson Algorithmus erfüllt Deadlock Freiheit und starken Bounded Bypass!
- Aber der N>2 Algorithmus erfüllt nur Deadlock Freiheit und schwachen Finite Bypass!

MUTEX

Tournament tree

- Ein einfaches Prinzip, um die Anzahl der Shared Memory-Zugriffe zu reduzieren, ist die Verwendung eines Turnierbaumes. Ein solcher Baum ist ein vollständiger Binärbaum.

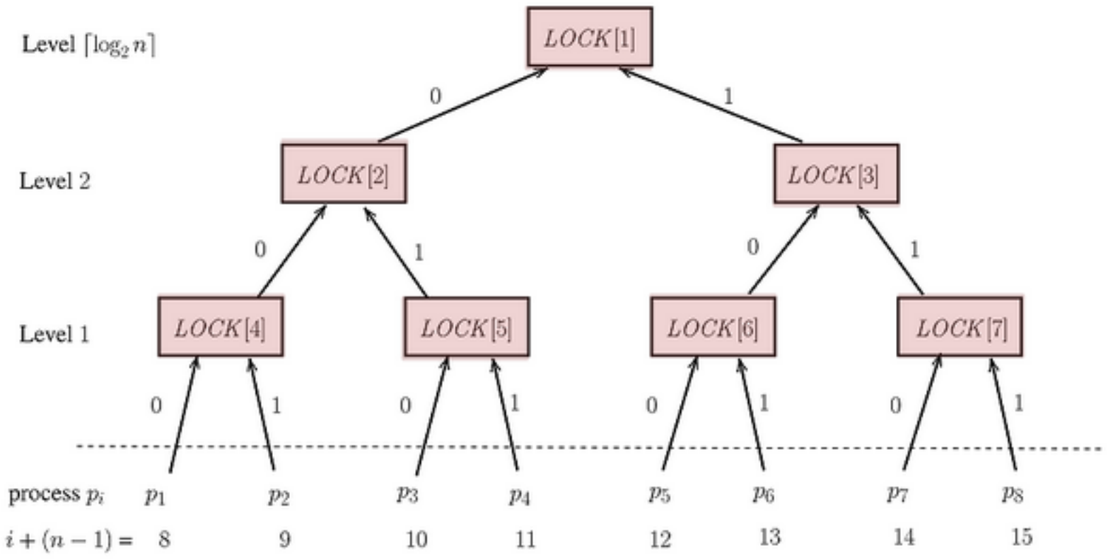


Abb. 34. N Prozesse führen Ebenenweise einen Lock jeweils mit einer Zweiprozessmutex durch bis sie den finalen Wettbewerb gewonnen haben. Die Zweiprozessmutex wird nur von zwei Prozessen geteilt und ist keine globale Ressource mehr!

SEMAPHORE

Eigenschaften

- Ein Semaphore Objekt *S* ist ein LOCK basiertes Synchronisationsobjekt und ein **geteilter Zähler mit Warteschlange** *S.counter*, das folgende Eigenschaften erfüllt:
  - » Die Bedingung *S.counter* ≥ 0 ist immer erfüllt.
  - » Der Semaphorenzähler wird mit einem nicht-negativen Wert s_0 initialisiert.
  - » Es gibt zwei wesentliche Operationen `{S.DOWN, S.UP}` um den konkurrierenden und synchronisierenden Zugriff auf eine Semaphore zu ermöglichen.
  - » Wenn *#(S.DOWN)* und *#(S.UP)* die Anzahl der Operationsaufrufe ist, die terminiert sind, so gilt die Invariante:

$$S.counter = s0 + \#(S.UP) - \#(S.DOWN)$$

SEMAPHORE

Operationale Semantik

Operation WAIT/DOWN

S.DOWN erniedrigt den Zähler *S.counter* atomar um den Wert 1 sofern *S.counter* > *0*.
Wenn *S.counter*=*1* und mehrere Prozesse versuchen gleichzeitig die *DOWN* Operation anzuwenden, so wird einer den Wettbewerb gewinnen (Mutualer Ausschluss = LOCK). Wenn *S.counter* = *0* dann werden Prozesse blockiert und in einer Warteschlange *S.Q* eingereiht.

Operation SIGNAL/UP

S.UP erhöht den Zähler *S.counter* atomar um den Wert 1. Diese Operation kann immer ausgeführt werden, und blockiert nicht die Ausführung des aufrufenden Prozesses.
Wenn *S.counter* = *0* ist und es blockierte Prozesse gibt, wird einer aus *S.Q* ausgewählt und freigegeben (d. h. der Zähler ändert sich effektiv nicht).

SEMAPHORE

*Eine Semaphore mit s_0=1 ist eine **binäre Semaphore** vergleichbar mit einer **Mutex**! Eine Semaphore ist stark wenn die blockierten (wartenden) Prozesse in der Reihenfolge wieder freigegeben werden in der sie blockiert wurden (First In First Out / FIFO Ordnung).*

Algorithmus

```
OBJECT SEMAPHORE(init) IS
  VAR counter:INT
  OBJECT Q:QUEUE, L:LOCK
  OPERATION down(i) IS
    L.lock()
    IF counter = 0 THEN
      Q := Q + {i}
      |L.unlock(),BLOCK(i)|
    ELSE
      counter := counter - 1
      L.unlock()
    END IF
    RETURN
  END OPERATION
```

```
OPERATION up(i) IS
  L.lock()
  IF counter = 0 THEN
    IF Q = {} THEN
      counter := counter + 1
    ELSE
      j := Head(Q), Q := Tail(Q)
      UNBLOCK(j)
    END IF
  ELSE
    counter := counter + 1
  END IF
  L.unlock()
  RETURN
END OPERATION
```

SEMAPHORE - PRODUZENTEN-KONSUMENTEN SYSTEME

- Semaphoren können eingesetzt werden um konkurrierende Produzenten-Konsumenten Probleme zu lösen. Beispiel ist ein Pufferspeicher mit First-In First-Out Reihenfolge, der eine endliche Anzahl von Speicherzellen eines geschützten Speichers *Buf*[0..*k*-1] mit *k*=*size* besitzt, und die beiden Zustände *S*={*FREE*, *FULL*} annehmen kann.
  - » Schutz des Speichers bedeutet: gleichzeitiger Zugriff zweier Prozesse auf gleiche Speicherzelle *Buf*[*x*] muss verhindert werden.
  - » Schutz durch zwei Semaphoren *FREE*(*k*) und *BUSY*(0).

Implementierung eines Pufferspeichers mit Semaphoren

```
OBJECT BUFFER (size) IS
  VAR BUF: datatype[size], in,out: INT
  OBJECT FREE: SEMA(size),
         BUSY:SEMA(0)
  OPEARTION produce(v) IS
    FREE.down()
    BUF[in].write(v)
    in := (in+1) mod size
    BUSY.up()
  END OPERATION
  OPERATION consume IS
    BUSY.down()
    r := BUF[out].read()
    out := (out+1) mod size
    FREE.up()
    RETURN r
  END OPERATION
END OBJECT
```

SEMAPHORE - PRODUZENTEN-KONSUMENTEN SYSTEME

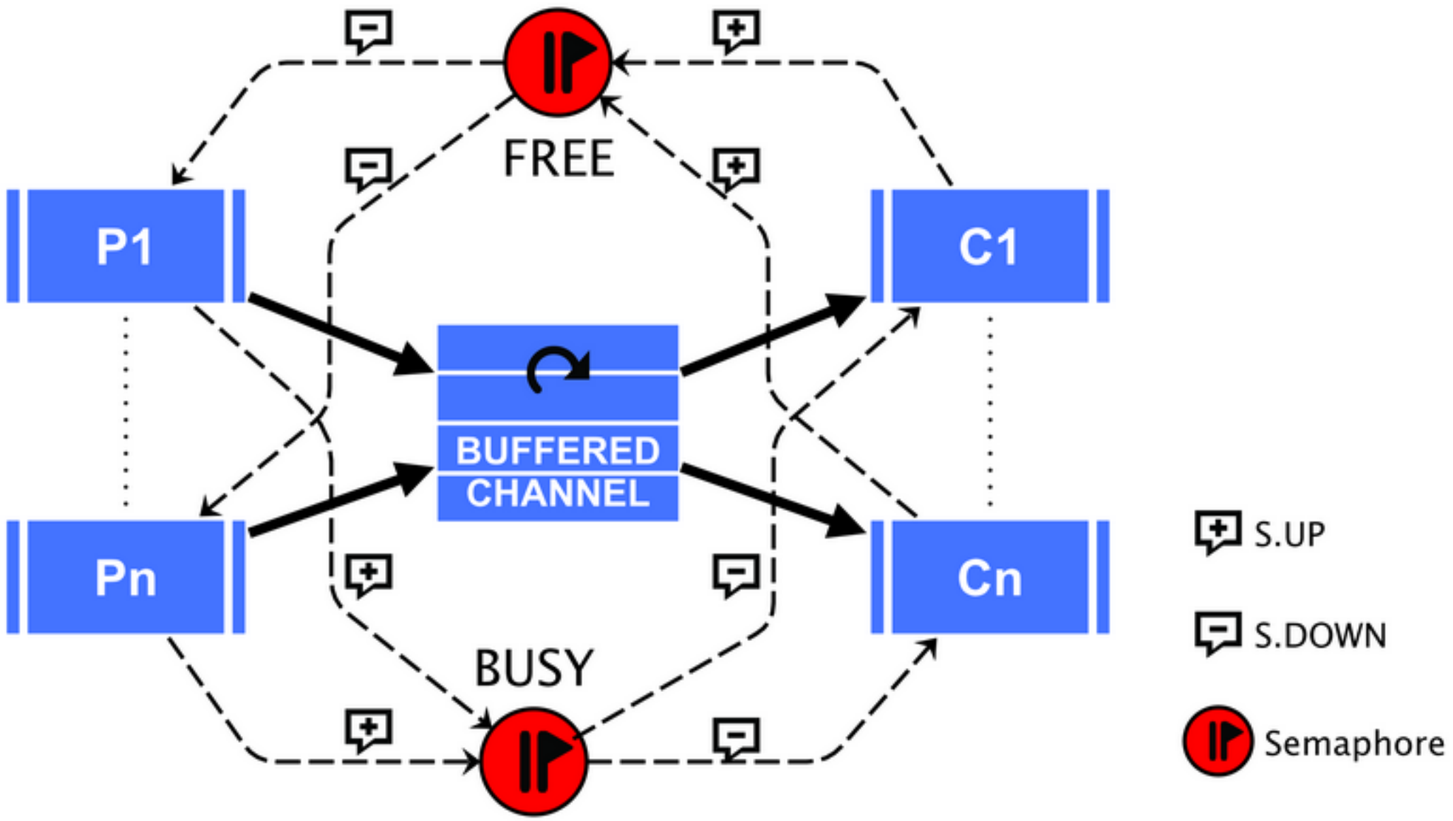
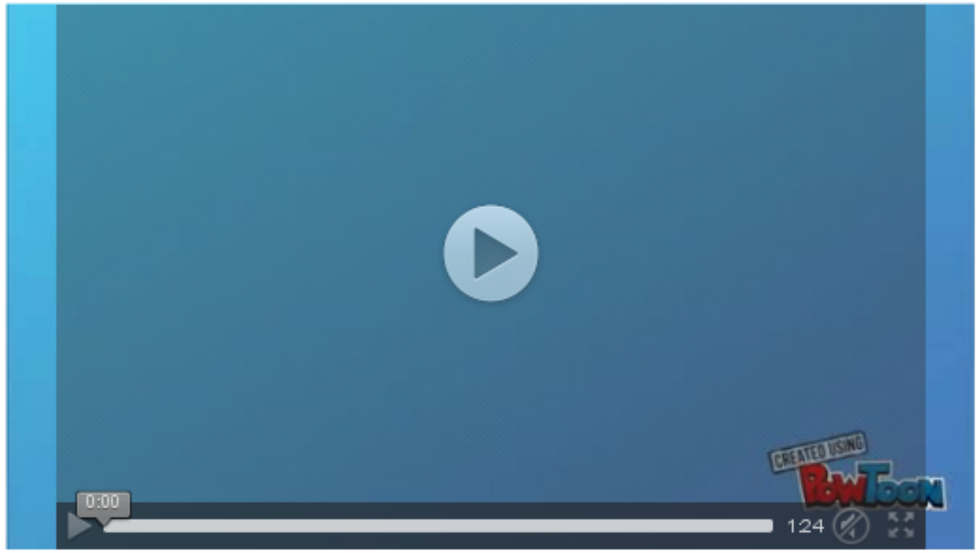


Abb. 35. Synchronisation von Produzenten und Konsumenten von Daten mittels zweier Semaphoren

SEMAPHORE - DINING PHILOSOPHERS

- Das Problem der fünf dinierenden Philosophen ist ein Beispiel für Deadlocks in verteilten und parallelen (asynchronen) Systemen mit dem Communicating Sequential Processes Modell!



- 5 Philosophen an einem Tisch
- 5 Gabeln, jeweils eine zwischen zwei Philosophen
- Ein Philosoph braucht zwei Gabeln zum Essen!
- Eine Gabel wird durch eine Semaphore repräsentiert (atomare Ressource, Startwert des Zählers ist 1)

SEMAPHORE - DINING PHILOSOPHERS

Algorithmus

```
OBJECT ARRAY Forks [N]: SEMA(1)
PROCESS ARRAY Philosopher [N]
  FOR try = 1 TO 10 DO
    THINKING ...
    Forks[#id].down()
    Forks[#id+1 % N].down()
    EATING ...
    Forks[#id+1 % N].up()
    Forks[#id].up()
  DONE
END
```

SEMAPHORE - DINING PHILOSOPHERS

- Klassisches Problems das zu Deadlocks führt!

Deadlockefreie Lösungen

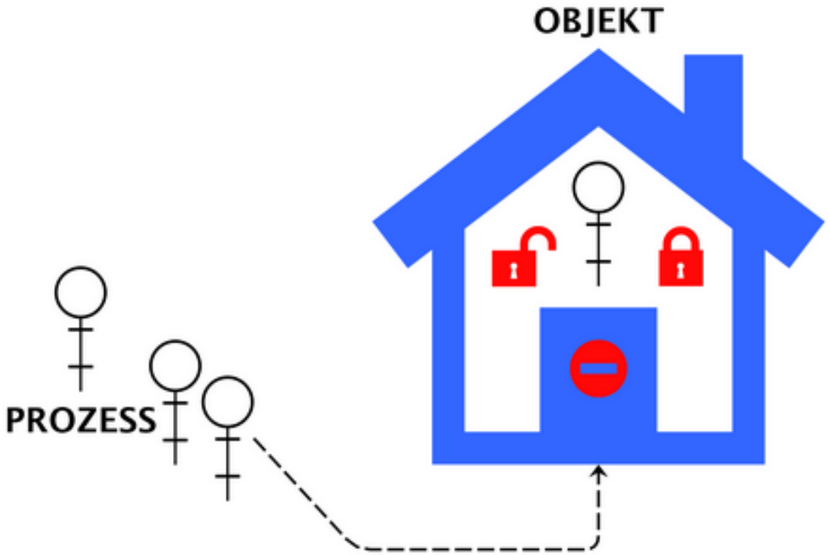
1. Einführung einer zentralen Instanz die die Gabeln mittels Prioritäswarteschlangen verwaltet
   - » Prozess fragt mit einer Nachricht zwei Gabeln an (Input Channel)
   - » Verwalter prüft Verfügbarkeit und gewährt die Gabeln durch eine Antwortnachricht (Output Channel)
   - » Ein Prozess gibt Gabeln mit einer Nachricht wieder zurück
2. Abfrage des Semaphorenwertes beider Gabeln bevor die Ressource angefragt wird ggf. über eine globale Mutex geschützt:

```
sema[left].down();
if (sema[right].level() == 1) sema[right].down();
else sema[left].up(),sleep(random time);
```

3. Münzwurf und Reihenfolge des Zugriffs: Eine Zufallszahl zwischen 0..1 entscheidet ob zuerst die linke und dann rechte Gabel beansprucht wird oder umgekehrt.

MONITOR

- Mutex und Semaphore sind low-level Synchronisationsobjekte
- Monitore erlauben die Definition von konkurrierenden Objekten auf der Abstraktionsebene der imperativen Programmiersprachen → high-level.
- **Ein Monitor ist eine Generalisierung eines Objekts, welches Daten und Operationen kapselt. Zugriff auf die interne Repräsentation erfolgt durch die Operationen mutual exklusiv.**



MONITOR

Konzepte

Mutualer Ausschluss

Der Monitor stellt sicher dass immer nur ein Prozess eine Operation des Objektes ausführen kann à geschützter kritischer Ausführungsbereich = mutualer Ausschluss.

> *Ein Prozess muss das Schloss des Monitors öffnen. Nachdem der Prozess den Monitor "betritt" (nutzt), schließt das Schloss automatisch wieder. Wenn der Prozess den Monitor verlässt wird das Schloss wieder automatisch geöffnet.*

Events: Bedingungsvariablen mit Queues

- Bedingungsvariablen C sind Objekte die interne Synchronisation ermöglichen sollen.
- Bedingungsvariablen können nur innerhalb des Monitors durch folgende Operationen verwendet werden:

MONITOR

Operation *C.wait(cond)*

Diese Operation blockiert = stoppt den aufrufenden Prozess p und reiht ihn in die Warteschlange $C.Q \leftarrow C.Q + \{p\}$ ein. Für den Prozess der nicht mehr aktiv ist, kann der Mutex LOCK aufgehoben werden.

» Die Wait Operation stellte eine boolesche Bedingung *cond* dar auf deren Erfüllung gewartet wird.

Operation *C.signal()*

Ein Prozess p ruft diese Operation auf und erfüllt damit die Bedingung *cond*.

» Wenn $C.Q = \emptyset$ ist dann hat die Operationen keinen Effekt

» Wenn $C.Q = \{q,\ldots\}$ dann wird ein Prozess q aus Q ausgewählt und aktiviert, so dass $C.Q \leftarrow C.Q \setminus \{q\}$

Bewahrung des mutualen Ausschlusses in Fall ii.) erforderlich:

- Der aktivierte Prozess q fährt innerhalb des Monitors fort (bekommt den Eintritt wieder) und führt die Anweisungen nach *C.wait()* aus.

MONITOR

- Der aktivierende Prozess *p* wird blockiert, hat aber höchste Priorität den Monitor danach wieder zu erlangen um die Anweisungen nach *C.signal*() auszuführen.

Operation *C.empty*()

Liefert das Ergebnis für die Bedingung $C.Q = \emptyset$

```
OBJECT CONDITION
  (M:MONITOR) IS
  OBJECT q: QUEUE
     p: PROCESS
  OPERATION wait IS
    p := MYSELF()
    q := q + {p}
    |BLOCK(p) & M.l.unlock()|
  END OPERATION
```

```
OPERATION signal IS
  IF q != {} THEN
    p := Head(q), q := Tail(q)
   UNBLOCK(p)
  END IF
 END OPERATION
```

Abb. 36. Algorithmik der Bedingungsvariablen (vereinfacht)

MONITOR

```
MONITOR SEM IS
  VAR s:INT
  OBJECT c: CONDITION
  OPERATION down IS          OPERATION up IS
    IF s=0 THEN c.wait()       s := s + 1
      s := s - 1                c.signal();
    RETURN                      RETURN
  END OPERATION              END OPERATION
```

Abb. 37. Implementierung einer Semaphore mit Bedingungsvariablen (vereinfacht)

- Um den mutualen Ausschluss zu garantieren muss Vorrang der Prozessausführung und Aktionen mit Bedingungsvariablen definiert werden.
  - » Wartende bereite Prozesse (Menge $\mathbb{W}$) deren Bedingung *cond* erfüllt ist.
  - » Signalisierende Prozesse (Menge $\mathbb{S}$)
  - » Prozesse die noch keinen Eintritt in den Monitor erhalten haben (blockiert, Menge $\mathbb{E}$)
  - » Prioritäten: Prio($\mathbb{W}$) > Prio($\mathbb{S}$) > Prio($\mathbb{E}$)

MONITOR

Ein signalisierter (blockierter) Prozess der auf eine Bedingung C wartet wird sofort aktiviert (d. h. die Bedingung ist erfüllt).

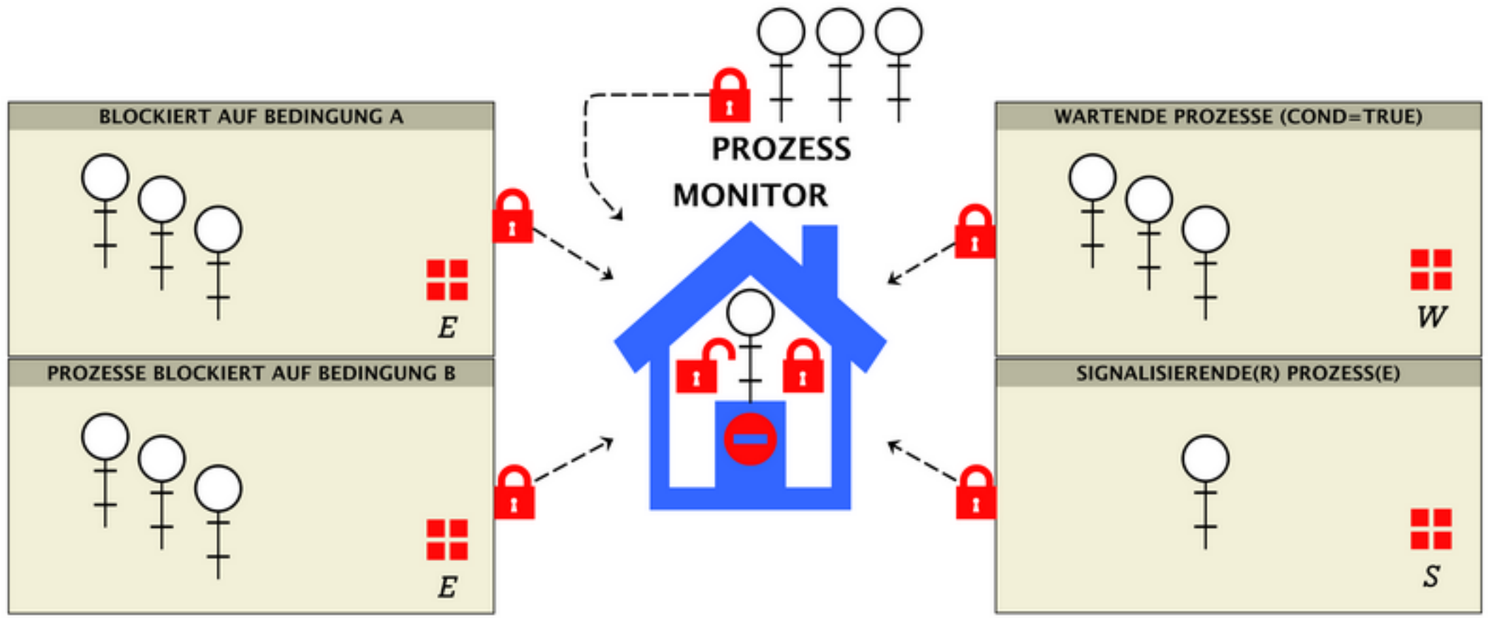


Abb. 38. Verschiedene Prozessmengen bei Monitoren

MONITOR

- Würde nach Aktivierung des wartenden Prozesses (*cond*=TRUE) auch der signalisierende Prozess im Monitor aktiv sein könnte er die Bedingung, die der Bedingungsvariablen zu Grunde liegt, wieder ungültig (*cond*=FALSE) machen und müsste ebenfalls wieder blockiert werden.
- Es gilt daher für den Monitor die Erfüllung eines Prädikats *P* (boolesche Bedingung), das die Datenkonsistenz des Objektes sicher stellen muss.

Varianten

signal-and-wait

Der signalisierende Prozess wartet nach der Signalisierung und der mutuale Ausschluss ist sichergestellt und die Wahrung *P*=TRUE. Der wartende Prozess führt aus:
IF ¬*P* THEN *C.wait*() END IF

signal-and-continue (re-check)

Der signalisierende Prozess läuft weiter! Das könnte zur Verletzung des Prädikats *P* führen, daher muss der signalisierte Prozess nochmals die Gültigkeit *P*=TRUE prüfen so dass für ihn auszuführen ist:
WHILE ¬*P* DO *C.wait* () END WHILE

EVENT

- Mit signal-and-continue Monitoren und Bedingungsvariablen lassen sich einfache temporale ereignisbasierte Synchronisationsobjekte implementieren.
- Mehrere Prozess $p_1, p_2, \ldots, p_{N-1}$ können auf ein Event warten (OP *E.await*). Eine geordnete Warteschlange ist nicht erforderlich.
- Dieses Event wird durch einen weiteren Prozess p_N ausgelöst (OP *E.wake-up*), so dass alle blockierten inklusive des signalisierenden Prozesse in der Ausführung fortfahren (auch zeitgleich).

Operationen

OPERATION await

Ein Prozess *p* wird blockiert bis das Ereignis (event) eintritt

OPERATION wakeup

Die Operation aktiviert alle auf das Ereignis wartenden Prozesse $\{p_i\}$.

EVENT

```
MONITOR EVENT IS
  OBJECT c: CONDITION
  OPERATION await IS     OPERATION wakeup
    c.wait()                c.signal();
    RETURN                  RETURN
  END OPERATION          END OPERATION
```

Abb. 39. Algorithmik des Event Objekts

Beispiel in JavaScript

```
var ev  = Event();
var mem = Memory(16000);

ProcessArray(function () {
  var sum=0;
  ev.await();
  for(var i=0;i<mem.length;i++) sum += mem[i];
  print(sum);
},N,{waitForData: ev, data:mem, N:Value(N)});
for(var i=0;i<mem.length;i++) mem[i]=Math.random()*256;
ev.wakeup();
```

BARRIERE

- Eine Barriere ist ein Rendezvous Objekt, oder auch ein selbst-auslösendes Event. Es gibt eine (vorher) bekannte Gruppe aus N Prozessen $\{p_1, p_2, \ldots, p_N\}$
- Jeder beliebige Prozess $i = 1..N-1$ nimmt an der Barriere durch die *B.await* Operation teil und wird blockiert.
- Der letzte (beliebige) Prozess p_N ruft ebenfalls die Barriere mit der *B.await* Operation auf, wird jedoch nicht blockiert und aktiviert dabei die wartenden Prozesse.
- Alle Prozesse haben damit einen gemeinsamen Kontrollpunkt passiert.

Operationen

OPERATION await

Der aufrufende Prozess *p* wird blockiert wenn $\#blocked(B) < N$, ansonsten fährt er in der Ausführung weiter und aktiviert alle blockierten Prozesse.

OPERATION init(*N*:numebr)

Setzt die Größe der Barrierengruppe (Anzahl der Prozesse *N*)

BARRIERE

```
MONITOR BARRIER (N) IS
  VAR counter: {0,1,..,N} := 0
  OBJECT q: condition
  OPERATION await IS
    counter := counter + 1
    IF counter < N THEN q.await()
    ELSE counter := 0
    q.signal() END IF
    RETURN
  END OPERATION
```

Abb. 40. Algorithmik der Barriere mit Monitor

TIMER

- Timer sind temporal selbstsynchronisierende Events.
- Anstelle eines Auslöseprozesses führt der Timer nach Ablauf eines Zeitintervalls die *wakeup* Operation aus.

Operationen

OPERATION await
Der aufrufende Prozess *p* wird blockiert bis der Timer ein *wakeup* ausführt.

OPERATION init(timeout:number,once:boolean)
Setzt das Timerintervall und ein Flag ob der Timer einmalig oder periodisch arbeitet

OPERATION start
Startet den Timer

CHANNEL

- Anders als bei den vorherigen Synchronisationsobjekten dienen Kanäle und Warteschlangen der synchronisierten Nachrichtenübertragung zwischen Prozessen: **Daten + Synchronisation**
- Kanäle werden auch häufig in der verteilten Programmierung eingesetzt.
- Kanäle sind daher Semaphoren (Produzenten-Konsumenten System!) mit Daten
- Ein Kanal kann multidimensional sein (tupelbasiert)

Operationen

OPERATION read(x_1,x_2,..) / receive(x_1,x_2,..)

Lesen eines Datenwertes oder Datensatzes und Übertragung der gelesenen Werte an die Variablenargumente (call-by-reference). Bedingung: Wenn der Kanal leer ist wird der lesende Prozess solange blockiert und in eine Prozesswarteschlange eingereiht bis ein schreibender Prozess teilnimmt.

CHANNEL

OPERATION write(ε_1,ε_2,..) / send(ε_1,ε_2,..)

Schreiben eines Datenwertes oder Datensatzes. Bedingung: Wenn der Kanal voll ist wird der schreibende Prozess solange blockiert und in eine Prozesswarteschlange eingereiht bis ein lesender Prozess teilnimmt.

