Parallele Algorithmen

Vorlesung gehalten im SS '98

Oliver Vornberger

Frank M. Thiesing

Fachbereich Mathematik/Informatik
Universität Osnabrück

Literatur

Vipin Kumar, Ananth Grama, Anshul Gupta, George Karypis:
'Introduction to Parallel Computing — Design and Analysis of Algorithms'
The Benjamin/Cummings Publishing Company, Inc. 1994

Michael J. Quinn:
'Parallel Computing — Theory and Practice (Second Edition)'
McGraw-Hill International Editions 1994

Danksagung

Wir danken ...

... Frau Gerda Holmann für sorgfältiges Erfassen des Textes und Erstellen der Grafiken,

... Herrn Frank Lohmeyer und Herrn Volker Schnecke für ihre engagierte Mitarbeit bei der inhaltlichen und äußerlichen Gestaltung des Textes,

... Herrn Axel Hädicke und Herrn Curd Reinert für sorgfältiges Korrekturlesen.

... Herrn Viktor Herzog für die Konvertierung des Skripts nach HTML.

HTML-Version
Der Inhalt dieser Vorlesung kann online betrachtet werden unter
`http://www-lehre.informatik.uni-osnabrueck.de/~pa`

Osnabrück, im Juli 1998

(Oliver Vornberger) (Frank M. Thiesing)

Inhaltsverzeichnis

1 Einführung **1**
- 1.1 Grand Challenges . . . 1
- 1.2 Historische Entwicklung . . . 2
- 1.3 Begriffsabgrenzungen . . . 4
- 1.4 Argumente gegen Parallelismus . . . 5
- 1.5 Definitionen . . . 7

2 Maschinenmodelle **9**
- 2.1 Kontrollmechanismus . . . 9
- 2.2 Speicherorganisation . . . 12
- 2.3 Verbindungsstruktur . . . 13
- 2.4 Granularität . . . 13
- 2.5 PRAM . . . 13

3 Topologien **17**
- 3.1 Dynamische Verbindungsnetzwerke . . . 17
  - 3.1.1 Crossbar Switching Netzwerk . . . 17
  - 3.1.2 Bus-basierte Verbindung . . . 18
  - 3.1.3 Multistage Verbindungsnetzwerk . . . 19
  - 3.1.4 Omega-Netzwerk . . . 20
- 3.2 Statische Verbindungsnetzwerke . . . 22
  - 3.2.1 Clique . . . 23
  - 3.2.2 Stern . . . 23
  - 3.2.3 Binärer Baum . . . 24
  - 3.2.4 Lineares Array/Ring . . . 25
  - 3.2.5 2D-Gitter . . . 26
  - 3.2.6 3D-Gitter . . . 27
  - 3.2.7 Hypercube . . . 28
  - 3.2.8 Butterfly . . . 30
  - 3.2.9 Cube Connected Cycles . . . 31
  - 3.2.10 Shuffle Exchange . . . 32
  - 3.2.11 de Bruijn . . . 33
- 3.3 Netzwerkeinbettungen . . . 35
  - 3.3.1 Ring in Hypercube . . . 35
  - 3.3.2 Gitter in Hypercube . . . 35
  - 3.3.3 Binärer Baum im Hypercube . . . 36

4 Basiskommunikation **39**
4.1 Kosten . . . 39
4.2 One-to-All Broadcast . . . 42
4.3 All-to-All Broadcast . . . 47
4.4 Echo-Algorithmus . . . 50
4.5 Terminierung . . . 51

5 Performance **53**

6 Matrix-Algorithmen **57**
6.1 Partitionierung . . . 57
6.2 Matrix-Transposition in Gitter und Hypercube . . . 58
6.3 Matrix-Vektor-Multiplikation im Ring . . . 61
6.4 Matrizenmultiplikation im Gitter . . . 62
6.5 Matrizenmultiplikation im Hypercube . . . 65

7 Lineare Gleichungssysteme **67**
7.1 Gauß-Jordan-Elimination auf PRAM . . . 68
7.2 Gauß-Elimination im Gitter . . . 69
7.3 Cholesky-Zerlegung im Ring . . . 72
7.4 Iterationsverfahren . . . 77

8 Sortierverfahren **81**
8.1 PRAM Sort . . . 81
8.2 Odd-Even-Transposition Sort . . . 82
8.3 Sortiernetzwerke . . . 84
8.4 Sortieren im Hypercube . . . 88
8.5 Sortieren im Shuffle-Exchange . . . 88
8.6 Quicksort im Hypercube . . . 90

9 Graphenalgorithmen **93**
9.1 Definitionen . . . 93
9.2 Implementation von Graphen . . . 95
9.3 Shortest Path . . . 96
9.4 All Shortest Paths . . . 99
9.5 Minimum Spanning Tree . . . 100
9.6 Zusammenhangskomponente . . . 102

10 Kombinatorische Optimierung **107**
10.1 Definitionen . . . 107
10.2 Sequentielles Suchen . . . 110
10.3 Paralleles Suchen . . . 114
10.4 Spielbaumsuche . . . 119
10.5 Dynamic Programming . . . 122

11 Programmiersprachen **125**

Kapitel 1

Einführung

Seit es Computer gibt, verlangen deren Benutzer nach mehr Rechenleistung. Begründet wird dieser Hunger mit speziellen Anwendungen, den sogenannten *Grand Challenges*, bei denen eine sehr große Zahl von Instruktionen in einer vorgegebenen Zeitspanne absolviert werden muß:

1.1 Grand Challenges

Simulation physikalischer Vorgänge

- Wettervorhersage,
- Strömungssimulation statt Windkanal,
- Steifigkeitsanalyse statt Crash-Test,
- Fahr- und Flugsimulatoren (Realzeit).

Künstliche Intelligenz

- Schrifterkennung mit OCR,
- Sprachverarbeitung,
- Bildverarbeitung,
- logische Inferenzen in Expertensystemen,
- Gewichte-Updates in Neuronalen Netzen.

Bioinformatik

- Human Genom Project,
- Proteinstrukturvorhersage.

Computergrafik

- Visualisierung,
- Virtual Reality.

Zum Beispiel soll eine Wettervorhersage für eine Fläche von 3000 × 3000 Meilen innerhalb einer Höhe von 11 Meilen berechnet werden. Dieser Raum sei in Würfel mit einer Kantenlänge von $\frac{1}{10}$ Meile partitioniert. Somit ergeben sich $(3000 \times 3000 \times 11)/0.1^3 \approx 10^{11} = 100$ Milliarden Würfel. Für eine 2-Tages-Simulation sei halbstündlich jeder Würfel mit etwa 100 Operationen upzudaten. Dies ergibt $10^{11} \cdot 96 \cdot 100 \approx 10^{15} = 1000$ Billionen Instruktionen. Auf einem Rechner mit einem Gigaflopsprozessor (10^9 Floating Point Operations per second) ergibt sich eine Gesamtzeit von 10^6 Sekunden ≈ 277 Stunden ≈ 11 Tage. Eine Verdoppelung der Auflösung in den drei räumlichen Dimensionen und in der zeitlichen Dimension verlängert die Rechenzeit um den Faktor 16 auf insgesamt 6 Monate.

1.2 Historische Entwicklung

Bild 1.1 zeigt, daß in den vergangenen Jahrzehnten eine beachtliche Leistungssteigerung möglich war: Etwa alle 5 Jahre verzehnfachte sich die Prozessorleistung.

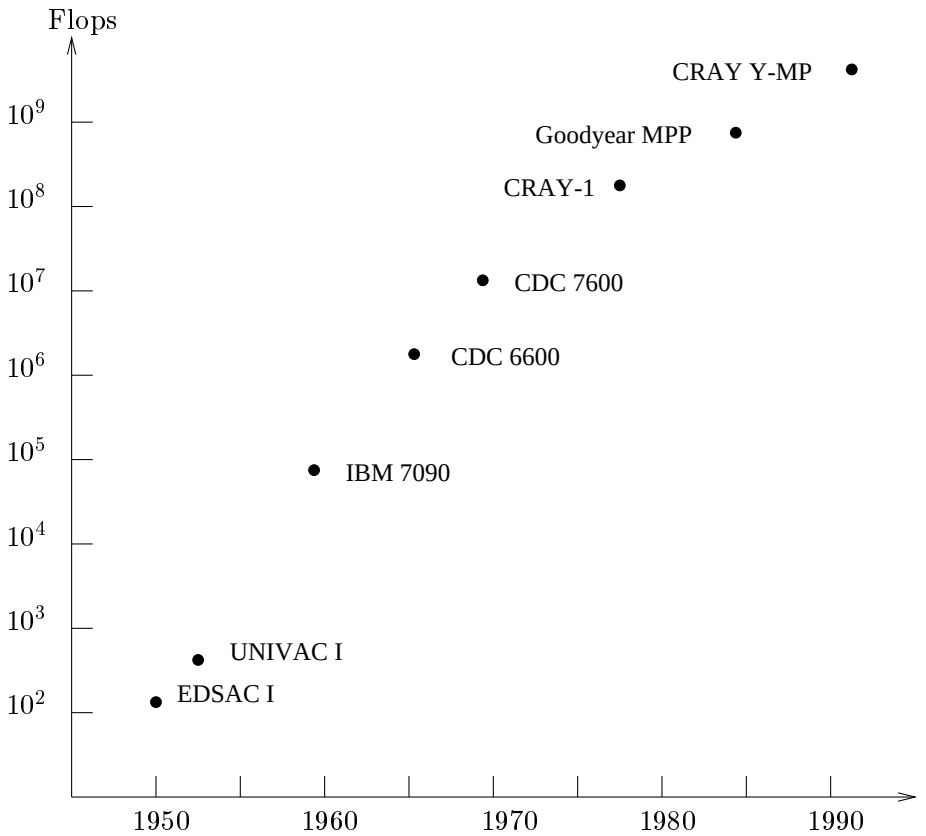


Abbildung 1.1: Entwicklung der Prozessorleistung

Ermöglicht wurde dieser Zuwachs durch eine Beschleunigung der Schaltlogik und durch Fortschritte in der Rechnerarchitektur:

- zunächst bit-seriell, dann bit-parallel,
- E/A-Kanäle entlasten CPU,
- verschränkter Speicher erlaubt gleichzeitige Zugriffe auf mehrere Speicherbänke,
- Speicherhierarchie nutzt zeitliche + räumliche Lokalität
  (Register – Cache – Primär – Sekundär),
- *Instruction look ahead* lädt Befehle auf Verdacht, da *fetch* langsamer als *decode*,
- multiple Funktionseinheiten für `INCR, ADD, MULT, SHIFT`
  (2 bis 3 gleichzeitig in Aktion),
- *Instruction Pipelining*
  instruction fetch – decode – operand fetch – execute
- Vektorprozessor für arithmetische Operationen auf Vektoren ($A = B + C$)

Ein Ende dieser Zuwachsraten ist abzusehen:

- *Pipelining* und Vektoroperationen haben einen beschränkten Parallelitätsgrad.
- Aufgrund von elektronischen Prinzipien läßt sich die Taktgeschwindigkeit eines Prozessors nicht mehr beliebig steigern.

Also liegt es nahe, mehrere Prozessoren zusammenzuschalten und sie gemeinsam an einem Problem arbeiten zu lassen. Dies erfordert eine Neuformulierung des verwendeten Lösungsverfahrens als parallelen Algorithmus!

1.3 Begriffsabgrenzungen

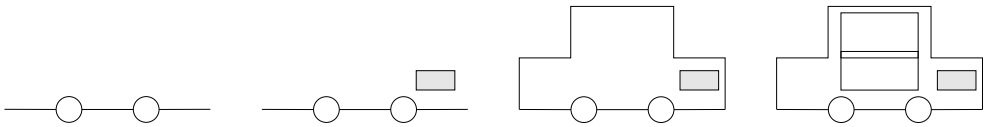
Multiprogramming: mehrere Prozesse teilen sich die CPU ereignisorientiert (I/O, Seitenfehler)

Timesharing: Multiprogramming mit Zeitscheiben

Pipelining: Rechnung besteht aus Phasen.
Ausgabe von Phase i ist Eingabe für Phase $i+1$. Prozessor i ist zuständig für Phase i. Nach Füllen der Pipeline wird an allen Phasen gleichzeitig gearbeitet. Beschleunigung beschränkt durch Anzahl der Phasen.

Parallel Processing: Rechnung erzeugt Arbeitspakete, die statisch oder dynamisch einer beliebig großen Prozessorzahl zugewiesen werden.

Beispiel: Automobilbau in 4 Phasen



Sequentiell: 1 Auto alle 4 Zeiteinheiten
Pipelining: 1. Auto nach 4 Zeiteinheiten,
dann 1 Auto pro Zeiteinheit
Parallel: 4 Autos alle 4 Zeiteinheiten auf 4 Bändern

# Autos	seq	pipe	par
1	4	4	4
2	8	5	4
3	12	6	4
4	16	7	4
5	20	8	8
6	24	9	8
7	28	10	8
8	32	11	8

Tabelle 1.1: Produktionszeiten bei sequentieller, pipelineorientierter und paralleler Arbeitsweise

1.4 Argumente gegen Parallelismus

Minsky's Conjecture (1971):

Speedup = $O(\log p)$ bei p Prozessoren

Antwort:

nur manchmal richtig, oft falsch.

Grosch's Law (1975):

Speed = $O(\text{cost}^2)$, d.h. doppelte Kosten = vierfache Leistung
$\Rightarrow$ 1 schneller Prozessor ist billiger als 2 langsame Prozessoren

Antwort:

nur richtig innerhalb einer Klasse (PC, Workstation, Mainframe).

Zwischen den Klassen gilt:

Speed = $O(\sqrt{\text{cost}})$, d.h. vierfache Kosten = doppelte Leistung
$\Rightarrow$ 2 langsame sind billiger als 1 schneller.

Geschichte:

Alle 5 Jahre wächst Leistung um Faktor 10. Also warten.

Antwort:

Parallelrechner profitieren auch davon.
Manche Probleme verlangen jetzt 100-fache Steigerung.

Architektur:

Vektorrechner reichen!

Antwort:

Viele Probleme basieren auf skalaren Daten (K.I.).

Amdahl's Law:

Sei $0 < f < 1$ der sequentielle Anteil eines Algorithmus.
Sei p die Anzahl der Prozessoren.
$\Rightarrow$ Speedup $\leq \frac{1}{f+(1-f)/p} < \frac{1}{f}$ (unabhängig von p)
Beispiel: $f = 0.1 \Rightarrow$ Speedup < 10

Antwort:

Viele Probleme haben nur konstanten sequentiellen Teil.

Fortran:

Wohin mit der vorhandenen Software?

Antwort:

Wegwerfen!

1.5 Definitionen

Sequentialzeit:	Dauer des besten sequentiellen Algorithmus
Parallelzeit:	Zeit zwischen Beginn des ersten und Ende des letzten Prozessors
Kosten:	Anzahl der Prozessoren · Zeit
Speedup:	$\frac{\text{Sequentialzeit}}{\text{Parallelzeit}}$
Effizienz:	$\frac{\text{Speedup}}{\text{Anzahl der Prozessoren}}$
Glaubenskampf:	Gibt es superlinearen Speedup?
	Nein! Denn dann könnte man das parallele Verfahren auf einem Prozessor in verkürzter Zeit simulieren. Aber: Eventuell reicht der Platz nicht!
	Ja! Denn im Einzelfall kann das Sequentialverfahren "Pech haben" und das Parallelverfahren "Glück haben".
	Aber: Im Mittel sollte sich das ausgleichen!

Ein paralleler Algorithmus heißt *kostenoptimal*, wenn seine Kosten von derselben Größenordnung sind wie die Kosten des schnellsten sequentiellen Algorithmus. D.h., das Prozessor-Zeit-Produkt ist bis auf einen konstanten Faktor gleich der sequentiellen Laufzeit.

Beispiel für superlinearen Speedup:

Gegeben sei ein $0-1$-String w, bestehend aus n Bits.
Problem: Befindet sich eine Null darunter?

Sequentieller Ansatz:
`Durchlaufe von vorne nach hinten`

Paralleler Ansatz mit 2 Prozessoren:
`Beginne gleichzeitig vorne und hinten`

w	Sequentialzeit	Parallelzeit	Speedup	
0000	1	1	1	
0001	1	1	1	
0010	1	1	1	
0011	1	1	1	
0100	1	1	1	
0101	1	1	1	
0110	1	1	1	
0111	1	1	1	
1000	2	1	2	
1001	2	2	1	
1010	2	1	2	
1011	2	2	1	
1100	3	1	3	← Superlinear
1101	3	2	1.5	
1110	4	1	4	← Superlinear
1111	4	2	2	
Gesamt	30	20	1.5	

Tabelle 1.2: Laufzeiten und Speedup für Suche nach einem Null-Bit

Also beträgt bei gleichverteilten Strings der Länge 4 der durchschnittliche Speedup 1.5.

Kapitel 2

Maschinenmodelle

Parallelrechner haben mehrere Prozessoren und unterscheiden sich in

- Kontrollmechanismus
- Speicherorganisation
- Verbindungsstruktur
- Granularität

2.1 Kontrollmechanismus

SISD *single instruction, single data*
von Neumann-Rechner
RAM *Random Access Machine*

SIMD *single instruction, multiple data*
ein Programm, jeder Befehl bezieht sich auf mehrere Daten gleichzeitig, synchrone Arbeitsweise
oft: Spezialprozessoren, variable Anzahl, pro Datum ein Prozessor

MIMD *multiple instruction, multiple data*
mehrere Programme (ggf. identisch, aber als Prozeß verschieden) bearbeiten ihre Daten.
asynchrone Arbeitsweise
meistens: Universalprozessoren, konstante Zahl, pro Teilaufgabe ein Prozessor.

PE: Processing Element

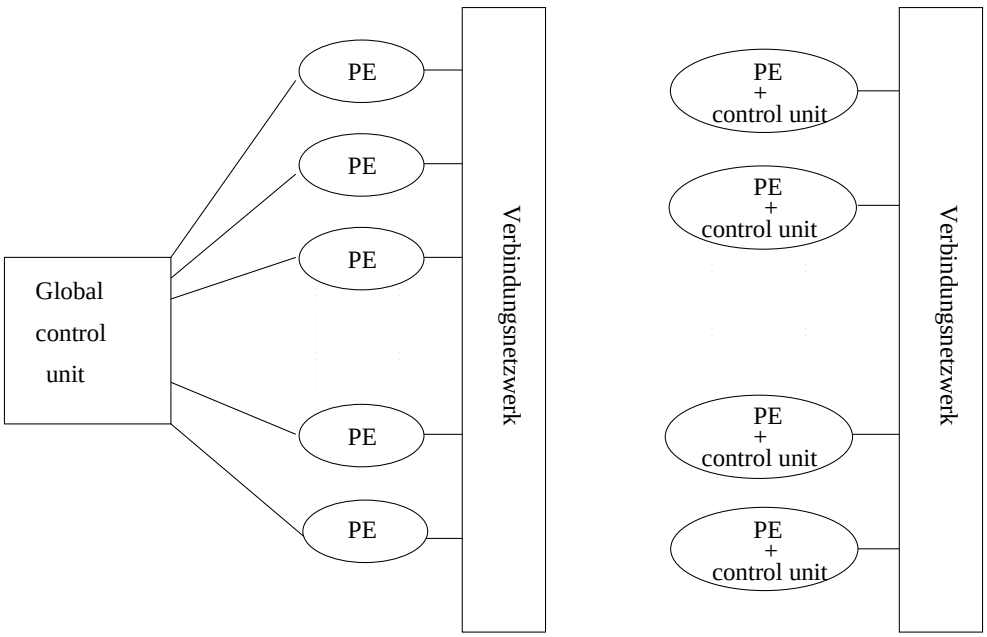


Abbildung 2.1: SIMD (links) versus MIMD (rechts)

SIMD-Rechner speichern den Programmcode nur einmal ab. Vorteil: Speicherersparnis. Nachteil: Alle Prozessoren bearbeiten jeweils denselben Befehl. Bei bedingten Anweisungen entstehen dadurch Leerzeiten (siehe Bild 2.2). Manche MIMD-Rechner (z.B. CM-5 von Thinking Machines Corporation) verfügen über spezielle Synchronisationshardware und können daher auch im SIMD-Modus arbeiten.

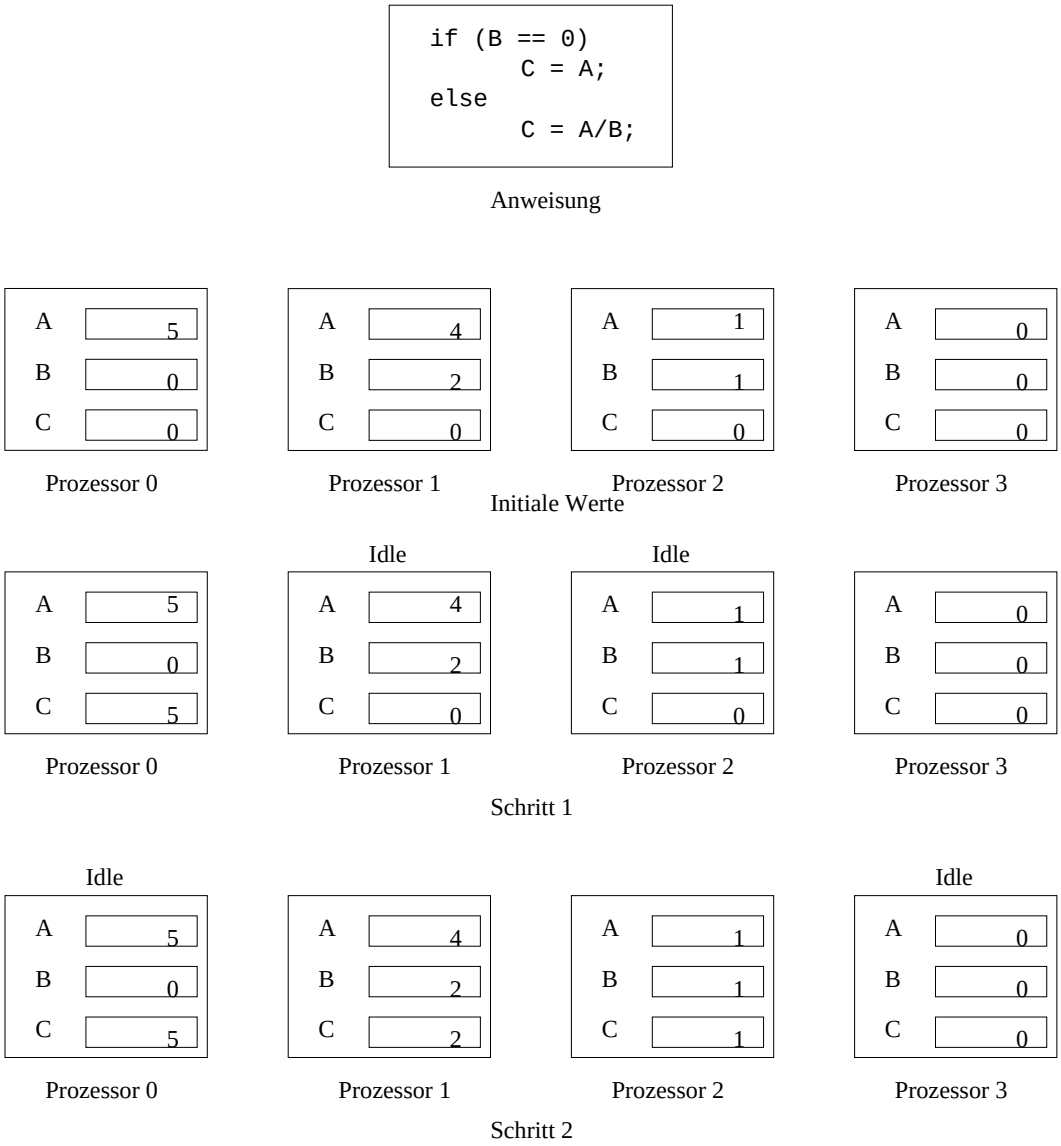


Abbildung 2.2: Abarbeitung einer bedingten Anweisung in einem SIMD-Rechner mit 4 Prozessoren. Nur jeweils die Hälfte der Prozessoren ist aktiv.

2.2 Speicherorganisation

Shared memory	alle Prozessoren operieren auf demselben Speicher, erreichbar über ein Verbindungsnetzwerk. Zugriffe sind entweder alle gleich schnell (*uniform*) oder zeitlich gestaffelt (*non uniform*).
Distributed memory = message passing	jeder Prozessor benutzt seinen lokalen Speicher und verschickt Nachrichten an andere Prozessoren über ein Verbindungsnetzwerk.

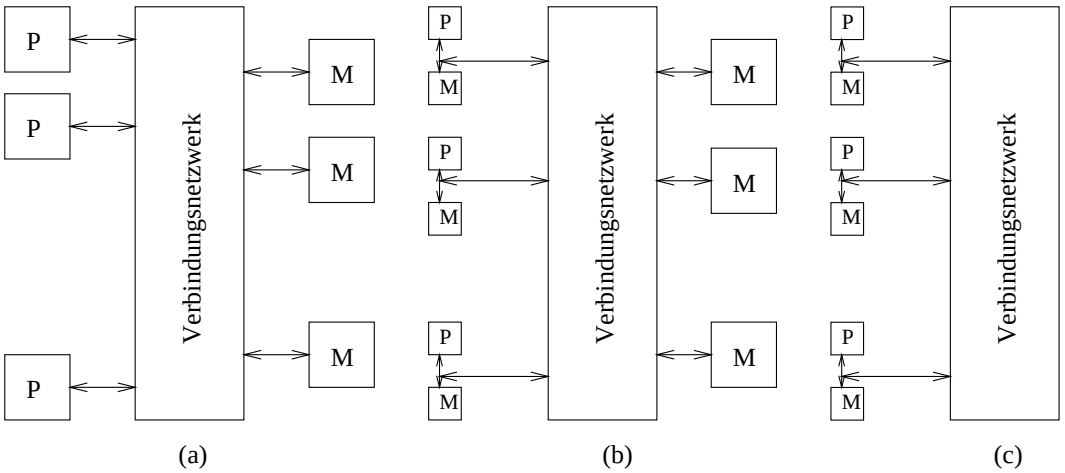


Abbildung 2.3: Shared-Memory Architekturen (P = Prozessor, M = Memory)
a) Uniform b) Non uniform mit lokalem/globalem Speicher
c) Non uniform mit lokalem Speicher

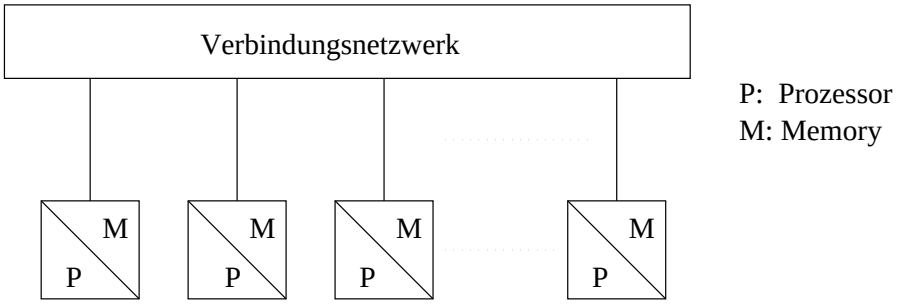


Abbildung 2.4: Distributed Memory Architektur

2.3 Verbindungsstruktur

Shared-Memory-Maschinen und Message-Passing-Systeme benötigen Verbindungsnetzwerke. Verbindungsnetzwerke sind entweder *statisch*, realisiert durch Punkt-zu-Punkt-Verbindungen zwischen den Prozessoren eines Message-Passing-Systems oder *dynamisch*, realisiert durch *Crossbar Switches* oder Busverbindungen zwischen den Prozessoren und ihren Speicherbänken in einem Shared-Memory-Rechner.

2.4 Granularität

Parallelrechner können sein

- grobkörnig: Dutzende von Hochleistungsprozessoren
  z.B. CRAY Y-MP hat 16 Gigaflops-Prozessoren
- mittelkörnig: Hunderte von schnellen Prozessoren
  z.B. GC/PP hat 256 Megaflops-Prozessoren (Power PC)
- feinkörnig: Tausende von langsamen Prozessoren
  z.B. CM-2 hat 65536 1-Bit-Prozessoren.

2.5 PRAM

Einen SIMD-Rechner mit variabler Prozessorzahl und shared memory bezeichnet man als PRAM (*Parallel Random Access Machine*). Man unterscheidet vier Varianten bzgl. der Gleichzeitigkeit von Lese- und Schreiboperationen:

EREW:	exclusive read, exclusive write
CREW:	concurrent read, exclusive write
ERCW:	exclusive read, concurrent write
CRCW:	concurrent read, concurrent write

Bei gleichzeitigem Schreiben muß die Semantik festgelegt werden,

z.B. Prozessor mit größter ID setzt sich durch.
z.B. ein zufällig gewählter Prozessor setzt sich durch.
z.B. nur erlaubt, wenn alle dasselbe schreiben.

Beispiel:

Gegeben: `VAR a: ARRAY[0..n-1] OF INTEGER;`
Gesucht: `antwort := Maximum der n Zahlen`

Zur Vereinfachung sei angenommen, daß alle Zahlen verschieden sind. Offenbar beträgt die Sequentialzeit $O(n)$.

EREW PRAM zur Maximumsuche auf n Zahlen
Verwendet werden $n/2$ Prozessoren $P_0, P_1, \ldots, P_{n/2-1}$

```
d := n;
REPEAT
   d := d DIV 2;
   FOR ALL 0 ≤ i ≤ d - 1 DO IN PARALLEL
      P_i : a[i] := maximum( a[2 * i], a[2 * i + 1] )
   END
UNTIL d = 1;
antwort := a[0];
```

Bemerkung: Statt des Maximums kann mit dieser Methode auch die Summe gebildet werden.

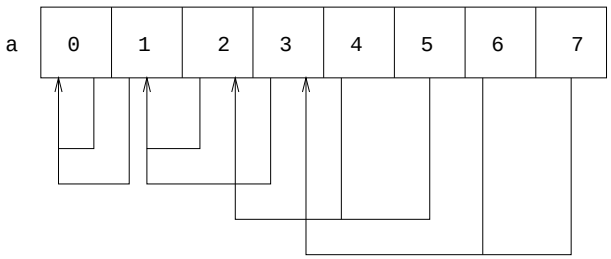


Abbildung 2.5: Zugriffspfade im ersten Schleifendurchlauf

Parallelzeit:	$O(\log n)$
Kosten:	$O(n \cdot \log n)$
Speedup:	$O(n/\log n)$
Effizienz:	$O(n/(n \cdot \log n)) = O(1/\log n)$

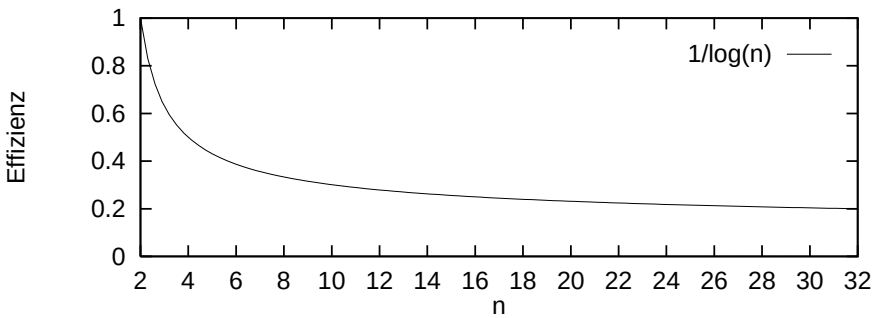


Abbildung 2.6: Effizienz (asymptotisch) bei Maximumsuche mit EREW PRAM

CRCW PRAM zur Maximumsuche auf n Zahlen
Verwendet werden n^2 Prozessoren $P_{00}, P_{01}, P_{02}, \ldots, P_{n-1n-1}$.
Beim gleichzeitigen Schreiben sei nur ein einziger Wert erlaubt!

```
VAR sieger : ARRAY [0..n-1] OF BOOLEAN;

FOR ALL 0 ≤ i ≤ n - 1 DO IN PARALLEL
   P_0i : sieger[i] := TRUE
END;

FOR ALL 0 ≤ i, j ≤ n - 1 DO IN PARALLEL
   P_ij : IF a[i] < a [j] THEN sieger[i] := FALSE END
END;

FOR ALL 0 ≤ i ≤ n - 1 DO IN PARALLEL
   P_0i: IF sieger[i] THEN antwort := a[i] END
END;
```

Parallelzeit:	$O(1)$
Kosten:	$O(n^2)$
Speedup:	$O(n)$
Effizienz:	$O(n/n^2) = O(1/n)$

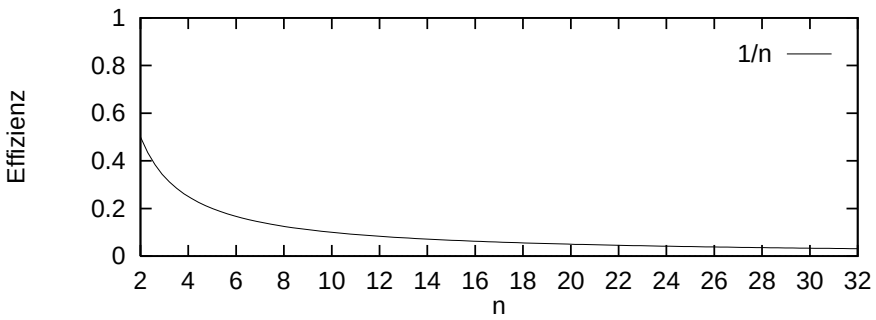


Abbildung 2.7: Effizienz (asymptotisch) bei Maximumsuche mit CRCW PRAM

CREW PRAM zur Matrizenmultiplikation
Verwendet werden n^3 Prozessoren $P_{000}, P_{001}, \ldots, P_{n-1,n-1,n-1}$.

Gegeben: zwei $n \times n$-Matrizen a, b.
Gesucht: ihr Matrizenprodukt c mit

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj}$$

```
VAR a,b : ARRAY [0..n-1] [0..n-1] OF REAL;


FOR ALL 0 ≤ i, j, k ≤ n - 1 DO IN PARALLEL
   P_ijk : tmp[i, j, k] := a[i, k] * b[k, j]
END
(* nun wird mit n^3/2 Prozessoren *)
(* das Array tmp [i, j, *] aufaddiert *)

d := n;
REPEAT
   d := d DIV 2;
   FOR ALL 0 ≤ k ≤ d - 1 DO IN PARALLEL
   P_ijk : tmp[i, j, k] := tmp[i, j, 2 * k] + tmp[i, j, 2 * k + 1]
   END
UNTIL d = 1;
```

Das Ergebnis c_{ij} befindet sich in `tmp[i, j, 0]`.

Sequentialzeit:	$O(n^3)$
Parallelzeit:	$O(\log n)$
Speedup:	$O(n^3/\log n)$
Effizienz:	$O(n^3/n^3 \cdot \log n) = O(1/\log n)$

Kapitel 3

Topologien

3.1 Dynamische Verbindungsnetzwerke

Die Prozessoren eines Shared-memory-Rechners referieren ihren globalen Speicher mit Hilfe von Verbindungsnetzwerken.

3.1.1 Crossbar Switching Netzwerk

Um p Prozessoren mit b Speicherbänken zu verbinden, wird ein *Crossbar Switch* mit $p \cdot b$ Schaltelementen benötigt.

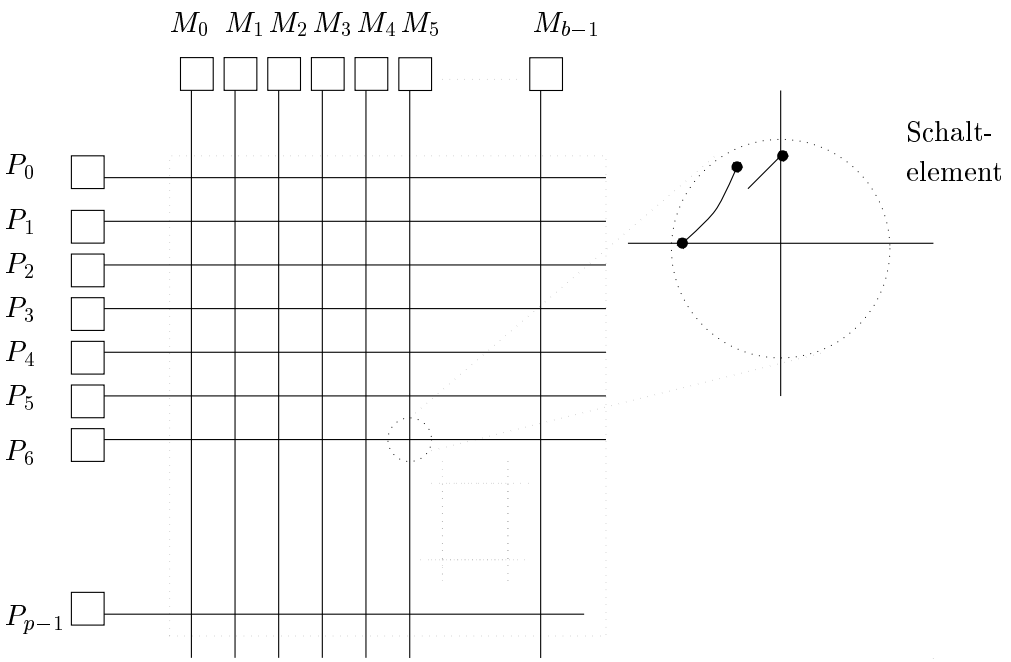


Abbildung 3.1: Crossbar Switch

Da sinnvollerweise $b \geq p$ gilt, wächst die Komplexität des Netzwerkes mit $O(p^2)$.

