Kapitel 9

Kommunikationsmodelle

In diesem Kapitel werden wir uns mit grundlegenden Modellen von Kommunikationssystemen, insbesondere mit Prozessor-Netzwerken, beschäftigen.

Die *Prozessor-Netzwerke*, die wir untersuchen werden, repräsentieren die Hauptmodelle der heutigen massiv-parallelen Computer, in denen kein globaler Speicher zum Datenaustausch zur Verfügung steht. Kommunikation zwischen den Prozessoren findet stattdessen über direkte Verbindungsleitungen statt.

Im ersten Teil untersuchen wir die Grundmodelle der sogenannten schnellen Netzwerke:

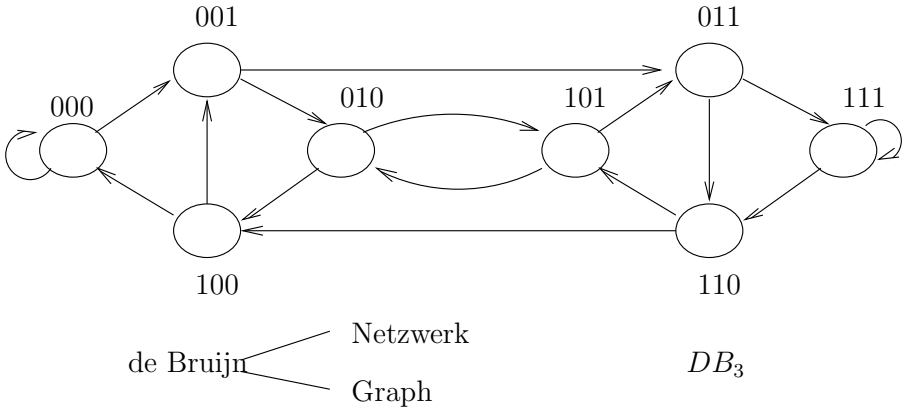
- Hypercubes,
- Shuffle-Exchange
- Perfect-Shuffle – de Bruijn Graph,
- Cube-Connected Cycles und
- Butterfly.

Wir zeigen die wichtigen Eigenschaften dieser Modelle und die Zusammenhänge zwischen verschiedenen Modellen. Wir präsentieren auch einige Beispiele von Algorithmen für diese Prozessor-Netzwerke.

Im zweiten Teil untersuchen wir in Details die Hauptprobleme der Berechnungen auf dem wichtigsten Modell der Netzwerke, *Hypercubes*, das Routing-Problem, die Implementation von PRAMs auf Hypercube-Netzen und die Einbettung von verschiedenen Netzwerken in Hypercubes.

9.1 Netzwerke und ihre Komplexitätsmaße

Definition 9.1 (Netzwerk):
Ein *Netzwerk* wird spezifiziert durch einen endlichen oder unendlichen Graphen $G = (V, E)$ mit endlichem Grad. (G heißt die „Topologie“ oder „Architektur“ des Netzwerks.) Die Knoten repräsentieren die Prozessoren (RAMs), Kanten bezeichnen lokale Nachbarschaften zwischen Prozessoren. □



Netzwerke sind in der Praxis sehr groß. Wirklich wichtig sind deshalb nur die Netzwerke, deren Graphen sehr regulär sind, d.h., alle Knoten haben den gleichen Grad und die gleichen Nachbarschaftsbeziehungen. Wichtig sind die *Familien von Netzwerken* mit der gleichen Topologie – d.h., deren Graphen vom selben Typ sind.

Viele sehr große und komplizierte Netzwerk-Graphen kann man sehr einfach beschreiben, wenn man die Knoten mit binären Strings bezeichnet und die Kanten durch String-Operationen oder Zahlen-Operationen (über den Zahlen, die diese Strings repräsentieren) definiert.

Z.B. den sogenannten n-dimensionalen *de-Bruijn-Graphen* DB_n kann man folgendermaßen definieren:

$$\begin{aligned} DB_n &= (V_n, E_n), \quad V_n = \{0,1\}^n \\ E_n &= \{(v_1 v_2 \ldots v_n, v_2 \ldots v_n 0), (v_1 v_2 \ldots v_n, v_2 \ldots v_n 1) | v_1 \ldots v_n \in V_n\} \end{aligned}$$

Definition 9.2:
Die *Komplexitätsmaße* für Netzwerk-Graphen $G = (V, E)$

Grad – $d(G)$**:** Maximum der Grade der Knoten von G (d.h., Maximum der eingehenden und ausgehenden Kanten für jeden Knoten)

Abstand zweier Knoten: die Länge, d.h. die Zahl der Kanten des kürzesten Weges zwischen den beiden Knoten

Durchmesser – $D(G)$: das Maximum der Abstände unter allen Knotenpaaren

Bisektionsweite, $B(G)$: (*Bisection-Width*) die minimale Anzahl der Kanten, die man entfernen muß, um den Graphen zu trennen, so dass beide Teile dieselbe Anzahl Knoten besitzen

Layout Area – $A(G)$: die minimale Fläche zur Einbettung von G

□

Z.B. gilt für de-Bruijn-Graphen DB_3:

$$\begin{aligned} d(DB_3) &= 4 \\ D(DB_3) &= 3 \\ B(DB_3) &= 4 \end{aligned}$$

In diesem Kapitel werden wir uns mit den sogenannten „schnellen“ Netzwerken beschäftigen, in denen der Durchmesser $O(\log n)$ ist, wobei n die Anzahl der Knoten in dem Graphen ist (die *Größe* des Graphen).

Die *Bisektionsweite* ist oft ein kritischer Faktor, um die Geschwindigkeit zu bestimmen, mit der ein Netzwerk eine Berechnung ausführen kann.

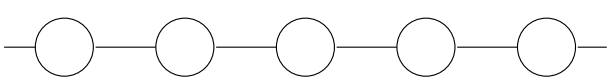
Für n-dimensionale de-Bruijn-Graphen DB_n gilt:

$$\begin{aligned} d(DB_n) &= 4 \\ D(DB_n) &= n \\ B(DB_n) &= \Theta\left(\frac{2^n}{n}\right) \end{aligned}$$

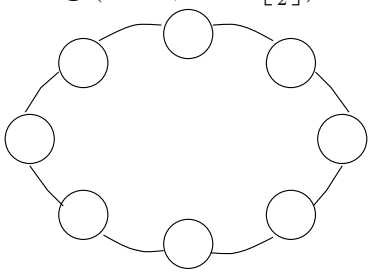
9.2 Die grundlegenden Netzwerke

n – Anzahl der Knoten
d – Grad
D –Durchmesser

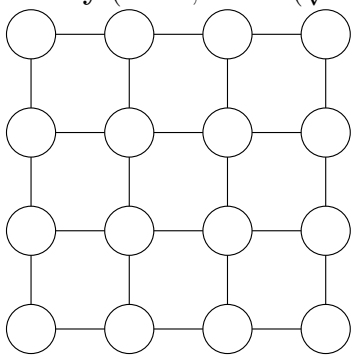
- **Linear Array:**



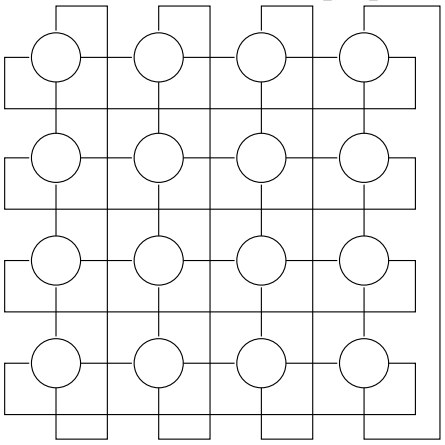
- **Ring** ($d = 2,\ D = \lfloor \frac{n}{2} \rfloor$):



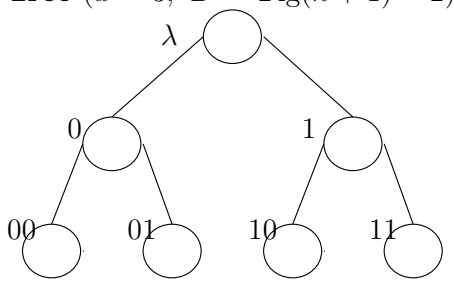
- **Array** ($d = 4,\ D = 2(\sqrt{n} - 1)$):



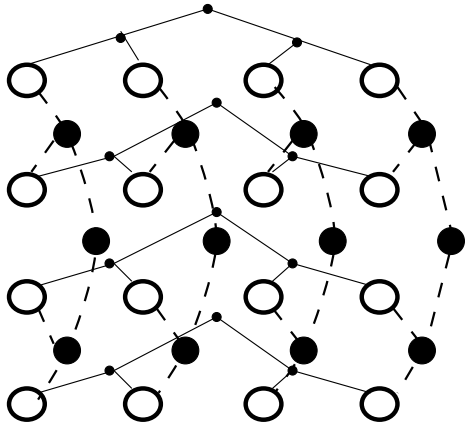
- **Toroid** ($d = 4,\ D = 2 \cdot \left\lfloor \frac{\sqrt{n}}{2} \right\rfloor$):



- **Tree** ($d = 3,\ D = 2\lg(n+1) - 2$):



- **Mesh of trees** ($d = 3,\ D = 4 \lg n$):

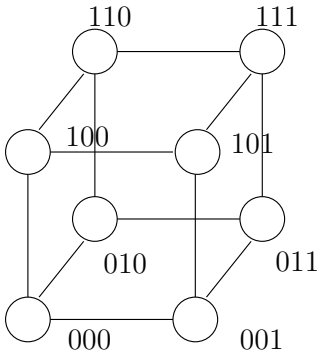


Eine Masche ist ein Array von zusammenhangslosen Knoten und ein Baum, gebildet auf jeder Reihe und Spalte.

Eine Masche von Bäumen (mesh of trees) M_n über einem $n \times n$-Array ($n = 2^k$) hat $3n^2 - 2n$ Knoten und benötigt $\Theta(n^2 \lg^2 n)$ Layout-Raum.

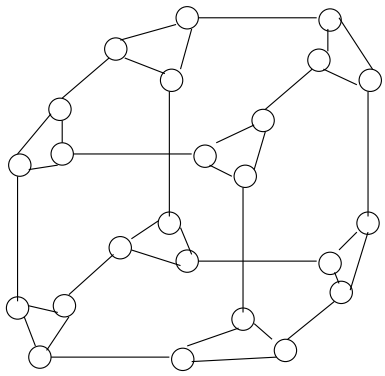
Dies ist eine zweidimensionale Masche von Bäumen, es gibt aber auch mehrdimensionale.

- **Cube** C_m $n = 2^m$ Knoten
  ($d = D = m$):



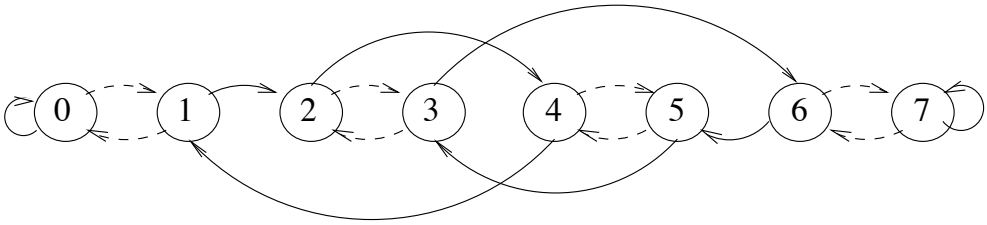
  $C_n = (V, E)$, $V = \{0,1\}^n$,
  $E = \{(u, v) \mid u$ und v unterscheiden sich in einem Bit$\}$

- **Cube connected cycles** CCC_m
  ($d = 3,\ D = 2m$):



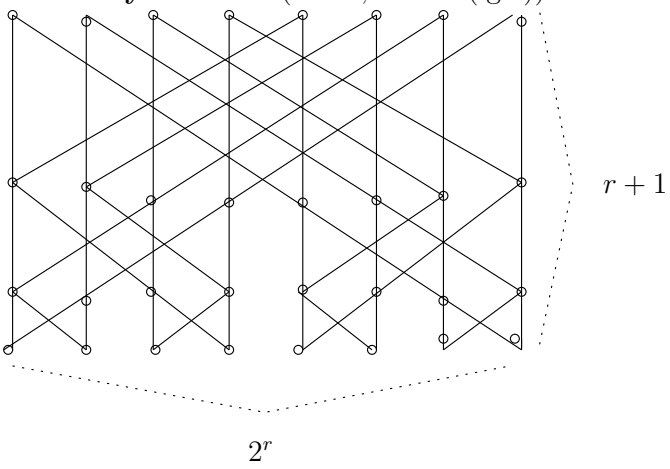
  $n = m2^m$ Knoten – werden aus C_m gewonnen, indem die Eckknoten „aufgeschnitten“ werden

- **Shuffle exchange network** SE_m ($d = 4,\ D = 2 \lg n - 1$):



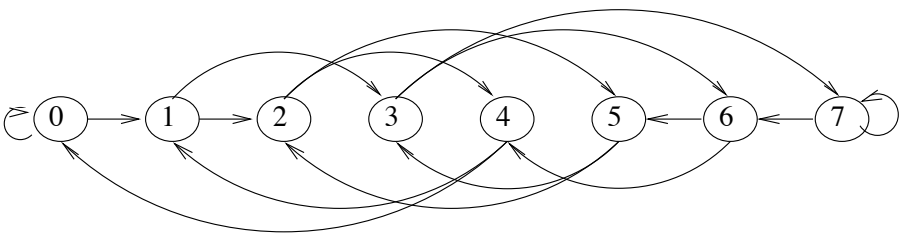
$SE_m = (V_m, E_m)$ hat $n = 2^m$ Knoten in der Menge $V_m = \{0,1\}^m$ und die Kantenmenge $E_m = \{(a_{m-1} \dots a_0 a_{m-1} \dots a_1 \bar{a}_0) \mid a_{m-1} \dots a_0 \in \{0,1\}^m\}$ $\cup$ $\{(a_{m-1} \dots a_0 a_{m-2} \dots a_1 a_0 a_{m-1}) \mid a_{m-1} \dots a_0 \in \{0,1\}^m\}$. Hier bedeutet $\bar{a}_i$, dass das bit a_i negiert wird.