OPERATION empty

Testet den Kanal ob er leer ist (keine Daten enthält)

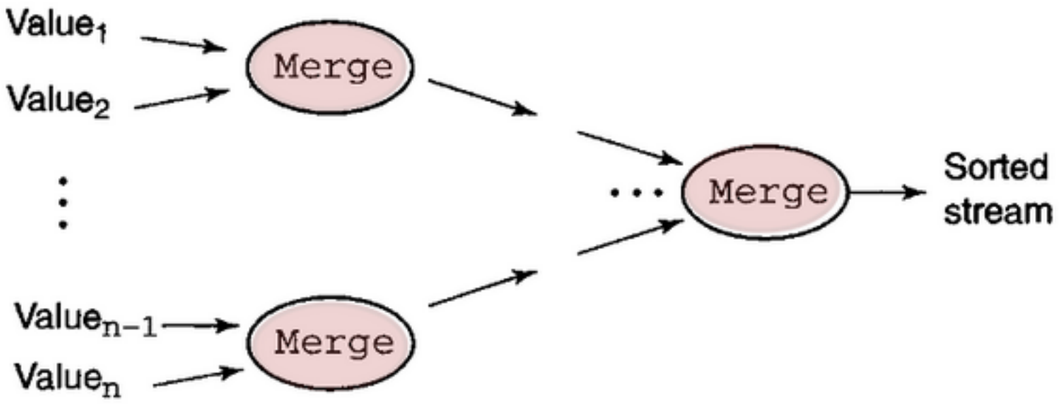


Abb. 41. Paralleler/Verteilter Map-reduce Algorithmus mit Kanälen zur Sortierung von Listen

PARALLELISIERUNG UND METRIKEN

PARALLELITÄT

- Unterteilung in räumliche und zeitliche Parallelität
- Parallele Datenverarbeitung bedeutet Partitionierung eines seriellen Programms in eine Vielzahl von Subprogrammen oder Tasks
- Weitere Unterteilung beider Dimensionen in Abhängigkeit von:
  - » Art der Tasks/Algorithmen
  - » Ausführungsmodell der Tasks und verwendete Rechnerarchitektur
  - » Art und Umfang der Wechselwirkung zwischen Tasks
  - » Kontroll- und Datenfluss eines Tasks

Räumliche Parallelität

Die Datenmenge D kann in Teilmengen $d_i \supset D$ zerlegt werden. Die minimal erreichbare Größe der Teilmenge gibt Granularität bei der Parallelisierung wieder. Die Datenmenge D wird durch eine Verarbeitungsstufe in eine neue Datenmenge D' transformiert, die dann von nachfolgenden Verarbeitungsstufen weiter verarbeitet wird. Beispiel : D=Bild ⇒ Glättung ⇒ D' ⇒ Objekterkennung ⇒ D''

PARALLELITÄT

Zeitliche Parallelität

Zeitliche Parallelität ist vorhanden, wenn eine Folge von gleichartigen Datenmengen $D(n)$ repitierend mit dem gleichen Algorithmus verarbeitet werden → Pipeline-Verfahren.

DATENABHÄNGIGKEIT

- Räumliche und zeitliche Parallelität führen zu räumlicher und zeitlicher Datenabhängigkeit.
- Räumliche Datenabhängigkeit findet auf Intra- und Intertaskebene statt.
  - » Intrataskebene: Subtasks tauschen Daten aus → Sequenzielle Ausführung. Beispiel: ST_1: a=x+y ; ST_2: b=a+1 ; → $b(a) \rightarrow ST_2(ST_1)$
  - » Intertaskebene: Übertragung von Daten zwischen Tasks in einer Pipeline.
- Zeitliche Datenabhängigkeit: Ergebnisse aus der Vergangenheit gehen in aktuelle Datenberechnung ein. Beispiel: Bewegungserkennung aus einer Bild-Sequenz.

DATENABHÄNGIGKEIT

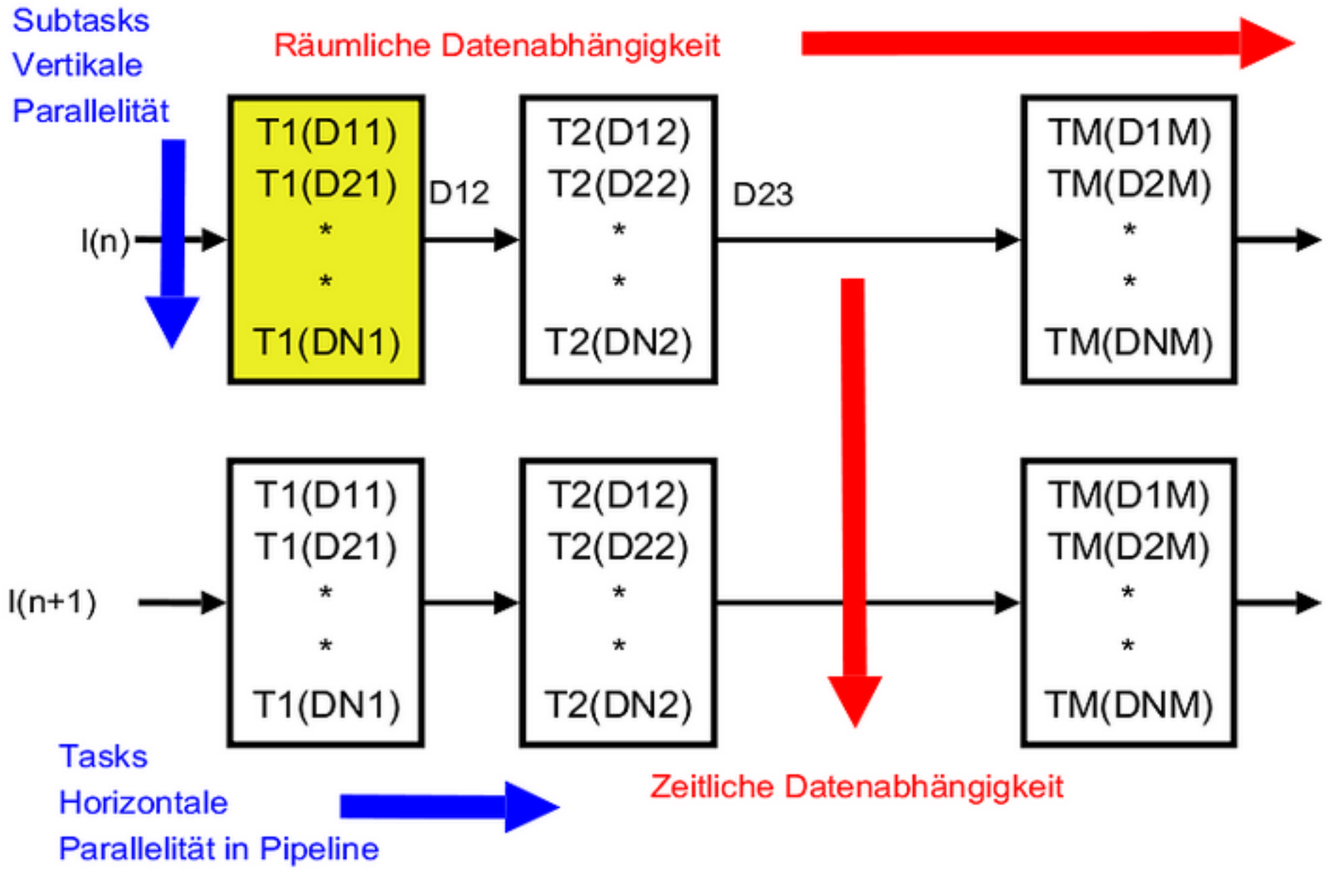


Abb. 42. Räumliche und zeitliche Datenabhängigkeit ⇔ Vertikalen und horizontale Parallelisierung

RECHENZEIT

- Die Rechenzeit t_{tot} für die Ausführung einer Pipeline $T_1 \ldots T_M$ enthält:
  1. Zeit zum Einlesen der Daten $I(n)$,
  2. Zeit für die Ausgabe der Daten und Ergebnisse,
  3. Summe aller Ausführungszeiten der Tasks in der Pipeline, die sich aus Rechen- und Kommuniaktionszeiten zusammensetzten.

$$t_{tot} = \sum_{i=1}^{m} \tau_i + \sum_{i=1}^{m-1} t_d(D_{i,i+1}) + t_{in} + t_{out}$$

$$\tau_i = \max\{t_{comp}(T_{i,j}(d_j)) | 1 \leqslant j \leqslant n_i\} + t_{comm}(T_i)$$

als die Zeit die benötigt wird, einen Task i unter Berücksichtigung von Datenabhängigkeiten der n_i Subtasks $T(d_j)$ zu bearbeiten.

RECHENZEIT

- Längste Bearbeitungszeit eines Subtasks bestimmt Bearbeitungszeit des Tasks τ_i!
- T_i ist der i-te Task in der horizontalen Pipeline,
- d_j die Teildatenmenge eines Subtasks $T_{i,i}(d_j)$ eines Tasks T_i,
- t_{comp} ist die Rechenzeit,
- t_{comm} die Intertaskkommunikationszeit,
- t_{in} und t_{out} die Zeit zum Datentransfer in und aus der Pipeline, und
- $t_d(D_{i,i+1})$ die Datentransferzeit zwischen zwei Tasks.

- In Vision-Systemen sind die ersten Tasks i.A. low- und mid-level Algorithmen, und die letzten Tasks high-level Algorithmen, die auf den Daten der unteren Ebenen aufbauen. Die einzelnen Datenströme D_i können daher von unterschiedlicher Größe und Art sein.

KLASSIFIKATION VON PARALLELEN ALGORITHMEN

Datenabhängigkeit

Lokal, statisch

Ausgangsdaten (≡ Ergebnisse) hängen nur von eng begrenzter kurzreichweitiger Region der Eingangsdaten ab. Die Größe der Eingangsdatenregion ist unveränderlich (statisch). → Kommunikationsbedarf ist gering.

Lokal, dynamisch

Die Größe der Eingangsdaten-Region ist parametrisiert und veränderlich (dynamisch). Z.B. bei mathematischer Faltung von Bildmatrizen ändert sich Größe der Region.

Global, statisch

Die Ausgangsdaten hängen gänzlich vom gesamter Eingangsdatenmenge ab. Abhängigkeit hängt nur von der Größe des Bildes, aber nicht von dessen Inhalt ab. Z.B. Fouriertransformation oder Histogramm-Funktionen. → Kommunikationsbedarf ist groß.

Global, dynamisch

Ausgangsdaten hängen von variierenden Ausschnittsregion der Eingangsdaten ab. Die Berechnung ist vollständig datenabhängig. Z.B. Hough-Transformation.

KLASSIFIKATION VON PARALLELEN ALGORITHMEN

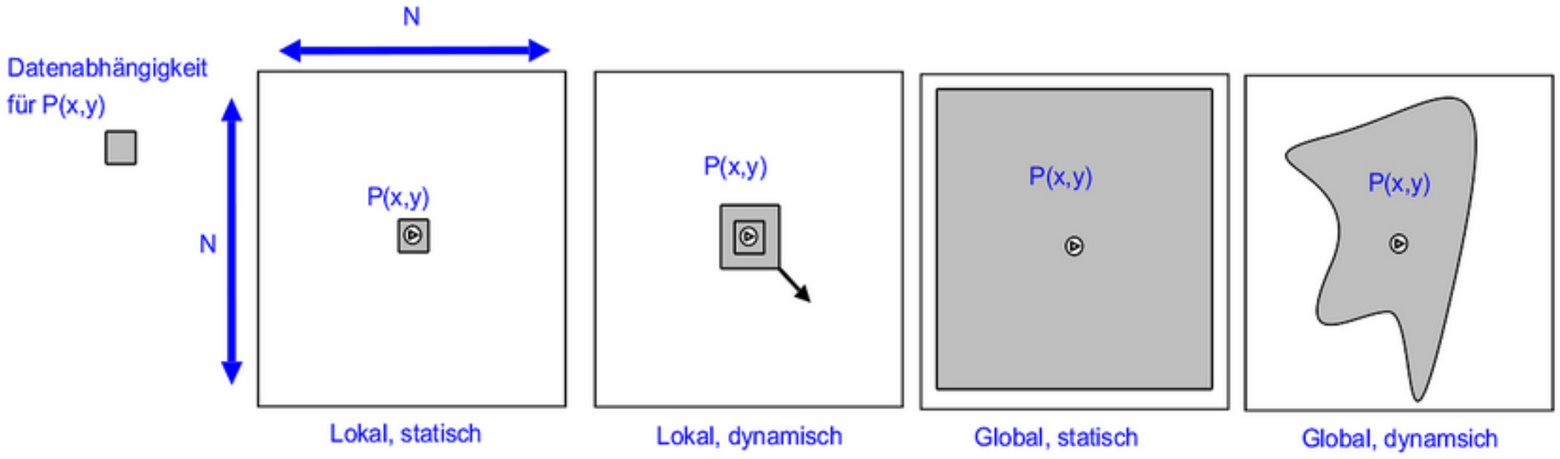


Abb. 43. Klassifikation nach Kommunikationsbedarf aufgrund Datenabhängigkeit zwischen einzelnen Tasks eines parallelen Programms

- Weiterhin Klassifikation nach:
  - » Eingabedatenabhängigkeit
  - » Ergebnisabhängigkeit

SPEEDUP

Bei der Datenverarbeitung gibt es drei Randbedingungen:

1. Gesamte Rechenzeit → Zeitdimension
2. Gesamte Ressourcenbelegung → Flächendimension
3. Bei der Parallelverarbeitung wird eine weitere Dimension hinzugefügt: Anzahl der Verarbeitungseinheiten *sN*

- Die Nutzung von Parallelität führt zu einem Performanzgewinn (Speedup) durch Vergleich sequenzielles Programm (N=1) und paralleles Programm (N Prozessoren):

$$Speedup = S(N) = \frac{Performanz(N)}{Performanz(1)}, S_t(n) = \frac{Rechnenzeit(1)}{Rechenzeit(N)}$$

- Die sog. Skalierung bei der Parallelisierung ist i.A. nicht linear:

$$0 < S(N) < N$$

- Kommunikation ist weitere Randbedingung bei der parallelen Datenverarbeitung.

KOSTEN UND LAUFZEIT

Beispiel: Matrixmultiplikation

```
FUN matmult(A:ARRAY (p,q), B: ARRAY(p,q)) -> C:ARRAY (p,r)
DEF matmult(A, B) =
  FOR i = 1 to p DO
    FOR j = 1 to r DO
      C[i,j] <- 0;
      FOR k = 1 to q DO
       C[i,j] <- C[i,j] + A[i,k] * B[k,j]
      END FOR k
    END FOR j
  END FOR i
END
```

- Partitionierung kann beliebig erfolgen, da die einzelnen Ergebniswerte $C_{i,j}$ nicht voneinander abhängen.
- Datenabhängigkeit des Problems: Die Berechnung eines $C_{i,j}$ Wertes hängt außerhalb der FOR-k-Schleife von keinem anderen Wert $C_{n,m}$ mit $n \neq i$ und $m \neq j$ ab.

KOSTEN UND LAUFZEIT

- Mögliche Partitionierungen der drei For-Schleifen auf *N* parallel arbeitenden Verarbeitungseinheiten (VE):

1. Jede VE berechnet einen $C_{i,j}$-Wert. D.h. eine VE führt die FOR-k-Schleife für ein gegebenes *i* und +j* durch.
   - » Jede VE benötigt dazu die i-te Zeile von *A* und die j-te Spalte von *B*.
   - » Keine weitere Datenabhängigkeit!
   - » Es werden $N=p \bullet r$ VEs benötigt.
2. Eine VE berechnet eine Ergebnisspalte, d.h. führt die FOR-k und FOR-j-Schleifen durch.
   - » Jede VE benötigt *A* und eine Spalte von *B*.
   - » Es werden $N=p$ VEs benötigt.

KOSTEN UND LAUFZEIT

3. Eine VE führt eine Multiplikation und Addition innerhalb der FOR-K-Schleife durch.
   - » Jede VE benötigt einen *A* und *B*-Wert.
   - » Es werden $N=p \bullet r \bullet q$ VEs benötigt.
   - » Jeweils eine VE für einen gebenen $C_{i,j}$-Wert führt die Zusammenführung der Zwischenwerte der FOR-k-VEs durch.

- Zusammenführung der Ergebnisdaten in den Fällen 1&2 trivial. Im Fall 3 besteht Zusammenführung im wesentlichen in der Summation der Zwischenergebnisse.

KOSTEN UND LAUFZEIT

Beispiel: p=q=r=2

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \times \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 1*5+2*7 & 1*6+2*8 \\ 3*5+4*7 & 3*6+4*8 \end{pmatrix}$$

Fall 1 ➽
VE1: {(1,2);(5,7)} ➽ C_{11}=1*5+2*7
VE2: {(3,4);(5,7)} ➽ C_{21}=3*5+4*7
VE3: {(1,2);(6,8)} ➽ C_{12}=1*6+2*8
VE4: {(3,4);(6,8)} ➽ C_{22}=3*6+4*8

Fall 2 ➽
VE1: {(1,2);(3,4);(5,7)} ➽
I. C_{11}=1*5+2*7
II. C_{21}=3*5+4*7
VE2: {(1,2);(3,4);(6,8)} ➽
I. C_{12}=1*6+2*8
II. C_{22}=3*6+4*8

Fall 3 ➽
VE1: {1;5}
VE2: {2;7} ➽ C_{11}=VE1⊕VE2
VE3: {3;5}
VE4: {4;7} ➽ C_{21}=VE3⊕VE4

Fall 3 ➽
VE5: {1;6}
VE6: {2;8} ➽ C_{12}=VE5⊕VE6
VE7: {3;6}
VE8: {4;8} ➽ C_{22}=VE7⊕VE8

Abb. 44. Mögliche Partitionierungen der Matrixmultiplaktion

KOSTEN UND LAUFZEIT

Laufzeit- und Kostenanalyse:
Vereinfachung: n=p=q=r

Fall 0: Sequenzieller Algorithmus
- Anzahl VE: 1
- Laufzeit: $\Theta(n^3)$
- Kosten: $\Theta(n^3)$

Fall 1
- Anzahl VE: n^2
- Laufzeit: $\Theta(n)$
- Kosten: $\Theta(n^3)$

Fall 2
- Anzahl VE: n
- Laufzeit: $\Theta(n^2)$
- Kosten: $\Theta(n^3)$

Fall 3
- Anzahl VE: n^3
- Laufzeit: $\Theta(\log n)$
- Kosten: $\Theta(n^3 \log n)$

Fall 3b (Master=+Worker)
- Anzahl VE: $n^3/\log n$
- Laufzeit: $\Theta(\log n)$
- Kosten: $\Theta(n^3)$!!!

Berechnung der C-Summe im Fall 3

$$c_{i,j} = \sum_k t_{i,j,k}$$

wobei je eine VE einen t-Wert berechnet:

$$V_{i,j,k} \Rightarrow t_{i,j,k} \text{ mit Laufzeit } \Theta(1)$$

Implementierung der Summe:

1. Sequenziell mit einem Addierer
   ➡ Laufzeit: $\Theta(n)$
2. Kette aus (n-1) Addierern
   ➡ Laufzeit $\Theta(n)$
3. Baum-Kaskade aus (n-1) Addierern
   ➡ Laufzeit $\Theta(\log n)$

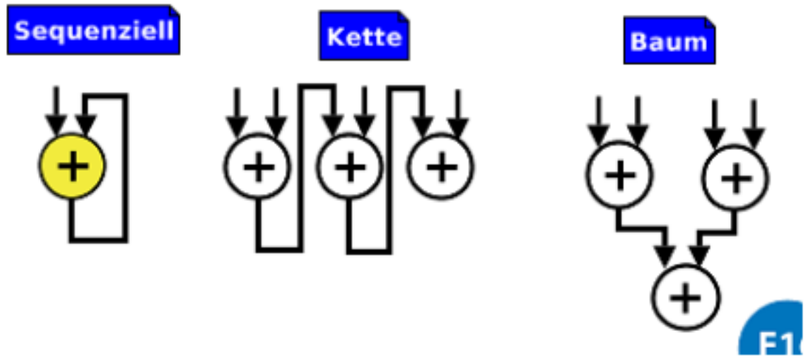


Abb. 45. Kosten und Laufzeitanalyse der verschiedenen Parititioneirung

KOMMUNIKATION

- Berechnung der Kommunikation mit Einheitswerten eines Nachrichten-austauschs: Message Passing (MP) → Aufwand eine Zelle einer Matrix zu versenden ist *MP*=1.

Vereinfachung: n=p=q=r
Berechnung eines C-Wertes mit einer VE erfordert:

$$VE_{i,j} \Rightarrow c_{i,j} \Rightarrow MP = MP(\rightarrow A_i \oplus \rightarrow B_j) + MP(C_{i,j} \rightarrow) = 2n + 1$$

Summe aller Nachrichten für Berechnung von C mit n^2 VEs:

$$\sum_{i,j} VE_{i,j} \Rightarrow C \Rightarrow MP = n^2(2n + 1) = 2n^3 + n^2 \Rightarrow \Theta(n^3)$$

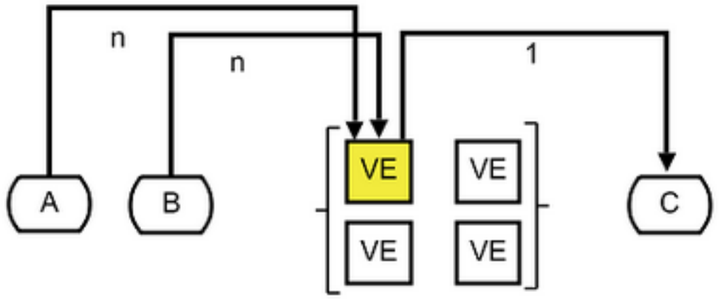


Abb. 46. Kommunikation Fall 1

KOMMUNIKATION

Vereinfachung: n=p=q=r
Berechnung einer C-Spalte mit einer VE erfordert:

$$VE_j \Rightarrow C_j \Rightarrow MP = MP(\rightarrow A_{i,j} \oplus \rightarrow B_j) + MP(C_{i,j} \rightarrow) = n^2 + n + n = n^2 + 2n$$

Summe aller Nachrichten für Berechnung von C mit n VEs:

$$\sum_j VE_j \Rightarrow C \Rightarrow MP = n(n^2 + 2n) = n^3 + 2n^2 \Rightarrow \Theta(n^3)$$

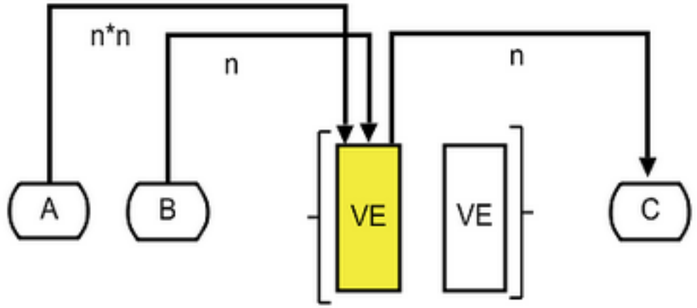


Abb. 47. Kommunikation Fall 2

KOMMUNIKATION

Vereinfachung: n=p=q=r
Berechnung eines t-Wertes mit einer VE erfordert:

$$VE_{i,j,k} \Rightarrow t_{i,j,k} \Rightarrow MP = MP(\rightarrow A_{i,j} \oplus \rightarrow B_{i,j}) + MP(t_{i,j,k} \rightarrow) = 3$$

Summe aller Nachrichten für Berechnung von C_{ij} mit n VEs:

$$\sum_{k} VE_{i,j,k} \Rightarrow C_{i,j} \Rightarrow MP = 3n$$

Summe aller Nachrichten für Berechnung von C mit n^3 VEs:

$$\sum_{i,j} VE_{i,j} \Rightarrow C \Rightarrow MP = n^2 3n = 3n^3 \Rightarrow \Theta(n^3)$$

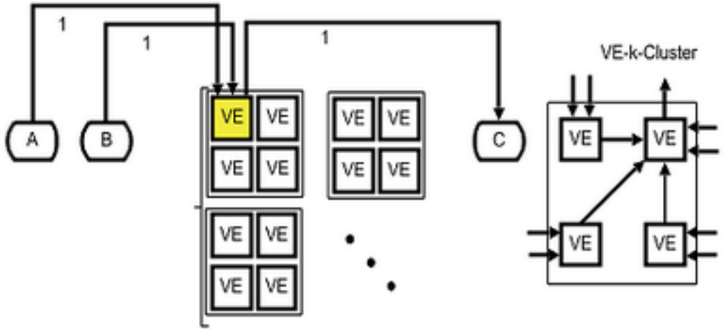


Abb. 48. Kommunikation Fall 3

KOMMUNIKATION

- Bisher konnten Matrizen ganzzahlig auf VEs verteilt werden. In der Realität ist aber p=q=r,n!
- Bisher wurde angenommen, dass alle VEs mit jeder anderen VE Daten mit einer Di- stanz/Ausdehnung D=1 austauschen kann. Nur möglich mit vollständig verbundenen Netzwerktopologien unter Verwendung von Kreuzschaltern.
- Gängige parallele und ökonomische Rechnertopologie: **Maschennetz**
  - Besteht aus $n \times n$ VEs.
  - Jede VE kann mit seinem direkten Nachbarn kommunizieren.
  - Eine Nachricht hat eine maximale Reichweite von $(2n-1)$ VEs, $\Theta(n)$.

KOMMUNIKATION

- Bisherige statische Partitionierung resultiert in zu hohen Kommunikationsaufwand in der Verteilung der Matrizen A und B sowie in der Zusammenführung der Matrix C.
  - » Dynamisch veränderliche Partitionierung passt sich effizienter an Netzwerktopologie an.
  - » Bei Matrixoperationen mit zwei Matrizen ($n=p=q=r$) ist dafür ein $2n \times 2n$ Netz optimal geeignet.

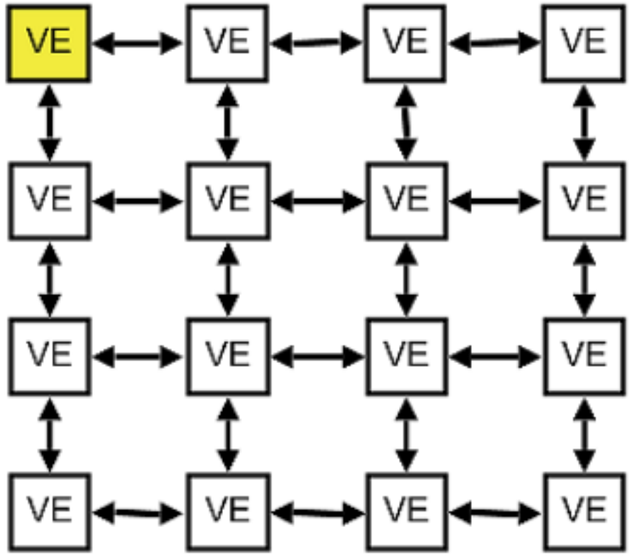


Abb. 49. Zweidimensionales Maschennetzwerk

KOMMUNIKATION

Dynamische Verteilung der Matrizen

- Das $2n \times 2n$ Netz wird in vier Quadranten der Größe $n \times n$ unterteilt.
  - » Die Matrizen *A* und *B* befinden sich initial im linken unteren und rechten oberen Quadranten
  - » Die Ergebnismatrix *C* wird schließlich im rechten unteren Quadranten zusammen- geführt.
- Alle VEs die ein $A_{1,j}$-Element besitzen werden dieses nach rechts verschieben.
- Alle VEs die ein $B_{i,1}$-Element besitzen werden dieses nach unten verschieben.
  - » Dieser Vorgang wird für weitere Zeilen (*A*) und Spalten (*B*) fortgesetzt.
  - » Die einzelnen Elemente von *A* und *B* passieren die VEs im C-Quadranten. Die einzelnen C-Werte können mittels Summation berechnet werden.

KOMMUNIKATION

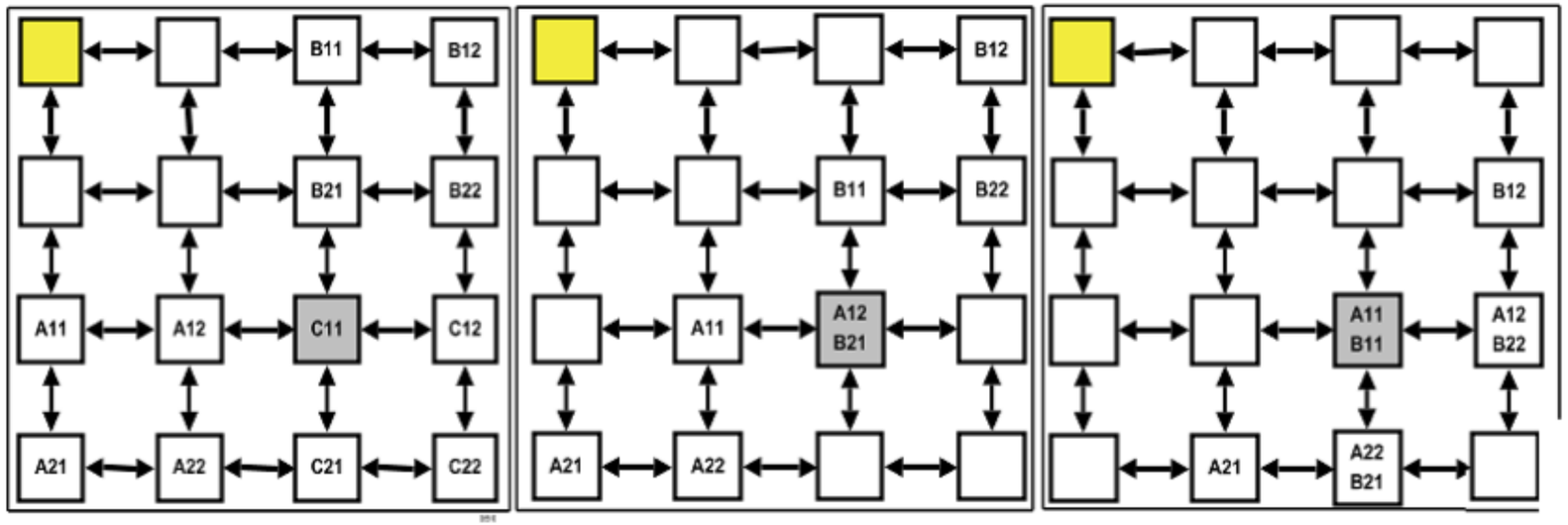


Abb. 50. Dynamische und überlagerte Verteilung der Matrizen *A*, *B*, und *C*

KOMMUNIKATION

- Die Laufzeit der Matrixmuliplikation auf dem Netz beträgt $\Theta(n)$, da n Verschiebungen durchgeführt werden.
- Die Kosten betragen $\Theta(n^3)$, vergleichbar mit sequenzieller Ausführung. D.h. diese Partitionierung und Architektur ist kostenoptimal.
- Das 2nx2n Netz wird nur in drei Quadranten genutzt. Benötigt werden daher nur $3n^2$ VEs.
- Aus Sicht der Rechnerarchitektur ungünstig zu implementieren und nicht generisch, d.h. abhängig vom verwendeten Algorithmus.
- Reduktion dieses Verfahrens auf $n \times n$ Netz möglich
  - Dazu werden die Matrizen *A*, *B* und *C* überlagert, d.h. $VE_{1,1} \Rightarrow \{A_{1,1}, B_{1,1}, C_{1,1}\}$.
- Die Zeilenelemente von *A* werden dann nach vorherigen Ablaufschema gegen den Uhrzeigersinn rotiert (nur horizontal), und die Spaltenelemente von *B* im Uhrzeigersinn vertikal).
- Man erhält gleiche asymptotische Grenzwerte für die Laufzeit $\Theta(n)$ und Kosten $\Theta(n^3)$!