3.1.2 Bus-basierte Verbindung

Alle Prozessoren benutzen zum Speicherzugriff einen gemeinsamen Datenweg, genannt *Bus*.

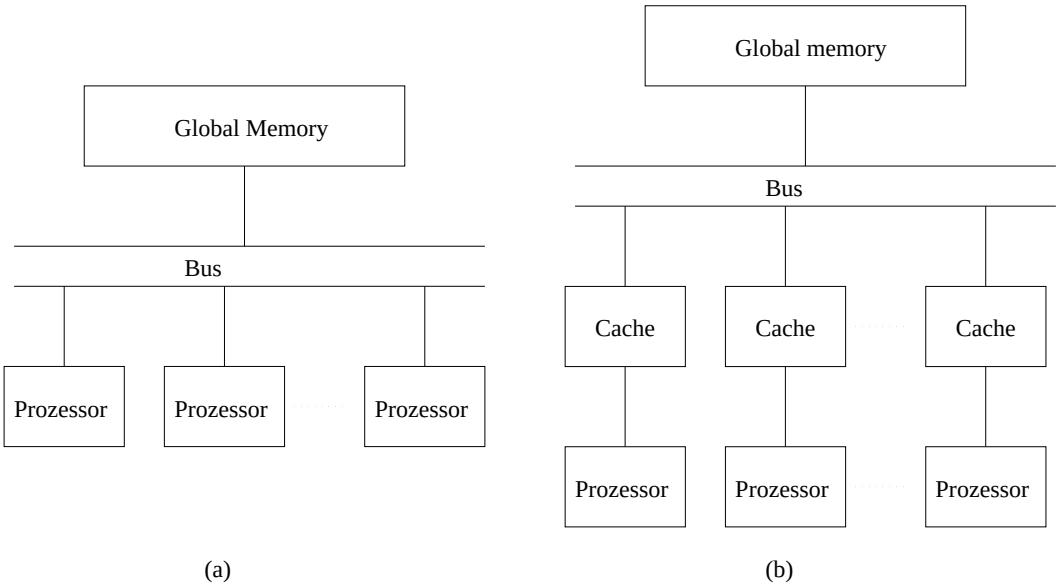


Abbildung 3.2: Bus-basierte Architektur ohne (a) und mit (b) Cache.

Der Bus kann allerdings zu einem Zeitpunkt nur eine begrenzte Menge von Daten zwischen Speicher und Prozessor transportieren, und somit steigt bei wachsender Prozessorzahl die Wartezeit für einen erfolgreichen Speicherzugriff. Daher spendiert man häufig jedem Prozessor einen lokalen Cache. Allerdings entsteht dadurch das Cache-Kohärenzproblem, da bei einem lokalen Update die existierenden Kopien berücksichtigt werden müssen.

3.1.3 Multistage Verbindungsnetzwerk

Crossbar-Switching-Netzwerke skalieren bzgl. der Leistung, aber nicht bzgl. der Kosten. Bus-basierte Netzwerke skalieren bzgl. Kosten, aber nicht bzgl. der Leistung. *Multistage*-Verbindungsnetzwerke liegen zwischen diesen Extremen.

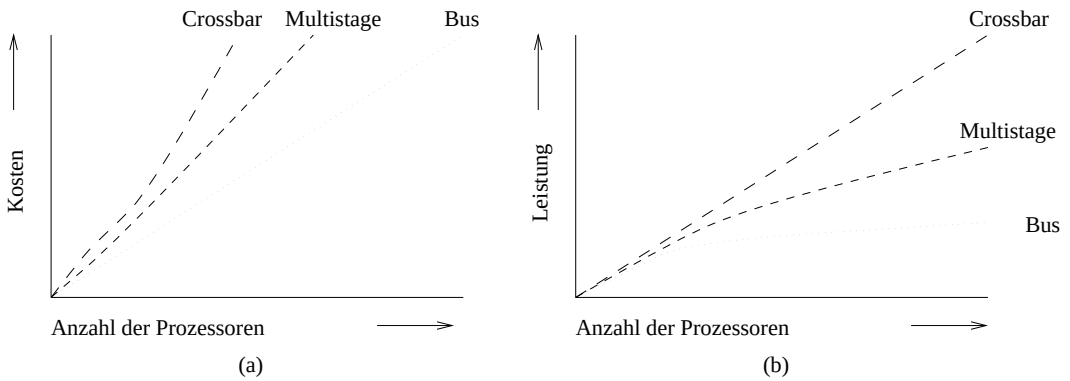


Abbildung 3.3: Skalierung von Kosten und Leistung bei Crossbar, Bus und Multistage.

Erreicht wird der Kompromiß durch einen mehrstufigen Aufbau

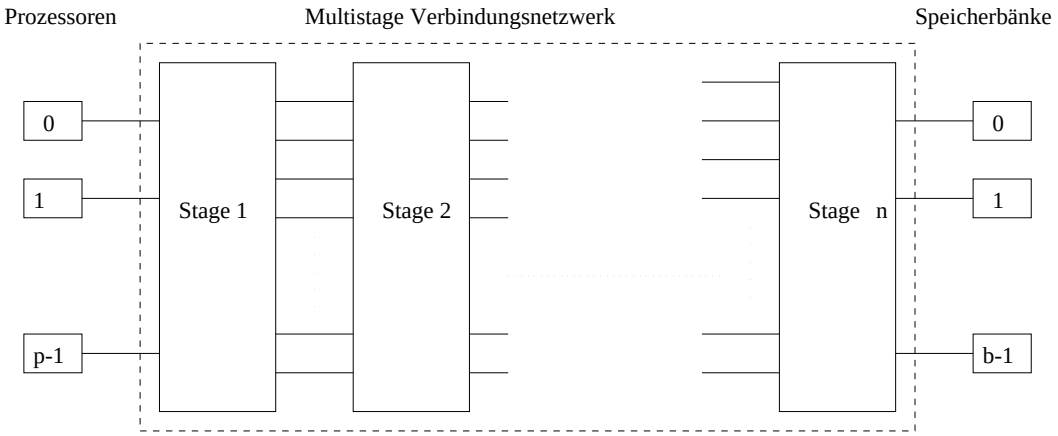


Abbildung 3.4: Schematischer Aufbau eines Multistage-Verbindungsnetzwerks zwischen p Prozessoren und b Speicherbänken.

3.1.4 Omega-Netzwerk

Eine weitverbreitete Multistage-Verbindungsstruktur ist das *Omega-Netzwerk*. Zwischen den $p = 2^k$ Prozessoren und den p Speicherbänken befinden sich $\log p$ Stufen mit jeweils $p/2$ Schaltelementen. Daher wachsen die Kosten mit $O(p \cdot \log p)$.
Jede Stufe verbindet ihren i-ten Input mit ihrem j-ten Output nach der Formel

$$j = \begin{cases} 2 \cdot i & \text{für } 0 \leq i \leq p/2 - 1 \\ 2 \cdot i + 1 - p & \text{für } p/2 \leq i \leq p - 1 \end{cases}$$

Diese Verbindung heißt *Perfect Shuffle* und bedeutet eine Linksrotation auf dem Binärmuster von i. Ihr Name rührt von der Beobachtung, daß alle n Zahlen wie beim Kartenmischen verschränkt werden.

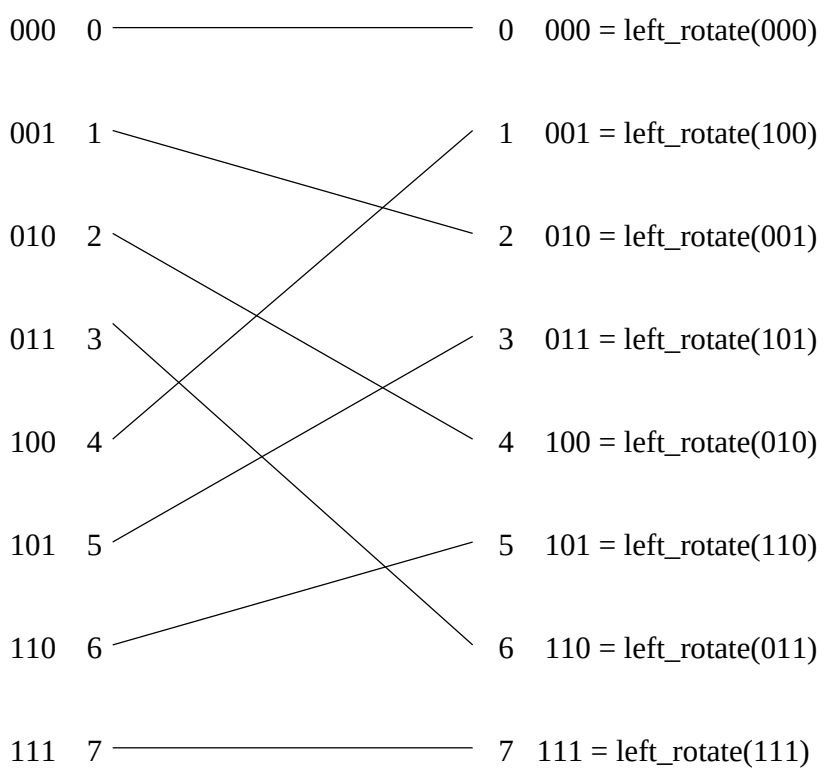


Abbildung 3.5: Perfect Shuffle zwischen 8 Inputs und 8 Outputs

Die Outputs einer Stufe werden paarweise in Schaltelemente geführt, welche ihre Eingänge entweder durchrouten oder vertauschen.

Ein Weg vom Startpattern s zum Zielpattern t entsteht durch systematisches Zusammensetzen der Zieladresse, wobei durch eine *Shuffle-Kante* das bereits erreichte Bitmuster zyklisch um ein Bit nach links geshiftet wird und durch das darauffolgende Schaltelement das letzte Bit ggf. invertiert werden kann.

Abbildung 3.6: Zustände eines Schaltelements: (a) Pass-Through (b) Cross-Over

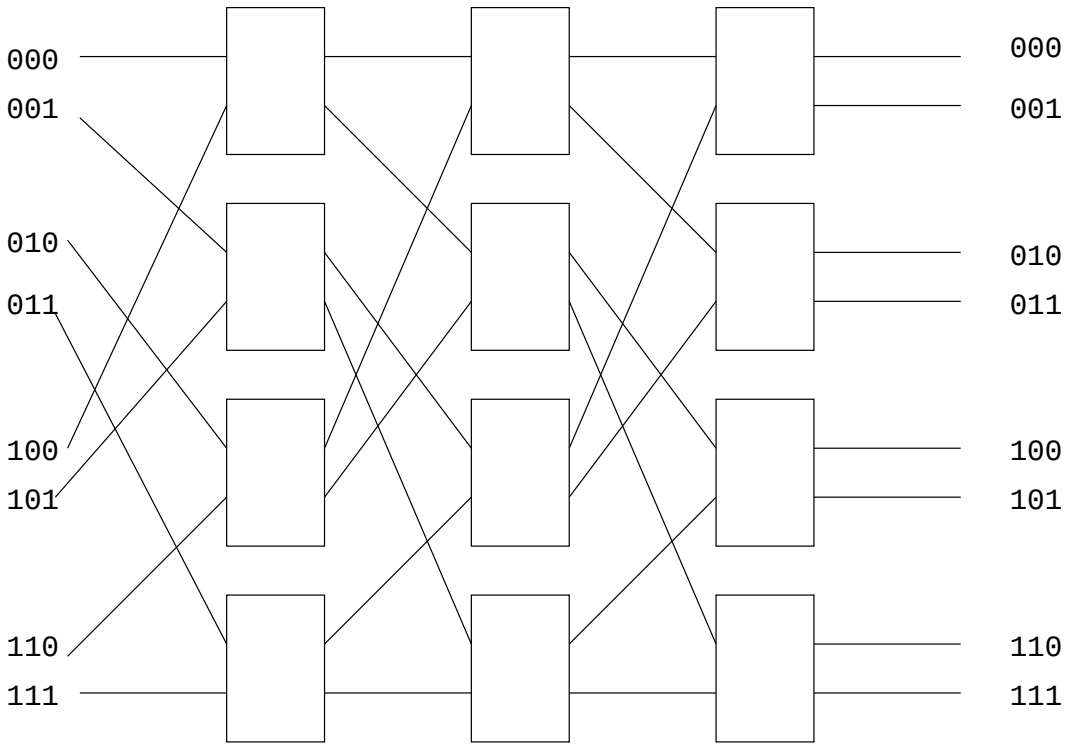


Abbildung 3.7: Vollständiges Omega-Netzwerk zwischen 8 Inputs und 8 Outputs

Omega-Netzwerke gehören zu den blockierenden Netzwerken, da zwei Kommunikationsströme ggf. über denselben Link laufen (siehe Bild 3.8).

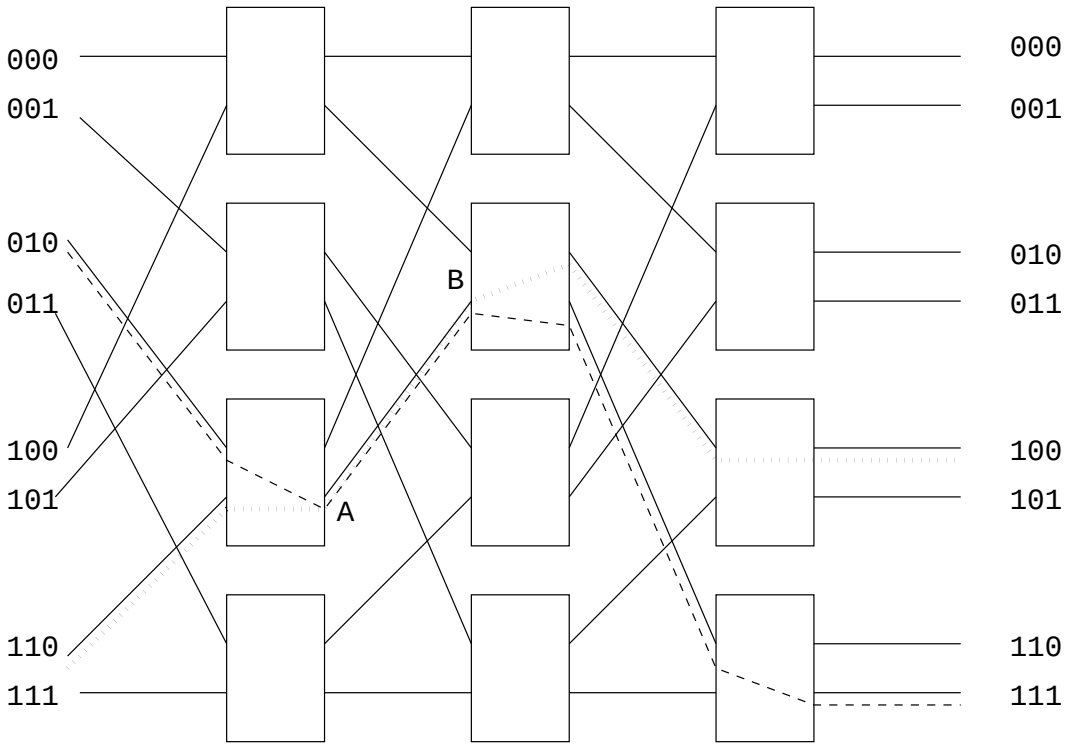


Abbildung 3.8: Die Wege 010 nach 111 und 110 nach 100 wollen beide die Verbindung AB benutzen.

3.2 Statische Verbindungsnetzwerke

Die p Prozessoren eines Message-Passing-Systems kommunizieren über Punkt-zu-Punkt-Verbindungen in einem statischen Verbindungsnetzwerk. Wichtige Kriterien zur Beurteilung einer gewählten Topologie sind:

- K_1: Skalierbarkeit (für beliebige p)
- K_2: max. Knotengrad (Anzahl der Nachbarn eines Knotens)
- K_3: Routing (Strategie zum Weiterleiten von Nachrichten)
- K_4: Durchmesser (maximaler Abstand zwischen zwei Knoten)
- K_5: Hamiltonkreis (geschlossener Weg über alle Knoten)
- K_6: Verbindungszusammenhang (*arc connectivity*; minimale Kantenzahl, nach deren Entfernung das Netzwerk zerfällt)
- K_7: Bisektionsweite (minimale Kantenzahl, nach deren Entfernung das Netzwerk in zwei gleich große Hälften zerfällt)
- K_8: Kosten (Anzahl der Kanten)

3.2.1 Clique

Eine Clique besteht aus p Knoten. Jeder Knoten ist mit jedem verbunden.

K_1: ja
K_2: $p-1$
K_3: wähle Ziel in einem Schritt
K_4: 1
K_5: ja

3.2.2 Stern

Ein Stern $S(p)$ besteht aus p Knoten. Ein ausgezeichneter Knoten (*Master*) ist mit jedem anderen Knoten (*Slave*) verbunden.

K_1: ja
K_2: $p-1$
K_3: wähle Ziel in zwei Schritten über *Master*
K_4: 2
K_5: nein

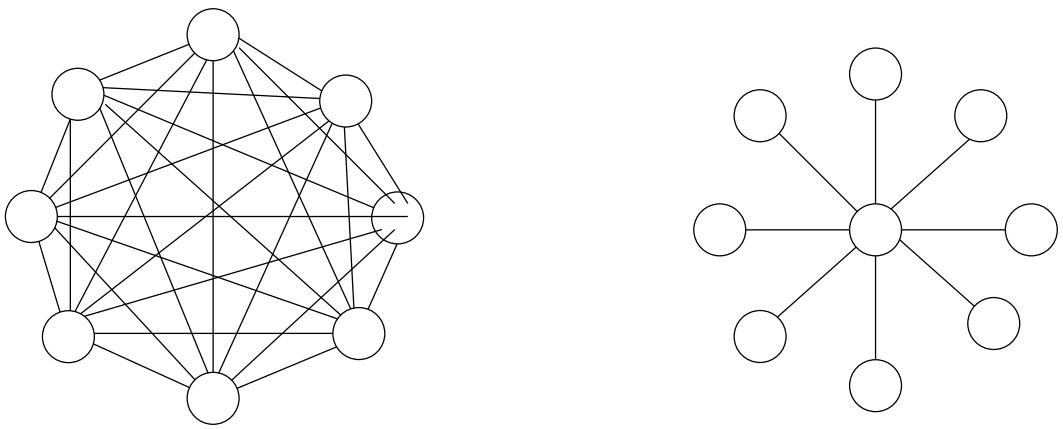


Abbildung 3.9: Clique (a) und Stern (b)

3.2.3 Binärer Baum

Der vollständige binäre Baum $B(k)$ der Höhe k hat $2^{k+1} - 1$ Knoten und besteht aus $k + 1$ Ebenen. Jeder Knoten (bis auf die Wurzel) hat einen Vater, jeder Knoten (bis auf die Blätter) hat zwei Söhne.

K_1: ja
K_2: 3
K_3: laufe vom Start aufwärts zum gemeinsamen Vorfahren, dann abwärts zum Ziel
K_4: $2 \cdot k$
K_5: nein

Zur Vermeidung eines Kommunikationsflaschenhalses werden in einem *Fat Tree* die Links nahe der Wurzel mehrfach ausgelegt. Außerdem repräsentieren nur die Blätter Prozessoren: innere Knoten sind Schaltelemente.

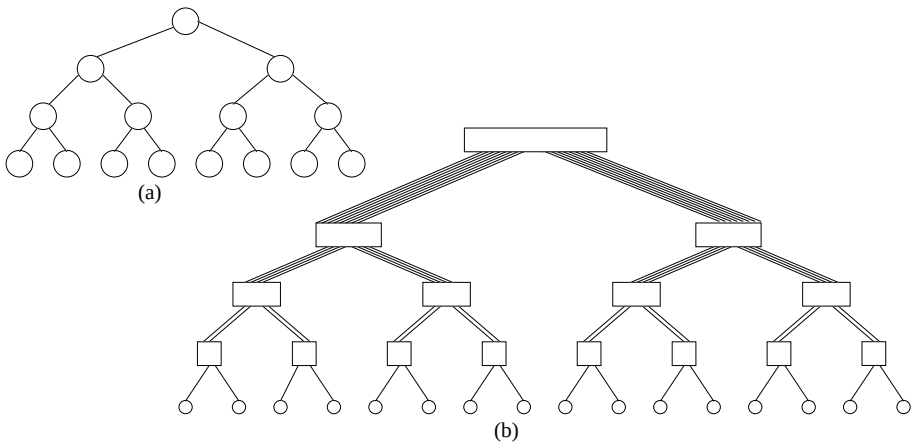


Abbildung 3.10: Binärer Baum $B(3)$ mit 15 Prozessoren (a)
Fat Tree mit 16 Prozessoren (b)

3.2.4 Lineares Array/Ring

Die Knoten eines linearen Arrays sind in einer Reihe angeordnet, ggf. mit *wraparound*. Im letzteren Falle liegt ein Ring vor, und jeder Knoten hat genau zwei Nachbarn ($MC^1(p)$).

K_1: ja
K_2: 2
K_3: wähle Richtung und laufe "geradeaus"
K_4: lineares Array: $p-1$
Ring: $\lfloor \frac{p}{2} \rfloor$
K_5: lineares Array: nein
Ring: ja

(a)

(b)

Abbildung 3.11: Lineares Array (a) und Ring (b)

3.2.5 2D-Gitter

Die Knoten eines quadratischen 2D-Gitters sind in Zeilen und Spalten angeordnet, ggf. mit *wraparound*. In diesem Fall liegt ein *Torus* vor, und jeder Prozessor hat genau vier Nachbarn ($MC^2(p)$).

K_1: ja
K_2: 4
K_3: Wandere horizontal bis zur Zielspalte,
wandere vertikal bis zur Zielzeile.
K_4: ohne *wraparound* $2(\sqrt{p}-1)$
mit *wraparound* $2(\lfloor\frac{\sqrt{p}}{2}\rfloor)$
K_5: mit *wraparound*: ja
ohne *wraparound*: nein, falls p ungerade, ja sonst.

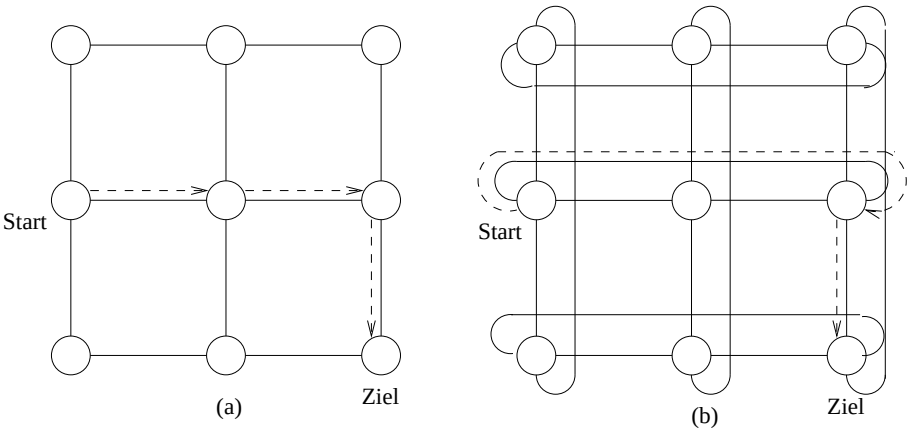


Abbildung 3.12: Routing im 2D-Gitter ohne *wraparound* (a) und mit *wraparound* (b)

3.2.6 3D-Gitter

Mehrere 2D-Gitter werden in der 3. Dimension repliziert, ggf. mit *wraparound*. In diesem Falle liegt ein 3-dimensionaler Torus vor, und jeder Knoten hat genau 6 Nachbarn ($MC^3(p)$).

K_1: ja
K_2: 6
K_3: wandere zur Zielfläche, danach zur Zielspalte,
danach zur Zielzeile
K_4: ohne *wraparound*: $3(\sqrt[3]{p}-1)$
mit *wraparound*: $3(\lfloor\frac{\sqrt[3]{p}}{2}\rfloor)$
K_5: für ungerade Prozessorzahl ohne *wraparound*: nein, sonst ja.

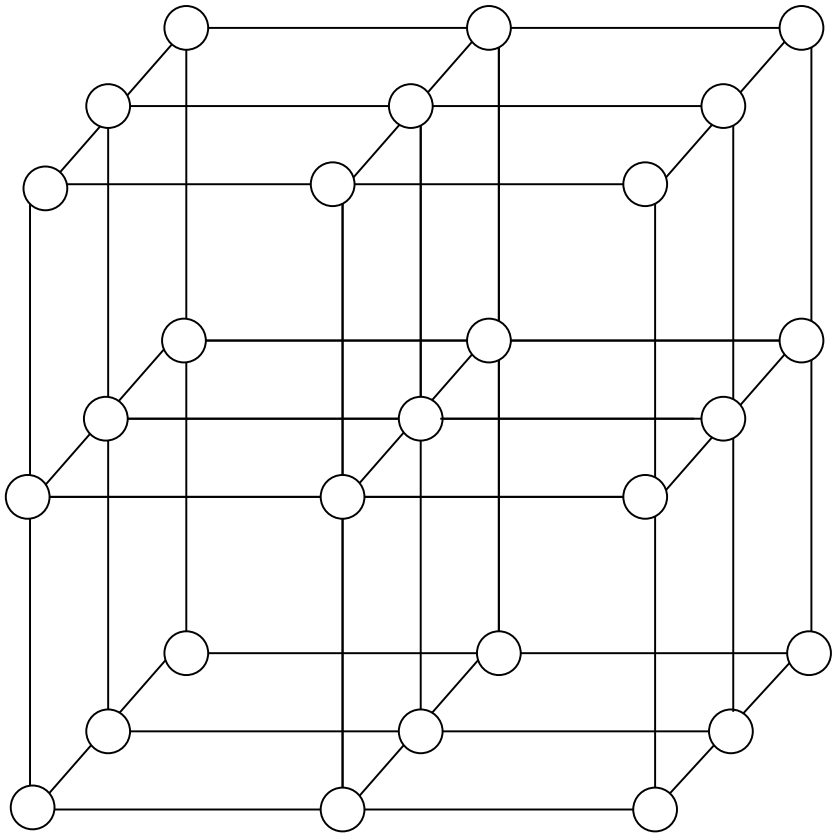


Abbildung 3.13: 3D-Gitter ohne *wraparound*

3.2.7 Hypercube

Ein Hypercube der Dimension k ($HC(k)$) besteht aus $p = 2^k$ Knoten. Jeder Knoten hat k Nachbarn, deren Adresse an genau einem Bit differieren.

K_1: ja
K_2: k
K_3: korrigiere alle zwischen Start- und Zieladresse differierenden Bits durch Benutzung der zuständigen Links
K_4: k
K_5: ja, für $k \geq 2$. Induktion über k: Hypercube der Dimension 2 hat Hamiltonkreis. Hypercube der Dimension k setzt sich zusammen aus 2 Hypercubes der Dimension $k-1$. Verbinde deren Hamiltonwege.

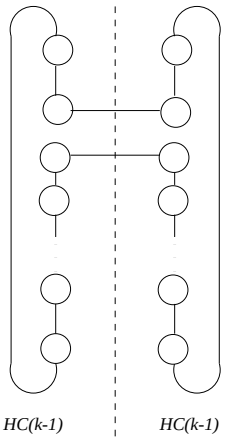


Abbildung 3.14: Verbinden zweier Hypercube-Hamiltonkreise

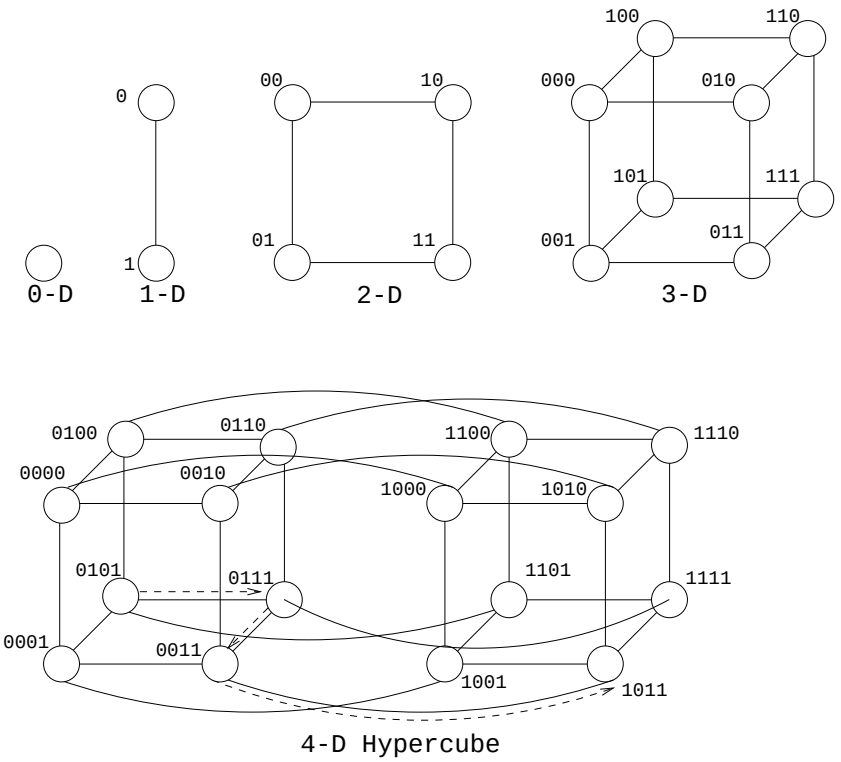


Abbildung 3.15: Hypercubes der Dimension 0, 1, 2, 3, 4.
Routing von Startadresse 0101 über 0111 und 0011 zu 1011.

Es gibt 2 Ansätze, den variablen Knotengrad des Hypercube auf eine Konstante zu drücken unter Beibehaltung der prinzipiellen Verbindungs- und Routing-Struktur: Beim *Butterfly*-Netzwerk existieren $\log p$ abgemagerte Kopien des Hypercube; bei den *Cube Connected Cycles* wird jeder Hypercubeknoten durch einen Ring mit $\log p$ Knoten ersetzt.

3.2.8 Butterfly

Ein Butterfly-Netzwerk $BF(k)$ hat $k+1$ Ränge zu je 2^k Knoten.
Sei (i,j) der j-te Knoten im Rang $i, 0 \leq j < 2^k, 0 \leq i < k$. Er hat Verbindung zu Rang $i+1$ zu den Knoten $(i+1,j)$ und $(i+1,\overline{j})$, wobei $\overline{j}$ aus j entsteht durch Invertierung des i-ten Bits.

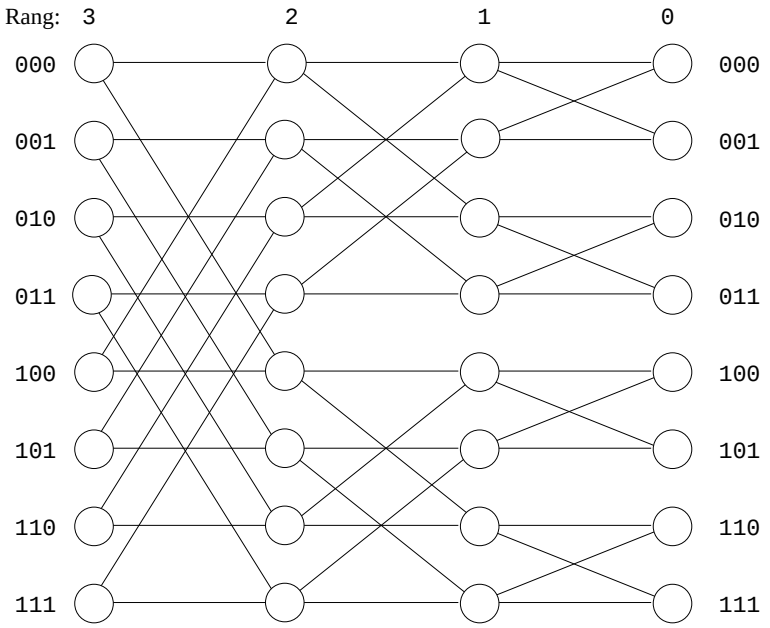


Abbildung 3.16: Butterfly-Netzwerk der Dimension 3

K_1: ja
K_2: 4
K_3: Von jedem Knoten des Rangs 0 läßt sich jeder Knoten des Rangs k in k Schritten erreichen (schrittweises Korrigieren der zwischen Start- und Zieladresse differierenden Bits). Zwei beliebige Start- und Zielknoten steuern zunächst Randränge an: Befindet sich Start im Rang s und Ziel im Rang t und gilt $s \leq t$, steuert Start den Rang 0 und Ziel den Rang k an, sonst umgekehrt. Von den beiden Randknoten aus können sie sich in k Schritten verbinden.
K_4: $2 \cdot k$ für $p = (k+1) \cdot 2^k \Rightarrow k \leq 2 \cdot \log(p)$
K_5: Ein *wraparound* $BF(k)$, bei dem die Knoten in den Rängen 0 und k identifiziert werden, hat einen Hamiltonkreis (s. F. Thomson Leighton: "Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes", Morgan Kaufmann Publishers, 1992, S. 465).

3.2.9 Cube Connected Cycles

Ein *Cube-Connected-Cycles*-Netzwerk der Dimension k ($CCC(k)$) besteht aus $p = k \cdot 2^k$ Knoten, gruppiert in 2^k Kreisen zu je k Knoten. Sei (i, j) der j-te Knoten in Kreis i. Zusätzlich zu den Kreisverbindungen gibt es eine Kante zum Knoten $(\overline{i}, j)$, wobei $\overline{i}$ aus i entsteht durch Invertierung des j-ten Bits.

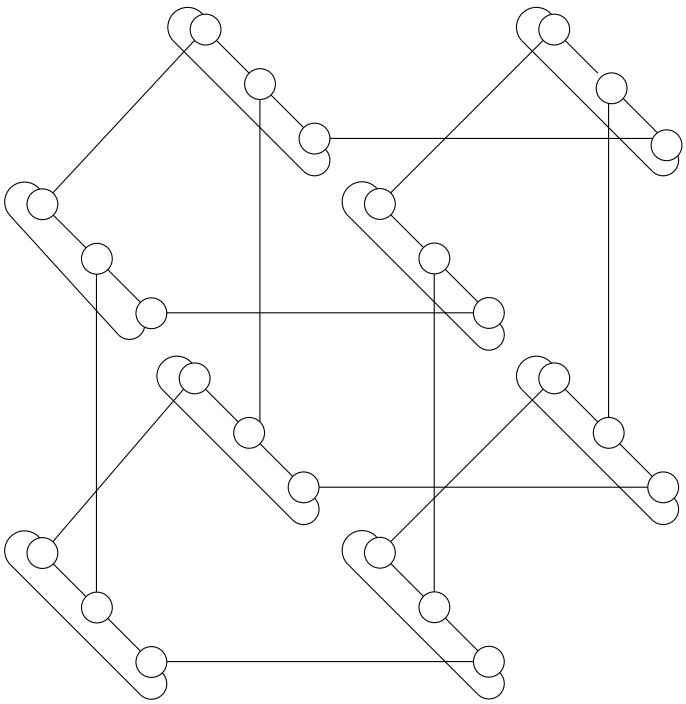


Abbildung 3.17: $CCC(3)$ mit $3 \cdot 2^3 = 24$ Knoten

K_1: ja

K_2: 3

K_3: Um von x nach y zu gelangen: Passe schrittweise die Bits von x den Bits von y an. Falls $x_i = \overline{y}_i$, dann wechsel den Kreis und rücke im neuen Kreis eins weiter, sonst rücke im alten Kreis eins weiter.

K_4: $\lfloor \frac{5k}{2} \rfloor - 2$

K_5: ja (s. F. Thomson Leighton: "Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes", Morgan Kaufmann Publishers, 1992, S. 466).

3.2.10 Shuffle Exchange

Ein *Shuffle Exchange*-Netzwerk der Dimension k ($SE(k)$) besteht aus $p = 2^k$ Knoten. Es gibt zwei Arten von Kanten:

`exchange`: zwischen Prozessoren, deren Adressen bis auf das *low-order*-Bit übereinstimmen,

`shuffle`: von i nach $(2 \cdot i) mod (p-1)$ für $i = 0, \ldots, p-2$ und von $p-1$ nach $p-1$.

Eine Shuffle-Kante bewirkt eine zyklische Linksrotation der Binärdarstellung.

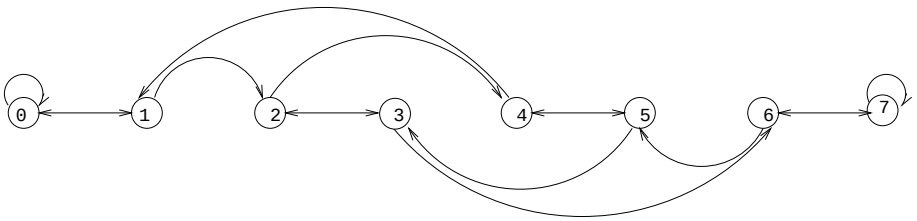


Abbildung 3.18: Shuffle-Exchange-Netzwerk der Dimension 3

K_1: ja

K_2: 3 (wenn Richtung ignoriert wird)

K_3: Um von x nach y zu gelangen: Passe schrittweise die Bits von x den Bits von y an. Konstruiere jeweils im letzten Bit (durch Übernahme des vordersten (shuffle) oder durch Übernahme des vordersten mit Invertierung (shuffle + exchange)) das nächste Bit der Zieladresse.

