Kommunikation in Parallelen Rechenmodellen

(Version 1.4)

Friedhelm Meyer auf der Heide
Heinz-Nixdorf-Institut und Institut für Informatik
Universität Paderborn
Fürstenallee 11
D-33102 Paderborn
E-Mail:fmadh@uni-paderborn.de

13. Juli 2009

Inhaltsverzeichnis

Vorwort

Begleitmaterial zur Vorlesung „Kommunikation in parallelen Rechenmodellen“. Die Ausarbeitung entstand auf der Grundlage von Vorlesungen von Friedhelm Meyer auf der Heide in Frankfurt, Dortmund und Paderborn und wurde von Rolf Wanka unter Mitarbeit von Wolfgang Dittrich, Michael Figge und Petra Haneball erstellt.

Jeder Leser dieses Textes ist aufgerufen, *alle* Arten von Fehlern (inhaltliche, formale, Tipp-) unter Angabe von Seiten- und **Versions**nummer an

Peter Mahlmann
E-Mail: `mahlmann@uni-paderborn.de`

oder

Jan Mehler
E-Mail: `jan.mehler@uni-paderborn.de`

zu schicken.

Kapitel 1

Einleitung

1.1 Parallele Rechenmodelle

Ein **Parallelrechner** besteht aus einer Menge $\mathcal{P} = \{P_0, \ldots, P_{n-1}\}$ von **Prozessoren** und einem **Kommunikationsmechanismus**.
Wir unterscheiden drei Typen von Kommunikationsmechanismen:

a) *Kommunikation über einen gemeinsamen Speicher*
Jeder Prozessor hat Lese- und Schreibzugriff auf die Speicherzellen eines großen **gemeinsamen Speichers (shared memory)**. Einen solchen Rechner nennen wir **shared memory machine**, **Parallele Registermaschine** oder **Parallel Random Access Machine** (**PRAM**). Dieses Modell zeichnet sich dadurch aus, daß es sehr komfortabel programmiert werden kann, da man sich „auf das Wesentliche“ beschränken kann, wenn man Algorithmen entwirft. Auf der anderen Seite ist es jedoch nicht (auf direktem Wege) realisierbar, ohne einen erheblichen Effizienzverlust dabei in Kauf nehmen zu müssen.

b) *Kommunikation über Links*
Hier sind die Prozessoren gemäß einem ungerichteten **Kommunikationsgraphen** $G = (\mathcal{P}, E)$ untereinander zu einem **Prozessornetzwerk** (kurz: **Netzwerk**) verbunden.
Die Kommunikation wird dabei folgendermaßen durchgeführt: Jeder Prozessor P_i hat ein **Lesefenster**, auf das er Schreibzugriff hat. Falls $\{P_i, P_j\} \in E$ ist, darf P_j im Lesefenster von P_i lesen.
P_i hat zudem ein **Schreibfenster**, auf das er Lesezugriff hat. Falls $\{P_i, P_j\} \in E$ ist, darf P_j in das Schreibfenster von P_i schreiben. Zugriffskonflikte (gleichzeitiges Lesen und Schreiben im gleichen Fenster) können auf verschiedene Arten gelöst werden. Wie dies geregelt wird, werden wir, wenn nötig, jeweils spezifizieren. Ebenso werden wir manchmal von der oben beschriebenen Kommunikation abweichen, wenn dadurch die Rechnungen vereinfacht werden können. Auch darauf werden wir jeweils hinweisen.
Die Zahl aller möglichen Leitungen, die es in einem Netzwerk mit n Prozessoren geben kann, ist $\binom{n}{2}$. Dies sind z. B. bei $n = 2^{10} = 1024$ Prozessoren 523776 mögliche Leitungen. Derartiges zu bauen ist darum höchst unrealistisch. In der Praxis baut man deshalb Netzwerke mit konstantem Grad. Ein Beispiel für ein Prozessornetzwerk ist das Transputersystem. Aus den Transputer-Prozessoren der Firma INMOS kann man beliebige Netzwerke vom Grad 4 konfigurieren.

c) *Kommunikation über Busse*
In einem **Busnetzwerk** haben wir zusätzlich zu den „normalen“ Kanten des Netzwerks noch Busse, d. h. Kanten, an denen mehrere Prozessoren hängen. Einer der Prozessoren darf auf diesen Bus schreiben, die anderen dürfen im nachfolgenden Schritt gleichzeitig auf diesem Bus lesen. Auch hier müssen wir wieder Einschränkungen vereinbaren, wenn wir nicht unrealistische Modelle untersuchen wollen. So sollten nicht zu viele Prozessoren an einem Bus hängen, und ein Prozessor sollte umgekehrt nicht an zu vielen Bussen beteiligt sein.

In dieser Vorlesung werden wir von der (z. B. bei Transputer-Systemen nicht gegebenen) Voraussetzung ausgehen, daß die Parallelrechner **synchron** arbeiten, d. h., daß alle Prozessoren den jeweils nächsten Rechenschritt gleichzeitig beginnen.

1.2 Inhalt der Vorlesung

Wir werden uns mit Problemen der Kommunikation zwischen Prozessoren in Netzwerken beschäftigen.

a) *Permutationsrouting, Paralleles Sortieren*
In einem Netzwerk M will jeder Prozessor P_i, $i \in \{0, \ldots, N-1\}$, eine Botschaft zu Prozessor $P_{\pi(i)}$ schicken, wobei π eine Permutation ist. Dies nennt man **Permutationsrouting**.
Wie können zugehörige Kommunikationsprotokolle aussehen, wie effizient kann Routing durchgeführt werden? Eng hiermit verbunden ist das Problem des **Parallelen Sortierens**.

b) *Simulationen zwischen Netzwerken*
Wie effizient kann ein Netzwerk ein anderes Netzwerk simulieren? Wie sehen effiziente universelle Netzwerke aus, also solche, die alle anderen Netzwerke (N Prozessoren, Grad c) simulieren können?

c) *Simulation eines gemeinsamen Speichers*
Wie effizient kann man für PRAMs geschriebene Programme auf Netzwerken simulieren?

Wir werden zu allen drei Punkten verschiedene Lösungen kennenlernen und auch einige untere Schranken für die Effizienz nachweisen.

1.3 Häufig benutzte Notationen

- $\mathbb{N} := \{0, 1, \ldots\}$, $\mathbb{N}^+ := \{1, \ldots\}$, $[n] := \{0, \ldots, n-1\}$.
  $\mathbb{Z}$ bezeichnet die Menge der ganzen Zahlen, $\mathbb{R}$ die Menge der reellen Zahlen.
  $(a, b) := \{x \in \mathbb{R} \mid a < x < b\}$.
- Zu einer Menge X bezeichnet $\mathfrak{P}(X)$ die Potenzmenge von X.
- $\mathrm{bin}_d(n)$ ist die Binärdarstellung von $n \in \mathbb{N}$ aus d Bits, also u. U. mit führenden Nullen, z. B. $\mathrm{bin}_4(3) = 0011$.

- Für $\bar{a} \in [n]^d$ bezeichnet $(\bar{a})_n$ die natürliche Zahl, die durch $\bar{a}$ zur Basis n dargestellt wird, z. B. $(201)_3 = 19$.
- Sei $G = (V, E)$ ein beliebiger Graph. $\Gamma_G(v) := \{u \mid \{u, v\} \in E\}$; $U \subseteq V : \ \Gamma_G(U) := \bigcup_{v \in U} \Gamma_G(v)$.
- $\Gamma_1(U) := \Gamma_G(U)$; $k > 1 : \ \Gamma_k(U) := \Gamma_{k-1}(U) \cup \Gamma_1(\Gamma_{k-1}(U))$. $\Gamma_k(U)$ heißt ***k*-Umgebung** oder auch ***k*-Nachbarschaft** von U.
- Sei G ein Graph und $\mathcal{U}$ eine Teilmenge der Knoten von G. $G\Big|_{\mathcal{U}}$ bezeichnet den Teilgraphen, der entsteht, wenn man alle Knoten, die nicht in $\mathcal{U}$ liegen, mitsamt den dazu inzidenten Kanten in G löscht.

  Zwei Graphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ heißen **isomorph**, wenn es eine bijektive Abbildung $\varphi : V_1 \to V_2$ gibt, so daß gilt: $\{u, v\} \in E_1 \iff \{\varphi(u), \varphi(v)\} \in E_2$. Wir schreiben dann: $G_1 \cong G_2$
- $g(n) = O(f(n)) \iff \exists c > 0 \, \exists n_0 \in I\!N \, \forall n \geq n_0 : g(n) \leq c \cdot f(n)$,
  $g(n) = \Omega(f(n)) \iff f(n) = O(g(n))$,
  $g(n) = \Theta(f(n)) \iff g(n) = O(f(n))$ und $g(n) = \Omega(f(n))$,
  $g(n) = o(f(n)) \iff \lim\limits_{n \to \infty} \dfrac{g(n)}{f(n)} = 0$,
  $g(n) = \omega(f(n)) \iff f(n) = o(g(n))$.
- $e = 2,7182818\ldots$ (Eulersche Zahl).
- $\log n := \log_2 n$, $\operatorname{loglog} n = \log(\log n)$,
  $\log^1 n := \log n$, $\log^i n := \log(\log^{i-1} n)$, $\log^* n := \min\{i \mid \log^i n \leq 2\}$.
  (*Achtung*: Manchmal sieht man auch in Aufsätzen die Schreibweise $\log^k n := (\log n)^k$, die wir *nicht* benutzen. Wenn man in einem Aufsatz eine derartige Schreibweise sieht, muß man sich immer vergewissern, was gemeint ist!)

1.4 Literaturangaben

In diesem Skript geben wir zu jedem Kapitel eine ausführliche Literaturübersicht. Wichtige Arbeiten werden oft auf Konferenzen zum ersten Male der Öffentlichkeit vorgestellt. Später werden diese Arbeiten häufig in überarbeiteter Form in einer Fachzeitschrift veröffentlicht. In unseren Literaturhinweisen geben wir immer die Zeitschriftenversion an, falls diese existiert.
Einige der bedeutenden Konferenzen für die hier untersuchten Gebiete sind:

- ACM Symposium on Theory of Computing (STOC)
- IEEE Foundations of Computer Science (FOCS)
- Symposium on Theoretical Aspects of Computer Science (STACS), Träger: GI & AFCET
- International Conference on Automata, Languages, and Programming (ICALP), Träger: EATCS
- ACM Symposium on Parallel Algorithms and Architectures (SPAA)
- Mathematical Foundations of Computer Science (MFCS)

Einige Zeitschriften mit weiter Verbreitung sind die folgenden:

- SIAM Journal on Computing (SICOMP)
- Journal of the ACM (JACM)
- IEEE Transactions on Computers
- Journal of Computer and System Sciences (JCSS)
- Combinatorica
- Algorithmica
- Theory of Computing Systems (alter Name: Mathematical Systems Theory)
- Acta Informatica
- Information Processing Letters (IPL)
- Journal of Algorithms
- Communications of the ACM (CACM)

Die veranstaltenden Organisationen sind:

- ACM, Association for Computing Machinery;
- IEEE, Institute of Electrical and Electronics Engineers;
- SIAM, Society of Industrial and Applied Mathematics;
- EATCS, European Association of Theoretical Computer Science;
- GI, Gesellschaft für Informatik;
- AFCET, Association Française pour la Cybernétique, Economique et Technique.

In der folgenden Literaturübersicht werden Lehrbücher aufgezählt, die für unser Thema relevante Kapitel enthalten.

1.5 Literatur zu Kapitel 1

[GR88] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, Cambridge, 1988.

[JáJ92] J. JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, Reading, MA, 1992.

[Knu98] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley, Reading, Massachusetts, 2nd edition, 1998.

[Lei92] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[Par87] I. Parberry. *Parallel Complexity Theory.* Pitman/Wiley, London, New York, Toronto, 1987.

[Qui87] M. J. Quinn. *Designing Efficient Algorithms for Parallel Computers.* McGraw-Hill, 1987.

[Qui88] M. J. Quinn. *Algorithmenbau und Parallelcomputer.* McGraw-Hill, 1988.

[Rei93] J. H. Reif, editor. *Synthesis of Parallel Algorithms.* Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[Sav98] J. E. Savage. *Models of Computation: Exploring the Power of Computing.* Addison-Wesley, Reading, 1998.

[vL90] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science: Volume A – Algorithms and Complexity.* Elsevier, Amsterdam, 1990.

Kapitel 2

Permutationsrouting auf (n, d)-Gittern

2.1 Einführung

Die erste Familie von Netzwerken, die wir untersuchen wollen, ist die Familie der Gitter. Das Gitter besitzt eine regelmäßige Struktur und kann daher relativ kostengünstig real gebaut werden, sofern seine Dimension nicht allzu groß ist.

Für diesen Netzwerktypus werden wir einen ersten effizienten Algorithmus für das sog. Permutationsrouting vorstellen. Wir gehen dabei so vor, daß wir die Netzwerkfamilie der Permutierer einführen, die dieses Routing (*to route* = befördern, weiterleiten, den Weg festlegen) sehr schnell durchführen können, und dann zeigen, wie das Gitter das Vorgehen der Permutierer simulieren kann.

Doch zuvor wollen wir die Gitter formal definieren:

2.1 Definition:
Seien $n, d \in \mathbb{N}^+$, $\bar{a} = (a_{d-1}, \ldots, a_0)$, $\bar{b} = (b_{d-1}, \ldots, b_0) \in [n]^d$. Der **Hamming-Abstand** zwischen $\bar{a}$ und $\bar{b}$ ist definiert als $\mathit{hamming}(\bar{a}, \bar{b}) := \sum_{i=0}^{d-1} |a_i - b_i|$.

2.2 Definition:
Seien $n, d \in \mathbb{N}^+$. Das **(n, d)-Gitter** $M(n, d)$ ist das Netzwerk mit Prozessormenge $\mathcal{P} = \{\bar{a} \mid \bar{a} \in [n]^d\}$ und Kommunikationsgraph $G = (\mathcal{P}, E)$, $E = \big\{\{\bar{a}, \bar{b}\} \mid \bar{a}, \bar{b} \in [n]^d,\ \mathit{hamming}(\bar{a}, \bar{b}) = 1\big\}$. Wir nennen dieses Netzwerk auch d-dimensionales Gitter der Kantenlänge n.
Eine Kante zwischen zwei Knoten $\bar{a}$ und $\bar{b}$ mit $|a_i - b_i| = 1$ heißt Kante in **Dimension** i.

2.3 Beispiel:
In Abbildung 2.1 sind $M(n, 1)$, $M(4, 2)$ und $M(2, 3)$ dargestellt. $M(n, 1)$ wird auch **lineares Prozessorarray** der Länge n genannt. Für $M(2, d)$ ist der Name ***d*-dimensionaler Hypercube** gebräuchlich.

2.4 Definition:
Für $G = (V, E)$ und $x, y \in V$ ist $\mathrm{dist}_G(x, y)$ die Länge eines **kürzesten Weges** zwischen x und y, d. h. die Anzahl der Kanten auf diesem Weg. $\mathrm{diam}(G) = \max\{\mathrm{dist}_G(x, y) \mid x, y \in V\}$ ist der **Durchmesser** von G.

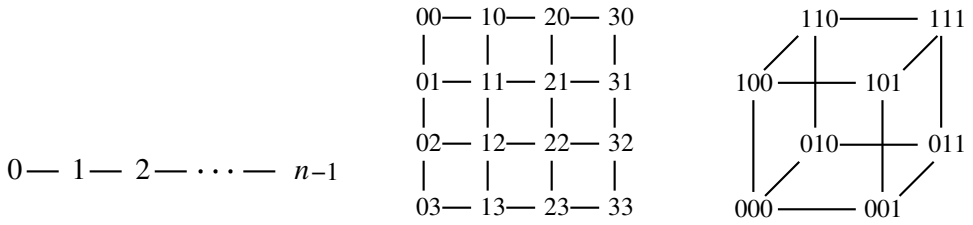


Abbildung 2.1: $M(n,1)$, $M(4,2)$ und $M(2,3)$

2.5 Bemerkung: (Eigenschaften von $M(n,d)$)
Im folgenden sei der betrachtete Graph immer das Gitter $M(n,d)$. Darum lassen wir bei dist den Index $M(n,d)$ weg.

a) *$M(n,d)$ hat n^d Knoten und $dn^d - dn^{d-1}$ Kanten.*

b) *$dist(\bar{a},\bar{b}) = hamming(\bar{a},\bar{b})$*

c) *$diam(M(n,d)) = (n-1)\cdot d$*

d) *$M(n,d)\Big|_{\{\bar{a}|a_i=\ell\}} \cong M(n,d-1)$ für $d>0$, und i,ℓ fest*

e) *$M(n,d)\Big|_{\{i\bar{b}|i\in[n]\}} \cong M(n,1)$ für $\bar{b}\in[n]^{d-1}$ fest*

Die Beweise dieser Aussagen sind dem Leser zur Übung überlassen.
Wir werden nun ein elementares Problem kennenlernen, das für parallele Netzwerke eine der wesentlichen Operationen darstellt. Es besteht darin, eine Menge von Informationen auf den Prozessoren umzuverteilen.

2.6 Definition: (Routing-Protokolle)
Sei $M=(\mathcal{P},E)$, $\mathcal{P}=[N]$, ein Netzwerk, $f:[N]\times[p]\to[N]$ eine Funktion, und seien $x_{0,0},\ldots,x_{0,p-1},x_{1,0},\ldots,x_{1,p-1},\ldots,x_{N-1,0},\ldots,x_{N-1,p-1}$ Nachrichten (die wir auch Botschaften nennen).
Wir sagen: **„M routet $x_{0,0},\ldots,x_{N-1,p-1}$ gemäß f“**, falls gilt: Wenn zu Beginn Prozessor i das Paket $(i,f(i,k),x_{i,k})$ speichert, so ist die Nachricht $x_{i,k}$ am Ende dem Prozessor $f(i,k)$ bekannt. Prozessor i heißt **Quelle**, Prozessor $f(i,k)$ **Senke** der Nachricht $x_{i,k}$. Den Vorgang bezeichnen wir mit **(Funktionen-)Routing**. Ist $p=1$ und beschreibt die Funktion f eine Permutation auf $[N]$, sprechen wir von **Permutationsrouting** und lassen den zweiten Parameter für f weg[1].
Ein **Routing-Protokoll** für M besteht aus Protokollen für alle Prozessoren i. i hat einen **Puffer**, in dem er Pakete aufbewahren kann.
In einem **Routing-Schritt** führt jeder Prozessor i folgendes aus:

- i wählt ein Paket aus seinem Puffer aus;
- i wählt einen seiner Nachbarn j aus;

[1]Beim Permutationsrouting wird die Funktion meist durch einen griechischen Buchstaben bezeichnet.

- i sendet das Paket nach j, d. h. legt es in den Puffer von j und streicht es aus dem eigenen Puffer. Falls der Puffer von j allerdings voll ist, beläßt i das Paket im eigenen Puffer.

Zusätzlich erlauben wir, daß unter Umständen die Pakete durch weitere (kurze) Informationen ergänzt werden, wie z. B. der bisherigen Wartezeit im Puffer oder ähnliches. In diesem Kapitel werden wir das noch nicht benötigen, jedoch wird dies in Kapitel 6.4 ein wichtiges Hilfsmittel darstellen.
M arbeitet **synchronisiert**, d. h. alle Prozessoren $0, \ldots, N-1$ beginnen den t-ten Routing-Schritt gleichzeitig. Die **Routing-Zeit** ist die Anzahl der Routing-Schritte, die **Puffergröße** die maximale Zahl von Paketen, die sich gleichzeitig in einem Puffer befinden.
Unter **Routing mit Preprocessing** (auch: **off-line Routing**) verstehen wir ein Routing-Protokoll, das von f abhängt. Zu Beginn des Routings wird eine Rechnung, eben in Abhängigkeit von f, durchgeführt, die die Protokolle der Prozessoren i erzeugt. Beim **Routing ohne Preprocessing** (auch: **on-line Routing**) hängt das Routing-Protokoll nicht von f ab.

2.7 Satz:
Auf $M(n,1)$ kann Permutationsrouting ohne Preprocessing mit Routing-Zeit $2(n-1)$ und Puffergröße 3 durchgeführt werden.

Beweis:
Sei π eine beliebige Permutation auf $[n]$. Der folgende Algorithmus schickt in einer ersten Phase alle Pakete, deren Zielprozessor links vom Startprozessor ist, ans Ziel, in einer zweiten Phase dann auch alle anderen.

```
for alle Prozessoren i ∈ [n] do
 Phase 1:
   for t := 1 to n − 1 do
     for alle Pakete (j, π(j), x_j) do
       if i ≤ π(j)
         then tue nichts
         else sende das Paket nach Prozessor i − 1
     done
   done
   {Resultat: alle Pakete mit π(j) ≤ j sind am Zielprozessor angekommen.}
 Phase 2:
   for t := 1 to n − 1 do
     for alle Pakete (j, π(j), x_j) do
       if i ≥ π(j)
         then tue nichts
         else sende das Paket nach Prozessor i + 1
     done
   done
   {Resultat: alle Pakete sind am Zielprozessor angekommen.}
done
```

Die Puffergröße ergibt sich folgendermaßen: In Phase 1 ist maximal in Prozessor i das Paket i, das Paket j mit $\pi(j) = i$ und ein weiteres, das weitergereicht werden muß. Analog können wir für die Phase 2 argumentieren.

Die Routingzeit ist dann klar. □

2.8 Beispiel:
Den Algorithmus wollen wir am folgenden Beispiel vorstellen, T bezeichnet dabei die aktuelle Nummer des Routingschrittes.

T	0	1	2	3	4	5	Prozessoren
0	2	3	4	1	5	0	Zieladressen
1	2	3	4,1	–	5,0	–	
2	2	3,**1**	4	0	5	–	
3	2	3,**1**	4,0	–	5	–	
4	2	3,**1**,0	4	–	5	–	
5	2,**0**	3,**1**	4	–	5	–	
6	**0**	2,**1**	3	4	–	**5**	
7	**0**	**1**	**2**	**3**	**4**	**5**	(alle am Ziel)
⋮							

2.2 Permutationsnetzwerke

Unser Ziel ist es ja, auf Gittern effizient Nachrichten zu routen. Dieses Ziel erreichen wir, indem wir ein spezielles Netzwerk konstruieren, das zum einen sehr gut für die Aufgabe des Routings geeignet ist, andererseits aber auch von den Gittern gut simuliert werden kann.

Da die Beweise dieses Abschnitts etwas Graphentheorie benötigen, fügen wir an dieser Stelle einen Exkurs über das ein, was wir später brauchen. Die in diesem Abschnitt betrachteten Graphen sind jeweils ungerichtete Graphen, in denen allerdings Mehrfachkanten erlaubt sind.

2.2.1 Graphentheoretischer Exkurs: Färbungen und Matchings

2.9 Definition:
Ein Graph $G = (V, E)$ heißt r-**regulär**, falls jeder Knoten in G den Grad r hat.

2.10 Definition:
Sei $G = (V, E)$ ein Graph. Eine Abbildung $\psi : E \to [k]$ heißt k-**Kantenfärbung** von G, falls für jedes Paar inzidenter Kanten $e_1, e_2 \in E$ (d. h. $e_1 \cap e_2 \neq \emptyset$ und $e_1 \neq e_2$) gilt: $\psi(e_1) \neq \psi(e_2)$.

2.11 Definition:
Ein **Matching** (Gegenstück, zusammenpassendes Paar) in einem Graphen G ist ein Teilgraph mit maximalem Grad 1. Ein **perfektes Matching** ist ein Matching, in dem jeder Knoten von G Grad 1 besitzt.