KOMMUNIKATION

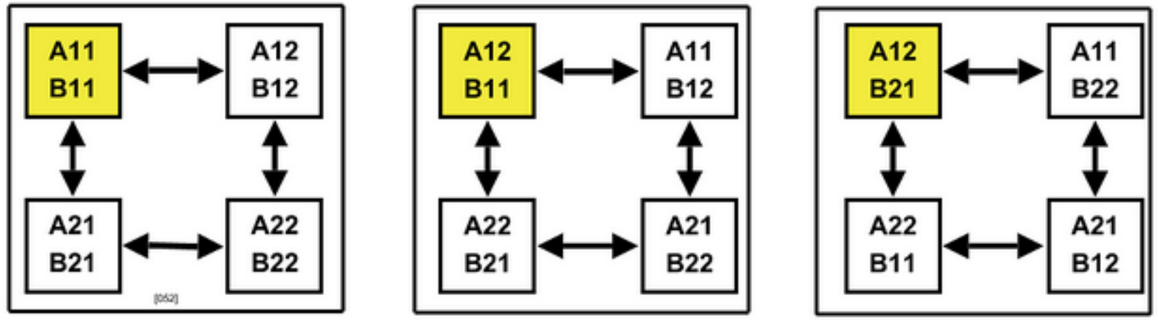


Abb. 51. Überlagerte Verarbeitung der Matrizen *A*, *B*, und *C* auf einem Maschennetzwerk

MASSZAHLEN FÜR PARALLELE SYSTEME

Berechnungszeit

Die Berechnungszeit T_{comp} (computation time) eines Algorithmus als Zeit, die für Rechenoperationen verwendet wird.

Kommunikationszeit

Die Kommunikationszeit T_{comm} (communication time) eines Algorithmus als Zeit, die für den Daten- bzw. Nachrichtenaustausch (Sende- und Empfangsoperationen) zwischen Subprogrammen und Verarbeitungseinheiten verwendet wird.

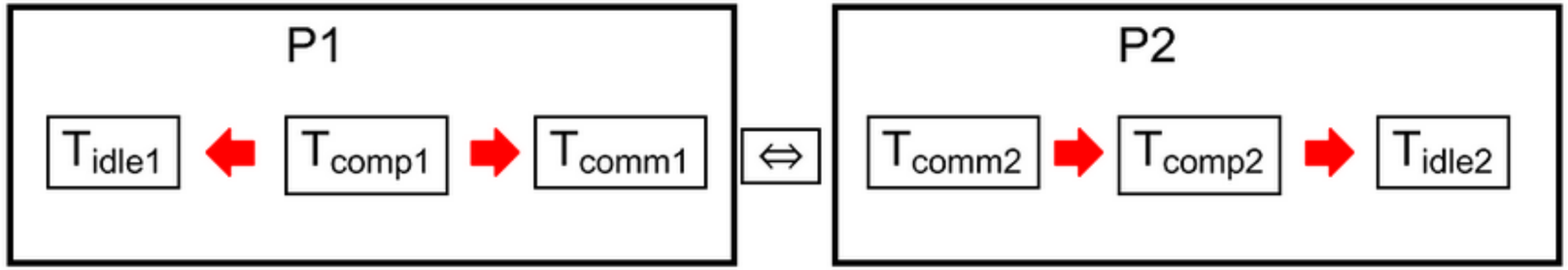
Untätigkeitszeit

Die Untätigkeitszeit T_{idle} (idle time) eines Systems als Zeit, die mit Warten (auf zu empfangende oder sendende Nachrichten) verbracht wird → Prozessblockierung trägt zur Untätigskeitszeit bei!

MASSZAHLEN FÜR PARALLELE SYSTEME

Es gilt:

$$T_{tot} = T_{comp} + T_{comm} + T_{idle} \approx 1/N \sum_{i=1..n} T_{comp,i} + T_{comm,i} + T_{idle,i}$$



MASSZAHLEN FÜR PARALLELE SYSTEME

Übertragungszeit

Die Übertragunszeit T_{msg} (message time) ist die Zeit, die für das Übertragen einer Nachricht mit der Länge L Datenwörter zwischen zwei Prozessen oder Prozessoren benötigt wird.

- Sie setzt sich aus einer Startzeit T_s (message startup time) und einer Transferzeit T_w für ein Datenwort zusammen.
- Es gilt: $T_{msg} = T_s + L \bullet T_w$
- Voraussetzung: Verbindungsnetz arbeitet konfliktfrei.

Startzeit

Die Startzeit wird durch die Kommunikationshard- und software bestimmt, die zur Initiierung eines Datentransfers benötigt wird, z.B. Overhead des Protokollstacks bei einer Software-Implementierung.

Transferzeit

Die Transferzeit wird durch die Bandbreite des Kommunikationsmediums und zusätzlich bei Software-Implementierung durch den Protokollstack (Datenkopie) bestimmt.

MASSZAHLEN FÜR PARALLELE SYSTEME

Parallelisierungsgrad P

Die maximalze Anzahl von binären Stellen (bits) pro Zeiteinheit (Taktzyklus) die von einer Datenverarbeitungsanlage verarbeitet werden kann.

» Es gilt: $P = W \bullet B$

Wortlänge W

Die Wortlänge oder Wortbreite gibt die Anzahl der Bits eines Datenpfades an.

Bitslice-Länge B

Die Bitslice-Länge setzt sich zusammen aus der Anzahl von Verarbeitungseinheiten VE, die parallel ausgeführt werden können, und der Anzahl der Pipeline-Stufen einer VE.

» Es gilt: $B = N_{VE} \bullet N_{Stages}$

MASSZAHLEN FÜR PARALLELE SYSTEME

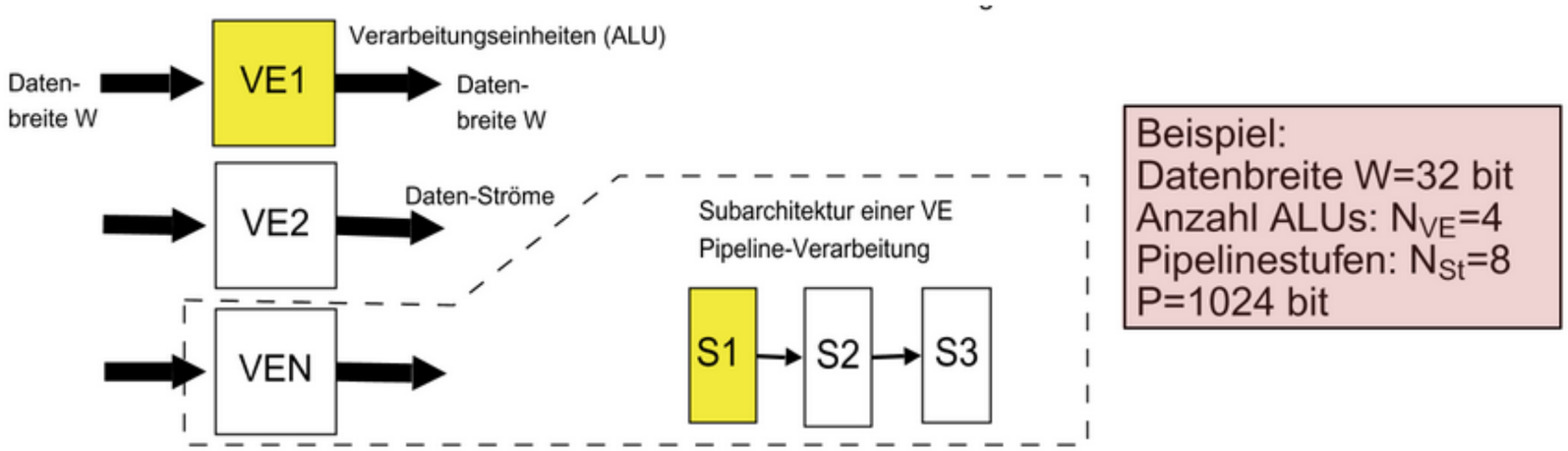


Abb. 52. Illustration Parallelisierungsgrad und Beispiel

MASSZAHLEN FÜR PARALLELE SYSTEME

Beschleunigung S und Kosten C

Die Beschleunigung gibt die Steigerung der Verabeitungsgeschwindigkeit bzw. die Re-duzierung der Verarbeitungszeit *T* an beim Übergang Anzahl Prozessoren $N=1 \rightarrow N>1$. Die Kosten *C* skaliert die Verarbeitungszeit (Komplexitätsklasse) mit den Ressourcen.

$$S(N) = \frac{T(1)}{T(N)}, C(N) = T(N)N$$

Effizienz E

Die Effizienz gibt die relative Verbesserung in der Verarbeitungsgeschwindigkeit an, da die Leistungssteigerung *S* mit der Anzahl der Prozessoren normiert wird.

$$E(N) = \frac{S(N)}{N}, \text{ mit } \frac{1}{N} \leqslant E(N) \leqslant 1$$

MASSZAHLEN FÜR PARALLELE SYSTEME

Mehraufwand R

Bezieht sich auf die Anzahl X der auszuführenden (Einheits-)Operationen des Programms:

$$R(N) = \frac{X(N)}{X(1)}$$

Parallelindex I

Der Parallelindex gibt die Anzahl der parallelen Operationen pro Zeit-/Takteinheit an.

$$I(N) = \frac{X(N)}{T(N)}$$

Auslastung U

Entspricht dem normierten Parallelindex:

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

MASSZAHLEN FÜR PARALLELE SYSTEME

Beispiel zur Auslastung

Ein Einprozessorsystem benötigt für die Ausführung von 1000 Operationen genau 1000 (Takt-)Schritte. Ein Multiprozessorsystem mit 4 Prozessoren benötigt dafür 1200 Operationen, die aber insgesamt in 400 Schritten ausgeführt werden können:

```
X(1)=1000 und T(1)=1000, X(4)=1200 und T(4)=400
S(4)=2.5 und E(4)=0.625,I(4)=3 und U(4)=0.75
```

Im Mittel sind 3 Prozessoren gleichzeitig aktiv, da jeder Prozessor nur zu 75% ausgelastet ist!

AMDAHL'S GESETZ

Eine kleine Zahl von sequenziellen Operationen kann den Performanzgewinn durch Parallelisierung signifikant reduzieren.

- Sequenzieller Anteil der Berechnungszeit T eines Algorithmus in [%]: η
- Paralleler Anteil dann: $(1-\eta)$
- Kommunikation (Synchronisation) zwischen nebenläufig ausgeführten Tasks oder Verarbeitungseinheiten verursacht immer $\eta>0$!
  - » Beispiel: Schutz einer geteilten Ressource mit einer Mutex.
  - » Beispiel: Datenverteilung über eine Queue
- Zugriff auf geteilte Ressourcen verursacht immer $\eta>0$!
  - » Beispiel: Geteilte Ressource Hauptspeicher in einer PRAM
- Der Kommunikationsanteil ist schwer im Voraus abzuschätzen, und der genaue Wert hängt auch von temporalen Konkurrenzverhalten ab (wie häufig gibt es verlorene Wettbewerbe)!

AMDAHL'S GESETZ

- Es gilt dann für die gesamte Berechnungszeit eines parallelen Systems:

$$T(N,\eta) = \eta T(1) + \frac{(1-\eta)T(1)}{N}$$

- Daraus lässt sich eine Obergrenze der Beschleunigung S mit zusätzlicher Abhängigkeit von η ableiten:

$$S(N,\eta) = \frac{T(1)}{T(N,\eta)} \leqslant \frac{N}{(\eta N - 1) + 1}$$

AMDAHL'S GESETZ

Beispiel

Ein Algorithmus mit einem sequenziellen Anteil von $\eta = 10\%$ und einem parallelen Teil von 90% wird A.) mit N=10 Prozessoren, und B.) mit N=20 Prozessoren ausgeführt. Die Obergrenze der Beschleunigung S ist dann:

```
A.) S(10)=5.26
B.) S(20)=6.0
```

- Verdopplung der Prozessoren erhöht nicht mehr signifikant die Beschleunigung!

AMDAHL'S GESETZ

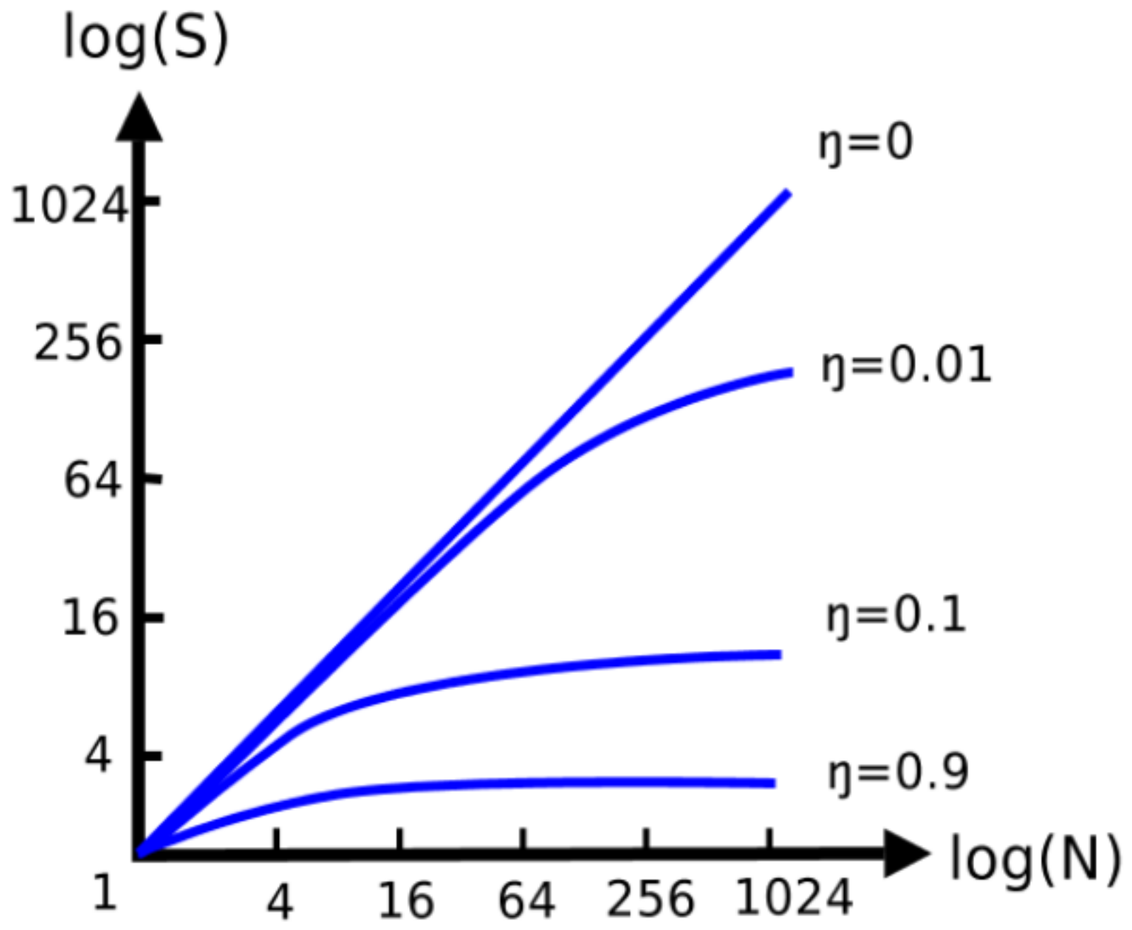


Abb. 53. Beschleunigung S in Abhängigkeit von η und Anzahl der Prozessoren N

PLATTFORMEN UND NETZWERKE

EIGENSCHAFTEN VON PARALLELARCHITEKTUREN

Rekonfigurierbarkeit

Unterschiedliche Ebenen der Algorithmik erfordern unterschiedliche Rechnerarchitekturen. Sowohl SI and MI-Ausführung muss dynamisch (re-)konfigurierbar möglich sein → Dynamische Anpassung der Architektur an Algorithmen.

Flexible Kommunikation

Fein granulierte und Hochgeschwindigkeitskommunikation ist für effiziente Ausführung der Algorithmen und Tasks erforderlich. Kommunikation zwischen Tasks und den unterschiedlichen Verarbeitungsebenen.

Ressourcen Allokation und Partitionierbarkeit

Gesamtsystem muss aus unabhängigen Teilsystemen bestehen, um effiziente Implementierung der (unterschiedlichen) Tasks zu ermöglichen. Ressourcen (Prozessoren, Speicher) müssen dynamisch und fein granuliert an die Tasks/Prozesse gebunden werden können.

EIGENSCHAFTEN VON PARALLELARCHITEKTUREN

Lastbalanzierung und Task Scheduling

Besonders für high-level (=komplexe) Algorithmen mit starker Datenabhängigkeit sind Lastverteilung und zeitliches Taskscheduling von großer Bedeutung, aber nicht trivial.

» In low-level Algorithmen mit geringer Datenabhängigkeit erreicht man Lastbalanzierung meistens durch Datenpartitionierung.

Unabhängigkeit von Topologie- und Datengröße

Durch die hohe Komplexität von Algorithmen (z.B. Vision) muss die Architektur unabhängig von detaillierten Annahmen bezüglich Datenstruktur (z.B. Matrixgrößen oder Datenbreite) und bestimmter Algorithmik sein!

» Betrifft auch dynamisch konfigurierbare und skalierbare Kommunikationsstrukturen.

EIGENSCHAFTEN VON PARALLELARCHITEKTUREN

Fehlertoleranz

Bei der Bearbeitung von komplexen Aufgaben mit komplexen Systemen spielt Fehlertoleranz eine wichtige Rolle. Ein Ausfall einer einzelnen Komponente darf nicht zum Ausfall des gesamten Systems führen. Statische Redundanz ist aber teuer (Ressourcen- bedarf)!

Ein- und Ausgabe (IO)

Neben geringer Bearbeitungs- und Rechenzeit ist der Datentransfer der Eingangs- und Ausgangsdaten (Ergebnisse) gleichbedeutend. Performanz und Architektur von IO ist ein Kernbestandteil paralleler Systeme!

KLASSIFIKATION RECHNERARCHITEKTUR

- Eine Datenverarbeitungsanlage enthält:

VE

Verarbeitungseinheit für Daten → Generischer Prozessor, Applikations-pezifischer Pro- zessor, Applikationspezifische Digitallogik, Teileinheit

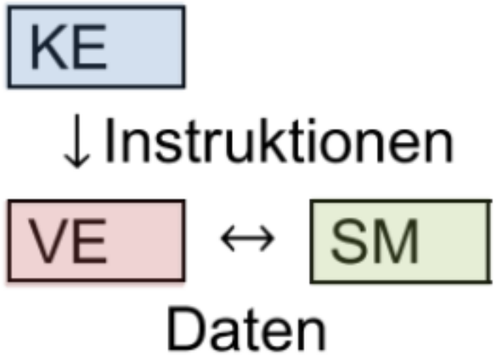
SM

Speichermodul → RAM, Registerbank, Cache

KE

Kontrolleinheit für die Ablaufsteuerung, kann in VE enthalten sein.

- Dabei sind diese drei Einheiten über **Instruktionsströme** und **Datenströ-me** gekoppelt:



KLASSIFIKATION RECHNERARCHITEKTUR

Klassifikation nach Flynn

- Durch Beschreibung von Daten- und Kontrollfluss:

SISD
Single-Instruction ⊗ Single-Data Stream

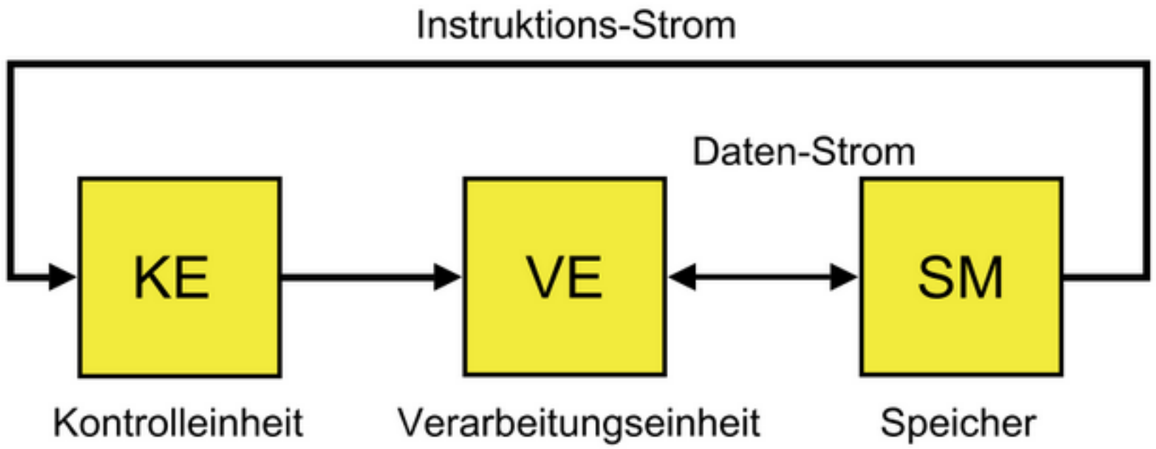
SIMD
Single-Instruction ⊗ Multiple-Data Stream

MISD
Multiple-Instruction ⊗ Single-Data Stream

MIMD
Multiple-Instruction ⊗ Multiple-Data Stream

SISD-ARCHITEKTUR EIGENSCHAFTEN

- Von-Neuman-Architektur gehört zur SISD- Klasse
- SISD-Architekturen verarbeiten einen sequenziellen Daten- und Kontrollstrom.
- Nur eine VE und KE wird benötigt.
- Ein zentraler Speicher für Daten und Instruktionen
- Keine algorithmische Parallelisierung implementierbar - nur implizit mittels Pipeline- Verfahren.
- Programmiermodell: explizite Ablaufsteuerung; simulierte konkurrierende Programmierung mit Software-Threads.



SISD-ARCHITEKTUR - VON-NEUMANN ARCHITEKTUR

Phasen der Befehlsausführung

I. Befehlsholphase: $\wp(BZ) \rightarrow BR$
II. Dekodierungsphase: $BR \rightarrow \{S1,S2,..\}$
III. Operandenholphase: $\wp \rightarrow R$
IV. Befehlsausführung: χ
V. Rückschreibephase: $R \rightarrow \wp$
VI. Adreßrechnung: $\Psi(BZ,SR,BR) \rightarrow BZ$

Architekturkomponenten

BZ, BR, S
Befehlszähler, Befehlsregister, Steuersignale

SR,R,$\wp$
Statusregister, Register, Speicher

A,D
Adress- und Datenbus

SISD-ARCHITEKTUR - VON-NEUMANN ARCHITEKTUR

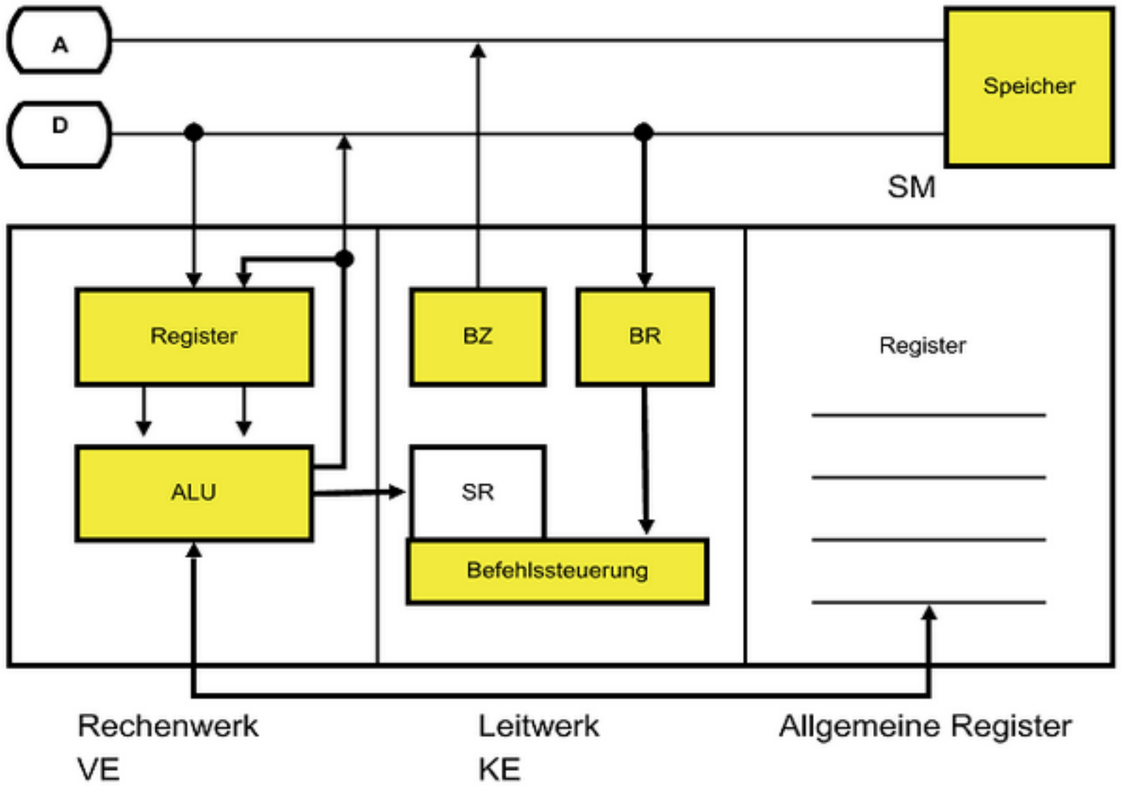
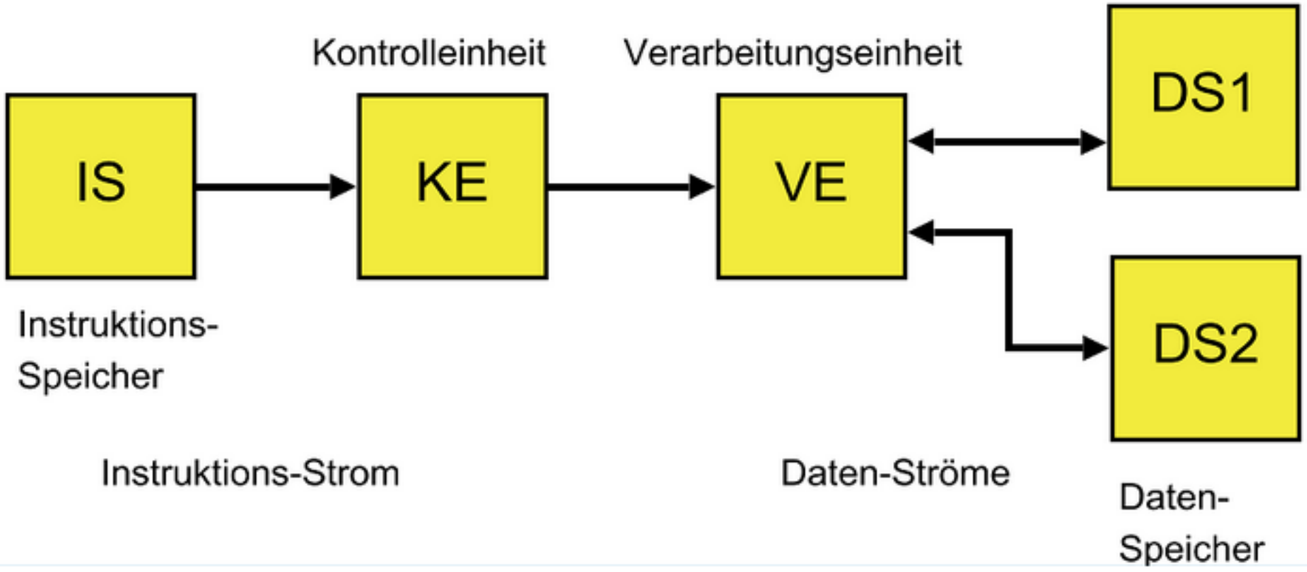


Abb. 54. Von-Neumann Architekturbild

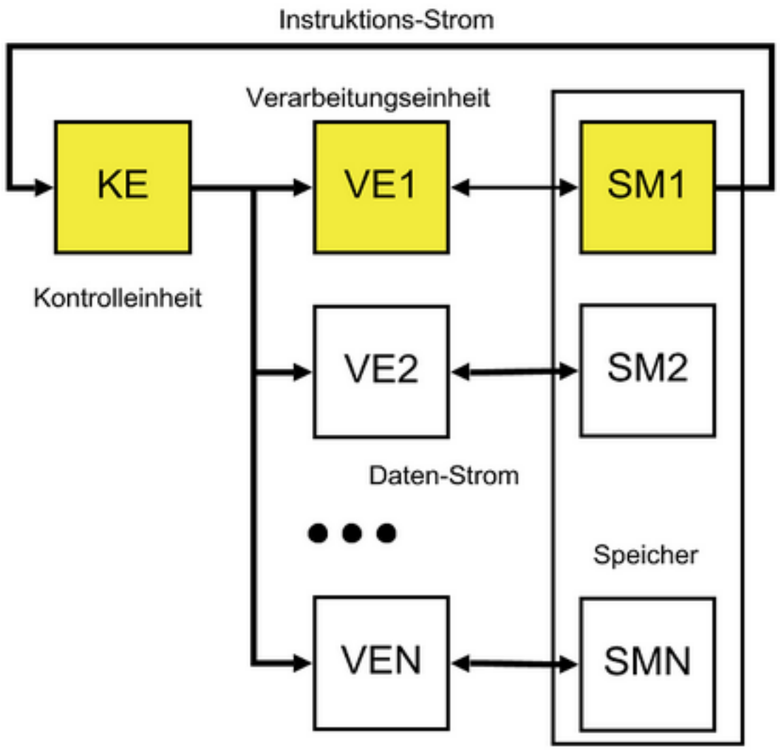
SISD-ARCHITEKTUR - HARVARD-ARCHITEKTUR

- Getrennte Daten- und Instruktionsspeicher
- Mehrere Datenbusse und Datenspeicher ä Spezialmaschine optimiert für datenintensive Algorithmen
- Parallelisierung auf Instruktionsebene: Teilphasen der Befehlsausführung können nebenläufig ausgeführt wer- den, z.B. Befehls- und Operanden-holphase.