K_4: $2k - 1$

K_5: nein

3.2.11 de Bruijn

Ein *de Bruijn*-Netzwerk der Dimension k ($dB(k)$) besteht aus $p = 2^k$ Knoten. Von einem Knoten mit dem Binärmuster $x_1 x_2 \ldots x_k$ führt eine Shuffle-Kante zu dem Knoten mit Binärmuster $x_2 \ldots x_k x_1$ und eine Shuffle-Exchange-Kante zu dem Knoten mit Binärmuster $x_2 \ldots x_k \overline{x}_1$.

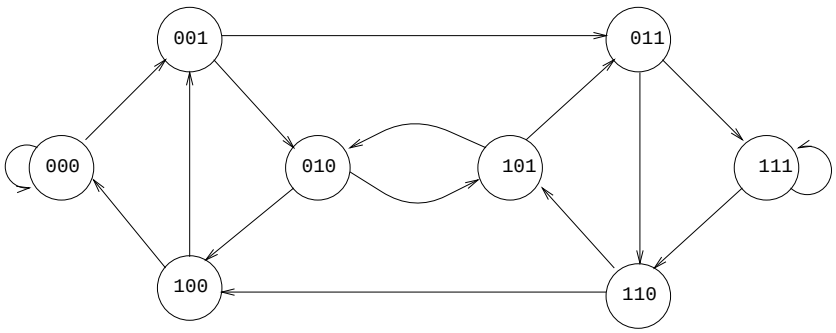


Abbildung 3.19: de Bruijn-Netzwerk der Dimension 3

K_1: ja
K_2: 4 (wenn Richtung ignoriert wird)
K_3: Um von x nach y zu gelangen: Passe schrittweise die Bits von x den Bits von y an. Konstruiere jeweils im letzten Bit (durch Übernahme oder Invertierung des vordersten Bits) das nächste Bit der Zieladresse.
K_4: k
K_5: ja

Zum Nachweis der Hamiltonkreis-Eigenschaft benötigen wir die Definition des Kantengraphen: Sei G = (V, E) ein gerichteter Graph. Der *Kantengraph* $\hat{G}$ von G ist definiert als $\hat{G} = (\mathrm{E}, \hat{E})$ mit

$$\hat{E} = \{(e_1, e_2) \mid \quad e_1, e_2 \in E, e_1 \neq e_2, \exists u, v, w \in V \text{ mit } e_1 = (u, v) \wedge e_2 = (v, w)\}$$

Offenbar hat $\hat{G}$ so viele Knoten wie G Kanten.

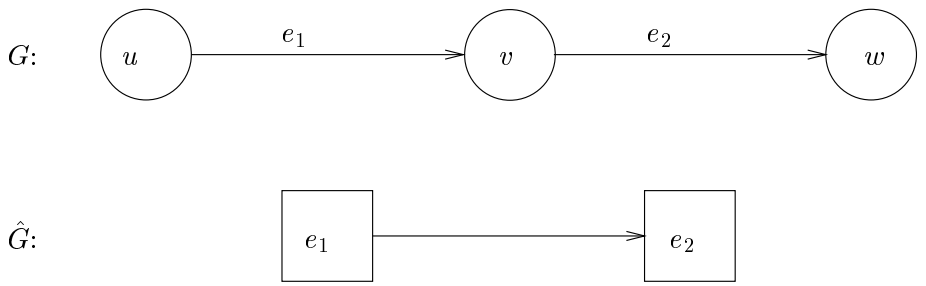


Abbildung 3.20: Beziehung zwischen Graph G und Kantengraph $\hat{G}$

Beim de Bruijn-Graphen läßt sich die Kante von u nach v mit uv_0 eindeutig beschreiben. Somit gilt

u	=	u_{k-1}	u_{k-2}	u_{k-3}	...	u_0		
v	=		u_{k-2}	u_{k-3}	...	u_0	v_0	
w	=			u_{k-3}	...	u_0	v_0	w_0
e_1	=	u_{k-1}	u_{k-2}	u_{k-3}	...	u_0	v_0	
e_2	=		u_{k-2}	u_{k-3}	...	u_0	v_0	w_0

Also entstehen in $\hat{E}$ genau die de Bruijn-Kanten, d.h., der Kantengraph von $dB(k)$ ist der Graph $dB(k+1)$.
Da der $dB(k)$ einen Eulerkreis hat (denn jeder Knoten hat 2 Eingangs- und 2 Ausgangskanten), besitzt $dB(k+1)$ einen Hamiltonkreis.

3.3 Netzwerkeinbettungen

Seien $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ ungerichtete Graphen. Eine injektive Abbildung $f : V_1 \to V_2$ heißt *Einbettung* von G_1 in G_2. Für die Kante $(x, y) \in E_1$ entsteht dabei als *Kantenstreckung* die Länge des kürzesten Weges im Graphen G_2 zwischen $f(x)$ und $f(y)$. Mit *Kantenauslastung* wird die Anzahl der Wege beschrieben, die in G_2 über eine Kante führen.

3.3.1 Ring in Hypercube

Ein Ring der Länge 2^k läßt sich in den Hypercube $HC(k)$ mit Kantenstreckung 1 mit Hilfe eines k-Bit Graycodes einbetten. Ein Graycode besteht aus einer Folge von Binärstrings, die sich jeweils an genau einem Bit unterscheiden. Ein *k-stelliger gespiegelter Graycode* entsteht aus einem $k-1$-stelligen gespiegelten Graycode durch Spiegelung und Voransetzen von 0 bzw. 1.

```
0  00  000  0000
1  01  001  0001
   11  011  0011
   10  010  0010
       110  0110
       111  0111
       101  0101
       100  0100
            1100
            1101
            1111
            1110
            1010
            1011
            1001
            1000
```

1, 2, 3, 4-Bit gespiegelte Graycodes

3.3.2 Gitter in Hypercube

Sei $G(i, d)$ der i-te String im d-stelligen Graycode. Ein $2^r \times 2^s$ *wraparound*-Gitter kann in einen $r + s$-dimensionalen Hypercube $HC(r + s)$ mit Kantenstreckung 1 und Kantenauslastung 1 eingebettet werden. Hierzu ordne den Knoten (i, j) dem Prozessor $G(i, r) \circ G(j, s)$ zu ($\circ$ bezeichnet die Konkatenation der Adressen).

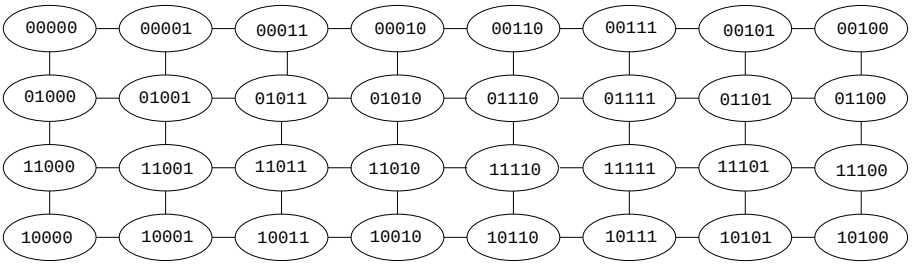


Abbildung 3.21: 4×8-Gitter beschriftet mit Hypercube-Adressen

3.3.3 Binärer Baum im Hypercube

Zunächst wird ein Doppelwurzelbaum $DWB(k)$ in den $HC(k+1)$ eingebettet.

Definition: Ein *Doppelwurzelbaum* $DWB(k)$ entsteht aus einem binären Baum $B(k)$, indem die Wurzel durch eine Kante mit zwei Knoten ersetzt wird.

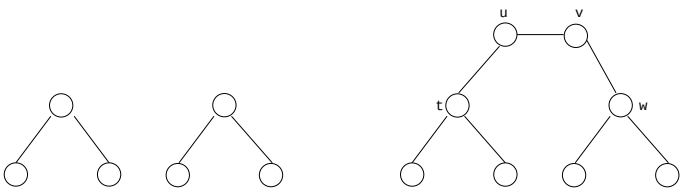


Abbildung 3.22: Zwei $B(1)$ und $DWB(2)$

Offenbar hat $DWB(k)$ 2^{k+1} Knoten.
Satz: $DWB(k)$ ist Teilgraph des $HC(k+1)$, Beweis durch Induktion.
Behauptung: $DWB(k)$ ist Teilgraph des $HC(k+1)$, und die drei Doppelwurzelkanten verlaufen in verschiedenen Dimensionen.
Verankerung: Bild 3.23 zeigt, wie der $DWB(2)$ in den $HC(3)$ eingebettet wird.

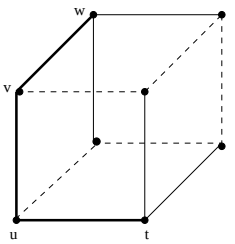


Abbildung 3.23: $DWB(2)$ eingebettet in $HC(3)$ (Doppelwurzelkanten fett)

Induktionsschritt: Sei bis k bewiesen. Durch Vertauschen von Bitpositionen bzw. Invertieren von Bitpositionen in allen Hypercubeadressen läßt sich erreichen, daß zwei $DWB(k)$ im $HC(k+2)$ mit der in Bild 3.24 gewählten Numerierung eingebettet sind. Wie in Bild 3.25 zu sehen, lassen sich beide DWBe der Dimension k zu einem $DWB(k+1)$ zusammenfügen, wobei die drei Doppelwurzelkanten in verschiedenen Dimensionen verlaufen.

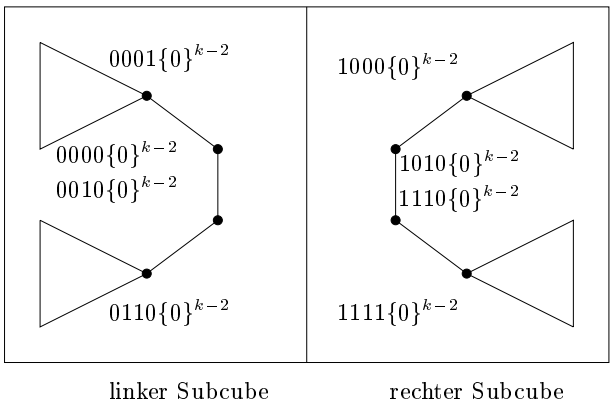


Abbildung 3.24: Gewählte Adressen für zwei $DWB(k)$

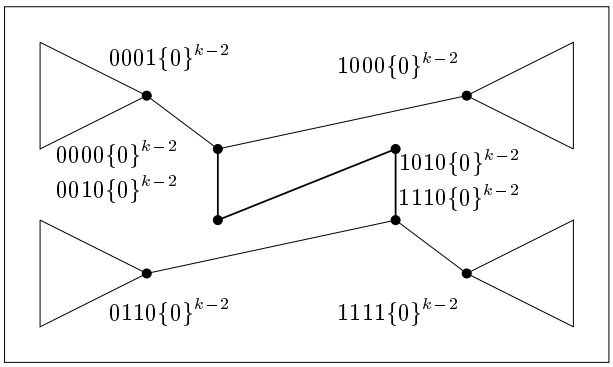


Abbildung 3.25: Zusammenfügen zweier $DWB(k)$ zu einem $DWB(k+1)$

Da ein binärer Baum $B(k)$ aus $DWB(k)$ durch Verschmelzen beider Wurzelknoten entsteht, folgt:

Korollar: Ein binärer Baum $B(k)$ läßt sich mit Kantenstreckung 2 (an einer einzigen Kante) in den $HC(k+1)$ einbetten.

Kapitel 4

Basiskommunikation

4.1 Kosten

Beim Versenden einer Nachricht entsteht eine Kommunikationslatenz, die sich zusammensetzt aus

Startup time t_s:	Aufbereitungszeit beim sendenden Prozessor
Per hop time t_h: = Knotenlatenz	Zeit zur Übertragung des Nachrichtenkopfes
Per word transfertime t_w:	Übertragungszeit pro Wort

Zwei wesentliche Routingstrategien werden benutzt:

- Store-and-Forward-Routing:
  Jeder beteiligte Prozessor empfängt die Nachricht komplett und sendet sie dann weiter. Das Senden von m Worten über insgesamt l Links dauert

$$t_{comm} = (t_s + t_h + m \cdot t_w) \cdot l$$

- Cut-Through-Routing:
  Die Nachricht wird in sogenannte *flow control digits = flits* zerteilt und flit-weise verschickt. Das Senden von m Worten über l Links dauert

$$t_{comm} = t_s + l \cdot t_h + m \cdot t_w$$

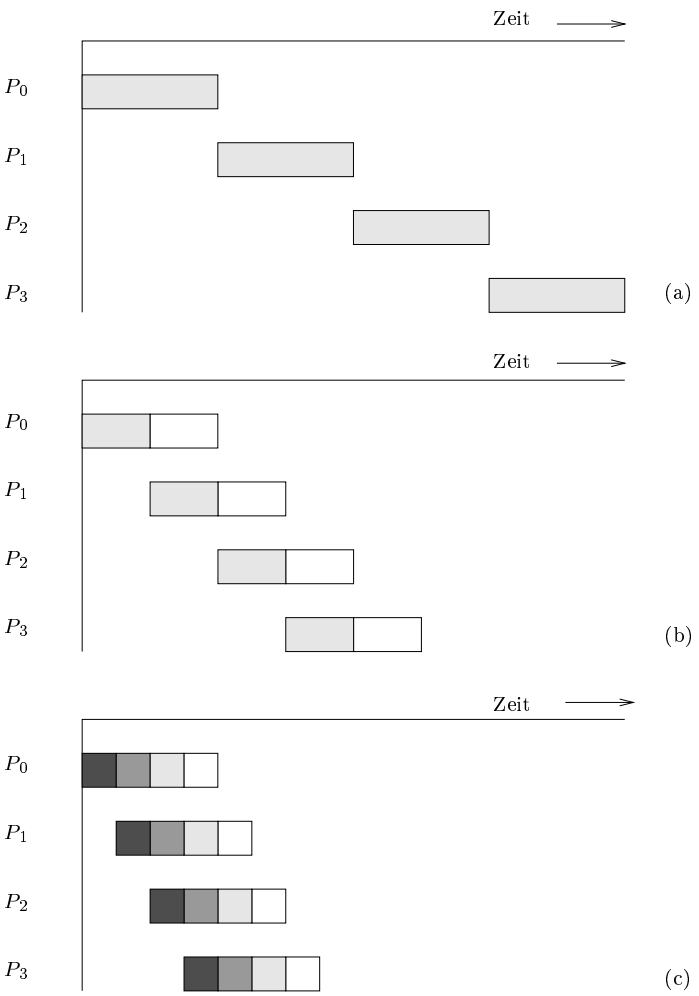


Abbildung 4.1: Kommunikationsablauf beim Store-and-Forward-Routing (a), Cut-Through mit 2 Paketen (b), Cut-Through mit 4 Paketen (c) (Startup und Per-hop-time sind jeweils als 0 angenommen)

Cut-Through-Routing, auch *Wormhole-Routing* genannt, ist schneller als Store-and-Forward-Routing, erhöht aber die Deadlockgefahr.

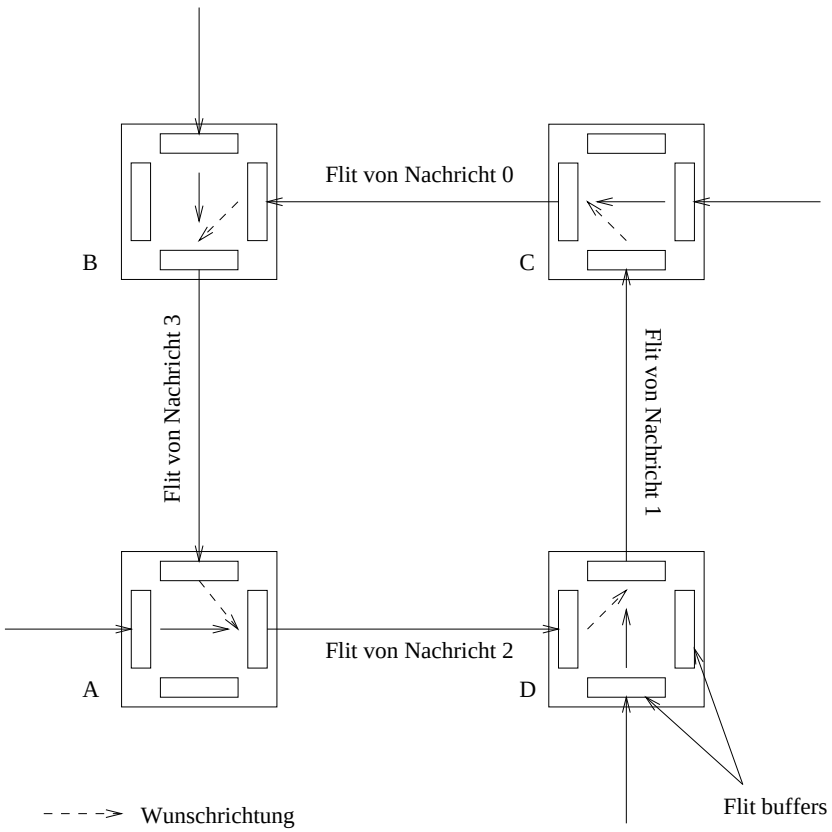


Abbildung 4.2: Deadlock beim Cut-Through-Routing

4.2 One-to-All Broadcast

Ein ausgezeichneter Prozessor verschickt an alle anderen $p-1$ Prozessoren dieselbe Nachricht der Länge m. Das duale Problem heißt *All-to-One Broadcast* und besteht darin, von allen p Prozessoren Daten der Größe m einzusammeln, zu verknüpfen und bei einem Prozessor abzuliefern. Die Verknüpfung ist assoziativ, und die durch die Verknüpfung erzeugte Nachricht hat weiterhin die Größe m.

Store-and-Forward

Im Ring wird, ausgehend vom *Master*, die Nachricht in zwei Richtungen propagiert.

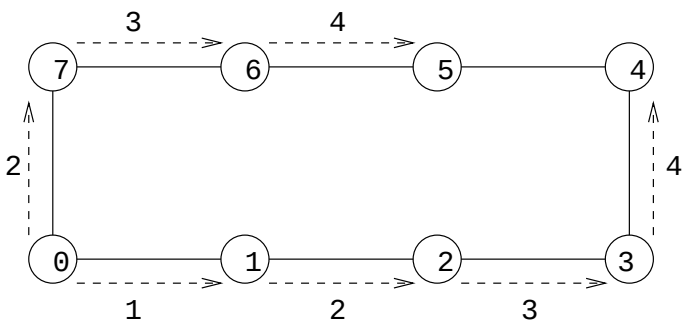


Abbildung 4.3: Store-and-Forward im Ring.
Gestrichelte Kanten sind mit dem jeweiligen Zeitschritt beschriftet.

Die Zeit beträgt

$$T_{one-to-all} = (t_s + t_h + t_w \cdot m) \cdot \lceil \frac{p}{2} \rceil$$

Im 2-dimensionalen Gitter wird zunächst eine Zeile als Ring mit der Nachricht versorgt, danach werden alle Spalten gleichzeitig wie Ringe versorgt.

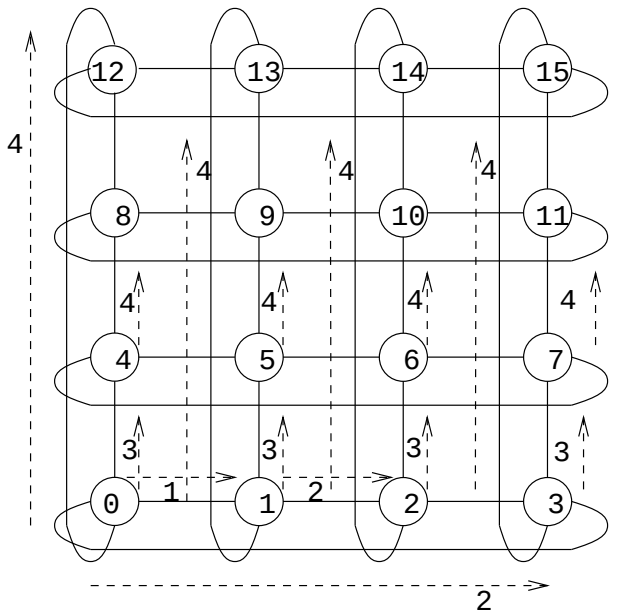


Abbildung 4.4: Store-and Forward im MC^2

Da eine Zeile bzw. Spalte $\sqrt{p}$ Prozessoren aufweist, beträgt die Zeit

$$T_{one-to-all} = 2 \cdot (t_s + t_h + t_w \cdot m) \cdot \lceil \frac{\sqrt{p}}{2} \rceil$$

Für ein dreidimensionales Gitter ergibt sich

$$T_{one-to-all} = 3 \cdot (t_s + t_h + t_w \cdot m) \cdot \lceil \frac{\sqrt[3]{p}}{2} \rceil$$

Im Hypercube sendet nacheinander für jede Dimension $d-1, d-2, \ldots, 0$ der Prozessor mit d-tem Bit $= 0$ an den Prozessor mit d-tem Bit $= 1$. Dabei sind nur solche Prozessoren aktiv, bis zu denen die Nachricht schon vorgedrungen ist.

```
procedure ONE_TO_ALL_BC(d, my_id, X) /* Broadcast X from 0 to all */
   mask := 2^d - 1; /* Set lower d bits of mask to 1 */
   for i := d - 1 downto 0 do /* Outer loop */
      mask := mask XOR 2^i; /* Set bit i of mask 0 */
      if (my_id AND mask) = 0 then
      /* if the lower i bits of my_id are 0 */
         if (my_id AND 2^i) = 0 then
            msg_destination := my_id XOR 2^i;
            send X to msg_destination;
         else
            msg_source := my_id XOR 2^i;
            receive X from msg_source;
      end
   end
end
```

Die Gesamtdauer beträgt

$$T_{one-to-all} = (t_s + t_h + t_w \cdot m) \cdot \log p$$

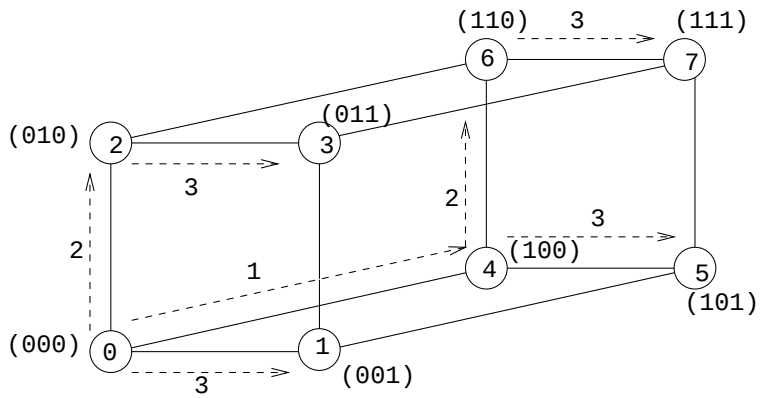


Abbildung 4.5: One-to-All im Hypercube, zeitlicher Verlauf gestrichelt

In abgewandelter Form kann die Prozedur `ONE_TO_ALL_BC` auch zum Einsammeln von Nachrichten verwendet werden. Es sendet nacheinander für Dimension $d-1, d-2, \ldots, 0$ der Prozessor mit d-tem Bit $= 0$ an den Prozessor mit d-tem Bit $= 1$, wo anschließend die Verknüpfung stattfindet. Dabei werden solche Prozessoren passiv, die ihren Beitrag bereits abgeschickt haben.

Cut-Through-Routing

Im Ring läßt sich die Kommunikationszeit durch CT-Routing verbessern. In jeder Iteration findet eine Kommunikation zwischen Partnern wie im `ONE_TO_ALL_BC`-Algorithmus für den Hypercube statt. D.h., bei 8 Prozessoren sendet zuerst `000` nach `100`, dann gleichzeitig `000` nach `010` und `100` nach `110`, dann gleichzeitig `000` nach `001`, `010` nach `011`, `100` nach `101` und `110` nach `111`.

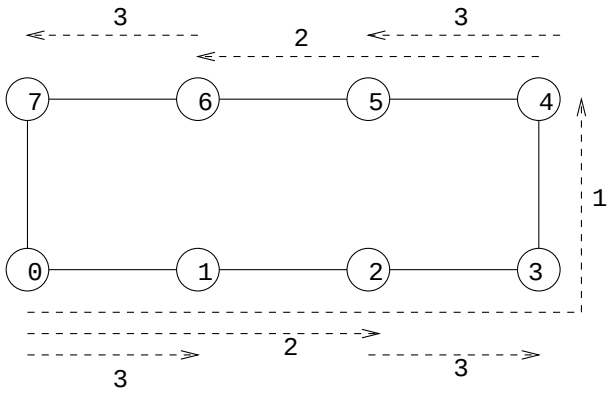


Abbildung 4.6: One-to-All mit CT im Ring

In der i-ten Iteration dauert die Kommunikation

$$t_s + t_w \cdot m + t_h \cdot \frac{p}{2^i}$$

Die Gesamtzeit lautet daher

$$T_{one-to-all} = \sum_{i=1}^{\log p} (t_s + t_w \cdot m + t_h \cdot \frac{p}{2^i})$$

$$= t_s \cdot \log p + t_w \cdot m \cdot \log p + t_h \cdot (p-1)$$

Für große m und kleine t_s, t_h bedeutet dies gegenüber SF-Routing eine Beschleunigung um den Faktor $\frac{p}{2 \cdot \log p}$.

Im Gitter läßt sich dieselbe Idee zunächst zum Versorgen einer Zeile anwenden, danach werden analog alle Spalten bearbeitet. Jede dieser beiden Phasen dauert

$$(t_s + t_w \cdot m) \cdot \log \sqrt{p} + t_h(\sqrt{p} - 1) \ .$$

Daraus ergibt sich

$$T_{one-to-all} = (t_s + t_w \cdot m) \log p + 2 \cdot t_h(\sqrt{p} - 1)$$

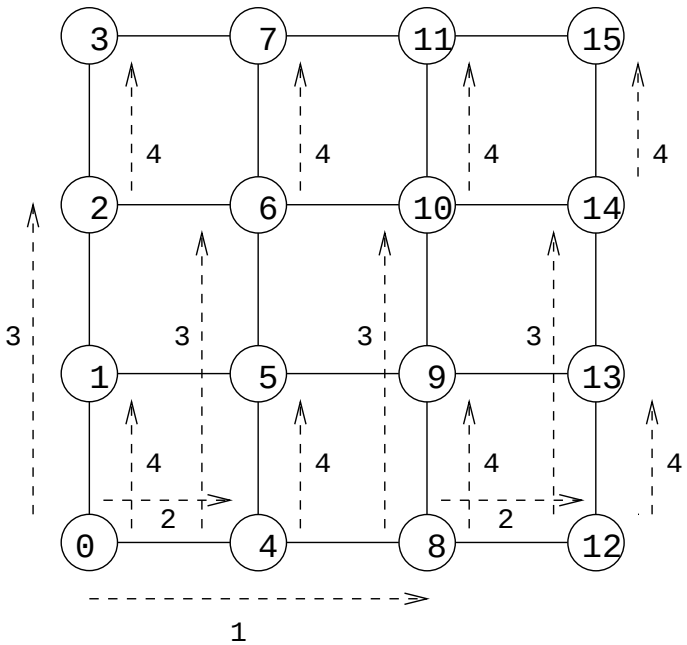


Abbildung 4.7: One-to-all mit *Cut-Through-Routing* im Gitter

4.3 All-to-All Broadcast

Jeder Prozessor verschickt an alle anderen Prozessoren seine spezielle Nachricht. Die Nachrichten werden vereinigt.

Store-and-Forward

Im Ring sendet jeder Prozessor zunächst seine Nachricht an seinen rechten Nachbarn und reicht danach von links erhaltene Nachrichten weiter.

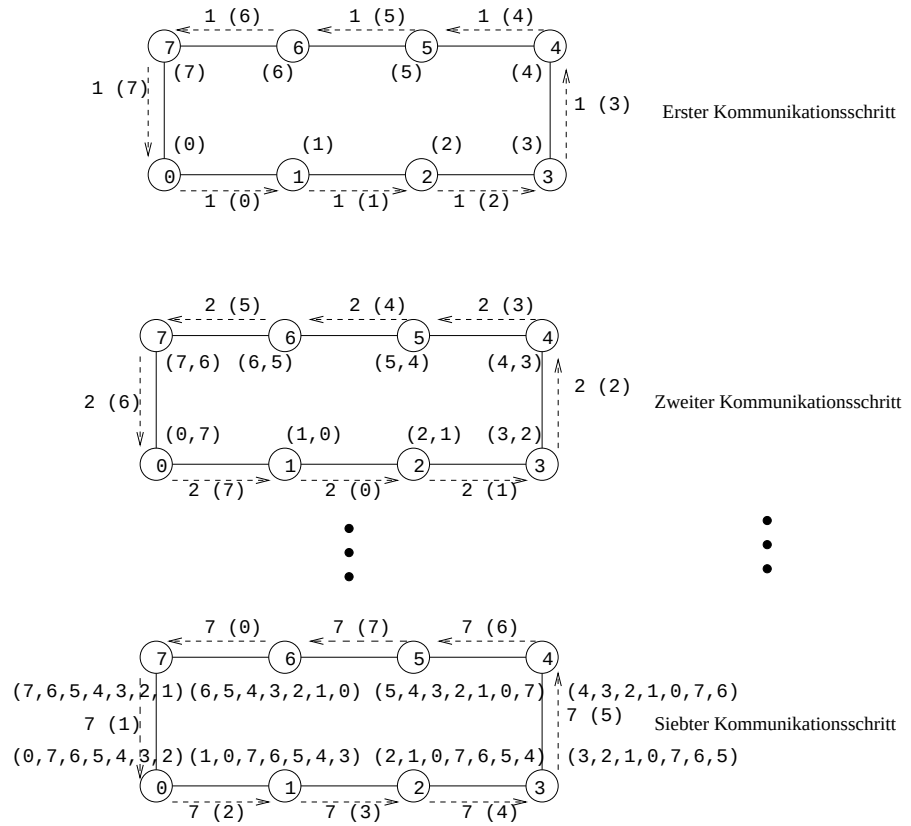


Abbildung 4.8: All-to-All im Ring
Gestrichelte Kanten sind markiert mit dem Zeitschritt und, in Klammern, mit der Absenderkennung der Nachricht.
Knoten sind beschriftet mit der Menge von vorliegenden Absenderkennungen.

Die Gesamtzeit beträgt

$$T_{all-to-all} = (t_s + t_h + t_w \cdot m) \cdot (p - 1)$$

Im Gitter erhält zunächst jeder Prozessor einer Zeile mit Hilfe von *All-to-All im Ring* die gesamte Zeileninformation. Danach werden innerhalb der Spalten diese angesammelten Informationen herumgeschickt. Die erste Phase dauert

$$(t_s + t_h + t_w \cdot m) \cdot (\sqrt{p} - 1) \ .$$

Die zweite Phase benötigt wegen der bereits angesammelten Daten

$$(t_s + t_h + t_w \cdot m \cdot \sqrt{p}) \cdot (\sqrt{p} - 1) \ .$$

Daraus ergibt sich

$$T_{all-to-all} = 2 \cdot (t_s + t_h) \cdot (\sqrt{p} - 1) + t_w \cdot m(p-1) \ .$$

Im Hypercube vereinigen im i-ten Durchlauf die Prozessoren, die sich im i-ten Bit unterscheiden, ihre Informationen (siehe Bild 4.9).

```
   result := my_msg;
   for i := 0 to d-1 do
      partner := my_id XOR 2^i;
      send result to partner;
      receive msg from partner;
      result := result ∪ msg;
   end;
end
```

In der i-ten Phase ist die Nachrichtengröße $m \cdot 2^i$. Ein Austausch dauert $t_s + t_h + 2^{i-1}$. Also beträgt die Gesamtzeit

$$T_{all-to-all} = \sum_{i=1}^{\log p - 1} (t_s + t_h + 2^i \cdot t_w \cdot m)$$

$$= (t_s + t_h) \cdot \log p + t_w \cdot m \cdot (p-1) \ .$$

Wird statt einer Vereinigung von Daten eine Verknüpfung durchgeführt, so entsteht aus *All-to-All* eine *Reduktion*, bei der die Informationsmenge in jedem Schritt gleich bleibt. Für die Summenbildung ist $m = 1$, und es folgt

$$T_{reduktion} = (t_s + t_h + t_w) \cdot \log p \ .$$

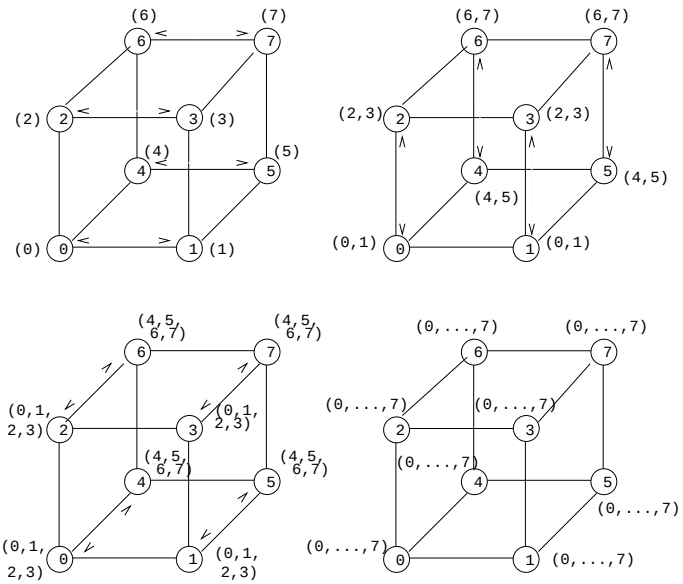


Abbildung 4.9: *All-to-All* im Hypercube

4.4 Echo-Algorithmus

Ein ausgezeichneter Prozessor fordert alle anderen Prozessoren auf, eine Nachricht an ihn zu schicken. Kenntnis der Topologie ist für keinen Prozessor erforderlich.

Arbeitsweise:
Zu Beginn sind alle Knoten weiß. Der Initiator der Nachricht wird rot und verschickt rote Nachrichten (Frage) an seine Nachbarn.
Ein weißer Knoten wird bei Erhalt einer roten Nachricht rot, merkt sich die Aktivierungskante und sendet über die restlichen Kanten rote Nachrichten.
Hat ein roter Knoten auf allen seinen Kanten (rote oder grüne) Nachrichten erhalten, so wird er grün und sendet eine grüne Nachricht (Antwort) über die Aktivierungskante.

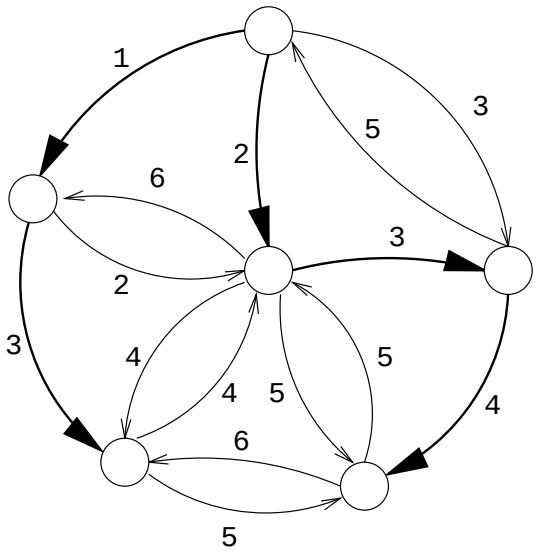


Abbildung 4.10: Verlauf des Echo-Algorithmus.
Die Kanten sind markiert mit dem Zeitpunkt der roten Nachricht.
Die Aktivierungskanten sind fett gezeichnet.

4.5 Terminierung

Prozesse seien im Zustand *aktiv* (noch Arbeit vorhanden) oder *passiv* (keine Arbeit vorhanden).

Ein aktiver Prozeß kann spontan passiv werden. Ein passiver Prozeß wird durch Erhalt eines Auftrags wieder aktiv.

Frage: Sind alle passiv?

Zur Klärung dieser Frage wird ein Hamilton-Kreis in der Topologie benutzt, auf dem ein Token weitergereicht wird, welches vom Master initiiert wird. Das Token wird nur von passiven Prozessen weitergereicht.

Zu Beginn sind alle Prozesse weiß. Ein Prozeß wird schwarz durch Verschicken eines Auftrags.
Master startet im passiven Zustand ein weißes Token und wird weiß.
Ein weißer Prozeß reicht Token so weiter wie erhalten. Ein schwarzer Prozeß reicht Token schwarz weiter und wird weiß.
Erhält weißer Master ein weißes Token, so sind alle passiv.
Erhält Master ein schwarzes Token, so reicht er es weiß weiter.

Kapitel 5

Performance

Zur Beurteilung eines parallelen Algorithmus werden verschiedene Maße verwendet.

Sequentialzeit	T_s:	Zeit zwischen Anfang und Ende der Rechnung auf einem Single-Prozessor-System
Parallelzeit	T_p:	Zeit zwischen Anfang und Ende der Rechnung auf einem Multiprozessorsystem mit p Prozessoren
Speedup	S:	T_s/T_p, wobei T_s die Sequentialzeit des besten sequentiellen Algorithmus ist
Effizienz	E:	S/p
Kosten	C:	$p \cdot T_p$

Ein paralleler Algorithmus heißt *kostenoptimal*, falls seine Kosten proportional zur Sequentialzeit sind.

Beispiel: Addieren von n Zahlen auf einem Hypercube mit n Prozessoren benötigt Zeit $O(\log n)$ gemäß *All-to-One-Broadcast* im Kapitel 4.2
$\Rightarrow$ Speedup $= O(n/\log n)$
$\Rightarrow$ Effizienz $= O(1/\log n)$
$\Rightarrow$ Kosten $= O(n \cdot \log n)$
$\Rightarrow$ nicht kostenoptimal

Für real existierende Multiprozessorsysteme ist die Anzahl p der Prozessoren fest (und daher unabhängig von n). Laufzeit, Speedup, Effizienz und Kosten sind daher Funktionen von n und p.