2.12 Satz: (Frobenius/Hall, Heiratssatz)
Sei $G = (V_1 \cup V_2, E)$ ein bipartiter Graph mit $|V_1| = |V_2|$. Dann gilt:

$$G \text{ enthält ein perfektes Matching} \iff \forall S \subseteq V_1 : |S| \leq |\Gamma_G(S)| \qquad (*)$$

Beweis:
„⇒“: offensichtlich

„⇐“: Wir führen eine vollständige Induktion nach $m := |V_1| = |V_2|$ durch.
Für $m = 1$ muß nichts gezeigt werden.
Sei nun $m > 1$. Wir unterscheiden zwei Fälle:

Fall 1: Für alle $S \subseteq V_1$, $0 < |S| < m$, gilt: $|\Gamma_G(S)| \geq |S| + 1$
Lege nun eine Kante e des Matchings fest, streiche die Endpunkte von e und alle zu e inzidenten Kanten aus G. Wir bekommen so einen Graphen G' mit $|V_1'| = |V_2'| = m - 1$. Da in G' noch immer die Bedingung $(*)$ der Behauptung gilt, gibt es nach Induktionsvoraussetzung ein perfektes Matching M' in G'. Dann ist $M := M' \cup \{e\}$ ein perfektes Matching in G.
Fall 2: Es gibt eine Menge $S \subseteq V_1$, $0 < |S| < m$, mit: $|\Gamma_G(S)| = |S|$.
Sei G' der Subgraph von G mit der Knotenmenge $S \cup \Gamma_G(S)$. Wir zeigen, daß G' und $G \setminus G'$ die Bedingung $(*)$ der Behauptung erfüllen. Wenden wir uns zuerst G' zu.

Für $S' \subseteq S$ ist $\Gamma_{G'}(S') = \Gamma_G(S')$. Dann ist $|\Gamma_{G'}(S')| = |\Gamma_G(S')| \overset{(*)}{\geq} |S'|$. Nach Induktionsannahme enthält G' ein perfektes Matching M'.
Jetzt behandeln wir noch den Rest des Graphen G, indem wir durch einen Widerspruch zeigen, daß auch dieser die Bedingung $(*)$ der Aussage erfüllt. Sei also $G'' := G \setminus G'$. Angenommen, es gäbe eine Menge $S' \subseteq V_1 \setminus S$ mit $|\Gamma_{G''}(S')| < |S'|$. Dann ist $\Gamma_G(S' \cup S) = \Gamma_G(S') \cup \Gamma_G(S) = \Gamma_{G''}(S') \cup \Gamma_G(S)$. Für die Mächtigkeiten gilt dann:

$$\begin{aligned} |\Gamma_G(S' \cup S)| &= |\Gamma_G(S') \cup \Gamma_G(S)| = |\Gamma_{G''}(S') \cup \Gamma_G(S)| \\ &\leq |\Gamma_{G''}(S')| + |\Gamma_G(S)| \\ &< |S'| + |S| = |S' \cup S| \ , \end{aligned}$$

im Widerspruch dazu, daß G die Bedingung $(*)$ erfüllt. Nach Induktionsannahme hat also auch G'' ein perfektes Matching M'', so daß $M' \cup M''$ ein perfektes Matching von G ist. □

Mit Hilfe des Heiratssatzes können wir den folgenden Satz beweisen.

2.13 Satz: (Färbungssatz)
Jeder bipartite n-reguläre Graph kann in n disjunkte perfekte Matchings zerlegt werden, d. h. er ist n-kantenfärbbar.

Beweis:
Diesen Beweis führen wir durch Induktion nach n.
Für $n = 1$ haben wir Graphen, in denen jeder Knoten an genau einer Kante beteiligt ist. Ein solcher Graph ist ein Matching, das offensichtlich 1-kantenfärbbar ist.
Sei nun $n > 1$. Wir zeigen, daß für alle $S \subseteq V_1$ gilt: $|\Gamma(S)| \geq |S|$. Aus S gehen $|S| \cdot n$ Kanten heraus. Jeder Knoten aus $\Gamma(S)$ kann höchstens Endpunkt von n dieser Kanten sein, also gilt: $|\Gamma(S)| \geq |S|$
Nach dem Heiratssatz enthält G darum ein perfektes Matching M, dessen Kanten wir die Farbe $n-1$ geben. Da $G \setminus M$ ein $(n-1)$-regulärer Graph ist, können wir gemäß der Induktionsannahme $G \setminus M$ mit $n - 1$ Farben einfärben. □

Die Aussage des Heiratssatzes läßt sich auch einfach auf nicht-reguläre Graphen verallgemeinern:

2.14 Korollar:
Jeder bipartite Graph mit maximalem Grad c ist c-kantenfärbbar, d. h. er kann in c disjunkte Matchings zerlegt werden.

Nach diesem Ausflug in die Graphentheorie können wir uns der Konstruktion eines speziellen Permutationsnetzwerks zuwenden.

2.2.2 Routing auf dem Permutationsnetzwerk

2.15 Definition:
Seien $n, d \in \mathbb{N}^+$. Das (n,d)-**Permutationsnetzwerk** (n,d)-PN ist das Netzwerk mit Prozessormenge $\mathcal{P} = \{(\ell, \bar{a}) \mid \ell \in \{-d, \ldots, -1, 1, \ldots, d\}, \bar{a} \in [n]^d\}$ und dem Kommunikationsgraphen $(\mathcal{P}, E)$,

$$\begin{aligned} E &= \Big\{\{(\ell,\bar{a}),(\ell+1,\bar{a}')\} \mid \ell < -1,\ a_i = a'_i \text{ für alle } i \neq |\ell| - 1\Big\} \\ &\cup \Big\{\{(\ell,\bar{a}),(\ell-1,\bar{a}')\} \mid \ell > 1,\ a_i = a'_i \text{ für alle } i \neq \ell - 1\Big\} \\ &\cup \Big\{\{(-1,\bar{a}),(1,\bar{a}')\} \mid a_i = a'_i \text{ für alle } i \neq 0\Big\} \end{aligned}$$

Die **Quellen** sind die Prozessoren $\{Q_{\bar{a}} \mid Q_{\bar{a}} = (-d, \bar{a}), \bar{a} \in [n]^d\}$ und die **Senken** sind die Prozessoren $\{S_{\bar{a}} \mid S_{\bar{a}} = (d, \bar{a}), \bar{a} \in [n]^d\}$. Wir nennen $Q_{\bar{a}}$ dann auch die $(\bar{a})_n$-te Quelle (bei den Senken analog).
Für $\ell \in \{-d, \ldots, -1, 1, \ldots, d\}$ heißen die Prozessoren $\{(\ell, \bar{a}) \mid \bar{a} \in [n]^d\}$ die Prozessoren auf **Level** ℓ.
Ein sehr bekanntes und häufig benutztes Netzwerk ist das ***d*-dimensionale Butterfly-Netzwerk** BF(d). Es entsteht aus $(2,d)$-PN durch Einschränkung der Knotenmenge auf $\{(\ell, \bar{a}) \mid \ell \in \{-1, 1, \ldots, d\}, \bar{a} \in [n]^d\}$. Dabei wird dann auch die -1 in den Prozessornamen durch 0 ersetzt. Werden zusätzlich die Knoten der Levels 0 und d miteinander identifiziert, so spricht man von einem Butterfly-Netzwerk mit **wrap-around**-Kanten.

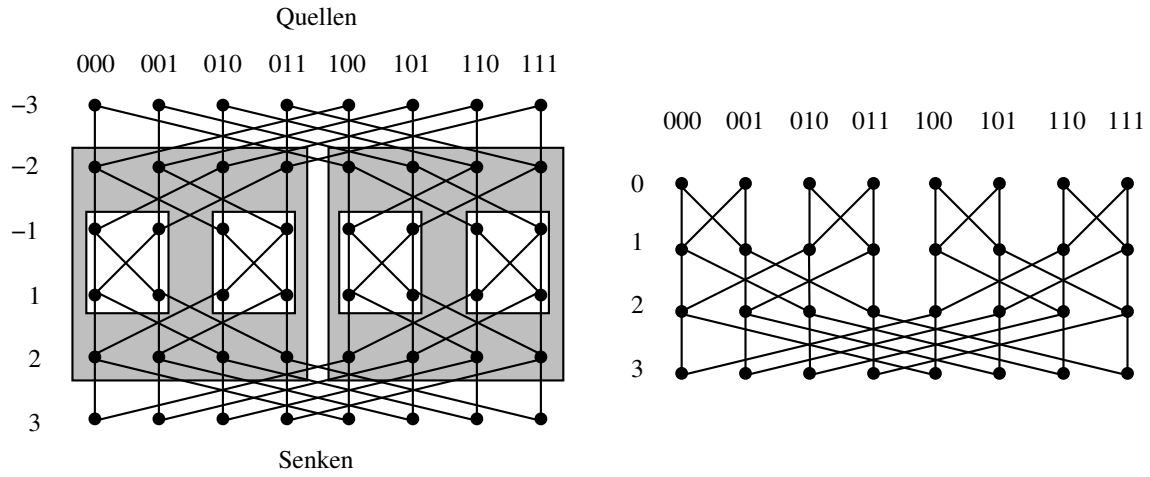
2.16 Beispiel:
Abbildung 2.2(a) zeigt das Netzwerk $(2,3)$-PN. Die darin enthaltenen Exemplare von (2,2)-PN und (2,1)-PN sind gekennzeichnet. Abbildung 2.2(b) zeigt das 3-dimensionale Butterfly-Netzwerk BF(3).

2.17 Bemerkung: (Eigenschaften von (n,d)-PN und BF(d))

a) *(n,d)-PN hat $2d \cdot n^d$ viele Prozessoren.*

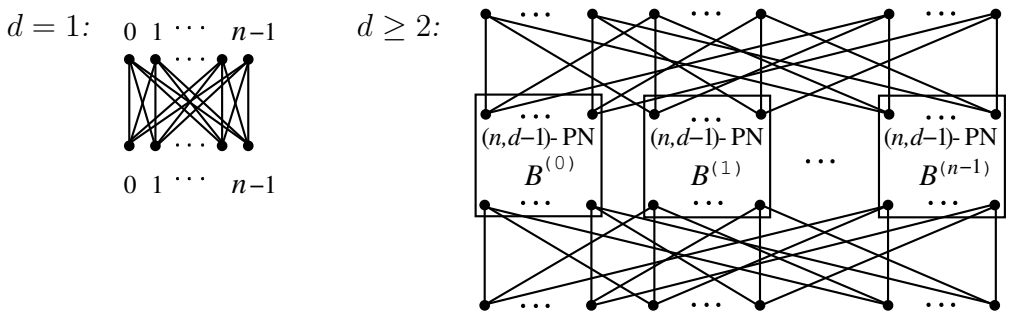
b) *(n,d)-PN hat Tiefe $2d-1$.*

c) *Sei $t < d$, $\bar{b} \in [n]^{d-t}$. Sei $\mathcal{P}_{\bar{b}} = \{(\ell, \bar{a}) \mid \ell \in \{-t, \ldots, -1, 1, \ldots, t\}$, $\bar{a} = \bar{b}\bar{c}$ für $\bar{c} \in [n]^t\}$. Dann ist (n,d)-PN$\Big|_{\mathcal{P}_{\bar{b}}} \cong (n,t)$-PN. Dies führt uns dann zu folgender rekursiver Zerlegung von (n,d)-PN. Für $i \in [n]$ und $t = d-1$ bezeichnen wir mit $B^{(i)}$ das Subnetzwerk $(n,d-1)$-PN$\Big|_{\mathcal{P}_i}$:*



(a) $(2,3)$-PN (b) BF(3)

Abbildung 2.2: Beispiele für das Permutations- und das Butterfly-Netzwerk.



d) BF(d) hat $(d+1)\cdot 2^d$ viele Prozessoren.

e) BF(d) hat Tiefe d.

f) Für je einen Prozessor $(0,\bar{a})$ und $(d,\bar{b})$ gibt es genau einen kürzesten Weg von $(0,\bar{a})$ nach $(d,\bar{b})$.

Nun wollen wir untersuchen, wie man in (n,d)-PN Permutationsrouting durchführen kann. Dabei gehen wir davon aus, daß Botschaften von den Quellen zu den Senken transportiert werden.

Wir zeigen jetzt erst einmal, daß wir für jede beliebige Permutation eine Menge von Wegen finden können, die von den Quellen zu den Senken verlaufen und die sich nie schneiden, d. h. die knotendisjunkt sind. In dem Beweis wenden wir die Dinge, die wir im Exkurs kennengelernt haben, an.

2.18 Lemma:
Für jede Permutation π auf $[n]^d$ gibt es in (n,d)-PN ein System $\mathcal{W}$ aus n^d knotendisjunkten Wegen $W_{\bar{a}}$ der Länge $2d-1$, so daß $W_{\bar{a}}$ bei $Q_{\bar{a}}$ beginnt und bei $S_{\pi(\bar{a})}$ endet.

Beweis:
Sei π eine beliebige Permutation auf $[n]^d$. Die Beweisidee sieht so aus, daß wir das Wegesystem $\mathcal{W}$ rekursiv entlang der gerade beschriebenen Zerlegung von (n,d)-PN definieren.

Für $d = 1$ ist $(n, 1)$-PN leicht zu initialisieren, da $(n, 1)$-PN der vollständige bipartite Graph mit jeweils n Knoten ist. In diesem Fall enthält $\mathcal{W}$ genau die Kanten $\{Q_{\bar{a}}, S_{\pi(\bar{a})}\}$. Diese Wege haben Länge $1 = 2 \cdot 1 - 1$.
Sei nun $d > 1$: Wir legen jetzt fest, welcher Weg $W_{\bar{a}}$ über welchen Teilpermutierer $B^{(i)}$ läuft. An die Festlegung stellen wir die beiden folgenden Forderungen:

(1) Über jede Quelle eines $B^{(i)}$ läuft genau ein Weg.

(2) Über jede Senke eines $B^{(i)}$ läuft genau ein Weg.

Wenn wir (1) und (2) erreichen können, müssen in den $B^{(i)}$ nur noch Permutationen auf $[n]^{d-1}$ realisiert werden, was nach Induktionsannahme möglich ist.
Wofür wir also sorgen müssen, ist, daß sich die Wege an den Ein- und Ausgängen der $B^{(i)}$ nicht treffen.
Betrachte darum den bipartiten Graphen G_π mit der Knotenmenge $\{q_{\bar{a}} \mid \bar{a} \in [n]^{d-1}\} \cup \{s_{\bar{a}} \mid \bar{a} \in [n]^{d-1}\}$ und den folgenden Kanten (auch Mehrfachkanten sind erlaubt): Für jedes Paar $(i\bar{x}, j\bar{y})$ mit $i, j \in [n]$, $\bar{x}, \bar{y} \in [n]^{d-1}$ und $\pi(i\bar{x}) = j\bar{y}$ ist $e_{i\bar{x}} := \{q_{\bar{x}}, s_{\bar{y}}\}$ eine Kante in G_π.
Da G_π bipartit und n-regulär ist, folgt aus dem Färbungssatz (Satz 2.13), daß G_π ein n-kantenfärbbarer Graph ist.
Sei φ eine solche n-Färbung. Führe den Weg $W_{\bar{a}}$ über das Teil-Permutationsnetzwerk $B^{(\varphi(e_{\bar{a}}))}$. Die Färbung garantiert uns, daß an den Quellen und den Senken von $B^{(\varphi(e_{\bar{a}}))}$ keine Kollisionen auftreten und von $B^{(\varphi(e_{\bar{a}}))}$ nur eine Permutation auf $[n]^{d-1}$ realisiert werden muß, was ja nach Induktionsannahme gemacht werden kann.
Die Weglänge ist $1 + 2(d-1) - 1 + 1 = 2d - 1$. □

Man beachte, daß dieser Beweis das Gerüst für einen Algorithmus bildet. Das einzige, was dazu noch fehlt, ist ein Färbungsalgorithmus für G_π. Solche Algorithmen sind aufwendig, allerdings nur, wenn n keine Zweierpotenz ist (vgl. auch Bemerkung 2.23).

2.19 Beispiel:
In Abbildung 2.3 ist für $\pi = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 4 & 1 & 0 & 3 & 2 & 6 & 5 & 7 \end{pmatrix}$ in (2,3)-PN ein mögliches Wegesystem eingezeichnet, das wir mit dem in dem Beweis angedeuteten Algorithmus erhalten. Dabei sieht man auch, daß es in G_π Mehrfachkanten geben kann.

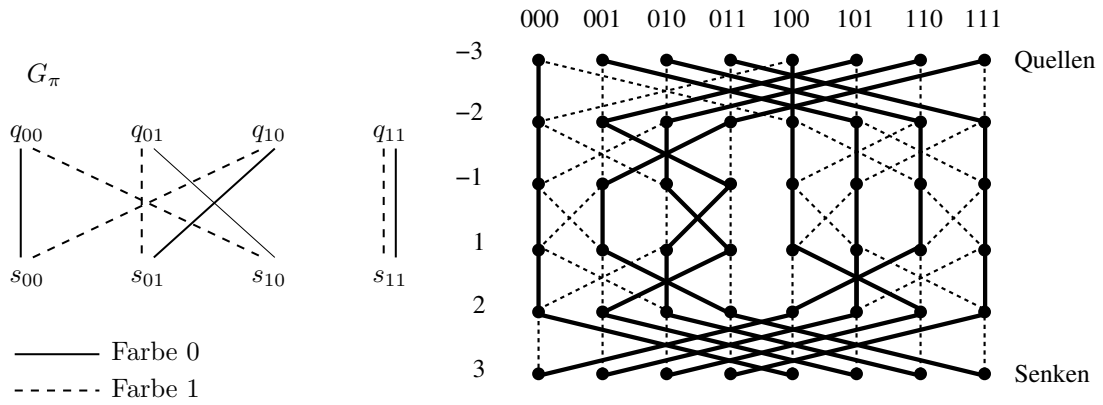


Abbildung 2.3: Konstruktion eines mögliches Wegesystems für π in (2,3)-PN.

Damit haben wir alle benötigten Aussagen zusammen, um den folgenden Satz über das Routing auf (n,d)-PN zu beweisen.

2.20 Satz:
(n,d)-PN kann bezüglich beliebiger Permutationen π auf $[n]^d$ mit Preprocessing routen, d. h. Botschaften von $Q_{\bar{a}}$ nach $S_{\pi(\bar{a})}$ senden (für alle $\bar{a} \in [n]^d$). Die Routingzeit ist $(2d-1)+(s-1)$, falls jede Botschaft aus s Einzelbotschaften besteht.

Beweis:
Konstruiere das Wegesystem $\mathcal{W}$ gemäß dem Algorithmus aus dem Beweis von Lemma 2.18. Die s Einzelbotschaften werden von $Q_{\bar{a}}$ nach $S_{\pi(\bar{a})}$ direkt aufeinanderfolgend entlang des Weges $W_{\bar{a}}$ durch (n,d)-PN geschickt. Da die Wege knotendisjunkt sind, entstehen unterwegs keine Kollisionen, d. h. kein Paket wird verzögert. Dadurch, daß die Pakete hintereinander losgeschickt werden, ist das letzte Paket gerade $2d+s-2$ Schritte unterwegs. Dieses Vorgehen wird **Pipelining** genannt. □

2.3 Simulation der Permutationsnetzwerke durch Gitter

In diesem Abschnitt schließen wir den Entwurf eines Permutationsalgorithmus für das Gitter $M(n,d)$ ab, indem wir das Permutationsnetzwerk (n,d)-PN auf $M(n,d)$ simulieren.
Wir führen dazu zur Vereinfachung der Beschreibung noch einige Bezeichnungen ein. Zu $M(n,d)$ bezeichnen wir für $\ell \in [n]$ mit M_ℓ das zu $M(n,d-1)$ isomorphe (vgl. Bemerkung 2.5) Gitter $M(n,d)\Big|_{\{\bar{a}|a_{d-1}=\ell\}}$, und für $\bar{b} \in [n]^{d-1}$ mit $A_{\bar{b}}$ das lineare Array $M(n,d)\Big|_{\{\bar{a}|\bar{a}=i\bar{b},i\in[n]\}}$. Es gilt:

- Die Gitter $M_0,\ldots,M_{n-1}$ zerlegen $M(n,d)$ disjunkt in n Exemplare von $M(n,d-1)$.
- Durch die $A_{\bar{b}}$ wird $M(n,d)$ in n^{d-1} viele Exemplare von $M(n,1)$ zerlegt.
- $A_{\bar{b}}$ enthält nacheinander je einen Prozessor aus $M_0, M_1, \ldots, M_{n-1}$, wie Abbildung 2.4 zeigt.

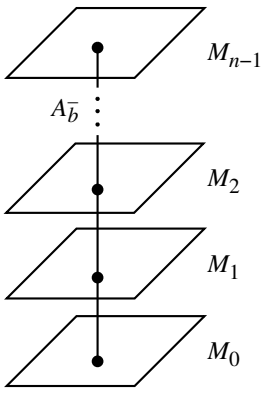


Abbildung 2.4: Zerlegung von $A_{\bar{b}}$.

2.21 Satz:
Auf $M(n,d)$ kann in Zeit $(4d-2)(n-1)$ eine beliebige Permutation mit Preprocessing mit Puffergröße 3 geroutet werden.

Beweis:
Zu einer beliebigen Permutation bestimme mit Hilfe des Graphen G_π wie im vorhergehenden Abschnitt eine Färbung $\varphi : [n]^d \to [n]$ für alle Knoten von $M(n,d)$.
Der folgende rekursive Algorithmus führt das Routing der Permutation durch. Dabei übernehmen die Gitter M_i die Aufgabe der $B^{(i)}$ der Permutationsnetzwerke, und die $A_{\bar{b}}$ übernehmen die Aufgabe der Kanten, die die Quellen mit den $B^{(i)}$ bzw. die $B^{(i)}$ mit den Senken verbinden.

1. Für alle $\bar{b} \in [n]^{d-1}$ permutiere die Pakete $(i\bar{b}, \pi(i\bar{b}), x_{i\bar{b}})$ auf dem linearen Array $A_{\bar{b}}$ so, daß $(i\bar{b}, \pi(i\bar{b}), x_{i\bar{b}})$ in Prozessor $(\varphi(i\bar{b})\bar{b})$ in $M_{\varphi(i\bar{b})}$ landet. Beachte dabei, daß $\varphi\Big|_{\{i\bar{b}|i\in[n]\}}$ bijektiv ist, da φ eine Färbung von G_π ist.

2. Permutiere rekursiv auf jedem M_j so, daß jedes $(i\bar{b}, \pi(i\bar{b}), x_{i\bar{b}})$ in demjenigen $A_{\bar{b}'}$ landet, in dem sein Zielprozessor liegt. Dies ist möglich, da jeder Zielprozessor nur ein Paket erhält (sonst wäre φ keine Färbung).

3. Permutiere auf jedem $A_{\bar{b}}$ die Pakete zu ihrem endgültigen Ziel.

Sei $T(n,d)$ die Laufzeit des Algorithmus auf $M(n,d)$. Dann gilt:

$$\begin{aligned} d=1: \qquad T(n,1) &= 2(n-1) \\ d>1: \qquad T(n,d) &= \underbrace{T(n,1)}_{1.} + \underbrace{T(n,d-1)}_{2.} + \underbrace{T(n,1)}_{3.} \end{aligned}$$

Aus Satz 2.7 wissen wir, daß $T(n,1) = 2(n-1)$ ist.
Die Lösung der Rekursion ergibt dann $T(n,d) = (4d-2)(n-1)$, was leicht ausgerechnet werden kann. □

Bei diesem Algorithmus ist nicht wie im Permutationsnetzwerk Pipelining möglich, da alle Prozessoren Quellen und Senken gleichzeitig sind. Im Permutationsnetzwerk konnten die inneren Prozessoren als Speicher genutzt werden.