- **Butterfly network** ($d = 4,\ D \in O(\lg n)$):



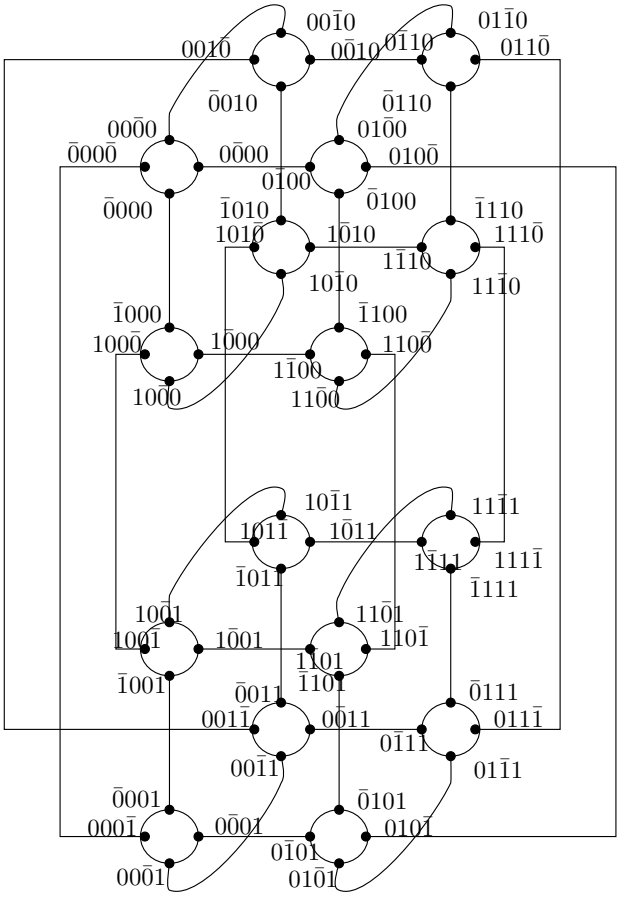
Ein Butterfly-Netzwerk mit Rang r hat $n = (r+1)2^r$ Knoten

- **Perfect shuffle network** ($\equiv$ de Bruijn Graph):



$PS_m = (V, E)$, mit $n = 2^m$, $V = \{i \mid 0 \leq i < 2^m\}$,
$E = \{(i, 2i \bmod 2^m), (i, (2i \bmod 2^m) + 1); i \in V\}$

Das *Cube-connected-cycle-network* CCC_4 der Dimension 4:



Connection rules für CCC_m:

$$a_1a_2\ldots\tilde{a}_{i-1}a_ia_{i+1}\ldots a_m \;-\; a_1a_2\ldots a_{i-1}\;\begin{array}{c}\tilde{a}_i\\|\\\tilde{\bar{a}}_i\end{array}\;a_{i+1}\ldots a_m \;-\; a_1a_2\ldots a_{i-1}a_i\tilde{a}_{i+1}\ldots a_m$$

9.3 Shuffle-Exchange-Netzwerk

m-dimemsiomales Shuffle-Exchange-Netzwerk:

$$\begin{aligned} SE_m &= \langle V_m, E_m\rangle, \qquad V_m = \{0,1\}^m \\ E_m &= \{(a_ma_{m-1}\ldots a_2a_1, a_ma_{m-1}\ldots a_2\bar{a}_1), (a_ma_{m-1}\ldots a_2a_1, a_{m-1}a_{m-2}\ldots a_2a_1a_m)\} \end{aligned}$$

Die Knoten von SE_n kann man mit natürlichen Zahlen in Dezimal- oder Binärdarstellung bezeichnen.

Ein *Shuffle-Exchange-Netzwerk* hat zwei Typen von Verbindungen (Kanten):

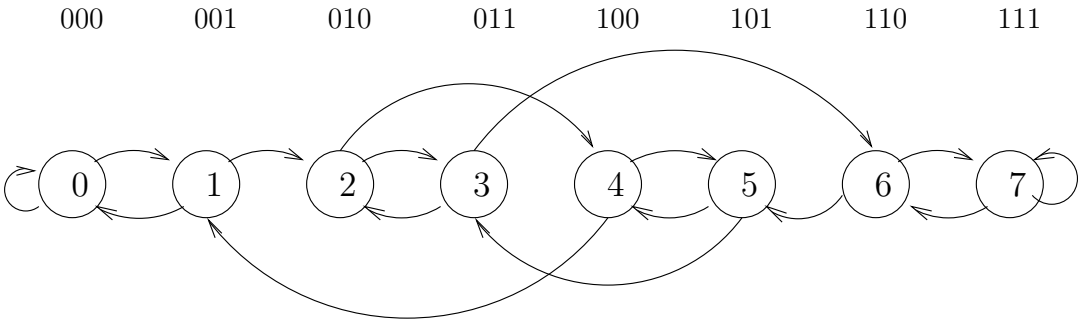


Abbildung 9.1: Shuffle-Exchange-Netzwerk

1. **Exchange-Kanten**:

$$\begin{aligned} Ex(a_m \dots a_2 a_1) &= a_m \dots a_2 \bar{a}_1 \quad \text{“rightmost bit is changed”} \\ \text{oder} \qquad Ex(i) &= \text{if } i \text{ is even then } i+1 \text{ else } i-1 \end{aligned}$$

2. **Perfect-Shuffle-Kanten**:

$$\begin{aligned} PS(a_m a_{m-1} \dots a_1) &= a_{m-1} \dots a_1 a_m \qquad \text{“leftmost bit} \\ &\quad \text{is moved to rightmost position”} \\ PS(i) &= \text{if } i < 2^m - 1 \text{ then } 2i \bmod (2^m - 1) \text{ else } 2^m - 1 \end{aligned}$$

Manchmal betrachtet man SE_m wie einen ungerichteten Graphen. In diesem Fall ist auch die folgende Verbindung (Kante) wichtig:

$$PS^{-1}(a_m \dots a_2 a_1) = a_1 a_m \dots a_2$$

Perfect Shuffle connections received its name from the following fact:
Let us have a deck of playing cards, numbered $0, 1, \dots, m-1$.

Cut the deck exactly in half to get cards $0, 1, \dots, \frac{m}{2} - 1$ in one half, and cards $\frac{m}{2}, \dots, m-1$ in the second half. Let us now shuffle deck perfectly. The resulting deck of cards is

$$0, \frac{m}{2}, 1, \frac{m}{2}+1, 2, \frac{m}{2}+2, \dots, \frac{m}{2}-1, m-1$$

Observe that card i moved into position $PS(i)$!

9.3.1 Speicher-Abbildungen in den Shuffle-Exchange-Netzwerken

Sei N ein Netzwerk von Prozessoren mit Shuffle-Exchange-Verbindungen, das heißt, jeder Prozessor $P_i, 0 \leq i < 2^m$, kann seine Daten zu den Prozessoren $P_{Ex(i)}, P_{PS(i)}$ und $P_{PS^{-1}(i)}$ schicken.

Nehmen wir an, dass der Prozessor P_i am Anfang einer Berechnung die Daten d_i enthält und dass in jedem Schritt entweder

1. jeder Prozessor P_i seine Daten zum Prozessor $P_{Ex(i)}$ schickt oder
2. jeder Prozessor P_i seine Daten zum Prozessor $P_{PS(i)}$ schickt oder
3. jeder Prozessor P_i seine Daten zum Prozessor $P_{PS^{-1}(i)}$ schickt.

Eine Tabelle, die die aktuelle Position für jedes d_i zeigt, heißt *Speicher-Abbildung*. Ein großer Vorteil eines Shuffle-Exchange-Netzwerks ist, dass es sehr einfach ist, diese Speicher-Abbildung zu erhalten. Ein zweiter Vorteil dieses Netzwerks ist, dass sehr schnell verschiedene wichtige Daten-Verschiebungen auszuführen sind.

Zum Beispiel ist nach drei Operationen PS, Ex, PS^{-1} der „Inhalt“ der ersten Hälfte der Prozessoren und der zweiten Hälfte völlig vertauscht:
$PS^{-1}(Ex(PS(a_1a_2\ldots a_m))) = \bar{a}_1a_2\ldots a_m$

i	PS	Ex	PS^{-1}	i	PS	Ex	PS^{-1}
0000	0000	0001	1000	1000	0001	0000	0000
0001	0010	0011	1001	1001	0011	0010	0001
0010	0100	0101	1010	1010	0101	0100	0010
0011	0110	0111	1011	1011	0111	0110	0011
0100	1000	1001	1100	1100	1001	1000	0100
0101	1010	1011	1101	1101	1011	1010	0101
0110	1100	1101	1110	1110	1101	1100	0110
0111	1110	1111	1111	1111	1111	1110	0111

9.3.2 Routing in den Shuffle-Exchange-Netzwerken

Das *Packet-Routing-Problem* in der einfachsten Form ist das Problem, ein Paket von einem Prozessor P_u zum Prozessor P_v durch die Verbindungen (Kanten) des Netzwerks und nur mit Hilfe der lokalen Steuerung zu schicken.

Für das Shuffle-Exchange-Netzwerk gibt es eine einfache Routing-Methode, um für ein Paket einen Weg vom Prozessor P_u zum Prozessor P_v zu finden. Um das zu sehen, nehmen wir an, dass gilt

$$u = u_1u_2\ldots u_m, \quad v = v_1v_2\ldots v_m \qquad u_i, v_i \in \{0,1\}, 1 \leq i < m$$

Um einen Weg von u nach v zu bilden, rotieren wir die Bits in u, indem wir Shuffle-Kanten benutzen und ändern Bits, indem wir je nach Bedarf Exchange-Kanten benutzen.

Zum Beispiel kann man einen Weg von u nach v folgendermaßen konstruieren:

$$
\begin{array}{lcl}
u & = & u_1u_2\ldots u_m \overset{PS}{\underset{}{\rightarrow}} u_2u_3\ldots u_mu_1 \\
 & & \overset{Ex?}{\underset{}{\rightarrow}} u_2u_3\ldots u_mv_1 \overset{PS}{\underset{}{\rightarrow}} u_3u_4\ldots u_mv_1u_2 \\
 & & \overset{Ex?}{\underset{}{\rightarrow}} u_3u_4\ldots u_mv_1v_2 \overset{PS}{\underset{}{\rightarrow}} u_4\ldots u_mv_1v_2u_3 \\
 & & \overset{Ex?}{\underset{}{\rightarrow}} u_4\ldots u_mv_1v_2v_3 \\
 & & \vdots\vdots\vdots \\
 & & \ldots \rightarrow u_mv_1\ldots v_{m-1} \overset{PS}{\underset{}{\rightarrow}} v_1\ldots v_{m-1}u_m \\
 & & \overset{Ex}{\underset{}{\rightarrow}} v_1\ldots v_{m-1}v_m = v
\end{array}
$$

Beachte, dass wenn $u_i = v_i$ gilt, man die „Exchange-Verbindung“

$$u_{i+1}\ldots u_mv_1\ldots v_{i-1}u_i \rightarrow^{Ex} u_{i+1}\ldots u_mv_1\ldots v_{i-1}v_i$$

nicht braucht.

Korollar 9.1 $D(SE_m) \leq 2m - [m > 0]$

Beispiel 9.1:

$$
\begin{array}{lcl}
u & = & 10011001 \quad v = 00000000 \\
 & \rightarrow^{Ex} & 10011000 \\
 & \rightarrow^{PS} & 00110001 \\
 & \rightarrow^{Ex} & 00110000 \\
 & \rightarrow^{PS} & 01100000 \\
 & \rightarrow^{PS} & 11000000 \\
 & \rightarrow^{PS} & 10000001 \\
 & \rightarrow^{Ex} & 10000000 \\
 & \rightarrow^{PS} & 00000001 \\
 & \rightarrow^{Ex} & 00000000
\end{array}
$$

Das Beispiel aus Abbildung 9.2 zeigt, dass der Weg, den man mit dieser naiven Methode finden kann, nicht immer der kürzeste ist.

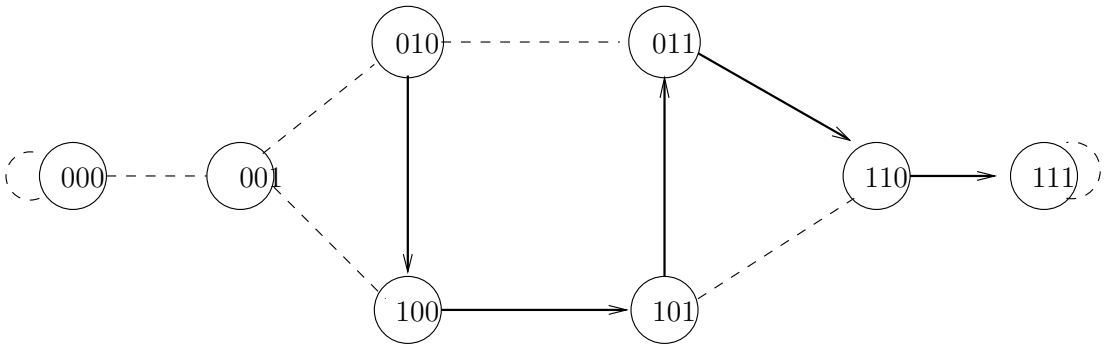


Abbildung 9.2: Der Weg von 010 nach 111, der mit der vorhergehenden Methode konstruiert wird.