Beispiel: Beim Addieren von n Zahlen auf einem Hypercube mit p Prozessoren werde jeweils ein Zeitschritt benötigt zum Addieren zweier Zahlen und zum Versenden einer Zahl. Jede der n/p Teilfolgen wird zunächst in $n/p-1$ Schritten lokal aufsummiert und dann in $\log p$ Phasen mit je einer Kommunikation und einer Addition zusammengeführt. Daraus ergibt sich $n/p-1+2\cdot\log p$. Für große n, p läßt sich der Term -1 vernachlässigen. Also gilt

$$\begin{aligned} T_p &= \frac{n}{p} + 2\cdot\log p \\ S &= \frac{n}{n/p + 2\cdot\log p} = \frac{n\cdot p}{n + 2\cdot p\cdot\log p} \\ E &= S/p = \frac{n}{n + 2p\cdot\log p} \\ C &= n + 2\cdot p\cdot\log p \end{aligned}$$

$\Rightarrow$ Solange $n = \Omega(p\cdot\log p)$, ist der Algorithmus kostenoptimal.

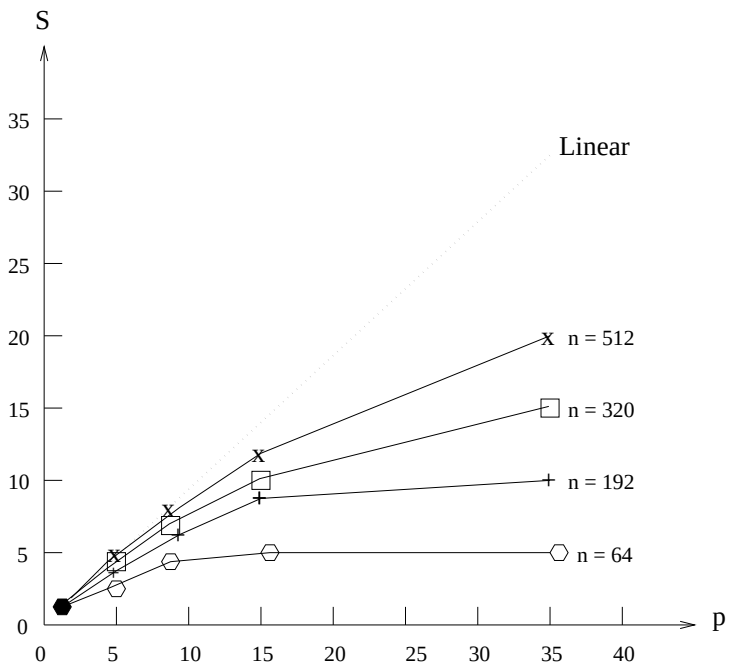


Abbildung 5.1: Speedupkurven für verschiedene Problemgrößen beim Addieren im Hypercube

n	$p=1$	$p=4$	$p=8$	$p=16$	$p=32$
64	1.0	0.80	0.57	0.33	0.17
192	1.0	0.92	0.80	0.60	0.38
320	1.0	0.95	0.87	0.71	0.50
512	1.0	0.97	0.91	0.80	0.62

Tabelle 5.1: Effizienzen für verschiedene Problemgrößen n und Prozessorzahlen p

Offenbar fällt die Effizienz mit wachsender Prozessorzahl und steigt mit wachsender Problemgröße. Ein paralleles System heißt *skalierbar*, wenn sich bei wachsender Prozessorzahl eine konstante Effizienz halten läßt durch geeignetes Erhöhen der Problemgröße.

Beispiel: Laut Tabelle 5.1 beträgt die Effizienz 0.80 für $n = 64$ Zahlen und $p = 4$ Prozessoren. Die Beziehung zwischen Problemgröße und Prozessorzahl lautet $n = 8p \cdot \log p$. Wird die Prozessorzahl auf $p = 8$ erhöht, muß daher die Problemgröße auf $n = 8 \cdot 8 \cdot \log 8 = 192$ wachsen, um die Effizienz von 80% zu halten.

Um eine Funktion für die Skalierbarkeit zu erhalten, definieren wir zunächst als *Problemgröße* W die Laufzeit des besten sequentiellen Algorithmus. Die *Overheadfunktion* T_0 drückt die Differenz zwischen den parallelen und sequentiellen Kosten aus:

$$T_0(W,p) = p \cdot T_p - W$$

Zum Beispiel beträgt der Overhead für das Addieren von n Zahlen im Hypercube:

$$p \cdot (\frac{n}{p} + 2 \cdot \log p) - n = 2 \cdot p \cdot \log p$$

Durch Umformen erhalten wir:

$$\begin{aligned} T_p &= \frac{W + T_0(W,p)}{p} \\ S &= \frac{W}{T_p} = \frac{W \cdot p}{W + T_0(W,p)} \\ E &= \frac{S}{p} = \frac{W}{W + T_0(W,p)} = \frac{1}{1 + T_0(W,p)/W} \end{aligned}$$

Daraus folgt:

$$\begin{aligned} \frac{1}{E} &= 1 + T_0(W,p)/W \\ \frac{1}{E} - 1 &= T_0(W,p)/W \\ \frac{1-E}{E} &= T_0(W,p)/W \\ W &= T_0(W,p) \cdot \frac{E}{1-E} \end{aligned}$$

Zu gegebener Effizienz E ist $K = \frac{E}{1-E}$ eine Konstante, d.h.

$$W \quad = \quad K \cdot T_0(W, p) \ .$$

Diese Beziehung, genannt *Isoeffizienzfunktion*, drückt das erforderliche Problemgrößenwachstum in Abhängigkeit von p aus.
Beispiel: Der Overhead für das Addieren im Hypercube beträgt $2 \cdot p \cdot \log p$. Für $E = 0.8$ lautet die Isoeffizienzfunktion

$$W = \frac{0.8}{0.2} \cdot 2 \cdot p \cdot \log p = 8p \cdot \log p \ .$$

Wächst die Prozessorzahl von p auf p', so muß die Problemgröße um den Faktor

$$(p' \cdot \log p')/(p \cdot \log p)$$

wachsen.

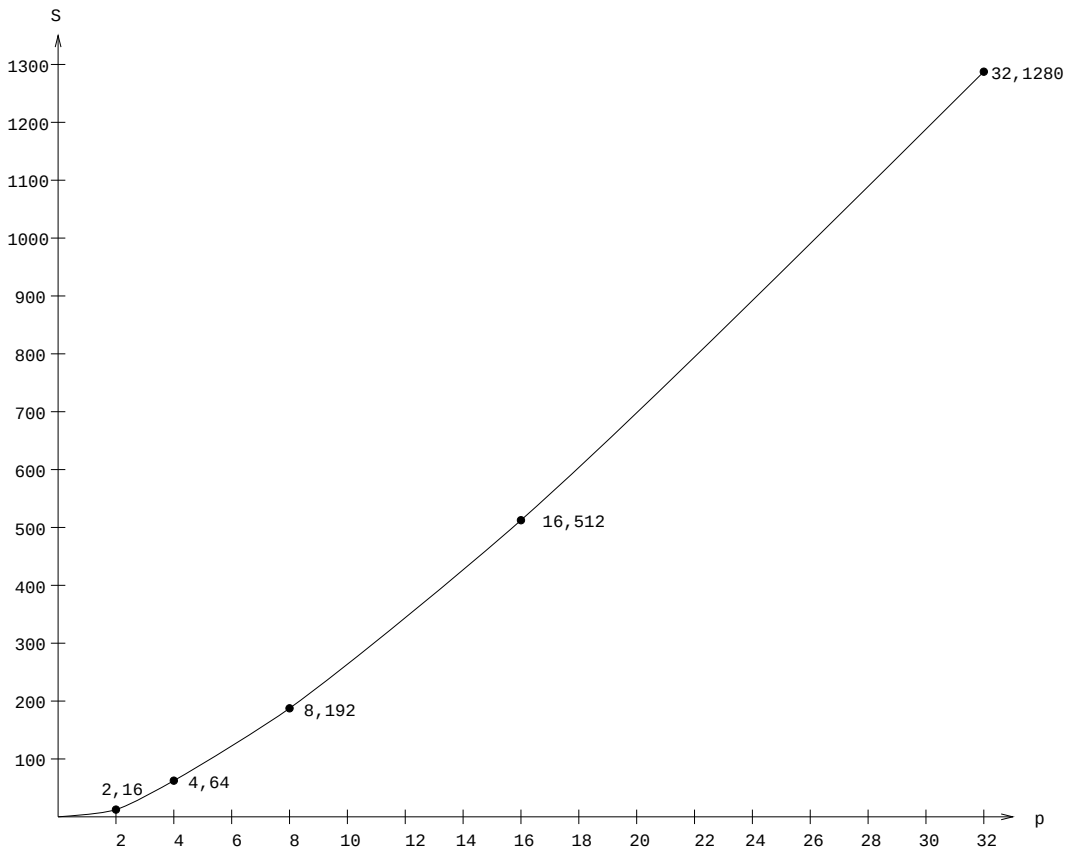


Abbildung 5.2: Isoeffizienzfunktion $8p \cdot \log p$

Kapitel 6

Matrix-Algorithmen

Es werden Verfahren zur Bearbeitung dicht besetzter Matrizen behandelt.

6.1 Partitionierung

Eine $n \times n$-Matrix wird auf ein System von p Prozessoren verteilt durch

Streifenpartitionierung: Folgen von jeweils n/p Zeilen bzw. Spalten werden in Blockstreifen verteilt (d.h., Prozessor P_i erhält $(n/p) \cdot i,\ (n/p) \cdot i + 1, (n/p) \cdot i + 2, ...,$ $(n/p) \cdot (i+1) - 1)$ oder in Zyklenstreifen verteilt (d.h., Prozessor P_i erhält $i, i + p,$ $i + 2 \cdot p, ...,\ i + n - p)$. Mögliche Granularität: n Prozessoren.

Schachbrettpartitionierung: Zusammenhängende Teilmatrizen werden an Prozessoren verteilt, d.h., die $n \times n$-Matrix wird in Blöcke der Größe $(n/\sqrt{p}) \times (n/\sqrt{p})$ partitioniert. Mögliche Granularität: n^2 Prozessoren.

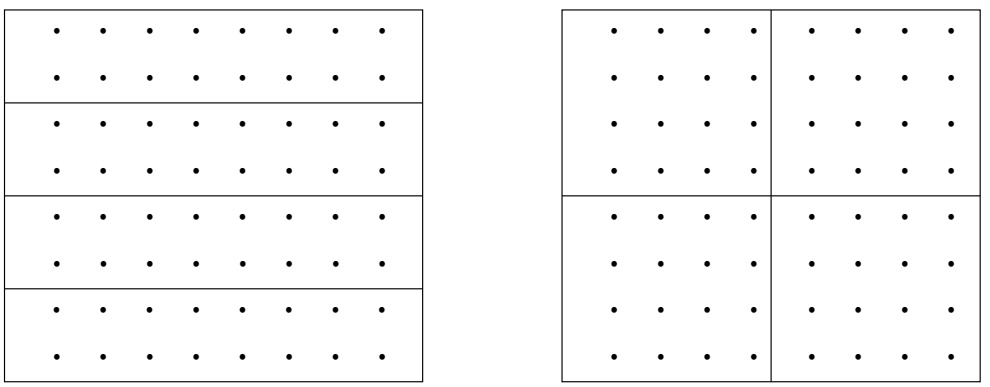


Abbildung 6.1: 8×8-Matrix partitioniert für 4 Prozessoren in zeilenorientierte Blockstreifen und in ein Schachbrettmuster

6.2 Matrix-Transposition in Gitter und Hypercube

Zur $n \times n$-Matrix A sei A^T zu bestimmen mit $A^T[i,j] := A[j,i]$ für $0 \leq i,j < n$. Hierfür eignet sich ein Schachbrettmuster, realisiert durch $\sqrt{p} \times \sqrt{p}$ Prozessoren. Jeder Block der Größe $n/\sqrt{p} \times n/\sqrt{p}$ wandert zunächst abwärts bzw. aufwärts (siehe Bild 6.2), danach nach links bzw. nach rechts. An ihrem Zielprozessor wird die Teilmatrix lokal transponiert.

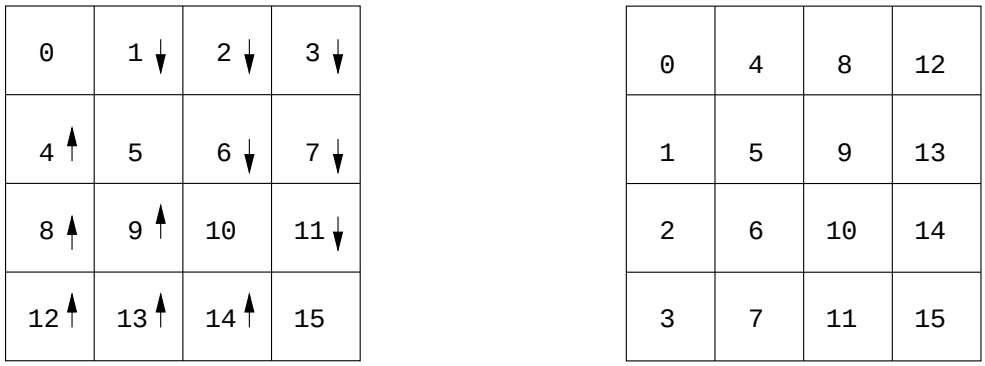


Abbildung 6.2: Verteilung der Teilmatrizen vor und nach der Transposition. Die Pfeile deuten die initiale Richtung an.

Die Laufzeit wird bestimmt von den beiden diagonal gegenüberliegenden Teilmatrizen, bei denen n^2/p Daten über eine Länge von $2\sqrt{p}$ transportiert werden müssen. Die lokale Transposition dauert n^2/p Schritte. Daraus resultieren eine Laufzeit von $O(n^2/p \cdot \sqrt{p})$ und Kosten von $O(n^2 \cdot \sqrt{p})$.

Eine Beschleunigung wird durch die rekursive Struktur der Matrixtransposition möglich.

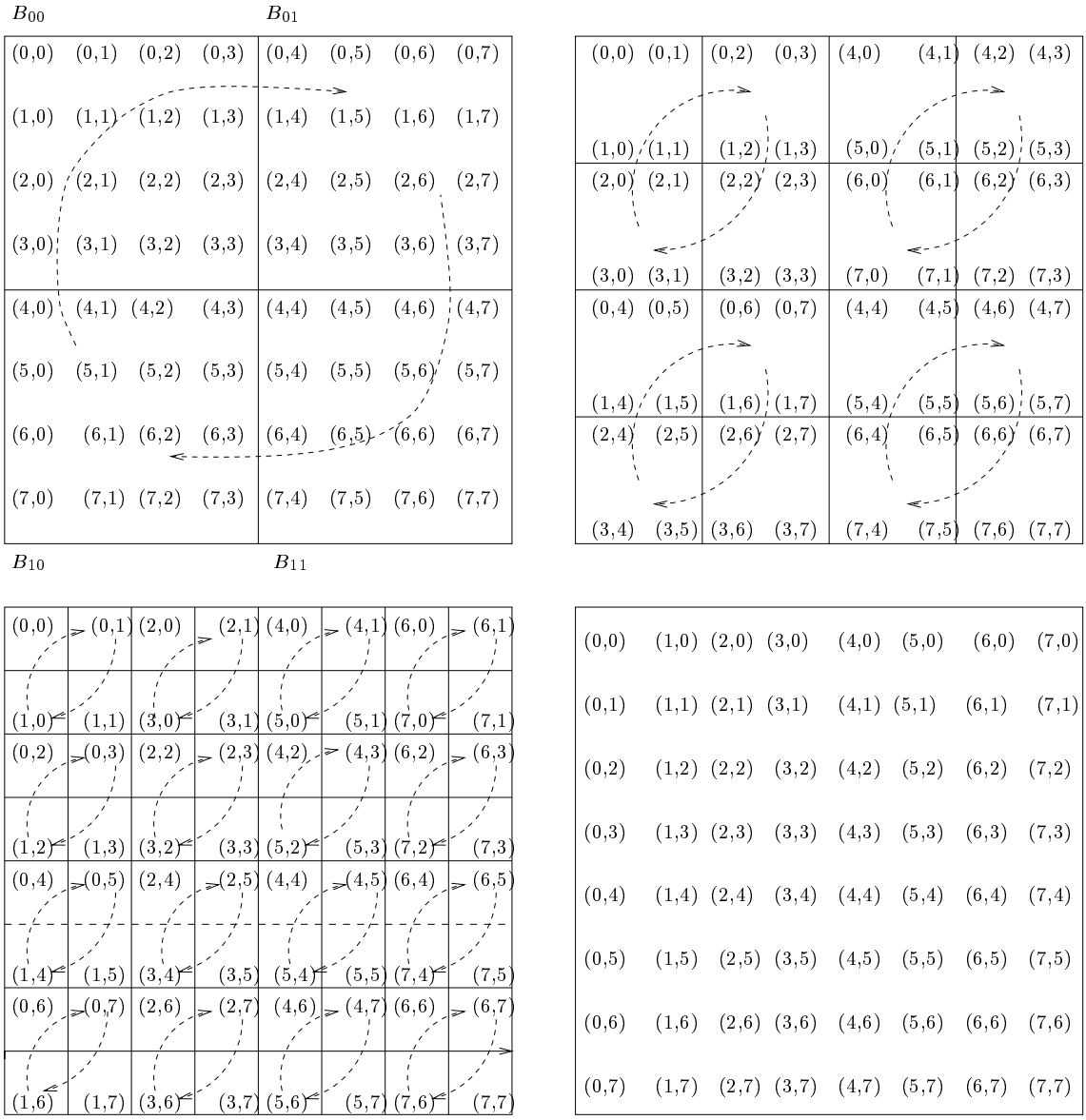


Abbildung 6.3: Rekursive Transposition einer 8×8-Matrix in 3 Phasen

Eine Implementierung auf dem Hypercube nutzt die rekursive Struktur der Matrixtransposition aus: Eine $n \times n$-Matrix A kann zunächst als eine 2×2-Matrix, bestehend aus vier $n/2 \times n/2$ Teilmatrizen $B_{00}, B_{01}, B_{10}, B_{11}$ aufgefaßt werden. Nach dem Tausch von B_{01} mit B_{10} werden alle 4 Teilmatrizen rekursiv weiter transponiert.

Sei $\sqrt{p}$ eine 2er-Potenz. Es sei jede der p Teilmatrizen gemäß ihrem laufenden Index einem Hypercubeprozessor zugeordnet. Aufgrund der Hypercubestruktur sind $B_{00}, B_{01}, B_{10}, B_{11}$ jeweils Subwürfel, deren Adressen folgende Gestalt haben:

`0..0.., 0..1., 1..0.` bzw. `1..1..`

Dadurch sind die jeweils auszutauschenden Teilmatrizen nur 2 Kanten entfernt.

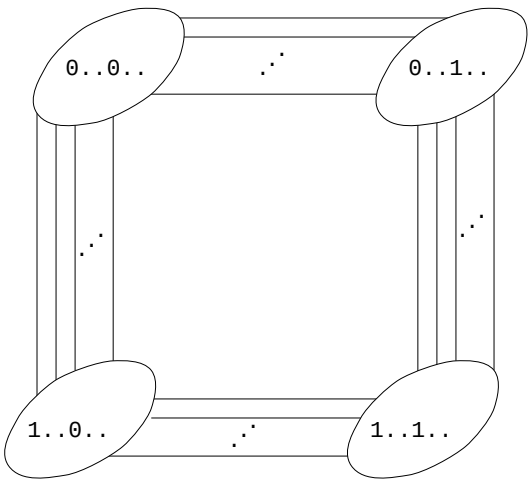


Abbildung 6.4: Partitionierung des Hypercubes in 4 Subcubes

0,4	0,5	0,6	0,7	000100	000101	000110	000111
1,4	1,5	1,6	1,7	001100	001101	001110	001111
2,4	2,5	2,6	2,7	010100	010101	010110	010111
3,4	3,5	3,6	3,7	011100	011101	011110	011111

Indizes innerhalb der rechten oberen Teilmatrix B_{01} und ihre Adressen in einem Hypercube der Dimension 6

Also entstehen $(\log \sqrt{p})$ Phasen, in denen jeweils gleichzeitig Matrizen der Größe n^2/p über 2 Links ausgetauscht werden. Die lokale Transposition benötigt $\frac{n^2}{p}$ Schritte. Daraus resultiert eine Laufzeit $T_p = O(n^2/p \cdot \log \sqrt{p})$. Der Overhead beträgt

$$T_0(W,p) = p \cdot T_p - W = O(n^2 \cdot \log \sqrt{p} - n^2) = O(n^2 \cdot \log \sqrt{p}) \ .$$

6.3 Matrix-Vektor-Multiplikation im Ring

Eine $n \times n$-Matrix A soll multipliziert werden mit einem $n \times 1$-Vektor x, d.h., gesucht ist $y = Ax$. Die Matrix A und der Vektor x seien zeilenweise in Blockstreifen verteilt. Nach einem initialen *All-to-All* zum Verteilen des Vektors kann jeder Prozessor seinen Teil des Ergebnisvektors y bestimmen.

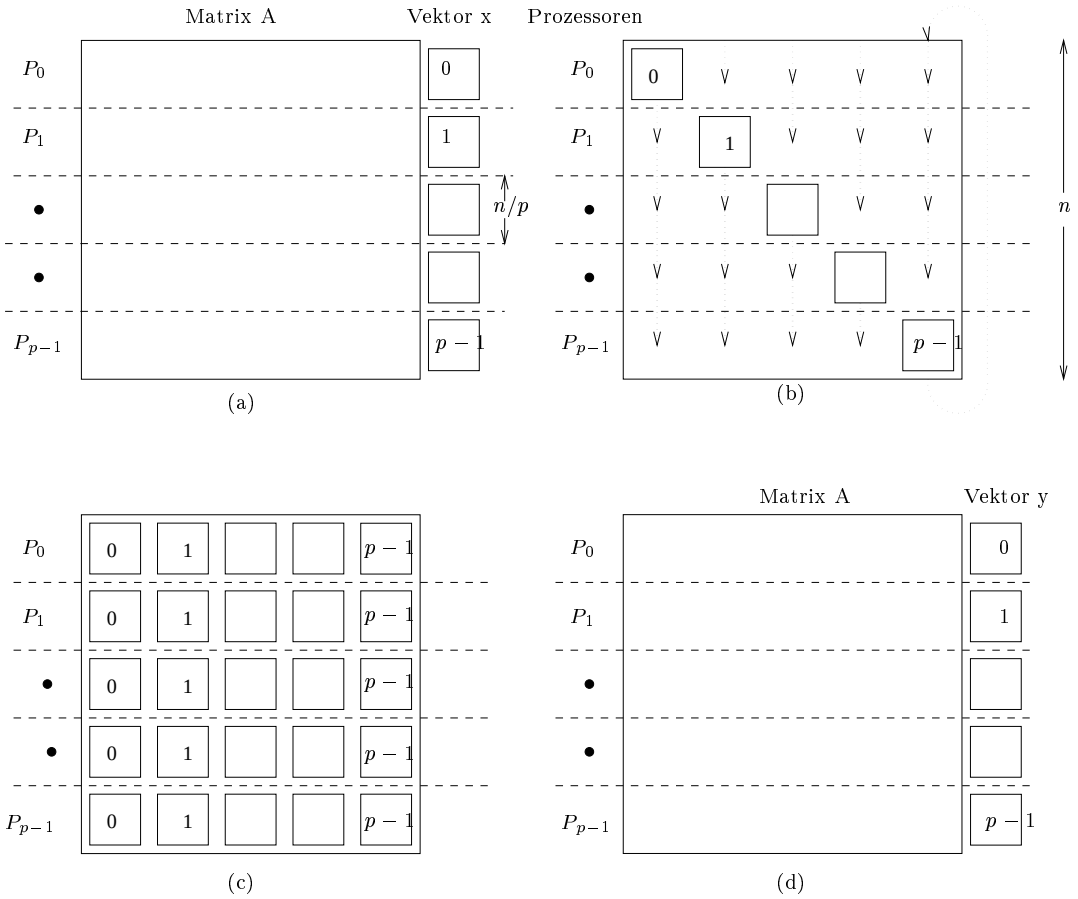


Abbildung 6.5: Matrix-Vektor-Multiplikation.
(a) Initiale Partitionierung von Matrix A und Vektor x
(b) Verteilung des ganzen Vektors x durch *All-to-All-broadcast*.
(c) Jede Komponente von x ist jedem Prozessor bekannt.
(d) Schlußverteilung von A und Ergebnisvektor y.

Im Ring benötigt *All-to-All* von Paketen der Größe n/p zwischen p Prozessoren $O(n)$ Zeit. Die Multiplikation von $\frac{n}{p}$ Zeilen von Matrix A mit dem Vektor x benötigt zusätzlich $O(\frac{n^2}{p})$. Die Gesamtzeit beträgt daher $O(n+\frac{n^2}{p}) = O(\frac{n^2}{p})$ bei p Prozessoren. Also ist der Algorithmus kostenoptimal für $p \leq n$.

6.4 Matrizenmultiplikation im Gitter

Gegeben zwei $n \times n$-Matrizen A, B. Gesucht ist die Matrix $C := A * B$ mit

$$c_{ij} := \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj} \ .$$

Eine Partition von A und B in jeweils p Teilmatrizen der Größe $n/\sqrt{p} \times n/\sqrt{p}$ erlaubt die Produktberechnung durch Multiplizieren und Addieren der korrespondierenden Teilmatrizen. Die beiden Matrizen seien gespeichert in einem quadratischen *wraparound*-Gitter mit p Prozessoren, d.h., jeder Prozessor speichert 2 Blöcke der Größe $n/\sqrt{p} \times n/\sqrt{p}$, genannt A' und B'. Zur initialen Aufstellung wird die i-te Zeile von A i-mal zyklisch nach links geshiftet, die j-te Spalte von B j-mal zyklisch nach oben geshiftet. In jeder Iteration werden dann die Teilmatrizen multipliziert, aufaddiert und zyklisch weitergeschoben, d.h., jeder Prozessor durchläuft folgendes Programm von Cannon:

```
S := 0;
FOR i := 1 TO √p DO
   S := S + A'*B';
   sende altes A' nach links;
   empfange neues A' von rechts;
   sende altes B' nach oben;
   empfange neues B' von unten;
END
```

Anschließend verfügt jeder Prozessor über eine Teilmatrix der Ergebnismatrix C.

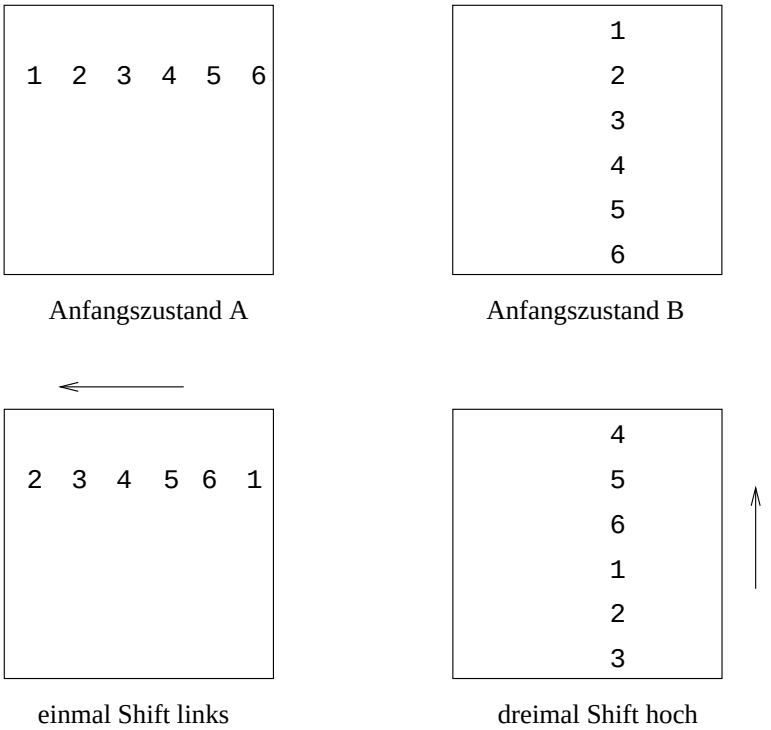


Abbildung 6.6: Initiale Aufstellung zweier 6×6-Matrizen; gezeigt ist Zeile 1 von Matrix A und Spalte 3 von Matrix B.

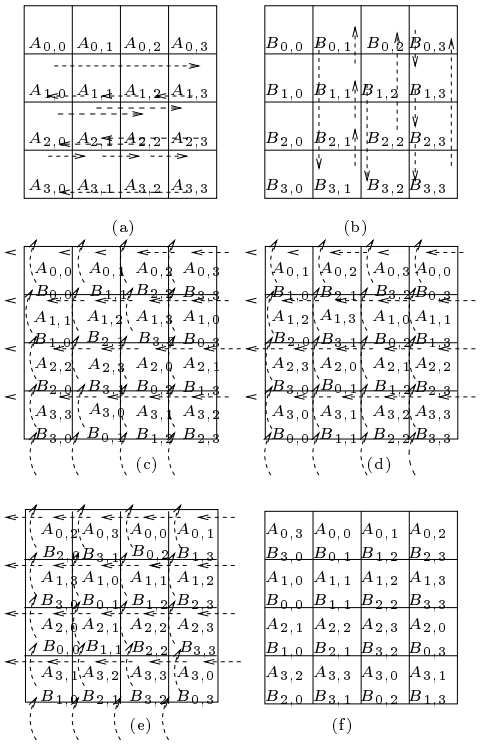
Die initiale Aufstellung erfordert $\frac{n^2}{p} \cdot \sqrt{p}$ Kommunikationszeit. In jeder der $\sqrt{p}$ Iterationen wird an Rechenzeit $(\frac{n}{\sqrt{p}})^3$ Schritte verbraucht, an Kommunikationszeit $O(\frac{n^2}{p})$. Daraus folgt als Gesamtzeit

$$O\left(\frac{n^2}{p} \cdot \sqrt{p} + \frac{n^3}{\sqrt{p}^3}\sqrt{p}\right) = O\left(\frac{n^3}{p} + \frac{n^2}{\sqrt{p}}\right) \ .$$

Somit beträgt der Overhead

$$T_0(W,p) = p \cdot T_p - W = p \cdot \frac{n^3}{p} + p \cdot \frac{n^2}{\sqrt{p}} - n^3 = n^2 \cdot \sqrt{p} \ .$$

Zu gegebener Effizienz von 50 % ergibt sich die Isoeffizienzfunktion als $n^3 = n^2 \cdot \sqrt{p}$
$\Rightarrow n = \sqrt{p}$, d.h., wächst p um den Faktor 4, so muß n um den Faktor 2 wachsen.



Matrizenmultiplikation nach Cannon in einem 4×4-*wraparound*-Gitter
(a) Initiale Verschiebungen für Matrix A
(b) Initiale Verschiebungen für Matrix B
(c) A und B in initialer Aufstellung. 1. Shift des Cannon-Algorithmus als Pfeile angedeutet.
(d) Teilmatrixpositionen nach dem 1. Shift
(e) Teilmatrixpositionen nach dem 2. Shift
(f) Teilmatrixpositionen nach dem 3. Shift

Abbildung 6.7:

6.5 Matrizenmultiplikation im Hypercube

Sei $n = 2^k$. Zwei $n \times n$-Matrizen A, B können in einem Hypercube mit $n^3 = 2^{3k}$ Prozessoren multipliziert werden nach der Idee von Dekel, Nassimi und Sahni (DNS) in Anlehnung an den CREW PRAM-Algorithmus aus Kapitel 2.5:

```
FOR ALL 0 ≤ i, j, k ≤ n − 1 DO IN PARALLEL
   P_ijk : tmp [i,j,k] := a[i,k] * b[k,j]
END
FOR ALL 0 ≤ i, j ≤ n − 1 DO IN PARALLEL
   P_ij* : c[i,j] := Σ_{k=0}^{n−1} tmp [i,j,k]
END
```

Die Bestimmung vom `tmp [i,j,k]` verursacht einen Schritt, das Aufaddieren des Vektors `tmp [i,j,*]` verursacht $\log n$ Schritte.
Zur Realisierung im Hypercube wird ein logisches $2^k \times 2^k \times 2^k$-Prozessorgitter gemäß Einbettungsidee aus Kapitel 3.3.2 mit Kantenstreckung 1 in einen Hypercube der Dimension $3k$ eingebettet. Der Hypercube hat n Ebenen, zu Beginn befinden sich die Matrizen A, B in Ebene 0, d.h., P_{ij0} speichert `a[i,j]` und `b[i,j]`.
Ziel ist es, Zeile i der Matrix A und Spalte j der Matrix B in die Prozessorvertikale P_{ij*} zu bekommen, genauer P_{ijk} soll `a[i,k]` und `b[k,j]` erhalten.

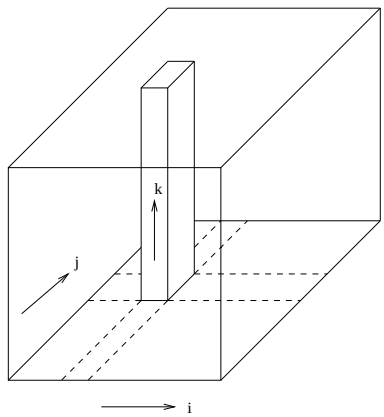


Abbildung 6.8: Vertikale im Prozessorwürfel zur Aufnahme von `a[i,*]` bzw. `b[*,j]`

Hierzu wird zunächst jede Spalte k von A auf die Ebene k kopiert, d.h., `a[i,k]` wandert von P_{ik0} nach P_{ikk}. Danach findet auf jeder Ebene k ein One-to-All Broadcast dieser Spalte statt, d.h., Prozessoren P_{i*k} erhalten Kopien von `a[i,k]` von P_{ikk}. Somit verfügt P_{ijk} über `a[i,k]`. Analog wird die Matrix B verteilt, indem jeweils die Zeilen nach oben wandern und dann ebenenweise durch Broadcast verteilt werden. Somit verfügt P_{ijk} über `b[k,j]`.

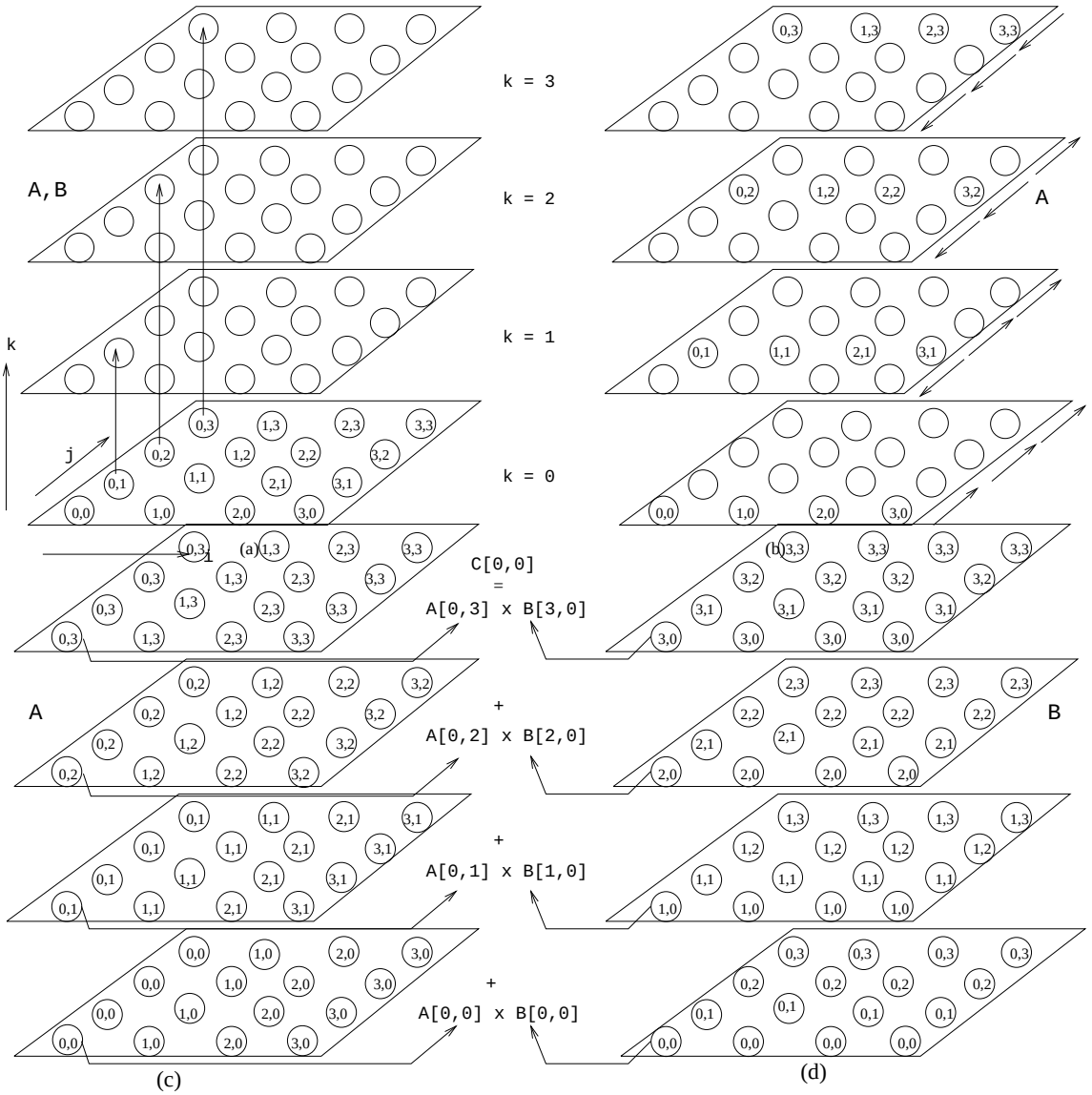


Abbildung 6.9: Kommunikationsschritte im DNS-Algorithmus für zwei 4×4-Matrizen auf 64 Prozessoren

Nach der Multiplikation `tmp[i,j,k] := a[i,k] * a[k,j]` muß jede Prozessorvertikale ihre Produkte `tmp [i,j,*]` aufaddieren und das Ergebnis nach P_{ij0} bringen.
Alle Phasen (Lift der Matrizen A, B, Broadcast in einer Ebene, Aufsummieren) finden in n-elementigen Subwürfeln des Hypercubes statt und benötigen daher $\log n$ Schritte. Die Gesamtlaufzeit für n^3 Prozessoren beträgt daher $O(\log n)$.