2.22 Bemerkung:

a) *Das obige Ergebnis ist bis auf den Faktor ≈ 4 optimal, da der Durchmesser von $M(n,d)$ nach Bemerkung 2.5 genau $(n-1)d$ ist.*

b) *Betrachten wir den obigen Algorithmus für den Hypercube $M(2,d)$, so geht der Algorithmus so vor, daß zuerst entlang der höchsten Dimension kommuniziert wird, dann entlang der zweithöchsten usw., bis schließlich die Folge der Dimensionen noch einmal steigend durchlaufen wird. Dieses Phänomen werden wir in Kapitel 4 genauer unter die Lupe nehmen.*

c) *Eine Verallgemeinerung dieses Konzeptes wird in[BA91] vorgestellt. Dort wird gezeigt, wie man aus bekannten Routing-Algorithmen für Netzwerke Routing-Algorithmen für das Kreuzprodukt von diesen Netzwerken erhält.*

2.4 Abschließende Bemerkungen

Ursprünge. Die Fähigkeit des Routing-Netzwerks, für jede Permutation eingestellt werden zu können, diese auf knotendisjunkten Wegen zu routen, ist bereits seit den sechziger Jahren bekannt[Ben64, Ben65, Wak68].

Effizientes Preprocessing. In der folgenden Bemerkung fassen wir zusammen, was über das Preprocessing für die Routing-Algorithmen bekannt ist, die wir in diesem Kapitel vorgestellt haben. Wir gehen dabei davon aus, daß das Preprocessing als paralleler Algorithmus auf (n,d)-PN bzw. $M(n,d)$ implementiert wird.

2.23 Bemerkung:
Nassimi/Sahni zeigen in[NS82] für $n = 2$ die Preprocessingzeit $O(d^4)$ auf (n,d)-PN. Für $n = 2^k$ wird in[LPV81] und[Fig89] eine Preprocessingzeit von $O\Big((d\cdot k)^2 \cdot \mathrm{sort}((n,d)\text{-PN})\Big)$ bewiesen, und in[Fig89] eine Preprocessingzeit von $O\Big((d\cdot k)^2 \cdot \mathrm{sort}(M(n,d))\Big)$, wobei sort(.) *die parallele Zeit ist, um auf dem entsprechenden Netzwerk n^d Zahlen zu sortieren.*
Für sonstige n sind keine effizienten parallelen Algorithmen bekannt. Das Preprocessing kann für beliebige n sequentiell in Zeit $O(d\cdot n\cdot (n^d)^2)$ durchgeführt werden.

2.24 Bemerkung:
Sind die Permutationen partiell, so können diese natürlich in der Preprocessing-Phase zu totalen Funktionen „umgebaut“ werden. Damit gelten alle in diesem Kapitel gezeigten Aussagen auch für partielle Permutationen.

In Abschnitt 7 werden wir ein Netzwerk untersuchen, in dem sogar kein Preprocessing nötig ist, um jede Permutation in logarithmischer Zeit zu routen.

Namensgebung. Für das Permutationsnetzwerk $(2,d)$-PN sind auch die Namen **Beneš-Netzwerk** und **Waksman-Netzwerk** in Gebrauch.
Eine ähnliche Begriffsvielfalt gibt es für das Butterfly-Netzwerk. Für BF(d) findet man auch manchmal die Bezeichnung **FFT-Netzwerk**, da die Struktur des Netzwerkes genau der Struktur des Algorithmus zur schnellen Fourier-Transformation entspricht. In Arbeiten, wo die Bezeichnung FFT-Netzwerk benutzt wird, versteht man dann unter dem Butterfly-Netzwerk das Netzwerk, das wir als Butterfly-Netzwerk mit wrap-around-Kanten vorgestellt hatten.

Offene Probleme. Mit Satz 2.20 haben wir gezeigt, daß $(2,d)$-PN für alle Permutationen $\pi : [2^d] \to [2^d]$ ein knotendisjunktes Wegesystem enthält. Dieses Netzwerk besteht aus zwei Kopien des Butterfly-Netzwerks BF(d), die „umgekehrt“ aneinandergesetzt worden sind. Hier stellt sich die Frage, ob das Vorgehen so gewählt werden muß, d. h. ob es notwendig ist, die Netzwerke derart aneinander zu kleben. Mit Satz 2.20 und etwas „Bastelarbeit“ können wir folgern, daß man mit vier hintereinander geschalteten Butterfly-Netzwerken jede Permutation über knotendisjunkte Wege führen kann. Parker[Par80] zeigt, daß drei Kopien ausreichen. Daß eine Kopie von BF(d) nicht ausreicht, ist leicht einzusehen (vgl. S. 63). Ob bereits zwei Kopien von BF(d) hinreichend sind, ist nicht bekannt.

Weitere Kommunikationsprobleme. Es gibt noch einige weitere sehr grundlegende Probleme der Kommunikation in Netzwerken. Dazu gehören:

- Das **Distributionsrouting**, bei dem die Eingabepakete an mehrere Zielprozessoren geschickt werden können. Das Distributieren werden wir in Abschnitt 5.2.1 näher betrachten.Das **Relationen-Routing**, das das Funktionen- und das Distributionsrouting zusammenfaßt.
- Das **Broadcasting**, bei dem ein Prozessor eine Botschaft besitzt, die an alle anderen Prozessoren geschickt werden soll, und das **Gossipping**, das diese Operation dahingehend verallgemeinert, daß jeder Prozessor eine Botschaft hat, die an alle anderen Prozessoren geschickt werden soll („alle bekommen alles“). Ein Übersichtsartikel dazu ist[HHL88].
- Das **Token Distribution** (auch: **Lastausgleich** oder **Load balancing**), bei dem die Pakete (**token** genannt), keine Zieladressen haben, sondern gleichmäßig über das gesamte Netzwerk verteilt werden müssen.

Die Wichtigkeit des Permutationsroutings besteht darin, daß sich viele der oben aufgeführten Probleme auf das Permutationsrouting reduzieren lassen, bzw. das Permutationsrouting eine grundlegende Hilfsfunktion zur Lösung der Probleme darstellt.

2.5 Literatur zu Kapitel 2

[BA91] M. Baumslag and F. Annexstein. A unified framework for off-line permutation routing in parallel networks. *Mathematical Systems Theory*, 24:233–251, 1991.

[Ben64] V. Beneš. Permutation groups, complexes, and rearrangeable multistage connecting networks. *Bell System Technical Journal*, 43:1619–1640, 1964.

[Ben65] V. Beneš. *Mathematical Theory of Connecting Networks and Telephone Traffic*. Academic Press, New York, NY, 1965.

[Bol98] B. Bollobás. *Modern Graph Theory*. Springer-Verlag, Heidelberg, 1998.

[Fig89] M. Figge. *Permutationsrouting auf hochdimensionalen Gittern*. Diplomarbeit, Universität Dortmund, 1989.

[HHL88] S. M. Hedetniemi, S. T. Hedetniemi, and A. L. Liestman. A survey of gossiping and broadcasting in communication networks. *Networks*, 18:319–349, 1988.

[LPV81] G. F. Lev, N. Pippenger, and L. G. Valiant. A fast parallel algorithm for routing in permutation networks. *IEEE Transactions on Computers*, 30:93–100, 1981.

[NS82] D. Nassimi and S. Sahni. Parallel algorithms to set up the Benes permutation network. *IEEE Transactions on Computers*, 31:148–154, 1982.

[Par80] D. Parker. Notes on shuffle/exchange-type switching networks. *IEEE Transactions on Computers*, C-29:213–222, 1980.

[Wak68] A. Waksman. A permuting network. *Journal of the ACM*, 15:159–163, 1968.

Kapitel 3

Sortiernetzwerke

In diesem Kapitel der Vorlesung untersuchen wir ein Problem, das uns schon aus dem Grundstudium bekannt ist, und das allgemein von großer Wichtigkeit ist, nämlich das *Sortierproblem*. Wir werden hier wieder zuerst ein spezielles Netzwerk kennenlernen, das extra für das Sortieren entworfen worden ist, und dann sehen, daß dieses Netzwerk auf dem Hypercube $M(2,d)$ effizient simuliert werden kann.

3.1 Das 0-1-Prinzip

3.1 Definition: (Sortiernetzwerk)
Sei $n \in \mathbb{N}$. Sei $\mathcal{A}(n) = (M_1, \ldots, M_{t(n)})$ eine Folge von Mengen M_k, $1 \leq k \leq t(n)$, wobei M_k aus Paaren (i,j) besteht, so daß $i,j \in \{1,\ldots,n\}$, $i \neq j$ gilt, und weder i noch j an weiteren Paaren in M_k beteiligt sind. Sei $\bar{x} = (x_1, \ldots, x_n)$ eine Folge natürlicher Zahlen. Dann verstehen wir unter der Anwendung des Paares (i,j) auf $\bar{x}$ die Ausführung der Anweisung „`if` $x_i > x_j$ `then` vertausche (x_i, x_j)“. Ein solches Paar (i,j) nennen wir darum auch **Komparator** oder **Vergleichselement**.
Einen Komparator (i,j) können wir folgendermaßen darstellen (siehe Abbildung 3.1): Wir haben zwei Leitungen mit den Namen i und j. Zwischen diesen Leitungen haben wir eine Kante, wobei ein Pfeil auf die Leitung j zeigt. Da i und j ja an keinem weiteren Komparator

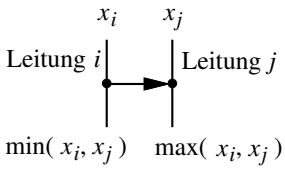


Abbildung 3.1: Ein Komparator.

in M_k beteiligt sind, können alle Komparatoren aus M_k gleichzeitig arbeiten. Darum nennen wir M_k einen **Vergleichsschritt**.
Die Darstellung aller Komparatoren von $\mathcal{A}(n)$ nennen wir **Komparatornetzwerk** der Tiefe $t(n)$. Wir identifizieren das Netzwerk mit $\mathcal{A}(n)$.
Die Ausgabe von $\mathcal{A}(n)$ bei Eingabe $\bar{x} = (x_1, \ldots, x_n)$ ist die Folge $(x_{\pi(1)}, \ldots, x_{\pi(n)})$, wobei $x_{\pi(i)}$ auf der Leitung i liegt. (Beachte: π ist eine Permutation, da ein Komparator weder etwas

löscht, noch etwas neu erzeugt oder überschreibt.)
Gilt für die Ausgabe $(x_{\pi(1)}, \ldots, x_{\pi(n)})$, daß $x_{\pi(1)} \leq \ldots \leq x_{\pi(n)}$, so hat $\mathcal{A}(n)$ die Eingabe $\bar{x}$ sortiert. Wird jede Eingabe sortiert, so nennen wir das zugehörige Netzwerk ein **Sortiernetzwerk** der Tiefe $t(n)$.

Die englisch-sprachigen Fachausdrücke sind **sorting network** oder auch, zur besseren Unterscheidung von den Netzwerken wie den Gittern oder Permutierern, **sorting circuit**.

3.2 Beispiel:
Abbildung 3.2 zeigt zwei Komparatornetzwerke der Tiefen 4 und 3, die Sortiernetzwerke sind.

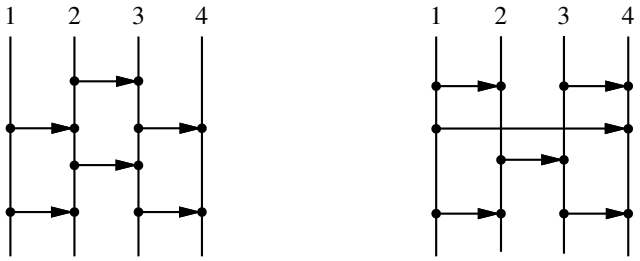


Abbildung 3.2: Zwei Sortiernetzwerke.

Um von einem Netzwerk $\mathcal{A}(n)$ nachzuweisen, daß es wirklich ein Sortiernetzwerk ist, müßte gezeigt werden, daß $\mathcal{A}(n)$ alle möglichen Eingabefolgen der Länge n sortiert. Oft ist es aber viel einfacher, über 0-1-Folgen zu argumentieren. Tatsächlich reicht dies auch aus, wie das nachstehende Lemma zeigt.

3.3 Lemma: (0-1-Prinzip)
Sei $\mathcal{A}(n) = (M_1, \ldots, M_{t(n)})$ ein Komparatornetzwerk. $\mathcal{A}(n)$ sortiert genau dann jede Folge ganzer Zahlen der Länge n, wenn $\mathcal{A}(n)$ jede Folge aus $\{0,1\}^n$ sortiert.

Beweis:
$\Rightarrow$: Diese Aussage ist offensichtlich.
$\Leftarrow$: Diese Richtung zeigen wir indirekt.
Annahme: Es gibt ein Komparatornetzwerk $\mathcal{A}(n)$, das zwar jede 0-1-Folge der Länge n sortiert, aber die Zahlenfolge $\bar{x} = (x_1, \ldots, x_n) \in \mathbb{N}^n$ nicht.
Wenden wir $\mathcal{A}(n)$ auf $\bar{x}$ an, so führt $\mathcal{A}(n)$ eine Permutation π auf $\bar{x}$ aus. Da die Eingabefolge ja nicht sortiert worden ist, gibt es einen kleinsten Index i_0, $1 \leq i_0 \leq n$, so daß gilt: $x_{\pi(i_0-1)} > x_{\pi(i_0)}$. Wir definieren uns nun die 0-1-Folge $\bar{y}$ wie folgt:

$$y_j := \begin{cases} 0 & \text{falls } x_j \leq x_{\pi(i_0)} \\ 1 & \text{falls } x_j > x_{\pi(i_0)} \end{cases}$$

Sortiert $\mathcal{A}(n)$ diese Folge? Nein, denn es gilt für alle $k \in \{1, \ldots, t(n)\}$: Treffen an einem Komparator $(\ell, m) \in M_k$ die Zahlen x_i und x_j (der ursprünglichen Eingabe) auf den Leitungen ℓ und m aufeinander, so treffen am gleichen Komparator auf den gleichen Leitungen auch y_i und y_j aufeinander, d. h. die gleiche Permutation wird auch auf $\bar{y}$ ausgeführt. Diese Aussage kann durch Induktion nach k einfach gezeigt werden:

Für $k = 1$ gilt die Aussage offensichtlich.
Nun gelte die Aussage für k. Wir zeigen, daß sie auch für $k + 1$ gilt. Da die Aussage für k gilt, treffen an einem Komparator $(\ell, m) \in M_k$ sowohl x_i und x_j als auch y_i und y_j auf den entsprechenden Leitungen aufeinander. Bei dem, was an diesem Komparator passiert, unterscheiden wir zwei Fälle:

(i) $x_i \leq x_j$ und darum auch $y_i \leq y_j$ (überlegen!):
Der Komparator sorgt dafür, daß x_i auf der Leitung mit der Nummer ℓ und x_j auf der Leitung mit der Nummer m bleibt. Gleiches gilt natürlich auch für y_i und y_j.

(ii) $x_i > x_j$ und darum auch $y_i \geq y_j$:
Hier sorgt der Komparator dafür, daß x_i auf die Leitung mit der Nummer m und x_j auf die Leitung mit der Nummer ℓ gelegt wird. Wieder passiert das gleiche mit y_i und y_j.

Man beachte, daß wir bei Gleichheit von y_i und y_j annehmen können, daß beide vertauscht werden oder nicht, je nach dem, ob x_i und x_j vertauscht werden oder nicht. Also gilt, daß x_i und y_i wie auch x_j und y_j auch nach Ausführung des Komparationsschrittes auf denselben Leitungen liegen.
Als Ergebnis können wir festhalten, daß auf der Folge $\bar{y}$ dieselbe Permutation ausgeführt wird wie auf $\bar{x}$. Was liegt nun am Ende auf den Leitungen $i_0 - 1$ und i_0 bei Eingabe $\bar{y}$? $y_{\pi(i_0-1)} = 1 > 0 = y_{\pi(i_0)}$, also wurde $\bar{y}$ von unserem Komparatornetzwerk ebenfalls nicht sortiert, im Widerspruch zur Annahme. □

Man beachte, daß das 0-1-Prinzip nur für Komparatornetzwerke gilt. Es reicht also nicht, einen Sortieralgorithmus zu schreiben, der 0-1-Folgen sortiert, indem die Anzahl der Einsen ausgerechnet und dann dementsprechend die Ausgabe mit Nullen und Einsen belegt wird.
Wir werden nun ein Sortiernetzwerk explizit konstruieren und mit Hilfe des 0-1-Prinzips dessen Korrektheit beweisen.

3.2 Konstruktion eines Sortiernetzwerks: Der Bitone Sortierer

Sortierverfahren wie Quicksort und Heapsort lassen sich nicht in ein Sortiernetzwerk umformen, da im Verlauf der Rechnung die Paare, die verglichen werden, von dem bisherigen Verlauf der Rechnung und damit von der Eingabefolge abhängen, womit diese Sortierverfahren adaptiv (d. h. anpassend) sind.
Mehr Erfolg zeigt eine Parallelisierung von Mergesort, für den wir hier eine rekursive Beschreibung vorstellen. Sei nun hier und im folgenden immer $n = 2^d$:

$d = 1$: sortiere (x_1, x_2)
$d \geq 2$: sortiere $(x_1, \ldots, x_{n/2})$;
sortiere $(x_{n/2+1}, \ldots, x_n)$;
mische die Ausgabefolgen der beiden Sortierdurchgänge ;

Die sequentielle Laufzeit dieses Algorithmus ist $O(n \log n)$.

3.4 Definition:
Eine 0-1-Folge $\bar{x}$ der Länge n heißt **biton**, falls es einen Index i gibt, so daß $(x_1, \ldots, x_i)$

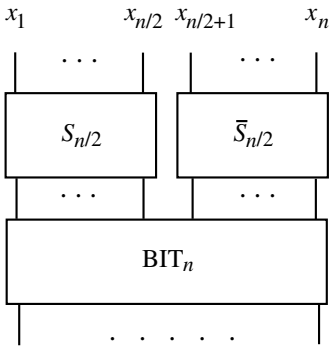
monoton steigend und $(x_{i+1}, \ldots, x_n)$ monoton fallend ist, oder genau umgekehrt.[1]
Ein Komparatornetzwerk, das alle bitonen 0-1-Folgen der Länge n sortiert, heißt **Bitoner n-Mischer**.

3.5 Beispiel:
Die Folgen $(1,1,0,0,0,0,1,1,1)$, $(0,0,0,0)$, $(0,0,0,1,1,0,0,0)$ und $(0,0,1,1)$ sind biton.

Wenn man einen Bitonen n-Mischer hat, dann kann man recht einfach ein Sortiernetzwerk konstruieren, wie die folgende Konstruktion zeigt.

3.6 Definition:
($n = 2^d$) Sei BIT_n ein Bitoner n-Mischer. S_n ist dann das folgende Netzwerk:

$d = 1$: 

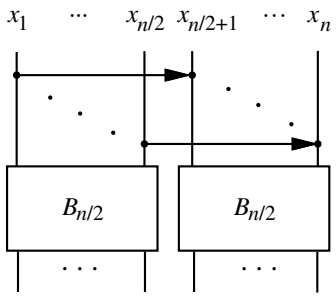
$d \geq 2$: 

Dabei bezeichnet $\bar{S}_{n/2}$ das Komparatornetzwerk, das wir erhalten, wenn wir $S_{n/2}$ an der Leitung mit der höchsten oder der niedrigsten Nummer spiegeln. Formal wird dies für den Komparator (i,j) durch die Abbildung $\sigma(i,j) = (n-i+1, n-j+1)$ beschrieben. Eine einfache Überlegung zeigt, daß $\bar{S}_{n/2}$ dann die Eingabe fallend sortiert.
Dieses Komparatornetzwerk heißt **Batchers Bitoner Sortierer** nach seinem „Erfinder".

Daß dieses Netzwerk ein Sortiernetzwerk ist, kann man sich mit Hilfe des 0-1-Prinzips nun einfach überlegen. Wir werden dies erst später formulieren, sobald wir einen Bitonen n-Mischer explizit konstruiert haben.

3.7 Definition:
Sei B_n das folgendermaßen rekursiv definierte Komparatornetzwerk:

$d = 1$: 

$d \geq 2$: 

Es heißt **Batchers Bitoner Mischer**.

3.8 Beispiel:
Abbildung 3.3 zeigt den Bitonen Mischer B_8.

[1] Beachte, daß diese Definition **ausschließlich** für 0-1-Folgen gilt. Eine Definition des Begriffs biton für beliebige Zahlenfolgen sieht folgendermaßen aus: Eine Zahlenfolge $\bar{x}$ der Länge n heißt **biton**, falls
(a) es einen Index i gibt, so daß $(x_1, \ldots, x_i)$ monoton steigend und $(x_{i+1}, \ldots, x_n)$ monoton fallend ist, oder
(b) $\bar{x}$ in zwei Folgen $\bar{a}$ und $\bar{b}$ mit $\bar{a}\bar{b} = \bar{x}$ so aufgetrennt werden kann, daß für $\bar{b}\bar{a}$ die Eigenschaft (a) gilt.

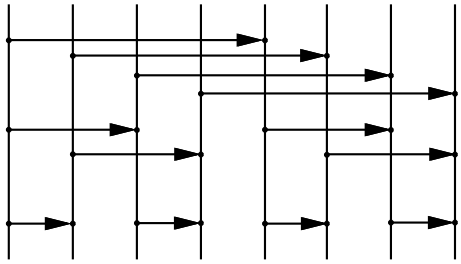


Abbildung 3.3: Der Bitone Mischer B_8.

3.9 Lemma:
B_n ($n = 2^d$) ist ein Bitoner Mischer der Tiefe $\log n$ und besteht aus $\frac{1}{2}n \log n$ Komparatoren. D. h. also, daß durch B_n jede bitone 0-1-Folge sortiert wird.

Beweis:
Sei $(x_1, \ldots, x_n)$ eine bitone 0-1-Folge und die Eingabe für B_n. Seien $(z_1, \ldots, z_n)$ die Zahlen, die auf den entsprechenden Leitungen liegen, nachdem die **ersten** Komparatoren durchlaufen worden sind (also die Eingaben für die beiden $B_{n/2}$), und sei $\bar{y} = (y_1, \ldots, y_n)$ die Ausgabe von B_n. Wir zeigen durch Induktion nach d, daß $\bar{y}$ sortiert ist, indem wir die folgenden Eigenschaften zeigen:

(*) $(z_1, \ldots, z_{n/2})$ und $(z_{n/2+1}, \ldots, z_n)$ sind biton.

(**) Es gilt: $(z_1, \ldots, z_{n/2}) = (0, \ldots, 0)$ oder $(z_{n/2+1}, \ldots, z_n) = (1, \ldots, 1)$

Da nach Induktionsannahme sowohl $(y_1, \ldots, y_{n/2})$ als auch $(y_{n/2+1}, \ldots, y_n)$ sortiert sind, und da wegen (**) gilt, daß $y_{n/2} \leq y_{n/2+1}$ ist, folgt unmittelbar, daß $\bar{y}$ sortiert ist.
Um noch die Eigenschaften (*) und (**) zu zeigen, unterscheiden wir 6 Fälle:

1. $\bar{x} = (\underbrace{0, \ldots, 0}_{m}, \underbrace{1, \ldots, 1}_{k} \,\big|\, \underbrace{1, \ldots, 1}_{j}, \underbrace{0, \ldots, 0}_{i})$ mit $m + k = j + i = \frac{1}{2}n$.