9.3.3 Shuffle-Exchange und ein Kartentrick

cards	top half	bottom half	2 shuffle outshuffle	operations inshuffle
000 A	A	E	A	E
001 B	B	F	E	A
010 C	C	G	B	F
011 D	D	H	F	B
100 E			C	G
101 F			G	C
110 G			D	H
111 H			H	D

If a volunteer picks up an arbitrary card from a deck of cards and puts it into shuffled deck into the position u, then the magician can easily determine which sequence of outshuffle and inshuffle operations to perform to get unknown card into the desirable position v. This is basically routing problem for shuffle exchange.

A card from the position $u_1 \dots u_m$ get into the position $u_2 \dots u_m u_1$ ($u_2 \dots u_m \bar{u}_1$) using *outshuffle* (*inshuffle*).

In order to perform this trick, the magician needs

1. to determine exactly the position u,
2. to calculate fast the routing from u to v,
3. to perform perfectly the desired sequence of outshuffle and inshuffle operations.

9.4 Butterfly-Netzwerke

Definition 9.3:
Ein *Butterfly-Netzwerk* B_m besteht aus den Knoten $\{(r,i)|0 \leq r \leq m, 0 \leq i < 2^m\}$, wobei $P_{r,i}$ der i-te Knoten der r-ten Reihe ist. Die Kanten ergeben sich wie folgt: $P_{r,i}$ ist verbunden mit $P_{r-1,i}$ und $P_{r-1,j}$, wobei die Binärdarstellungen von j und i sich im r-ten Bit von links voneinander unterscheiden (vgl. Abbildung 9.3).

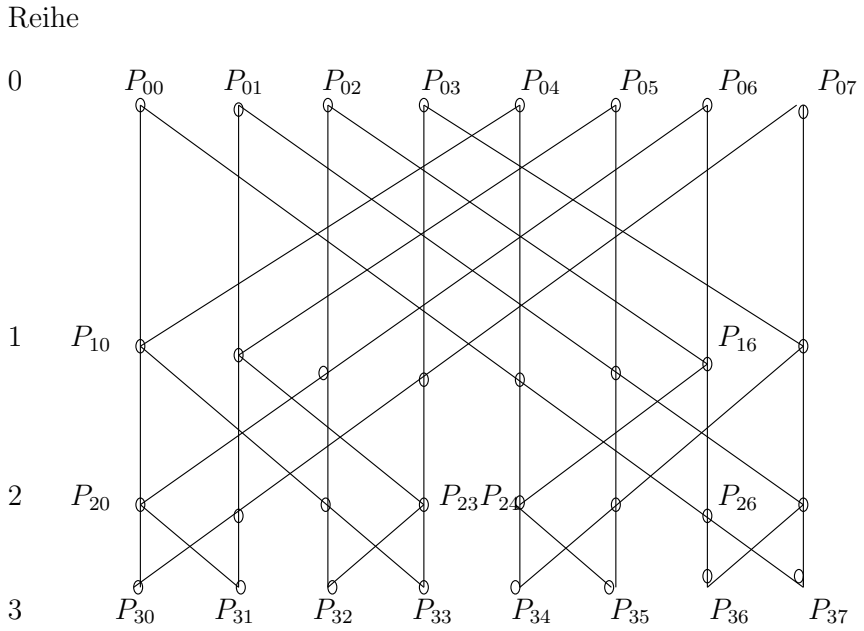


Abbildung 9.3: Ein dreidimensionales Butterfly-Netzwerk

□

Werden in B_m der erste und letzte Knoten in jeder Spalte als identisch betrachtet, dann bekommt man fast genau einen CCC_m (siehe nächste Seite). Die Knoten jeder Spalte in B_m formen dann einen Zyklus in CCC_m.

Butterfly-Netzwerke und Cube-connected-cycles können einander einfach und schnell simulieren.

Die Verbindung von $P_{r,i}$ nach $P_{r-1,i}$ in B_m ist auch eine direkte Verbindung in CCC_m. Die Verbindung von $P_{r,i}$ nach $P_{r-1,j}, i \neq j$, ist in CCC_m durch zwei Verbindungen realisiert: von $P_{r,i}$ nach $P_{r,j}$ und dann nach $P_{r-1,j}$.

Definition 9.4 (Cube Connected Cycle):
Ein *Cube-Connected-Cycle* CCC_m besteht aus den Knoten: $\{(r,i)|0 \leq r \leq$

$m, 0 \leq i < 2^m\}$ und den Kanten nach folgender Regel: $P_{r,i}$ ist verbunden mit $P_{r-1,i}, P_{r+1,i}$ und $P_{r,j}$, wobei i und j sich im r-ten Bit von links voneinander unterscheiden (vgl. Abbildung 9.4).

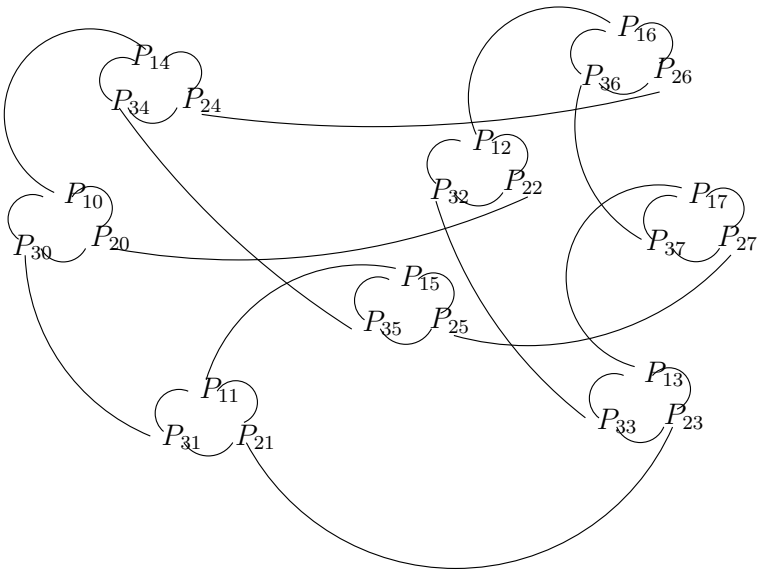


Abbildung 9.4: Ein dreidimensionaler Cube-Connected-Cycle (CCC_3)

□

9.4.1 Simulation von Butterfly-Netzwerken auf Shuffle-Exchange-Netzwerken

Definition 9.5:
Ein Algorithmus für ein Butterfly-Netzwerk heißt *normal*, wenn zu jedem Zeitpunkt alle einschlägigen Daten in den Prozessoren derselben Reihe sind und es zu jedem Zeitpunkt entweder keine Kommunikation zwischen den Prozessoren gibt oder Daten an die Nachbarn der vorigen (nächsten) Reihe übergeben werden. □

Satz 9.2 *Jeder $T(m)$-zeitbeschränkte normale Butterfly-Algorithmus kann durch einen $O(T(m))$-zeitbeschränkten Shuffle-Exchange-Algorithmus simuliert werden.*

Beweis:
Sei A ein normaler Butterfly-Algorithmus. Wir nehmen an, dass die Berechnung von A in den Prozessoren $P_{mi}, 0 \leq i < 2^m$, von B_m startet.

Wir zeigen jetzt, wie A auf SE_m simuliert werden kann, so dass der Prozessor $P_{ri}, 1 \leq r \leq m, 0 \leq i < 2^m$ von B_m durch den Prozessor $S_{PS^r(i)}$ von SE_m simuliert wird.

Wir zeigen hier nur die Simulation von einem Schritt von A, in dem A die Daten von den Prozessoren der Reihe r zu den Prozessoren der Reihe $r-1$ schickt. (Die Simulation der anderen Schritte ist ähnlich.)

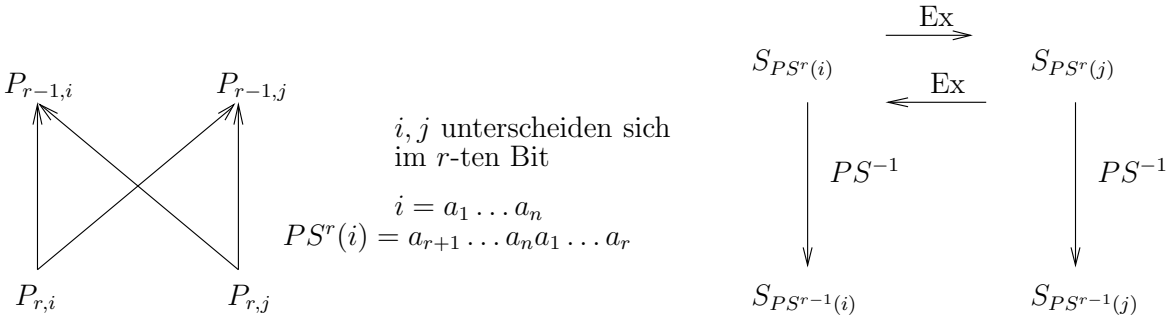


Abbildung 9.5: Zur Simulation eines Butterfly-Netzwerkes auf einem Shuffle-Exchange-Netzwerk

□

9.4.2 Simulation von Butterfly auf dem Hypercube

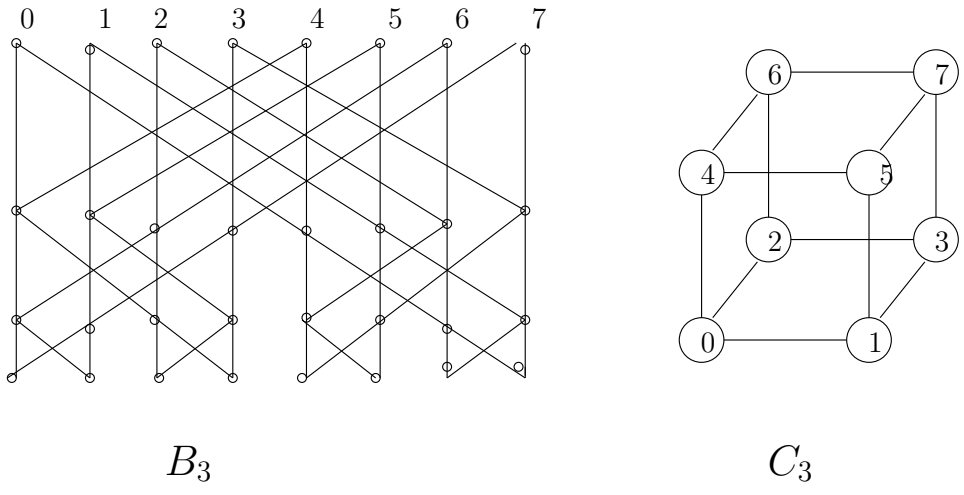


Abbildung 9.6: Zur Simulation eines Butterfly-Netzwerkes auf einem Hypercube

Den Hypercube C_m erhält man aus dem Butterfly-Netzwerk B_m durch Zusammenfassen aller Spalten von B_m.

Daraus folgt:

Satz 9.3 *Jeden T-zeitbeschränkten normalen Butterfly-Algorithmus kann man ohne Zeitaufwand auf dem Hypercube-Netzwerk simulieren (d.h., in Zeit $T(m)$).*

Die folgende Konstruktion zeigt, dass man das Butterfly-Netzwerk in natürlicher Art und Weise aus dem Perfect-Shuffle-Netzwerk bekommen kann. Das bedeutet, dass die Netzwerke Hypercube, Cube-Connected-Cycles, Perfect-Shuffle, Shuffle-Exchange und Butterfly alle eng zusammenhängen.

9.4.3 Sortierung auf dem Butterfly

Beobachte zunächst: Wenn man alle Knoten der Reihe 0 sowie alle ihre Kanten entfernt, dann bekommt man aus einem Butterfly der Stufe m zwei Butterfly-Netzwerke der Stufe $m-1$, wobei die ursprünglichen Reihen $1, 2, \ldots, m$ die Rolle der Reihen $0, 1, \ldots, m-1$ spielen.

Ähnlich, wenn alle Knoten (und ihre Kanten) der letzten Reihe entfernt werden, entstehen zwei isolierte (und überlappende) Butterfly-Netzwerke der Stufe $m-1$ mit jeweils den Spalten mit gradzahligem bzw. ungeradem Index.