SIMD-ARCHITEKTUR

- Parallelität auf Datenebene
- Ablaufsteuerung mit einer gemeinsamen Kontrolleinheit KE
- Zentrales Speichermodell, aber jede Verarbeitungseinheit VE kann eigenen Speicher besitzen
- Jede VE bearbeitet gleiche Instruktion Datenoperation wird auf *N* VEs verteilt
- Gleiche Operation kann auf verschiedenen Operanden oder einem Operanden der Datenbreite W=N•W' ausgeführt werden
- Keine Programmsynchronisation erforderlich



SIMD-ARCHITEKTUR

Vektor- & Array-Struktur

- Sind Subarchitekturen von SIMD
- Die Programmierung der SIMD-Architektur hängt von der verwendeten Subarchitektur ab.

Vektorarchitektur

Die Verarbeitungseinheiten VE sind linear als Vektorstruktur angeordnet. Jede VE verfügt über eigenen Speicher. Die einzelnen VEs sind mit ihrem jeweiligen linken und rechten Nachbarn verknüpft, und können über diese Verbindungen Daten austauschen.

Arrayarchitektur

Die Verarbeitungseinheiten VE sind als zweidimensionale Matrixstruktur verknüpft. Jede VE besitzt Kommunikationsverknüpfungen mit bis zu maximal vier Nachbareinheiten, so dass Zeilen und Spaltenverbindungen bestehen.

SIMD-ARCHITEKTUR

- Beide SIMD-Architekturen sind Spezialmaschinen.
- Sie sind gut geeignet für Algorithmen auf regulären Datenstrukturen, bei denen die gleiche Operation auf eine Vielzahl von Operanden angewendet werden soll.
  - » Numerische Matrixoperationen wie Matrix- und Vektormultiplikation
  - » Digitale Bildverabeitung mit zweidimensionalen Bildmatrizen

SIMD-ARCHITEKTUR

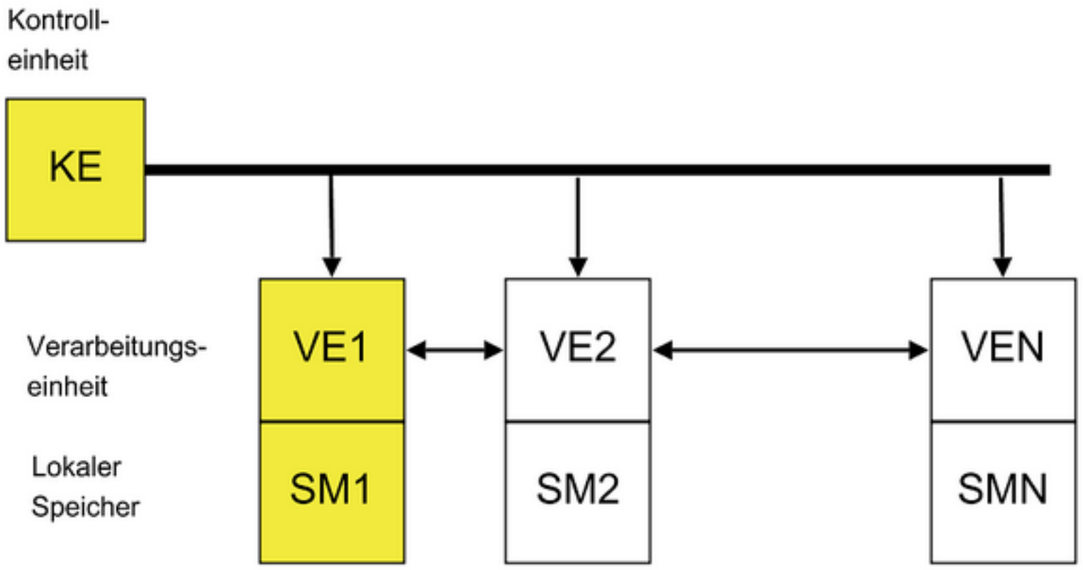


Abb. 55. Vektorrechner

SIMD-ARCHITEKTUR

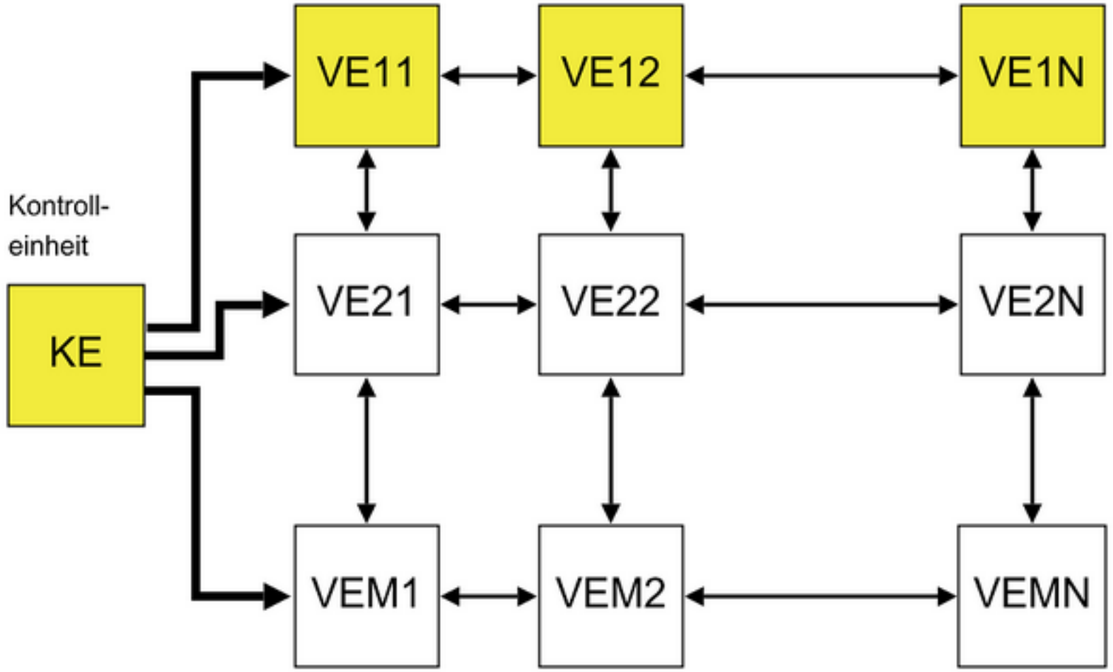


Abb. 56. Matrixrechner

MISD-ARCHITEKTUR

- Diese Architektur verarbeitet mehrere Instruktionen parallel. Jede VE hat eigenen IS.
- Alle Instruktionen operieren auf dem gleichen Datensatz oder über einen Datenstrom.
- Pipeline-Computer (Instruktions-Pipeline in der CPU) gehören zur MISD-Klasse
- Spezialmaschine;
  - Anwendung in der digitalen Bildverarbeitung und der Robotik Anwendung.
  - Z.B. Objektklassifikator: Das gleiche Video-Bild (oder Ausschnitt) wird auf verschiedene Algorithmen angewendet (Objektklassifikation).
  - Z.B. Space Shuttle flight control computer!

MISD-ARCHITEKTUR

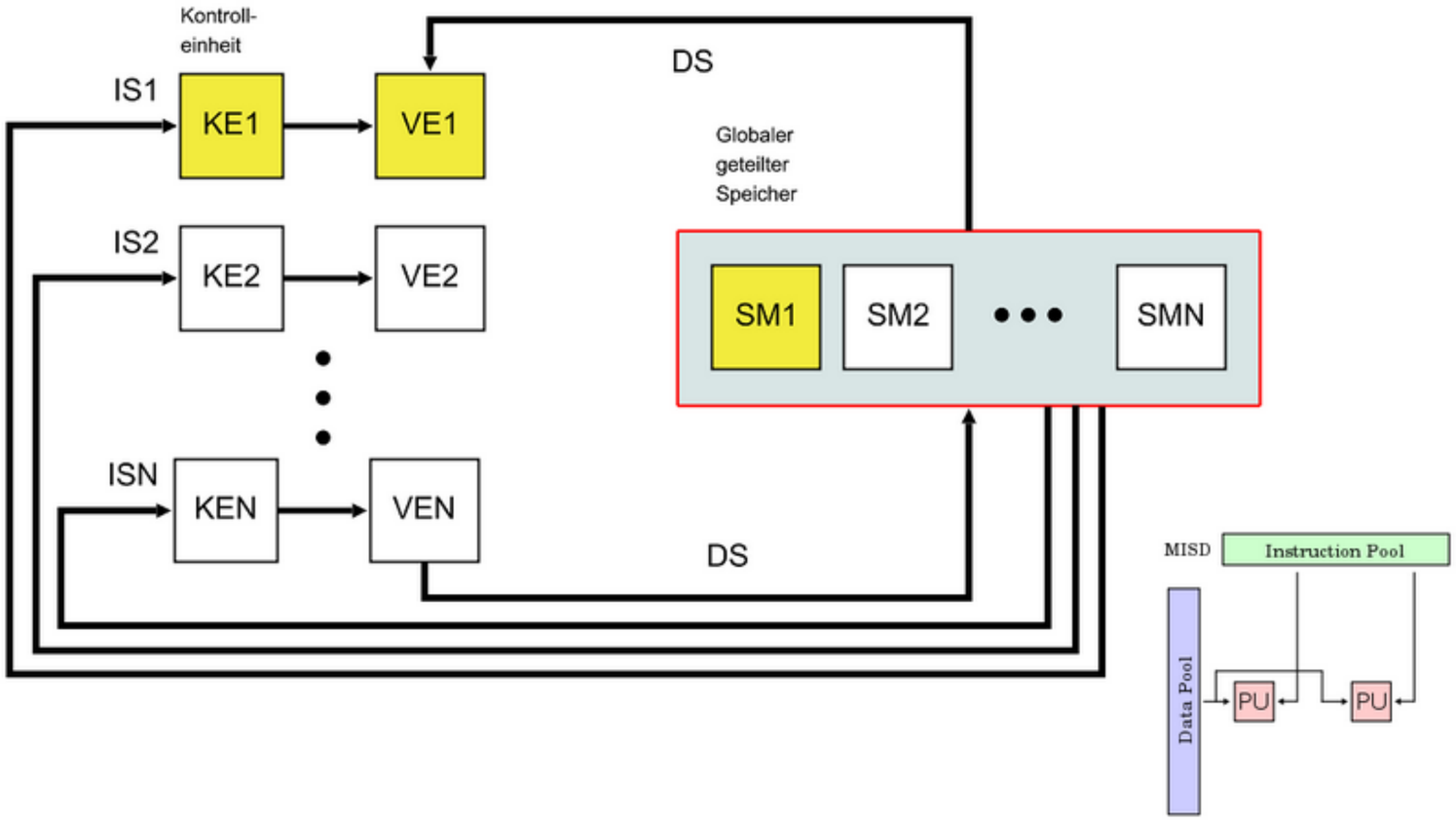
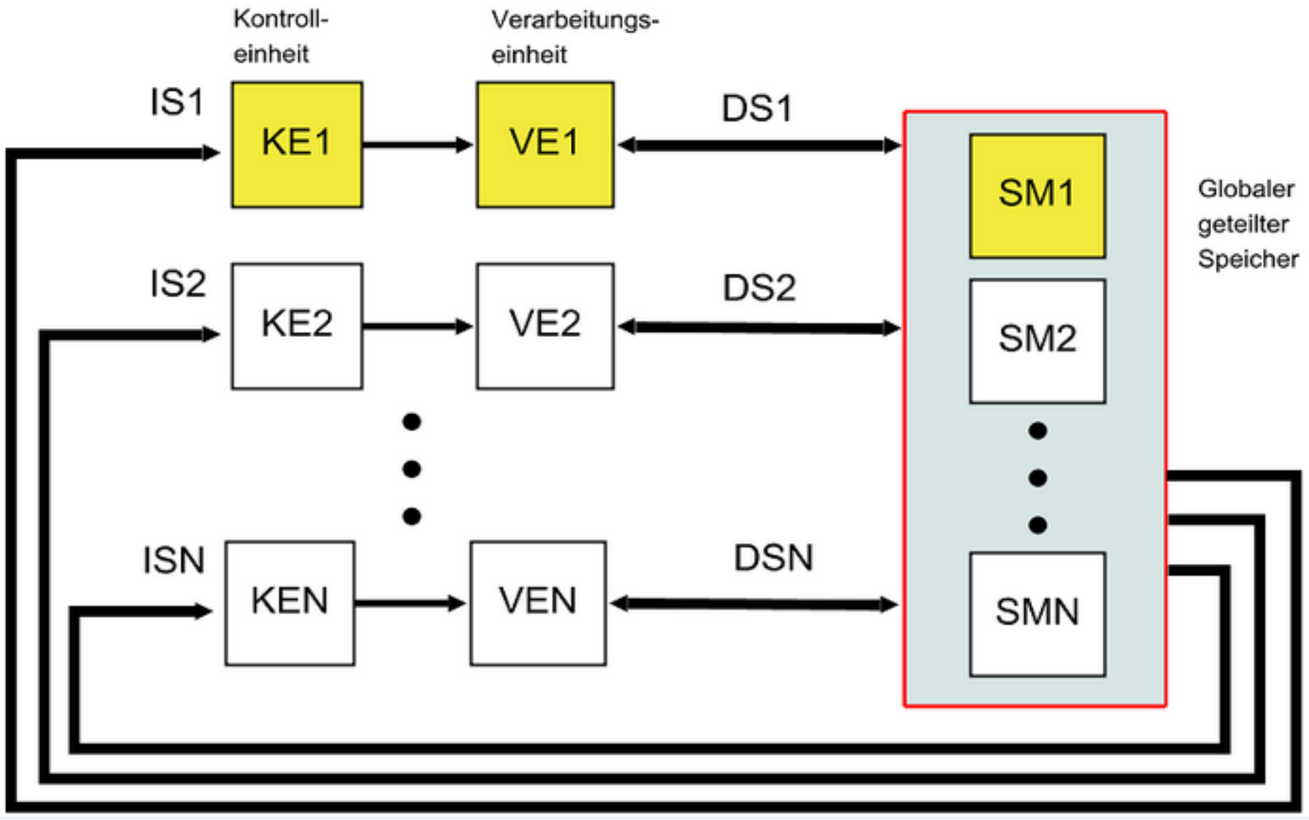


Abb. 57. MISD-Architekturaufbau; Der Hauptspeicher kann segmentiert sein (wie bei Harvard)

MIMD-ARCHITEKTUR

- Jede VE arbeitet mit eigenen Instruktionsstrom.
- Jede VE arbeitet mit eigenen Datenstrom.
- Synchronisation zwischen den einzelnen VE ist erforderlich.
- Synchronisationsobjekte sind von allen VEs geteilte Ressourcen.



SPEICHERMODELLE

Exklusives Lesen (ER - ExclusiveRead)

Pro Zyklus kann höchstens ein Prozessor dieselbe Speicherzelle lesen.

Exklusives Schreiben (EW - ExclusiveWrite)

Pro Zyklus kann höchstens ein Prozessor dieselbe Speicherzelle beschreiben.

Konkurrierendes Lesen (CR - Concurrent Read)

Pro Zyklus können mehrere Prozessoren dieselbe Speicherzelle lesen.

Konkurrierendes Schreiben (CW - Concurrent Write)

Pro Zyklus können mehrere Prozessoren dieselbe Speicherzelle beschreiben. Es besteht Konfliktgefahr und Dateninkonsistenz, die mit verschiedenen Methoden gelöst werden können:

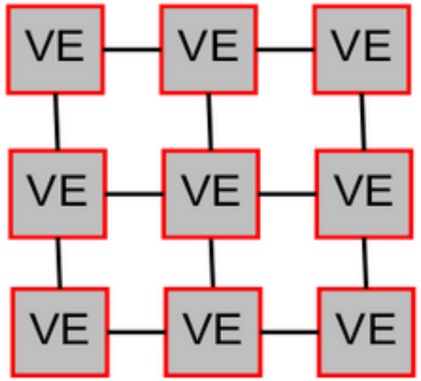
- Common (C-CW) Gleichzeitige Schreibzugriffe auf dieselbe Speicherzelle sind nur erlaubt, wenn alle Schreibzugriffe den gleichen Datenwert liefern.
- Arbitrary (A-CW) Einer der schreibenden Prozessoren gewinnt und belegt die Speicherzelle - Schutzmechanismus einer geteilten Ressource (Mutual Exclusion). Die anderen Schreibzugriffe werden ignoriert!
- Priority (P-CW) Auswahl des Prozessors nach Prioritätengewichtung (z.B. Prozessorindex).

RECHNERARCHITEKTUR FÜR VISION-SYSTEME (I)

Gitternetzwerk

- Über Maschennetz verbundene Prozessoren/Rechner
- Reguläre zweidimensionale Verbindungsstruktur mit großer Anzahl von Verarbeitungseinheiten (VE) quadratisch angeordnet.
  - » Häufig ist jede VE mit seinen direkten Nachbarn (4) verknüpft.
  - » Jede VE besitzt i.A. lokalen Speicher.
- Zweidimensionale Datenstrukturen wie Bilder lassen sich einfach auf diese Struktur abbilden.
- Maximale Parallelität nur erreichbar bei Algorithmen und Operationen auf Pixel-Ebene, d.h. bei kurzer "Berechnungslänge".
- Kommunikation in Netzen ist über weite Distanzen teuer (Zeit).
- Algorithmen wie Gruppierung oder Mustererkennung erfordern langreichweitige Berechnungen. Führt zu hohem Kommunikationsüberhang zwischen den VEs.

RECHNERARCHITEKTUR FÜR VISION-SYSTEME (I)



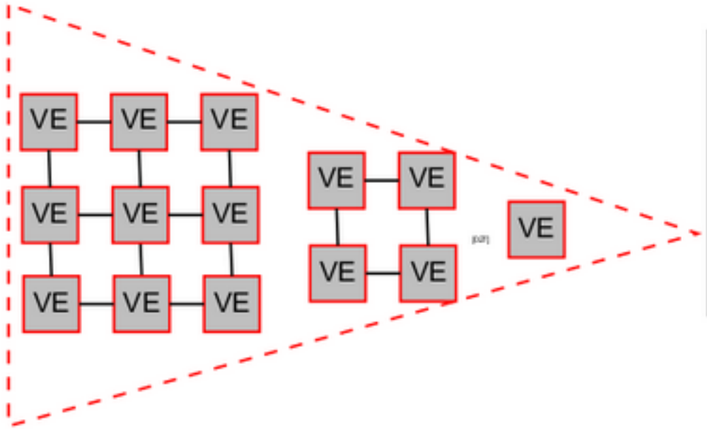
- Bei Maschennetz-Architekturen führen häufig alle VEs gleiche Instruktion durch - ungeeignetes Programmiermodell für Highlevel-Algorithmen (SIMD).
- Grenzwert: jede VE verarbeitet nur 1-Bit Daten, $N_{VE} >> 1000$
- Läßt sich Datenmatrix nicht vollständig auf Netzstruktur partitionieren, ist Effizienz ebenfalls reduziert.

RECHNERARCHITEKTUR FÜR VISION-SYSTEME (II)

Pyramidennetzwerk

- Irreguläre dreidimensionale Verbindungsstruktur mit großer Anzahl von Verarbeitungseinheiten (VE), mehrstufig in (quadratischen) Ebenen angeordnet.
  - » Jede VE ist mit seinen (4) direkten Nachbarn auf gleicher Ebene,
  - » jeweils mit (4) VEs der unteren Ebene und (1) der oberen Ebene verknüpft (Σ=9).
  - » Eine Pyramide besitzt 1/2log(N+1) Ebenen bei *N* VEs.
- Gut geeignet für Divide-and-Conquer-Verfahren, wo ein Problem immer weiter mit einem Teilungsfaktor 2 zerlegt wird.
- Nachteil: hoher Verbindungsaufwand (Netzwerk).
- Auf verschiedenen Ebenen können Bilder mit unterschiedlicher Auflösung/Matrixgröße parallel verarbeitet werden.

RECHNERARCHITEKTUR FÜR VISION-SYSTEME (II)



- Eine Pyramidenarchitektur ist geeignet, alle Ebenen der Algorithmik im Vision-Bereich effizient bearbeiten zu können.
- Sowohl SIMD- als auch MIMD- (verschiedene VEs bearbeiten unterschiedliche Instruktionen) Betrieb ist möglich.

RECHNERARCHITEKTUR FÜR VISION-SYSTEME - NETRA

NETRA ist ein rekursiv definierter und hierarchischer Multiprozessor speziell für Vision-Systeme entwickelt.

DSP (Distributing-and-Scheduling-Processors)
Taskverteilung und Kontrolleinheiten. Erlauben dynamische Partitionierung (zeitlich- wie räumlich) von Programmcode auf Verarbeitungseinheiten.

PE (Processing Element)
Verarbeitungseinheiten. Organisiert in Clustern der Größe 16-64 PEs. Bis 150 Cluster können gebildet werden. Jedes PE ist ein generischer High-Perfomance-Prozessor mit schneller FPU (Floating-Point-Unit).

C (Cluster)
Bindung von PEs zu einer Einheit. Jeder Cluster teilt sich einen gemeinsamen Datenspeicher. Dieser Speicher ist in einer Pipeline angeordnet. Alle PEs können über einen Kreuzmatrixschalter C mit den Datenspeichern M verbunden werden; die DSPs sind auch über C mit den PEs verknüpfbar.

RECHNERARCHITEKTUR FÜR VISION-SYSTEME - NETRA

IC (Interconnect)

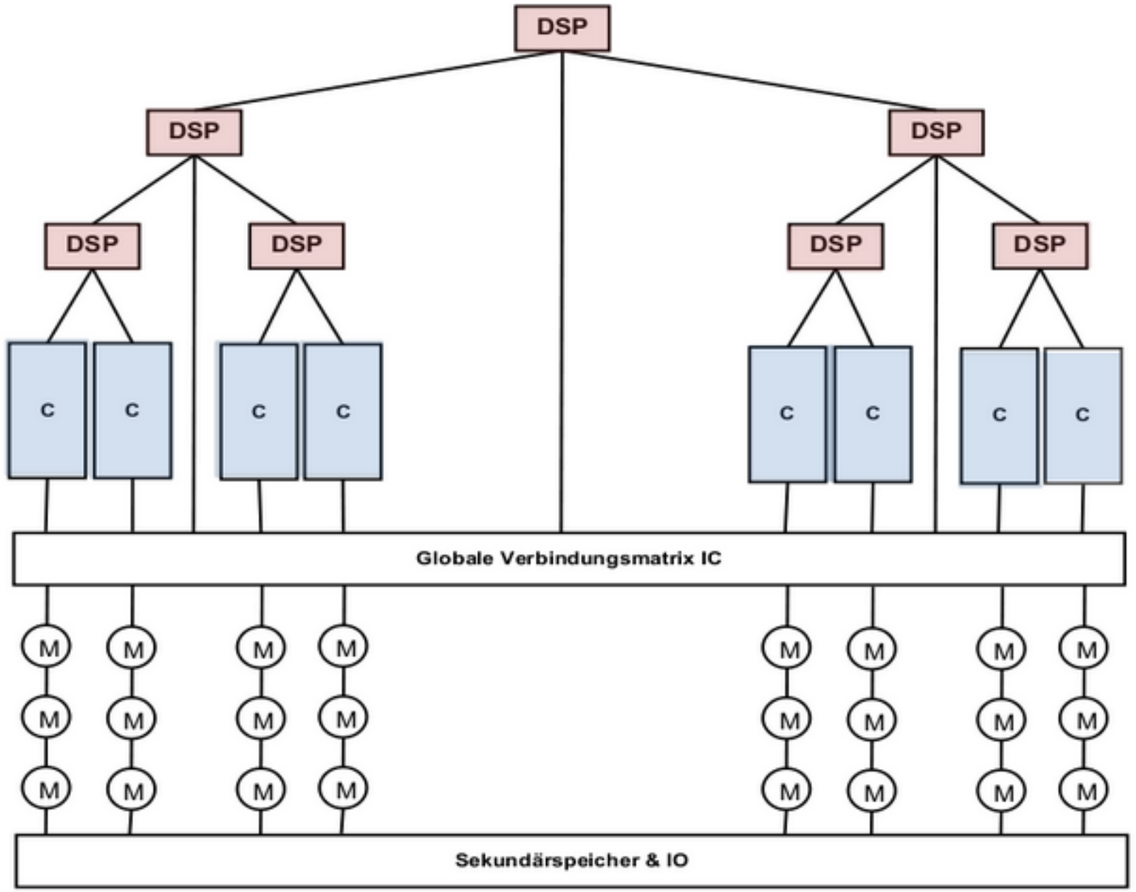
Globale Verbindungsmatrix die die Cluster mit Speichermodulen M verbindet. Aufgebaut als vollständiges unidirektionales Verbindungsnetz M×N mit einer Kreuzschaltermatrix. Vollständiges Verbindungsnetz benötigt keine Synchronisation (Mutex/Arbiter).

M

Speichermodul welches parallelen und geteilten Zugriff unterstützt. Die Speichermodule M sind in einer Pipeline angeordnet. Die Zugriffszeit eines Moduls sei T Taktzyklen. Jede Pipeline besteht aus T Speichermodulen. Die Speicher-Pipeline kann daher einen Speicherzugriff pro Taktzyklus durchführen!

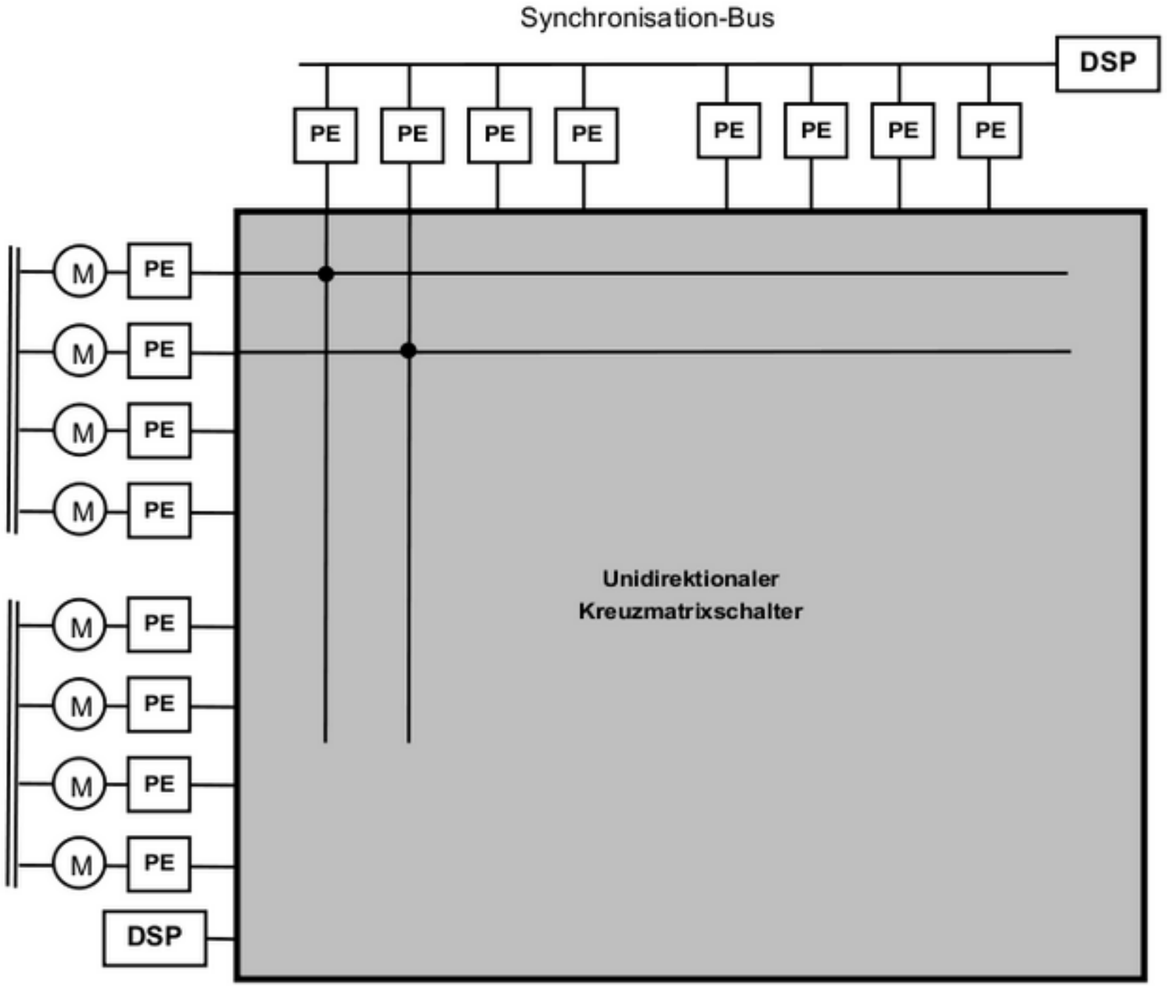
RECHNERARCHITEKTUR FÜR VISION-SYSTEME - NETRA

Systemarchitektur



RECHNERARCHITEKTUR FÜR VISION-SYSTEME - NETRA

Prozessorcluster



KOMMUNIKATIONSSTRUKTUREN

- Ein Verbindungsnetz (Netzwerk) ist das Medium, über das die Kommunikation der Verarbeitungs- und Kontrolleinheiten VE&KE (Prozessoren) untereinander und der Zugriff auf gemeinsame Ressourcen wie Hauptspeicher stattfindet.