   (a) $k \leq i$:
   Dann sind: $(z_1, \ldots, z_{n/2}) = (0, \ldots, 0)$ (also (**)) und
   $(z_{n/2+1}, \ldots, z_n) = (\underbrace{1, \ldots, 1}_{j}, 0, \ldots, 0, \underbrace{1, \ldots, 1}_{k})$ (also zusammen (*) und (**)).

   (b) $k > i$:
   $(z_{n/2+1}, \ldots, z_n) = (1, \ldots, 1)$ (also (**)) und
   $(z_1, \ldots, z_{n/2}) = (\underbrace{0, \ldots, 0}_{m}, 1, \ldots, 1, \underbrace{0, \ldots, 0}_{i})$ (also wieder zusammen (*) und (**)).

2. $\bar{x} = (\underbrace{0, \ldots, 0}_{n/2} \,\big|\, \underbrace{0, \ldots, 0, 1, \ldots, 1, 0, \ldots, 0}_{n/2})$
   Nun ist $\bar{z} = \bar{x}$.

3. $\bar{x} = (\underbrace{0, \ldots, 0, 1, \ldots, 1, 0, \ldots, 0}_{n/2} \,\big|\, \underbrace{0, \ldots, 0}_{n/2})$
   Hier sind $(z_1, \ldots, z_{n/2}) = (x_{n/2+1}, \ldots, x_n)$ und $(z_{n/2+1}, \ldots, z_n) = (x_1, \ldots, x_{n/2})$.

Die Fälle 4. bis 6. sind die analogen Fälle, die wir bekommen, wenn wir oben Nullen und Einsen vertauschen. Sie werden genauso bewiesen.
Sei jetzt $t(n)$ die Tiefe von B_n. Dann gilt:

$$\left.\begin{array}{lrcl} n=2: & t(2) & = & 1 \\ n>2: & t(n) & = & t(n/2)+1 \end{array}\right\} \Rightarrow t(n) = \log n$$

Für die Zahl $s(n)$ der Komparatoren gilt: $s(n) = \frac{1}{2}n \cdot t(n)$ und somit $s(n) = \frac{1}{2}n \log n$. □

Damit können wir nun den folgenden Satz beweisen:

3.10 Satz:
S_n ist ein Sortiernetzwerk der Tiefe $\frac{1}{2}\log n(\log n + 1)$ und hat $\frac{1}{4}n \log n(\log n + 1)$ viele Komparatoren.

Beweis:
Wenn $S_{n/2}$ alle 0-1-Folgen der Länge $n/2$ aufsteigend sortiert und $\bar{S}_{n/2}$ alle 0-1-Folgen der Länge $n/2$ fallend sortiert, so ergibt die Konkatenation der Ausgabefolgen dieser beiden Netzwerke eine bitone 0-1-Folge, die dann wiederum von B_n gemäß Lemma 3.9 korrekt sortiert wird. D. h. also, daß durch S_n jede 0-1-Folge der Länge n sortiert wird und somit wegen des 0-1-Prinzips (Lemma 3.3) auch jede beliebige Folge natürlicher Zahlen.
Sei $t(n)$ die Tiefe von S_n. Dann gilt:

$$\left.\begin{array}{lrcl} n=2: & t(2) & = & 1 \\ n>2: & t(n) & = & t(n/2)+\log n \end{array}\right\} \Rightarrow t(n) = \tfrac{1}{2}\log n(\log n + 1)$$

Für die Zahl $s(n)$ der Komparatoren gilt wiederum: $s(n) = \frac{1}{2}n \cdot t(n)$ und somit $s(n) = \frac{1}{4}n \log n(\log n + 1)$ □

Man beachte bei der gesamten Beweisführung folgendes. Wir haben nur gezeigt, daß B_n bitone 0-1-Folgen sortiert, d. h. für B_n haben wir keinerlei Aussagen gemacht, was es mit beliebigen Zahlenfolgen macht.[2] Erst als wir gezeigt haben, daß S_n jede 0-1-Folge der Länge n sortiert, haben wir wegen des 0-1-Prinzips auch etwas über beliebige Zahlenfolgen bewiesen.

3.11 Beispiel:
Abbildung 3.4 zeigt das Netzwerk S_8. Man beachte, wie die rekursive Konstruktion ihren Niederschlag in der Richtung der Pfeile an den Komparatoren findet.

Eine genaue Betrachtung des obigen Beispiels zeigt, daß, wenn wir die Leitungen als jeweils einen Prozessor auffassen, und die Komparatoren jeweils als Verbindungen dieser Prozessoren, wir wiederum wie beim Permutationsnetzwerk das Gitter $M(2,3)$ und allgemein das Gitter $M(2,d)$, das wir ja auch Hypercube der Dimension d nennen, erhalten. Somit ergibt sich folgende Aussage:

3.12 Satz:
Auf dem Hypercube $M(2,d)$ können $n = 2^d$ Zahlen in Zeit $O(d^2)$ sortiert werden.

[2] Man kann aber zeigen, daß B_n die in der Fußnote auf S. 20 als biton bezeichneten Zahlenfolgen ebenfalls sortiert.

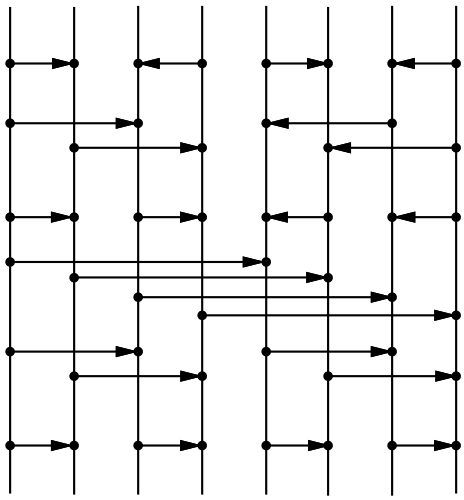


Abbildung 3.4: Der Bitone Sortierer S_8.

Beweis:
Die Aufgabe einer jeden Leitung i kann der Prozessor $\text{bin}_d(i-1)$ des Hypercubes direkt übernehmen. □

Eine weitere Analogie zum Verhältnis des Permutationsnetzwerkes zum Gitter besteht darin, daß während eines Schrittes immer nur entlang einer Dimension kommuniziert wird, und daß sogar die Reihenfolge, in der Dimensionen durchlaufen werden, sehr strukturiert ist.
Die Untersuchung dieses Phänomens, die zu den Ascend/Descend-Programmen führt, findet im nächsten Kapitel statt.

3.3 Zusammenfassung und Überblick

Parallele Sortieralgorithmen sind das klassische Gebiet der parallelen Datenverarbeitung. Schon in den 50er Jahren sind die ersten Untersuchungen auf diesem Gebiet durchgeführt worden.

Deterministische Sortiernetzwerke. Wir haben in diesem Kapitel Batchers Bitonen Sortieralgorithmus kennengelernt, der auf $O(n)$ Prozessoren des Hypercubes $M(2, \log n)$ in Zeit $O((\log n)^2)$ Zahlenfolgen der Länge n sortieren kann. Das Ergebnis stammt aus[Bat68]. Die erzielte Beschleunigung durch die $O(n)$ Prozessoren ist nicht optimal, da wir bei einer Sequentialisierung des Bitonen Sortierens einen Algorithmus der Laufzeit $O(n(\log n)^2)$ erhalten, was z. B. von Heapsort und Mergesort unterboten wird.
Weitere $O((\log n)^2)$-Verfahren sind

- Batchers *Odd-Even Merge Sorting Network*[Bat68],
- Pratts Parallelisierung von Shell-Sort[Pra71],
- das *Periodic Balanced Sorting Network* von Dowd et al.[DPSR89],
- das *Pairwise Sorting Network* von Parberry[Par92].

Viele fundamentale Aussagen für Sortiernetzwerke, so z. B. das 0-1-Prinzip, finden sich in Knuths monumentalem Werk[Knu98].

1983 ist es Ajtai, Komlós und Szeméredi[AKS83] erstmalig gelungen, ein Sortiernetzwerk der Tiefe $O(\log n)$ zu konstruieren. Der Autoren wegen ist es unter dem Namen **AKS-Netzwerk** bekannt. Eine verständliche Beschreibung des Verfahrens gibt Paterson[Pat90]. Das AKS-Netzwerk ist jedoch leider nur von theoretischem Interesse, da der in der O-Notation versteckte konstante Faktor enorm groß ist (Er liegt zur Zeit bei ca. 1830[Chv92]). Dieses Verfahren kann auch nicht auf dem Hypercube derart simuliert werden, daß die Größenordnung der Laufzeit erhalten bleibt. Für praktische Anwendungen ist das in diesem Kapitel vorgestellte Verfahren, das aus dem Jahr 1968 stammt, das bislang mit Abstand effizienteste bekannte Verfahren.

1992 hat Plaxton[Pla92], ausgehend von einer gemeinsamen Arbeit mit Leighton[LP90], ein Sortiernetzwerk der Tiefe $o((\log n)^{1+\varepsilon})$ für beliebig kleine $\varepsilon > 0$ konstruiert, das auf dem Hypercube direkt simuliert werden kann.

Adaptive Verfahren Hat man mehr Prozessoren als Zahlen, die sortiert werden sollen, so ist *sparse enumeration sort* von Nassimi/Sahni[NS82] ein schneller Sortieralgorithmus.

Cypher hat mit einem Ansatz, der viele der Techniken aus Kapitel 2 und diesem Kapitel benutzt, einen Sortieralgorithmus für den Hypercube entworfen[CP93], der in Zeit $O(d^{3/2})$ auf $M(2, d)$ zu sortieren vermag. Dieses Verfahren ist jedoch adaptiv in dem Sinne, daß die Folge der Schlüsselvergleiche auch von der Eingabe abhängt. Darum kann dieses Verfahren nicht durch ein Komparatornetzwerk dargestellt werden.

Darauf aufbauend ist es Cypher und Plaxton gelungen[CP93], auf $M(2, d)$ in Zeit $O(d(\log d)^2)$ zu sortieren. Leider ist einerseits die Konstante wiederum sehr groß, andererseits ist das angewandte Verfahren ebenfalls adaptiv und somit wiederum nicht durch ein Komparatornetzwerk zu beschreiben. Dieses Verfahren liegt jedoch dem oben erwähnten $o((\log n)^{1+\varepsilon}$-Verfahren zugrunde.

Probabilistische Sortierverfahren Von Leighton/Plaxton stammt ein Sortiernetzwerk (es basiert auf dem Butterfly-Netzwerk), das in $\approx 7 \log n + o(\log n)$ fast alle Zahlenfolgen der Länge n sortiert[LP90].

Ein probabilistischer Algorithmus für den Hypercube von Reif und Valiant[RV87] sortiert in erwarteter Zeit $O(\log n)$.

Andere Rechenmodelle Auf PRAMs mit n Prozessoren $O((\log n)^2)$-Verfahren zu entwickeln, ist relativ einfach. Ein Algorithmus der Laufzeit $O(\log n \operatorname{loglog} n)$ ist bei Borodin/Hopcroft[BH85] zu finden. Cole ist es 1986 gelungen (veröffentlicht in[Col88]), einen Sortieralgorithmus zu finden, der mit $O(n)$ Prozessoren in Zeit $O(\log n)$ sortiert, wobei die in der O-Notation verborgene Konstante nicht sehr groß ist. Dieses Verfahren kann jedoch nicht durch ein Komparatornetzwerk modelliert werden, da die Reihenfolge der Vergleiche von der Eingabe abhängt.

3.4 Literatur zu Kapitel 3

[AKS83] M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in $c \cdot \log n$ parallel steps. *Combinatorica*, 3:1–19, 1983.

[Bat68] K. E. Batcher. Sorting networks and their applications. In *AFIPS Conf. Proc. 32*, pages 307–314, 1968.

[BH85] A. Borodin and J. E. Hopcroft. Routing, merging, and sorting on parallel models of computation. *Journal of Computer and System Sciences*, 30:130–145, 1985.

[Chv92] V. Chvátal. Lecture notes on the new AKS sorting network. Technical Report DCS-TR-294, Rutgers University, Computer Science Department, November 1992.

[Col88] R. Cole. Parallel merge sort. *SIAM Journal on Computing*, 17:770–785, 1988.

[CP93] R. Cypher and C. G. Plaxton. Deterministic sorting in nearly logarithmic time on the hypercube and related computers. *Journal of Computer and System Sciences*, 47:501–548, 1993.

[DPSR89] M. Dowd, Y. Perl, M. Saks, and L. Rudolph. The periodic balanced sorting network. *Journal of the ACM*, 36:738–757, 1989.

[Knu98] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, Reading, Massachusetts, 2nd edition, 1998.

[LP90] T. Leighton and C. G. Plaxton. A (fairly) simple circuit that (usually) sorts. In *Proceedings of the 31st IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 264–274, 1990.

[NS82] D. Nassimi and S. Sahni. Parallel permutation and sorting algorithms and a new generalized connection network. *Journal of the ACM*, 29:642–667, 1982.

[Par92] I. Parberry. The pairwise sorting network. *Parallel Processing Letters*, 2(2 & 3):205–211, 1992.

[Pat90] M. S. Paterson. Improved sorting networks with $O(\log n)$ depth. *Algorithmica*, 5:75–92, 1990.

[Pla92] C. G. Plaxton. A hypercubic network with nearly logarithmic depth. In *Proceedings of the 24th ACM Symposium on Theory of Computing (STOC)*, pages 405–416, 1992.

[Pra71] V. R. Pratt. *Shellsort and Sorting Networks*. PhD thesis, Stanford University, 1971.

[RV87] J. H. Reif and L. G. Valiant. A logarithmic time sort for linear size networks. *Journal of the ACM*, 34:60–76, 1987.

Kapitel 4

Ascend/Descend-Programme, das Cube-Connected Cycles- und das Shuffle-Exchange-Netzwerk

4.1 Ascend/Descend-Programme

Erinnern wir uns kurz zurück an die Arbeitsweise des Gitters $M(2,d)$, des d-dimensionalen Hypercubes, bei der Durchführung des Permutationsrouting in Abschnitt 2.3. Dort wurden die Dimensionen nacheinander zuerst in fallender, dann in steigender Folge durchlaufen, oder genauer: die Kommunikation in den ersten d Schritten verläuft hintereinander in Richtung der Dimension $d-1$, $d-2$, ..., 0, in den nächsten d Schritten rückwärts in Richtung der Dimension 0, 1, ..., $d-1$.

Ein ganz ähnliches Vorgehen konnten wir auch beim Bitonen Sortierer in Abschnitt 3.2 feststellen. Dort sind allerdings die Durchläufe durch die Dimensionen nicht vollständig, wie ein Blick auf S_8 auf Seite 22 zeigt. Füllen wir die „fehlenden“ Kommunikationsrunden mit dummy-Befehlen auf, so haben wir wieder dasselbe Muster wie beim Permutieren.

Wir durchlaufen also die natürliche Ordnung der Dimensionen auf dem Hypercube steigend oder fallend und führen auf den Prozessoren, die in der jeweils aktuellen Dimension zueinander benachbart sind, eine Operation aus. Ein solches Vorgehen ist für viele, gerade auch rekursive Algorithmen typisch. Denn die ersten $d-1$ Kommunikationen auf einem $M(2,d)$ sehen dann so aus, daß ein Problem auf zwei $M(2,d-1)$ gelöst wird, und daß dann im letzten Kommunikationsschritt aus den beiden Lösungen eine Lösung für $M(2,d)$ konstruiert wird.

Formal führen wir darum das Programmierschema der Ascend/Descend-Programme ein, in dem wir Algorithmen wie die oben beschriebenen einfach niederschreiben können.

Mit Hilfe des so beschriebenen Schemas können wir uns dann dem Problem zuwenden, den Nachteil des Hypercubes zu umgehen, der darin besteht, daß der Grad der Prozessoren ja d ist, d. h. von der Größe des Hypercubes abhängt, indem wir Ascend/Descend-Programme auch effizient auf Netzwerken mit konstantem Grad simulieren.

4.1 Definition: (Ascend/Descend-Programme)
Zu $\bar{a} \in [2]^d$ und $\ell \in [d]$ sei $\bar{a}(\ell) = (a_{d-1}, \ldots, a_{\ell+1}, \bar{a}_\ell, a_{\ell-1}, \ldots, a_0)$. Ein Ascend-Programm auf $M(2,d)$ ist von der Form:

`var` T : `array` $[2]^d$ `of` Information ;

```
procedure ASCEND(procedure Oper) ;
   var   ā : [2]^d ;
         dim : [d] ;
   begin
     for dim := 0 to d − 1 do
       for alle Prozessoren ā ∈ [2]^d mit a_dim = 0 do in parallel
         Die Prozessoren ā und ā(dim) führen Oper(ā, ā(dim), dim, T[ā], T[ā(dim)]) aus ;
   end ;
```

Ein Descend-Programm auf $M(2,d)$ ist von der Form:

```
procedure DESCEND(procedure Oper) ;
   var   ā : [2]^d ;
         dim : [d] ;
   begin
     for dim := d − 1 downto 0 do
       for alle Prozessoren ā ∈ [2]^d mit a_dim = 0 do in parallel
         Die Prozessoren ā und ā(dim) führen Oper(ā, ā(dim), dim, T[ā], T[ā(dim)]) aus ;
   end ;
```

Wichtig dabei ist, daß die Prozedur Oper ausschließlich von der Dimension, den beiden Nummern der in dieser Dimension verbundenen Prozessoren und den in den beiden Prozessoren gespeicherten Informationen abhängen darf. Die Werte von $T[\bar{a}]$ und $T[\bar{a}(dim)]$ können verändert werden, sind also call-by-reference-Parameter.

Die einmalige Anwendung von ASCEND(Oper) bzw. DESCEND(Oper) nennen wir einen **Ascend-** bzw. **Descend-Lauf**. Die Anwendung mehrerer Operationen und Läufe nennen wir **Ascend/Descend-Programm**.

Da in der überwiegenden Zahl der Anwendungen die Laufzeit von Oper konstant ist, nehmen wir dies im weiteren Verlauf an.

4.2 Beispiel:
Die Informationen der Prozessoren seien natürliche Zahlen, also für unser obiges Schema:

```
type Information = integer ;
```

Betrachte die folgende Operation minimum:

```
procedure minimum(x, y : [2]^d; dim : [d]; var T_1, T_2 : Information) ;
   begin
     if (x_dim, ..., x_0) = 0^(dim+1)
         then T_1 := min{T_1, T_2}
   end ;
```

Der Ascend-Lauf ASCEND(minimum) führt dann dazu, daß im Prozessor 0 das Minimum der Eingabe steht. Abbildung 4.1 zeigt, wie das Programm auf $M(2,3)$ abgearbeitet wird.

Die folgende Bemerkung weist auf eine weitere Beziehung zwischen Ascend/Descend-Programmen und rekursiven Algorithmen für den Hypercube hin.

4.3 Bemerkung:
Ein Ascend-Lauf auf $M(2,d)$ ist das gleiche wie die Ausführung der Operation Oper *zwischen zwei Exemplaren von $M(2,d-1)$ auf Prozessoren mit gleicher Nummer (bzgl. $M(2,d-1)$), die sich in $M(2,d)$ im 0-ten Bit unterscheiden (also der Kommunikation in die Richtung der Dimension 0), und einem Ascend-Lauf auf den beiden $M(2,d-1)$.*

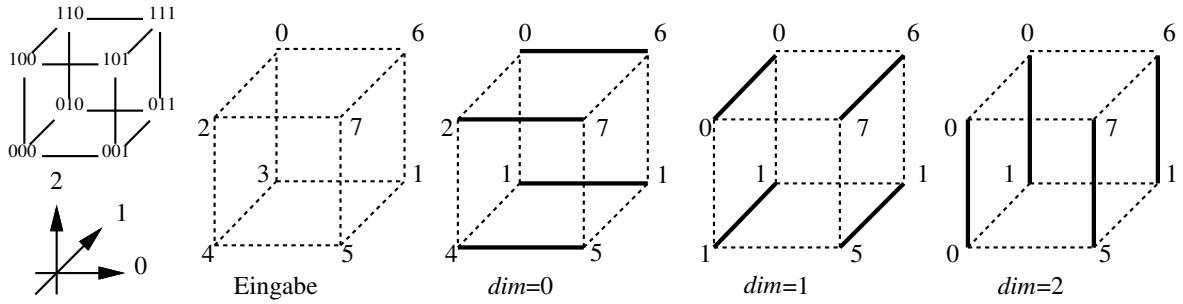


Abbildung 4.1: Minimum-Berechnung durch einen Ascend-Lauf.

Ein Descend-Lauf auf $M(2,d)$ ist das gleiche wie die Ausführung der Operation Oper *zwischen zwei Exemplaren von $M(2,d-1)$ auf Prozessoren mit gleicher Nummer (bzgl. $M(2,d-1)$), die sich in $M(2,d)$ im $(d-1)$-ten Bit unterscheiden (also der Kommunikation in die Richtung der Dimension $d-1$), und einem Descend-Lauf auf den beiden $M(2,d-1)$.*

Wegen unserer einleitenden Diskussion können wir den folgenden Satz formulieren:

4.4 Satz:

a) *Mit Hilfe eines Descend- und eines Ascend-Laufs kann auf $M(2,d)$ in Zeit $2d$ bezüglich einer beliebigen Permutation π mit Preprocessing permutiert werden.*
Im Preprocessing werden für jeden Prozessor Hilfsinformationen berechnet, die dann in „Information“ gespeichert sind.

b) *Mit Hilfe von d Descend-Läufen kann auf $M(2,d)$ in Zeit $O(d^2)$ sortiert werden.*

Weitere Beispiele für Probleme, für die es effiziente Ascend/Descend-Programme gibt, sind die schnelle Fourier-Transformation (FFT), die Matrix-Multiplikation und als Folge davon Berechnung von Zusammenhangskomponenten auf Graphen und ähnliches mehr. Auch viele Permutationen können bereits ohne Preprocessing durch kurze Ascend/Descend-Programme realisiert werden.

Der Vorteil dieses Schemas liegt auf der Hand: um einen Algorithmus zu beschreiben, reicht es aus, die Operation Oper näher zu spezifizieren, danach kann er direkt auf dem Hypercube „implementiert“ werden.

Der oben angeschnittene Nachteil des hohen Grades wird damit ja nun nicht ausgeräumt. Diesen Nachteil zu vermeiden, schaffen wir, wenn wir die Ascend/Descend-Programme auf Netzwerken simulieren, deren Grad nicht von der Netzwerkgröße abhängt. Natürlich soll unser Ziel dabei auch sein, diese Simulation so schnell wie möglich zu gestalten und dabei die Zahl der Prozessoren nicht anwachsen zu lassen.

Als erstes Netzwerk werden wir das lineare Array der Länge $n = 2^d$, also $M(n,1)$, daraufhin untersuchen, wie man auf ihm Ascend/Descend-Programme für $M(2,d)$ abarbeiten kann. Wir werden dann sehen, daß der dabei auftretende Zeitverlust (natürlich) sehr groß ist (schließlich ist der Durchmesser des linearen Arrays sehr groß). Danach werden wir ein Netzwerk kennenlernen, das die Simulation sehr viel effizienter durchführen kann und dabei die Simulation des Arrays ausnutzt (Die Arrays werden dann nur sehr viel kürzer sein, so daß der Zeitverlust dann nicht ins Gewicht fällt; dies ist ein oft benutzter Trick im Entwurf paralleler Algorithmen).