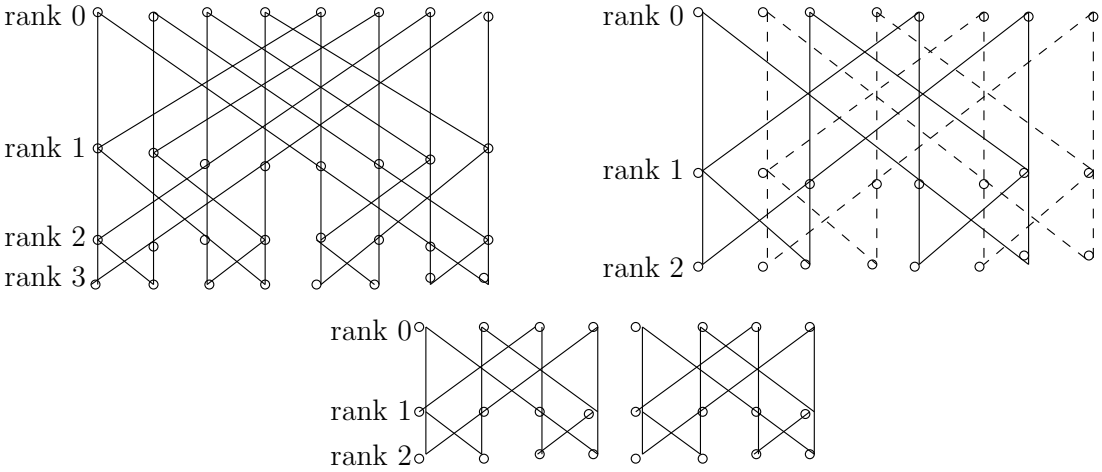


Abbildung 9.7: Aufspalten eines Butterfly-Netzwerkes

Algorithm (to sort $n = 2^k$ elements) Zeit: $O(\log^2 n)$

```
procedure sort(k);
begin
   if k = 0  then done;
   else copy rank 0 into rank 1;
      par [for both butterflies obtained by removing rank 0]
         sort(k-1);
      merge(k)       {using all ranks}
   fi
end
```

Die **Merge-Prozedur** ist auf dem folgenden Lemma begründet:

Lemma 9.4 *Seien $a_1, \ldots, a_n$ und $b_1 \ldots, b_n$ zwei sortierte Listen. Nehmen wir an, dass wir die Listen $a_1, a_3, \ldots, a_{n-1}$ und $b_2, b_4, \ldots, b_n$ mergen, um die Folge $c_1, c_2, \ldots, c_n$ zu bekommen, und dass wir die Listen $a_2, a_4, \ldots, a_n$ und $b_1, b_3, \ldots, b_{n-1}$ mergen, um die Folge $d_1, d_2, \ldots, d_n$ zu bekommen.*

Dann kann man die völlig sortierte Folge bekommen, wenn man zunächst eine Folge der Listen $c_1, c_2, \ldots, c_n$ und $d_1, d_2, \ldots, d_n$ durch Mischen der Elemente bildet:
$c_1, d_1, c_2, d_2, \ldots, c_n, d_n$
und wenn nötig die Elemente in den Paaren (c_i, d_i) vertauscht (???).

```
procedure merge(k);                                          n = 2^k
begin
  if k = 1 then begin d_00 := min{d_10, d_11}, d_01 := max{d_10, d_11}
                        d_10 := d_00, d_11 := d_01 end;
  else begin
            copy-exchange d_{k,0}, ..., d_{k,n/2-1}
            into d_{k-1,0}, ..., d_{k-1,n/2-1}
            copy d_{k,n/2}, ..., d_{k,n-1} into d_{k-1,n/2}, ..., d_{k-1,n-1};
         par [for both butterflies in ranks 0, ..., k-1]
           merge (k-1);
         par[0 ≤ i ≤ n]
         if odd(i) then d_ki := max{d_{k-1,i-1}, d_{k-1,i}}
                   else d_ki := min{d_{k-1,i+1}, d_{k-1,i}}
  end
end
```

9.5 Bisektionsweite von Shuffle-Exchange

Die Bisektionsweite des Netzwerks ist ein kritischer Faktor, von dem es abhängt, wie schnell dieses Netzwerk verschiedene Berechnungen ausführen kann.

d_{ri}-Element speichert in P_{ri}

copy-exchange does
`if` odd(i) `then` $d_{k-1,i} := d_{k,i-1}$
`else` $d_{k-1,i} := d_{k,i+1}$

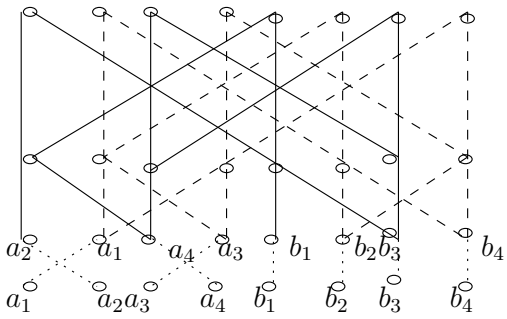


Abbildung 9.8: Merge

Es ist oft nicht einfach, die Bisektionsweite eines Graphen zu bestimmen. Für Shuffle-Exchange gibt es aber einen eleganten Beweis, dass $B(SE_m) = \Theta\left(\frac{2^m}{m}\right)$ ist. Wir zeigen hier aber nur $B(SE_m) = O\left(\frac{2^m}{m}\right)$.

$$\begin{aligned} SE_m &= (V_m, E_m) \\ V_m &= \{0,1\}^m \\ E_m &= \{(u_1u_2\ldots u_{m-1}u_m, u_1u_2\ldots u_{m-1}\bar{u}_m), (u_1u_2\ldots u_{m-1}u_m, u_2u_3\ldots u_mu_1) \\ &\quad |u_1u_2\ldots u_{m-1}u_m \in V_m\} \end{aligned}$$

Sei $\omega_m = e^{\frac{2\pi i}{m}}$ die m-te primitive Wurzel von 1 (d.h., $\omega_m^m = 1$ und $\omega_m^i \neq 1, 1 \leq i < m$).

Gegeben sei ein Knoten

$$u = (u_1u_2\ldots u_m) \in V_m$$

(d.h., $u_i \in \{0,1\}, 1 \leq i \leq m$).

Wir definieren

$$\begin{aligned} \sigma(u) &= u_1\omega_m^{m-1} + u_2\omega_m^{m-2} \\ &\quad + \cdots + u_m \\ &= \sum_{i=1}^{m} u_i\omega_m^{m-i} \end{aligned}$$

Das ist ein Punkt der komplexen Ebene.

9.5.1 Die grundlegenden Eigenschaften der Einbettung von Shuffle-Exchange in die komplexe Ebene

1. Die *Exchange-Kanten* werden zu horizontalen Segmenten der Länge 1. Wirklich
$$\begin{aligned} \sigma(u_1\ldots u_{m-1}1) &= u_1\omega_m^{m-1} + \cdots + u_{m-1}\omega_m + 1 \\ &= \sigma(u_1\ldots u_{m-1}0) + 1 \end{aligned}$$

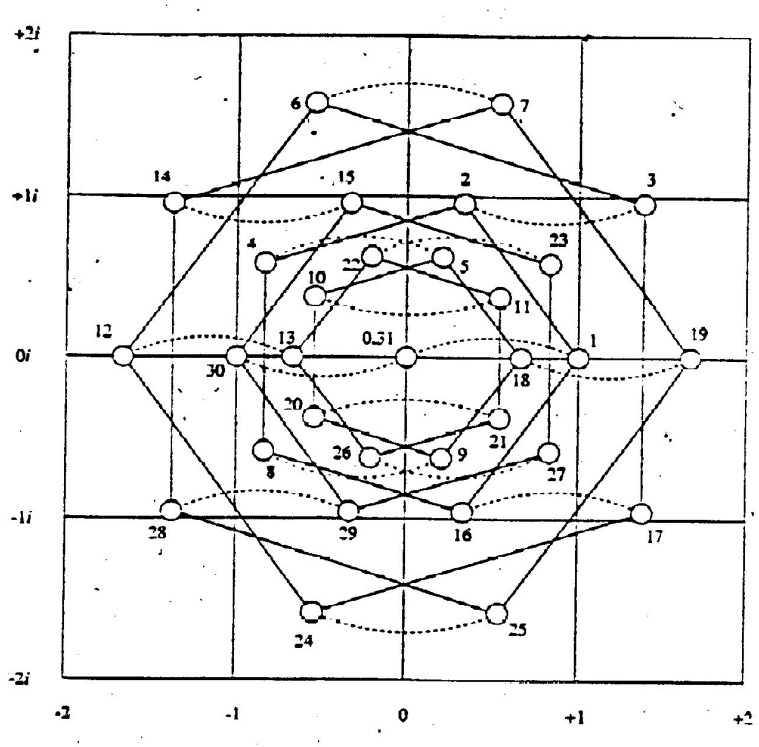


Abbildung 9.9: The complex plane diagram for the 32-node shuffle-exchange graph. Dashed lines denote exchange edges, and solid lines denote shuffle edges. For simplicity, each node is labelled with its value instead of its binary string. (By the *value* of a node, we mean the integer value of the associated binary string. For example, the value of 01101 is 13.)

2. Die *Shuffle-Kanten* erscheinen in den Zyklen (sie heißen *necklaces*), die symmetrisch um den Ursprung angeordnet sind.

$$\begin{aligned} \omega_m \sigma(u_1 u_2 \dots u_m) &= u_1 \omega_m^m + \dots + u_m \omega_m \\ &= u_2 \omega_m^{m-1} + \dots + u_m \omega_m + u_1 \\ &= \sigma(u_2 u_3 \dots u_m u_1) \end{aligned}$$

3. Die *necklaces* haben höchstens die Länge n – das sind *full necklaces*. Die *necklaces*, die weniger als n Knoten besitzen (z.B. 1010 $\rightarrow$ 0101 $\rightarrow$ 1010 für $n = 4$) heißen *degeneriert* und werden von σ auf den Koordinatenursprung abgebildet.

4. Die Anzahl der Knoten in der oberen Hälfte der komplexen Ebene ist die-

selbe wie in der unteren Hälfte der Ebene:

$$\begin{array}{lll} \text{Wenn} \quad u = u_1 \dots u_n & & \bar{u} = \bar{u}_1 \dots \bar{u}_n \\ \\ \text{dann} \quad \sigma(u) + \sigma(\bar{u}) & = & \sum_{i=1}^{n} (u_i + \bar{u}_i)\omega_n^{n-i} \\ & = & \sum_{i=0}^{n-1} \omega_n^i = 0 \end{array}$$

1. Zunächst zeigen wir, dass σ höchstens $O(2^n/n)$ Knoten auf den Koordinatenursprung abbildet.

   (a) Jedem Knoten $u_1 u_2 \dots u_n$, der auf den Koordinatenursprung abgebildet wird, kann man den Knoten $u_1 u_2 \dots u_{n-1}\bar{u}_n$ zuordnen, der entweder auf (-1,0) oder auf (1,0) abgebildet wird.

   (b) Jeder Knoten, der auf (-1,0) oder (1,0) abgebildet wird, muß in einer *full necklace* enthalten sein.

   (c) Da jede *full necklace* n Knoten besitzt, gibt es höchstens $2^n/n$ *full necklaces*. Daraus folgt, dass höchstens $2 \times 2^n/n$ Knoten auf die Punkte (-1,0) und (1,0) abgebildet werden. Deshalb liegen höchstens $2 \times 2^n/n$ Knoten auf dem Koordinatenursprung.

2. Wir zeigen jetzt, dass höchstens $O(2^n/n)$ Kanten die Gerade der reellen Zahlen schneiden.

   Höchstens zwei Kanten von jeder *full necklace* können die reelle Zahlengerade kreuzen, und eine „Exchange-Kante“ kann die reelle Zahlengerade nur kreuzen, wenn ihre beiden Knoten auf der reellen Zahlengerade liegen.

3. Wenn man alle Kanten, die die reelle Zahlengerade kreuzen, und alle Kanten, die auf der reellen Zahlengeraden liegen, entfernt und eine Hälfte der Knoten –auf der reellen Geraden– dem „oberen Teil“ zuordnet und die andere Hälfte dem „unteren Teil“ der Ebene, bekommt man eine „Bisektion“ des Graphen (d.h., man hat $O(2^n/n)$ Kanten entfernt).

9.6 Permutation-Netzwerke

Eine Methode, wie eine PRAM (oder einen „shared-memory-computer“) implementiert werden kann, ist, ein *Permutations-Netzwerk* zu benutzen, das die Prozessoren $P_1 \dots P_n$ und die Speichermoduln $M_1 \dots M_n$ verbindet, so dass jeder Prozessor P_i auf jedes Speichermodul M_j zugreifen kann.

Ein *Permutations-Netzwerk* ist ein Verbindungs-Netzwerk, dessen Elemente *Schalter* (2-state-switches) sind und das jede Permutation zwischen Inputs und Outputs realisieren kann.

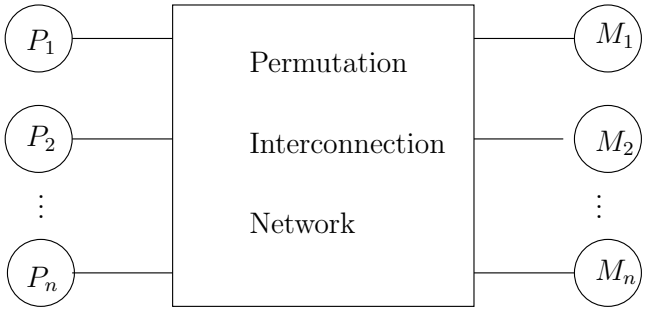


Abbildung 9.10: Permutations-Netzwerk

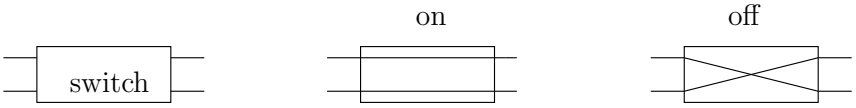


Abbildung 9.11: Die Schalterstellungen in einem Permutations-Netzwerk

Sei $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ eine Permutation. Man sagt, dass ein Netzwerk N π realisiert, wenn es in N eine Schalterstellung gibt, so dass der i-te Input mit dem $\pi(i)$-ten Output verbunden ist.

Das grundlegende Permutations-Netzwerk ist ein $n \times n$-*cross bar switch* („Kreuzschienenverteiler“; vgl. Abbildung 9.12). Auf dem Schnittpunkt der i-ten Input-Linie und der j-ten Output-Linie liegt ein Schalter mit zwei Stellungen.

Kosten: n^2 Schalter.

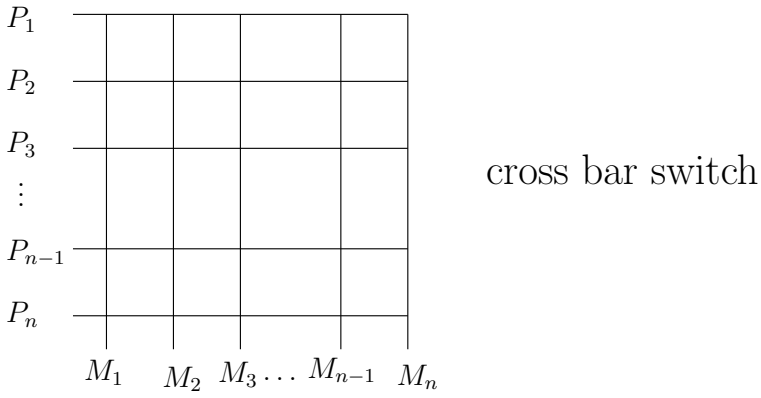


Abbildung 9.12: Kreuzschienenverteiler

Es gibt viele Sorten von Permutations-Netzwerken. Natürlich sucht man Permu-

tations-Netzwerke mit sowenig Schaltern wie möglich (und mit einer regulären Struktur). Es gilt

Satz 9.5 *Jedes Permutations-Netzwerk mit n Ein- und Ausgaben hat $\Omega(n \log n)$ Schalter.*

Beweis:
Wenn ein Permutations-Netzwerk s Schalter hat, dann hat es 2^s Zustände. Da dieses Netzwerk jede Permutation implementieren soll, muß gelten

$$2^s \geq n! \approx \frac{1}{\sqrt{2\pi n}} n^n e^{-(n+0.5)} \quad \rightarrow \quad s \geq n \log n - c_1 n - c_2$$

$$c_1, c_2 \text{ Konstante}$$

□

Natürliche Frage: Gibt es ein $O(n \log n)$-Permutations-Netzwerk (d.h., asymptotisch optimal)?