Kapitel 7

Lineare Gleichungssysteme

Gegeben sei eine nicht-singuläre $n \times n$-Matrix A und ein $n \times 1$-Vektor b.
Gesucht wird ein $n \times 1$-Vektor x mit $Ax = b$.

Beispiel: Das Gleichungssystem $Ax = b$ mit

$$A = \begin{pmatrix} 3 & 4 \\ 2 & 5 \end{pmatrix} \text{ und b } = \begin{pmatrix} 1 \\ -4 \end{pmatrix}$$

hat die Lösung

$$x = \begin{pmatrix} 3 \\ -2 \end{pmatrix} .$$

Es gibt
direkte Lösungsverfahren

- *Gauß-Jordan*-Elimination
- *Gauß*-Elimination
- *Cholesky*-Zerlegung

iterative Lösungsverfahren

- *Gauß-Seidel*
- *Jacobi*

Es sei a_{ij} gespeichert in `a[i,j]` mit $0 \leq i, j < n$,
es sei b_i gespeichert in `a[i,n]` mit $0 \leq i < n$.

7.1 Gauß-Jordan-Elimination auf PRAM

Idee: In Phase k wird ein geeignetes Vielfaches der Zeile k von allen anderen Zeilen subtrahiert, so daß die Spalte k (bis auf a_{kk}) zu Null wird. Ergebnis ist eine Diagonalmatrix. Der zentrale Schritt lautet daher

$$a_{ij} := a_{ij} - \frac{a_{ik}}{a_{kk}} \cdot a_{kj}$$

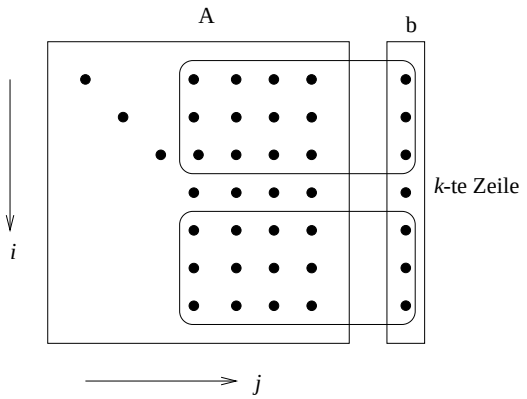


Abbildung 7.1: Matrixelemente ungleich 0 zu Beginn der k-ten Phase im Gauß-Jordan-Algorithmus. Markiert ist der aktive Teil.

Es wird eine PRAM mit $n(n+1)$ Prozessoren verwendet, wobei in Phase k Prozessor P_{ij} die Elemente $a[i,j]$ und $a[k,j]$ verknüpft.

```
FOR k := 0 TO n-1 DO
   FOR ALL 0 ≤ i < n, k ≤ j ≤ n, i ≠ k DO IN PARALLEL
         P_ij: a[i,j] := a[i,j] - (a[i,k]/a[k,k])* a[k,j]
   END
END
FOR ALL 0 ≤ i ≤ n − 1 DO IN PARALLEL
   P_in: x[i] := a[i,n]/a[i,i]
END
```

Bei $O(n^2)$ Prozessoren entstehen $O(n^3)$ Kosten, also ist der Algorithmus kostenoptimal.

7.2 Gauß-Elimination im Gitter

Idee: In Phase k wird ein geeignetes Vielfaches der Zeile k von allen Zeilen unterhalb von k subtrahiert, so daß die Spalte k unterhalb von k zu Null wird. Ergebnis ist eine obere Dreiecksmatrix D und Vektor c, deren Lösung $Dx = c$ auch $Ax = b$ löst.

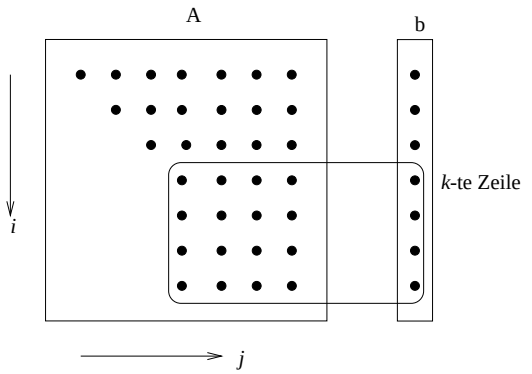


Abbildung 7.2: Matrixelemente ungleich 0 zu Beginn der k-ten Phase im Gauß-Algorithmus. Markiert ist der aktive Teil.

Die sequentielle Version lautet:

```
FOR k = 0 TO n-1 DO (* Phase k *)
   FOR j := k+1 TO n DO (* Division *)
      a[k,j] := a[k,j]/a[k,k]
   END
   a[k,k] := 1;
   FOR i := k+1 TO n-1 DO
      FOR j := k+1 TO n DO (* Elimination *)
         a[i,j] := a[i,j] - a[i,k] * a[k,j]
      END
      a[i,k] := 0;
   END
END
```

Gelöst wird $Dx = c$ durch das sukzessive Auflösen der jeweils letzten Zeile und Rückwärtseinsetzen der Lösung. Dieses Verfahren wird *Backsubstitution* genannt.

Es wird ein MC^2 verwendet, bei dem die lokale Variable des Prozessors P_{ij} mit dem Matrixelement a_{ij} initialisiert wird. Der Eliminationsschritt kann umformuliert werden als

```
a[i,j] := a[i,j] - a[i,k] * a[k,j]/a[k,k]
```

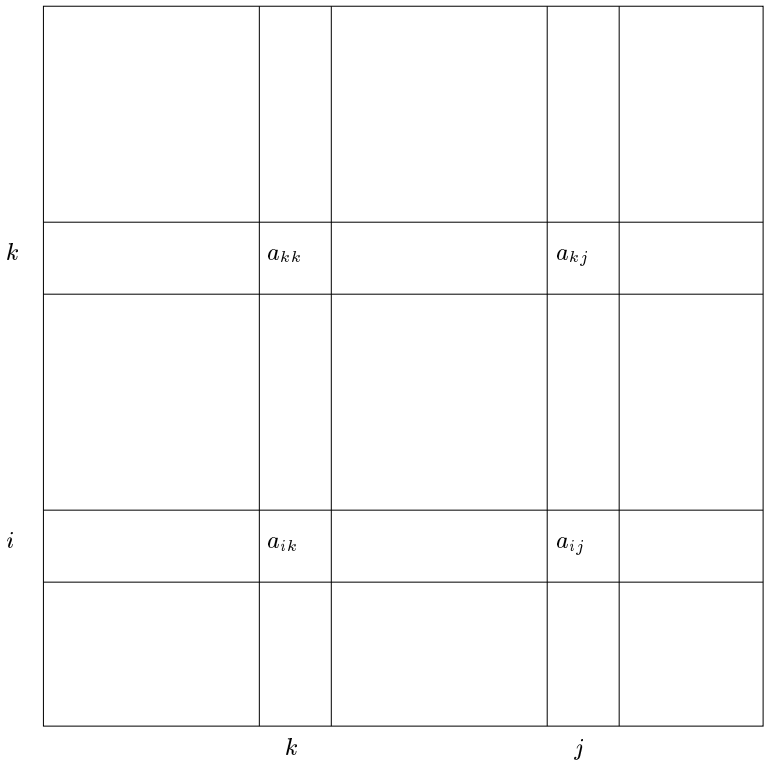


Abbildung 7.3: An der Modifikation von a_{ij} beteiligte Matrixelemente

Die k-te Phase wird gestartet durch Prozessor P_{kk}, der seinen momentanen Wert a_{kk} nach rechts schickt zu $P_{k,k+1}, P_{k,k+2}, \ldots, P_{k,n}$ und seinen Wert a_{kk} dann auf 1 setzt. Jeder Prozessor $P_{kj}, j > k$, dividiert nach Erhalt von a_{kk} sein a_{kj} durch a_{kk} und kann dann sein modifiziertes a_{kj} nach unten schicken. Prozessor P_{ij}, der von oben einen Wert b und von links einen Wert c erhalten hat, reicht diese nach unten resp. nach rechts weiter und subtrahiert das Produkt von seinem lokalen Matrixelement, d.h., er bildet

$$a_{ij} := a_{ij} - \frac{a_{ik}}{a_{kk}} \cdot a_{kj} \ .$$

Alle Phasen laufen in Pipeline-Manier überlappend, d.h., Phase $k+1$ wird von $P_{k+1,\ k+1}$ eingeleitet, sobald alle für $P_{k+1,k+1}$ bestimmten Nachrichten eingetroffen sind.

Da jede Phase $O(n)$ Schritte dauert und zwischen zwei Phasenstarts konstante Zeit liegt, beträgt die Gesamtlaufzeit $O(n)$. Bei n^2 Prozessoren entstehen Kosten von $O(n^3)$. Der Algorithmus ist daher kostenoptimal.
In Anschluß daran findet eine Backsubstitution statt.

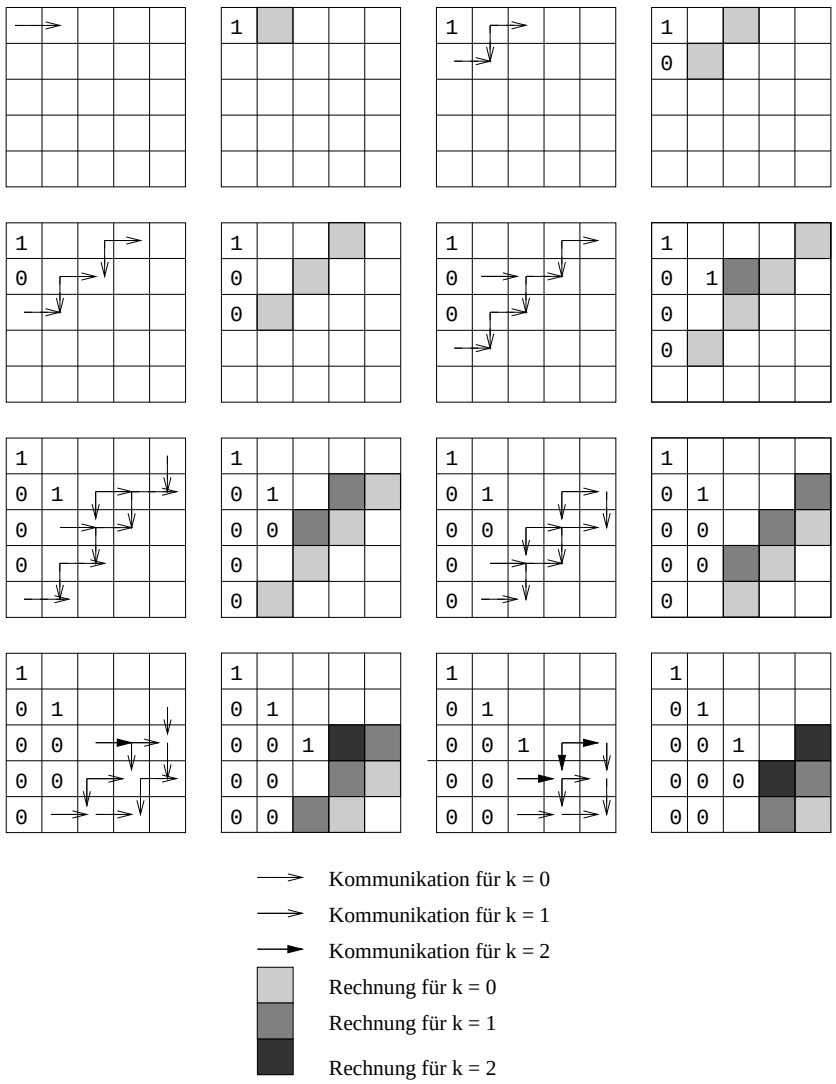


Abbildung 7.4: Pipeline Gauß-Elimination

7.3 Cholesky-Zerlegung im Ring

Sei nun A symmetrisch und positiv definit (d.h. $v^T Av > 0 \; \forall \; v \in \mathbb{R}^n, v \neq 0$).

Idee:

1. Bestimme obere Dreiecksmatrix U mit $U^T U = A$.

2. Bestimme y mit $U^T \cdot y = b$.

3. Bestimme x mit $Ux = y$.

   Dann gilt $Ax = b$, denn aus $Ux = y$ folgt $U^T Ux = U^T y$ mit $U^T U = A$ und $U^T y = b$.

zu 1.) Sei U bis zur Zeile $i-1$ bereits bestimmt. Aus

$$a_{ii} = \sum_{k=0}^{i} (u_{ki})^2 \text{ und } a_{ij} = \sum_{k=0}^{i} (u_{ik}^T \cdot u_{kj}) \text{ folgt}$$

$$u_{ii} := \sqrt{a_{ii} - \sum_{k=0}^{i-1} (u_{ki})^2}$$

$$u_{ij} := (a_{ij} - \sum_{k=0}^{i-1} (u_{ki} \cdot u_{kj}))/u_{ii}$$

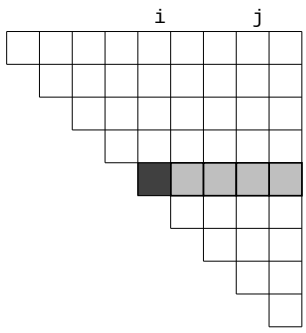


Abbildung 7.5: Zeilenweises Bestimmen der Matrix U zunächst der dunkle, dann der helle Teil

```
FOR i:= 0 TO n-1 DO (* i-te Zeile *)
   tmp := a[i,i];
   FOR k := 0 TO i-1 DO
      tmp := tmp - u[k,i]*u[k,i]
   END;
   u[i,i] := sqrt(tmp); (* Diagonalelement *)
   FOR j := i+1 TO n-1 DO
      tmp := a[i,j];
      FOR k := 0 TO i-1 DO
         tmp := tmp - u[k,i]*u[k,j];
      END
      u[i,j] := tmp/u[i,i]
   END
END
```

zu 2.)

$$\text{Aus } b_i = \sum_{j=0}^{i} u_{ij}^T \cdot y_j \text{ folgt}$$

$$y_i := (b_i - \sum_{j=0}^{i-1} u_{ji} \cdot y_j)/u_{ii}$$

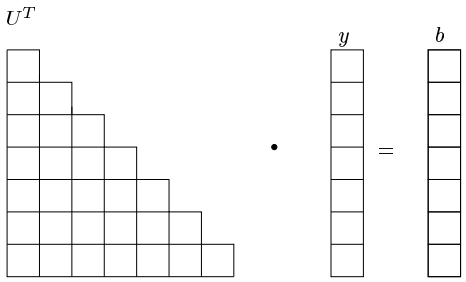


Abbildung 7.6: Forward-Substitution

```
FOR i := 0 TO n-1 DO
   tmp := b[i];
   FOR j := 0 TO i-1 DO
      tmp := tmp - u[j,i]*y[j]
   END;
   y[i] := tmp/u[i,i]
END
```

zu 3.)

$$\text{Aus } y_i = \sum_{j=i}^{n-i} u_{ij} \cdot x_j \text{ folgt}$$

$$x_i := (y_i - \sum_{j=i+1}^{n-1} u_{ij} \cdot x_j)/u_{ii}$$

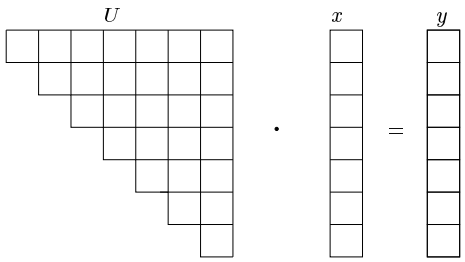


Abbildung 7.7: Backward-Substitution

```
FOR i := n-1 DOWNTO 0 DO
   tmp := y[i];
   FOR j := i + 1 TO n-1 DO
      tmp := tmp - u[i,j]*x[j]
   END;
   x[i] := tmp/u[i,i]
END
```

Zur parallelen Cholesky-Zerlegung wird ein Ring von n Prozessoren verwendet. Zu Beginn speichert Prozessor j Spalte j von Matrix A. Während der Rechnung ermittelt Prozessor j die Spalte j von Matrix U. Dabei können die Matrixelemente von A mit denen von U überschrieben werden. Für die parallele Backward-Substitution ist es erforderlich, daß Prozessor i über die i-te Zeile von U verfügt. Dies kann dadurch erreicht werden, daß während der parallelen Zerlegungsphase die entsprechenden Matrixelemente beim Durchreichen einbehalten werden (nicht im Algorithmus berücksichtigt!). Die parallele Forward-Substitution kann vereinfacht werden, indem b als zusätzliche Spalte n von A schon in der Zerlegung behandelt wird (nicht im Algorithmus berücksichtigt).

Parallele Zerlegung

```
FOR i := 0 TO n - 1 DO
   (* bestimme Zeile i von U *)
   FOR ALL i ≤ j ≤ n DO IN PARALLEL
      P_j: falls j = i: berechne u[i,i] aus a[i,i], u[*,i]
             und verschicke Spalte i = u[*,i];
         falls j > i: erhalte Spalte i;
            gib Spalte i weiter (falls j < n - 1);
            berechne u[i,j] aus a[i,j], u[*,i], u[*,j];
   END
END
```

Parallele Forward-Substitution

Prozessor P_i kennt Spalte i von U (= Zeile i von U^T).

```
FOR ALL 0 ≤ i ≤ n − 1 DO IN PARALLEL
   P_i: tmp[i] := b[i]
END
FOR j := 0 TO n-1 DO
   FOR ALL j ≤ i ≤ n − 1 DO IN PARALLEL
      P_i: falls i = j: y[i] := tmp[i]/u[i,i];
             verschicke y[i];
         falls i > j: erhalte y[j];
            reiche ggf. weiter;
            tmp[i] := tmp[i]-u[j,i]*y[j];
```

Parallele Backward-Substitution

Prozessor P_i kennt Zeile i von U.

```
FOR ALL 0 ≤ i ≤ n − 1 DO IN PARALLEL
   P_i: tmp[i] := y[i]
END
FOR j := n-1 DOWNTO 0 DO
   FOR ALL 0 ≤ i ≤ j DO IN PARALLEL
      P_i: falls i = j: x[i] := tmp[i]/u[i,i];
             verschicke x[i];
         falls i < j: erhalte x[j];
            reiche ggf. weiter;
            tmp[i] := tmp[i]-u[i,j]*x[j];
```

Der sequentielle Cholesky-Algorithmus profitiert von dünn besetzten Matrizen, die bei FEM-Verfahren auftreten. Z.B. beträgt bei einem $2D$-Problem mit $n = 50.000$ die Bandbreite etwa $\sqrt{n} = 250$).

Permutiere A so, daß alle Nicht-Null-Einträge nahe der Hauptdiagonale sind. Haben die Nicht-Null-Einträge den maximalen Abstand Δ von der Hauptdiagonale, so hat A die Bandbreite $2 \cdot \Delta$. Dann hat Matrix U mit $U^T \cdot U = A$ die Bandbreite Δ.
Also schränkt sich der Indexbereich ein:

$$u_{ij} := (a_{ij} - \sum_{k=i-\Delta+1}^{i-1} u_{ki} \cdot u_{kj})/u_{ii}$$

Zur parallelen Cholesky-Zerlegung einer dünn besetzten Matrix mit Bandbreite Δ werden $p < \Delta$ Prozessoren im Ring verwendet. Idee: Verteile die Spalten von A nach *Round-Robin* auf die Prozessoren. Rechne nur innerhalb der Skyline.

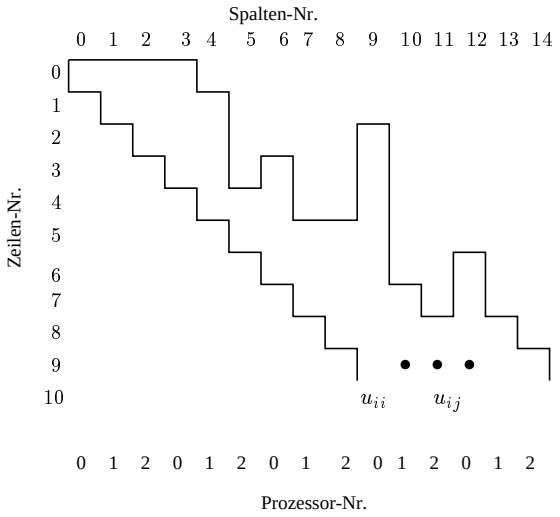


Abbildung 7.8: Cholesky-Zerlegung mit $p = 3$ Prozessoren. Markiert sind drei Elemente der 9. Zeile, die gleichzeitig bestimmt werden können.

7.4 Iterationsverfahren

Für jede Zeile i gilt:

$$\sum_{j=0}^{n-1} a_{ij} \cdot x_j = b_i$$

Für $a_{ii} \neq 0$ gilt also

$$x_i = (b_i - \sum_{j \neq i} a_{ij} \cdot x_j)/a_{ii}$$

Aus Kenntnis von $x_0, x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_{n-1}$ läßt sich x_i berechnen. Oder: Aus einer Näherung für $x_0, x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_{n-1}$ läßt sich eine Näherung für x_i bestimmen.

Sequentielles Gauß-Seidel-Iterationsverfahren

Initialisiere Näherungsvektor x (z.B. mit 0).

```
REPEAT
   diff := 0;
   FOR i := 0 TO n-1 DO
      alt := x[i];
      sum := b[i];
      FOR j := 0 TO n-1 DO
         IF i<>j THEN sum := sum - a[i,j]*x[j] END
      END;
      x[i] := sum/a[i,i];
      diff := diff + abs(alt - x[i])
   END
UNTIL diff < epsilon
```

Obacht: Durch die Implementation der FOR-Schleife gilt

$$x_i^{t+1} := (b_i - \sum_{j<i} a_{ij} \cdot x_j^{t+1} - \sum_{j>i} a_{ij} \cdot x_j^t)/a_{ii}$$

Gauß-Seidel ist inhärent sequentiell, da zur Bestimmung von x_i^{t+1} erst alle x_j^{t+1} bestimmt werden müssen mit $j < i$.

Zur Parallelisierung geeigneter ist *Jacobi*:

$$x_i^{t+1} := (b_i - \sum_{j \neq i} a_{ij} \cdot x_j^t)/a_{ii}$$

Paralleles Iterationsverfahren nach Jacobi im Ring

Prozessor P_i kennt b_i, Zeile i von A, Startwert x.

```
REPEAT
   FOR i := 0 TO n-1 DO IN PARALLEL
      sum := b[i];
      FOR j := 0 TO n-1 DO
         IF (i<>j) THEN sum := sum - a[i,j]*x[j] END
      END;
      x[i] := sum/a[i,i];
      FOR i := 0 TO n-1 DO
         IF ODD(i) THEN
            erhalte update von P(i-1) mod n;
            sende update nach P(i+1) mod n;
         ELSE
            sende update nach P(i+1) mod n;
            erhalte update von P(i-1) mod n;
         END
      END
   END
UNTIL fertig
```

Anwendung paralleler Iterationsverfahren auf Gitter

Bei der Simulation physikalischer Vorgänge ist es häufig notwendig, die auftretenden Differentialgleichungen numerisch zu lösen. Dazu wird eine Diskretisierung des betrachteten Problems durchgeführt, auf der dann die Differentialgleichungen durch einfache Näherungen ersetzt werden. Dies führt häufig zu sehr regelmäßigen Gleichungssystemen, die dann mit geeigneten Iterationsverfahren gelöst werden. Zum Beispiel führt die Bestimmung des Temperaturverlaufs in einem Wasserbad mit vorgegebenen Randwerten zu einer Diskretisierung mit 2-dimensionaler Gitterstruktur.

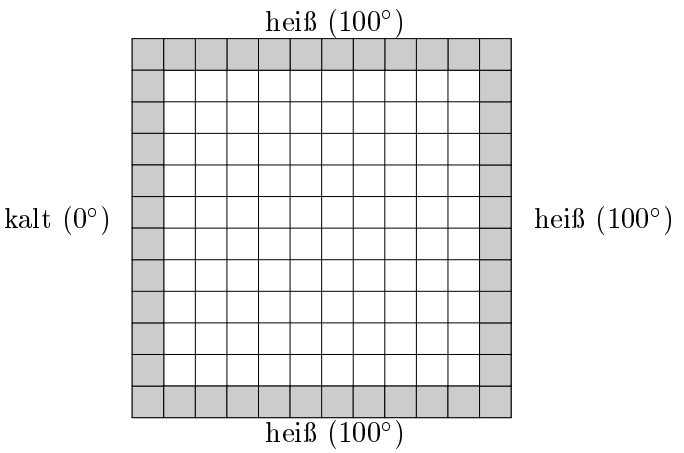


Abbildung 7.9: Vorgabe für Temperaturverlauf im Wasserbad

Die Temperatur u_{ij} wird mit Hilfe der vier Nachbartemperaturen modifiziert:

$$u_{ij}^{t+1} := \frac{u_{i-1,j}^t + u_{i,j+1}^t + u_{i+1,j}^t + u_{i,j-1}^t}{4}$$

Zur Synchronisation des Datenaustausches wird eine Partitionierung des 2D-Gitters in schwarze und weiße Prozessoren durchgeführt (Schachbrettfärbung).

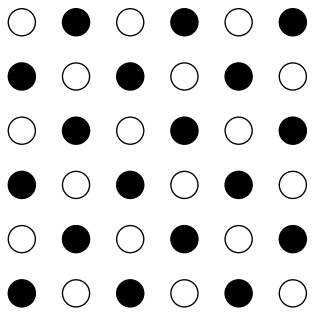


Abbildung 7.10: Schachbrettfärbung eines 6×6 Prozessorgitters

Der parallele Algorithmus lautet dann:

```
REPEAT
   FOR ALL 0 ≤ i, j ≤ n − 1 DO IN PARALLEL
      P_ij: IF weisser Prozessor
         THEN
            empfange vier schwarze Nachbarwerte;
            update u_ij;
            sende u_ij an vier schwarze Nachbarn;
            warte;
         ELSE
            sende u_ij an vier weisse Nachbarn;
            warte;
            empfange vier weisse Nachbarwerte;
            update u_ij;
         END

UNTIL fertig
```

Kapitel 8

Sortierverfahren

8.1 PRAM Sort

Gegeben n Zahlen $a_0, a_1, \ldots, a_{n-1}$; alle verschieden. Die Aufgabe besteht darin, die n Zahlen zu sortieren.

Idee: Beschreibe eine Matrix c mit dem Ergebnis aller paarweisen Vergleiche. Ermittele dann für jede Zahl, wie viele Zahlen kleiner sind.

Verwendet wird eine CREW PRAM mit n^2 Prozessoren.

```
FOR ALL 0 ≤ i,j ≤ n − 1 DO IN PARALLEL
   P_ij: IF a[i] < a[j] THEN c[i,j] := 1
         ELSE c[i,j] := 0
      END
END
```

Die Anzahl der Einsen in der j-ten Spalte der Matrix c gibt an, wie viele Zahlen kleiner sind als a_j. Also liefert die Spaltensumme

$$b_j := \sum_{i=0}^{n-1} c_{ij}$$

die Position von Zahl a_j in der sortierten Folge.

n^2 Prozessoren berechnen c_{ij} in Zeit $O(1)$.
n^2 Prozessoren berechnen b_j in Zeit $O(\log n)$.
Gesamtzeit: $O(\log n)$, Kosten $O(n^2 \cdot \log n)$.

8.2 Odd-Even-Transposition Sort

Gegeben n Zahlen $a_0, a_1, \ldots, a_{n-1}$, gespeichert in einem linearen Prozessorarray $P_0, P_1, P_2, \ldots, P_{n-1}$.

Idee: Vertausche so lange Nachbarn in falscher Relation, bis Folge sortiert ist.

```
FOR j := 1 TO n DIV 2 DO
    FOR i := 0, 2, 4, ... DO IN PARALLEL
       Compare_Exchange (a_i, a_{i+1})
    END
    FOR i := 1, 3, 5, ... DO IN PARALLEL
       Compare_Exchange (a_i, a_{i+1})
    END
END
```

```
7-3 6-5 8-1 4-2
3 7-5 6-1 8-2 4
3-5 7-1 6-2 8-4
3 5-1 7-2 6-4 8
3-1 5-2 7-4 6-8
1 3-2 5-4 7-6 8
1-2 3-4 5-6 7-8
1 2-3 4-5 6-7 8
```

Vertauschungen beim Odd-Even-Transposition Sort für 8 Zahlen

Offenbar sind für manche Eingaben mindestens n Iterationen erforderlich (z.B. wenn die größte Zahl vorne steht).

Behauptung: Nach n Iterationen ist das Array sortiert.

Beweis: Induktion über n

Sei die Behauptung bis $n-1$ bewiesen. Betrachte das Sortieren von n Zahlen. Wie Bild 8.1 zeigt, zerfällt das Schedule längs des Weges, den das Maximum durchläuft, in zwei Teile. Diese beiden Teile lassen sich zu einem Schedule für $n-1$ Zahlen zusammenfügen. Nach Induktionsvoraussetzung hat dieses Schedule die Höhe $n-1$. Also verursachen n Zahlen ein Schedule der Höhe n.

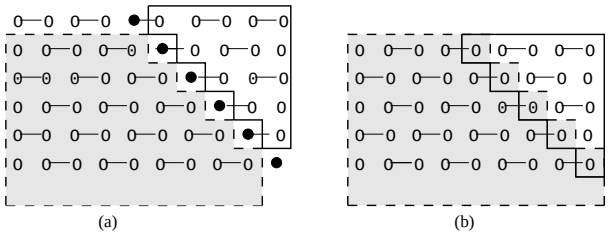


Abbildung 8.1: Wandern des Maximums • beim Odd-Even-Transposition Sort (a)
Zusammengesetzter neuer Schedule mit $n-1$ Zahlen (b)

Die Kosten betragen also $O(n^2)$, daher liegt kein kostenoptimaler Algorithmus vor.

Es seien nun $p < n$ Prozessoren vorhanden. Jeder Prozessor speichert zu Beginn n/p Zahlen. Zunächst sortiert jeder Prozessor P_i seine Folge sequentiell zu S_i. Dies dauert

$$O(\frac{n}{p} \cdot \log \frac{n}{p})$$

In jeder Iteration wird statt *compare-exchange* (a_i, a_{i+1}) jetzt *merge-and-split* (S_i, S_{i+1}) aufgerufen. Diese Prozedur tauscht zwei sortierte Listen aus, mischt sie und entfernt dann den jeweils kleineren bzw. größeren Teil. Dauer $= O(n/p)$.

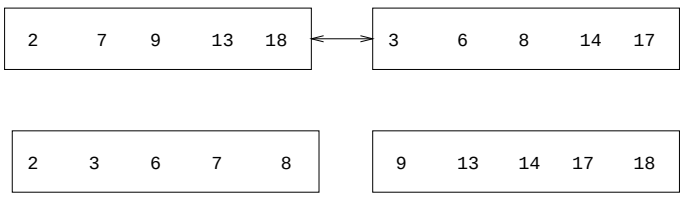


Abbildung 8.2: Vor und nach *merge-and-split*

Bei p Iterationen entsteht also als Gesamtzeit $O(\frac{n}{p} \cdot \log \frac{n}{p}) + p \cdot O(\frac{n}{p})$ und bei p Prozessoren als Kosten $O(n \cdot \log \frac{n}{p}) + O(n \cdot p)$.
Für $p < \log n$ ist dies $O(n \cdot \log n)$, somit liegt ein kostenoptimaler Algorithmus vor.

8.3 Sortiernetzwerke

Zur Formulierung paralleler Sortieralgorithmen eignen sich Sortiernetzwerke, in denen der Datenfluß mittels *Compare-Exchange*-Bausteinen gesteuert wird. Die Laufzeit wird bestimmt durch die Anzahl der Baugruppen, die hintereinander durchlaufen werden.

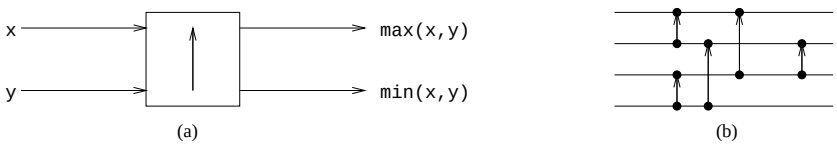


Abbildung 8.3: *Compare-Exchange*-Baustein (a)
vereinfachte Darstellung im Netzwerk (b)

Definition: Eine Folge $a_0, a_1, \ldots, a_{n-1}$ heißt *bitonisch* $\Leftrightarrow$

1. $\exists j : a_0 \leq a_1 \leq \ldots \leq a_j \geq a_{j+1} \geq a_{j+2} \geq \ldots \geq a_{n-1}$ oder
2. die Folge erfüllt Eigenschaft 1 nach zyklischem Shift

Beispiel: 3, 7, 12, 14, 13, 8, 5, 1 ist bitonisch
8, 5, 1, 3, 7, 12, 14, 13 ist bitonisch.

Satz: Sei $a_0, a_{1,}, \ldots, a_{2n-1}$ bitonisch. Dann ist auch $d_i = \min(a_i, a_{i+n}), i = 0, \ldots, n-1$ und $e_i = \max(a_i, a_{i+n}), i = 0, \ldots, n-1$ bitonisch. Außerdem gilt für $0 \leq i, j \leq n-1 : d_i \leq e_j$.
Beweisidee: siehe Bild 8.4.

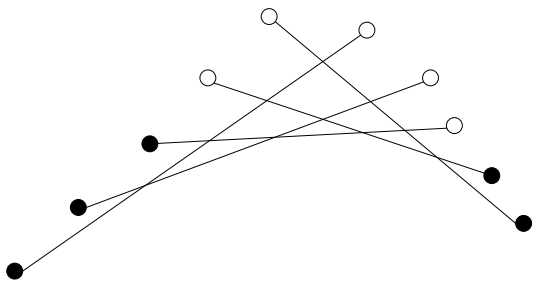
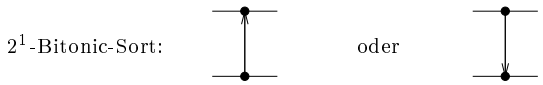


Abbildung 8.4: Minimum ($\bullet$) und Maximum ($\circ$) der Paare (a_i, a_{i+n})

Idee des bitonischen Sortierens: Eine bitonische Folge $a_0, a_1, \ldots, a_{2n-1}$ sortiert man, indem man die Folgen $d_i, i = 0, \ldots, n-1$ und $e_i, i = 0, \ldots, n-1$ bildet, diese sortiert und das Ergebnis konkateniert. Eine beliebige Folge $a_0, a_1, \ldots, a_{2n-1}$ sortiert man, indem man $a_0, \ldots, a_{n-1}$ aufsteigend sortiert, $a_n, \ldots, a_{2n-1}$ absteigend sortiert, die Ergebnisse konkateniert und als bitonische Folge sortiert.

Konstruktion eines bitonischen Sortierers

Input: bitonische Folge der Länge 2^k
Output: sortierte Folge der Länge 2^k



2^k-Bitonic-Sort teilt den Input in zwei bitonische Folgen auf und sortiert diese mit jeweils einem 2^{k-1}-Bitonic-Sort.

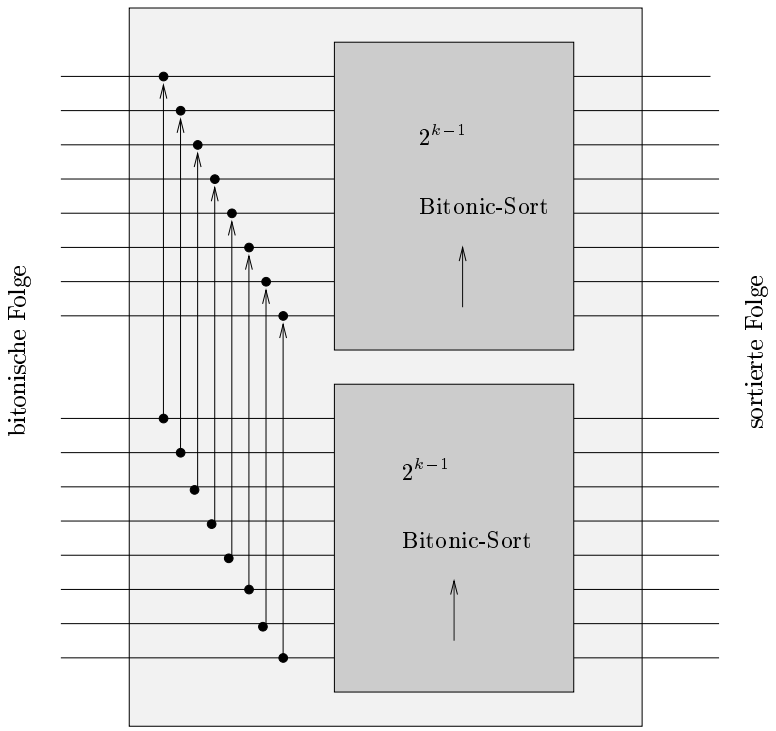
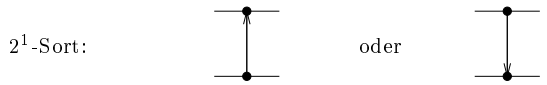


Abbildung 8.5: Rekursive Darstellung eines 2^k-Bitonic-Sort↑

Sei $g(k)$ die Laufzeit für 2^k-Bitonic-Sort
$g(1) = 1$
$g(k) = 1 + g(k-1)$
$\Rightarrow g(k) = k$

Konstruktion eines Sortierers mit Hilfe von Bitonic-Sort

Input: beliebige Folge der Länge 2^k
Output: sortierte Folge der Länge 2^k



2^k-Sort sortiert beide Input-Hälften gegenläufig jeweils mit einem 2^{k-1}-Sort und schickt das Ergebnis (welches bitonisch ist) in einen 2^k-Bitonic-Sort.