4.2 Ascend/Descend-Programme auf dem linearen Array

In den folgenden Untersuchungen gehen wir davon aus, daß die Operationen Oper immer in konstanter Zeit ausgeführt werden können. D. h. also, daß ein Ascend- oder Descend-Lauf auf $M(2,d)$ immer $O(d)$ Schritte benötigt.
Weiterhin sei noch darauf hingewiesen, daß wir eine beliebige natürliche Zahl mit ihrer Binärdarstellung identifizieren.
Die Idee, einen Ascend- oder Descend-Lauf auf dem linearen Array zu simulieren, besteht darin, die Informationen T so geschickt in benachbarte Prozessoren zu bringen, daß schon „fast“ die Vorbereitung für den nächsten Schritt gemacht worden ist. Dazu ist die folgende Permutation *Unshuffle* (*to shuffle* = (die Karten) mischen) besonders geeignet.

4.5 Definition:
Seien $d, d' \in \mathbb{N}$, $d' < d$, $n = 2^d$. $Unshuffle_{d'} : [2]^d \to [2]^d$ ist die folgende Permutation:

$$Unshuffle_{d'}(i_{d-1} \dots i_{d-d'} i_{d-d'-1} \dots i_1 i_0) := (\underbrace{i_{d-1} \dots i_{d-d'}}_{\text{unverändert}} \underbrace{i_0 i_{d-d'-1} \dots i_1}_{\text{zykl. Rechtsshift}})$$

4.6 Beispiel:
Für $d = 4$, d. h. $n = 16$, sind Abbildung 4.2 $Unshuffle_0$, $Unshuffle_1$ und $Unshuffle_2$ durch die Striche dargestellt, während die Zahlen die Hintereinanderausführung der Permutationen darstellen. Man achte schon jetzt darauf, daß in benachbarten Positionen gerade die Nummern der Prozessoren stehen, die in einem Ascend-Lauf miteinander kommunizieren müssen. Allgemein können wir folgendes sagen: Die Ausführung von $Unshuffle_{d'}$ findet auf Teilarrays

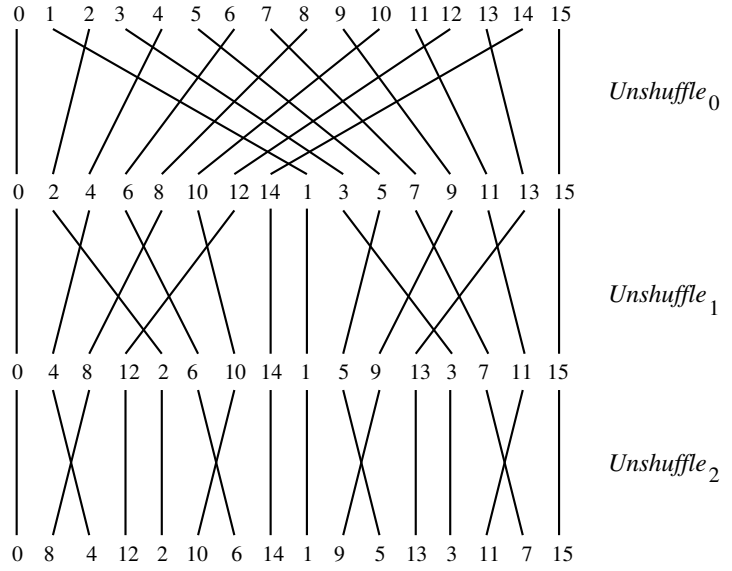


Abbildung 4.2: $Unshuffle_0$, $Unshuffle_1$ und $Unshuffle_2$

der Länge $2^{d-d'}$ statt, die durch die 0-1-Folgen der Länge d', die unverändert bleiben, eindeutig identifiziert sind. In jedem Teilarray werden die Elemente mit gerader Nummer, also diejenigen mit einer 0 als letztem Bit, in die vordere Hälfte ihres Teilarrays gebracht, und die mit

ungerader Nummer, also die mit einer 1 als letztem Bit, in die hintere Hälfte ihres Teilarrays, ohne daß dabei die Reihenfolge der Elemente beider Teilfolgen sich untereinander ändert.

4.7 Lemma:
Seien $d, d' \in \mathbb{N}$, $d' < d$, $n = 2^d$. Die Ausführung der Permutationen $Unshuffle_{d'}$ und $Unshuffle_{d'}^{-1}$ auf $M(n,1)$ dauert $O(2^{d-d'})$ Schritte, sofern d' allen Prozessoren bekannt ist.

Beweis:
Offensichtlich wird bei der Ausführung von $Unshuffle_{d'}$ nur auf $2^{d'}$ vielen Teilarrays geroutet, die zu $M(2^{d-d'},1)$ isomorph sind. Da jeder Prozessor aus d' und seiner Adresse berechnen kann, zu welchem Teilarray er gehört und wohin sein Paket geschickt werden muß, können wir nach Satz 2.7 auf $M(2^{d-d'},1)$ in $O(2^{d-d'})$ Schritten routen.
Analog kann für $Unshuffle_{d'}^{-1}$ argumentiert werden. □

Wie schon im obigen Beispiel verdeutlicht, liefert die Permutation $Unshuffle_{d'}$ die Möglichkeit, Ascend/Descend-Läufe auf dem linearen Array zu simulieren.

4.8 Satz:
Sei $n = 2^d$. $M(n,1)$ kann einen Ascend- oder einen Descend-Lauf in $O(n)$ Schritten abarbeiten.

Beweis:
Betrachte die folgende Prozedur als Ersatz für die Prozedur ASCEND aus Definition 4.1:

```
procedure ASCEND_lin_Array(procedure Oper) ;
   var ᾱ : [2]^{d-1} ;
       ā : [2]^d ;
       c : array [2]^d of [2]^d ;
       dim: [d] ;
begin
   for alle ā ∈ [2]^d do in parallel
      c[ā] := ā ;
   for dim := 0 to d − 1 do    (* Hauptschleife *)
      for alle ᾱ ∈ [2]^{d-1} do in parallel
         Die Prozessoren ᾱ0 und ᾱ1 führen
         Oper(c[ᾱ0], c[ᾱ1], dim, T[c[ᾱ0]], T[c[ᾱ1]]) aus ;                    (1)
      Route die Botschaften x_ā:=[c[ā], T[c[ā]]] mit ā ∈ [2]^d gemäß Unshuffle_dim ;   (2)
      for alle ā ∈ [2]^d do in parallel
         c[ā] := c[Unshuffle^{-1}_dim(ā)] ;                                   (3)
   done
   for dim:= d − 2 downto 0 do
      Route die Botschaften x_ā:=[c[ā], T[c[ā]]] mit ā ∈ [2]^d gemäß Unshuffle^{-1}_dim ;
end ;
```

Dieser Algorithmus führt nichts anderes aus als das, was in Abbildung 4.2 auf Seite 29 dargestellt ist. Die Beschriftung der Zeilen ist dabei der jeweilige Wert von $c[\bar{a}]$, der durch die Anweisung (3) aktualisiert wird. Wir sagen, daß der Prozessor $\bar{a}$ von $M(n,1)$ den Prozessor $c[\bar{a}]$ des Hypercubes **simuliert**. Denn dies kann er, wenn er die Information $T[c[\bar{a}]]$ kennt. Die Korrektheit des Algorithmus wird durch Induktion nach der Dimension d gezeigt:
Für $d = 1$ arbeitet der Algorithmus offensichtlich korrekt.

Sei nun $d > 1$. Wir nehmen an, daß er für $d-1$ korrekt arbeitet. Die Idee besteht darin, zu erkennen, daß, nachdem die Hauptschleife für $dim = 0$ durchlaufen ist, der Algorithmus auf zwei Exemplaren von $M(2^{d-1}, 1)$ ausgeführt wird, für die er nach Annahme richtig arbeitet. Die Korrektheit folgt dann zusammen mit der Beobachtung aus Bemerkung 4.3. Was also zu zeigen ist, ist,

a) daß er für $dim = 0$ zu Beginn des Schleifenrumpfes die Anweisung (1) ausführen kann, und

b) daß nach Durchführung der Permutation $Unshuffle_0$ sich die Informationen derart in benachbarten Prozessoren befinden, daß nach Weglassen des 0-ten Bits in allen Ausgangsadressen der Informationen zwei normal numerierte lineare Arrays $M(n/2, 1)$ entstehen.

a) ist offensichtlich erfüllt, und b) gilt wegen der in Beispiel 4.6 erkannten Eigenschaft von $Unshuffle_0$, die Elemente mit gerader Nummer in die vordere Hälfte und die mit ungerader Nummer in die hintere Hälfte des Arrays zu bringen, ohne daß dabei die Reihenfolge der Elemente beider Teilfolgen sich untereinander ändert.

Am Ende des Algorithmus ist wieder jede Information $T[\bar{a}]$ im Prozessor $\bar{a}$.

Wir können ähnlich auch eine Prozedur DESCEND_lin_Array angeben. Diese sieht so aus wie ASCEND_lin_Array, wobei zu Beginn die Botschaften $\boxed{\bar{a}, T[\bar{a}]}$ für $i = 0$ bis $d-2$ gemäß $Unshuffle_i$ geroutet werden[1]. Es wird also die Situation hergestellt, wie sie in Abbildung 4.2 auf Seite 29 in der letzten Zeile zu sehen ist. Dann kann der obige Algorithmus ausgeführt werden, wobei dim von $d-1$ bis 0 herabgezählt wird, und statt $Unshuffle$ die Permutation $Unshuffle^{-1}$ geroutet wird. Der Algorithmus zur Simulation von Ascend wird also „rückwärts“ durchlaufen. Diese Simulation eines Descend-Laufs ist dann wegen der Diskussion für ASCEND_lin_Array korrekt.

Die Laufzeit beider Prozeduren wird durch die Laufzeit für die Ausführung der $Unshuffle$-Permutationen in (2) bestimmt, ist nach Lemma 4.7 also

$$O\Big(2 \cdot \sum_{dim=0}^{d-1} 2^{d-dim}\Big) = O(2^d) = O(n),$$

womit die Behauptung bewiesen ist. □

Der Zeitverlust (d. h. das Verhältnis der beiden Laufzeiten der Ascend/Descend-Läufe) unserer Simulation ist also $O(n/\log n)$. Da Prozessor 0 mit Prozessor $\frac{1}{2}n$ kommunizieren muß, und da $\mathrm{dist}_{M(n,1)}(0, \frac{1}{2}n) = \frac{1}{2}n = \Theta(n)$ ist, gibt es kein Verfahren, das bzgl. der Größenordnung schneller ist als das oben vorgestellte. D. h., daß wir Ascend/Descend-Programme zwar mit asymptotisch optimalem Zeitverlust auf dem linearen Array abarbeiten können, daß dieser Zeitverlust aber viel zu groß ist, um das lineare Array als guten Simulator für Ascend/Descend-Programme zu verwenden. Dazu ist das im folgenden Abschnitt vorgestelle Netzwerk viel besser geeignet.

[1] Die so ausgeführte Permutation wird übrigens **Bit-Reversal-Permutation** (BRP) genannt, da als Ergebnis jede Botschaft, die anfangs im Prozessor $(a_{d-1} \dots a_0)$ stand, sich in Prozessor $(a_0 \dots a_{d-1})$ befindet. Daß dies der Fall ist, möge sich der Leser überlegen.

4.3 Das Cube-Connected Cycles-Netzwerk

Um den hohen Grad des Hypercubes $M(2,d)$ zu reduzieren, wenden wir die folgende Konstruktion an. Jeder Prozessor von $M(2,d)$ wird durch einen Kreis der Länge d von Prozessoren ersetzt, wobei der Kreis sich an der Numerierung der Dimensionen orientiert. Formal ergibt sich folgendes Netzwerk:

4.9 Definition:
Sei $d \in I\!N$. Das **Cube-Connected Cycles-Netzwerk** CCC(d) ist das Netzwerk mit Prozessormenge $\mathcal{P} = \{(\bar{a}, p) \mid \bar{a} \in [2]^d, p \in [d]\}$ und Kommunikationsgraph $(\mathcal{P}, E)$, wobei

$$\begin{aligned} E &= \Big\{\{(\bar{a},p),(\bar{a},(p+1) \bmod d)\} \mid \bar{a} \in [2]^d, p \in [d]\Big\} \\ &\quad \cup \Big\{\{(\bar{a},p),(\bar{b},p)\} \mid \bar{a},\bar{b} \in [2]^d, p \in [d], \bar{a}(p) = \bar{b}\Big\} \end{aligned}$$

ist. Die Kanten der ersten Menge heißen Kreiskanten, die der zweiten Würfelkanten. Den Würfelkanten weisen wir in natürlicher Weise die Dimensionen 0 bis $d-1$ zu.

4.10 Beispiel:
Abbildung 4.3 zeigt zwei Darstellungen für CCC(3).

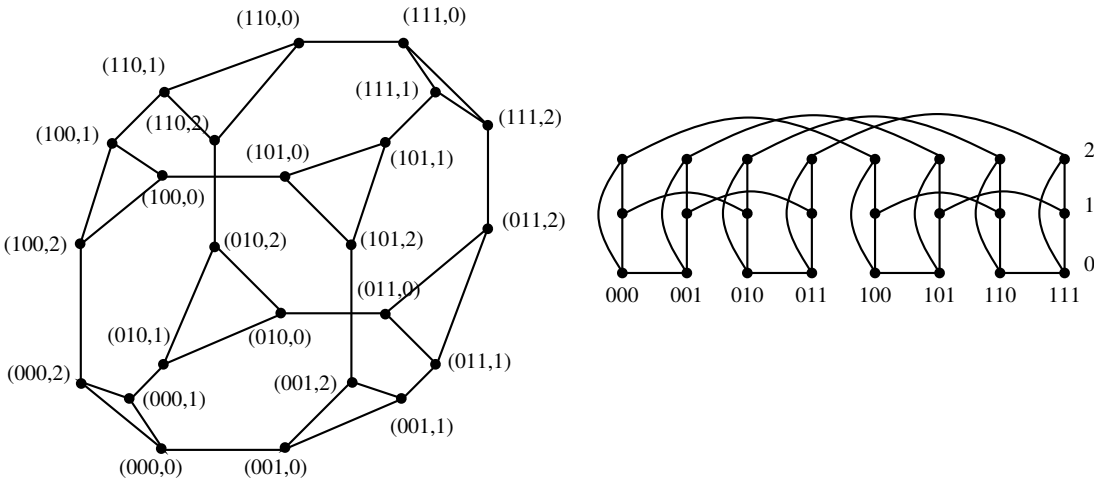


Abbildung 4.3: CCC(3).

4.11 Bemerkung:

a) *CCC(d) hat $d \cdot 2^d$ Knoten, $3d \cdot 2^{d-1}$ Kanten und Grad 3.*

b) *Der Durchmesser von CCC(d) ist $\lfloor \frac{5}{2}d \rfloor - 2$ für $d > 3$.*

c) *CCC(d) besitzt einen Hamiltonschen Kreis.*

4.12 Satz:
Sei $d = 2^k$. CCC(d) kann einen Ascend- oder einen Descend-Lauf für $M(2, k+d)$ in Zeit $O(d)$ abarbeiten, d. h. es ist bis auf einen konstanten Faktor so schnell wie $M(2, k+d)$.

Beweis:
Sei $p \in [d]$ und $(p_{k-1} \dots p_0)_2$ die Binärdarstellung von p. Sei weiter $\bar{a} \in [2]^d$. Zu Beginn soll der Prozessor $(\bar{a}, p)$ von CCC(d) den Prozessor $(a_{d-1} \dots a_0 p_{k-1} \dots p_0)$ von $M(2, d+k)$ simulieren. Wir beschreiben, wie CCC(d) einen Ascend-Lauf von $M(2, k+d)$ abarbeitet. Die Simulation eines Descend-Laufs kann dann wieder analog durchgeführt werden.
Die Simulation eines Ascend-Laufs führen wir in zwei Phasen durch.

Phase (1): Die Kommunikation entlang der Dimensionen 0 bis $k-1$ von $M(2, d+k)$ erfordert ausschließlich Kommunikationen zwischen Prozessoren, die sich auf demselben Kreis der Länge d befinden. Auf diesem können wir den Algorithmus ASCEND_lin_Array ausführen. Nach Satz 4.8 ist die dafür benötigte Zeit $O(d)$. Für jeden Prozessor $\bar{a} = (a_{d+k-1} \dots a_k \underbrace{a_{k-1} \dots a_0}_{(*)})$ von $M(2, k+2^k)$ sind abschließend die Kommunikationen mit Prozessoren, die sich in den Bitpositionen $(*)$ von $\bar{a}$ unterscheiden, durchgeführt.

Phase (2): Nach der ersten Phase bleibt nur noch ein Ascend-Lauf auf $M(2, d)$ übrig, da die Prozessoren auf den Kreisen ihre Kommunikation untereinander abgeschlossen haben. D. h., daß jeder Prozessor noch eine Kommunikation in den Richtungen 0 bis $d-1$ des CCC(d) durchzuführen hat, die den Dimensionen k bis $d+k-1$ von $M(2, d+k)$ entsprechen. Dies können wir realisieren, indem wir die Kreise so an den Würfelkanten der Dimensionen 0, 1, …, $d-1$ vorbeidrehen, daß an den Kanten der Dimension *dim* des CCC(d) gerade die Prozessoren zu stehen kommen, die entlang der Dimension $dim + k$ miteinander kommunizieren wollen. Unter „Drehen“ verstehen wir hier natürlich den Transport der Informationen T. Man beachte, daß in dieser Simulation viele Dimensionen des CCC(d) gleichzeitig genutzt werden, und daß nicht alle Kommunikationen einer Dimension des Hypercubes gleichzeitig abgearbeitet werden. Nach den ersten $d-1$ Drehungen (d. h. Transporten von Prozessor $(\bar{a}, p)$ nach Prozessor $(\bar{a}, (p+1) \bmod d)$) ist der Prozessor $(\bar{a}, 1)$ bei der Würfelkante der Dimension 0 angekommen und kann dann erst die Kommunikation in Richtung k ausführen. (Vgl. auch das Bild in Bsp. 4.13.) Wegen dieses Pipelining-Effekts müssen die Kreise zweimal vollständig gedreht werden. Die Laufzeit der Phase (2) ist somit $O(d)$.

Nachdem diese beiden Phasen ausgeführt sind, ist der gesamte Ascend-Lauf für $M(2, d+k)$ in Zeit $O(d)$ simuliert worden. □

Bei der Simulation von Ascend/Descend-Programmen auf dem linearen Array in Abschnitt 4.2 ist es so, daß alle Prozessoren immer gleichzeitig an der Simulation der Rechenschritte mit Kommunikation in die gleiche Richtung arbeiten. Dies ist bei der oben beschriebenen Simulation auf CCC(d) nicht der Fall, wie man auch im folgenden Beispiel sehen kann. Einige Prozessoren des Hypercubes sind schon weiter simuliert worden als andere, die Simulation läuft gewissermaßen asynchron ab. Diese Eigenschaft zeichnet diese Simulation auch gegenüber der im nächsten Abschnitt vorgestellten Simulation aus.

4.13 Beispiel:
Abbildung 4.4 zeigt die Phase (2) für den durch 1011 charakterisierten Kreis der CCC(4), die den $M(2, 6)$ simulieren. Die gestrichelten Linien stellen ungenutzte Würfelkanten dar, die Werte in den eckigen Klammern geben die nach dem aktuellen Schritt höchste simulierte Dimension

von $M(2,6)$ an. Die Phase (2) beginnt damit, daß alle Prozessoren die Kommunikationen der Dimensionen 0 und 1 bereits ausgeführt haben.

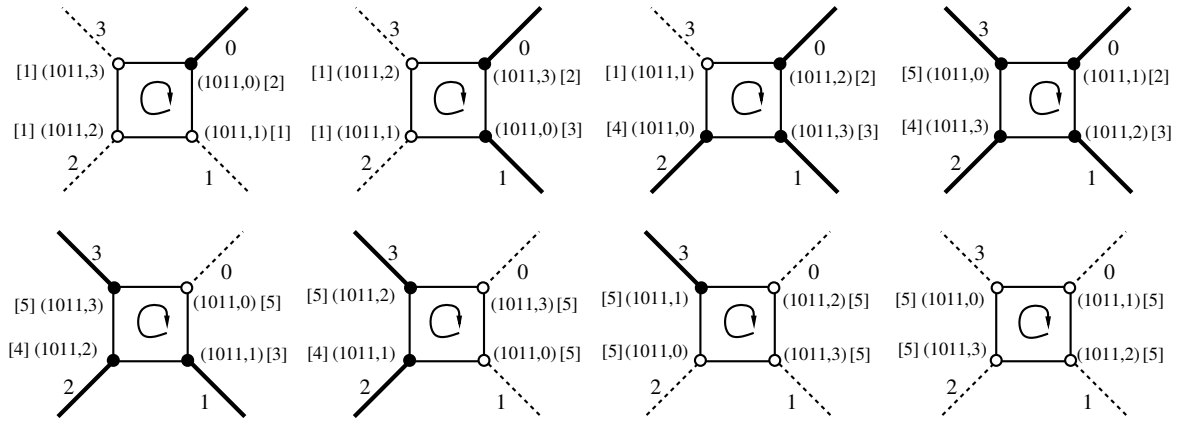


Abbildung 4.4: Teil der Simulation auf CCC(4).

4.14 Korollar:
Sei $d = 2^k$, $n = d \cdot 2^d$. Dann gilt:

a) *Auf CCC(d) können beliebige Permutationen mit Preprocessing in Zeit $O(\log n)$ geroutet werden. Die Zeit für das Preprocessing ist $O((\log n)^4)$.*

b) *Auf CCC(d) können n Zahlen in Zeit $O((\log n)^2)$ sortiert werden.*

4.4 Das Shuffle-Exchange-Netzwerk

In diesem Abschnitt lernen wir ein weiteres Netzwerk kennen, das in der Lage ist, Ascend/Descend-Programme effizient abzuarbeiten, das sogenannte Shuffle-Exchange-Netzwerk. Analog zur Simulation auf dem linearen Array benutzen wir auch hier ein Paar von Permutationen, das es uns ermöglicht, die Informationen in benachbarte Prozessoren zu schicken, die benötigt werden, um einen Schritt ausführen zu können.

4.15 Definition:
Sei $d \in \mathbb{N}$.

a) $lshift : [2]^d \to [2]^d$ ist die folgende Permutation: $lshift(a_{d-1}a_{d-2}\dots a_0) = (a_{d-2}\dots a_0 a_{d-1})$. Es wird auf den Bits ein **zyklischer Linksshift** durchgeführt.

b) $rshift : [2]^d \to [2]^d$ ist die folgende Permutation: $rshift(a_{d-1}\dots a_1 a_0) = (a_0 a_{d-1}\dots a_1)$. Es wird auf den Bits ein **zyklischer Rechtsshift** durchgeführt.

Für $i \in \mathbb{N}$ bezeichnen wir mit $lshift^i$ bzw. $rshift^i$ die i-fache Hintereinanderausführung des zyklischen Links- bzw. Rechtsshifts.
Offensichtlich ist *lshift* die Umkehrpermutation von *rshift* und umgekehrt.

Im Gegensatz zum Vorgehen in Abschnitt 4.2, wo wir die Permutation *Unshuffle* auf dem linearen Array routen mußten, definieren wir uns hier ein Netzwerk, das gerade so konstruiert ist, daß ein Rechts- oder Linksshift der Informationen direkt durchgeführt werden kann.