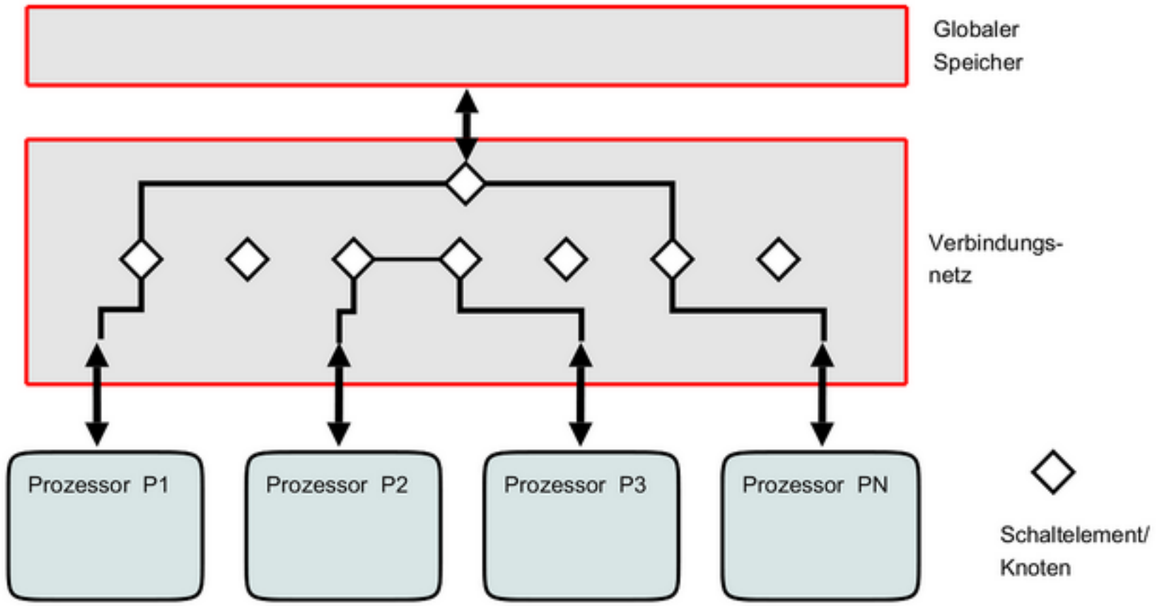


Abb. 58. Verbinungsnetz der Parallel Random Access Machine (PRAM)

KOMMUNIKATIONSSTRUKTUREN

Metriken

Komplexität

Hardware-Aufwand für das Verbindungsnetz gemessen an der Anzahl und Art der Schaltelemente und Verbindungen.

Verbindungsgrad

Der Verbindungsgrad eines Kommunikationsknoten beschreibt die Anzahl der Verbindungen, die von dem Knoten zu anderen Knoten bestehen.

Geometrische Ausdehnung

Maximale Distanz für die Kommunikation (Anzahl der Knoten oder Schaltelemente) zwischen zwei Kommunikationsteilnehmern (VE/KE) [maximale Pfadlänge].

Regelmäßigkeit

Regelmäßige Strukturen in der Verbindungsmatrix lassen sich i.A. einfacher implementieren und modellieren (simulieren).

Erweiterbarkeit

Dynamische oder statische Anpassung an Veränderung der Parallelarchitektur.

KOMMUNIKATIONSSTRUKTUREN

Latenz, Durchsatz/Übertragungsbandbreite

Maximale, meist theoretisch errechnete Übertragungsleistung des Verbindungsnetzes oder einzelner Verbindungen in [MBits/s]

Skalierbarkeit

Fähigkeit eines Verbindungsnetzes, bei steigender Erhöhung der Knotenzahl die wesentlichen Eigenschaften beizubehalten, wie z.B. den Datendurchsatz oder die Kommunikationslatenz.

Blockierung

Ein Verbindungsnetz heißt blockierungsfrei, falls jede gewünschte Verbindung zwischen zwei Kommunikationsteilnehmern (inklusive Speicher) unabhängig von bestehenden Verbindungen hergestellt werden kann.

Ausfalltoleranz oder Redundanz

Möglichkeit, Verbindungen zwischen einzelnen Knoten selbst dann noch zu schalten, wenn einzelne Elemente des Netzes (Schaltelemente, Leitungen) ausfallen.

» Ein fehlertolerantes Netz muss also zwischen jedem Paar von Knoten mindestens einen zweiten, redundanten Weg bereitstellen.

KOMMUNIKATIONSSTRUKTUREN

Komplexität der Wegfindung

Art und Weise, wie zwischen einem Kommunikationsstart- und endpunkt eine Nachricht vom Sender zum Zielknoten bestimmt wird: Routing. Die Wegfindung sollte einfach sein, und mittels eines schnellen Hardware-Algorithmus in jedem Verbindungselement implementierbar sein.

KOMMUNIKATIONSSTRUKTUREN

Kommunikationsklassen

Durchschalte-oder Leitungsvermittlung

Direkte Verbindung zwischen Sender und Empfänger wird für die Zeit der Informationübertragung hergestellt.

- » Durchschaltnetze erreichen i.A. höhrer Durchsatzraten als Netze mit Paketvermittlung, aber Dauerverbindung reduziert u.U. Anzahl der möglichen Verbindungen.

Paketvermittlung

Daten werden in Nachrichten (Paketen) gekapselt, i.A. mit fester Länge, und vom Sender an der Empfänger übertragen. Eine Datenkodierung und -dekodierung ist notwendig. Weiterhin ist i.A. eine Datenfragmentierung und -defragmentierung erforderlich. Die Nachrichten werden vom Sender zum Empfänger entsprechend einem Wegfindealgorithmus zwischen einzelnen Knoten geroutet.

- » Erhöhter Verwaltungsaufwand im Vergleich zu Durchschaltenetzen.

KOMMUNIKATIONSSTRUKTUREN

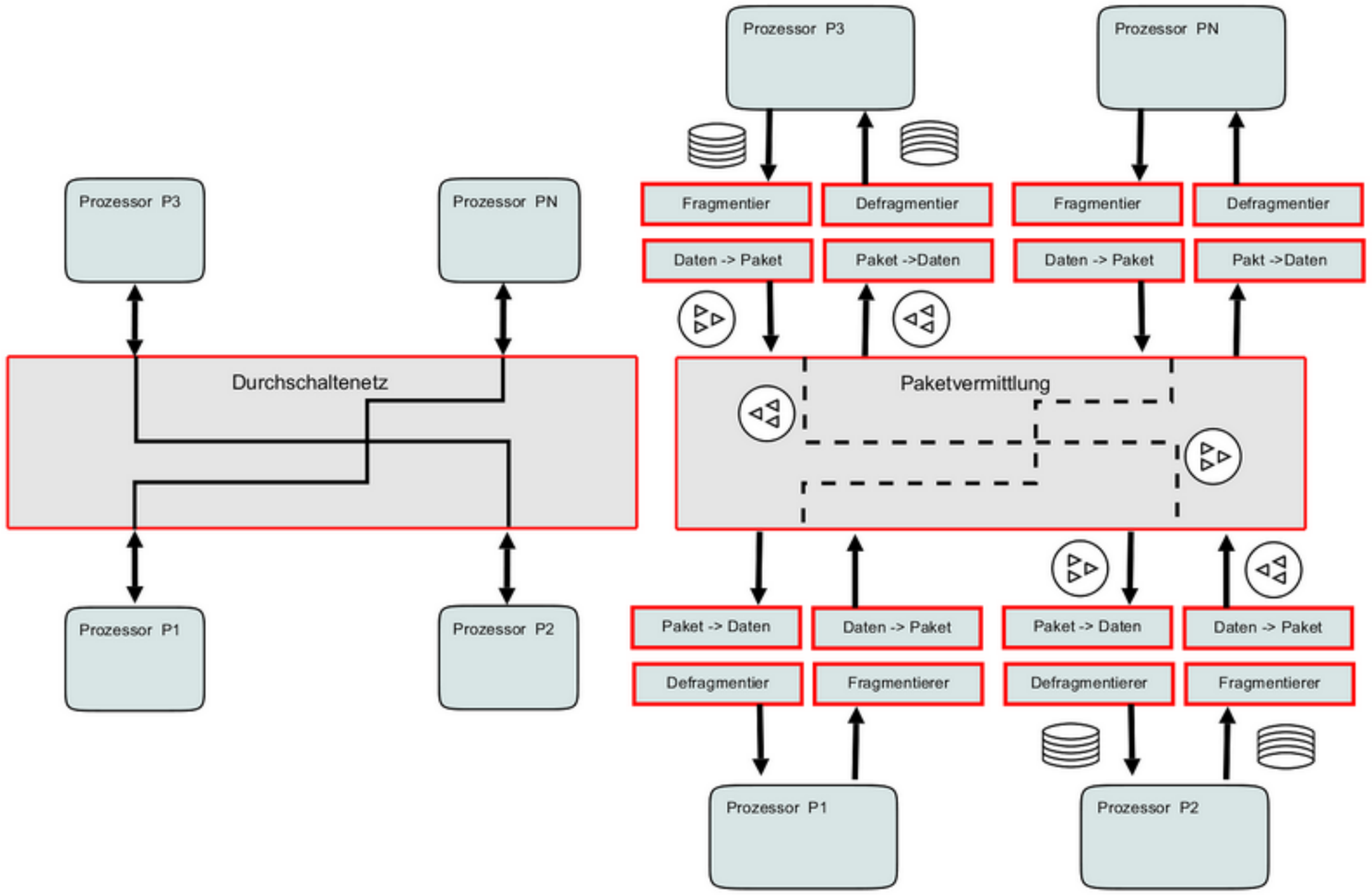


Abb. 59. Kommunikationsklassen: Schalt- zu Paketvermittlung

KOMMUNIKATIONSSTRUKTUREN

Vorteile der Schaltnetze

- Keine Protokollimplementierung für den Nachrichtenaustausch erforderlich.
- Die maximale Übertragungsbandbreite (Durchsatz) eines Kommunikationskanals kann erreicht werden.

Nachteile der Schaltnetze

- Jeder Knoten muss mit N:1-Multiplexern ausgestattet werden.
- Eine universelle $N \otimes N$ - Schaltmatrix (jeder Kommunikationsteilnehmer N_i kann mit jedem anderen Teilnehmer N_j mit $i \neq j$ verbunden werden) erfordert großen Hardware-
- Aufwand, und ist bei großen N nicht mehr wirtschaftlich. Synchronisation muss explizit erfolgen.

KOMMUNIKATIONSSTRUKTUREN

Vorteile der Paketvermittlung und Nachrichtenaustausch

- Über einen Kommunikationskanal können Verbindungen zwischen verschiedenen Kommunikationsteilnehmern aufgebaut werden.
- Auch komplexe Netze können mit geringen Hardware-Aufwand (Routing) realisiert werden.
- Implizite Synchronisation bereits vorhanden.

Nachteile der Paketvermittlung und Nachrichtenaustausch

- Implementierung eines Protokollstacks mit Datenfrag- und defragmentierung und Routing.
- Geringerer Netto-Datendurchsatz durch Header- und Trailerdaten im Nachrichtenpaket.
- Höhere Übertragungslatenz durch Paketadministration und Routing.

Beispiel: Eine Implementierung eines IP-Protokollstacks mit Digitallogik benötigt ca. 1M Gates (ca. 4 Millionen Transistoren)!

NETZWERKARCHITEKTUREN UND TOPOLOGIEN

Ein Netzwerk besteht aus Knoten N={n1,n2,…} und Verbindungen (Kanten) C={c1,c2,..}, die die Knoten untereinander verbinden. Das Netzwerk kann als ein Graph G(N,C) beschrieben werden, der i. A. zyklisch und gerichtet ist.

Parameter und Metriken von Netzwerken

Knotenzahl N

Gesamtzahl der Knoten, die ein Netzwerk oder ein Subnetzwerk bilden

Knotengrad (Konnektivität) K

Jeder Knoten hat mindestens eine, maximal K Verbindungen zu Nachbarknoten

Skalierbarkeit

Lässt sich das Netzwerk für belibiege Anzahl N von Knoten erweitern?

Routing

Strategie für die Weiterleitung und Zustellung von Daten von einem Senderknoten ns zu einem oder mehreren Empfängerknoten {ne1,ne2,…}

NETZWERKARCHITEKTUREN UND TOPOLOGIEN

- Unicast: nur ein Empfänger
- Multicast: eine Gruppe von ausgezeichneten Empfängern
- Broadcast: alle Knoten (in einem Bereich) sind Empfänger

Ausdehnung D

Maximaler Abstand (Distanz) zwischen zwei Knoten

Kosten O

Anzahl der Kanten NC

Effizienz η

Anzahl der Kanten NC im Verhältnis zur Anzahl der Knoten N

Latenz τ

Verzögerung τ bei der Zustellung von Daten durch Vermittlung (normiert und vereinfacht in Anzahl von Zwischenknoten)

Durchsatz B

Datendurchsatz (Bandbreite)

NETZWERKARCHITEKTUREN UND TOPOLOGIEN

- In nachrichtenbasierten Vermittlungsnetzwerken ist ein Knoten eines Netzwerkes:
  1. Quelle von Nachrichten, die Daten kapseln,
  2. Empfänger von Nachrichten, und
  3. Vermittler (Router) von Nachrichten (Zwischenknoten)
- In einem geschalteten Netzwerk mit direkter Datenvermittlung ist ein Knoten eines Netzwerkes
  1. Quelle von Daten, und
  2. Empfänger von Daten

DYNAMISCHE VERBINDUNGSNETZWERKE

- Direkte Schaltung von Verbindungen ermöglicht die Übertragung von einem Sender- zu einem Empfängerknoten.

Crossbar Switch

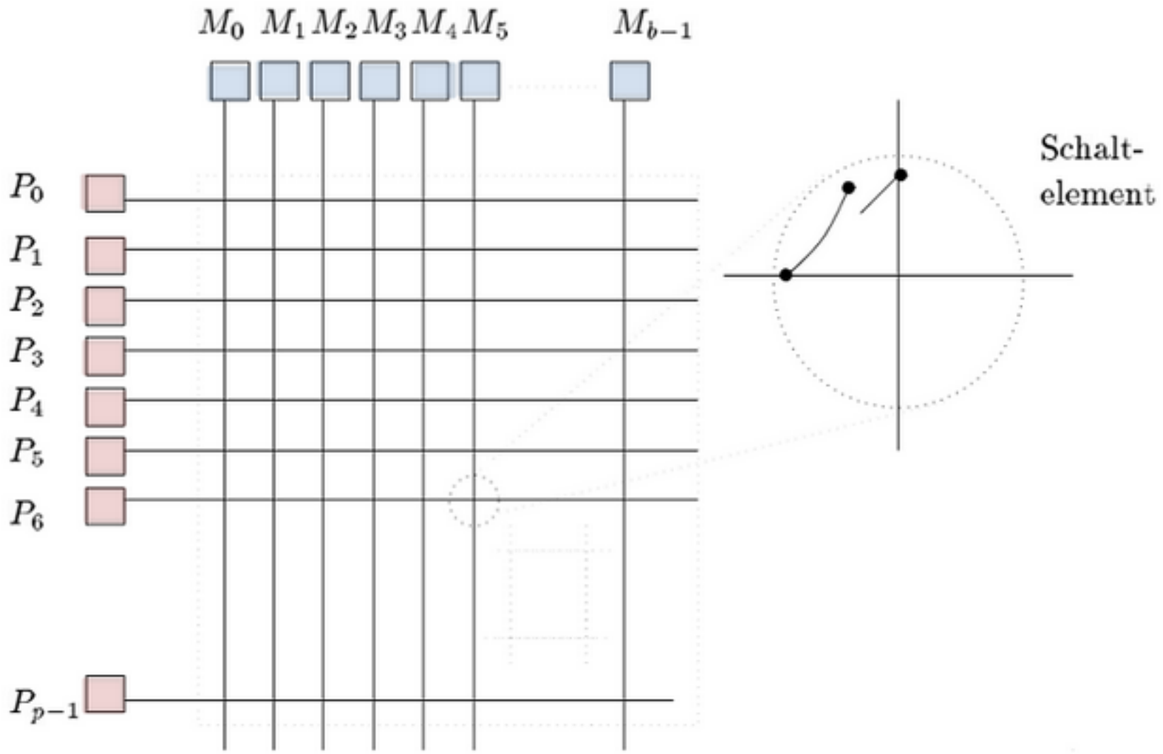


Abb. 60. Vollständiger Kreuzschalter (Crossbar Switch) mit Schaltelementen [PA, Vornberger,1998]; Hier Verbindung Prozessor P mit Speichermodul M

DYNAMISCHE VERBINDUNGSNETZWERKE

Parameter

- Knotenzahl: N
- Knotengrad: 1
- Skalierbar: Ja
- Routing: Nicht erforderlich - konfliktfrei
- Ausdehnung: 1
- Kosten: N2

Vorteile

- Jeder Knoten kann mit jedem anderen Knoten jederzeit verbunden (geschaltet) werden → Dynamische Konnektivität des Netzwerkes
- Ausdehnung optimal klein (und Latenz minimal)
- Keine Nachrichtenkapselung und kein Kommunikationsprotokoll erforderlich
- Skalierung bezüglich Leistung ist gut

DYNAMISCHE VERBINDUNGSNETZWERKE

Nachteile

- Hohe Kosten und hoher Hardware-Aufwand, der quadratisch mit der Anzahl der Knoten wächst
- Skalierung bezüglich Kosten ist schlecht
- Fehlende Synchronisation zwischen Sender und Empfänger

Mehrstufen (Multistage) Verbindungsnetzwerke

- Mehrstufiger Aufbau des Netzwerkes
- Kompromiss bezüglich Skalierung zwischen Kreuzschaltern und Bussystemen

Parameter

- Knotenzahl: N
- Knotengrad: 1
- Skalierbar: Ja
- Routing: Nicht erforderlich - aber nicht konfliktfrei!
- Ausdehnung: 1 (oder n - Stufenzahl)
- Kosten: $N\log_2 N$

DYNAMISCHE VERBINDUNGSNETZWERKE

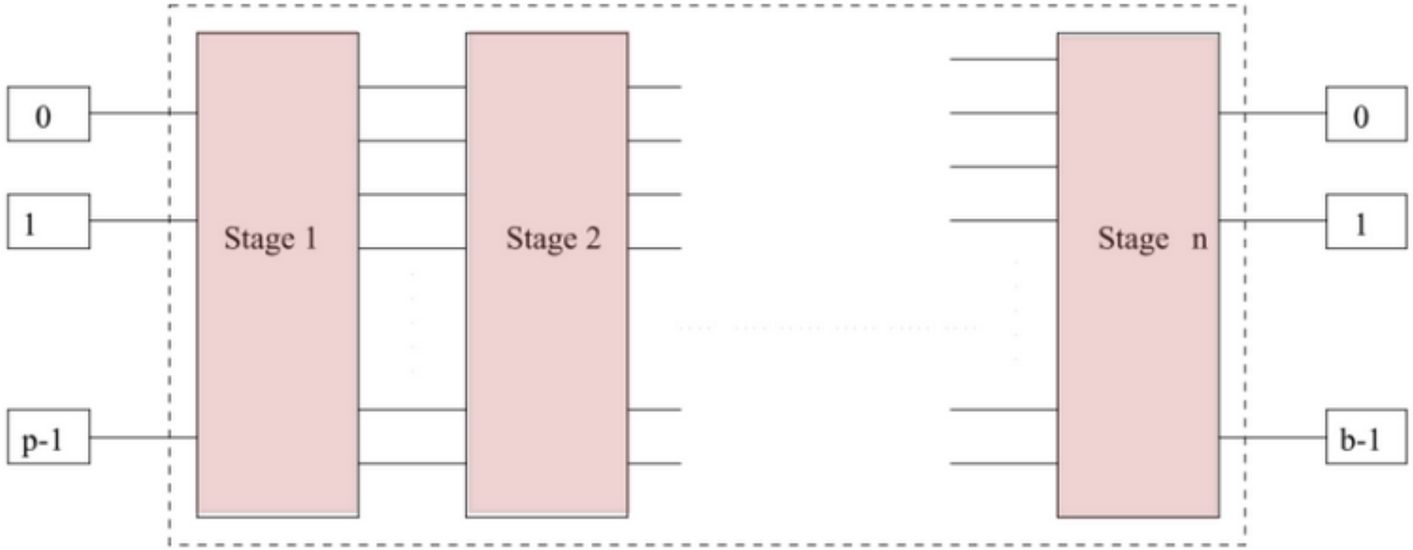


Abb. 61. Aufbau eines Mehrstufen Netzwerks (n-stufig) [PA, Vornberger, 1998]

DYNAMISCHE VERBINDUNGSNETZWERKE

Mehrstufiges Permutations-Netzwerk

- Jede Stufe besteht aus Binärschaltern, die entweder durchgeschaltet oder gekreuzt geschaltet sind.
- Die Ausgänge einer Stufe k werden über ein Permutationsverfahren mit den Eingängen der nächsten Stufe k+1 verbunden.

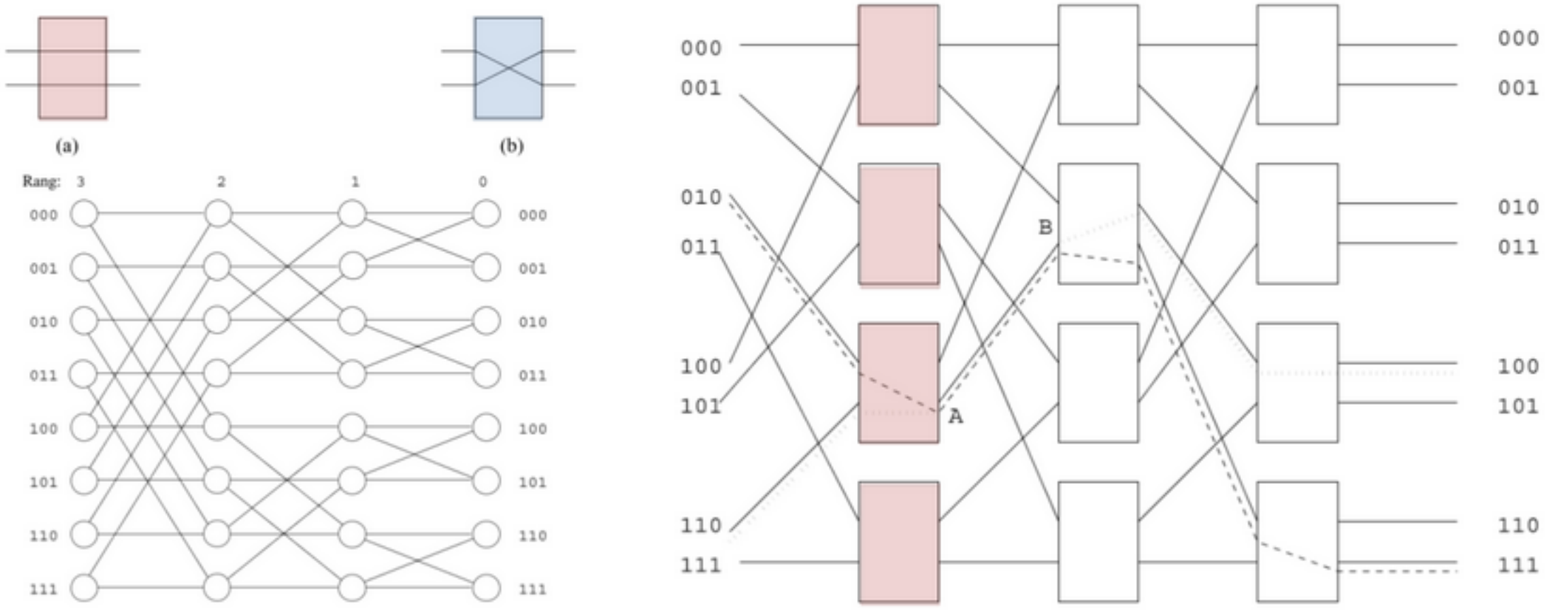


Abb. 62. Nicht konfliktfreies Mehrstufennetzwerk [PA, Vornberger, 1998]

DYNAMISCHE VERBINDUNGSNETZWERKE

Bus

- Alle Knoten nutzen eine gemeinsame Kommunikationsverbindung
- Bus-basierte Netzwerke skalieren bezüglich Kosten gut, aber nicht bezüglich der Leistung (Flaschenhals!)

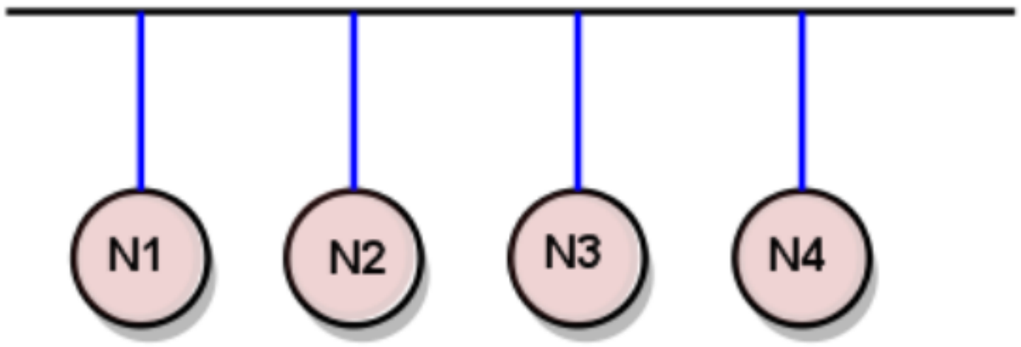


Abb. 63. Topologie des Bus-basierten Netzwerkes

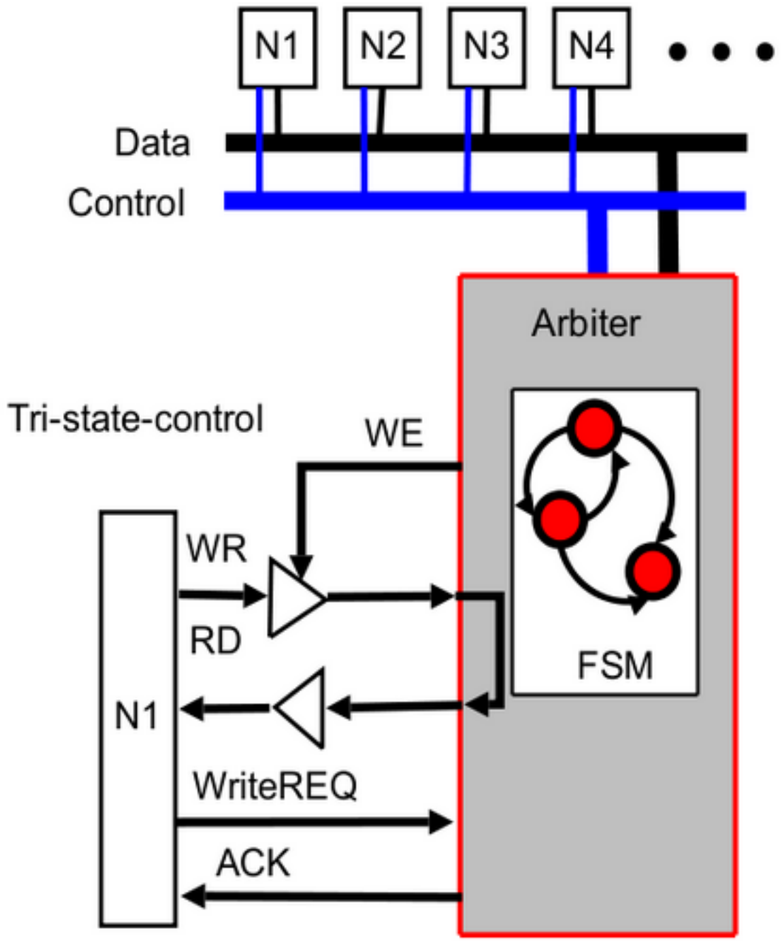
DYNAMISCHE VERBINDUNGSNETZWERKE

Parameter

- Knotenzahl: N
- Knotengrad: 1
- Skalierbar: Jein
- Routing: Nicht erforderlich - aber nicht konfliktfrei
- Ausdehnung: 1
- Kosten: 1

DYNAMISCHE VERBINDUNGSNETZWERKE

Bus Arbiter und Architektur



Protokoll

LISTEN

Aktueller Eigentümer des Busses (ACK=1) sendet eine Listen Nachricht an alle Teilnehmer. Nachfolgend wird nur der adressierte Teilnehmer Daten empfangen.