Die Antwort ist positiv – z.B. hat ein *Beneš-Netzwerk* B_n diese Eigenschaft (vgl. Abbildung 9.13). Für die Anzahl der Schalter $S(n)$ in einem $n \times n$-*Beneš-Netzwerk* gilt:

$$S(n) = 2S\left(\frac{n}{2}\right) + n, \quad S(2) = 1$$

und mit den Methoden von Kapitel 1 kann gezeigt werden, dass

$$S(n) = n \log n - \frac{n}{2}$$

(Beachten Sie, dass ein Beneš-Netzwerk aus zwei „Rücken-an-Rücken"-Butterfly-Netzwerken besteht.)

Der Beweis, dass das Beneš-Netzwerk alle Permutationen realisiert, stützt sich auf folgendes Theorem:

Satz 9.6 (Hall) *Sei S eine endliche Menge und $C = \{A_i \mid 1 \leq i \leq n\}$ eine Familie (nicht notwendig disjunkter) Teilmengen von S, so dass jede Teilfamilie aus k Mengen von C ($k \leq n$), mindestens k Elemente von S enthalte.*

Dann gibt es eine Menge mit n Elementen $\{a_1, \ldots, a_n\}$, wobei $a_i \in A_i$ ist und $a_i \neq a_j$ falls $i \neq j$.

Beweis:
Der Beweis ist eine Induktion nach n für Netzwerke der Größe $n = 2^k$. Der Fall $k = 0$ ist klar.

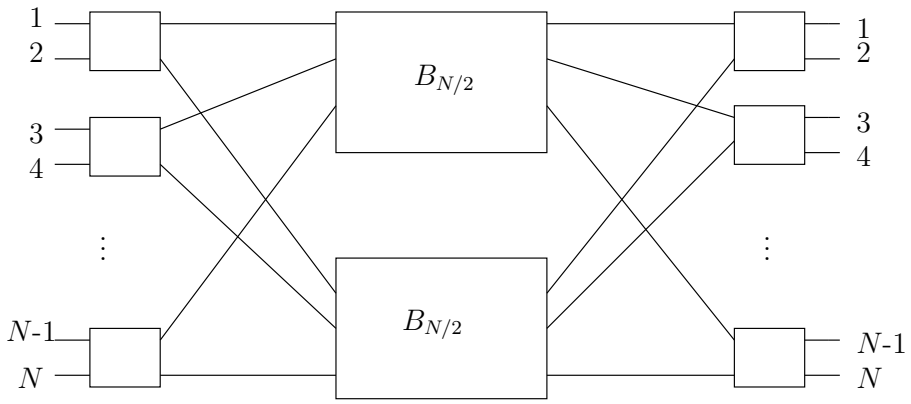


Abbildung 9.13: Eine rekursive Definition eines Beneš-Netzwerks für $n = 2^k$.

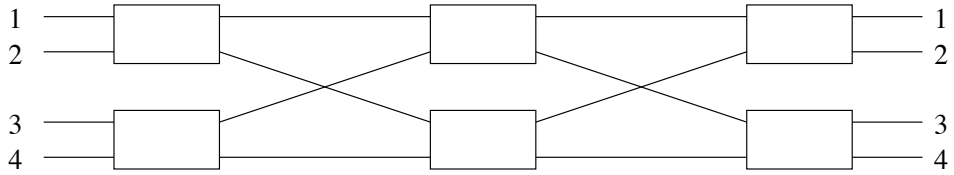


Abbildung 9.14: Beneš-Netzwerk B_4

Sei $\pi : \{1, \ldots n\} \to \{1, \ldots n\}$ eine Permutation.

Für $1 \leq i \leq \frac{n}{2}$ definieren wir

$$A_i = \left\{ \left\lfloor \frac{\pi(2i-1)}{2} \right\rfloor, \left\lfloor \frac{\pi(2i)}{2} \right\rfloor \right\}$$

Dies ist die Menge von Schaltern der letzten Stufe, mit denen die Eingaben des i-ten Schalters der ersten Stufe verbunden werden sollte, wenn die Permutation π realisiert wird. (Beachten Sie, dass jedes A_i eine Menge mit ein oder zwei Elementen ist!)

Sei $A_{i_1}, \ldots A_{i_k}$ irgendeine Teilmenge von k solcher Mengen. Die Vereinigung $\bigcup_{j=1}^{k} A_{i_j}$ enthält Zahlen aller Ausgabe-Schalter, die Werte der Permutation π für $2k$ Eingaben von Schaltern $i_1, \ldots, i_k$ der Eingabe-Stufe enthalten. Da die Anzahl solcher Werte $2k$ beträgt, muß $\bigcup_{j=1}^{k} A_{i_j}$ mindestens k Elemente enthalten.

Die Mengen A_i ($1 \leq i \leq \frac{n}{2}$) erfüllen daher die Annahmen von Hall's Theorem. Demzufolge gibt es eine Menge von $\frac{n}{2}$ verschiedenen Zahlen

$$a_1, \ldots a_{\frac{n}{2}}$$

mit $a_i \in A_i$, so dass a_i die Nummer eines Schalters der Ausgabe-Stufe ist, mit dem der i-te Schalter der Eingabe-Stufe verbunden werden sollte, falls die Permutation π realisiert wird.

Es ist daher möglich, $\frac{n}{2}$ Paare $(i_j, \pi(i_j))$ mit $1 \leq j \leq \frac{n}{2}$ auszuwählen, so dass i_j eine Eingabe des j-ten Eingabe-Schalters ist und die $\pi(i_j)$ von verschiedenen Schaltern der Ausgabe-Stufe kommen. Also können diese $\frac{n}{2}$ Verbindungen realisiert werden, indem der obere Teil der Schalter der mittleren Stufe des Beneš-Netzwerks benutzt wird. Danach reduziert sich das Problem auf den Fall $n_1 = \frac{n}{2}$, und dies kann unter Anwendung der Induktions-Hypothese gelöst werden. □

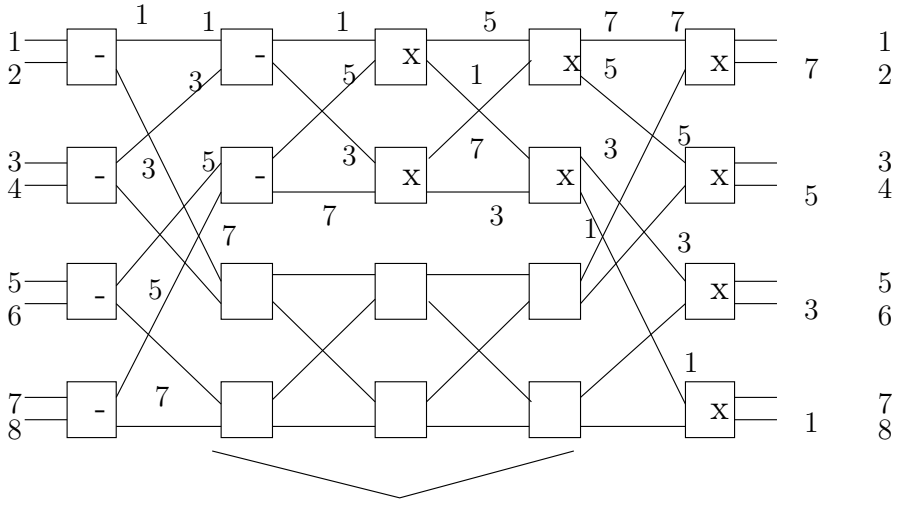
Beispiel 9.2:
Realisierung der Permutation

$$\pi : \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \end{pmatrix}$$

1. Schritt: Wir benutzen den oberen Teil der mittleren Stufe, um die Permutation

$$\begin{matrix} 1 & 3 & 5 & 7 \\ 8 & 6 & 4 & 2 \end{matrix}$$

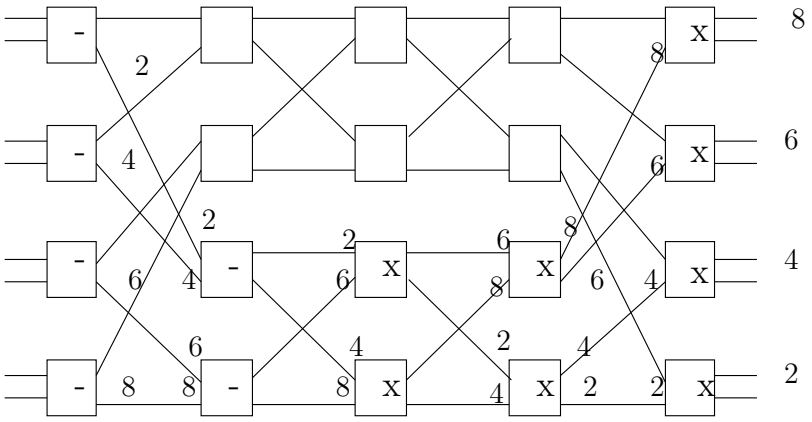
zu realisieren (sie hat eine Eingabe in jedem Schalter der Eingabe-Stufe und eine Ausgabe in jedem Schalter der Ausgabe-Stufe).



2. Schritt: Wir benutzen den unteren Teil der mittleren Stufe, um die Permutation

$$\begin{matrix} 2 & 4 & 6 & 8 \\ 7 & 5 & 3 & 1 \end{matrix}$$

zu realisieren.



9.7 Routing auf Hypercubes

Ein anderes wichtiges Modell für Verbindungs-Netzwerke ist das des *Packet-switching*-Netzwerks.

In solchen Netzwerken schicken Prozessoren *Pakete* – das sind *Nachrichten*, die ein Ersuchen (*request*) enthalten, zu einem Zielknoten zu gelangen, sowie eine *Routing-Information*, die spezifiziert (oder empfiehlt), auf welcher Route durch das Netzwerk das Paket geschickt werden soll. Es gibt mehrere Sorten von *Packet-switching*--Netzwerken.

Permutation-Routing: Jedes Paket hat seine eigene Zieladresse, die von der anderer Pakete verschieden ist.

Partielles Routing: Die Zieladressen aller Pakete sind verschieden, aber nicht alle möglichen Ziele sind auch tatsächlich Adressen von Paketen. Die Zahl der möglichen Ziele ist größer als die der aufgesuchten.

Many-One-Routing: Die Ziele der Pakete brauchen nicht unterschiedlich zu sein.

Satz 9.7 *Es gibt ein Packet-switching-Netzwerk, auf dem man jedes Permutation Routing in* $O(\log^2 n)$ *Zeit ausführen kann.*

Beweis:
Nehmen wir an, dass jedes Paket sein Ziel wie eine Routing-Information mit

sich trägt. Sortieren wir die Pakete nach ihrem Ziel, so dass das Paket mit dem „kleinsten Reiseziel“ nach Knoten 0 befördert wird, das nächste zum Knoten 1, usw.

Da es um Permutation-Routing geht, ist Ziel i genau der i-te Speicher. Deshalb bewirkt eine *Sortierung* der Pakete ein korrektes Routing. Wie wir bereits wissen, kann die Sortierung in $O(\log^2 n)$ Zeit auf dem Butterfly-Netzwerk (und auch auf dem Hypercube, Shuffle-Exchange,...) implementiert werden! □

Die Grundidee des *Partiellen Routing* ist es, in jedem Prozessor ein Paket abzusenden und das Paket dann seinem Weg folgen zu lassen. Die Schwierigkeiten bestehen darin, dass zu viele Pakete gleichzeitig versuchen könnten, denselben Knoten oder dieselbe Verbindung (Kante) zu benutzen.

Beispiel 9.3:

Betrachten wir einen 10-dimensionalen Hypercube (mit $2^{10} = 1024$ Knoten). Nehmen wir an, dass eine *left-to-right*-Routingstrategie benutzt wird.

Wenn ein Paket mit dieser Strategie vom Knoten $a_1a_2 \dots a_n$ zum Knoten $b_1b_2 \dots b_n$ geschickt werden soll, dann geht es zunächst vom Knoten $a_1a_2 \dots a_n$ zum Knoten $c_1c_2 \dots c_n$, wobei sich $c_1c_2 \dots c_n$ aus $a_1a_2 \dots a_n$ ergibt, indem das am weitesten links stehende Bit a_i, das von b_i verschieden ist, verändert wird. Von $c_1c_2 \dots c_n$ nach $b_1b_2 \dots b_n$ gelangt man mit derselben Strategie.

Beispiel 9.4:

Gesucht wird eine Route von $\overbrace{1011000101}^{A}$ nach $\overbrace{1101010001}^{B}$:

$$\begin{array}{ccccccc} A & & A_1 & & A_2 & & A_3 \\ 1011000101 & & 1111000101 & & 1101000101 & & 1101010101 \\ & \rightarrow & & \rightarrow & & \rightarrow & \downarrow \\ 1\underline{1}01010001 & & 11\underline{0}1010001 & & 1101\underline{1}0001 & & 1101010\underline{0}01 \\ B & & B & & B & & B \end{array}$$

Nehmen wir jetzt an, dass Routing erfordere, dass das Paket von jedem der $2^{10} = 1024$ vorhandenen Knoten $a_1 \dots a_{10}$ genau zum Knoten $a_{10}a_9 \dots a_1$ befördert wird.