4.16 Definition:
Das **Shuffle-Exchange-Netzwerk** SE(d) ist das Netzwerk mit Prozessormenge $\mathcal{P} = [2]^d$ und Kommunikationsgraph $(\mathcal{P}, E)$, wobei

$$\begin{aligned} E &= \left\{\{a\bar{\alpha}, \bar{\alpha}a\} \mid a \in [2], \bar{\alpha} \in [2]^{d-1}\right\} \\ &\quad \cup \left\{\{\bar{\alpha}0, \bar{\alpha}1\} \mid \bar{\alpha} \in [2]^{d-1}\right\} \end{aligned}$$

ist. Die Kanten der ersten Menge heißen Shuffle-Kanten, die der zweiten Exchange-Kanten.

4.17 Beispiel:
Abbildung 4.5 zeigt SE(3) und SE(4). Im Gegensatz zu CCC(d), das wir strukturiert zeichnen können, ist für das Shuffle-Exchange-Netzwerk keine einfache und anschauliche Darstellung bekannt.

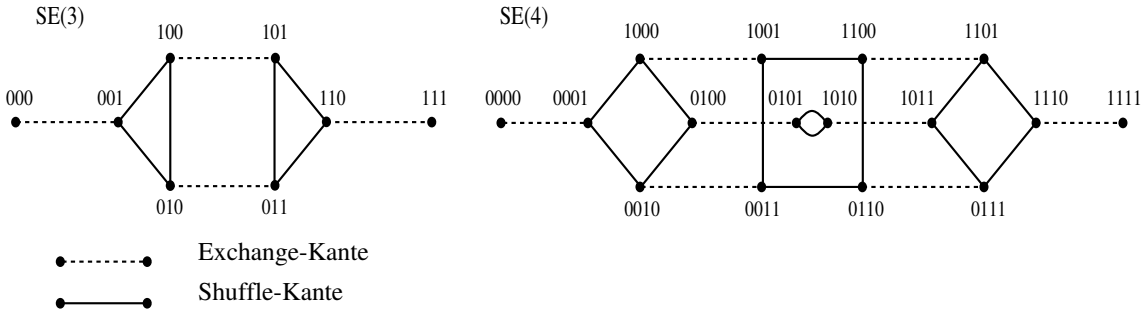


Abbildung 4.5: SE(3) und SE(4).

4.18 Bemerkung:

a) SE(d) hat 2^d Prozessoren, höchstens $\frac{3}{2} \cdot 2^d$ Kanten und maximalen Grad 3.

b) SE(d) hat den Durchmesser $2d - 1$.

c) SE(d) besitzt einen Hamiltonschen Pfad.

4.19 Satz:
SE(d) *kann einen Ascend- oder einen Descend-Lauf für $M(2, d)$ in Zeit $O(d)$ abarbeiten.*

Beweis:
Wir simulieren wieder einen Ascend-Lauf und verweisen darauf, daß die Simulation von Descend dann analog verläuft.

```
procedure ASCEND_Shuffle_Exchange(procedure Oper) ;
   var ᾱ : [2]^{d-1} ;
       ā : [2]^d ;
       c : array [2]^d of [2]^d ;
       dim: [d] ;
begin
   for alle ā ∈ [2]^d do in parallel
      c[ā] := ā ;
   for dim := 0 to d − 1 do    (* Hauptschleife *)
      for alle ᾱ ∈ [2]^{d-1} do in parallel
```

Die Prozessoren $\bar{\alpha}0$ und $\bar{\alpha}1$ führen
Oper$(c[\bar{\alpha}0], c[\bar{\alpha}1], dim, T[c[\bar{\alpha}0]], T[c[\bar{\alpha}1]])$ aus ; (1)
Route die Botschaften $x_{\bar{a}} := \boxed{c[\bar{a}], T[c[\bar{a}]]}$ mit $\bar{a} \in [2]^d$ gemäß *rshift* ; (2)
`for` alle $\bar{a} \in [2]^d$ `do in parallel`
$c[\bar{a}] := c[lshift(\bar{a})]$; (3)
`done`
`end` ;

Offensichtlich simuliert der Prozessor $\bar{a}$ von SE(d) im *dim*-ten Durchlauf der Hauptschleife den Prozessor $rshift^{dim}(\bar{a})$ des Hypercubes $M(2, d)$. Da sich die durch die Exchange-Kanten verbundenen Prozessoren von SE(d) genau im letzten Bit unterscheiden, unterscheiden sich die von ihnen im *dim*-ten Durchlauf simulierten Prozessoren genau im *dim*-ten Bit, also kann der *dim*-te Schritt des Ascend-Laufs in (1) ausgeführt werden. Am Ende der Hauptschleife simuliert jeder Prozessor $\bar{a}$ auch wieder den Prozessor $\bar{a}$ des Hypercubes.

Die Laufzeit wird bestimmt durch die Zeit, die benötigt wird, um das Routing der Permutation *rshift* durchzuführen. Dies ist auf SE(d) über die Shuffle-Kanten in einem Schritt möglich, also ist die Gesamtlaufzeit $O(d)$.

Die Simulation eines Descend-Laufs verläuft entsprechend „rückwärts“, ähnlich zum Vorgehen auf dem linearen Array. □

4.20 Beispiel:
Abbildung 4.6 zeigt die Simulation eines Ascend-Laufs für $M(2, 3)$ auf SE(3).

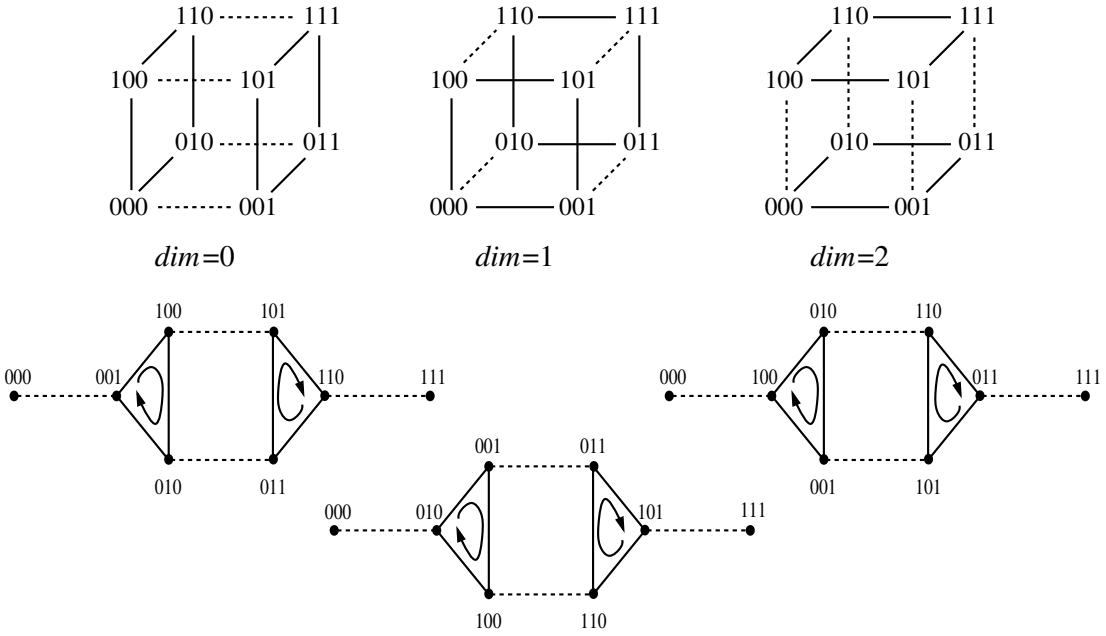


Abbildung 4.6: Simulation eines Ascend-Laufs auf SE(3).

4.21 Korollar:
Sei $n = 2^d$. Dann gilt:

a) *Auf SE(d) können beliebige Permutationen mit Preprocessing in Zeit $O(\log n)$ geroutet werden. Die Zeit für das Preprocessing ist $O((\log n)^4)$.*

b) *Auf SE(d) können n Zahlen in Zeit $O((\log n)^2)$ sortiert werden.*

4.5 Abschließende Bemerkungen

Ist die Laufzeit der Operation Oper nicht konstant, so können durch einmalige Anwendung von Oper mehr als konstant große Informationen erzeugt werden. Beim Drehen der Kreise in den CCC können hier Schwierigkeiten auftreten, da der Datentransport beim Pipelining so lange dauert, wie die längste Variable vom Typ „Information“ lang ist.

Das Cube-Connected Cycles-Netzwerk sowie die Beschreibung von Ascend/Descend-Programmen stammt aus[PV81]. Daß es einen Hamiltonschen Kreis besitzt, zeigen Schwartz und Loui[SL87]. Die einzige Stelle, an der wir ausnutzen, daß die Knoten des Hypercubes durch *Kreise* ersetzt worden sind, ist beim „Drehen“ der Kreise im Simulationsalgorithmus im Beweis von Satz 4.12. Parberry zeigt in [Par86], daß es ausreicht, die Knoten des Hypercubes durch lineare Arrays zu ersetzen, um Ascend/Descend-Programme ohne wesentlichen Zeitverlust zu simulieren. Das so beschriebene Netzwerk wird **Cube-Connected Lines-Netzwerk** genannt.

Für den Fall, daß die Dimension des durch die CCC zu simulierenden Hypercubes keine Zweierpotenz ist, haben Preparata und Vuillemin ihre Konstruktion der CCC dahingehend erweitert, daß die Kreise derart verlängert werden (also mehr Prozessoren erhalten), daß nicht alle mit weiteren Nachbarkreisen verbunden sind. Dann ist es möglich, beliebige Ascend/Descend-Läufe auf diesen modifizierten Cube-Connected Cycles zu simulieren.

Eine unendliche Version der Cube-Connected Cycles, in der für jedes beliebige d ein Anfangsstück abgespaltet werden kann, das die CCC(d) enthält, wird in[Mey83] eingeführt.

Das Shuffle-Exchange-Netzwerk stammt aus[Sto71]. Daß es einen Hamiltonschen Pfad besitzt, zeigen Feldmann/Mysliwietz[FM96].

Auch auf dem Butterfly-Netzwerk aus N Prozessoren können Ascend/Descend-Programme für den Hypercube gleicher Größe mit konstantem Zeitverlust implementiert werden[Sch93].

Das Konzept der Ascend/Descend-Programme wird von Leighton[Lei92] zum Konzept der **normalen Hypercube-Algorithmen** erweitert. Hierbei wird nicht gefordert, daß die Dimensionen vollständig in einer Richtung durchlaufen werden, sondern hier ist es wesentlich, daß nach der Kommunikation in Richtung der Dimension dim die Dimension $dim+1$ oder die Dimension $dim-1$, jeweils modulo d, benutzt wird.

4.6 Literatur zu Kapitel 4

[FM96] R. Feldmann and P. Mysliwietz. The Shuffle Exchange network has a Hamiltonian path. *Mathematical Systems Theory*, 29:471–485, 1996.

[Lei92] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[Mey83] F. Meyer auf der Heide. Infinite Cube-Connected Cycles. *Information Processing Letters*, 16:1–2, 1983.

[Par86] I. Parberry. On recurrent and recursive interconnection patterns. *Information Processing Letters*, 22:285–289, 1986.

[PV81] F. Preparata and J. Vuillemin. The Cube-Connected Cycles: a versatile network for parallel computation. *Communications of the ACM*, 24:300–309, 1981.

[Sch93] E. J. Schwabe. Constant-slowdown simulations of normal hypercube algorithms on the butterfly network. *Information Processing Letters*, 45:295–301, 1993.

[SL87] A. M. Schwartz and M. C. Loui. Dictionary machines on cube-class networks. *IEEE Transactions on Computers*, 36:100–105, 1987.

[Sto71] H. Stone. Parallel processing with the perfect shuffle. *IEEE Transactions on Computers*, 20:153–161, 1971.

Kapitel 5

Simulationen zwischen Netzwerken

5.1 Simulationstechniken

In diesem Kapitel beschäftigen wir uns damit, wie Netzwerke andere Netzwerke simulieren können. Derartige Simulationen haben wir schon mehrfach (informal) benutzt. So haben wir das Permutationsnetzwerk auf den Gittern simuliert, das Bitone Sortiernetzwerk von Batcher auf dem Hypercube und schließlich Ascend/Descend-Programme, die ja für den Hypercube entworfen wurden, auf dem linearen Array, dem Cube-Connected Cycles-Netzwerk und dem Shuffle-Exchange-Netzwerk.

Die allgemeine Wichtigkeit der gegenseitigen Simulation besteht darin, daß es natürlich nicht vertretbar ist, sich für jedes Problem eine geeignete Verdrahtung der Prozessoren auszudenken, und diese dann real zu verwirklichen.

Wir unterscheiden drei verschiedene Simulationstypen ([Mey83a]):

5.1 Definition: (Simulationstypen)
Seien M und M_0 zwei Netzwerke.

a) Bei einer **Typ-1-Simulation** (**Einbettung**) wird jeder Prozessor von M durch genau einen Prozessor von M_0 simuliert.

b) Bei einer **Typ-2-Simulation** (**Mehrfacheinbettung**) wird jeder Prozessor von M durch mindestens einen Prozessor von M_0 simuliert.

c) Bei einer **Typ-3-Simulation** (**Dynamische Einbettung**) wird jeder Prozessor von M zu jedem Zeitpunkt durch mindestens einen Prozessor von M_0 simuliert.

5.2 Universelle Netzwerke

In Kapitel 4 haben wir eine Klasse von Algorithmen daraufhin untersucht, wie sie auf speziellen Netzwerken effizient abgearbeitet werden können. Wir hatten Ascend/Descend-Programme, die in erster Linie auf den Hypercube zugeschnitten sind, auf anderen Netzwerken simuliert.

In diesem Kapitel werden wir dieses Konzept derart verallgemeinern, daß wir spezielle Netzwerke konstruieren, die in der Lage sind, jeden Algorithmus für beliebige Netzwerke mit beschränktem Grad effizient zu simulieren.

Abschließend werden wir einige Grenzen dieser Simulationen kennenlernen.

5.2.1 Simulation von Netzwerken mit konstantem Grad

Die meisten Netzwerk-Familien, die in der Informatik untersucht werden, haben einen (unabhängig von der Netzwerkgröße) konstanten Grad. Bei den bislang untersuchten Netzwerken haben wir als solche Familien das Butterfly-Netzwerk mit Grad 4, das Cube-Connected-Cycles-Netzwerk und das Shuffle-Exchange-Netzwerk mit Grad 3 kennengelernt. Der Grund dafür wurde bereits mehrmals angesprochen: Netzwerke mit beliebigem Grad können nicht realisiert werden, da die Zahl der Links je Prozessor natürlich begrenzt ist.
Für jeden in der Praxis auftauchenden Anwendungsfall nun immer das „am besten“ geeignete Netzwerk wirklich zu bauen, ist nicht sinnvoll, da man damit die Vorteile anderer Netzwerke verlieren würde. Darum überlegt man sich, ob es vielleicht möglich ist, **eine** Netzwerk-Familie anzugeben, die in der Lage ist, alle anderen Netzwerke mit einem vertretbaren Aufwand an zusätzlichen Ressourcen (Zahl der Prozessoren, Rechenzeit) zu simulieren. Bevor wir eine solche Familie angeben, müssen wir erst einmal unser Modell genauer beschreiben.

5.2 Definition: (Konfiguration)
Sei M ein Netzwerk mit Prozessormenge $[p]$. Eine **Konfiguration** K_i eines Prozessors $i \in [p]$ ist die vollständige Beschreibung des Zustandes von i. Diese Beschreibung besteht aus dem Programm für i, dem aktuellen Inhalt des Befehlszählers und dem lokalen Speicher von i. $\mathcal{K} = (K_0, \ldots, K_{p-1})$ ist eine **Konfiguration** von M.
Falls M in der Konfiguration $\mathcal{K}$ ist und dann t Schritte ausführt und Konfiguration $\mathcal{K}' = (K'_0, \ldots, K'_{p-1})$ erreicht, ist $\mathcal{K}'$ die t-te **Nachfolgekonfiguration** von $\mathcal{K}$ und K'_i die t-te Nachfolgekonfiguration von $\mathcal{K}$ für Prozessor i.

5.3 Definition: (Simulation)
Sei $\mathcal{K} = (K_0, \ldots, K_{p-1})$ eine Konfiguration von M. Ein Netzwerk M_0 mit q Prozessoren **simuliert schwach** t Schritte eines Netzwerks M mit p Prozessoren ($q \geq p$) mit **Zeitverlust** k, falls

- es zu Beginn p nichtleere disjunkte Prozessormengen $A_0, \ldots, A_{p-1}$, $A_i \subseteq M_0$, gibt, so daß alle Prozessoren aus A_i die Konfiguration K_i des Prozessors P_i von M kennen (die A_i heißen **Vertreter** von P_i),
- und gilt: Führt M_0 höchstens $k \cdot t$ Schritte aus, so gibt es für alle i mindestens einen Prozessor $Q^{(i)}$ von M_0, der die t-te Nachfolgekonfiguration von $\mathcal{K}$ für P_i kennt.

M_0 **simuliert (stark)** t Schritte von M mit Zeitverlust k, falls

- M_0 das Netzwerk M mit Zeitverlust k schwach simuliert,
- und **jeder** Prozessor aus A_i die t-te Nachfolgekonfiguration von $\mathcal{K}$ für P_i kennt.

Beachte, daß wir nicht gesagt haben, daß die Simulation eines einzelnen Schrittes von M maximal k Schritte kostet, sondern daß wir über eine größere Anzahl an simulierten Schritten sprachen. Es kann also sein, daß z. B. t Schritte simuliert werden, wobei die Simulation der ersten $t-1$ Schritte den Zeitverlust 1 verursacht und die Simulation des Schrittes t genau t Schritte kostet. Insgesamt haben wir dann t Schritte mit Zeitverlust $k = (t-1+t)/t \approx 2$ simuliert. In solchen Fällen spricht man auch von **amortisiertem** Zeitverlust.

Beachte, daß die Zeit, die man für die (erste) Initialisierung von M_0 benötigt, nicht in den Zeitverlust mit eingeht.
Der Unterschied zwischen starker und schwacher Simulation besteht darin, daß wir nach der schwachen Simulation von t Schritten nicht wissen, ob M_0 die Simulation mit dem gegebenen Zeitverlust fortsetzen kann. Dagegen gilt für die Simulationen offensichtlich die folgende Bemerkung:

5.4 Bemerkung:
Kann M_0 t Schritte von M mit Zeitverlust k simulieren, so kann M_0 jede Anzahl $T \geq t$ an Schritten von M in Zeit $\lceil T/t \rceil \cdot t \cdot k$, also mit Zeitverlust höchstens $2k$ simulieren.

Allerdings kann man eine schwache Simulation in eine starke verwandeln, indem die Prozessoren $Q^{(i)}$, die die t-te Nachfolgekonfiguration kennen, dafür sorgen, daß alle Prozessoren aus A_i diese ebenfalls kennenlernen. Das kann aber unter Umständen sehr lange dauern, weshalb der Zeitverlust nicht beibehalten werden kann.

5.5 Definition:
Eine Klasse $\mathcal{F} = \{M_1, M_2, \ldots\}$ von Netzwerken heißt **realistisch**, falls es eine Konstante c gibt, so daß kein Netzwerk $M_i \in F$ einen Grad größer als c besitzt.
Wir werden nicht nur Klassen von Netzwerken, sondern auch einzelne Netzwerke als realistisch bezeichnen und meinen damit Netzwerke mit Grad höchstens 10.

5.6 Bemerkung:
Jedes realistische Netzwerk mit n Prozessoren kann mit konstantem Zeitverlust durch Netzwerke mit $O(n)$ Prozessoren und Grad 3 simuliert werden. Beachte, daß Grad 2 nicht ausreicht. (Man überlege sich, warum nicht.)

5.7 Definition:
Sei $\mathcal{F}$ eine realistische Klasse von Netzwerken. Ein realistisches Netzwerk M_0 ist **k-universell** für $\mathcal{F}$, falls es jedes Netzwerk $M \in \mathcal{F}$ mit Zeitverlust $O(k)$ simulieren kann. Ist $\mathcal{F}$ die Klasse aller realistischen Netzwerke vom Grad c mit n Prozessoren, nennen wir M_0 auch (n, k)-universell.

5.8 Beispiel:
Sei $\mathcal{K}(N)$ die Menge der Netzwerke mit N Knoten und maximalem Grad 2. Das in Abbildung 5.1 dargestellte Netzwerk UK(N) mit N Knoten ist 1-universell für $\mathcal{K}(N)$. D. h. alle Zusammensetzungen von Netzwerken aus Kreisen, linearen Arrays und einzelnen Prozessoren können hier direkt simuliert werden.

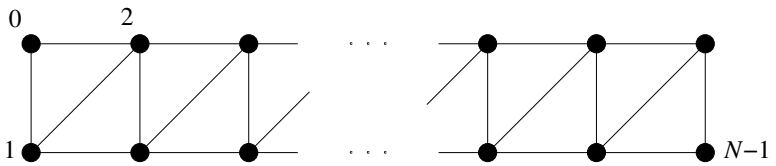


Abbildung 5.1: UK(N).

Als erstes werden wir zeigen, daß wir mit den bereits erzielten Resultaten auch schon ganz gute universelle Netzwerke haben.
Dazu nutzen wir das folgende graphentheoretische Lemma aus.

5.9 Lemma:
Jeder Graph vom Grad c ist $(c+1)$-kantenfärbbar. D. h.: Jeder Graph vom Grad c kann so in $c+1$ Matchings zerlegt werden, daß die Vereinigung der Kanten der Matchings gerade die Kantenmenge des Graphen ergibt.

Den Beweis dieses schönen (und klassischen) Resultats findet man z. B. in [Bol98, S. 153f.].

5.10 Satz:
a) *Sei $n = d \cdot 2^d$. CCC(d) ist $(n, \log n)$-universell.*

b) *Sei $n = 2^d$. SE(d) ist $(n, \log n)$-universell.*

Beweis:
a) Sei M ein realistisches Netzwerk mit n Prozessoren $\{P_0, \ldots, P_{n-1}\}$ und Grad c, und seien die Prozessoren von CCC(d) beliebig von 0 bis $n-1$ durchnumeriert.
Wir beschreiben zuerst, wie die Cube-Connected Cycles initialisiert werden. Man beachte, daß die hierzu benötigte Zeit nicht in den Zeitverlust mit eingeht.
Der Prozessor P_i von M wird durch den i-ten Prozessor von CCC(d) simuliert, d. h. Prozessor i kennt die Konfiguration K_i von P_i und soll die Nachfolgekonfiguration dazu berechnen.
Aus Korollar 4.14 wissen wir, daß die CCC(d) in Zeit $O(\log n)$ bezüglich einer fest vorgegebenen Permutation n Pakete permutieren können. Diese Eigenschaft benutzen wir, um die Werte, die kommuniziert werden sollen, zwischen den entsprechenden Prozessoren zu transportieren.
Da wir nicht im Voraus wissen, welche Prozessoren mit welchen kommunizieren wollen, gehen wir in jedem Schritt davon aus, daß jeder Prozessor mit allen seinen Nachbarn reden will. Wir werden sehen, daß dies nicht zu teuer ist, da M ja konstanten Grad hat.
Färbe jetzt also die Kanten von M mit höchstens $c+1$ vielen Farben aus $[c+1]$ gemäß Lemma 5.9. Zu jeder Farbe $j \in [c+1]$ wird eine partielle Permutation π_j konstruiert, für die gilt:

$$\pi_j(i) = \ell, \text{ falls } \{P_i, P_\ell\} \text{ Kante im Graph } M \text{ mit Farbe } j \text{ ist.}$$

Wir präparieren jetzt CCC(d) für die Permutationen $\pi_0, \ldots, \pi_c$. Da auch diese Präparationen zu Beginn der Simulation ausgeführt werden und nur vom Graphen M abhängen — sich also während der Simulation nicht ändern —, werden die Präparationszeiten ebenfalls nicht zu der Simulationszeit hinzugerechnet.
Nun sind die CCC(d) zur Simulation von M vorbereitet. Wir beschreiben im folgenden den Simulationsalgorithmus des Prozessors i.

```
for j := 0 to c do
    if P_i will nach P_{π_j(i)} die Information x_i schicken
        then baue das Paket (i, π_j(i), x_i);
    route die gerade gepackten Pakete bzgl. der Permutation π_j
done;
{Jeder Q_i kennt nun alle Informationen, um den nächsten Schritt von P_i zu simulieren.}
simuliere einen Schritt von P_i
```

Die Laufzeit ist $c \cdot O(\log n) = O(\log n)$, also können wir einen Schritt von M durch $O(\log n)$ Schritte von CCC(d) simulieren.