TALK

Der aktuelle Eigentümer des Busses gibt seine Adresse bekannt.

UNLISTEN

Alle Teilnehmer warten auf Kontrollkommandos

UNTALK

Kein Teilnehmer ist Datensender.

DYNAMISCHE VERBINDUNGSNETZWERKE

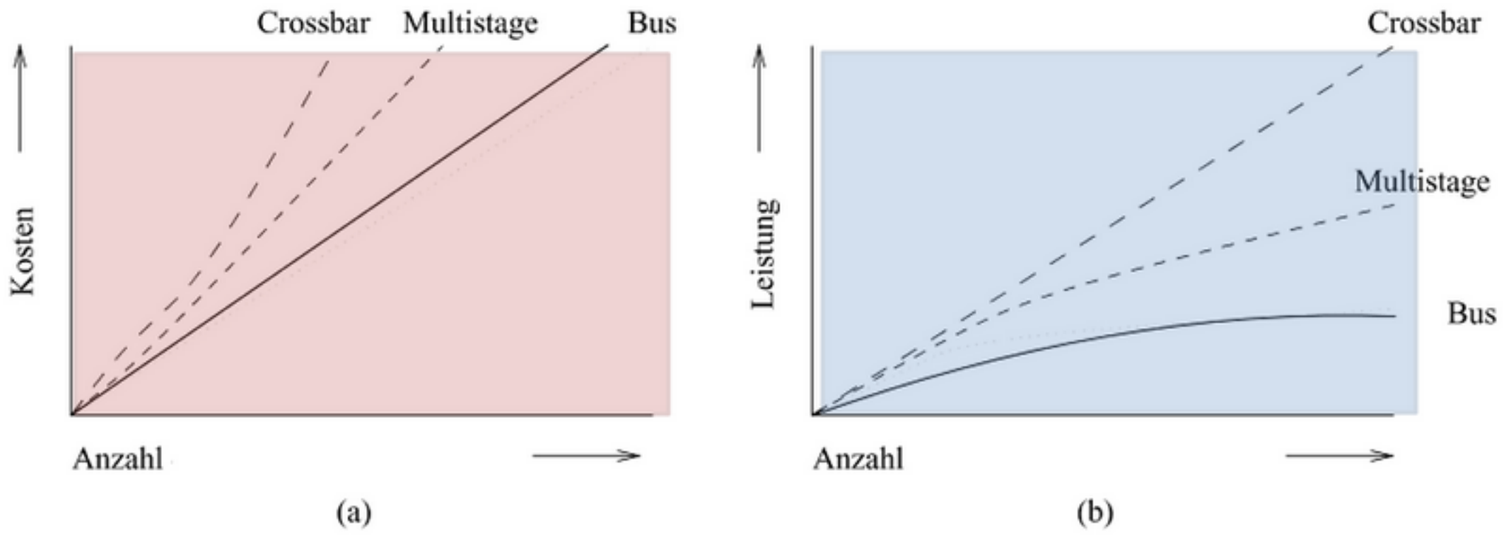


Abb. 64. Skalierung von Kosten und Leistung der verschiedenen dyn. Verbindungsnetze in Abhängigkeit der Knotenzahl *N* [PA, Vornberger, 1998]

STATISCHE VERBINDUNGSNETZWERKE

- Nachrichtenbasierte Kommunikation, d.h. die zu übertragenen Daten werden in Nachrichten gekapselt
- In den meisten Topologien ist die Vermittlung von Nachrichten durch andere Knoten entlang eines Pfades (S→ni1→ni2→…→E) erforderlich

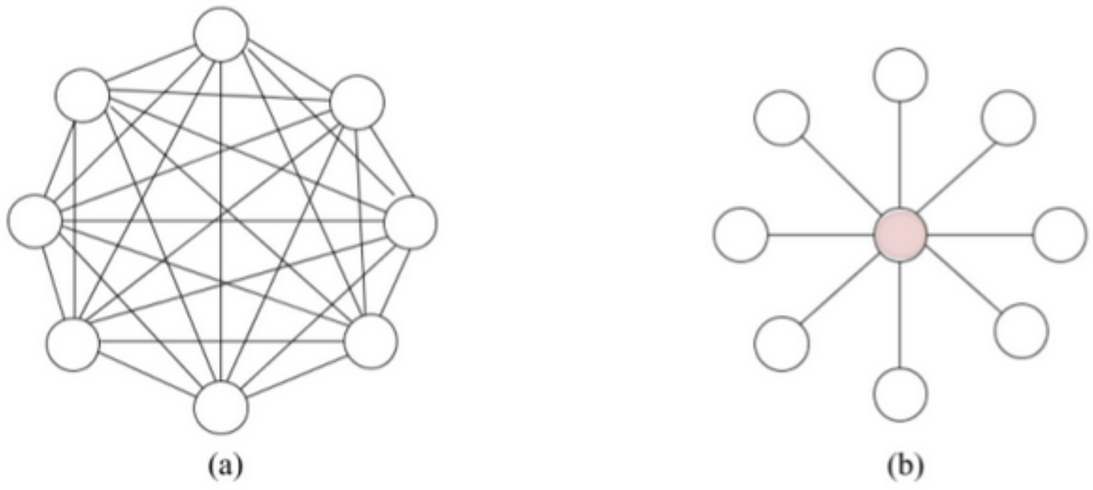


Abb. 65. Zwei Grenzfälle: Vollständig verbundenes Maschennetzwerk und Sternnetzwerk

STATISCHE VERBINDUNGSNETZWERKE

Sternnetzwerk

- Master-Slave Netzwerk Hierarchie
- Ein ausgezeichneter Knoten ist Master, der mit jedem anderen Knoten (Slave) verbunden ist
- Kommunikation findet immer über den Master (=Router) statt

Parameter

» Knotenzahl: N
» Knotengrad: 1 (Slave), N-1 (Master)
» Skalierbar: Ja
» Routing: erforderlich - *wähle Ziel in zwei Schritten über Master*
» Ausdehnung: 2
» Geschlossen: nein
» Kosten: N-1

STATISCHE VERBINDUNGSNETZWERKE

Binärer Baum

Parameter

- Knotenzahl: N
- Knotengrad: 3
- Skalierbar: Ja
- Routing: erforderlich - *laufe vom Start aufwärts zum gemeinsamen Vorfahren und dann abwärts zum Ziel*
- Ausdehnung: 2*k (k - Höhe des Baums)
- Geschlossen: nein
- Kosten: $N\log_2 N$

STATISCHE VERBINDUNGSNETZWERKE

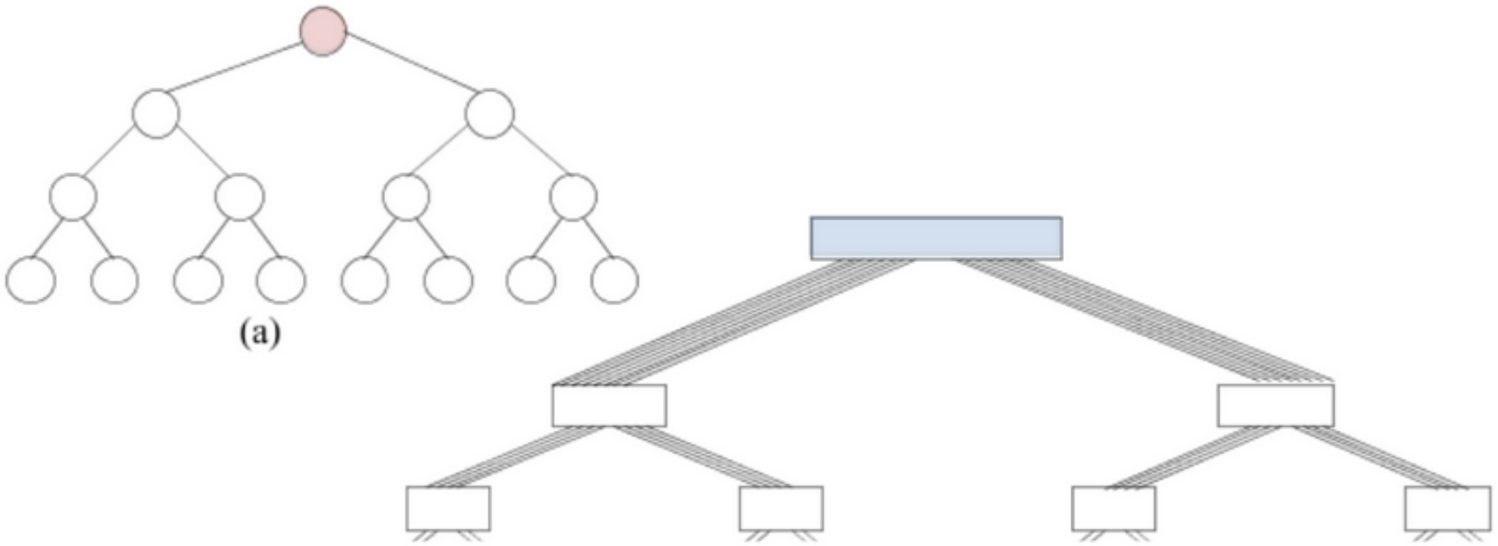


Abb. 66. Topologie Binärbaum (a) und N-Baum (b) Netzwerk [PA, Vornberger, 1998]

STATISCHE VERBINDUNGSNETZWERKE

Lineares Array - Kette - und Ring

Parameter

- Knotenzahl: N
- Knotengrad: 2
- Skalierbar: Ja
- Routing: erforderlich - *wähle Richtung und laufe in diese Richtung*
- Ausdehnung: N-1 (Kette), N/2 (Ring)
- Geschlossen: nein (Kette), ja (Ring)
- Kosten: N-1 (Kette), N (Ring)

Abb. 67. Kette und Ring (technologisch schwierig) [PA, Vornberger, 1998]

STATISCHE VERBINDUNGSNETZWERKE

2D Gitter

- ökonomisch und techn. geeignet für material-integrierte verdrahtete Sensornetzwerke (elektrische und optische Kommunikationstechnologien)

Parameter

- Knotenzahl: N
- Knotengrad: 4
- Skalierbar: Ja
- Routing: erforderlich (Δ-Distanz Routing)
- Ausdehnung: $2*(\sqrt{N}-1)$(ohne wraparound), $1+(\sqrt{N}-1)$ (mit wraparound)
- Geschlossen: nein (ohne wraparound), ja (mit wraparound)
- Kosten: $2(N-\sqrt{N})$ (ohne wraparound)

STATISCHE VERBINDUNGSNETZWERKE

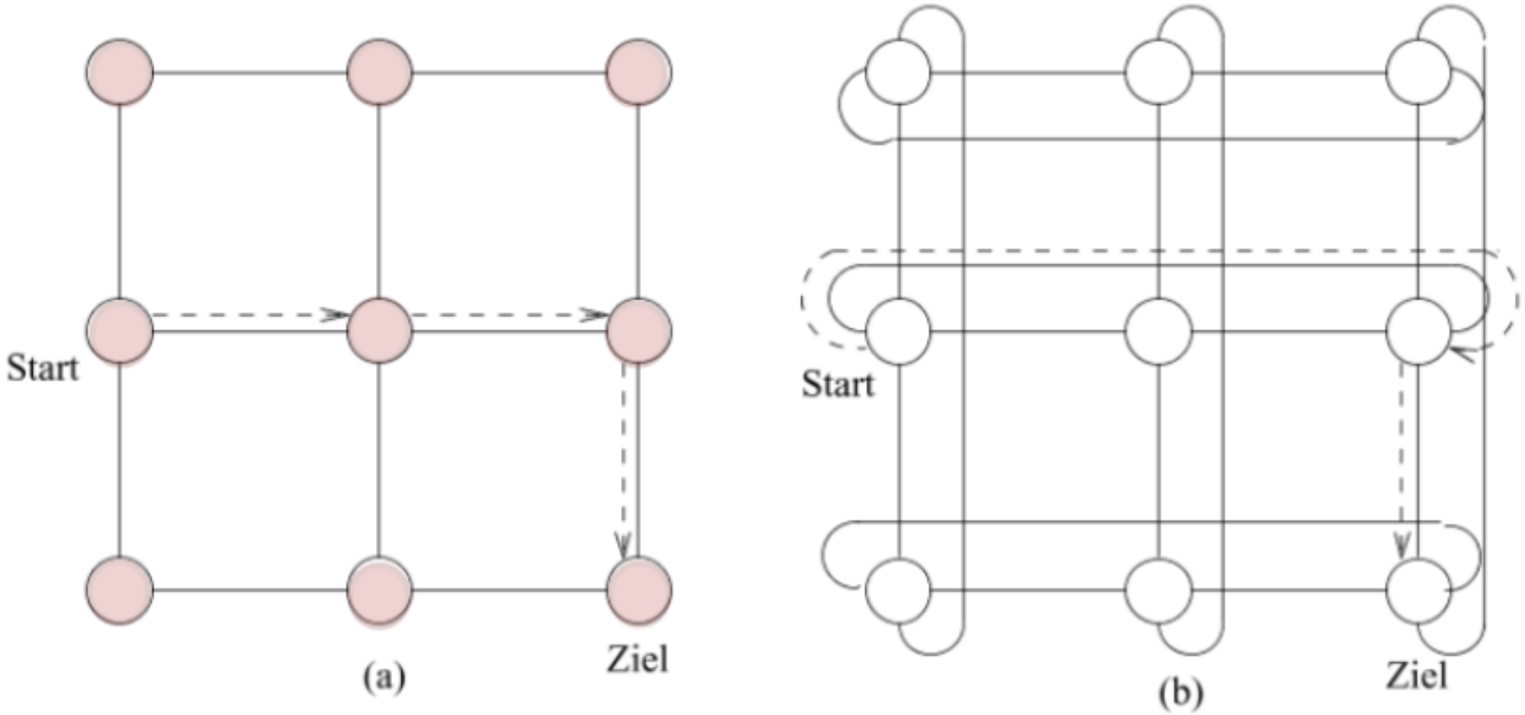


Abb. 68. 2D Gitter und Torus (technologisch schwierig) [PA, Vornberger, 1998]

STATISCHE VERBINDUNGSNETZWERKE

Δ-Distanz Routing

- Routing: Wandere horizontal (erste Dimension) bis zur Zielspalte, dann wandere vertikal bis zur Zielzeile (zweite Dimension)
- Einfaches Δ-Distanz Routing in m-dimensionalen Gitternetzwerken:
  - M: Nachricht
  - moveto: vermittelt Nachricht an Nachbarknoten in Richtung *DIR*
  - Δ0: relative Distanz zwischen Sender und Empfänger in Knoteneinheiten
  - Δ: bei jeder Vermittlung geänderter Vektor, anfänglich Δ0=Δ, am Ziel: Δ=(0,0,..)
  - Δi: i-te Komponente von Δ
  - Nullvektor **0**=(0,0,..) → Ziel erreicht!

STATISCHE VERBINDUNGSNETZWERKE

```
TYPE DIM = {1,2,...,m}
TYPE DIR = {NORTH,SOUTH,EAST,WEST,UP,DOWN,...}
// Numerical mapping of directions
DEF dir = function(i) →
  match i with
  -2→NORTH | -1→WEST | 1→EAST | 2→SOUTH ...
// Routing and message passing
DEF route_xy = function (Δ,Δ0,M) →
  if Δ ≠ 0 then
    for first i ∈ DIM with Δi ≠ 0 do
      if Δi > 0 then
        moveto(dir(i));
        route_xy(Δ with Δi:=Δi-1,Δ0,M)
      else if Δi < 0 then
        moveto(dir(-i));
        route_xy(Δ with Δi:=Δi+1,Δ0,M)
  else deliver(M)
```

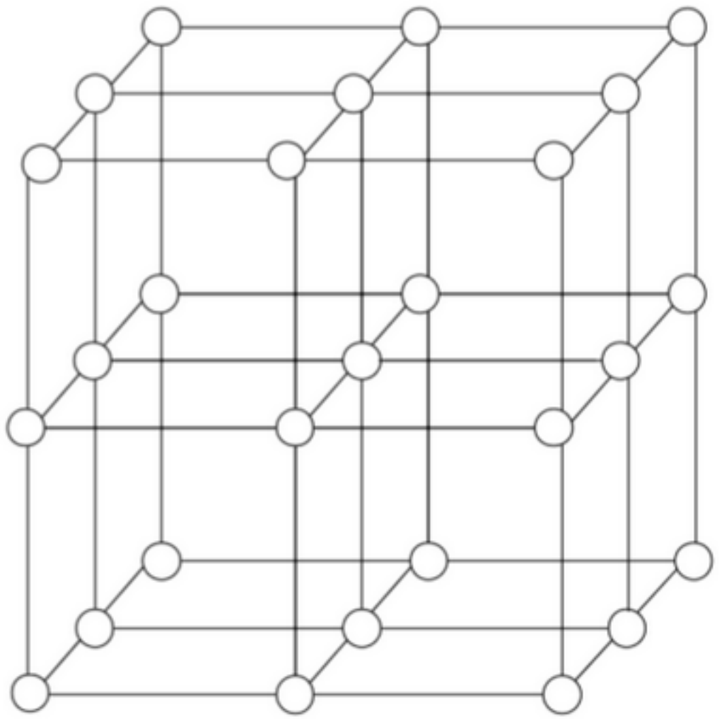
STATISCHE VERBINDUNGSNETZWERKE

3D Gitter

- ökonomisch und techn. geeignet für material-integrierte verdrahtete Sensornetzwerke

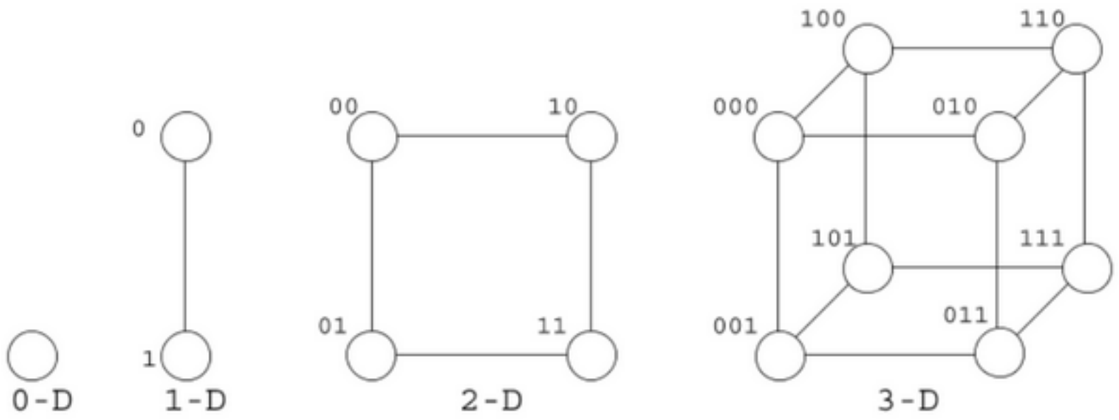
Parameter

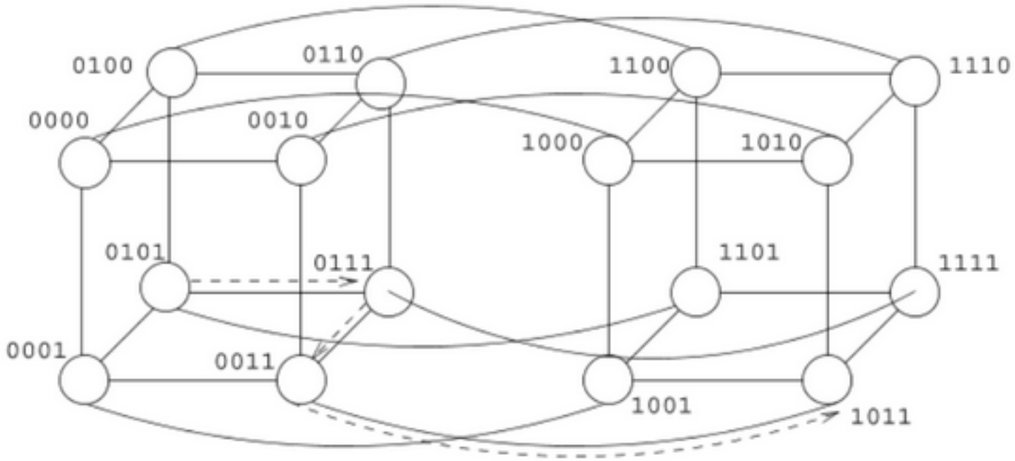
» Knotenzahl: N
» Knotengrad: 6
» Skalierbar: Ja
» Routing: erforderlich (Δ-Distanz Routing)
» Ausdehnung: $3*(\sqrt{3N}-1)$
» Kosten: $3(N-\sqrt{3N})$



STATISCHE VERBINDUNGSNETZWERKE

Zusammenfassung Gitternetzwerke





4-D Hypercube

KOMMUNIKATIONSSTRUKTUREN - ROUTING

Routing: Flusssteuerung bei nachrichtengekoppelten Netzen

Store-and-forward

- Nachricht wird von jedem Zwischenknoten in Empfang genommen
- Nachricht wird vollständig zwischengespeichert
- Sendung der Nachricht auf Pfad zum nächsten Knoten nach vollständigen Empfang

Virtual-cut-through

- Nachricht wird als Kette von Übertragungseinheiten transportiert
- Kopfteil der Nachricht enthält die Empfängeradresse und bestimmt den Weg
- Bei Ankunft der ersten Übertragungseinheit wird diese sofort an den nächsten Knoten weitergeleitet (Pipeline-Verfahren)
- Nachrichtenspeicherung nur im Konfliktfall (wenn Pfad belegt ist)

KOMMUNIKATIONSSTRUKTUREN - ROUTING

Wormhole-routing

- Ist der Pfad zum nächsten Knoten nicht belegt, Verhalten wie Cut-through
- Ist Kommunikationskanal belegt, müssen alle vorherigen Knoten ebenfalls Datentransfer einstellen, d.h. die Vorgänger-Knoten werden blockiert.
- D.h. keine Nachrichtenspeicherung in den Verbindungsknoten
- Es werden nur Puffer für eine Übertragungseinheit benötigt (Kopf).

KOMMUNIKATIONSSTRUKTUREN - ROUTING

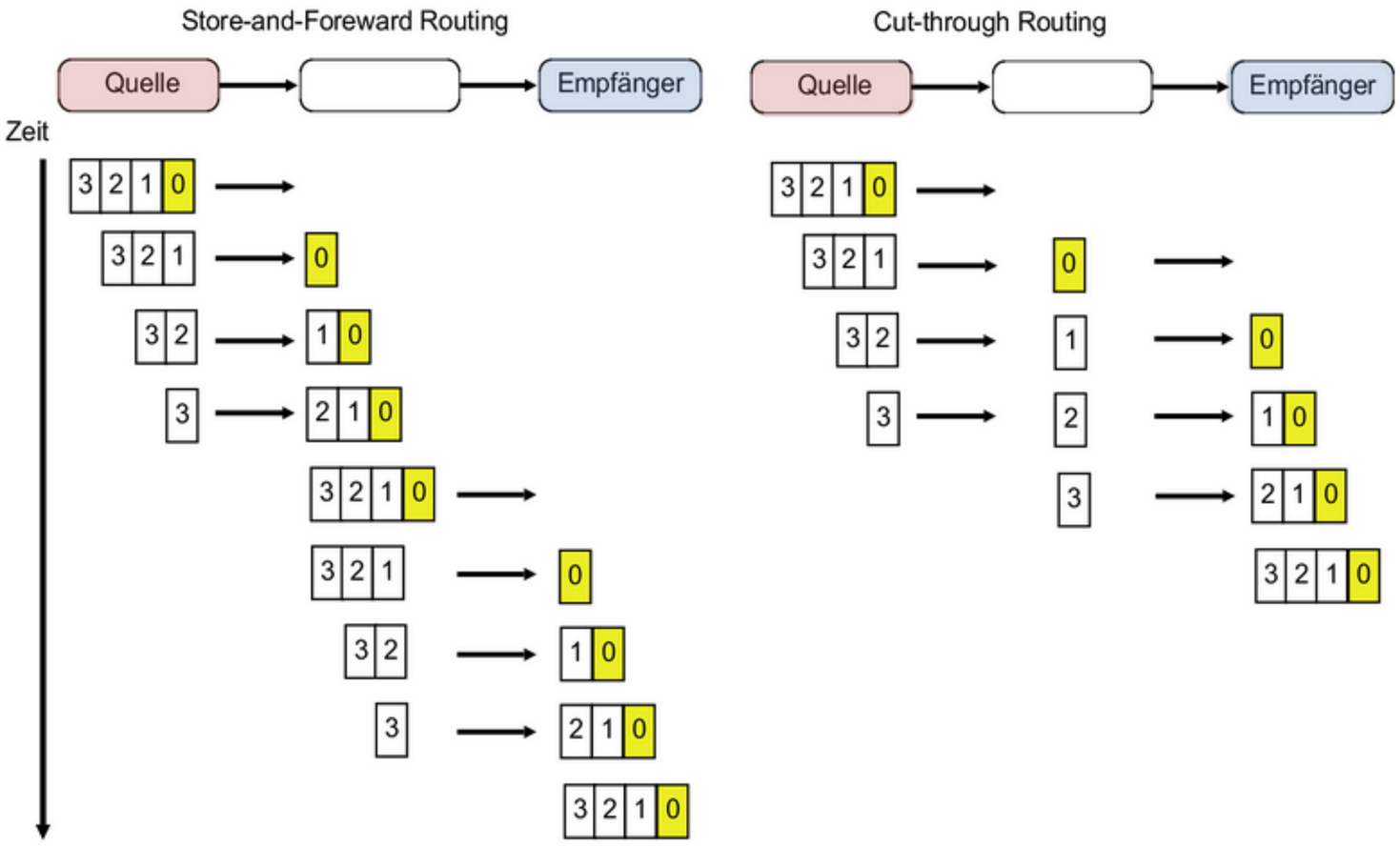


Abb. 69. Vergleich S&T und VC Routing: Das Cut-through Routing benötigt deutlich weniger Kommunikationsschritte als das Store-and-Foreward Routing-Verfahren.

KOMMUNIKATION

Paketstruktur

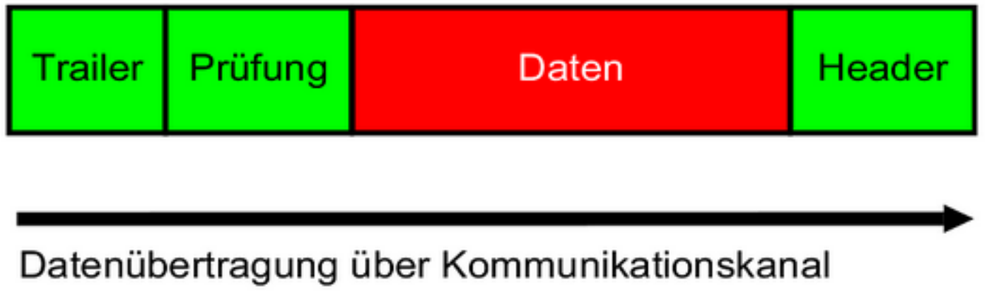


Abb. 70. Aufbau eines Nachrichtenpakets: Header (Zustell- und Absendeinfomationen, Overhead), Nutzdaten (Datenlast), und optional Trailer (Prüfsumme, EOP Marker, .., Overhead)

Latenz

Die Zeit zum Übertragen einer Nachricht zwischen zwei Netzwerkknoten mit *N* Byte setzt sich zusammen aus:

$$T(N)_{S \to D} = T_{overhead} + T_{routing} + T_{transfer} + T_{conflict}$$

KOMMUNIKATION

$T_{overhead} \rightarrow \Theta(1)$

Zeit die benötigt wird, ein Paket vom Speicher in den Kommunikationskanal und aus dem Kanal in den Speicher zu übertragen → abhängig von Hardware

$T_{transfer} \rightarrow \Theta(N)$

Zeit die zur eigentlichen Datenübertragung benötigt wird.

$T_{conflict}$

Mittlere Zeit in der ein Kommunikationskanal blockiert ist. Abhängig von Auslastung des Kanals.

- Die Transferzeit von *N* Byte Nutzdaten erhöht sich durch die Paketadministration (Header & Trailer) N_{PA} und ergibt sich aus der Bandbreite *B* [Byte/s] des Kommunikationskanals:

$$T_{transfer} = \frac{N + N_{PA}}{B}$$

KOMMUNIKATION

- Die Routingzeit (Latenz) hängt vom verwendeten Routing-Verfahren ab:

$$T_{routing-SF}(N,H) = H\left(\frac{N_{paket}}{B} + \Delta\right)$$

$$T_{routing-CS}(N,H) = \frac{N_{paket}}{B} + H\Delta$$

- Dabei ist H die Anzahl von Knoten, die ein Paket auf seiner Route durchlaufen muss.
- Ein Hardwareoverhead beim Senden und Empfangen eines Knotens wird mit Δ gekennzeichnet.