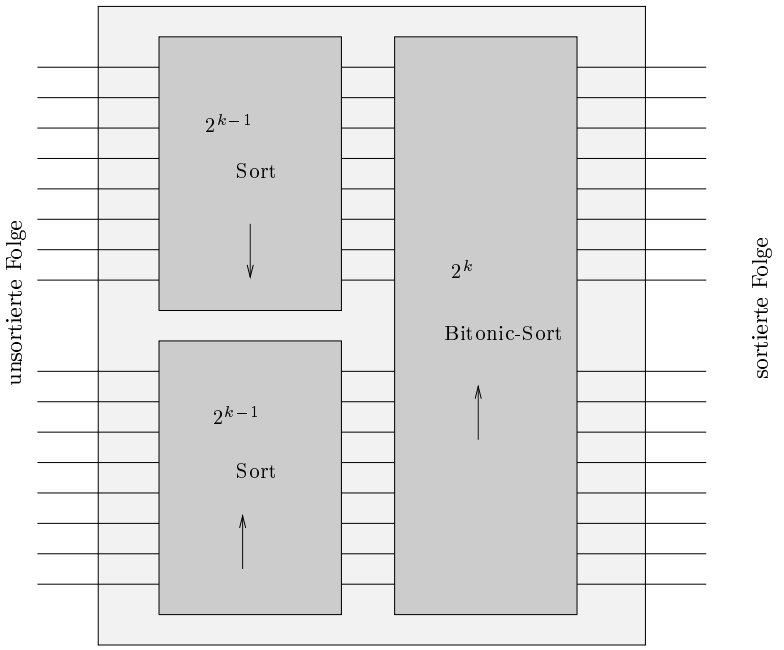


Abbildung 8.6: Rekursive Darstellung eines 2^k-Sort↑

Sei $h(n)$ die Laufzeit für 2^k-Sort
$h(1) = 1$
$h(k) = h(k-1) + g(k)$
$\Rightarrow h(k) = k \cdot (k+1)/2$
$\Rightarrow$ Laufzeit für 2^k Zahlen $= O(k^2)$
Laufzeit für n Zahlen $= O(\log^2 n)$

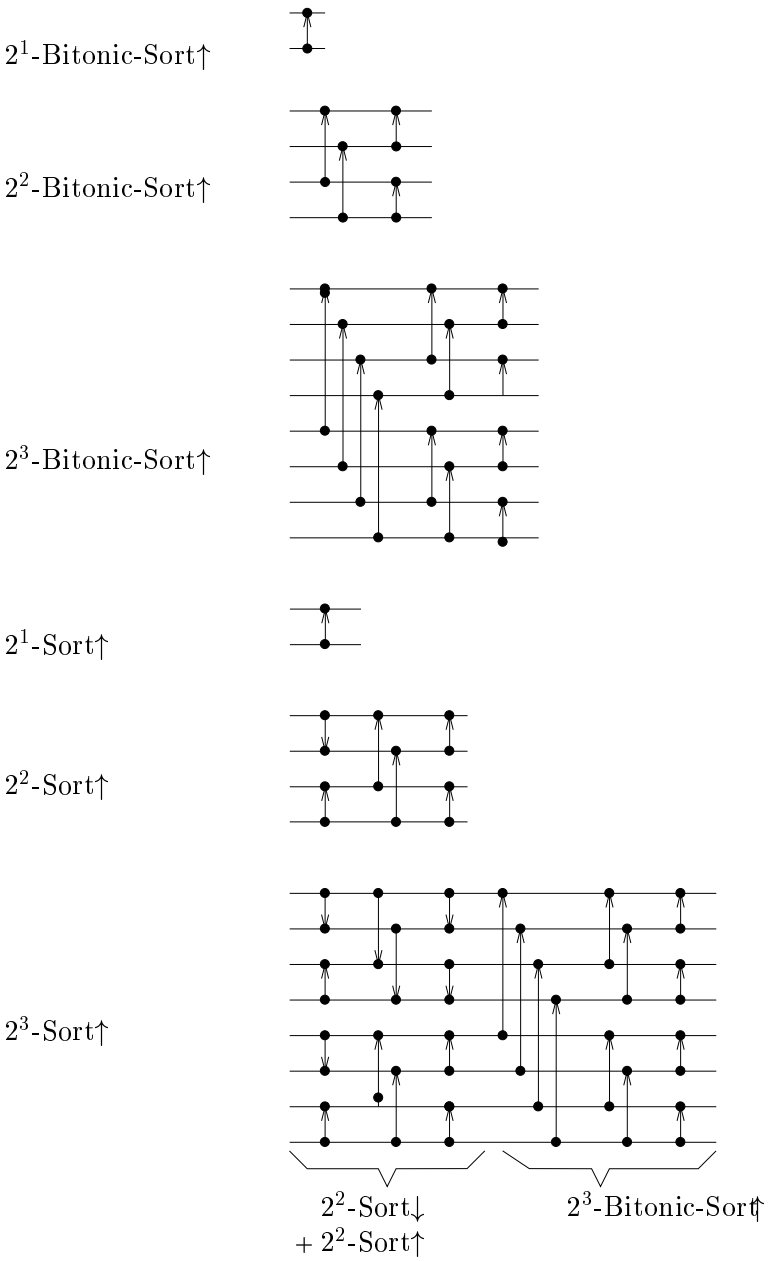


Abbildung 8.7: Explizite Darstellung eines 2^3-Sort↑ und seiner Bestandteile

8.4 Sortieren im Hypercube

Im Sortiernetzwerk bezieht sich ein *Compare-Exchange*-Baustein nur auf Linien, deren Kennungen sich um genau ein Bit unterscheiden. Also kann ein Signalverlauf der Länge t durch das Sortiernetzwerk auf dem Hypercube in Zeit $O(t)$ simuliert werden. Folgendes Programm skizziert den Hypercube-Algorithmus; wobei nicht spezifiziert wurde, welcher Prozessor nach einem Datenaustausch jeweils das Maximum und welcher das Minimum behält:

```
PROCEDURE HC_bitonic_sort(my_id,k)
BEGIN
   FOR i := 0 TO k-1 DO
      FOR j := i DOWNTO 0 DO
         Compare_exchange bzgl. Dimension j
      END
   END
END
```

Also können n Zahlen auf einem Hypercube mit n Knoten in $O(\log^2 n)$ Zeit sortiert werden. Die Kosten betragen $O(n \cdot \log^2 n)$, also liegt kein kostenoptimaler Algorithmus vor.

8.5 Sortieren im Shuffle-Exchange

Um 2^k Zahlen im Hypercube zu sortieren, führt die Prozedur `HC_bitonic_sort` aus dem vorigen Abschnitt $k \cdot (k+1)/2$ *Compare-Exchange*-Schritte durch. Diese sind strukturiert in k Gruppen, wobei in Gruppe i die Kanten der Dimension $i, i-1, \ldots, 0$ benutzt werden. Z.B. ergibt sich für $k = 6$ die folgende Sequenz von benutzten Dimensionen:

```
0 10 210 3210 43210 543210
```

Um eine Gruppe von k *Compare-Exchange*-Operationen im Hypercube längs der Dimensionen $k-1, k-2, \ldots, 0$ im Shuffle-Exchange-Netzwerk zu simulieren, werden jeweils über Shuffle-Kanten die Operanden aus Prozessor $0w$ und $1w$ in die Prozessoren $w0$ bzw. $w1$ geschickt und dort mittels der Exchange-Kante der Compare-Exchange-Schritt ausgeführt. Also werden k Schritte im Hypercube durch $2 \cdot k$ Schritte im Shuffle-Exchange-Netzwerk simuliert. Für die Simulation der Gruppen, die nicht mit der höchsten Dimension beginnen, wird zunächst unter Verwendung der Shuffle-Kanten die erforderliche Ausgangsposition hergestellt.

Also läßt sich die Idee des Bitonischen Sortierens im Shuffle-Exchange-Netzwerk wie folgt formulieren (Minimum-Maximum-Problematik ignoriert):

```
PROCEDURE SE_bitonic_sort(my_id,k)
BEGIN
   FOR i := 0 TO k-1 DO
      FOR j := k-1 DOWNTO i+1 DO
         Schicke Daten ueber Shuffle-Kante;
      END;
      FOR j := i DOWNTO 0 DO
         Schicke Daten ueber Shuffle-Kante;
         Compare-Exchange ueber Exchange-Kante
      END
   END
END
```

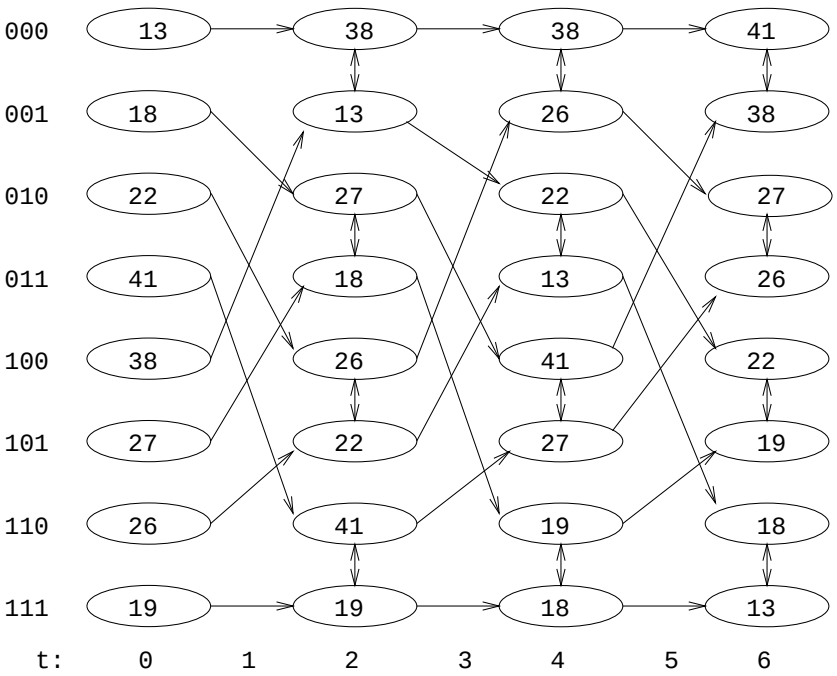


Abbildung 8.8: Bitonisches Sortieren im Shuffle-Exchange mit 8 Zahlen ($k = 3$). In der w-ten Zeile in der t-ten Spalte steht der Inhalt von Prozessor w zum Zeitpunkt t. Kanten stellen Kommunikationswege dar.

Also können n Zahlen auf einem Shuffle-Exchange-Netzwerk mit n Knoten in $O(\log^2 n)$ Zeit sortiert werden. Die Kosten betragen $O(n \cdot \log^2 n)$, also liegt kein kostenoptimaler Algorithmus vor.

8.6 Quicksort im Hypercube

Die rekursive Sortiermethode *Quicksort* läßt sich als sequentielles Programm wie folgt darstellen:

```
PROCEDURE quicksort(Menge M)
BEGIN
   waehle Pivotelement x;
   partitioniere M in M_1 und M_2 mit M_1 ≤ x < M_2;
   quicksort (M_1);
   quicksort (M_2)
END
```

Bei optimaler Wahl des Pivotelements und bei linearem Aufwand zum Partitionieren gilt für die Laufzeit $t(n)$ zum Sortieren von n Zahlen

$$t(n) \leq c \cdot n + 2 \cdot t(\frac{n}{2}) \Rightarrow t(n) = O(n \cdot \log n)$$

Würden beide rekursive Aufrufe parallel laufen, beträgt die Laufzeit

$$t(n) \leq c \cdot n + t(\frac{n}{2}) \Rightarrow t(n) = O(n) \ .$$

Bei n Prozessoren entstehen Kosten $O(n^2)$. Eine Verbesserung ist nur zu erreichen, wenn die Partitionierung beschleunigt wird.
Folgende Quicksort-Variante verwendet $p = 2^k$ Prozessoren im Hypercube. Zu Beginn hat jeder Prozessor $\frac{n}{p}$ Zahlen in seiner lokalen Menge M gespeichert. In jeder Iteration wird in jedem Subwürfel ein Pivotelement zufällig gewählt und durch one-to-all-broadcast im Subwürfel verbreitet.

```
PROCEDURE HC_quicksort(Menge M)
BEGIN
   sortiere M sequentiell;
   FOR i := 0 TO k-1 DO
      x := Pivotelement;
      partitioniere M in M_1 und M_2 mit M_1 ≤ x < M_2;
      IF BIT (my-id, i) = 0
      THEN
         verschicke M_2 ueber Dimension i;
         erhalte C ueber Dimension i;
         M := M_1∪ C durch Mischen
            ELSE
         erhalte C ueber Dimension i;
         verschicke M_1 ueber Dimension i;
         M := M_2∪ C durch Mischen
      END
   END
END
```

Analyse der Laufzeit:

Es werden p Prozessoren verwendet, die jeweils $\frac{n}{p}$ Zahlen speichern. Zu Beginn sortiert jeder Prozessor seine Liste in $O(\frac{n}{p} \cdot \log \frac{n}{p})$. Als Pivotelement wird der Median genommen, der bei einer sortierten Liste in konstanter Zeit ermittelt werden kann. Das Broadcasten des Pivotelements dauert in der i-ten Phase $k - i + 1$ Schritte. Das Aufspalten der Liste bzgl. des Pivotelements erfolgt in $O(\log \frac{n}{p})$. Der Transfer benötigt $O(\frac{n}{p})$, das Mischen ebenfalls $O(\frac{n}{p})$. Also ergibt sich für die parallele Laufzeit

$$\underbrace{O\left(\frac{n}{p} \cdot \log \frac{n}{p}\right)}_{\text{lokales Sortieren}} + \underbrace{O(\log^2 p)}_{\text{Pivot Broadcast}} + \underbrace{O\left(\frac{n}{p} \cdot \log p\right)}_{\text{Split + Transfer + Merge}}$$

Die Kosten betragen daher

$$O\left(n \cdot \log \frac{n}{p}\right) + O(p \cdot \log^2 p) + O(n \cdot \log p)$$

Für $p \leq n$ ist der erste und letzte Term $O(n \cdot \log n)$.
Für $p \leq n/\log n$ ist der zweite Term $O(n \cdot \log n)$.

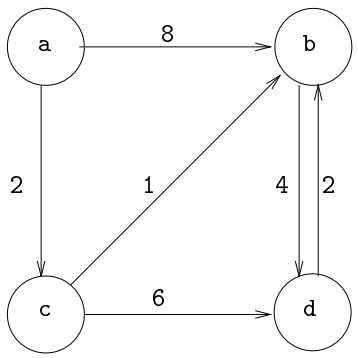
Also ist der Algorithmus für bis zu $n/\log n$ Prozessoren kostenoptimal.

Kapitel 9

Graphenalgorithmen

9.1 Definitionen

Ein *gerichteter Graph* $G = (V, E)$ besteht aus
Knotenmenge V und *Kantenmenge* $E \subseteq V \times V$



$V = \{a, b, c, d\}$
$E = \{(a, c), (a, b), (c, b), (c, d), (d, b), (b, d)\}$

Abbildung 9.1: gerichteter, gewichteter Graph

Kanten können gewichtet sein durch eine *Kostenfunktion* $c : E \to \mathbb{Z}$.
Ein *ungerichteter Graph* $G = (V, E)$ besteht aus
Knotenmenge V und *Kantenmenge* $E \subseteq P_2(V) =$ 2-elem. Teilmengen von V.

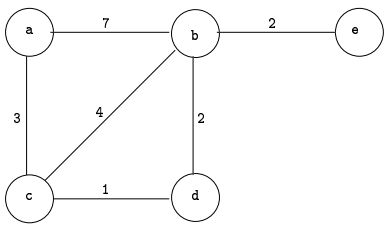


Abbildung 9.2: ungerichteter, gewichteter Graph

Mit Graphen können zwischen Objekten (≡ Knoten) binäre Beziehungen (≡ Kanten) modelliert werden.

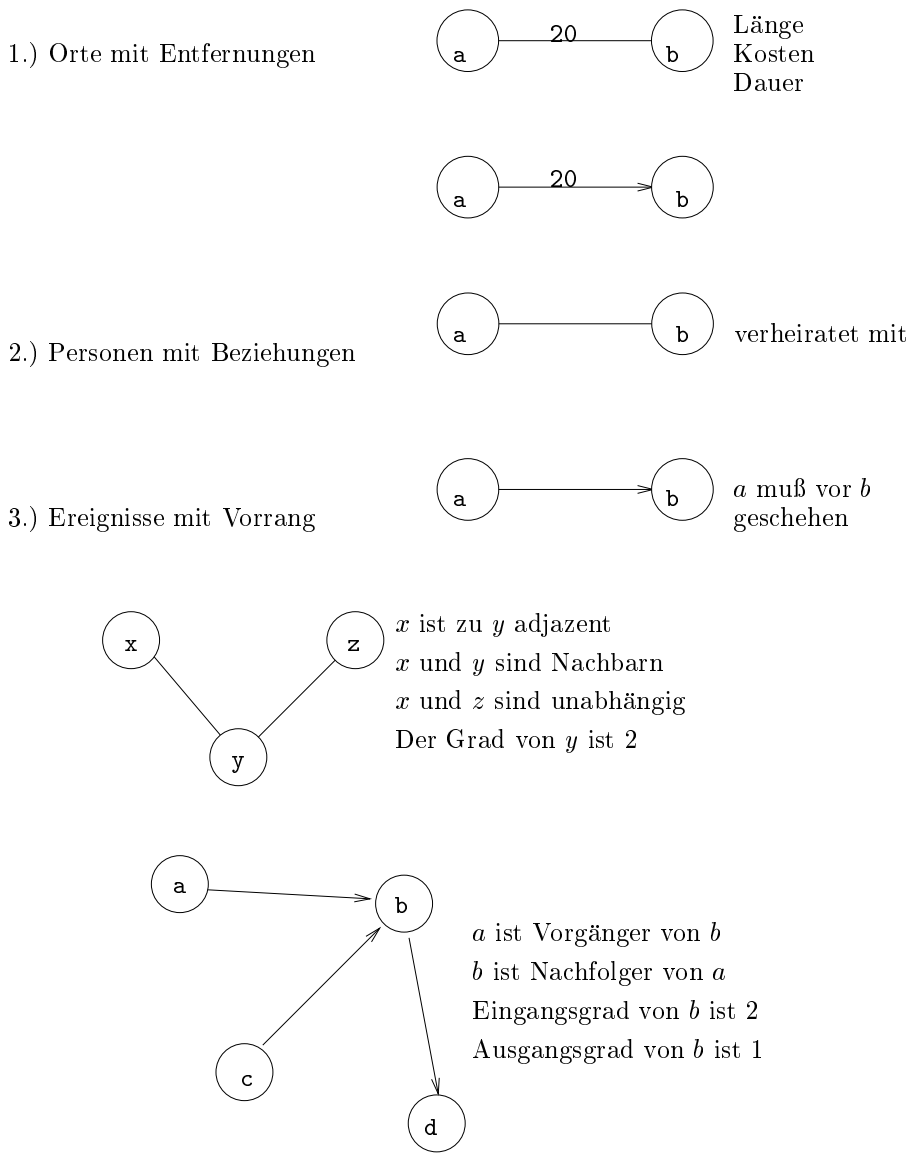


Abbildung 9.3: Modellierung und Definitionen

Ein *Weg* ist eine Folge von adjazenten Knoten.
Ein *Kreis* ist ein Weg mit Anfangsknoten = Endknoten.
Ein *Spannbaum* ist ein kreisfreier Teilgraph, bestehend aus allen Knoten.
Eine *Zusammenhangskomponente* ist ein maximaler Teilgraph, bei dem zwischen je zwei Knoten ein Weg existiert.

9.2 Implementation von Graphen

Es sei jedem Knoten eindeutig ein Index zugeordnet.

Index	Knoten
0	a
1	b
2	c
3	d

Implementation durch Adjazenzmatrix

	0	1	2	3
0	0	1	1	0
1	0	0	0	1
2	0	1	0	1
3	0	1	0	0

$$m[i,j] := \begin{cases} 1, & \text{falls } (i,j) \in E \\ 0 & \text{sonst} \end{cases}$$

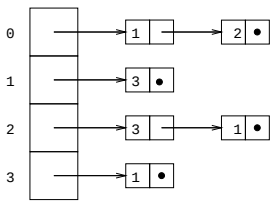
	0	1	2	3
0	0	8	2	∞
1	∞	0	∞	4
2	∞	1	0	6
3	∞	2	∞	0

$$m[i,j] := \begin{cases} c(i,j), & \text{falls } (i,j) \in E \\ 0, & \text{falls } i = j \\ \infty & \text{sonst} \end{cases}$$

Adjazenzmatrix

Platzbedarf $= O(|V|^2)$.
Direkter Zugriff auf Kante (i, j) möglich.
Kein effizientes Verarbeiten der Nachbarn eines Knotens.
Sinnvoll bei dichtbesetzten Graphen.
Sinnvoll bei Algorithmen, die wahlfreien Zugriff auf eine Kante benötigen.

Implementation durch Adjazenzlisten



i-te Liste enthält j
falls $(i, j) \in E$

Adjazenzlisten

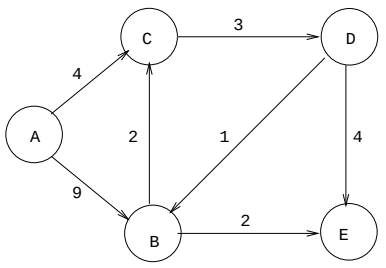
Platzbedarf $= O(|E|)$
Kein effizienter Zugriff auf gegebene Kante.
Sinnvoll bei dünn besetzten Graphen.
Sinnvoll bei Algorithmen, die, gegeben ein Knoten x, dessen Nachbarn verarbeiten müssen.

9.3 Shortest Path

Gegeben: Gerichteter Graph $G = (V, E)$, gewichtet mit Kostenfunktion.
Gesucht: Kürzester Weg von x zu allen anderen Knoten.

Idee von Moore: Initialisiere $d[i] := \infty$ für alle Knoten i; $d[x] := 0$; d bezeichnet die vorläufige Weglänge.
Es wird eine Schlange verwendet, die solche Knoten enthält, die noch zur Verbesserung beitragen können.

```
enqueue(s,x);
WHILE NOT emptyqueue(s) DO
   u := front(s); dequeue(s);
   FOREACH Nachbar v von u DO
      tmp := d[u] + c[u,v];
      IF tmp < d[v] THEN
         d[v] := tmp;
         IF v ist nicht in Schlange s
            THEN enqueue (s,v)
         END
      END
   END
END
```



A	B	C	D	E	Schlange
0	∞	∞	∞	∞	A
0	9	4	∞	∞	BC
0	9	4	∞	11	CE
0	9	4	7	11	ED
0	9	4	7	11	D
0	8	4	7	11	BE
0	8	4	7	10	E
0	8	4	7	10	

Ablauf des Moore-Algorithmus
mit Graph, Distanzvektor, Schlange und
Startknoten A.

Parallele Version des *Moore* -Algorithmus für p Prozessoren, shared memory

```
enqueue (Q,x);
WHILE NOT emptyqueue (Q) DO
   FOR ALL 0 ≤ i ≤ p − 1 DO IN PARALLEL
   P_i : hole Menge von Knoten Q_i aus der Schlange Q
      und bearbeite jedes Element aus Q_i einmal
      Ergebnis ist Schlange Q'_i
      Gliedere Q'_i in Q ein
   END
END
```

Menge Q ist gespeichert in

```
VAR Q : ARRAY [0..max-1] OF INTEGER;
```

`Q[i] > 0 =>  Q[i]` ist Knotenname
`Q[i] < 0 => -Q[i]` ist Index für Array-Element.

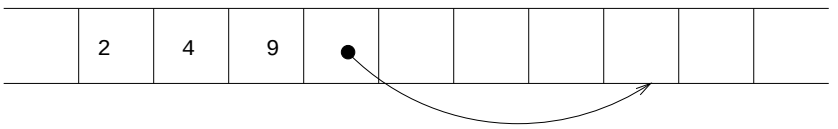


Abbildung 9.4: Knotennamen und Verweise im Array

Prozessor i bildet sein Q_i, indem er, beginnend bei Position i, jedes p-te Arrayelement aufsammelt (dabei bei negativen Einträgen dem Zeiger folgt). Q'_i wird, beginnend bei Position s_i, hintereinander abgespeichert in Q. Hinter dem letzten Knoten von Q'_i folgen p Verweise auf die ersten p Elemente der Menge Q'_{i+1}. Jeder Prozessor hat dieselbe Anzahl (± 1) von Knoten zu bearbeiten.

Obacht:
`VAR d: ARRAY [0..n-1] OF INTEGER (* vorläufige Distanzen *)`
`VAR inqueue: ARRAY [0..n-1] OF BOOLEAN (* Knoten in Schlange *)`
sind global zugreifbar. Hierdurch entsteht ein Synchronisationsproblem.

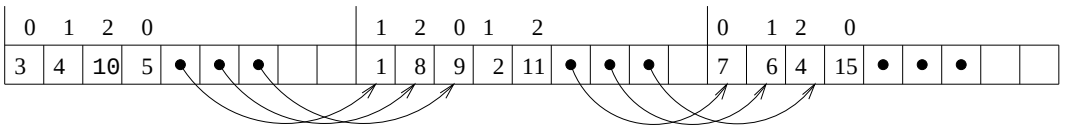


Abbildung 9.5: Schlange Q mit $Q_0 = \{3, 5, 9, 7, 15\}, Q_1 = \{\ 4, 1, 2, 6\}, \ Q_2 = \{10, 8, 11, 4\}$

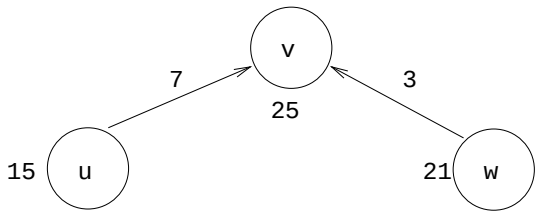


Abbildung 9.6: Synchronisationsproblem zwischen 2 Prozessoren, die die Kanten (u, v) bzw. (w, v) bearbeiten.

```
tmp = d[u] + c[u,v] = 22 => d[v] := 22
tmp = d[w] + c[w,v] = 24 => d[v] := 24
```

Das Update von v auf 22 geht verloren.

Also:
```
lock d[v];
tmp := d[u] + c [u,v];
IF tmp < d[v] THEN d[v] := tmp END;
unlock d[v];
```

Analog:
```
lock in_queue[x]; ...
unlock in_queue[x];
```

9.4 All Shortest Paths

Gegeben: Gerichteter Graph $G = (V, E)$, gewichtet mit Kostenfunktion.
Gesucht: Matrix D mit $d[i, j]$ = Länge des kürzesten Weges von i nach j.
Betrachte $D^k = d^k[i, j]$ = Länge des kürzesten Weges von i nach j, der höchstens k Kanten benutzt.

$$
\begin{array}{lcl}
d^1[i,j] & := & c[i,j] \\
d^k[i,j] & := & \begin{cases} \min\limits_m \{d^{k/2}[i,m] + d^{k/2}[m,j]\} & \text{falls } k \text{ gerade} \\ \min\limits_m \{d^{k-1}[i,m] + c[m,j]\} & \text{falls } k \text{ ungerade} \end{cases}
\end{array}
$$

Die Berechnung der Matrix D^k geschieht analog zur Matrizenmultiplikation. Statt multipliziert wird addiert, statt addiert wird minimiert.

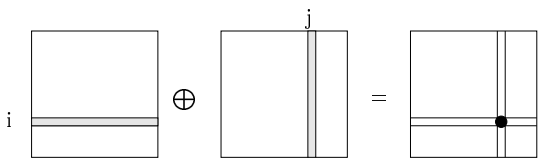


Abbildung 9.7: Verknüpfung von Zeile i mit Spalte j

Zur Berechnung von D^n sind $\log n$ Matrixverknüpfungen erforderlich (gemäß Hornerschema):
Sei Binärdarstellung von $n = n_{k-1}n_{k-2}\ldots n_0$.

```
E := 1;
FOR j := k - 1 DOWNTO 0 DO
     E := E ⊕ E;
     IF n_j = 1 THEN E := E ⊕ C END
END
```

Beispiel: $D^{13} = D^{1101} = (((C\ )^2 \cdot C\ \ )^2\ \ )^2 \cdot C$

$$
\begin{array}{lcccccc}
 & & \uparrow & & \uparrow & \uparrow & \uparrow \\
\text{nach} & & 1. & & 2. & 3. & 4.\ \text{Durchlauf}
\end{array}
$$

Also können n^3 Prozessoren auf dem Hypercube in $O(\log^2 n)$ alle kürzesten Wege eines n-elementigen Graphen bestimmen.

9.5 Minimum Spanning Tree

Gegeben: Ungerichteter Graph $G = (V, E)$, gewichtet mit Kostenfunktion.
Gesucht: Billigster Spannbaum.

Idee von Kruskal:
Lasse einen Wald wachsen mit der jeweils billigsten, zulässigen Kante.

```
Initialisiere Wald mit n isolierten Knoten;
initialisiere Heap mit allen Kanten gemäß ihrer Kosten.

REPEAT
   entferne billigste Kante e aus Heap;
   falls e keinen Kreis verursacht
      dann fuege e in Wald ein.
UNTIL  Wald besteht aus einem Baum
```

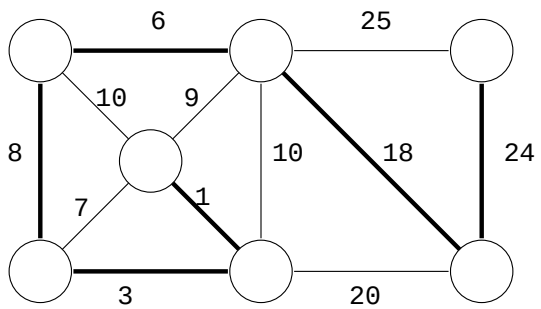


Abbildung 9.8: Gewichteter Graph mit Minimum Spanning Tree

Testen der Endpunkte und Vereinigen der Teilbäume werden mit der *Union-Find-Prozedur* in $O(1)$ gelöst. Die jeweils billigste Kante liefert ein Heap in $O(\log |E|)$.
Also benötigt ein Prozessor $O(|E| \cdot \log |E|)$.
Unter Verwendung des *Pipeline-Heap-Algorithmus* benötigen $\log |E|$ Prozessoren $O(|E|)$.

Pipeline-Heap-Algorithmus

Ziel: $\log m$ Prozessoren entfernen in konstanter Zeit das kleinste Element aus einem Heap mit m Elementen.

Idee: Prozessor P_0 entfernt in jedem zweiten Takt das Wurzelelement. Prozessor $P_i, 1 \leq i \leq \log m$, füllt das Loch in Ebene $i-1$ mit dem zuständigen Sohn aus Ebene i und vermerkt die Position des neuen Lochs in `loch[i]`. Löcher der letzten Ebene werden mit ∞ gefüllt.

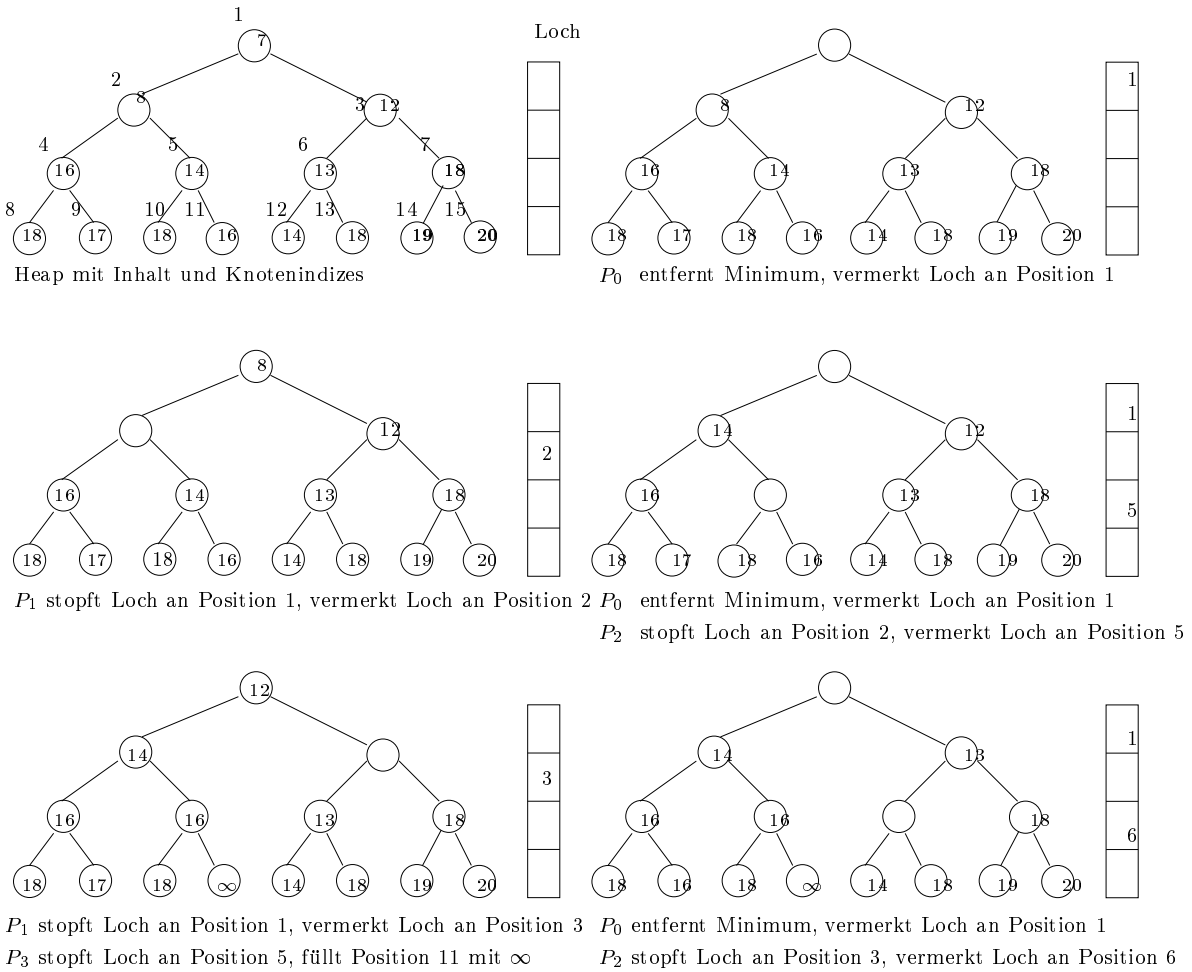


Abbildung 9.9: Pipeline-Heap-Algorithmus

9.6 Zusammenhangskomponente

Gegeben: Ungerichteter, ungewichteter Graph $G = (V, E)$
Gesucht: Zusammenhangskomponenten, d.h. `zhk[i] = j`, falls Knoten i sich in der j-ten Zusammenhangskomponente befindet.

1. Möglichkeit: Berechne transitive Hülle.

`huell[i,j]= 1` $\Leftrightarrow$ es gibt Weg von i nach j.
Sei A die (boole'sche) Adjazenzmatrix.
Dann bilde $A^j = A^{j-1} \oplus A \cong$ Wege der Länge j.
Läßt sich abkürzen durch $A, A^2, A^4, A^8, \ldots$.
$\Rightarrow$ $\log n$ boole'sche Matrixmultiplikationen
$\Rightarrow$ $\log n$ Schritte für eine CRCW-PRAM mit n^3 Prozessoren.

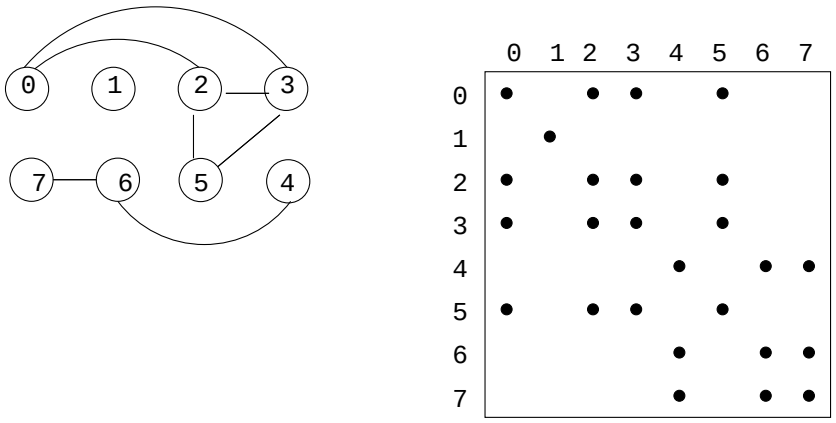


Abbildung 9.10: Graph und seine transitive Hülle

Liegt der erste Eintrag von Zeile i in Spalte j, so gilt `zhk[i] = j`.
Kosten: $O(n^3 \cdot \log n)$.