Es ist klar dass alle Pakete von den 32 Knoten

$$a_1a_2a_3a_4a_500000$$

versuchen, während der ersten fünf Schritte über Knoten 0000000000 zu gehen, und die Hälfte von ihnen wird versuchen, über die Kante von hier weiter zum Knoten 0000010000 zu gehen. Daraus folgt, dass bei dem Routing $a_1 \dots a_{10} \rightarrow a_{10} \dots a_1$ einige Pakete zum Reisen und Warten mindestens 32 Schritte brauchen, um den Zielknoten zu erreichen, der nur bis zu zehn Schritte entfernt ist!

Trotz dieses deprimierenden Beispiels gibt es einen guten Routing-Algorithmus für Hypercubes, der eine einfache, probabilistische Modifikation der Left-to-right-Routing-Strategie ist.

Probabilistisches Routing für Hypercube (Valiant, Brebner 1981):

1. Für jeden Knoten i, der ein Paket zu einem Zielknoten $d(i)$ schicken möchte, wähle zufällig einen Knoten $z(i)$, schicke das Paket zunächst von i nach $z(i)$ und dann von $z(i)$ nach $d(i)$ und benutze für beide „Reisen“ die *Left-to-right-Routing-Strategie*. (Jedes Paket hat also außer der Endadresse $d(i)$ auch die Adresse $z(i)$ bei sich.)

2. Wenn mehrere Pakete versuchen, über dieselbe Kante zu laufen, dann wird ein Paket zufällig gewählt, und die anderen müssen warten. Pakete, die auf dem Weg zu ihrem zufälligen Ziel sind, haben Vorrang.

Satz 9.8 *Für jedes Partielle Routing beträgt die Wahrscheinlichkeit, dass das probabilistische Routing auf einem d-dimensionalen Hypercube mehr als $8d$ Schritte braucht, weniger als 0.74^d.*

Experimente zeigen, dass gewöhnlich $2d$ Schritte ausreichen. Theoretisch jedoch kann auch ein Deadlock entstehen!

Valiant's probabilistisches Routing ist eine wichtige Routing-Methode für Hypercubes.

9.8 Implementation von PRAMs auf Hypercubes

Die PRAM ist ein Modell von parallelen Computern, das sehr gut dazu geeignet ist, parallele Algorithmen zu konstruieren. Gibt es aber eine effiziente Implementierung von PRAMs? Das ist heutzutage vielleicht das wichtigste Problem des parallelen Computing.

Wir wollen zunächst zeigen, dass es eine Implementation der CREW PRAM auf dem Hypercube gibt, mit der jedes Problem, das auf einer CREW PRAM mit N Prozessoren in $T(n)$ Zeit gelöst werden kann, auf einem Hypercube-Netzwerk mit N Prozessoren in Zeit

$$T(n) \log N$$

zu lösen ist.

Um das zu zeigen, muß man die folgenden Probleme lösen:

Problem 1: Wie läßt sich das *gleichzeitige Lesen* in $\log N$ Zeit implementieren?

Problem 2: Wie kann man ein *Many-One-Routing* in $\log N$ Zeit implementieren?

Wichtige Teilprobleme:

- Das Sortieren auf Hypercubes ist in Zeit $\log N$ auszuführen. (Dies läßt sich mit Valiant's probabilistischem Routing-Algorithmus machen.)
- Das Partielle Routing ist in $\log N$ Zeit auszuführen. (Auch das geht mit Valiant's probabilistischem Routing-Algorithmus.)
- Daten-Distribution ist in $\log N$ Zeit auszuführen.

9.8.1 Daten-Distribution

Das folgende Problem ist wichtig für die Implementierung von CREW PRAMs auf Hypercubes.

Seien $0, 1, \ldots, n-1$ Knoten. Seien einige von ihnen *Leiter*. Nehmen wir an, dass die Leiter Daten besitzen und sie diese Daten zu allen Knoten mit höherer Nummer bis zum nächsten Leiter senden sollen.

Beispiel 9.5:

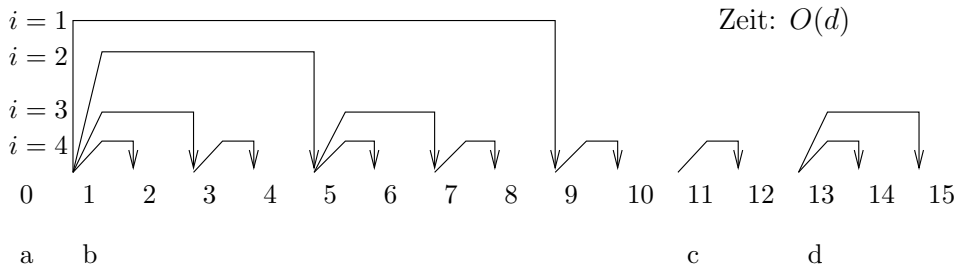
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Daten:	a	b										c		d		

Leiter: $\{0, 1, 11, 13\}$

Der folgende Algorithmus setzt voraus, dass es eine Kante von jedem Knoten j nach $j + 2^i$ gibt, wenn $j + 2^i < n = 2^d$, wobei d die Dimension des Hypercube ist.

begin { initially, only leaders are "alive"}
  for $i := 1$ **to** d **do**
    par $[0 \leq j < n]$ **do**
      if node j is alive
      then pass data and leader number from node j to $j + 2^{d-i}$;
        if the leader number at $j + 2^{d-i}$
          agrees with the one passed
        then wake node $j + 2^{d-i}$ up;
          store the data in the node

fi **fi**
od **od**
end { It is assumed that all nodes know their leaders }



Lemma 9.9 *Der Daten-Distributions-Algorithmus verteilt die Daten von jedem Leiter zu allen seinen Nachfolgern.*

Beweis:

Wir zeigen (durch Induktion), dass die folgende Behauptung für die Berechnung gilt, die der Daten-Distributions-Algorithmus ausführt.

Wenn $k < l$ zwei Knoten mit demselben Leiter sind, und k vor Phase i „lebendig“ wird, wobei i in der Binärdarstellung von $l - k$ die Position der am weitesten links stehenden 1 ist, dann schickt k seine Daten zu l. ($k + 2^{d-i} \leq l < k + 2^{d-i+1}$)

(Beachten Sie, dass in der Phase i ein Knoten versucht, die Daten an den Knoten mit der Distanz 2^{d-i} zu schicken.)

Der Beweis erfolgt durch Induktion über die Anzahl der Einsen in der Binärdarstellung von $l - k$.

1. Wenn die Anzahl der Einsen in $l - k$ 1 ist, dann schickt k in der i-ten Phase Daten genau zu l, weil gilt $l = k + 2^{d-i}$.
2. Induktionsschritt: Nehmen wir an, dass $l - k$ mehr als eine 1 enthält und dass die am weitesten links stehende 1 in der i-ten Position ist. Dann wird k seine Daten in der i-ten Phase zum Knoten $m = k + 2^{d-i} < l$ schicken.

   Jetzt haben die Knoten m und l denselben Leiter, $m < l$, und $l - m$ enthält weniger Einsen als $l - k$, und alle diese Einsen stehen rechts von der i-ten Position. Man kann deshalb die Induktionsvoraussetzung für m und l benutzen, um zu zeigen, dass die Daten, die von k nach m geschickt wurden, dann auch von m nach l geschickt werden.

□

Der Daten-Distributions-Algorithmus ist ziemlich einfach, aber es ist nicht klar, ob es möglich ist, diesen Algorithmus in $O(\log n)$ Zeit auf dem Hypercube oder Butterfly zu implementieren.

Betrachten wir zuerst eine Implementierung des Daten-Distributions-Algorithmus auf einem Butterfly-Netzwerk.

Das Hauptproblem: Ist es stets möglich, die Daten vom Knoten j
nach $j + 2^i,\ 0 \leq i < d$,
in $O(\log n)$ Zeit zu schicken?

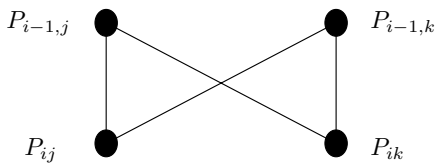
Es ist nicht klar, ob das möglich ist. Außerdem: selbst wenn es möglich wäre, scheint es nur einen $O(\log^2 n)$ Algorithmus zu geben, weil der Außenzyklus die Länge $O(\log n)$ hat. Sogar das ist nicht offensichtlich, weil es nicht klar ist, ob man das Routing so ausführen kann, dass es keine unbeschränkte Stockung während des Routings gibt!

Man kann aber zeigen, dass man mit einer natürlichen Routing-Strategie und ihrer cleveren Implementierung eine $O(\log n)$-zeitbeschränkte Implementierung des Daten-Distributions-Algorithmus auf dem Hypercube bekommen kann (wobei zu jedem Zeitpunkt maximal zwei Datenpakete versuchen, dieselbe Kante zu benutzen).

Zunächst zeigen wir die Implementierung des Daten-Distributions-Algorithmus auf dem Butterfly.

Zugrundeliegende Routing-Strategie:
Die i-te Iteration der äußeren Schleife wird implementiert, indem man Daten von Prozessoren des i-ten Ranges des *Butterfly* aussendet – d.h., die Distribution der Daten von Prozessor P_j an $P_{j+2^{d-i}}$ wird bei Prozessor P_{ij} des *Butterfly* starten.

Die *Butterfly*-Verbindungen:



j und k unterscheiden sich nur im i-ten Bit;
bei j steht eine 0, bei k eine 1 im i-ten Bit;
$n = 2^d$, $\left(2^{d-i}\right)_b = 0^{i-1}10^{d-i}$

Nehmen wir an, wir möchten Daten von P_j nach $P_{j+2^{d-i}}$ senden, und wir betrachten Prozessor P_{ij} des Butterfly. Sei $j = a_1 \dots a_d$.

Fall 1: $a_i = 0$

Dann unterscheiden sich j und $j + 2^{d-i}$ nur im i-ten Bit von links, und deshalb können wir, um Daten zu senden, die Butterfly-Verbindung zwischen P_{ij} und $P_{i-1,j+2^{d-i}}$ benutzen und dann die Verbindungen nach $P_{i,j+2^{d-i}}$ und $P_{i+1,j+2^{d-i}}$, um dort die nächste Iteration zu beginnen.

Fall 2: $a_i = 1$

Dann gilt $a_1 \dots a_i = w01^k$ für irgendein w und $k > 0$. Dann schicken wir Daten zu dem Knoten, der $P_{j+2^{d-i}}$ repräsentiert, und zwar entlang der folgenden Sequenz von Knoten:

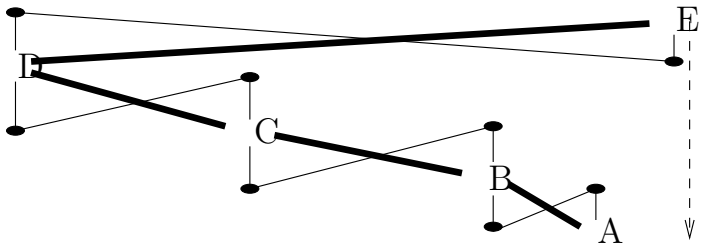
$$j = w01^k x, w01^{k-1}0x, w01^{k-2}00x, \dots, w0^{k+1}x, w10^k x = j + 2^{d-i}$$

Die beiden ersten Wörter unterscheiden sich im i-ten Bit, das zweite und dritte Wort unterscheiden sich im $(i-1)$-ten Bit, das dritte und vierte im $(i-2)$-ten Bit, und deshalb ist es kein Problem, dafür Butterfly-Verbindungen zu benutzen!

Auf diese Weise gelangen wir vom j-ten Knoten in Rang i zum $j + 2^{d-i}$-ten Knoten in Rang $i - k$, und wir müssen dann Spalten-Verbindungen benutzen, um zum Rang $i + 1$ zu gelangen.

Achtung: Im zweiten Fall schicken wir die Daten zuerst nach links, um einen Betrag, der sich bei jeder Bewegung verdoppelt, bis wir auf eine Null in der Binärdarstellung treffen – dann senden wir die Daten nach rechts um den Betrag, der die Summe der linkswärtigen Bewegung um exakt 2^{d-i} überschreitet.

Die folgende Zeichnung illustriert den Pfad der Daten, genannt *trajectory* („Flugbahn“), durch den Butterfly. Diese Bahn startet im Knoten A und erstreckt sich bis E – danach sollte ein „Spalten-Shift“ starten.



So können wir jedes Senden von Daten in $O(\log n)$ Zeit implementieren. Dies garantiert jedoch noch keinen $O(\log^2 n)$-Algorithmus, weil wir dafür erst einmal verifizieren müssen, dass es eine obere Grenze für die Anzahl der Pakete an jedem Knoten zu jeder Zeit gibt.

Wir werden folgendes zeigen: Obgleich unsere Implementation des Daten-Distributions-Algorithmus auf dem Butterfly kein normaler Algorithmus sein wird, ist es möglich, ihn auf einem Hypercube zu implementieren, und zwar so, dass zu jeder Zeit höchstens ein Paket auf einer Kante entlangreist.

Der nächste Schritt besteht darin zu zeigen, dass wir mit einer sorgfältig ausgewählten Choreographie-Strategie erreichen können, dass die Datenübertragungen (Paketübermittlungen) alle Iterationen in $O(\log n)$ Schritten beenden. Des

weiteren zeigen wir, dass wir dies auch auf dem Hypercube können – so dass es also auf dem Butterfly nicht nur eine obere Grenze der Anzahl von Paketen in jedem Knoten zu jedem Zeitpunkt gibt, sondern auch eine Schranke der Anzahl von Paketen *in jeder Spalte* zu jedem Zeitpunkt.

Schlüssel-Idee: Der Absendezeitpunkt der Daten vom j-ten Knoten während der i-ten Iteration ist

$$time(j,i) = 3r + 4s$$

wobei r die Anzahl der Einsen unter den ersten $i-1$ Bits von j angibt und s die Anzahl Nullen in den ersten $i-1$ Bits von j.