Es gilt:

$$T_{routing-CS}(N,H) < T_{routing-SF}(N,H)$$

KOMMUNIKATION - ZUSAMMENFASSUNG

Hardware

- Es gibt zwei wesentliche Architekturklassen die parallele und verteilte Datenverarbeitung sehr stark beeinflusst → Performanz → maximaler / effektiver Speedup:
  - » Rechnerarchitektur (Horizontale und vertikale Parallelisierung, Speicher)
  - » Kommunikationsarchitektur (Art und Topologie), Ein- und Ausgabegeräte, Konflikte und Blockierung

Software

- Auf der algorithmischen Seite beeinflussen folgenden Klassen die Performanz, Effizienz, die Rechenzeit, und den Speicherbedarf von parallelen und verteilten Systemen:
  - » Partitionierung des Algorithmus in Prozesse
  - » Datenabhängigkeiten, Datenverteilung und Datenaggregation
  - » Synchronisation → Kommunikation → Nachrichtenaustausch (explizit)
  - » Kommunikations- und Routingprotokolle!

KOMMUNIKATION - ZUSAMMENFASSUNG

Kommunikation erhöht den sequenziellen Anteil eines parallelen Programms und reduziert den (max.) Performanzgewinn gegenüber rein sequenziellen Programmen gleicher Art

VERTEILTE PROGRAMMIERUNG UND SYSTEME

PARALLELE UND VERTEILTE ARCHITEKTUREN

Functionality, Applications	Computer Clusters [10,28,38]	Peer-to-Peer Networks [34,46]	Data/ Computational Grids [6,18,51]	Cloud Platforms [1,9,11,12,30]
Architecture, Network Connectivity, and Size	Network of compute nodes interconnected by SAN, LAN, or WAN hierarchically	Flexible network of client machines logically connected by an overlay network	Heterogeneous clusters interconnected by high-speed network links over selected resource sites	Virtualized cluster of servers over data centers via SLA
Control and Resources Management	Homogeneous nodes with distributed control, running UNIX or Linux	Autonomous client nodes, free in and out, with self-organization	Centralized control, server-oriented with authenticated security	Dynamic resource provisioning of servers, storage, and networks
Applications and Network-centric Services	High-performance computing, search engines, and web services, etc.	Most appealing to business file sharing, content delivery, and social networking	Distributed supercomputing, global problem solving, and data center services	Upgraded web search, utility computing, and outsourced computing services
Representative Operational Systems	Google search engine, SunBlade, IBM Road Runner, Cray XT4, etc.	Gnutella, eMule, BitTorrent, Napster, KaZaA, Skype, JXTA	TeraGrid, GriPhyN, UK EGEE, D-Grid, ChinaGrid, etc.	Google App Engine, IBM Bluecloud, AWS, and Microsoft Azure

Abb. 71. Vergleich verschiedener paralleler und verteilter Berechnungssysteme [5]

CLUSTER COMPUTING

- Ein Computer-Cluster besteht aus miteinander verbundenen eigenständigen Computern, die kooperativ als eine einzige integrierte Computer-Ressource arbeiten.
- In der Vergangenheit haben Computer-Cluster eindrucksvolle Ergebnisse bei der Bewältigung hoher Arbeitslasten mit großen Datenmengen gezeigt.

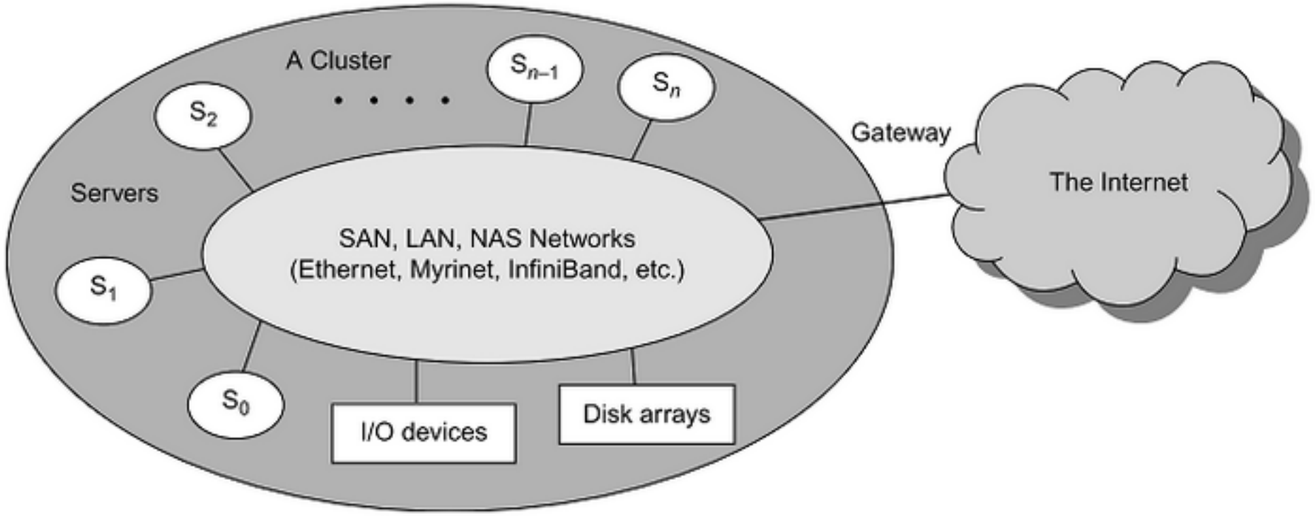


Abb. 72. Cluster von Servern mit Disk Arrays, Hochgeschwindigkeitsverbindung, Ein- und Ausgabegeräte, und Ankopplung an das Internet [5]

CLUSTER COMPUTING

- Ein idealer Cluster mit mehrere Systembilder sollte zu einem Single-System-Image (SSI) zusammengeführt werden.
- Cluster-Entwickler wünschen ein Cluster-Betriebssystem oder eine Middleware, um SSI auf verschiedenen Ebenen zu unterstützen, einschließlich der gemeinsamen Nutzung von CPUs, Arbeitsspeicher und E/A über alle Cluster-Knoten hinweg.
- Ein SSI ist eine durch Software oder Hardware erzeugte Illusion, die eine Sammlung von Ressourcen als eine integrierte Ressource darstellt.
- SSI lässt den Cluster wie eine einzige Maschine für den Benutzer erscheinen. Ein Cluster mit mehreren Systemabbildern ist nichts anderes als eine Sammlung unabhängiger Computer &arr; One Big Virtual Machine!

CLOUD COMPUTING

- Cloud Computing bedeutet u.A. die Bereitstellung von virtualisierten Ressourcen von Rechenzentren zu einer Internet-Cloud, die mit Hardware, Software, Speicher, Netzwerk und Services für bezahlte Benutzer zur Verfügung gestellt wird, um ihre Anwendungen auszuführen.

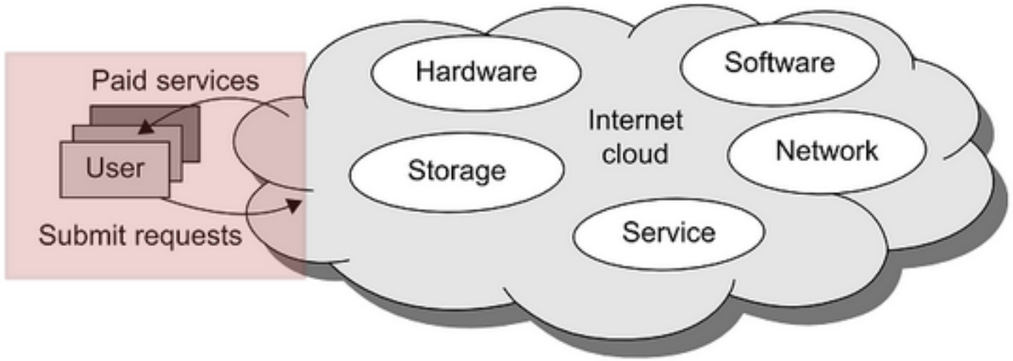


Abb. 73. Nutzer können virtualisierte Ressourcen anfragen → Auslagerung von Berechung!

VIRTUELLE NETZWERKE

- Neben der Virtualisierung von Ressourcen wie Speicher und CPUs ist die Virtualisierung und Transformation der Netzwerkstrukturen zentrale Methodik

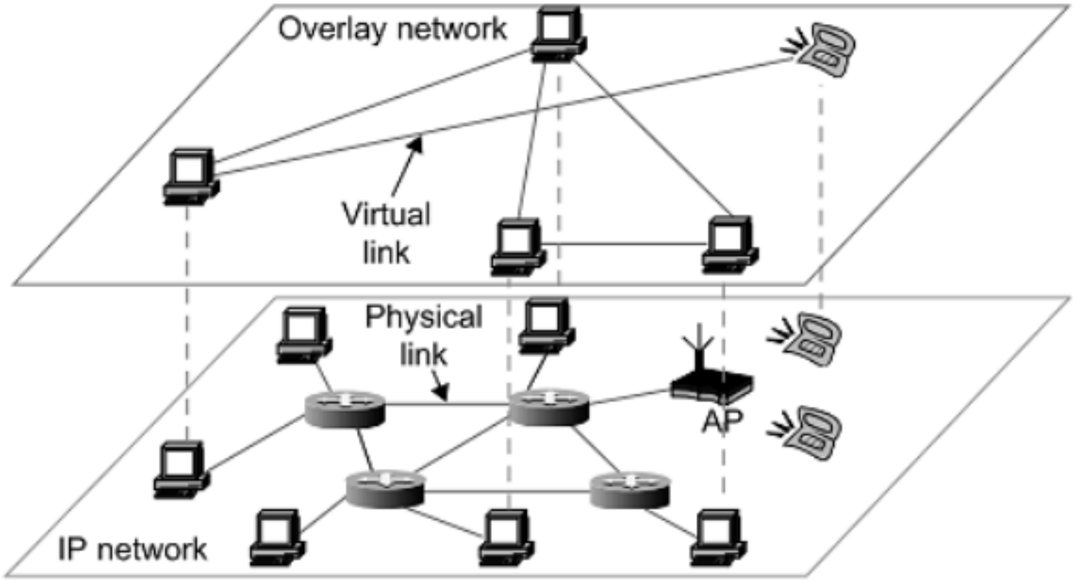


Abb. 74. Die Struktur eines P2P-Systems durch Abbildung eines physischen IP-Netzwerks auf ein Overlay-Netzwerk mit virtuellen Verbindungen. [5]

PARTITIONIERUNG UND KOMMUNIKATION

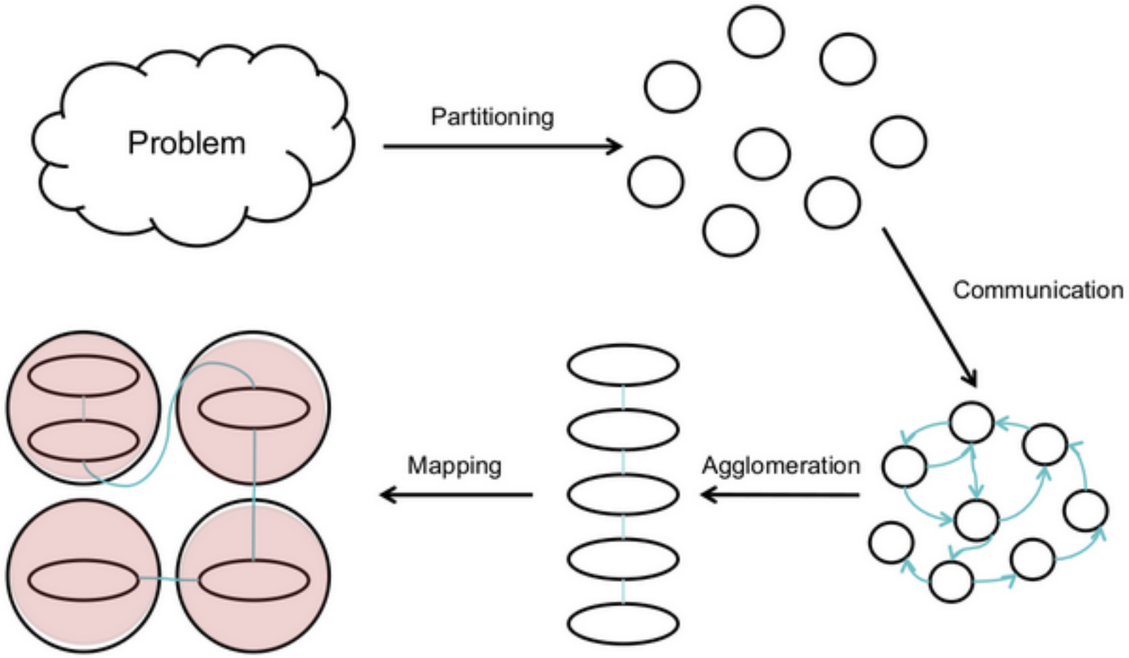


Abb. 75. Problemzerlegung durch Partitionierung → Abbildung auf Kommunikationsnetzwerke [Janetzko, MPI Practice 2014]

GUSTAFSON GESETZ

- In parallelen Systemen drückt das Amdahlsche Gesetz den maximalen Speedup in Abhängigkeit vom sequenziellen Programmteil (bzw. dessen Ausführung) aus
  - » Ein hoher Speedup und daher eine effiziente Ausnutzung (Workload) aller Verarbeitungseinheiten/Rechner ist nur durch Minimierung des sequenziellen Teils η (verursacht durch Kommunikation) möglich

Um bei Verwendung eines großen Clusters eine höhere Effizienz zu erzielen, muss die Größe des Problems so skalieren, dass sie der Cluster-Fähigkeit entspricht!

» Problem bei der Anwendung des Amdahlschen Gesetzes: Ein **konstanter Workload** wird für den sequenziellen und parallelen Anteil angenommen!

- Dies führt zu dem folgenden Beschleunigungsgesetz, das von John Gustafson (1988) vorgeschlagen wurde und als skalierte Arbeitslastbeschleunigung bezeichnet wird.

GUSTAFSON GESETZ

- Sei W die Arbeitslast in einem bestimmten Programm. Bei Verwendung eines n-Prozessor-Systems skaliert der Benutzer die Arbeitslast zu $W^* = \eta W + (1-\eta)\, nW$.
  - » Nur der parallelisierbare Teil der Auslastung wird im zweiten Ausdruck n-mal skaliert.
- Diese skalierte Arbeitslast W^* ist im Wesentlichen die sequentielle Ausführungszeit auf einem einzelnen Prozessor.
- Die parallele Ausführungszeit einer skalierten Arbeitslast W^* auf n Prozessoren wird wie folgt durch eine skalierte Arbeitslast-Beschleunigung definiert:

$$S^* = W^*/W = [\eta W + (1-\eta)nW]\,/W = \eta + (1-\eta)n$$

- Amdahls und Gustafsons Gesetze werden bei unterschiedlichen Workloads W angewendet!

MAP & REDUCE

- Ein Web-Programmiermodell für die skalierbare Datenverarbeitung in großen Clustern über große Datenmengen.
- Das Modell wird häufig in Web-Scale-Search- und Cloud-Computing-Anwendungen eingesetzt.
- Aber auch funktionale und parallele Programmierung macht von MapReduce Methoden Gebrauch
- **Methode**:
  - Es wird eine Map-Funktion angegeben, um eine Gruppe von Schlüssel/Wert-Zwischenpaaren zu generieren.
  - Dann wird eine Reduce-Funktion auf diesen Paaren angewendet, um alle Zwischenwerte mit demselben Zwischenschlüssel zusammenzuführen.
- MapReduce ist hochgradig skalierbar, um hohe Parallelitätsgrade auf verschiedenen Arbeitsebenen zu erreichen.

MAP & REDUCE

- Ein typischer MapReduce-Berechnungsprozess kann Terabytes an Daten auf Zehntausenden oder mehr Client-Computern verarbeiten. Hunderte von MapReduce-Programmen können gleichzeitig ausgeführt werden.
  - » Tatsächlich werden jeden Tag Tausende von MapReduce-Jobs in Clustern von Google ausgeführt.
  - » Das Hadoop Framework bietet für WEB Anwendungen MapReduce Services an → Master-Slave Architektur!

Die Map-Funktion verarbeitet ein Paar (Schlüssel, Wert) und gibt eine Liste von Zwischenpaaren (Schlüssel, Wert) zurück:

$$map(k_1, v_1) \rightarrow list(k_2, v_2)$$

Die Reduzierungsfunktion führt alle Zwischenwerte zusammen, die die gleichen Zwischenschlüssel haben:

$$reduce(k_2, list(v_2)) \;\rightarrow\; list(v_3)$$

MAP & REDUCE

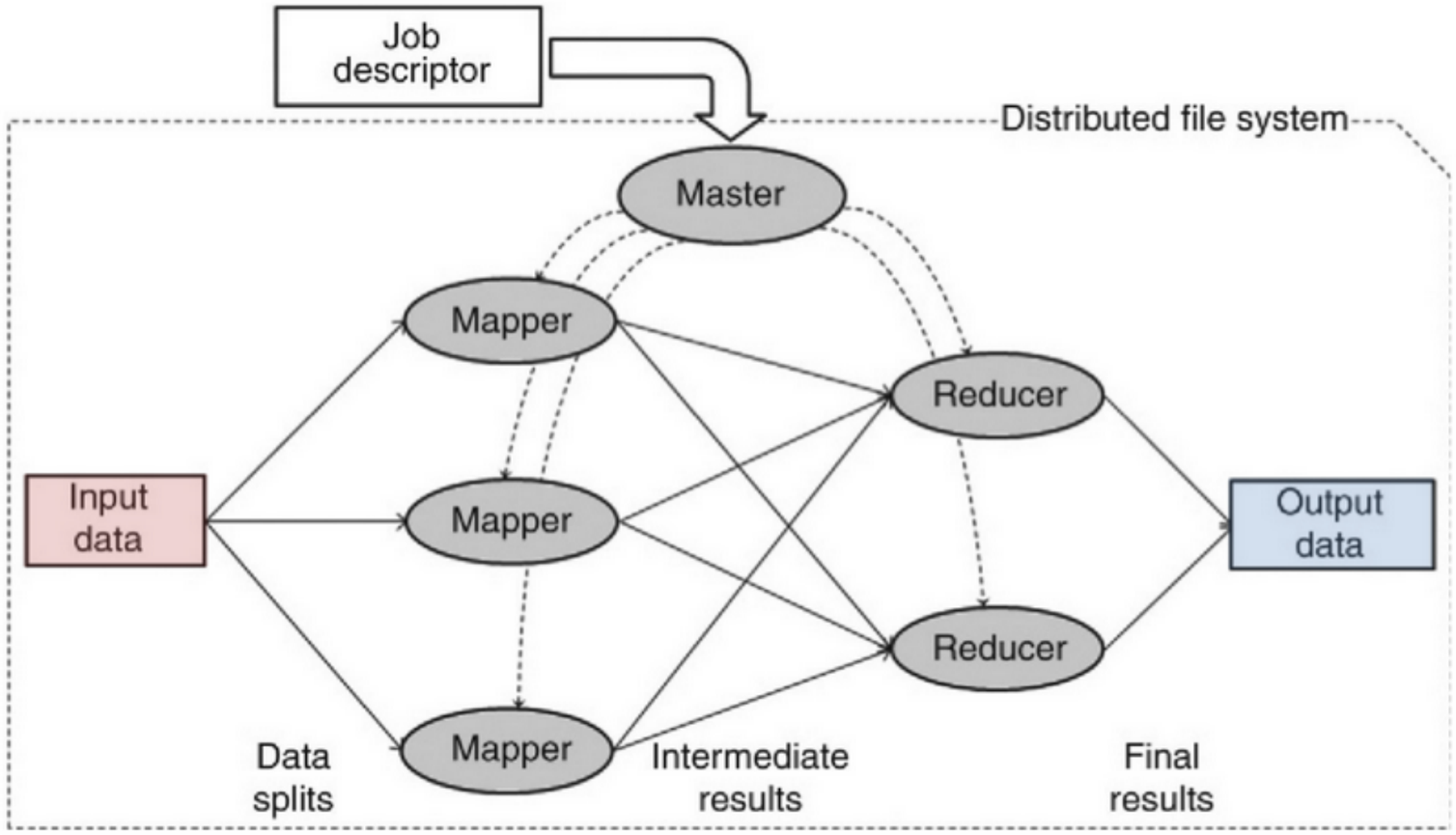


Abb. 76. Ausführungsphasen einer generischen MapReduce Applikation []

MAP & REDUCE

MapReduce Phasen

1. Ein Master-Prozess erhält einen Jobdeskriptor, der den auszuführenden MapReduce-Job angibt. Der Jobdeskriptor enthält neben anderen Informationen den Ort der Eingabedaten, auf die unter Verwendung eines verteilten Dateisystems zugegriffen werden kann.
2. Gemäß dem Jobdeskriptor startet der Master eine Anzahl von Mapper- und Reducer-Prozessen auf verschiedenen Maschinen. Gleichzeitig startet es einen Prozess, der die Eingabedaten von seinem Speicherort liest, diese Daten in eine Gruppe von Aufteilungen unterteilt und diese Aufteilungen an verschiedene Zuordner verteilt.
3. Nach dem Empfang seiner Datenpartition führt jeder Zuordnungsvorgang die Zuordnungsfunktion aus (die als Teil des Jobdeskriptors bereitgestellt wird), um eine Liste von Zwischenschlüssel / Wert-Paaren zu erzeugen. Dann werden diese Paare auf der Basis ihrer Schlüssel gruppiert.

MAP & REDUCE

4. Alle Paare mit den gleichen Schlüsseln sind dem gleichen Reduziervorgang zugeordnet. Daher führt jeder Reduzierprozess die Reduktionsfunktion (definiert durch den Jobdeskriptor) aus, die alle Werte vereinigt, die mit dem gleichen Schlüssel assoziiert sind, um einen möglicherweise kleineren Satz von Werten zu erzeugen.
5. Dann werden die von jedem Reduktionsprozess erzeugten Ergebnisse gesammelt und an einen durch den Jobdeskriptor spezifizierten Ort geliefert, um die endgültigen Ausgabedaten zu bilden.

MAP & REDUCE

Beispiel

```
// Sequential Processing
function fib(n) {
  return n < 2 ? 1 : fib(n - 1) + fib(n - 2);
}
function sum(a,b) { return a+b }
var data = [41,42,43,44,45,46,47,48]
var res = data.map(fib).reduce(sum)
console.log(res)

// Parallel Processing
var Parallel = require('parallel.lib')
var p = new Parallel(data,options)
function sum(av) { return av[0]+av[1] }
p.map(fib).reduce(sum).then(console.log)
```

MPI

Message Passing Interface: MPI

Ziele und Eigenschaften

- Anwendungsprogrammierschnittstelle (nicht unbedingt für Compiler oder eine Systemimplementierungsbibliothek).
- Effiziente Kommunikation:
  - » Vermeidung von Arbeitsspeicher-Arbeitsspeicher Kopien
  - » Überlappung von Berechnung und Kommunikation
  - » Auslagerung auf Kommunikations-Coprozessoren, soweit verfügbar.
- Implementierungen, die in einer heterogenen Umgebung verwendet werden können (verschiedene Hostplattformen).
- Einfache Einbindung in Programmiersprachen und Bibliotheken und Plattformunabhängigkeit
- Zuverlässige Kommunikation: Nutzer/Programmier muss sich nicht um Kommunikationsfehler kümmern

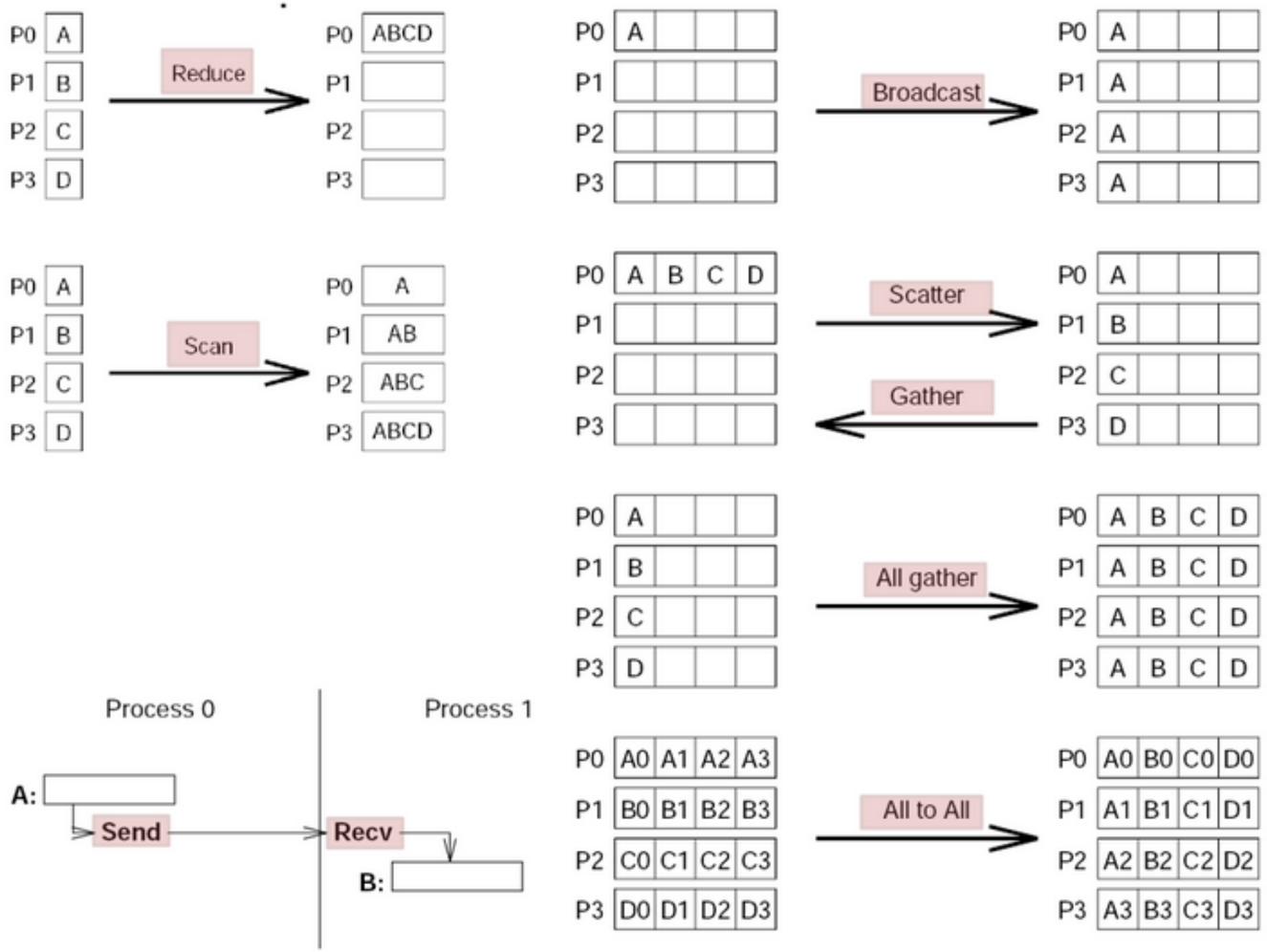
MPI

API

- Point-to-point communication,
- Datatypes,
- Collective operations,
- Process groups,
- Communication contexts,
- Process topologies,
- Environmental management and inquiry,
- The Info object,
- Process creation and management,
- One-sided communication,
- External interfaces,
- Parallel file I/O,

MPI

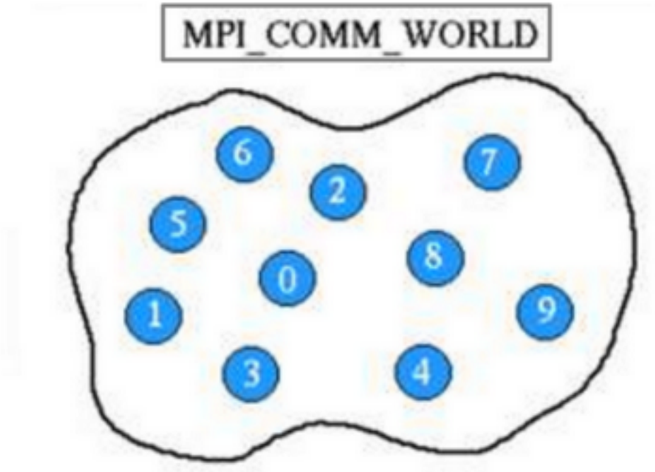
Operationen



MPI

Communicator

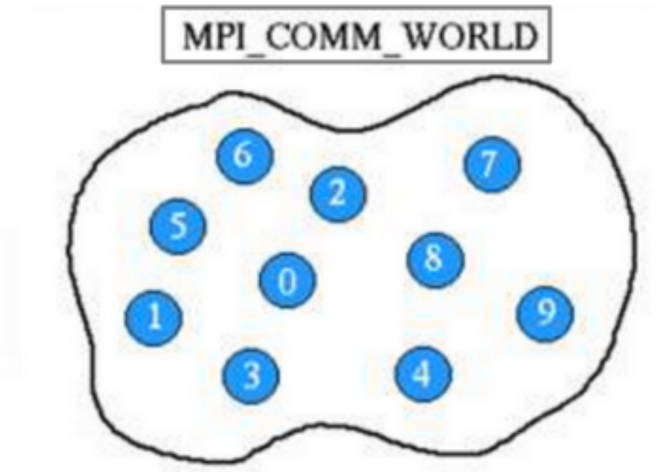
- Kommunikationswelt mit einer Gruppe aus Prozessen die gemeinsam Nachrichten austauschen können
- Nachrichten können nur innerhalb der Kommunikationswelt ausgetauscht werden
- MPI_COMM_WORLD ist die Standardwelt