Für die Leseschritte initialisieren wir die CCC(d) zusätzlich für die Umkehrpermutationen $\pi_0^{-1}, \ldots, \pi_c^{-1}$ und schicken „leere“ Pakete zu den Prozessoren, in denen gelesen werden soll.

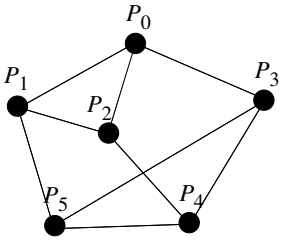
In diese Pakete werden die Inhalte der Lesefenster gelegt. Mittels der Umkehrpermutationen werden die Pakete dann an ihr Ziel geschickt. Auch hier ist die Laufzeit $O(\log n)$.

b) Die Aussage für das Shuffle-Exchange-Netzwerk wird analog bewiesen, da SE(d) nach Korollar 4.21 in Zeit $O(\log n)$ Permutationen zu routen vermag. □

Diese Aussagen sind mithin nichts anderes als Folgerungen aus dem folgenden allgemeineren Satz.

5.11 Satz:
Sei M_0 ein realistisches Netzwerk aus n Prozessoren, das beliebige Permutationen über $[n]$ (nach einer Preprocessing-Phase) in Zeit $t(n)$ routen kann. Dann ist M_0 $(n, t(n))$-universell.

5.12 Beispiel:
Gegeben sei der in Abbildung 5.2 dargestellte Graph M mit $n = 6$ und $c = 3$. Dieser Graph ist sogar 3-kantenfärbbar. Die Tabelle zeigt drei für die Schreibphase zu realisierende Permutationen.



	0	1	2	3	4	5
π_0	1	0	4	5	2	3
π_1	2	5	0	4	3	1
π_2	3	2	1	0	5	4

Abbildung 5.2: Ein Graph und seine Überdeckung mit drei Permutationen.

Hier schließt sich sofort die Frage an, ob es nicht vielleicht möglich ist, die Simulation zu beschleunigen. Wir werden nun ein universelles Netzwerk angeben, das nur konstanten Zeitverlust bei den Simulationen aufweist; dafür hat es allerdings $O(n^{1+\varepsilon} \log n)$ Prozessoren für beliebig kleine $\varepsilon > 0$ (dabei ist $1/\varepsilon$ dann ein Faktor im konstanten Zeitverlust). Der zugrundeliegende Mechanismus, der die Kommunikation gewährleist (im letzten universellen Netzwerk war dies der Permutationsmechanismus), ist in diesem Fall jedoch erheblich komplizierter.

Wie kann man denn „schneller“ simulieren? Im gerade angegebenen Simulationsalgorithmus dauerte es lange (nämlich $\Theta(\log n)$ Schritte), bis eine Information, die im simulierten Netzwerk einen Zeitschritt unterwegs ist, an dem Ziel-Prozessor angekommen ist. Um diese Verschickerei zu umgehen, tragen wir dafür Sorge, daß sich Duplikate der Prozessoren, die die Informationen verschicken wollen, ganz in der Nähe des Prozessors befinden, der die Information haben will.

Dazu betrachten wir zuerst für eine feste Zahl t und einen festen Prozessor P_i des zu simulierenden realistischen Netzwerks M (mit n Prozessoren und Grad c) das Netzwerk $T(i)$, das entsteht, wenn wir den Graphen von M, beginnend bei P_i, t-mal abwickeln. Kennen in $T(i)$ jeweils die „richtigen“ Prozessoren die Konfiguration K_j eines Prozessors P_j von M, so kennt nach $O(t)$ Schritten ein ausgezeichneter Prozessor von $T(i)$ die t-te Nachfolgekonfiguration K' für P_i.

Im folgenden Beispiel führen wir eine solche „Abwicklung“ einmal durch.

5.13 Beispiel:
Ein Graph M mit $n = 6$ und $c = 3$ ist noch einmal in Abbildung 5.3 dargestellt sowie die Abwicklung $T(0)$ dazu, wobei $t = 3$ ist.

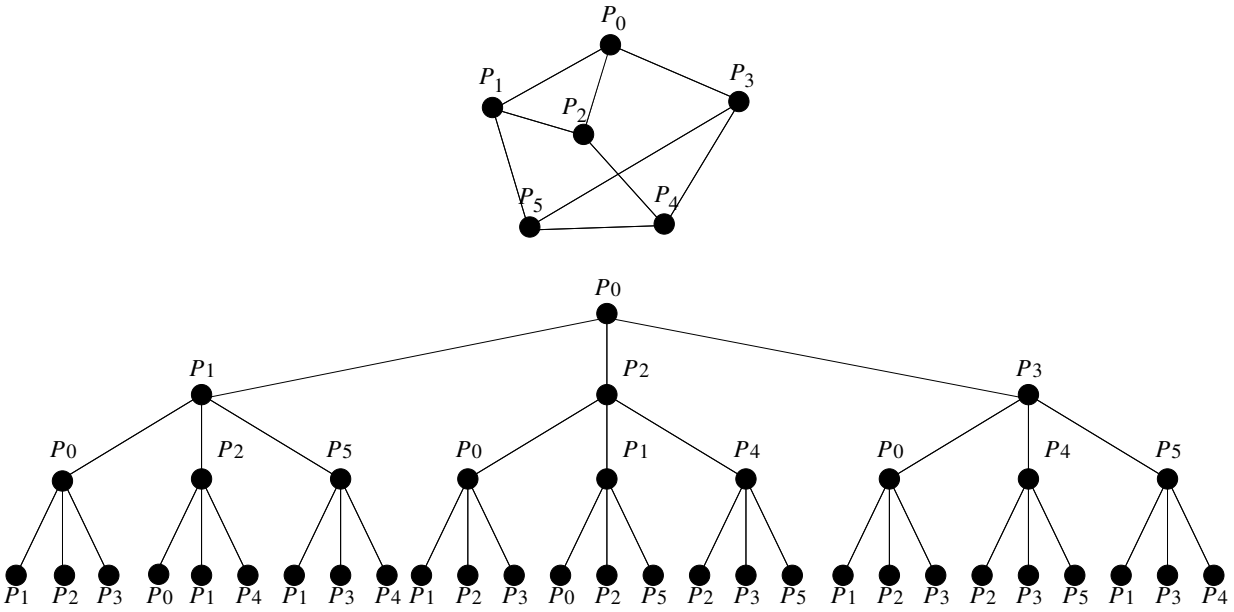


Abbildung 5.3: Ein Beispielgraph und die Abwicklung $T(0)$ mit $t = 3$.

Wir stellen fest:

- $T(0)$ ist ein Baum mit Verzweigungsgrad 3 und Tiefe $t = 3$. (Diese Baumstruktur hängt nicht vom zu simulierenden Netzwerk M ab.)
- Falls die Prozessoren von $T(0)$ wie oben initialisiert sind (ein Prozessor mit Beschriftung P_i kennt die Konfiguration K_i von P_i), kann $T(0)$ genau 3 Schritte von M so simulieren, daß anschließend die Wurzel von $T(0)$ die dritte Nachfolgekonfiguration von P_0 enthält.

Wir verallgemeinern dieses Konzept:

5.14 Definition:
Sei T das Netzwerk vom Grad $c + 1$, dessen Kommunikationsgraph ein vollständiger Baum mit Wurzel Q, Verzweigungsgrad c und Tiefe t ist (t entspricht der Anzahl von Kanten auf dem längsten Weg von der Wurzel zu den Blättern).
T ist für Prozessor P_i von M bei Konfiguration $\mathcal{K} = (K_0, \ldots, K_{n-1})$ **initialisiert** (dann schreiben wir $T(i)$), falls gilt:
Die Wurzel Q kennt K_i von P_i, und falls der Knoten Q' von T die Konfiguration K_j kennt, und P_j in M die Nachbarn $P_{j_1}, \ldots, P_{j_{c'}}$ mit $c' \leq c$ hat, so kennen die ersten c' Nachfolger von Q' in $T(i)$ die Konfigurationen $K_{j_1}, \ldots, K_{j_{c'}}$ von $P_{j_1}, \ldots, P_{j_{c'}}$. Die eventuell übrigen Nachfolger von Q' bleiben unbenutzt.

5.15 Lemma:
Die Wurzel von $T(i)$ kann in $O(t)$ Schritten die t-te Nachfolgekonfiguration von K_i für P_i berechnen. T hat Grad $c + 1$ und $\frac{c^{t+1}-1}{c-1}$ Prozessoren.

Beweis:

Der Grad ist klar, und die Anzahl der Prozessoren ist $\sum_{i=0}^{t} c^i = \frac{c^{t+1}-1}{c-1} = \Theta(c^t)$.

Zur Berechnung:

Den ersten Schritt kann jeder Knoten, der kein Blatt ist, ausführen. Falls Q' die Konfiguration K_j kennt, kann er in konstanter Zeit eine interne Operation von P_j ausführen, aber auch eine Kommunikation, da alle Konfigurationen K_ℓ von Nachbarn P_ℓ von P_j in Nachfolgern von Q' gespeichert sind. D. h. nur die Blätter von T können im ersten Schritt nicht die jeweilige Nachfolgekonfiguration berechnen.

Es bleibt noch ein Baum der Tiefe $t-1$ übrig, in dem jeder Prozessor, der vorher K_j kannte, nun die Nachfolgekonfiguration K'_j für P_j kennt. Induktiv können $t-1$ weitere Schritte ausgeführt werden, so daß anschließend Q die t-te Nachfolgekonfiguration von K für P_i kennt. □

5.16 Korollar:
Sei $T^* = (T(0), \ldots, T(n-1))$ ein Wald, welcher aus n Exemplaren von vollständigen c-ären Bäumen der Tiefe t und mit Wurzeln $Q_0, \ldots, Q_{n-1}$ besteht.

Falls jeder Baum $T(i)$ für P_i bei Konfiguration $\mathcal{K} = (K_0, \ldots, K_{n-1})$ initialisiert ist, kann T^* dann t Schritte von M in Zeit $O(t)$ (gestartet mit $\mathcal{K}$) **schwach simulieren**, *d. h. nach Ende der Simulation kennt jede Wurzel Q_i von $T(i)$ die t-te Nachfolgekonfiguration von K'_i von $\mathcal{K}$ für P_i.*

Beweis:

Zum Beweis wende Lemma 5.15 auf jedes T_i an. □

Damit haben wir jetzt eine schwache Simulation beschrieben. Kein simulierender Prozessor der Bäume kann mehr als einen Schritt direkt simulieren, da alle bekannten Konfigurationen bis auf die der Wurzeln veraltet sind. Um also die Simulation fortsetzen zu können, müssen wir dafür sorgen, daß die Pakete, die die Wurzeln während der Simulation von ihren Nachbarn bekommen haben — diese entsprechen der Kommunikation in M und sind insgesamt t viele —, zu den anderen Prozessoren im Inneren der Bäume geschickt werden.

Um jetzt t Schritte zu simulieren, werden wir den folgenden, oben bereits angedeuteten **Makroalgorithmus** realisieren:

M habe Startkonfiguration $\mathcal{K}$.
Solange M nicht hält, tue auf T^* folgendes:
- a) Initialisiere jeden Baum $T(i)$ für K_i von $\mathcal{K}$.
- b) Simuliere schwach t Schritte von M in T^*;
  {damit kennen die Wurzeln Q_i der Bäume $T(i)$ die t-te Nachfolgekonfiguration $\mathcal{K}'$ von $\mathcal{K}$.}
- c) Q_j schickt den Prozessoren, die den Prozessor P_j simulieren, die Zahlenfolge, die P_j in den letzten t Schritten gelesen hat

done

Der Schritt c) dieses Algorithmus ist ein verallgemeinertes Routing-Problem, wovon man sich leicht überzeugen kann, das unter dem Namen **Distribution** oder **Generalization** bekannt ist.

Dieses Problem lösen wir mit dem folgenden **Distributor**.

5.17 Definition:
Sei $a \leq b$. Ein (a, b)-**Distributor** $D_{a,b}$ ist ein realistisches Netzwerk mit Quellen $\mathcal{I} = \{I_0, \ldots, I_{a-1}\}$ und Senken $\mathcal{O} = \{O_0, \ldots, O_{b-1}\}$.
Jeder Prozessor O_i habe eine Zahl $c_i \in [a]$ gespeichert. Dann kann $D_{a,b}$ so präpariert werden, daß anschließend gilt:
Falls jedes I_j ein $\bar{x}_j \in \mathbb{N}^*$ als Eingabe bekommt, so kann $D_{a,b}$ die $(\bar{x}_0, \ldots, \bar{x}_{a-1})$ gemäß $(c_0, \ldots, c_{b-1})$ verteilen, d. h. $\bar{x}_j$ zu jedem Prozessor O_i transportieren mit $c_i = j$ für alle $j \in [a]$. Die Zeit für die Präparation heißt **Präparationszeit**, die Zeit für die Verteilung heißt **Verteilungszeit** oder auch **Distributionszeit**.

5.18 Beispiel:
Ein solches Distributionsproblem für $a = 4$ und $b = 7$ stellt Abbildung 5.4 dar.

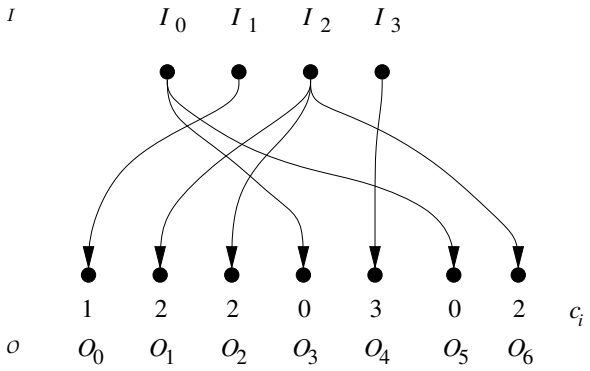


Abbildung 5.4: Ein Distributionsproblem.

Die Konstruktion eines $D_{a,b}$ baut auf dem Permutationsnetzwerk $(2, d)$-PN auf.

5.19 Definition:
Sei B_n das Netzwerk, welches aus einem $(2, d)$-PN mit $n = 2^d$ besteht und den zusätzlichen Kanten, die in Abbildung 5.5, das B_8 zeigt, dick eingezeichnet sind.

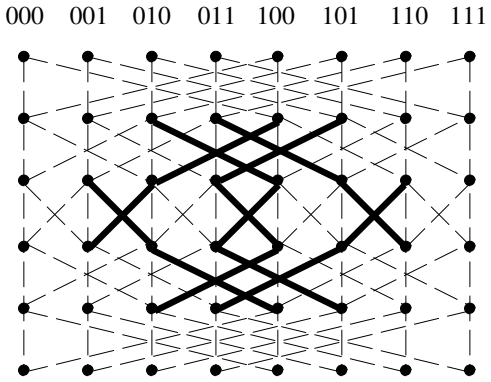


Abbildung 5.5: Der Distributor B_8.

Auf die Funktionsweise von B_n wollen wir hier nicht näher eingehen. Jedenfalls gilt die folgende Aussage:

5.20 Satz:
B_n ist ein (a,b)-Distributor für beliebige $a,b \leq n$ mit $2n\log n$ Prozessoren, Grad 6 und Distributionszeit $O(\log n + s)$ bei Botschaftenlänge s. Die Präparationszeit ist $O((\log n)^4)$.

Beweis:
Der Beweis dazu kann in[Mey86] nachgelesen werden. □

Wir können nach Korollar 5.16 den Teil b) des Makroalgorithmus in $O(t)$ Schritten ausführen. Das Problem besteht darin, die Teile a) und c) schnell zu machen, und dafür benutzen wir unseren Distributor.
Sei M_0 das Netzwerk, welches aus T^* und einem $(n, n\cdot c^t)$-Distributor besteht. Dessen I_i's sind mit den Q_i's, und dessen O_j's mit je einem Blattprozessor von T^* verbunden.
Eine schematische Darstellung von M_0 zeigt Abbildung 5.6:

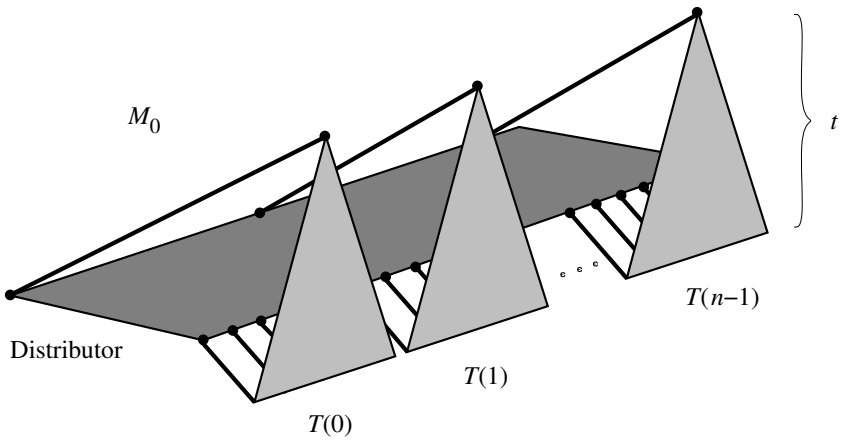


Abbildung 5.6: Das universelle Netzwerk M_0.

5.21 Satz:
Sei $\varepsilon > 0$ beliebig, $t = \lfloor \varepsilon \log_c n \rfloor$. Dann gilt: M_0 ist $(n,1)$-universell und hat Grad $\max\{6, c+1\}$ sowie $O(n^{1+\varepsilon}\log n)$ Prozessoren.

Beweis:
Wir haben schon in Korollar 5.16 gesehen, daß M_0 die t Schritte von M in $O(t) = O(\log n)$ Schritten schwach simulieren kann. Um eine solche Simulation stark zu machen, müssen wir anschließend T^* neu initialisieren. Dafür muß die „Information über die letzten t Schritte von P_i“ von Q_i zu jedem Blatt-Prozessor von T^* gebracht werden, der P_i simuliert. Dieses ist genau das, was der Distributor machen muß: Die O_j's, d. h. die Blatt-Knoten von T^*, die den Prozessor P_i simulieren, wollen diese Informationen von der Wurzel von T_i, also von I_i, erhalten.
Hierfür können wir M_0 gemäß Satz 5.20 präparieren.
Entlang der so eingestellten Wege müssen wir nun die „Informationen über die letzten t Schritte von P_i“ schicken. Wir könnten einfach eine ganze Beschreibung von K_i senden, aber die ist eventuell sehr lang ($\omega(\log n)$). Damit wäre die Verteilungszeit nicht von der gleichen Größenordnung wie die Zeit für die schwache Simulation. Also würde die Simulation von $\lfloor \varepsilon \log_c n \rfloor$

Schritten (schwache Simulation plus Neuinitialisierung) $S = \omega(\log n)$ Schritte benötigen, d. h. der Zeitverlust wäre $\frac{S}{\varepsilon \log_c n} = \omega(1)$, also nicht konstant.
Wir senden deshalb nur den String, der aus den Werten besteht, die P_i in den letzten t Schritten von Nachbarn gelesen hat. Diese Zahlenfolge hat nur Länge $t = O(\log n)$. Wenn ein Blattprozessor Q' den String erhält, kann Q', da er die Startkonfiguration K_i von P_i kennt, in $O(t)$ Schritten die letzten t Schritte von P_i nachholen, was darum auch alle Prozessoren im Inneren der Bäume können, denn nun bekommen sie von ihren Sohnprozessoren über den Zeitraum von t Schritten die benötigten Informationen.
Es werden nur Strings der Länge $t = O(\log n)$ verteilt, also benötigt das Verteilen gemäß Satz 5.20 nur $O(\log n)$ Schritte, d. h. $t = \lfloor \varepsilon \cdot \log_c n \rfloor$ Schritte werden in

$$O(\log n) \cdot (\text{Zeit für die schwache Simulation}) + O(\log n) \cdot (\text{Zeit für die Neuinitialisierung})$$

Schritten simuliert.
Damit ist der Zeitverlust insgesamt

$$O\Big(\frac{\log n + \varepsilon \cdot \log_c n}{\varepsilon \cdot \log_c n}\Big) = O(1 + \tfrac{1}{\varepsilon}) = O(1).$$

Zum Grad: Die Knoten im Inneren des Distributors haben Grad 6, die I_i's haben Grad $2 + 1$ vom Distributor und der Verbindung zu den T_j, die O_i's haben Grad $2 + 1$ vom Distributor und der Verbindung zu den T_j, alle Baumknoten bis auf die Blätter (die haben Grad 2) haben Grad $c + 1$, also auch die Wurzeln.
Die Größe von M_0 folgt durch unmittelbares Einsetzen in Lemma 5.15 und Satz 5.20. □

Spielt man ein bißchen mit der Tiefe t der Bäume, so kann man den obigen Beweis abändern, um den folgenden Satz zu formulieren.

5.22 Satz:
Für alle $k \leq \text{const} \cdot \log n$ gibt es ein (n, k)-universelles Netzwerk mit $n \cdot \ell$ Prozessoren, wobei ℓ die Gleichung $k \cdot \log \ell = O(\log n)$ erfüllt.

Beweis:
Übung □

5.23 Offene Frage:
Gilt folgende Aussage?
Ist M_0 ein beliebiges (n, k)-universelles Netzwerk mit $n \cdot \ell$ Prozessoren, so ist $k \cdot \log \ell = \Omega(\log n)$.

Weitere Aussagen zu unteren Schranken werden wir in Abschnitt 5.3 machen.

5.2.2 Simulation von Netzwerken mit beliebigem Grad

Im vorangehenden Abschnitt hatten wir uns noch darauf beschränkt, nur Netzwerke zu simulieren, die konstanten Grad haben. Wir werden jetzt alle unrealistischen Netzwerke durch ein realistisches Netzwerk simulieren.
Dazu wollen wir allerdings unser Modell ein wenig modifizieren. Jeder Prozessor P besitzt ein Lesefenster, aus dem **alle** Nachbarn von P **gleichzeitig** lesen können und auf das nur

P Schreibzugriff hat. In das Schreibfenster von P, das P ja nur lesen darf, darf zu einem Zeitpunkt weiterhin nur ein Nachbar von P etwas schreiben.

Große Auswirkungen hat eine solche Erweiterung für realistische Netzwerke nicht, da man so nur konstant viele Kommunikationen einsparen kann.

5.24 Definition:
Ein realistisches Netzwerk ist ein **(n, k)-Simulator**, wenn es alle unrealistischen Netzwerke aus n Prozessoren mit Zeitverlust $O(k)$ simulieren kann.

5.25 Satz:
Sei M_0 ein realistisches Netzwerk, das n Zahlen in Zeit $t(n)$ sortieren kann. Dann ist M_0 ein $(n, t(n))$-Simulator.