Beispiel 9.6:
$time(j,1) = 0$ für jedes j. Dies impliziert zu recht, dass wir den Start der Pakete für die erste Iteration zum Zeitpunkt 0 beginnen können.

$$time(38,5) = time(\underline{10011}0,5) = 3*2+4*2 = 14$$

Um unser Hauptziel zu beweisen, müssen wir zuerst zwei Dinge beweisen:

1. Ist in der i-ten Iteration des Daten-Distributions-Algorithmus Knoten i lebendig, dann empfängt er vor dem Zeitpunkt $time(j,i)$ Daten, die er während der i-ten Iteration übertragen muß.

2. Wird ein Butterfly von einem Hypercube repräsentiert, d.h., Spalten sind zu einem einzigen Knoten zusammengefaßt, dann gibt es niemals mehr als zwei Pakete in einem Knoten.

Diese beiden Tatsachen werden in den folgenden zwei Lemmata bewiesen.

Die folgenden Lemmata beziehen sich auf Fälle, in denen die Knoten wie im Hypercube organisiert sind – und deshalb die Zeit, eine (Flug-)Bahn (trajectory) zu traversieren, gleich der Anzahl der „Seitwärts-Sprünge“ ist. (Wir zählen nicht die Bewegungen innerhalb einer Spalte, da eine Butterfly-Spalte einem einzigen Knoten im Hypercube entspricht.) Eine einfache Modifikation der Zeitfunktion genügt um zu zeigen, dass der Algorithmus in $O(\log n)$ Zeit auch auf dem Butterfly implementiert werden kann.

Lemma 9.10 *Nehmen wir an, dass die Zeit, eine Bahn (*trajectory*) zu traversieren, gleich der Anzahl der Seitwärts-Sprünge in diesem* trajectory *ist. Dann kommt in jeder Iterationsphase i und jedem Knoten j das Paket, das Knoten j in der Phase $i-1$ empfangen hat, vor der Zeit $time(j,i)$ an.*

Beweis:

Fall 1: $j = 0^{i-1}x,\ |x| = d - i + 1$. Dann gilt $j < 2^{d-i+1}$, und weil in der Phase $i-1$ von jedem Prozessor Daten verteilt werden an denjenigen Prozessor, der in Entfernung $+2^{d-i+1}$ liegt, kann der Prozessor im Knoten j keinerlei Daten empfangen.

Fall 2: $j = w1x,\ |w| = i - 2$. Dann kommt in Phase $i-1$ das Paket von dem Knoten mit der Nummer $w0x$. Die Zeitpunkt, zu dem dieses Paket abgesandt wird, ist $time(w0x, i-1)$. Der Absendezeitpunkt dieses Paket im Knoten j in der Phase i ist $time(w1x, i) = time(w0x, i-1) + 3$. Allerdings dauert es nur eine Zeiteinheit, um vom Knoten $w0x$ nach $w1x = j$ zu gelangen, und deshalb trifft das Paket rechtzeitig ein.

Fall 3: $j = w10^kx,\ |w10^k| = i - 1,\ k \geq 1$. Dann kommt in Phase $i-1$ das Paket vom Knoten $w01^kx$. Es gilt:

$$time(w10^kx, i) = time(w01^kx, i-1) + k + 2$$

Um von Knoten $w01^kx$ zum Knoten $w10^kx$ zu gelangen, braucht ein Paket genau $k+1$ Sprünge. So kommt das Paket gerade rechtzeitig beim Knoten j an für die Absendung von j aus in der i-ten Phase. □

Lemma 9.11 *Werden Pakete abgesandt zu der Zeit, die in der time-Funktion definiert ist, und überqueren sie eine Kante in jeder Zeiteinheit, bis sie ihr Ziel im Hypercube erreichen, dann halten sich niemals mehr als zwei Pakete pro Zeiteinheit in einem Knoten auf, und diese beiden Pakete werden im nächsten Zeitschritt auf verschiedenen Kanten weiterreisen.*

Beweis:

Betrachten wir *trajectories*, die durch einen Knoten hindurchgehen wie es in der obigen Abbildung beim Knoten D der Fall ist. Sei D der j-te Knoten in Rang i.

Nehmen wir nun an, dass ein Paket, das in Phase $i+k,\ k > 1$ abgesandt wurde, beim Knoten j ankommt. Dann muß der Absender des Pakets in Phase $i+k$ Knoten $w1^kx,\ |w| = i$ sein, und $j = w0^kx$.

Der Absendezeitpunkt dieses Pakets ist $time(w1^kx, i+k)$, und es reist k Zeiteinheiten, um rechtzeitig anzukommen,

$$time(w1^kx, i+k) + k = time(w0^kx, i+k)$$

Dies impliziert, dass alle Pakete, die auf ihrer (Flug-)Bahn durch Spalte j reisen, dies zu unterschiedlichen Zeiten tun – denen der Form $time(j, p)$ für verschiedene Werte von p.

Deshalb gibt es zu jeder Zeit höchstens ein Paket, das durch Knoten j läuft, und eines, das abgesandt wird. Sie gehen allerdings in verschiedene Richtungen. □

Satz 9.12 *Der Daten-Distributions-Algorithmus kann auf dem Hypercube in $O(\log n)$ Zeit ausgeführt werden, wobei n die Anzahl der Knoten ist.*

Beweis:
Lemma 9.10 besagt: Falls Pakete nicht verzögert werden, dann implementiert ein schedule der Absendezeiten, die durch die *time*-Funktion gegeben sind, den Daten-Distributions-Algorithmus. Lemma 9.11 bestätigt, dass es keine Verzögerungen gibt. Der letzte Zeitpunkt, zu dem ein Paket abgesandt wird, ist $time(4, d) = 4d - 4$. Deshalb ist die Gesamtzeit $O(\log n)$. □

9.8.2 Eine Many-One-Routing-Implementierung des gleichzeitigen Lesens von CREW PRAMs

Der folgende Algorithmus für das Many-One-Routing benutzt die Algorithmen für das Partielle Routing und für die Daten-Distribution.

Die Hauptidee besteht darin, dass für jeden gleichzeitigen Request für einen Speicher ein Leiter gewählt wird aus dem Kreise der Prozessoren, die zugreifen wollen. Nur dieser Leiter greift auf den Speicher zu und verteilt dann die Daten an alle Prozessoren, die Daten angefordert haben.

Algorithmus für Many-One-Routing:

1. Der Prozessor i versuche den Speicher m_i zu lesen. Er generiert ein Paket, das i, m_i sowie die Adresse der benötigten Daten aus m_i enthält.

2. Sortiere alle Pakete nach dem Wert von m_i, d.h., nach dem Reiseziel. (Danach befinden sich alle Pakete, die an einen Speicher geschickt werden sollen, in aufeinanderfolgenden Knoten. Der Knoten, der m_i enthält, muß jedoch keinen Bezug haben zu dem Knoten, der den Speicher m_i besitzt.)

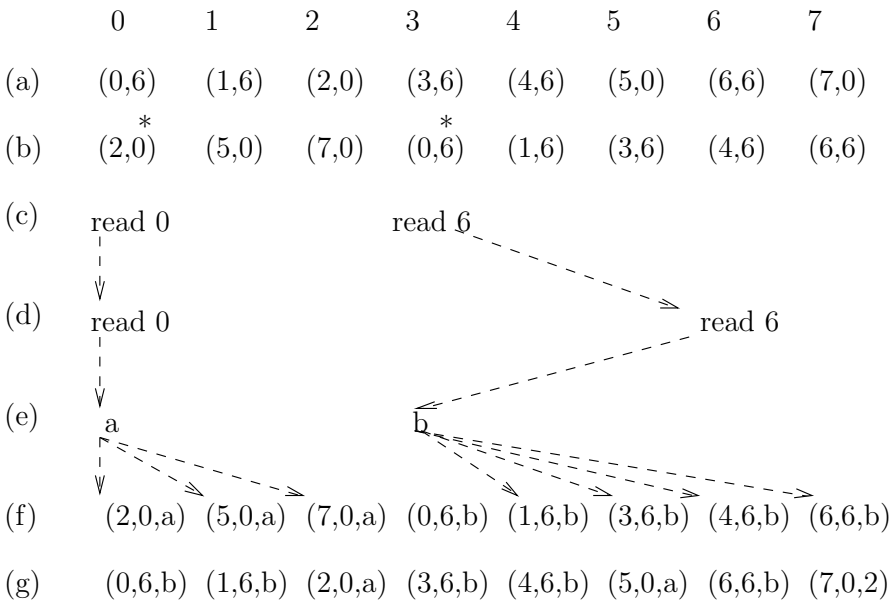
3. Bestimme, welche Knoten Leiter sind. (Dies kann z.B. geschehen, indem alle Knoten ihre Reiseziele zum rechten Nachbarn schicken.) Die Leiter fungieren dann als Vertreter für alle Requests auf einzelne Speicher.

4. Jeder Leiter verschickt dann ein Request-Paket an einen Speicher. Die Speicher schicken die Daten zurück. (Dies wird mit einem Algorithmus für Partielles Routing gelöst.)

5. Die Leiter verteilen die Daten an alle Knoten, deren Leiter sie sind mittels eines Daten-Distributions-Algorithmus.

6. Alle Knoten (Prozessoren) schicken ihre Daten zurück zu den Prozessoren, die die Anfrage gestellt haben (d.h., das Paket (i, m_i) geht zum Knoten i zusammen mit Daten aus m_i).

Beispiel 9.7:
Seien $P_0, P_1, \ldots P_7$ Prozessoren, und nehmen wir an, dass die Prozessoren P_2, P_5, P_7 den Speicher #0 anfordern, und alle anderen Prozessoren den Speicher #6:



(a) – Erzeugung von Requests;
(b) – Sortierung und Wahl der Leiter (bezeichnet mit „$*$“);
(c) – Erzeugen von Requests für Partielles Routing;
(d) – Requests weitergegeben;
(e) – Daten zu den Leitern geschickt;
(f) – Leiter verteilen Daten an ihre Nachfolger;
(g) – Requests und Daten gehen an die Prozessoren, die Daten angefordert haben

Die vorherigen Resultate kann man folgendermaßen zusammenfassen:

Satz 9.13 *Das Many-One-Routing kann auf dem Hypercube in $O(\log n)$ Zeit mit einer Wahrscheinlichkeit, die gegen 1 geht, implementiert werden (wobei n die Anzahl der Knoten des Hypercube ist).*

9.9 Optimale Simulation von PRAMs auf Hypercubes

Parallele Computer und parallele Berechnungen sind ein neues Phänomen in der Informatik, und nur langsam entwickelt sich ein Verständnis dafür, wie man diese Methoden am besten benutzen kann und soll.

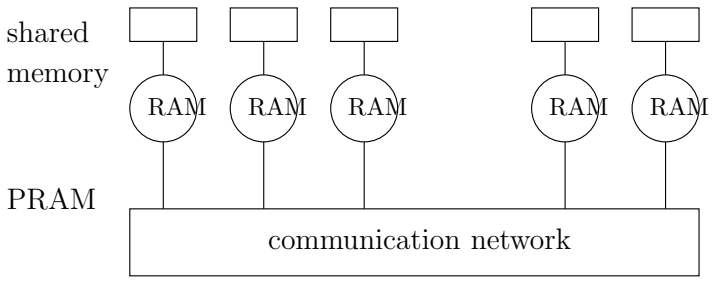
Die erste – naive – Idee war *Parallelisierung*:
Der Mensch entwickelt die Programme für einen Ein-Prozessor-Rechner (d.h., für einen sequentiellen Rechner) und ein Compiler übersetzt sie für einen n-Prozessor-Rechner (d.h., für einen Parallelrechner)

Heutzutage ist es praktisch klar, dass das nicht funktioniert. Eine neue Idee ist *parallel slackness*:
Der Mensch entwickelt die Programme für einen m-Prozessor-Rechner, $m > n \log n$ (d.h., für einen hochgradig parallelen Rechner) und ein Compiler übersetzt sie für einen n-Prozessor-Rechner (d.h., für einen wenig parallelen Rechner)

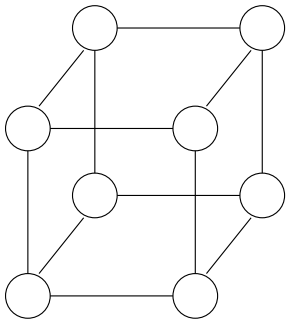
Valiant (1990) hat gezeigt, dass es dann eine optimale (entsprechend des *Zeit-Prozessor-Produkts*) Simulation von PRAMs auf Hypercubes gibt:

Satz 9.14 *Jeder $T/\log N$-zeitbeschränkte Algorithmus für eine $N \log N$-Prozessor-CRCW PRAM kann auf einem N-Prozessor-Hypercube in $O(T)$ Zeit (probabilistisch) simuliert werden.*

Grundidee:
Jeweils $\log N$ Prozessoren von einer $N \log N$-Prozessor-CRCW PRAM



werden durch einen Prozessor des Hypercube simuliert, so dass *ein Schritt der PRAM* durch $\log N$ *Schritte auf dem Hypercube* simuliert werden.



Hashing wird benutzt, um die Daten von den lokalen Speichern der PRAM an die Knoten des Hypercube zu verteilen.

Randomized Routing wird benutzt, um die PRAM-Kommunikationen zu implementieren.

9.10 Einbettungen in Hypercubes

Hypercube-Netzwerke kann man nicht nur dazu benutzen, PRAM-Algorithmen optimal zu implementieren, sondern auch Algorithmen für verschiedene andere grundlegende Netzwerke. Dies ist eine der Konsequenzen der Tatsache, dass es einfache Methoden gibt, verschiedene reguläre Graphen effektiv in den Hypercube-Graphen einzubetten. Diese Einbettungs-Methoden repräsentieren wichtige Programmiermethoden für Hypercube-Netzwerke.