MPI

Rank

- Einheitliche Prozessnummer innerhalb einer Kommunikationswelt
- Werden vom System bei der Initialisierung vergeben und werden fortlaufend ab 0 nummeriert
- Rank IDs werden bei der Kommunikation zur Identifikation von Empfänger und Sender verwendet (Kommunikationsendpunkte)
- Rank IDs dienen zur Programmdifferenzierung (*if rank==0 then do this else do that*)

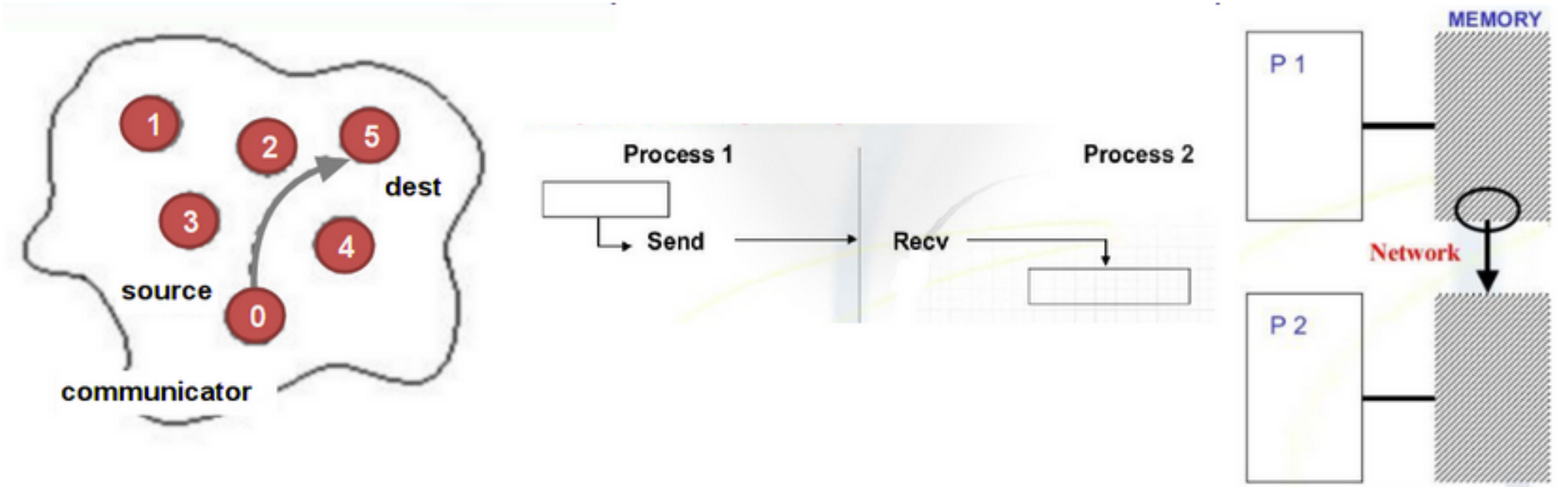


MPI

Point-to-Point Kommunikation

- Kommunikation zwischen zwei Prozessen
- Quellprozess sendet eine Nachricht (Typ, Daten) an einen Zielprozess unter Angabe der Rank ID
- Kommunikation kann nur innerhalb eines Communicators stattfinden

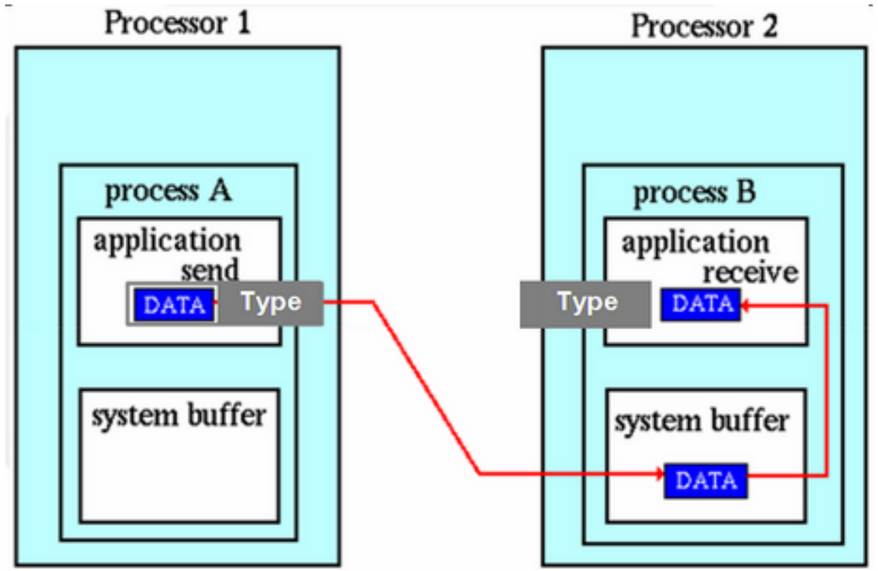
```
MPI.send(dest:number,message:{type:string,content:string})
```



MPI

- Damit der Zielprozess die Nachricht empfangen kann muss er einen Handler für den entsprechenden Nachrichtentyp einrichten:

```
MPI.recv(type:string,callback:function (message))
```



MPI

Broadcast Kommunikation

- Ein Quellprozess kann eine Nachricht an alle Prozesse innerhalb eines Communicators senden

```
MPI.broadcast(message:{type:string,content:string})
```

SICHERHEIT UND LEBENDIGKEIT

- Ein Algorithmus, z.B. die Wahl eines Leaders in einer Prozessgruppe, gelte als **sicher** wenn *maximal ein Leader* unter allen Umständen gewählt wird. Dies ist auch der Fall wenn einzelne Prozesse der Prozessgruppe fehlerhaft sind (nicht erreichbar sind oder terminiert sind).
- Ein Algorithmus, z.B. die Wahl eines Leaders in einer Prozessgruppe, gelte als **lebendig** wenn *irgendwann mindestens ein Leader* unter allen Umständen gewählt wird.

Robustheit und Fehler

- Angenommen eine Prozessgruppe bestehe aus *N* Prozessen die in einem Netzwerk verteilt sind.
- Die Netzwerkknoten sind miteinander verbunden.
- Nun wird aufgrund einer technischen Störung das Netzwerk partitioniert (z.B. in zwei getrennte Bereiche geteilt).

SICHERHEIT UND LEBENDIGKEIT

- Der verteilte Algorithmus wird in jeder der Partitionen unabhängig arbeiten. Dann ist das verteilte System zwar noch lebendig (es werden unabhängig zwei Leader in den Gruppenpartitionen gewählt), aber nicht mehr sicher !!!!
  - » Die Invariante des Algorithmus ist verletzt worden durch Ausfall/Störung!

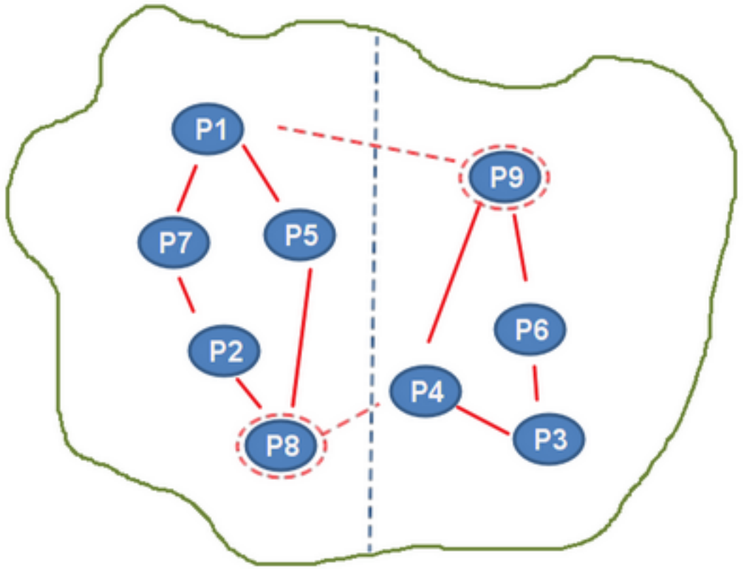


Abb. 77. Leaderelection: Eine ursprünglich zusammenhängende Gruppe wird durch Netzwerkpartitionierung (Störung der Kommunikation) zweigeteilt und es werden jetzt zwei Leader gewählt!

MUTUALER AUSSCHLUSS

Verteilter Algorithmus nach Lamport

LA1: Um einen kritischen Bereich zu erlangen (Mutex Acquire), sendet ein Prozess eine zeitgestempelte Anforderung an jeden anderen Prozess im System und fügt die Anforderung auch in seiner lokalen Queue *Q* hinzu.

LA2: Wenn ein Prozess eine Anforderung empfängt, wird sie in *Q* platziert. Wenn sich der Prozess nicht in seinem *CS* befindet, sendet er eine zeitgestempelte Bestätigung an den Absender. Andernfalls wird das Senden der Bestätigung bis zum Verlassen des *CS* verzögert (Mutex Release).

LA3: Ein Prozess tritt in seinen *CS* ein, wenn (1) seine Anfrage vor allen anderen Anfragen (d.h. der Zeitstempel seiner eigenen Anfrage ist kleiner als die Zeitstempel aller anderen Anfragen) in seinem lokalen *Q* angeordnet ist und (2) Es hat die Antworten von jedem anderen Prozess als Antwort auf seine aktuelle Anfrage erhalten.

LA4: Um die *CS* zu verlassen, löscht ein Prozess (1) die Anfrage von seiner lokalen Warteschlange *Q* und (2) sendet eine zeitgestempelte Freigabenachricht an alle anderen Prozesse.

MUTUALER AUSSCHLUSS

LA5: Wenn ein Prozess eine Freigabenachricht erhält, entfernt er die entsprechende Anforderung aus seiner lokalen Warteschlange Q.

Verteilter Algorithmus nach Ricart–Agrawala'

- Verbesserte Version

RA1: Jeder Prozess, der den Eintritt in seinen *CS* anfordert (Mutex Acquire), sendet eine zeitgestempelte Anfrage an jeden anderen Prozess im System.

RA2: Ein Prozess, der eine Anforderung empfängt, sendet eine Bestätigung an den Absender zurück, nur wenn

- (1) der Prozess nicht an dem Eintritt in seinen *CS* interessiert ist (Mutex Acquire) oder
- (2) der Prozess versucht, seine *CS* zu erlangen, aber sein Zeitstempel größer ist als der des Absenders.
- Wenn sich der Prozess bereits in seinem *CS* befindet (besitzt den Lock) oder sein Zeitstempel kleiner als der des Absenders ist, puffert er alle Anforderungen bis zum Verlassen des *CS*.

MUTUALER AUSSCHLUSS

RA3: Ein Prozess tritt in seine *CS* ein, wenn er von jedem der verbleibenden (n-1) Prozesse eine Bestätigung erhält.

RA4: Nach dem Verlassen seiner *CS* muss ein Prozess eine Rückmeldung an jede der ausstehenden Anfragen senden, bevor er eine neue Anfrage macht oder andere Aktionen ausführt.

MUTUALER AUSSCHLUSS

Token Algorithmen

- Eine andere Klasse verteilter mutualer Ausschlussalgorithmen verwendet das Konzept eines expliziten variablen Tokens, das als eine Erlaubnis für den Eintritt in die CS dient (Mutex Acquire) und von einem anfordernden Prozess durch das Prozesssystem weitergereicht werden kann.
- Immer wenn ein Prozess in seinen CS eintreten möchte, muss er den Token erwerben. Der erste bekannte Algorithmus, der zu dieser Klasse gehört, ist auf Suzuki und Kasami zurückzuführen.
- Da es nur ein Token gibt ist der mutuale Ausschluss (Sicherheit) garantiert. Die Lebendigkeit aber nicht unbedingt (Verlust des Tokens).

MUTUALER AUSSCHLUSS

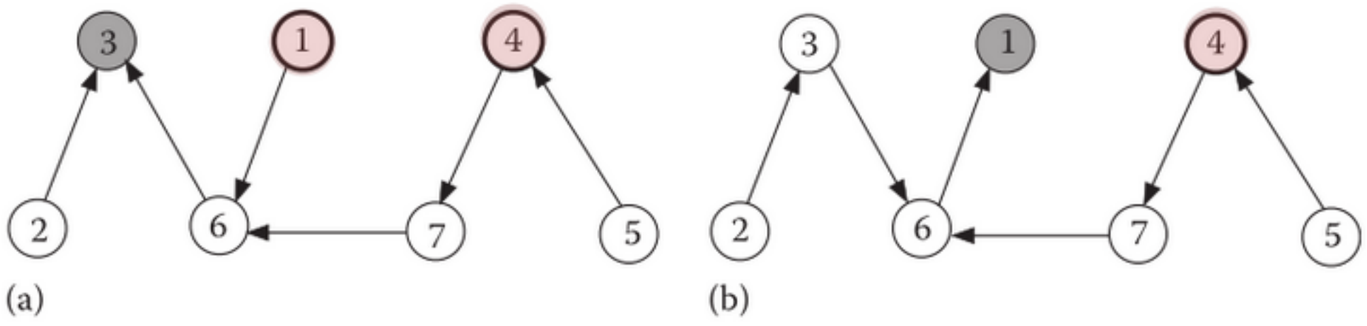


Abb. 78. Prozess 3 hält den Token und Prozesse 1 und 4 fordern ihn (über 6) an; schließlich erhält ihn 4

VERTEILTER KONSENS

- Ein verteilter Konsensalgorithmus hat das Ziel in einer Gruppe von Prozessen oder Agenten eine gemeinsame Entscheidung zu treffen
- Zentrale Eigenschaften:
  - Zustimmung/Übereinstimmung
  - Terminierung; Lebendigkeit und Deadlockfreiheit
  - Gültigkeit; Robustheit gegenüber Störungen wie fehlerhaften Nachrichten oder Ausfälle von Gruppenteilnehmern
- Beim Konsens kann ein Master-Slave Konzept oder ein Gruppenkonzept mit Leader/Commander und Workern verwendet werden.
  - Beim Master-Slave Konzept kommunizieren nur Slaves mit dem Master
  - Bei Gruppenkonzept (i.A. mit einem Leader) kommunizieren auch alle Gruppenteilnehmer untereinander
- Durch Störung (Fehler oder Absicht) kann es zu fehlerhaften bis hin zu fehlgeschlagenen Konsens kommen.

VERTEILTER KONSENS

- Bedingungen für Interaktive Konsistenz:
  - » IC1: Jeder Worker empfängt die *gleiche* Anweisung vom Leader!
  - » IC2: Wenn der Leader *fehlerfrei* arbeitet, dann empfängt jeder *fehlerfreie* Worker die Anweisung die der Leader sendete!

Byzantinisches Generalproblem

- Beispiel: In einer Gruppe aus drei Prozessen/Agenten ist einer fehlerhaft bzw. versendet fehlerhafte Nachrichten (durch Störung oder Absicht) mit Anweisungen (schließlich ein Konsensergebnis) [E]

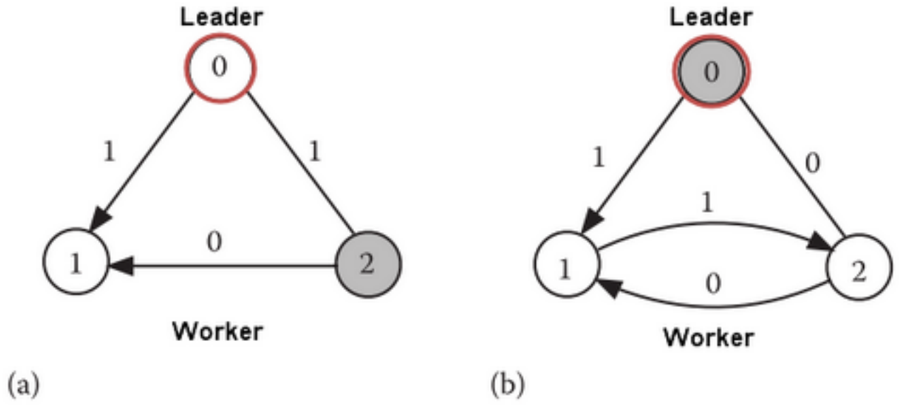


Abb. 79. Byzantinisches Generalproblem: (a) Leader 0 ist fehlerfrei, Worker 2 ist fehlerhaft (b) Leader 0 ist fehlerhaft, Worker 1 und 2 sind fehlerfrei [E]

VERTEILTER KONSENS

- Jeder Worker der Nachrichten empfängt ordnet diese nach direkten und indirekten (von Nachbarn)
- **Fall (a)**: Prozess 2 versendet fehlerhafte Nachricht mit Anweisung 0, Prozess 1 empfängt eine direkte Nachricht mit Anweisung 1 und eine indirekte mit (falschen) Inhalt Anweisung 0
  - » Bedingung IC1 ist erfüllt. Unter Annahme von Bedingung IC2 wird Worker 1 die direkte Anweisung 1 von Prozess 0 (Commander) auswählen → Konsens wurde gefunden
- **Fall (b)**: Prozess 0 (Leader) versendet an Prozess 1 richtige Nachricht mit Anweisung 1 und falsche Nachricht mit Anweisung 0 an Prozess 1
  - » Würde Prozess 1 wieder zur Erfüllung von IC2 eine Entscheidung treffen (Anweisung 1 auswählen), dann wäre IC1 verletzt. Wie auch immer Prozess 1 entscheidet ist entweder IC1 oder IC2 verletzt → **Unentscheidbarkeit** → Kein Konsens möglich

VERTEILTER KONSENS

Das nicht-signierte Nachrichtenmodell erfüllt die Bedingungen:

1. Nachrichten werden während der Übertragung nicht verändert (aber keine harte Bedingung).
2. Nachrichten können verloren gehen, aber die Abwesenheit von Nachrichten kann erkannt werden.
3. Wenn eine Nachricht empfangen wird (oder ihre Abwesenheit erkannt wird), kennt der Empfänger die Identität des Absenders (oder des vermeintlichen Absenders bei Verlust).

- Algorithmen zur Lösung des Konsensproblems müssen m fehlerhafte Prozesse annehmen (bzw. fehlerhafte Nachrichten)

Der OM(m) Algorithmus

- Ein Algorithmus der einen Konsens erreicht bei Erfüllung der Bedingungen IC1 und IC2 mit bis zu m fehlerhaften Prozesse bei insgesamt $n \geq 3m+1$ Prozessen mit nicht signierten ("mündlichen") Nachrichten.

  i. Leader i sendet einen Wert $v \in \{0, 1\}$ an jeden Worker $j \neq i$.
  ii. Jeder Worker j akzeptiert den Wert von i als Befehl vom Leader i.

VERTEILTER KONSENS

Definition 17.
Algorithmus OM(m)

1. Leader i sendet einen Wert $v \in \{0, 1\}$ an jeden Worker $j \neq$ i.
2. Wenn $m > 0$, dann beginnt jeder Worker j, der einen Wert vom Leader erhält, eine neue Phase, indem er ihn mit OM(m-1) an die verbleibenden Worker sendet.
   - » In dieser Phase fungiert j als Leader.
   - » Jeder Arbeiter erhält somit (n-1) Werte: (a) einen Wert, der direkt von dem Leader i von OM (m) empfangen wird und (b) (n-2) Werte, die indirekt von den (n-2) Workern erhalten werden, die aus ihrem Broadcast OM(m-1) resultieren.
   - » Wird ein Wert nicht empfangen wird er durch einen Standardwert ersetzt.
3. Jeder Worker wählt die Mehrheit der (n-1) Werte, die er erhält, als Anweisung vom Leader i.

VERTEILTER KONSENS

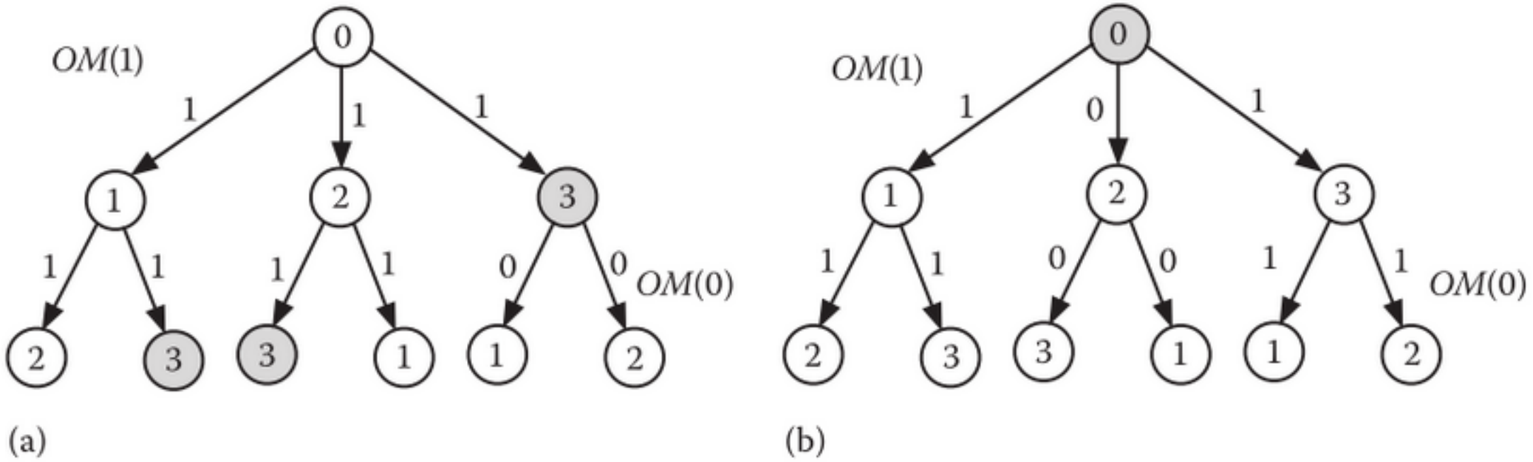


Abb. 80. Eine Illustration von OM(1) mit vier Prozessen und einem fehlerhaften Prozess: die Nachrichten auf der oberen Ebene spiegeln die Eröffnungsnachrichten von OM(1) wider und die auf der unteren Ebene spiegeln die OM(0)-Meldungen wider, die von den Mitteilungen der oberen Ebene ausgelöst werden. (a) Prozess 3 ist fehlerhaft. (b) Prozess 0 (Leader) ist fehlerhaft. [E]

VERTEILTER KONSENS

Der Paxos Algorithmus

- Paxos ist ein Algorithmus zur Implementierung von fehlertoleranten Konsensfindungen.
- Er läuft auf einem *vollständig verbundenen Netzwerk* von n Prozessen und toleriert bis zu m Ausfälle, wobei $n \geq 2m+1$ ist.
- Prozesse können abstürzen und Nachrichten können verloren gehen, byzantinische Ausfälle (absichtliche Verfälschung) sind jedoch zumindest in der aktuellen Version ausgeschlossen.
- Der Algorithmus löst das Konsensproblem bei Vorhandensein dieser Fehler auf einem *asynchronen System von Prozessen*.

» Obwohl die Konsensbedingungen Zustimmung, Gültigkeit und Terminierung sind, garantiert Paxos in erster Linie die Übereinstimmung und Gültigkeit und nicht die Beendigung - es ermöglicht die Möglichkeit der Beendigung nur dann, wenn es ein ausreichend langes Intervall gibt, in dem kein Prozess das Protokoll neu startet.

VERTEILTER KONSENS

- Ein Prozess kann drei verschiedene Rollen wahrnehmen:
  - » Antragsteller,
  - » Akzeptor und
  - » Lerner.

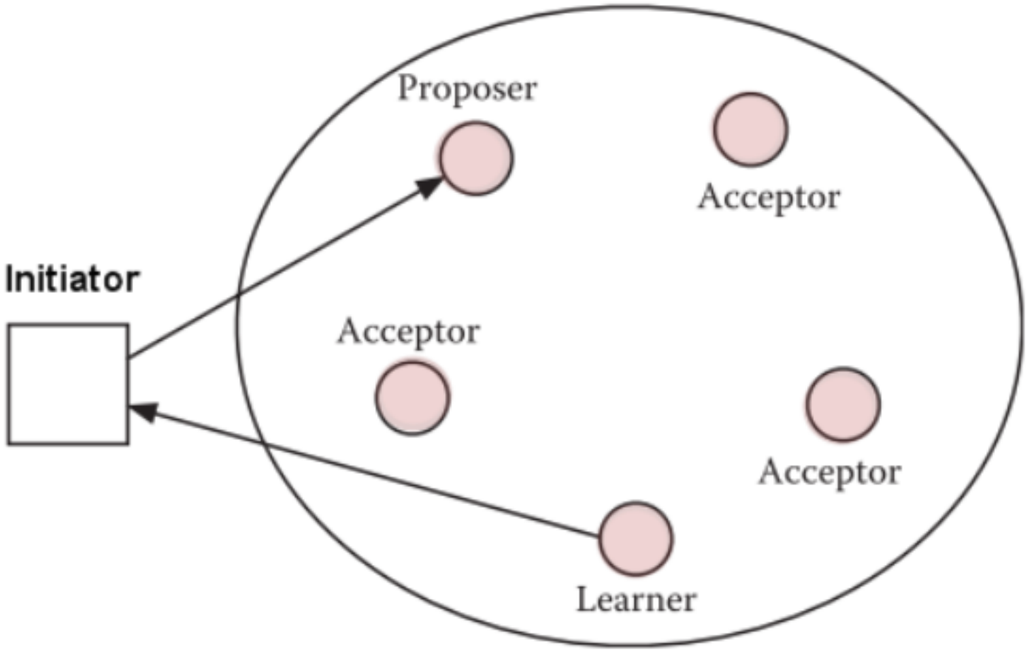


Abb. 81. Typische Rollenverteilung beim Paxos Algorithmus

VERTEILTER KONSENS

- Die **Antragsteller** reichen die vorgeschlagenen Werte im Namen eines Initiators ein,
- die **Akzeptoren** entscheiden über die Kandidatenwerte für die endgültige Entscheidung und
- die **Lernenden** sammeln diese Informationen von den Akzeptoren und melden die endgültige Entscheidung dem Initiator zurück.

- Ein Vorschlag, der von einem Antragsteller gesendet wird, ist ein Tupel (v, n), wobei v ein Wert und n eine Sequenznummer ist.
- Wenn es nur einen Akzeptor gibt, der entscheidet, welcher Wert als Konsenswert gewählt wird, dann wäre diese Situation zu einfach. Was passiert, wenn der Akzeptor abstürzt? Um damit umzugehen, gibt es mehrere Akzeptoren.
- Ein Vorschlag muss von mindestens einem Akzeptor bestätigt werden, bevor er für die endgültige Entscheidung in Frage kommt.

VERTEILTER KONSENS

- Die Sequenznummer wird verwendet, um zwischen aufeinander folgenden Versuchen der Protokollanwendung zu unterscheiden.
- Nach Empfang eines Vorschlags mit einer größeren Sequenznummer von einem gegebenen Prozess, verwerfen Akzeptoren die Vorschläge mit niedrigeren Sequenznummern.
- Schließlich akzeptiert ein Akzeptor die Entscheidung der Mehrheit.

Phasen des Paxos Algorithmus

1. **Die Vorbereitungsphase**
   - » Jeder Antragsteller sendet einen Vorschlag (v, n) an jeden Akzeptor
   - » Wenn n die größte Sequenznummer eines von einem Akzeptor empfangenen Vorschlags ist, dann sendet er ein *ack* $(n, \perp, \perp)$ an seinen Vorschlager
   - » Hat der Akzeptor einen Vorschlag mit einer Sequenznummer $n' < n$ und einem vorgeschlagenen Wert v akzeptiert, antwortet er mit $ack(n, v, n')$.

VERTEILTER KONSENS

2. **Aufforderung zur Annahme eines Eingabewertes**
   - Wenn ein Antragsteller $ack(n, \perp, \perp)$ von einer Mehrheit von Akzeptoren empfängt, sendet er an alle Akzeptoren $accept(v, n)$ und fordert sie auf, diesen Wert zu akzeptieren.
   - Wenn ein Akzeptor in Phase 1 einen $ack(n, v, n')$ an den Antragsteller zurücksendet, muss der Antragsteller den Wert v mit der höchsten Sequenznummer in seiner Anfrage an die Akzeptoren einbeziehen.
   - Ein Akzeptor akzeptiert einen Vorschlag (v, n), sofern er nicht bereits zugesagt hat, Vorschläge mit einer Sequenznummer größer als n zu berücksichtigen.
3. **Die endgültige Entscheidung**
   - Wenn eine Mehrheit der Akzeptoren einen vorgeschlagenen Wert akzeptiert, wird dies der endgültige Entscheidungswert. Die Akzeptoren senden den akzeptierten Wert an die Lernenden, wodurch sie feststellen können, ob ein Vorschlag von einer Mehrheit von Akzeptoren akzeptiert wurde.

REFERENZEN

BÜCHER

1. P. S. Pacheco, An introduction to parallel programming. Elsevier, MK, 2011.
2. T. Bräunl, Parallel Image Processing. Springer Berlin, 2001.
3. T. Rauber and G. Rünger, Parallele und verteilte Programmierung. Springer Berlin, 2007.
4. G. R. Andrews, Foundations of Multithreaded, Parallel, and Distributed Programming. Addison-Wesley, Pearson, 2000.
5. K. Hwang, G. C. Fox, and J. J. Dongarra, Distributed and Cloud Computing: From Parallel Processing to the Internet of Things. Elsevier B.V., 2012.
6. S. Ghosh, Distributed Systems - An Algorithmic Approach. Chapman & Hall/CRC, 2015.
7. D. Talia, P. Trunfio, and F. Marozzo, Data Analysis in the Cloud. Elsevier Ltd, 2016.

VIDEOS

A. Dining Philosophers, https://www.youtube.com/watch?v=fg-NiDlcoS8