2. Möglichkeit: Tiefensuche

Partitioniere die Adjazenzmatrix in p Streifen.
Jeder Prozessor berechnet einen Spannwald durch Tiefensuche. Anschließend werden die Spannwälder mit UNION-FIND ineinandergemischt. Dazu wird jede Kante (x, y) vom sendenden Prozessor beim empfangenden Prozessor daraufhin getestet, ob x und y in derselben ZHK liegen. Falls ja, wird (x, y) ignoriert, falls nein, werden $ZHK(x)$ und $ZHK(y)$ verbunden. Da jeder Wald höchstens n Kanten enthält, benötigt das Mischen $O(n)$. Im Hypercube mit p Prozessoren entstehen nach der initialen Tiefensuche mit Zeit $O(\frac{n^2}{p})$ anschließend $\log p$ Mischphasen der Zeit $O(n)$. Also beträgt die Gesamtzeit $O(n^2/p) + O(n \cdot \log p)$, die Kosten betragen $O(n^2) + O(p \cdot \log p \cdot n)$. Für $p < n/\log n$ ist der Algorithmus kostenoptimal.

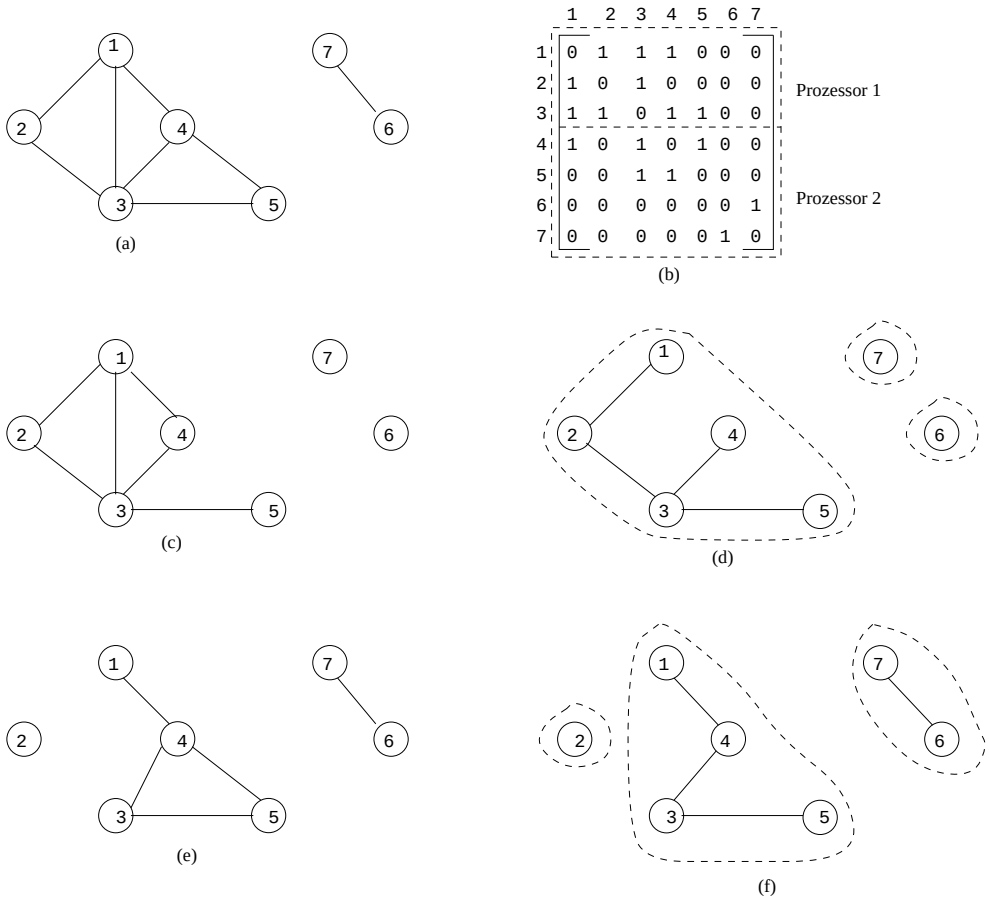


Abbildung 9.11:

Graph (a)	Partitionierung der Adjazenzmatrix (b)
Teilgraph für P_1 (c)	Spannwald berechnet von P_1 (d)
Teilgraph für P_2 (e)	Spannwald berechnet von P_2 (f)

3. Möglichkeit: Verschmelzen von Superknoten

Während des Ablaufs existiert eine Partition der Knoten in vorläufige Zusammenhangskomponenten. Jede vorläufige Zusammenhangskomponente wird repräsentiert durch den an ihr beteiligten Knoten mit der kleinsten Nummer, genannt *Superknoten*. Dieser Knoten ist Vater für alle Knoten der Zusammenhangskomponente, einschließlich für sich selbst. In jeder Iteration sucht sich jede unfertige Zusammenhangskomponente einen Partner und vereinigt sich mit diesem. Zu Beginn ist jeder Knoten sein eigener Vater und somit Superknoten. Jede Iteration hat drei Phasen:

Phase 1: Jeder Knoten sucht sich als *Freund* den kleinsten benachbarten Superknoten, d.h. von den Vätern seiner Nachbarn den kleinsten.
(Dabei wird nach Möglichkeit ein anderer Superknoten gewählt.)

Phase 2: Jeder Superknoten sucht sich als neuen Vater den kleinsten Freund seiner Söhne.
(Dabei wird nach Möglichkeit ein anderer Superknoten gewählt.)

Phase 3: Jeder Knoten sucht sich als neuen Vater das Minimum seiner Vorfahren.

Es wird eine CREW-PRAM mit n^2 Prozessoren verwendet. Da sich in jeder Iteration die Anzahl der Zusammenhangskomponenten mindestens halbiert, entstehen höchstens $\log(n)$ Iterationen. Jede Iteration benötigt:

Aufwand für Phase 1:

Je n Prozessoren bearbeiten Knoten x.

Für alle Knoten, die x zum Nachbarn haben, wird über ihren Vater minimiert.

$\Rightarrow O(\log n)$

Aufwand für Phase 2:

Je n Prozessoren bearbeiten Superknoten x.

Für alle Knoten, die x zum Vater haben, wird über ihren Freund minimiert.

$\Rightarrow O(\log n)$

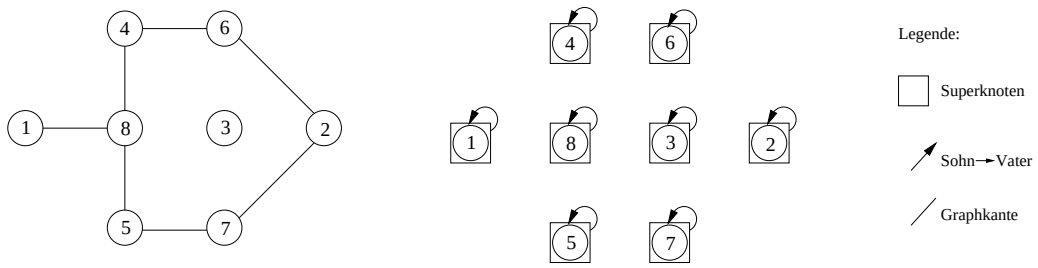
Aufwand für Phase 3:

n Prozessoren ersetzen $\log n$ mal bei jedem Knoten den Vater durch den Großvater.

$\Rightarrow O(\log n)$

Die Gesamtlaufzeit beträgt daher $O(\log^2 n)$, die Kosten $O(n^2 \cdot \log^2 n)$.

a)

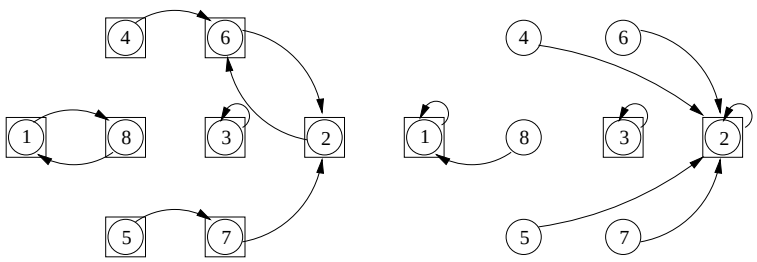


Der Graph

initiale Zusammenhangskomponenten

b)

Knoten	1	2	3	4	5	6	7	8
Freund des Knotens	8	6	3	6	7	2	2	1



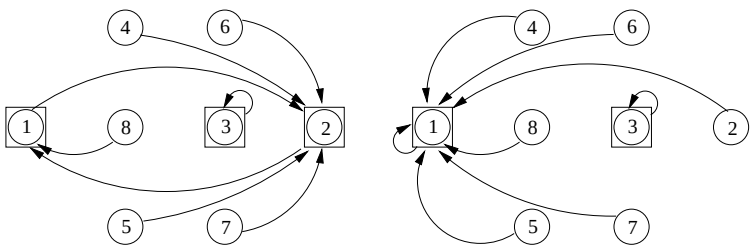
nach Phase 1

nach Phase 2

nach Phase 3

c)

Knoten	1	2	3	4	5	6	7	8
Freund des Knotens	1	2	3	1	1	2	2	2



nach Phase 1

nach Phase 2

nach Phase 3

Abbildung 9.12:
a) Ausgangsgraph
b) 1. Iteration
c) 2. Iteration

Kapitel 10

Kombinatorische Optimierung

10.1 Definitionen

Ein kombinatorisches Optimierungsproblem kann als Tupel $< S, f >$ ausgedrückt werden. S ist eine endliche oder abzählbare Menge von zulässigen Lösungen, die gewissen Randbedingungen genügen. Die Kostenfunktion $f : S \to \mathbb{R}$ bewertet die zulässigen Lösungen. Ziel ist es, eine Lösung x_{opt} zu finden mit

$$f(x_{opt}) \leq f(x) \text{ für alle } x \in S \ .$$

Beispiel (0/1-Integer-Linear-Programming): Gegeben: $m \times n$-Matrix A, $m \times 1$-Vektor b, $n \times 1$-Vektor c.
Gesucht ist $n \times 1$-Vektor $\overline{x} \in \{0,1\}^n$ mit $A\overline{x} \geq b$, wobei $f(\overline{x}) = c^T\overline{x}$ zu minimieren ist.
S ist die Menge aller 0/1-Vektoren $\overline{x}$, die $A\overline{x} \geq b$ erfüllen. f ist die Funktion $c^T\overline{x}$.

Beispiel für eine 0/1-Integer-Linear-Programming Instanz:

$$A = \begin{bmatrix} 5 & 2 & 1 & 2 \\ 1 & -1 & -1 & 2 \\ 3 & 1 & 1 & 3 \end{bmatrix}, \ b = \begin{bmatrix} 8 \\ 2 \\ 5 \end{bmatrix}, \ c = \begin{bmatrix} 2 \\ 1 \\ -1 \\ -2 \end{bmatrix}$$

Daraus ergeben sich die Randbedingungen

$$\begin{array}{rcrcrcrcl} 5x_1 & + & 2x_2 & + & x_3 & + & 2x_4 & \geq & 8 \\ x_1 & - & x_2 & - & x_3 & + & 2x_4 & \geq & 2 \\ 3x_1 & + & x_2 & + & x_3 & + & 3x_4 & \geq & 5 \end{array}$$

Zu minimieren ist

$$f(x) \ = \ 2x_1 \ + \ x_2 \ - \ x_3 \ - \ 2x_4 \ .$$

Beispiel (8-Puzzle-Problem): Gegeben ist ein 3×3-Feld mit 8 beweglichen Plättchen, numeriert von 1 bis 8. Durch eine Folge von Verschiebeoperationen soll die Startkonfiguration in eine Zielkonfiguration überführt werden. S ist die Menge aller Zugsequenzen, die vom Start zum Ziel führen. Die Kostenfunktion f ordnet einer Sequenz die Anzahl der beteiligten Züge zu.

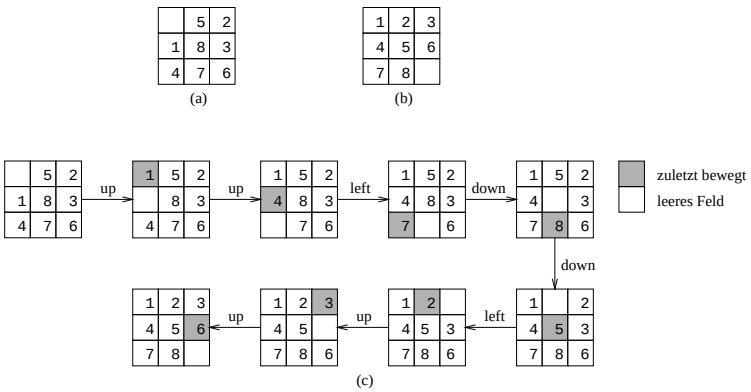


Abbildung 10.1: 8-Puzzle-Instanz
Startkonfiguration (a)
Zielkonfiguration (b)
Zugfolge (c)

Üblicherweise ist die Menge S so groß, daß sie nicht vollständig durchlaufen werden kann. Man formuliert daher das kombinatorische Optimierungsproblem als Suche in einem kantengewichteten Graphen, in dem ein kostengünstiger Weg von einem Startknoten zu einem von mehreren Zielknoten ermittelt werden muß. Der Graph heißt *Zustandsraum*, seine Knoten heißen *Zustände*. Knoten ohne Nachfolger heißen *Terminalknoten*. Knoten mit Nachfolgern heißen *Nonterminalknoten*.

Beim 8-Puzzle-Problem bildet die Startkonfiguration den Startknoten und die Zielkonfiguration den einzigen Zielknoten. Wird der Suchraum baumartig aufgespannt, so tritt der Zielknoten mehrfach auf. Die Kanten zwischen den Zuständen entsprechen den möglichen Zügen, sie sind bewertet mit 1.

Beispiel: 0/1**-Integer-Linear-Programming**

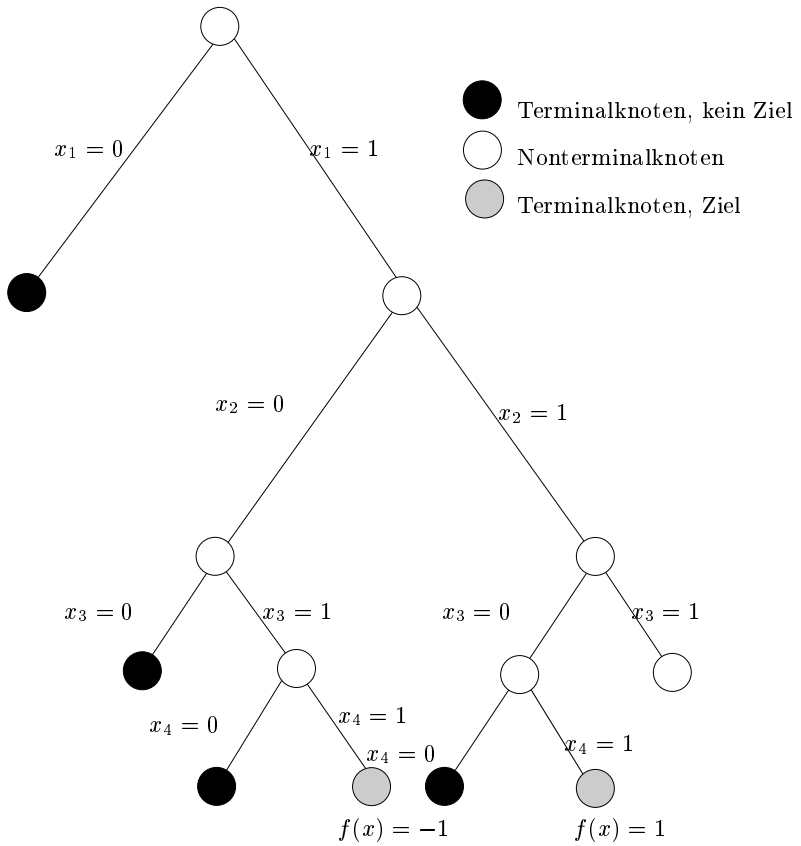


Abbildung 10.2: Zustandsraum für 0/1 Integer-Linear-Programming-Instanz

Das 0/1-Integer-Linear-Programming-Problem läßt sich als Wegsuche im Zustandsraum wie folgt formulieren: Im Startknoten sind alle Variablen noch unbesetzt. Jeder Nonterminalknoten hat zwei Söhne, in dem eine noch nicht fixierte Variable alternativ auf 0 oder 1 gesetzt wird. Ein Knoten mit mindestens einer freien Variablen und der Eigenschaft

$$\left(\sum_{x_j \text{ ist frei}} \max\{A[i,j], 0\} + \sum_{x_j \text{ ist fixiert}} A[i,j] \cdot x_j \right) \geq b_i, i = 1, \ldots, m$$

ist ein Nonterminalknoten, da durch weitere Fixierung noch die Möglichkeit besteht, die Randbedingung einzuhalten. Die Kanten für die Fixierung der Variablen x_i mit 1 wird mit c_i bewertet, alle anderen Kanten mit 0. Die Bewertung der Zielknoten ergibt sich aus der Summe der verwendeten Kanten.

Bei einigen Problemen ist es für Nonterminalknoten möglich, die Kosten zum Erreichen eines Zielknotens abzuschätzen. Seien $g(x)$ die Kosten, um den Zustand x vom Startknoten zu erreichen. Sei $h(x)$ eine heuristische Schätzung für die Kosten, um von x aus einen Zielknoten zu erreichen. Ist $h(x)$ eine untere Schranke, so wird h *zulässig* genannt. Die Funktion

$$l(x) := g(x) + h(x)$$

ist eine untere Schranke für jede Lösung, die durch Erweiterung des Wegs vom Startknoten über den Zwischenknoten x entsteht.

Für das 8-Puzzle-Problem ergibt sich eine zulässige heuristische Schätzung wie folgt: Für zwei Feldpositionen (x_1, y_1) und (x_2, y_2) sei die *Manhattan*-Distanz

$$|x_1 - x_2| + |y_1 - y_2| \ .$$

Für zwei Puzzlezustände ist

> $h(x) =$ Summe der Manhattan-Distanzen zwischen korrespondierenden Positionen aller Plättchen

eine untere Schranke für die Zahl der Verschiebeoperationen.

10.2 Sequentielles Suchen

Die Organisation der Suche hängt davon ab, ob der Zustandsraum einen Graphen bildet oder einen Baum. Beim Baum kann ein Zustand nur über einen Weg erreicht werden (z.B. 0/1-Integer-Linear-Programming), beim Graphen gibt es mehrere Wege zu einem Zustand, und es muß überprüft werden, ob der Zustand bereits erzeugt wurde.

Backtracking ist eine Tiefensuche, die bei der ersten zulässigen Lösung endet. Bei geordnetem Backtracking wird die Reihenfolge beim Besuchen der Söhne eines Knotens durch eine Heuristik bestimmt.

Depth-First Branch-&-Bound ist eine Tiefensuche, die den Zustandsraum abläuft und dabei aufgrund einer Schätzung solche Teile ausläßt, die die momentan vorhandene Lösung nicht verbessern können.

Iterative Deepening ist eine tiefenbeschränkte Tiefensuche, bei der die maximale Tiefe schrittweise erhöht wird. D.h., wurde innerhalb der Suchtiefe k keine zulässige Lösung gefunden, so wird eine komplett neue Suche gestartet mit einer größeren Suchtiefe, z.B. $k + 1$. Auf diese Weise wird eine Lösung mit den wenigsten Kanten gefunden, aber nicht notwendigerweise mit dem billigsten Weg.

Iterative Deepening A^* (IDA^*) benutzt die l-Werte der Knoten (d.h. $g(x) + h(x)$), um die Suche zu begrenzen. Es wird eine Tiefensuche durchgeführt mit einer vorgegebenen Kostenschranke b. Falls $l(x) > b$, so wird nicht weiter expandiert. Wird keine Lösung innerhalb der momentanen Kostenschranke gefunden, wird eine neue Suche mit einer größeren Kostenschranke gestartet. Die erste Kostenschranke ist $l(s)$ mit s = Startknoten. Wegen $g(s) = 0$ folgt $l(s) = h(s)$. Das Minimum der l-Werte der erzeugten, aber wegen der Kostenschranke nicht weiter verfolgten Knoten aus Suche i wird zur Kostenschranke für Suche $i + 1$. Falls h zulässig ist, so findet IDA^* das Optimum.

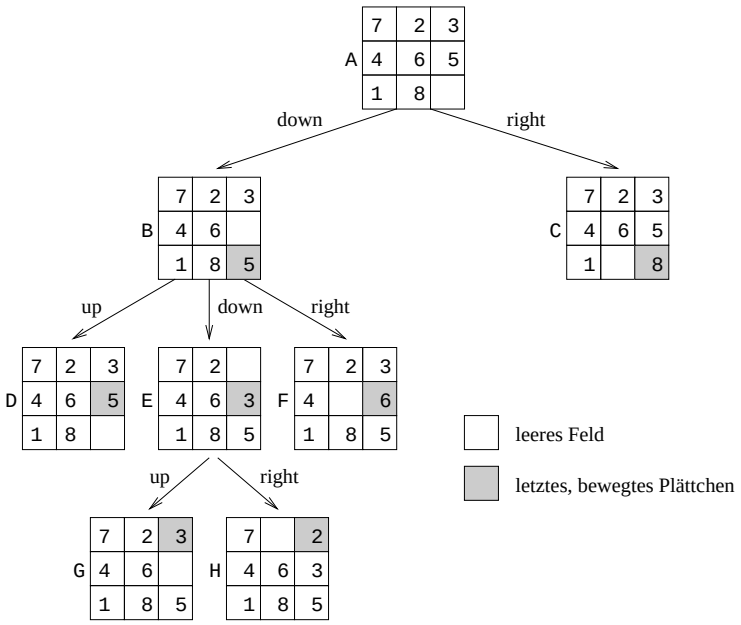


Abbildung 10.3: Teil des Zustandsraums bei Tiefensuche für ein 8-Puzzle-Problem

Zur Verwaltung der Tiefensuche bietet sich ein Keller an, auf dem die unbesuchten Alternativen zusammen mit ihren Vaterknoten abgelegt sind.

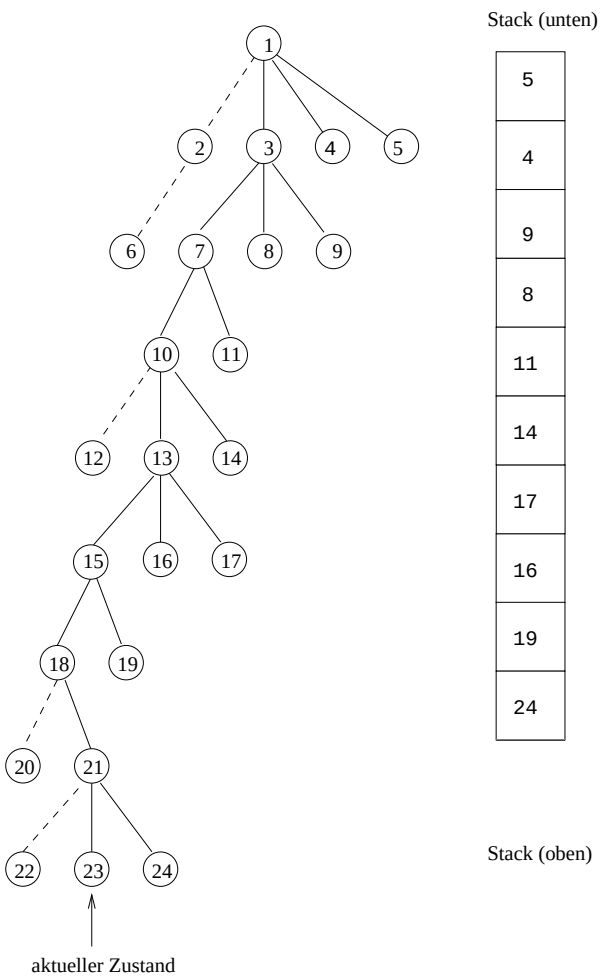


Abbildung 10.4: Zustandsgraph und Kellerinhalt bei Tiefensuche

Best First Search operiert nicht wie *Depth First Search* am letzten besuchten Knoten, sondern an dem Knoten mit der größten Erfolgsaussicht. Hierfür entsteht Speicherbedarf proportional zur Größe des durchsuchten Zustandsraums.
Der A^*-Algorithmus expandiert jeweils den Knoten mit dem niedrigsten l-Wert. Dessen Söhne kommen auf die sogenannte *OPEN-List* (es sei denn, sie befinden sich bereits dort), der expandierte Knoten kommt auf die *CLOSED-List* (es sei denn, er befindet sich bereits dort).

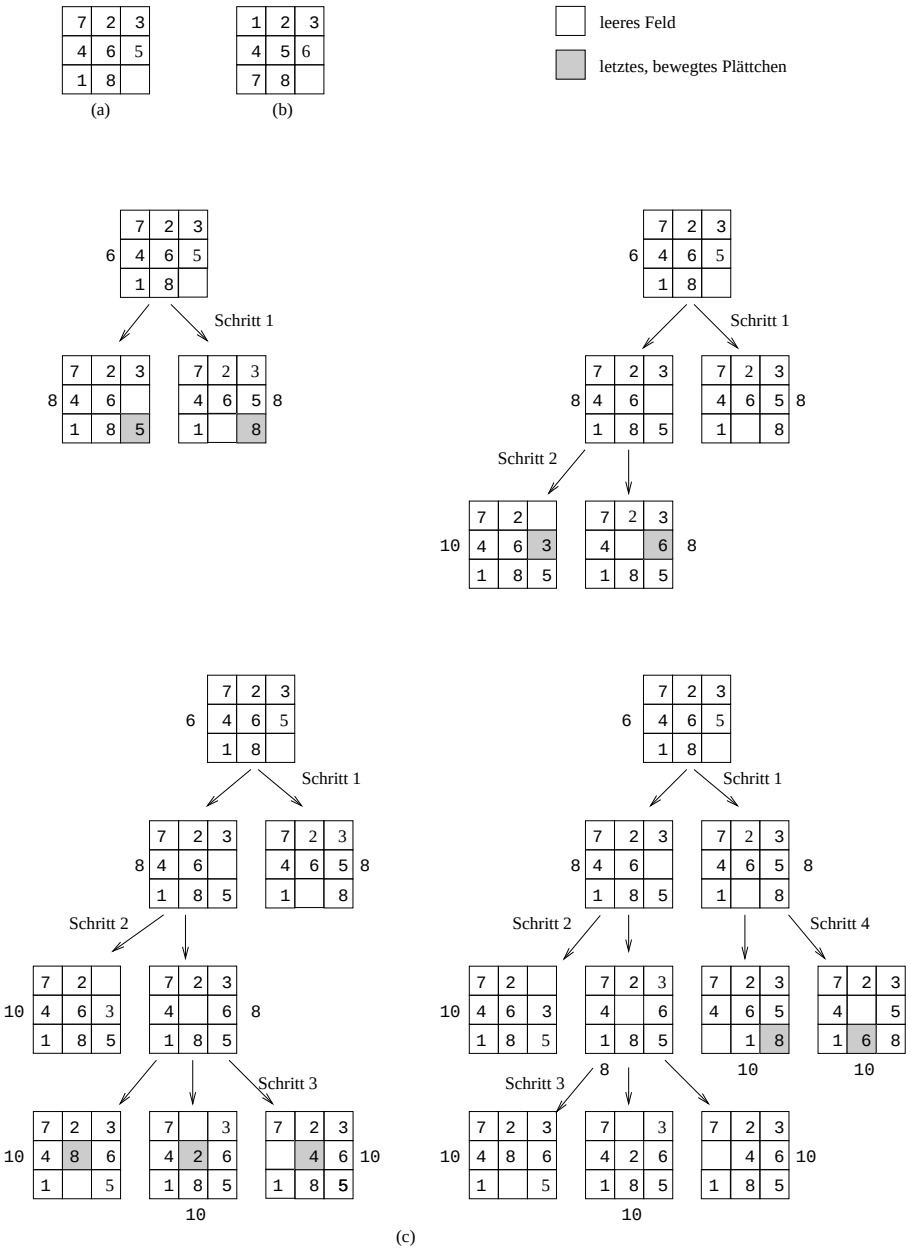


Abbildung 10.5: Best First Search für ein 8-Puzzle
Startkonfiguration (a)
Zielkonfiguration (b)
Zustände erzeugt durch 4 Schritte Best First Search (c)
Zustände sind markiert mit ihrem l-Wert = Manhattandistanz zwischen Zustand und Zielzustand + bisherige Weglänge

10.3 Paralleles Suchen

Parallele Suchverfahren verursachen einen Kommunikationsoverhead aufgrund von

- Datentransfer
- idle times (Leerlauf wegen Lastungleichheit)
- memory contention (gleichzeitiger Speicherzugriff)

Zusätzlich kann ein Suchoverhead entstehen, da der parallele Algorithmus ggf. andere Teile des Suchraums exploriert als der sequentielle Algorithmus.

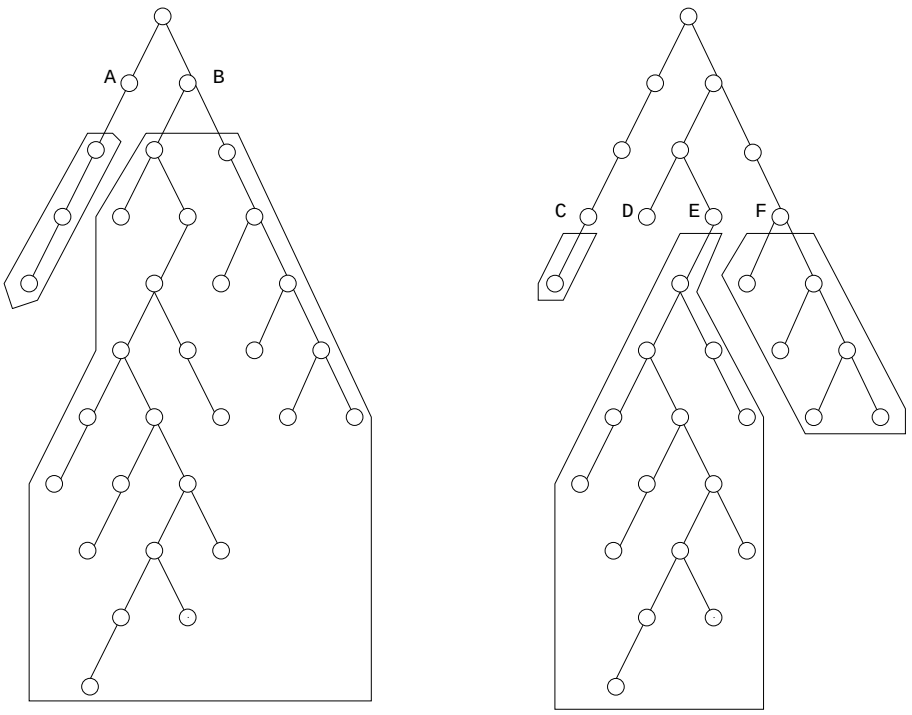


Abbildung 10.6: Lastungleichgewicht bei Aufteilung für 2 bzw. 4 Prozessoren

Offenbar kann eine statische Lastverteilung zu großem Ungleichgewicht führen. Also muß zur Laufzeit eine dynamische Lastverteilung stattfinden.

Dynamische Lastverteilung für Paralleles Backtracking

Ein unbeschäftigter Prozessor wendet sich an eine zentrale Datenstruktur (z.B. Keller) bzw. sucht sich unter seinen unmittelbaren Nachbarn oder unter allen Prozessoren im Netzwerk einen Spender aus und bittet ihn um Arbeit. Der betroffene Spender gibt einen Teil seines Arbeitsvolumens ab.

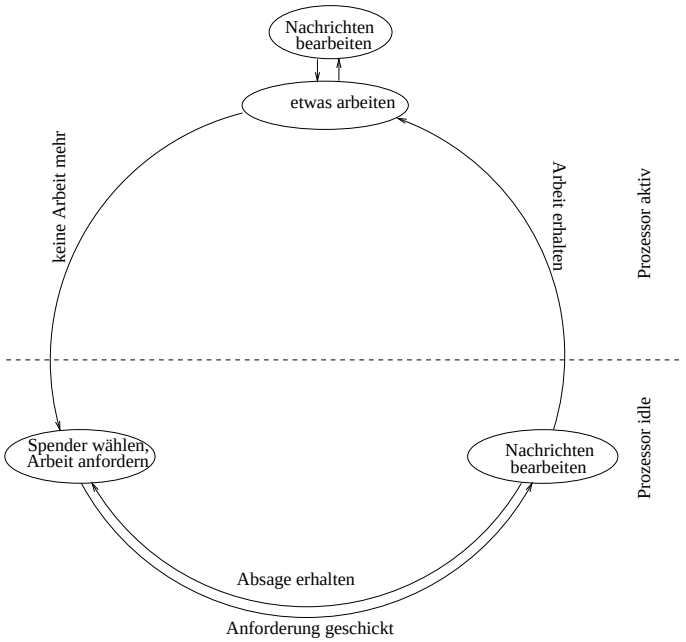


Abbildung 10.7: Generelles Schema für dynamische Lastverteilung

Asynchrones Round Robin

Jeder Prozessor verwaltet eine lokale Variable `spender_id`. Ein unbeschäftigter Prozessor fordert Arbeit an von dem Prozessor mit der Kennung `spender_id` und erhöht `spender_id` um eins (modulo Anzahl der Prozessoren).

Global Round Robin

Beim Prozessor P_0 wird eine globale Variable `spender_id` verwaltet. Ein unbeschäftigter Prozessor fordert Arbeit an von dem Prozessor mit der Kennung `spender_id` und erhöht `spender_id` um eins (modulo Anzahl der Prozessoren).

Random Polling

Ein unbeschäftigter Prozessor fordert Arbeit an von einem zufällig ausgewählten Prozessor.

Idealerweise gibt der Empfänger einer Arbeitsanforderung die Hälfte seiner im Keller gespeicherten Arbeitslast ab (*half split*). Um das Verschicken zu kleiner Arbeitspakete zu vermeiden, werden Knoten unterhalb der *Cutoff-Tiefe* nicht abgegeben.

Beispiel für Baumsuche ohne Zielfunktion:

Gegeben: gerichteter Graph $G = (V, E)$
Frage: Hat G einen Hamiltonkreis?

Ein Expansionsschritt erzeugt aus einem Graphen G anhand einer Kante e zwei Graphen G_e und $G_{\overline{e}}$:

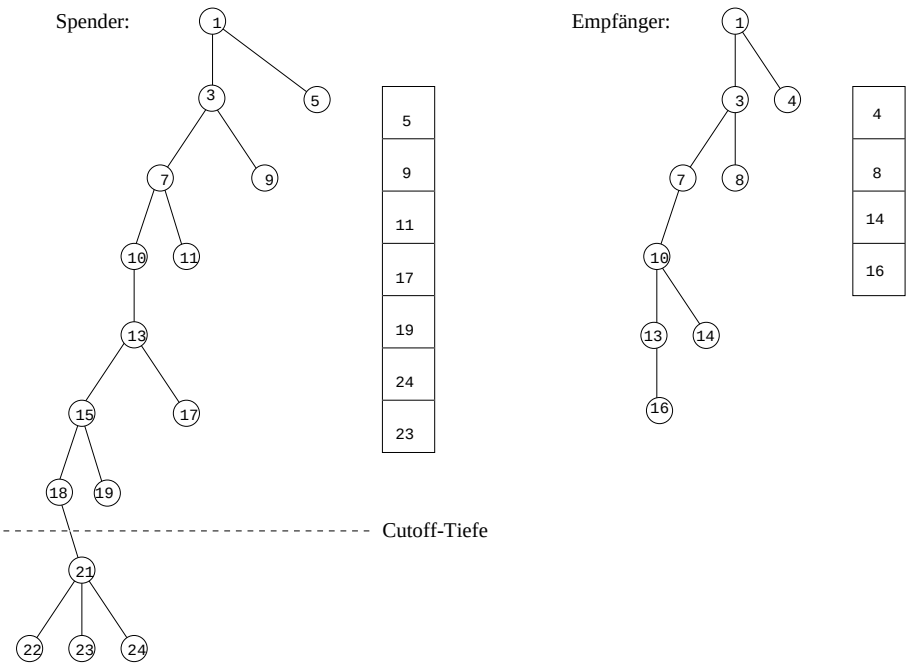


Abbildung 10.8: Ergebnis eines *Half Split* des Kellerinhalts von Bild 10.4

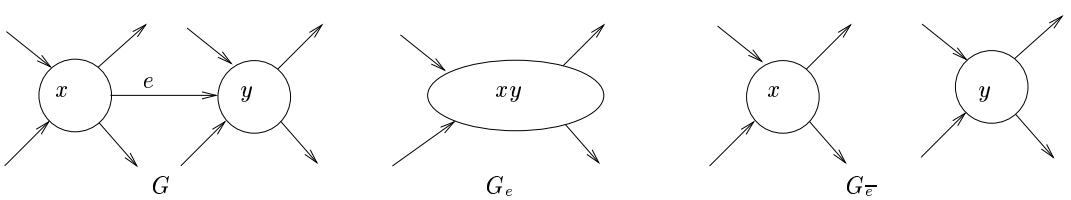


Abbildung 10.9: Expandieren beim Hamiltonkreis-Problem

Dynamische Lastverteilung für Paralleles Best First Search

Die Wahl des Spenders erfolgt nach denselben Kriterien wie beim *Parallelen Backtracking*, d.h., entweder existiert eine zentrale Datenstruktur (z.B. *Heap*), oder Teilaufgaben werden von anderen Prozessoren angefordert.