Definition 9.6:
Seien $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ Graphen. Eine *Einbettung* von G_1 in G_2 ist eine injektive Abbildung $f : V_1 \to V_2$.

Folgende Komplexitätsmaße von Einbettungen sind wichtig:

$$\textit{dilation factor} \quad D_f = \max_{x \neq y} \frac{d(f(x), f(y))}{d(x,y)} \qquad x, y \in V$$

$$\textit{expansion factor} \quad E_f = \frac{|V_2|}{|V_1|}$$

Wobei $d(\alpha, \beta)$ den Abstand der Knoten α und β bezeichnet.

□

Im Folgenden betrachten wir eine spezielle Kodierung von natürlichen Zahlen, die für verschiedene Einbettungen in den Hypercube wichtig ist.

Gray-Code:
$G(n)$ bezeichnet die Folge aller n-stelligen binären Wörter in dem *Gray-Code*.

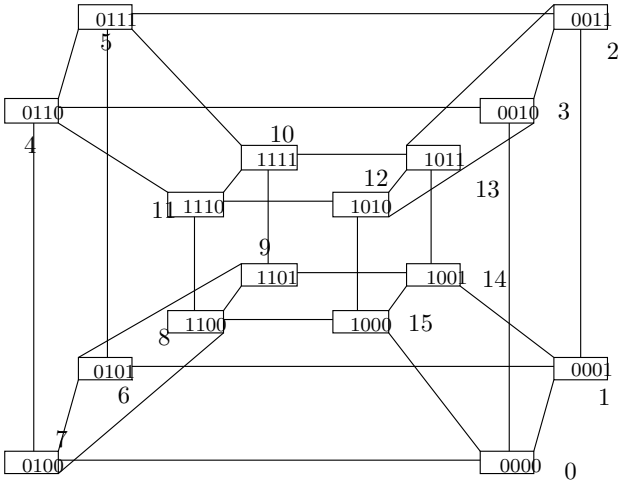


Abbildung 9.15: Eine Einbettung eines 16-Knoten-Rings auf dem Hypercube C_4 mit *dilation 1* und *expansion 1*

$$G(1) = (0, 1)$$

wenn für $n \geq 1$

$$G(n) = (G_0(n), G_1(n), \ldots, G_{2^n-1}(n))$$

dann

$$G(n+1) = (0G_0(n), 0G_1(n), \ldots, 0G_{2^n-1}(n), 1G_{2^n-1}(n), \ldots, 1G_1(n), 1G_0(n))$$

Kodierung – Dekodierung:
Wenn

$$i = i_{n+1} i_n \ldots i_1, \; i_{n+1} = 0$$

und

$$G_i(n) = g_n g_{n-1} \ldots g_1$$

dann

$$g_j = i_j \oplus i_{j+1} \text{ und } i_j = g_{j+1} \oplus \ldots \oplus g_n$$

i	i_b	$G(5)$	i	i_b	$G(5)$
0	00000	00000	16	10000	11000
1	00001	00001	17	10001	11001
2	00010	00011	18	10010	11011
3	00011	00010	19	10011	11010
4	00100	00110	20	10100	11110
5	00101	00111	21	10101	11111
6	00110	00101	22	10110	11101
7	00111	00100	23	10111	11100
8	01000	01100	24	11000	10100
9	01001	01101	25	11001	10101
10	01010	01111	26	11010	10111
11	01011	01110	27	11011	10110
12	01100	01010	28	11100	10010
13	01101	01011	29	11101	10011
14	01110	01001	30	11110	10001
15	01111	01000	31	11111	10000

Eigenschaften:

1. $G_i(n)$ und $G_{i+1}(n),\ 0 \leq i < 2^n - 1$ unterscheiden sich in genau einem Bit.

2. $G_i(n)$ und $G_{2^n-i-1}(n),\ 0 \leq i < 2^n - 1$ unterscheiden sich in genau einem Bit.

Einbettungen von Arrays und Ringen

1. **Einbettung von linearen Arrays**
   mit Knoten (Prozessoren) $P_0, \ldots P_{k-1}$, $k \leq 2^n$
   auf dem Hypercube C_n mit *dilation 1*:
   P_i wird eingebettet in den Knoten von C_n, dessen binäre Darstellung $G_i(n)$ ist.

2. **Einbettung von Ringen**
   mit Knoten (Prozessoren) $P_0, \ldots P_{k-1}$, $k \leq 2^n$
   auf dem Hypercube C_n mit *dilation 1*:

   (a) $k = 2^n$
   P_i wird eingebettet in den Knoten, dessen binäre Darstellung $G_i(n)$ ist. (Eine solche Einbettung ist in der Zeichnung zu Beginn dieses Abschnitts zu sehen.)

   (b) k ist gerade, $m = k/2$
   Der i-te Knoten des Rings wird eingebettet in den i-ten Knoten der Folge $G_{0:m-1}(n), G_{2^n-m:2^n-1}(n)$, wobei

   $$G_{i:j}(n) = G_i(n), \ldots, G_j(n)$$

   Beispiel: $k = 10, N = 16$: Knoten von C_4
   0000, 0001, 0011, 0010, 0110, 1110, 1010, 1011, 1001, 1000

3. **Einbettung von zweidimensionalen Arrays**
   Ein $2^l \times 2^k$-Array mit den Knoten (Prozessoren) $P_{i,j}$,
   $0 \leq i < 2^l,\ 0 \leq j < 2^k$
   Der Prozessor P_{ij} wird eingebettet in den Knoten von C_{l+k}, dessen binäre Darstellung

   $$G_i(l)G_j(k)$$

   ist. *Dilation* dieser Einbettung ist *1* und *expansion 1*.

Einbettung von Bäumen

Zunächst sei ein negatives Resultat wiedergegeben:

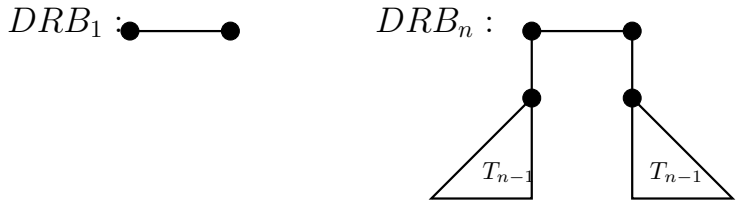
Satz 9.15 *Es gibt keine Einbettung des vollständigen binären Baumes T_n mit $2^n - 1$ Knoten in den Hypercube C_n mit dilation 1.*

Beweis:
Nehmen wir an, dass es eine solche Einbettung gibt, d.h., dass T_n ein Teilgraph von C_n ist. Für einen Knoten $v \in T_n$ sei $\varphi(v) = 0(= 1)$, wenn die binäre Darstellung von v eine gerade (ungerade) Anzahl von Einsen enthält. Es ist offensichtlich, dass genau die Hälfte der Knoten von C_n den φ-Wert 1 hat. Außerdem besitzen alle Knoten derselben Tiefe denselben φ-Wert, der sich von den φ-Werten der Knoten der vorangegangenen und folgenden Tiefe unterscheidet. Daraus folgt, dass mehr als die Hälfte der Knoten von C_n denselben φ-Wert wie die Blätter von T_n haben. Das ist aber ein Widerspruch. □

Andererseits gilt:

Satz 9.16 *Es gibt eine Einbettung eines vollständigen binären Baumes T_n mit $2^n - 1$ Knoten in den Hypercube C_n mit dilation 2.*

Um diesen Satz zu beweisen, genügt es zu zeigen (das geht mit vollständiger Induktion), dass es eine Einbettung des sogenannten „double-rooted“ Baumes DRB_n in C_n mit *dilation 1* gibt, wobei:



Es gibt eine Einbettung eines vollständigen binären Baumes T_n mit $2^n - 1$ Knoten in den Hypercube C_{n+1} mit *dilation 1.*

Satz 9.17 *Ein vollständiger binärer Baum mit $2^n - 1$ Knoten kann in einen C_{n+1}-Cube mit dilation 1 eingebettet werden.*

Beweis:

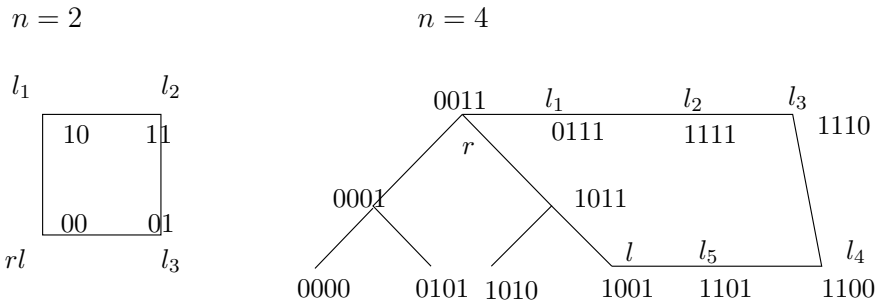
Wir beweisen eine stärkere Aussage, nämlich dass der Graph $G_n = (V_n, E_n)$ eingebettet werden kann in einen 2^n-Cube, wobei

$$V_n = V'_n \cup V''_n \text{ und } E_n = E'_n \cup E''_n$$

(V'_n, E'_n) ist ein vollständiger binärer Baum mit $2^{n-1} - 1$ Knoten.

$$V''_n = \{l_1, \ldots, l_{n+1}\},\ E''_n = \{(l_i, l_{i+1}) \mid 1 \leq i \leq n\} \cup \{(r, l_1), (l_{n+1}, l)\}$$

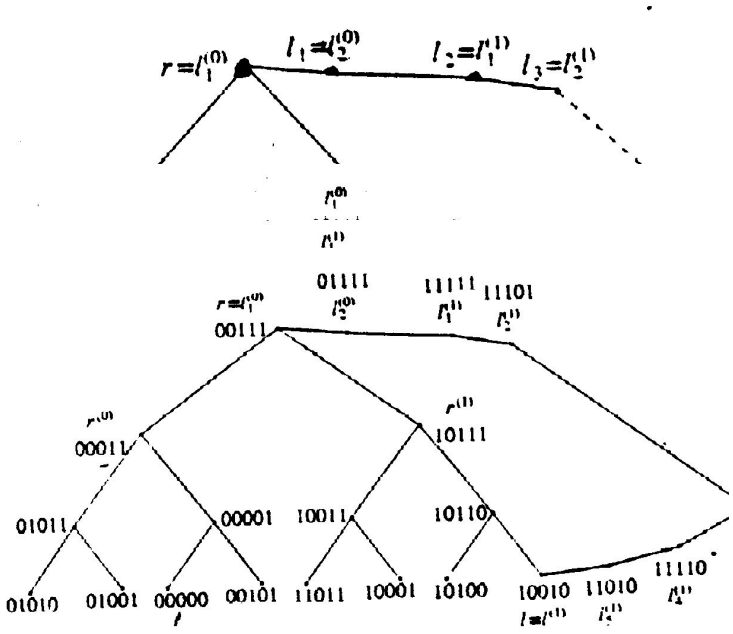
wobei r die Wurzel ist und l das Blatt rechts außen des binären Baumes (V'_n, E'_n).



Bewiesen wird dies durch Induktion. Der Fall $n = 2$ ist klar. Gelte das Theorem für alle $n < m \geq 3$. Betrachten wir den Fall $n = m$. Ein 2^n-Cube kann rekonfiguriert werden in zwei 2^{n-1}-Cubes, die zum Beispiel am führenden Bit entlang projiziert werden.

Betten wir nun G_{n-1} in beide diese
se, dass die Knoten $r^{(1)}$ und $l_1^{(1)}$ N
Einbettung in den zweiten Cube sin

Dies kann geschehen, weil die Einb
der Etikettierung der Knoten im Cu
die neue Wurzel. Nehmen wir die K
Knoten $l_3^{(0)}, \ldots, l_n^{(0)}$ samt daranhäng

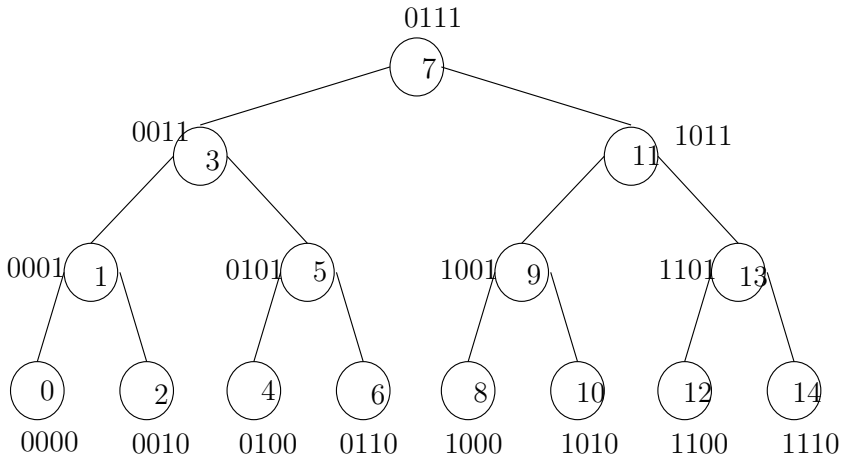


□

Korollar 9.18 *Der größte vollständige binäre Baum, der in einen binären Cube mit dilation 1 eingebettet werden kann, benutzt weniger als die Hälfte der Knoten des Cube.*

Es gibt eine einfache Methode für die Einbettung des vollständigen binären Baunes T_n in den Hypercube C_n mit *dilation 2*:

Numeriere die Knoten von T_n mit der „Inorder-Numerierung“:



Das Problem der Einbettung von anderen Bäumen und Arrays in den Hypercube ist viel komplizierter.

Literaturverzeichnis

[Lei92] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees Hypercubes.* Morgan Kaufmann, San Mateo, Ca. (1992).
[T LEI #17215 KI-Labor / #17714 59156].

[Ull84] J. D. Ullman. *Computational Aspects of VLSI.* Computer Science Press (1984).