Bei Verwendung einer zentralen Datenstruktur erhält der anfordernde Prozessor das günstigste Problem. Nachdem er es expandiert hat, werden die Nachfolger wieder eingefügt.

Bei verteilter Datenhaltung verwaltet jeder Prozessor einen lokalen Heap, aus dem er das jeweils günstigste Problem entfernt und nach der Expansion die Nachfolger wiederum einfügt. Um zu vermeiden, daß Prozessoren an ungünstigen Problemen arbeiten, obwohl im Netzwerk günstigere existieren, verteilt ein Prozessor von Zeit zu Zeit einige seiner günstigsten Teilprobleme an andere Prozessoren. Je nach Topologie werden beliebige Empfänger gewählt oder auch nur Nachfolger und Vorgänger bzgl. eines fest gewählten Hamiltonkreises im Netzwerk.

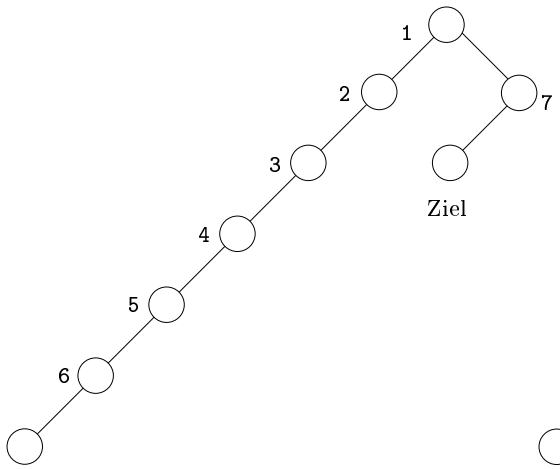
Die Wahl des Zeitpunkts zum Informationsaustausch mit den Nachbarn kann z.B. ausgelöst werden durch das Ansteigen der lokalen unteren Schranke. Eine andere Methode basiert auf einem andauernden Versenden eigener günstiger Probleme. Erhält Prozessor A vom Nachbarn B günstigere Probleme, als er selbst hat, so wird die Sendefrequenz für Kanal AB auf "niedrig" gesetzt; erhält Prozessor A vom Nachbarn B ungünstigere Probleme, als er selbst hat, so wird die Sendefrequenz für Kanal AB auf "hoch" gesetzt.

Als Ergebnis der Lastverteilung gleichen sich die lokalen unteren Schranken an, wodurch ein globaler Heap, auf den mehrere Prozessoren zugreifen, simuliert wird.

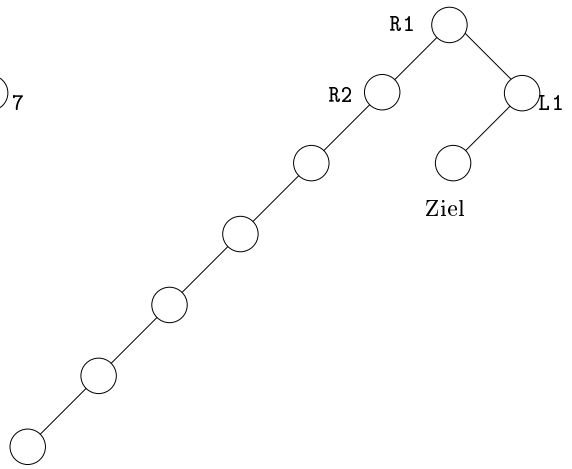
Bei Suchverfahren in Zustandsgraphen, die mehrfache Exploration durch Abgleich mit der *OPEN*-List und der *CLOSED*-List vermeiden wollen, entsteht im parallelen Fall zusätzlicher Overhead: Durch eine Hashfunktion f wird jeder Knoten des Suchraums auf eine Prozessorkennung $0, \ldots, p-1$ abgebildet. Ein Prozessor, der einen Knoten x erzeugt, schickt ihn zur weiteren Bearbeitung an Prozessor $f(x)$, der ihn mit dem Bestand seiner lokalen Listen abgleicht.

Speedup-Anomalien

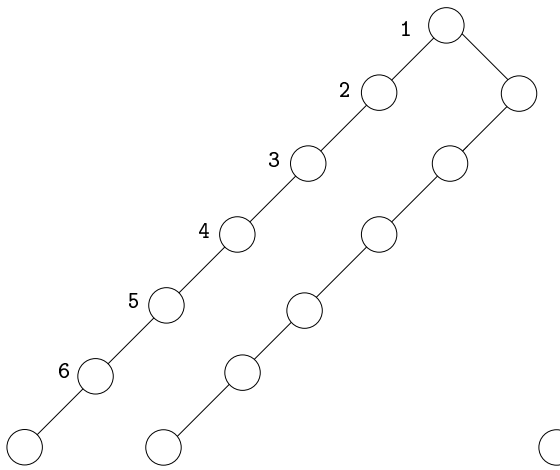
Durch die unterschiedliche Vorgehensweise beim parallelen Suchen können gegenüber der sequentiellen Suche weniger oder mehr Knoten besucht werden. Dadurch entsteht superlinearer bzw. sublinearer Speedup. Die Bilder 10.10 und 10.11 zeigen Anomalien bei *Depth First Search* bzw. *Best First Search*.



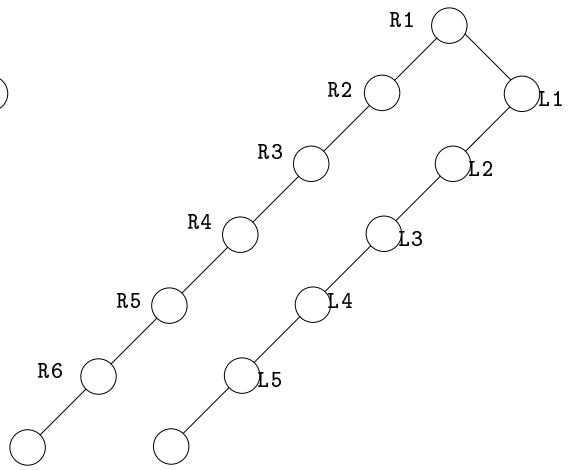
Ziel erzeugt von einzigem Prozessor
bei seiner 7. Expansion



Ziel erzeugt von Prozessor L
bei seiner 1. Expansion. Speedup $= \frac{7}{2} = 3.5 > 2$



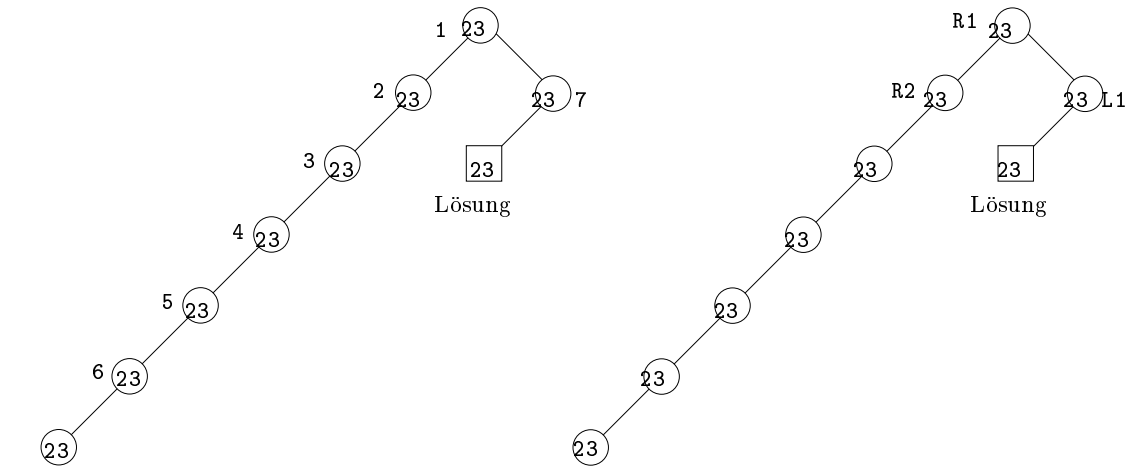
Ziel erzeugt von einzigem Prozessor
bei seiner 6. Expansion



Ziel erzeugt von Prozessor R
bei seiner 6. Expansion. Speedup $= \frac{6}{6} = 1 < 2$

Abbildung 10.10: Anomalien bei *Depth First Search*

Sei *opt* der optimale Zielfunktionswert. Ein Knoten im Zustandsraum mit einem l-Wert $b < opt$ muß von jedem sequentiellen und parallelen Algorithmus expandiert werden. Ein Knoten mit l-Wert $b = opt$ muß nur dann expandiert werden, wenn zu diesem Zeitpunkt noch keine Lösung mit diesem Wert vorliegt.



Lösung erzeugt von einem Prozessor bei seiner 7. Expansion

Lösung erzeugt von Prozessor L bei seiner 1. Expansion. Speedup $= \frac{7}{2} = 3.5 > 2$

Abbildung 10.11: Anomalien bei *Best First Search*

10.4 Spielbaumsuche

Ein Spielbaum hat zwei Typen von Knoten: Minimum-Knoten und Maximum-Knoten. Die Knoten repräsentieren Spielstellungen in einem 2-Personen-Spiel. Der Wert eines Blattes wird bestimmt durch eine statische Stellungsbewertung. Der Wert eines Minimum-Knotens ist das Minimum der Werte seiner Söhne. Der Wert eines Maximum-Knotens ist das Maximum seiner Söhne.

```
PROCEDURE minmax (s: Spielbaum): INTEGER;
BEGIN
   IF blatt(s) THEN RETURN statisch(s)
   ELSE
      bestimme Nachfolger s[1], s[2], ..., s[d];
      IF typ(s) = max THEN t := -INFINITY ELSE t := +INFINITY END;
      FOR i := 1 TO d DO
         m := minmax (s[i});
         IF typ(s) = max AND m > t THEN t := m END;
         IF typ(s) = min AND m < t THEN t := m END
    END
    RETURN t
END
```

Der Wert der Wurzel läßt sich durch eine komplette Tiefensuche bis zu den Blättern ermitteln. Eine Beschleunigung wird dadurch erreicht, daß bei einem Knoten dann keine weiteren Söhne bearbeitet werden, wenn ihre Werte keinen Einfluß auf den Wert der Spielbaumwurzel haben.

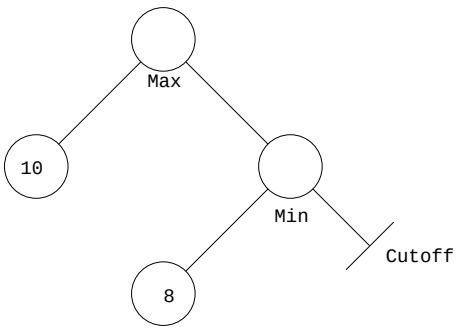


Abbildung 10.12: *Cutoff* im Spielbaum

Hierzu werden zwei Schranken α und β (für die Maximierungs- bzw. Minimierungsebenen) übergeben, die zu einem vorzeitigen Cutoff führen können. D.h., ein Maxknoten verursacht einen Abbruch bei Überschreiten von β, ein Minknoten verursacht einen Abbruch bei Unterschreiten von α. Die Wurzel des Baumes wird mit $\alpha = -\infty,\ \beta = +\infty$ aufgerufen.

Bemerkung: Bei Tiefe h und Verzweigungsgrad d erzeugt `minmax` d^h Blätter. Unter günstigen Umständen (alle Söhne sind nach Qualität sortiert) erzeugt `alphabeta` $2 \cdot d^{h/2}$ Blätter, also eine Reduktion von n auf $\sqrt{n}$.

```
PROCEDURE alphabeta (s: Spielbaum; α, β : INTEGER): INTEGER;
BEGIN
   IF blatt(s) THEN RETURN statisch(s)
   ELSE
      bestimme Nachfolger s_1, s_2, ..., s_d;
            FOR i := 1 TO d DO
         m := alphabeta(s_i, α, β);
         IF typ(s) = max AND m > α THEN α := m END;
         IF typ(s) = min AND m < β THEN β := m END;
         IF α ≥ β THEN RETURN m;
      END
      IF typ (s) = max THEN RETURN α ELSE RETURN β
   END
```

Bei der parallelen Spielbaumsuche bearbeiten Prozessoren lokale Teilbäume (analog wie bei Tiefensuche), die sie bei Bedarf als *Auftragnehmer* von einem anderen Prozessor, genannt *Auftraggeber*, anfordern. Zusätzlich entsteht Kommunikationsbedarf:

- Der Auftragnehmer meldet das Ergebnis seiner Teilbaumauswertung an den Auftraggeber zurück, da es dort benötigt wird zur Bestimmung des Vater-Wertes.
- Der Auftraggeber meldet sich verkleinernde $\alpha - \beta$-Fenster an seine Auftragnehmer weiter, da sie dort zu zusätzlichen Cutoffs führen können.

Können die Söhne eines Knotens durch eine Heuristik vorsortiert werden, so sollten erst nach Auswertung des (vermutlich) besten Sohns dessen Brüder an Auftragnehmer abgegeben werden. Somit entsteht ein unvermeidbarer Tradeoff zwischen unbeschäftigten Prozessoren und überflüssiger Suche.

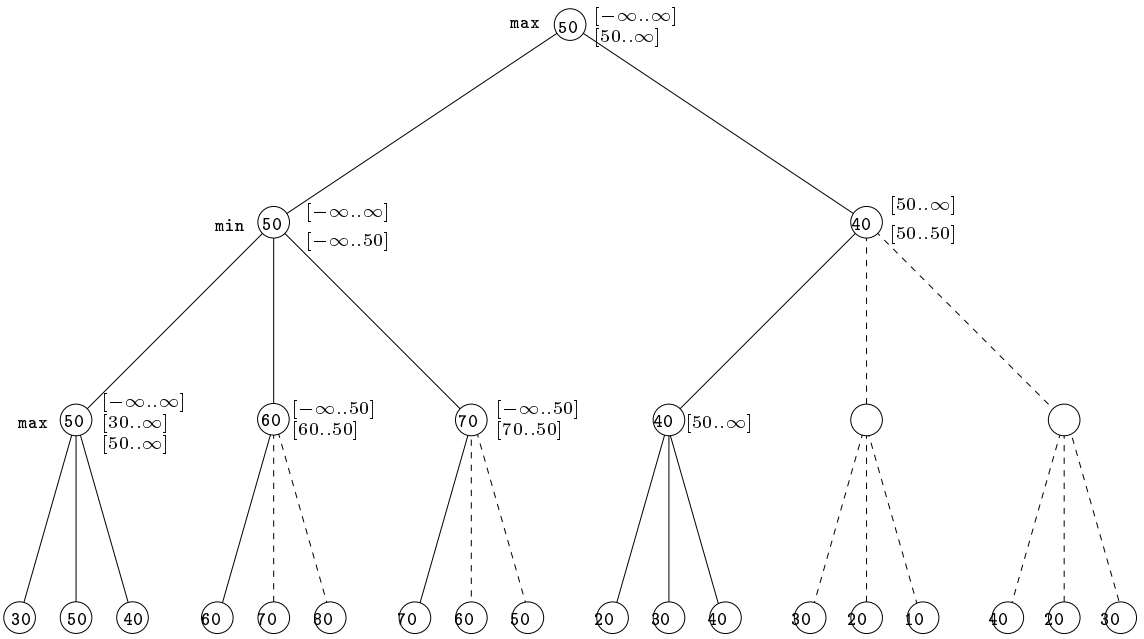


Abbildung 10.13: Alpha-Beta-Suche in einem Spielbaum. Vermerkt an den Knoten sind die sich ändernden Suchfenster. *Cutoffs* sind durch gestrichelte Kanten angedeutet.

Bemerkung: Das Paderborner Schachprogramm *ZUGZWANG* erreichte auf 1024 Prozessoren einen maximalen Speedup von 400 und einen mittleren Speedup von 344.

10.5 Dynamic Programming

Dynamic Programming ist eine Lösungstechnik für kombinatorische Optimierungsprobleme, bei der sich die Kosten eines Problems x durch Komposition der Kosten einiger Teilprobleme $x_1, x_2, \ldots, x_k$ ermitteln läßt, d.h.

$$f(x) := g(f(x_1), f(x_2), f(x_3), \ldots, f(x_k)) \ .$$

Seien z.B. $f(x)$ die Kosten des kürzesten Weges vom Knoten 0 zum Knoten x in einem azyklischen Graph ($(x,y) \in E \Rightarrow x < y$) gegeben durch

$$f(x) = \begin{cases} 0 & ; \quad x = 0 \\ \min\limits_{0 \leq j < x} \{f(j) + c(j,x)\} & ; \quad 1 \leq x \leq n-1 \end{cases}$$

0/1-Rucksack-Problem

Gegeben ein Rucksack mit Kapazität c. Gegeben n Objekte $1, 2, 3, \ldots, n$. Objekt i habe Gewicht w_i und Profit p_i. Ziel ist es, den Rucksack mit größtmöglichem Profit zu füllen, d.h., gesucht sind $v_1, v_2, \ldots, v_n \in \{0, 1\}$ mit

$$\sum_{i=1}^{n} w_i \cdot v_i \leq c$$

und

$$\sum_{i=1}^{n} p_i \cdot v_i$$

wird maximiert.

Der naive Lösungsansatz überprüft alle 2^n Möglichkeiten. Ein *Dynamic-Programming*-Ansatz definiert

$$F[i,x] \ := \ \begin{array}{l} \text{maximaler Profit, den die Objekte } 1, \ldots, i \text{ in einem} \\ \text{Rucksack mit Kapazität } x \text{ erreichen können.} \end{array}$$

Offenbar

$$F[i,x] = \begin{cases} 0 & i = 0, x \geq 0 \\ -\infty & i = 0, x < 0 \\ \max\{F[i-1,x], F[i-1, x-w_i] + p_i\} & 1 \leq i \leq n \end{cases}$$

Eine iterative Formulierung füllt die Profitmatrix F zeilenweise:

```
FOR x := 1 TO w_1 -1 DO F[1,x] := 0 END;
FOR x := w_1 TO c DO F[1,x] := p_1 END;
FOR i := 2 TO n DO
   FOR x := 1 TO c DO
      F[i, x] := max {F[i-1,x], F[i-1, x-w_i] + p_i}
   END
END
```

Die Laufzeit beträgt $O(n \cdot c)$.

Bemerkung: Dies ist ein *Exponentialzeitalgorithmus*, da der Wert von c exponentiell zu seiner Darstellung ist.

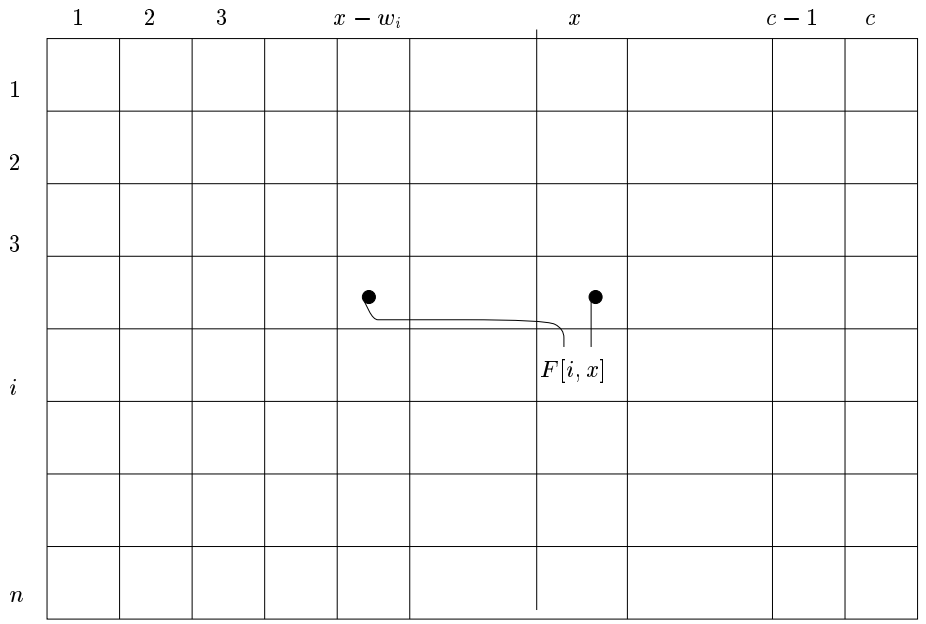


Abbildung 10.14: Einträge der Profitmatrix F für das 0/1-Rucksack-Problem. Für die Berechnung $F[i, x]$ sind $F[i-1, x]$ und $F[i-1, x-w_i]$ notwendig.

Zur parallelen Abarbeitung mit einer CREW-PRAM verwendet man c Prozessoren. Während der i-ten Iteration ist Prozessor P_x zuständig für die Bestimmung von $F[i, x]$. Die Laufzeit beträgt offenbar $O(n)$, die Kosten $O(n \cdot c)$, also liegt ein kostenoptimaler Algorithmus vor.

Zur parallelen Abarbeitung auf einem Hypercube verwendet man c Prozessoren. Jeder Prozessor kennt alle Gewichte w_i und alle Profitwerte p_i. Prozessor P_x ist zuständig für Spalte x der Profitmatrix F. Während der i-ten Iteration kann P_x auf das lokal vorhandene $F[i-1, x]$ zugreifen; der Wert $F[i-1, x-w_i]$ muß besorgt werden durch einen zyklischen Shift über die Distanz w_i, ausgeführt von allen Prozessoren. Die Laufzeit hierfür beträgt $\log(c)$. Die Gesamtzeit beträgt daher $O(n \cdot \log c)$, die Kosten $O(n \cdot c \cdot \log c)$.

Bei p Prozessoren im Hypercube ist jeder Prozessor für c/p Spalten zuständig. Während der i-ten Iteration kann Prozessor P_x auf c/p lokale Werte zugreifen und muß weitere c/p Werte durch einen zyklischen Shift besorgen. Die Zeit dafür beträgt $c/p + \log p$. Die Gesamtzeit beträgt daher $O(n \cdot c/p + n \cdot \log p)$, die Kosten $O(n \cdot c + n \cdot p \cdot \log p)$. Für $c = \Omega(p \cdot \log p)$ ist dies kostenoptimal.

Kapitel 11

Programmiersprachen

Die algorithmische Idee eines *All-to-All-Broadcast im Ring* (Kapitel 4.3.1) wird durch den folgenden Pseudocode präzisiert. Hierbei bilden alle Prozessoren im Ring die Summe über alle Prozessorkennungen, indem sie diese Kennungen zyklisch weiterreichen:

```
/*****************************************************************************/
/*                                                                           */
/*                    Summe im Ring als Pseudocode                           */
/*                                                                           */
/*****************************************************************************/
FOR p=0 TO n-1 DO IN PARALLEL

   id   = p;                   /* besorge die eigene Prozessorkennung          */
   anz  = n;                   /* besorge die Anzahl der Prozessoren           */
   odd  = id % 2;              /* lege fest, ob ungerade ID vorliegt           */

                               /* Topologie festlegen:                         */
   pre = LINK TO (id-1) % anz; /* Vorgaenger                                   */
   suc = LINK TO (id+1) % anz; /* Nachfolger                                   */

                               /* parallele Summe berechnen:                   */
   sum = id;                   /* vorlaeufige Summe                            */
   out = id;                   /* naechster zu uebertragender Wert             */
   FOR z = 1 TO anz-1 DO       /* anz-1 mal tue:                               */
      IF (odd)                 /* falls ungerade ID                            */
         RECV(pre, in );       /* erhalte vom Vorgaenger einen Wert fuer in    */
         SEND(suc, out);       /* schicke zum Nachfolger den Wert von out      */
      ELSE                     /* falls gerade ID                              */
         SEND(suc, out);       /* schicke zum Nachfolger den Wert von out      */
         RECV(pre, in );       /* erhalte vom Vorgaenger den Wert fuer in      */
      sum += in;               /* Summe erhoehen                               */
      out = in;                /* naechste Ausgabe vorbereiten                 */
   END
END
```

Auf den nächsten Seiten wird dieser Pseudocode in der Syntax von PARIX, MPI und PVM formuliert.

```
/*****************************************************************************************/
/*                                                                                       */
/*        Summe im Ring als Parix-Programm mit synchroner Kommunikation                  */
/*                                                                                       */
/*****************************************************************************************/
#include <sys/root.h>
#include <stdlib.h>

void main (int argc, char **argv)
{
   int anz, id, odd, sum, in, out, z;
   LinkCB_t *pre, *suc;                                /* Zeiger auf Link-Kontrollblocks     */
   int error;                                          /* Variable fuer Fehlermeldung        */
                                                       /* logische Topologie festlegen :     */

   anz = GET_ROOT()->ProcRoot->nProcs;                 /* Macro liefert Anzahl der Proz.     */
   id  = GET_ROOT()->ProcRoot->MyProcID;               /* Macro liefert Prozessor-ID         */
   odd = id % 2;                                       /* lege fest, ob ungerade ID vorliegt */
                                                       /* die Kommunikationspartner muessen  */
                                                       /* sich gleichzeitig mit derselben ID */
                                                       /* auf ein Link verstaendigen:        */
   if (odd) {
      suc = ConnectLink((id+1+anz) % anz, 42, &error); /* definiere Link zum Nachfolger      */
      pre = ConnectLink((id-1+anz) % anz, 42, &error); /* definiere Link zum Vorgaenger      */
   } else {
      pre = ConnectLink((id-1+anz) % anz, 42, &error); /* definiere Link zum Vorgaenger      */
      suc = ConnectLink((id+1+anz) % anz, 42, &error); /* definiere Link zum Nachfolger      */
   }
                                                       /* Parallele Summe berechnen:         */

   sum = out = id;                                     /* initialisiere Variablen            */

   for (z = 1; z < anz; z++) {                         /* anz-1 mal tue:                     */
      if (odd) {
         RecvLink(pre, &in,  sizeof(int));             /* ueber Link pre empfangen nach in   */
         SendLink(suc, &out, sizeof(int));             /* ueber Link suc versenden von out   */
      } else {
         SendLink(suc, &out, sizeof(int));             /* ueber Link suc versenden von out   */
         RecvLink(pre, &in,  sizeof(int));             /* ueber Link pre empfangen nach in   */
      }
      sum += in;                                       /* Summe erhoehen                     */
      out  = in;                                       /* naechste Ausgabe vorbereiten       */
   }
   exit(0);                                            /* Programm beenden                   */
}
```

Nach Übersetzen der Quelle lautet der Aufruf von der Shell-Ebene für ein 4×4-Gitter:

```
$ run -c 0 4 4 summe_im_ring.px
```

```
/*****************************************************************************/
/*                                                                           */
/*  Summe im Ring als Parix-Programm unter Verwendung einer virtuellen Topologie        */
/*                                                                           */
/*****************************************************************************/
#include <sys/root.h>
#include <stdlib.h>
#include <virt_top.h>

void main (int argc, char **argv)
{
   int anz, id, odd, sum, in, out, z;
   int ring, pre, suc;
   RingData_t  *ring_data;                      /* Zeiger auf Topologiestruktur         */

                                                /* logische Topologie festlegen:        */

   anz = GET_ROOT()->ProcRoot->nProcs;          /* Macro liefert Anzahl der Prozessoren */

   ring = MakeRing(1, anz, MINSLICE, MAXSLICE,  /* bilde Ring in ein 3D-Gitter ab       */
                           MINSLICE, MAXSLICE,  /* dabei soll jeweils pro Dimension     */
                           MINSLICE, MAXSLICE); /* das gesamte Intervall genutzt werden */
   ring_data = GetRing_Data(ring);              /* besorge Topologieinformation         */

   id = ring_data->id;                          /* logische ID bzgl. des Rings          */
   odd = id % 2;                                /* lege fest ob ungerade                */

   suc = 1;                                     /* Name fuer Nachfolgerlink bzgl. Ring  */
   pre = 0;                                     /* Name fuer Vorgaengerlink bzgl. Ring  */

                                                /* Parallele Summe berechnen:           */


   sum = out = id;                              /* initialisiere Variablen              */

   for (z = 1; z < anz; z++) {                  /* anz-1 mal                            */

      if (odd) {
          Recv(ring, pre, &in,  sizeof(int));   /* ueber Link pre im Ring empfangen     */
          Send(ring, suc, &out, sizeof(int));   /* ueber Link suc im Ring verschicken   */
      } else {
          Send(ring, suc, &out, sizeof(int));   /* ueber Link suc im Ring verschicken   */
          Recv(ring, pre, &in,  sizeof(int));   /* ueber Link pre im Ring empfangen     */
      }
      sum += in;                                /* Summe erhoehen                       */
      out  = in;                                /* naechste Ausgabe vorbereiten         */
   }

   FreeTop(ring);                               /* Programm beenden                     */
   exit(0);
}
```

```
/*****************************************************************************/
/*                                                                           */
/*    Summe im Ring als Parix-Programm unter Verwendung asynchroner Kommunikation    */
/*                                                                           */
/*****************************************************************************/


#include <sys/root.h>
#include <stdlib.h>
#include <virt_top.h>

void main (int argc, char **argv)
{
   int anz, id, odd, sum, in, out, z;
   int ring, pre, suc;
   RingData_t  *ring_data;
   int result;


   anz = GET_ROOT()->ProcRoot->nProcs;

   ring = MakeRing(1, anz, MINSLICE, MAXSLICE,
                           MINSLICE, MAXSLICE,
                           MINSLICE, MAXSLICE);
   ring_data = GetRing_Data(ring);

   id = ring_data->id;
   odd = id % 2;

   suc = 1;
   pre = 0;

   AInit(ring, -1, -1);                              /* Vorbereitung fuer Threads      */
                                                     /* welche die Kommunikation       */
                                                     /* durchfuehren sollen            */
   sum = out = id;

   for (z = 1; z < anz; z++) {

      ASend(ring, suc, &out, sizeof(int), &result); /* asynchrones Verschicken        */
      ARecv(ring, pre, &in,  sizeof(int), &result); /* asynchrones Empfangen          */
      ASync(ring, -1);                               /* Warten auf Abschluss der       */
                                                     /* Kommunikation                  */

      sum += in;
      out  = in;
   }

   AExit(ring);
   exit(0);
}
```

```
/*****************************************************************************/
/*                                                                           */
/*           Summe im Ring als MPI-Programm                                  */
/*                                                                           */
/*****************************************************************************/
#include "mpi.h"
int main(int argc, char **argv)
{
   int id, anz, odd, pre, suc, sum, in, out, z;
   MPI_Status status;

   MPI_Init ( &argc, &argv );              /* Initialisiere MPI                 */

                                           /* logische Topologie festlegen:     */
   MPI_Comm_size ( MPI_COMM_WORLD, &anz ); /* besorge Anzahl der Prozessoren    */
   MPI_Comm_rank ( MPI_COMM_WORLD, &id  ); /* besorge Prozessor-ID              */

   odd = anz % 2;                          /* lege fest, ob ungerade            */
   pre = ( id - 1 + anz ) % anz;           /* ID des Vorgaengers                */
   suc = ( id + 1       ) % anz;           /* ID des Nachfolgers                */

                                           /* Parallele Summe berechnen:        */

   sum = out = id;                         /* initialisiere Variablen           */

   for (z=1; z < anz; z++) {               /* anz-1 mal                         */

      if (odd) {
         MPI_Recv (&in,                    /* lege ab bei Adresse von in        */
                   1,                      /* ein Datum                         */
                   MPI_INT,                /* nach Bauart MPI_INT               */
                   pre,                    /* erhalten vom Vorgaenger           */
                   42,                     /* versehen mit dem Tag 42           */
                   MPI_COMM_WORLD,         /* bzgl. des allgemeinen Kommunikators */
                   &status );              /* Adresse fuer Fehlerstatus         */
         MPI_Send (&out,                   /* entnehme beginnend bei Adresse out */
                   1,                      /* ein Datum                         */
                   MPI_INT,                /* nach Bauart MPI_INT               */
                   suc,                    /* verschicke an Nachfolger          */
                   42,                     /* versehen mit Tag 42               */
                   MPI_COMM_WORLD );       /* bzgl. des allgemeinen Kommunikators */
      } else {
          MPI_Send ( &out, 1, MPI_INT, suc, 42, MPI_COMM_WORLD );
          MPI_Recv ( &in,  1, MPI_INT, pre, 42, MPI_COMM_WORLD, &status );
      }
      sum += in;
      out  = in;
   }
      MPI_Finalize ();                     /* Programm beenden                  */
}
```

Nach Übersetzen der Quelle lautet der Aufruf von der Shell-Ebene für 16 Prozessoren:

```
$ mpirun -np 16 summe_im_ring
```

```
/*****************************************************************************/
/*                                                                           */
/*       Summe im Ring als MPI-Programm unter Verwendung von reduce          */
/*                                                                           */
/*****************************************************************************/


#include "mpi.h"

int main(int argc, char **argv)
{
    int id, sum;

    MPI_Init ( &argc, &argv );              /* initialisiere MPI                 */

                                            /* logische Topologie festlegen:     */

    MPI_Comm_rank ( MPI_COMM_WORLD, &id );  /* bestimme Prozessor-ID             */

                                            /* Parallele Summe berechnen:        */

    MPI_Allreduce (   &id,                  /* Eingabeparameter: id              */
                      &sum,                 /* Ausgabeparameter: sum             */
                      1,                    /* 1 Datum                           */
                      MPI_INT,              /* von der Bauart MPI_INT            */
                      MPI_SUM,              /* zu bestimmen ist die Summe        */
                      MPI_COMM_WORLD );     /* innerhalb des globalen Kommunikators */


    MPI_Finalize ();                        /* Programm beenden                  */

}
```

```
/*****************************************************************************/
/*                                                                           */
/*           Summe im Ring als PVM-Programm: Master                          */
/*                                                                           */
/*****************************************************************************/


#include "pvm3.h"

void  main ( int  argc, char  **argv )
{
   int  anz, z;
   int  *tids;

   anz = atoi ( argv[1] );                 /* besorge Anzahl der Prozessoren            */
   tids = (int*) malloc (anz*sizeof(int)); /* besorge Speicherplatz fuer Task-Id-Vektor */

   pvm_spawn (  "slave",                   /* Starte das Programm slave                 */
                (char **) NULL,            /* ohne Argumente                            */
                PvmTaskArch,               /* eingeschraenkt auf eine Architektur       */
                "SUN4",                    /* vom Typ SUN4                              */
                anz,                       /* anz mal                                   */
                tids );                    /* erhalte einen Vektor von Task-IDs zurueck */

                                           /* globale Task-Informationen verteilen      */

   for ( z = 0; z < anz; z++ ) {           /* anz mal                                   */
           pvm_initsend ( PvmDataRaw );    /* Sende-Puffer vorbereiten                  */
           pvm_pkint ( &z, 1, 1 );         /* den Wert von z verpacken                  */
           pvm_pkint ( &anz, 1, 1 );       /* den Wert von anz verpacken                */
           pvm_pkint ( tids, anz, 1 );     /* den Task-ID-Vektor verpacken              */
           pvm_send ( tids[z], 0 );        /* an den z-ten Prozessor verschicken        */
   }


   pvm_exit ( );                           /* Task beenden                              */
}
```

1. Virtuelle Maschine zusammenstellen durch Start des PVM-Dämons auf jedem Host (laufen unabhängig im Hintergrund)
2. Programme übersetzen:

   ```
   $ gcc -o summe_im_ring master.c -lpvm3
   $ gcc -o slave slave.c -lpvm3
   ```

3. Aufruf für Ring mit 16 Tasks:

   ```
   $ summe_im_ring 16
   ```

```
/*****************************************************************************/
/*                                                                           */
/*          Summe im Ring als PVM-Programm: Slave                            */
/*                                                                           */
/*****************************************************************************/
#include "pvm3.h"

void  main ( int  argc, char  **argv )
{
   int  id, anz, odd, in, sum, out, pre, suc, z;
   int  *tids;

                                        /* Logische Topologie festlegen:     */

   pvm_recv   ( pvm_parent ( ), -1 );   /* erhalte vom aufspannenden Vater   */
   pvm_upkint ( &id, 1, 1 );            /* entpacke id                       */
   pvm_upkint ( &anz, 1, 1 );           /* entpacke anz                      */
   tids = (int*) malloc (anz*sizeof(int)); /* besorge Platz fuer Task-ID-Vektor */
   pvm_upkint ( tids, anz, 1 );

   odd = id % 2;                        /* lege fest, ob ungerade id vorliegt */
   pre = tids[(id+anz-1)%anz];          /* Task-ID des Vorgaengers           */
   suc = tids[(id+1)%anz];              /* Task-ID des Nachfolgers           */

                                        /* Parallele Summe berechnen:        */
   sum = out = id;
   for ( z = 1; z < anz; z++ ) {        /* anz-1 mal                         */
      if ( odd ) {
              pvm_recv ( pre, -1 );     /* erhalte vom Vorgaenger            */
              pvm_upkint ( &in, 1, 1 ); /* entpacke nach in                  */

              pvm_initsend ( PvmDataRaw ); /* initialisiere Ausgabepuffer    */
              pvm_psend ( suc,          /* versende zum Nachfolger           */
                          0,            /* mit dem Tag 0                     */
                          &out,         /* beginnend bei Adresse von out     */
                          1,            /* ein Datum                         */
                          PVM_INT);     /* nach Bauart PVM_INT               */
      } else {
              pvm_initsend ( PvmDataRaw );
              pvm_psend ( suc, 0, &out, 1, PVM_INT );
              pvm_recv ( pre, -1 );
              pvm_upkint ( &in, 1, 1 );
      }

      sum += in;                        /* Summe erhoehen                    */
      out = in;                         /* naechsten Ausgabewert vorbereiten */
   }

   pvm_exit ( );                        /* Programm beenden                  */
}
```