Beweis:
Wir wollen das Netzwerk M mit n Prozessoren $\{P_1, \ldots, P_n\}$ auf M_0 simulieren. Dazu simuliert Q_i, d. h. der Prozessor von M_0, der beim Sortieren die Zahl mit Rang i bekommt, den Prozessor P_i von M.

Wir zeigen hier nur eine vereinfachte Aussage. Wir nehmen an, daß in jedem Schritt jeder Prozessor bei einem Nachbarn schreibt.

Falls Prozessor P_i zu P_{j_i} einen Wert x_i schreiben will, baut er das Paar (j_i, x_i) auf. Die Gesamtheit all dieser Paare wird dann nach der ersten Komponente sortiert. Jetzt kennt der Prozessor P_{j_i} den Wert x_i.

Somit haben wir einen Schritt von M durch Sortieren in M_0 in $t(n)$ Schritten simuliert.

Ein etwas komplizierteres Vorgehen (mit mehr Sortierphasen) erlaubt es auch, einen parallelen Schritt zu simulieren, wenn nicht jeder Prozessor etwas bei einem Nachbarn schreiben will.

Die Lesephase kann ebenfalls mit diesem Sortiertrick und dem Verschicken von zu Beginn leeren Paketen (wie in Satz 5.10) simuliert werden. Hierzu benötigt man aber noch eine zusätzliche Distributionsphase, mit der die mehrfachen gleichzeitigen Lesewünsche befriedigt werden können. □

5.26 Korollar:
a) *Sei $d = 2^k$, $n = d \cdot 2^d$. CCC(d) ist ein $(n, (\log n)^2)$-Simulator.*

b) *Sei $n = 2^d$. SE(d) ist ein $(n, (\log n)^2)$-Simulator.*

Beweis:
Nach den Korollaren 4.14 und 4.21 sind CCC(d) und SE(d) Netzwerke, die n Zahlen in Zeit $O((\log n)^2)$ sortieren können. □

5.27 Bemerkung:
a) *Es gibt sogar ein realistisches Netzwerk mit $O(n)$ Prozessoren, das in $O(\log n)$ Schritten sortieren kann (Kombination des bereits erwähnten AKS-Sortiernetzwerkes[AKS83] mit einem trickreichen, Pipelining ausnutzenden Sortierverfahren von Leighton[Lei85]). Dieses Netzwerk ist also ein $(n, \log n)$-Simulator.*

b) *In den Kapiteln 6 und 6.4 werden wir deterministische und probabilistische Routingverfahren auf realistischen Netzwerken der Größe n kennenlernen, die damit deterministische und probabilistische $(n, \log n)$-Simulatoren ergeben.*

5.3 Untere Schranken für den Zeitverlust bei Simulationen

5.3.1 Untere Schranken für den Zeitverlust allgemeiner Simulationen

Nach Bemerkung 5.27 ergibt sich die Frage, ob wir, ähnlich wie beim Vorgehen in Abschnitt 5.2.1, wo wir zum Schluß sehr schnelle universelle Netzwerke konstruieren konnten, auch Simulatoren mit Zeitverlust $o(\log n)$ bauen können.

Wir werden zeigen, daß das nicht möglich ist, d. h. wir werden eine untere Schranke für den Zeitverlust von Simulationen angeben.

Dazu werden wir zuerst ein abstraktes Modell von Simulationen vorstellen.

Da wir die Rechenkraft der Prozessoren nicht eingeschränken wollen, ist der Teil der Simulation, der schwierig ist, die Realisierung der Kommunikation.

Wir wollen den unrealistischen, vollständigen Graphen M mit Prozessoren $P_1, \ldots, P_n$ simulieren.

Einen Rechenschritt von M beschreiben wir durch einen gerichteten Graphen F auf $P_1, \ldots, P_n$. (P_i, P_j) ist Kante in F, wenn P_i in diesem Schritt bei P_j lesen will. Da in einem Schritt ein Prozessor nur bei einem seiner Nachbarn lesen kann hat F Ausgangsgrad 1, aber beliebigen Eingangsgrad. Ein Beispiel für einen solchen Graphen zeigt Abbildung 5.7.

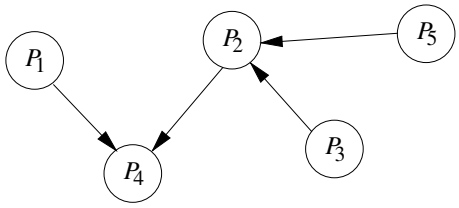


Abbildung 5.7: Der Graph F zu einem Rechenschritt.

Eine Berechnung von M ist dann eine Folge $F_1, F_2, \ldots$ von Rechenschritten. Eine Simulation von M sieht folgendermaßen aus:

In jedem Schritt t von M spezifizieren wir, welche Prozessoren von M_0 welche Prozessoren von M simulieren.

Das sind Vertretermengen $A_1^t, \ldots, A_n^t$ von M_0. (In A_i^t sind die Vertreter von P_i zur Zeit t. M_0 hat die Prozessormenge $\{Q_1, \ldots, Q_m\}$.) Es gilt für alle $i \neq j$:

$$\emptyset \neq A_i^t \subseteq \{Q_1, \ldots, Q_m\} \text{ und } A_i^t \cap A_j^t = \emptyset$$

Und es ist: $|A_1^0| = \ldots = |A_n^0| = 1$.

Die Menge der Prozessoren, die P_i simuliert (also A_i^t), kann sich ständig ändern.

5.28 Beispiel:
Als Beispiel für eine solche Rechnung können wir die Simulation in Satz 5.21 betrachten: Ein Prozessor P_i wird von der Wurzel Q_i von $T(i)$ und zu Beginn von vielen anderen Prozessoren in den Bäumen simuliert. Doch eine Zeit lang werden die Vertretermengen immer kleiner, bis nur noch die Wurzeln die aktuelle Konfiguration kennen.

Falls nun der t-te Rechenschritt $F = \{(P_i, P_{j_i}) \mid i = 1, \ldots, n\}$ simuliert wird, so muß jeder Prozessor $Q \in A_i^{t+1}$ entlang von **t-Transportwegen** in M_0 mit einem $Q' \in A_i^t$ (eventuell ist $Q' = Q$) und einem $Q'' \in A_{j_i}^t$ verbunden sein.
Der Zeitverlust eines solchen t-ten Simulationsschrittes ist damit mindestens

$$k_t = \max\{\text{Länge der } t\text{-Transportwege}\}.$$

Der Zeitverlust der Simulation von $F_1, \ldots, F_\ell$ ist $k = \frac{1}{\ell} \sum_{t=1}^{\ell} k_t$.
Dieses Simulationsmodell ist sehr allgemein, umfaßt allerdings keine **asynchronen** Simulationen, wie wir sie z. B. bei der Simulation von Ascend/Descend-Programmen auf dem Cube-Connected Cycles-Netzwerk in Abschnitt 4.3 kennengelernt hatten (vgl. dazu besonders Beispiel 4.13).
Wir zeigen nun:

5.29 Satz:
Sei M_0 ein (n, k)-Simulator. Dann ist $k = \Omega(\log n)$

Beachte, daß diese Aussage nicht von der Größe von M_0 abhängt. M_0 kann also beliebig groß sein, bei der Simulation des vollständigen Graphen über n Knoten wird der Zeitverlust auf jeden Fall $\Omega(\log n)$ sein.
Beweis:
Sei M_0 nun ein (n, k)-Simulator der Größe m mit Grad c. Wir werden für M_0 Schritt für Schritt eine Berechnung $F_1, F_2, \ldots$ von M beschreiben, für deren Simulation M_0 lange braucht.
Oder anders gesagt: Gegeben ist ein Simulator M_0 (mit den obigen Parametern), der **jeden** Algorithmus (und damit auch jedes Kommunikationsmuster) für das vollständige Netzwerk aus n Prozessoren der Laufzeit $t(n)$ in $O(k \cdot t(n))$ Schritten simulieren kann. D. h., daß natürlich auch der Simulationsalgorithmus (sozusagen das Betriebssystem für M_0) bereits fest vorgegeben ist („Beliebig aber fest“). Wir geben nun ein „Programm“ (d. h. Kommunikationsmuster) für das vollständige Netzwerk an, dessen Simulation M_0 viel Zeit kostet.
Die Beweisidee sieht folgendermaßen aus:
Es gibt immer einen Rechenschritt, bei dessen Simulation

(i) entweder viel Zeit ($\gamma \log n$ Schritte) verbraucht wird,

(ii) oder viele Prozessoren „verbraucht“ werden, d. h. eine Vertretermenge stark schrumpft.

Der zweite Fall kann nicht „zu oft“ hintereinander (gemessen an der Gesamtzahl der simulierten Schritte) eintreten, da die Zahl der Vertreter nicht kleiner als n werden kann. Auch hier hilft uns wieder zur Anschauung die Simulation aus Satz 5.21: Kann ein Schritt „schnell“ simuliert werden, so schrumpfen die Vertretermengen, oder ein Schritt dauert lange, weil der Distributor benutzt wird.

5.30 Lemma:
Für jedes $\varepsilon \in (0, \frac{1}{2})$ gibt es ein $\gamma > 0$, so daß für jede Simulationssituation mit Vertretermengen $A_1, \ldots, A_n$ gilt: Es gibt einen Rechenschritt F, so daß für die Vertretermengen $A'_1, \ldots, A'_n$ nach Ausführung von F gilt:

(i) Entweder war der Zeitaufwand für die Simulation von F mindestens $\gamma \log n$,

(ii) oder es gilt: $\sum_{i=1}^{n} |A_i'| \leq \frac{1}{n^\varepsilon} \sum_{i=1}^{n} |A_i|$

Beweis:
Sei $i \in \{1, \ldots, n\}$ fest. Für $i \neq j$ sei F^j der einzelne Rechenschritt $\{(i,j)\}$. $\varepsilon \in (0, \frac{1}{2})$ sei fest und $A_1, \ldots, A_n$ seien wie oben beschrieben.

Behauptung: Es gibt ein $\gamma > 0$, so daß gilt: falls es für jedes $j \neq i$ möglich ist, daß M_0 den Rechenschritt F^j in weniger als $\gamma \log n$ Schritten simuliert, dann gilt folgendes:
Es gibt ein $j' \neq i$ derart, daß, falls M_0 den Rechenschritt $F^{j'}$ tatsächlich in weniger als $\gamma \log n$ Schritten simuliert, dann anschließend $|A_i'| \leq |A_i|/n^\varepsilon$ ist.

Behauptung ⇒ Lemma:
Falls es einen Rechenschritt $\{(i,j)\}$ mit $i \neq j$ gibt, so daß zur Simulation dieses Rechenschrittes mindestens $\gamma \log n$ Schritte notwendig sind, so setzen wir $F := F^j$, und (i) gilt (mindestens $\gamma \log n$ Schritte zur Simulation heißt: es gibt einen Prozessor $Q \in A_i^{t+1}$, so daß jeder Weg von Q zu jedem $Q' \in A_j^t$ oder jedem $Q'' \in A_i^t$ eine Länge von mindestens $\gamma \log n$ hat).
Falls ein solcher Schritt $\{(i,j)\}$ nicht existiert, ist für jedes i die Behauptung erfüllt, d. h. zu jedem i gibt es ein $j'(= j_i)$, so daß die Aussage der Behauptung gilt.
Sei $F = \{(i, j_i) \mid i = 1, \ldots, n\}$. Falls der Zeitverlust kleiner als $\gamma \log n$ ist, gilt nach der Behauptung für jedes i: $|A_i'| \leq |A_i|/n^\varepsilon$, womit (ii) gilt. □ (Lemma 5.30)

Beweis der Behauptung:
$\gamma > 0$ sei hier fest gewählt, wird später noch festgelegt.
Angenommen, jedes F^j kann in höchstens $k < \gamma \log n$ Schritten von M_0 simuliert werden.
Sei $A_i^{(j)}$ die Vertretermenge von P_i nach der Simulation von F^j (in höchstens k Schritten).
Fr die folgenden Beziehungen siehe Abbildung 5.8. Seien $C_j \subseteq A_j$ die Prozessoren, von denen dann t-Transportwege der Länge höchstens k nach $A_i^{(j)}$ gehen. Da auch von A_i zu jedem Knoten aus $A_i^{(j)}$ t-Transportwege der Länge höchstens k gehen, ist $C_j \subseteq \Gamma_{2k}(A_i)$ Da alle C_j disjunkt sind (wegen $C_j \subseteq A_j$), und da ein Prozessor von M_0 nur für die Simulation von höchstens einem Prozessor von M zuständig ist, ist

$$\left| \bigcup_{j \neq i} C_j \right| = \sum_{j \neq i} |C_j| \leq |\Gamma_{2k}(A_i)| \leq |A_i| \cdot c^{2k} \ .$$

Darum gibt es mindestens ein $j' \neq i$ derart, daß

$$|C_{j'}| \leq \frac{|A_i| \cdot c^{2k}}{n-1} \leq \frac{|A_i| \cdot c^{3k}}{n}$$

ist (Durchschnittsargument). Also: Falls $F^{j'}$ ausgeführt wird in höchstens k Schritten, so ist $A_i^{(j')} \subseteq \Gamma_k(C_{j'})$, und damit

$$|A_i^{(j')}| \leq c^k \cdot |C_{j'}| \leq c^k \cdot \frac{|A_i| \cdot c^{3k}}{n} = \tfrac{1}{n} \cdot |A_i| \cdot c^{4k}$$

Wähle k so, daß $\frac{1}{n} c^{4k} = \frac{1}{n^\varepsilon}$ ist, also

$$k = \tfrac{1}{4} \log_c(n^{1-\varepsilon}) = \underbrace{\frac{1-\varepsilon}{4 \log c}}_{\gamma'} \cdot \log n.$$

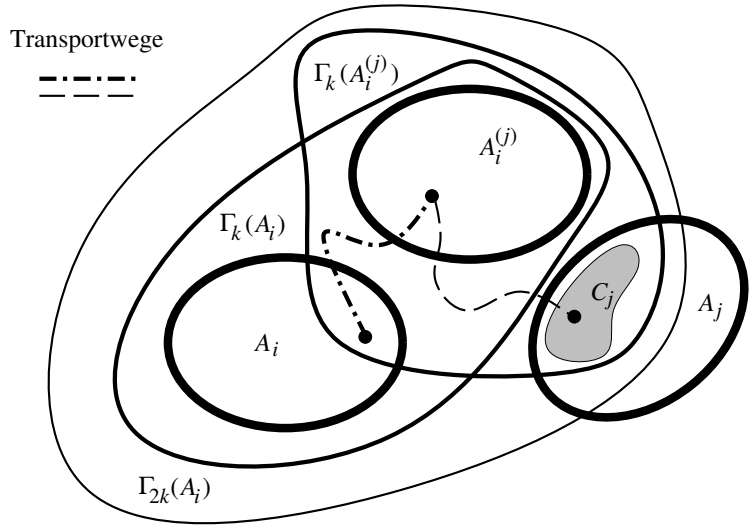


Abbildung 5.8:

Wählen wir γ etwas größer als γ', so haben wir $|A_i^{(j')}| \leq \dfrac{|A_i|}{n^\varepsilon}$ □ (Behauptung)

Beweis des Satzes:
Die Rechnung $F_1, F_2, \ldots, F_\ell$ sei Schritt für Schritt durch das Lemma definiert.
$h_t = \sum_{i=1}^{n} |A_i^t|$ sei die Zahl der Vertreter zur Zeit t. $k_1, \ldots, k_\ell$ seien die Zeitverluste der Simulation dieser Schritte.
Sei S die Menge der Indizes t mit $k_t < \gamma \log n$, $s = |S|$. Dann gilt:

$$\sum_{t=1}^{\ell} k_t \geq \sum_{t \notin S} k_t \geq (\ell - s)\gamma \log n \tag{5.1}$$

Wir müssen zeigen, daß s nicht zu groß ist.
Wir wissen, daß $h_0 = n$ ist (siehe S. 50), und daß zu jedem Zeitpunkt t gilt: $h_t \geq n$. Weiterhin wird während der Simulation s mal die Zahl der Vertreter um mindestens einen Faktor $1/n^\varepsilon$ verringert, nämlich in jedem Schritt $t \in S$. In jedem Schritt $t \notin S$ wird die Zahl der Vertreter um höchstens einen Faktor $c + c \cdot (c-1) + c \cdot (c-1)^2 + \ldots + c \cdot (c-1)^{k_t - 1} \leq c^{k_t+1}$ erhöht.
Damit ist

$$n \leq h_\ell \leq h_0 \cdot \underbrace{\left(\frac{1}{n^\varepsilon}\right)^s}_{\text{Verringerung}} \cdot \underbrace{\prod_{t \notin S} c^{k_t+1}}_{\text{Erhöhung}} \leq \frac{n}{n^{\varepsilon s}} \cdot c^{\ell + \sum_{t=1}^{\ell} k_t}.$$

Falls $\sum_{t=1}^{\ell} k_t > \ell \cdot (\frac{1}{2}\varepsilon \log_c n - 1)$, sind wir fertig, sonst gilt:

$$\begin{aligned} n &\leq \frac{n}{n^{\varepsilon s}} \cdot c^{\ell + \ell \cdot (\frac{1}{2}\varepsilon \log_c n - 1)} = n^{1 - \varepsilon s + \ell \frac{\varepsilon}{2}} \\ &\Rightarrow \quad n^{\varepsilon s} \leq n^{\frac{\ell}{2}\varepsilon} \\ &\Rightarrow \quad s \leq \tfrac{1}{2}\ell \end{aligned}$$

Also gilt mit Ungleichung (5.1):

$$\sum_{t=1}^{\ell} k_t \geq (\ell - s)\gamma \log n \geq \tfrac{1}{2}\ell\gamma \log n = \Omega(\ell \log n)$$

und damit $k = \Omega(\log n)$. □ (Satz 5.29)

Auch für universelle Netzwerke lassen sich untere Schranken zeigen ([Mey83a, MSW97]):

5.31 Satz:
Sei M_0 ein (n,k)-universelles Netzwerk der Größe m. Dann ist $m \cdot k = \Omega(n \cdot \log n)$. Insbesondere heißt das für konstanten Zeitverlust, also $k = O(1)$: $m = \Omega(n \cdot \log n)$

Mit der in Abschnitt 5.2.1 vorgestellten Konstruktion haben wir dann für die Größe m von universellen Netzwerken:

$$\Omega(n \cdot \log n) = m = O(n^{1+\varepsilon})$$

Diese Schranken zu verschärfen, ist eine interessante Aufgabe für die Forschung.

Man beachte in der obigen Darstellung, daß wir nur $O(n^{1+\varepsilon})$ anstatt $O(n^{1+\varepsilon'} \log n)$ geschrieben haben, da für alle $\varepsilon > \varepsilon'$ sogar gilt: $n^{1+\varepsilon'} \log n = o(n^{1+\varepsilon})$.

5.4 Abschließende Bemerkungen

Es ist nicht notwendig, in das Permutationsnetzwerk die zusätzlichen Kanten einzufügen, um das Distributieren auf $(2,d)$-PN mit derselben Initialisierungs- und Distributionszeit realisieren zu können([NS80] oder[Lei92]).

In[KLM+97] wird ein noch allgemeineres Simulationsmodell eingeführt, das die asynchronen Simulationen einschließt und zuläßt, daß die Größe m des (n,k)-universellen Netzwerks kleiner als n ist. Der Term n/m heißt dann **Last** oder auch **Load**. Die untere Schranke aus Satz 5.31 gilt auch für dieses zur Zeit allgemeinste Simulationsmodell.

5.5 Literatur zu Kapitel 5

[AKS83] M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in $c \cdot \log n$ parallel steps. *Combinatorica*, 3:1–19, 1983.

[BCH+96] S. N. Bhatt, F. R. K. Chung, J.-W. Hong, F. T. Leighton, B. Obrenić, A. L. Rosenberg, and E. J. Schwabe. Optimal emulations by Butterfly-like networks. *Journal of the ACM*, 43:293–330, 1996.

[Bol98] B. Bollobás. *Modern Graph Theory*. Springer-Verlag, Heidelberg, 1998.

[GP83] Z. Galil and W. Paul. An efficient general-purpose parallel computer. *Journal of the ACM*, 30:360–387, 1983.

[KLM+97] R. R. Koch, F. T. Leighton, B. M. Maggs, S. B. Rao, A. L. Rosenberg, and E. J. Schwabe. Work-preserving emulations of fixed-connection networks. *Journal of the ACM*, 44:104–147, 1997.

[Lei85] T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, 34:344–354, 1985.

[Lei92] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[Mey83a] F. Meyer auf der Heide. Efficiency of universal parallel computers. *Acta Informatica*, 19:269–296, 1983.

[Mey83b] F. Meyer auf der Heide. Infinite Cube-Connected Cycles. *Information Processing Letters*, 16:1–2, 1983.

[Mey86] F. Meyer auf der Heide. Efficient simulations among several models of parallel computers. *SIAM Journal on Computing*, 15:106–119, 1986.

[MS90] B. Monien and H. Sudborough. Embedding one interconnection network in another. *Computing*, 7:257–282, 1990.

[MSW97] F. Meyer auf der Heide, M. Storch, and R. Wanka. Optimal tradeoffs between size and slowdown for universal parallel networks. *Theory of Computing Systems*, 30:627–644, 1997.

[MW89] F. Meyer auf der Heide and R. Wanka. Time-optimal simulations of networks by universal parallel computers. In *Proc. 6th Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 120–131, 1989.

[NS80] D. Nassimi and S. Sahni. Data broadcasting in SIMD computers. *IEEE Transactions on Computers*, 30:101–106, 1980.

[Sch80] J. T. Schwartz. Ultracomputers. *ACM Transactions on Programming Languages and Systems*, 2:484–521, 1980.

[Wan88] R. Wanka. *Über die Effizienz von Simulationen eingeschränkter Netzwerkklassen auf universellen parallelen Rechnern*. Diplomarbeit, Universität Dortmund, 1988.

Kapitel 6

Oblivious Routing

In Kapitel 2 haben wir off-line Routing, d. h. Routing mit Preprocessing, kennengelernt. Viele Anwendungen erlauben jedoch kein Preprocessing, d. h. es werden on-line-Verfahren gefordert. Im vorliegenden Kapitel werden wir uns mit besonders einfachen derartigen Verfahren beschäftigen, nämlich solchen, in denen der Routingweg eines Pakets von Knoten a nach Knoten b a priori festliegt, also nicht, wie in Kapitel 2, abhängig vom jeweiligen Routingproblem geschickt gewählt wird. Solche Verfahren nennt man **oblivious** (der Begriff wird in Definition 6.4 formalisiert). Die meisten genutzten Routingverfahren (z. B. in Parallelrechnern und im Internet) sind oblivious.

6.1 Definition: (Routingprobleme)
Seien $N, p \in \mathbb{N}$, M ein Netzwerk mit Prozessormenge $[N]$. Sei $F_p := \{f \mid f : [N] \times [p] \to [N]\}$. Jeder Prozessor $i \in [N]$ habe p Pakete mit jeweils Nummern von 0 bis $p-1$. Die **Ident-Nummer** des j-ten Paketes in Prozessor i sei $i \cdot p + j$. Wir schreiben dafür $D_{i \cdot p + j}$.
Unter dem **Routing der Funktion $f \in F_p$** verstehen wir die Aufgabe, das j-te Paket in Prozessor i von i zum Prozessor $f(i, j)$ zu schicken.
Gilt $|f^{-1}(k)| = p$ für alle $k \in [N]$, so sprechen wir auch von einem ***p*-*p*-Routing-Problem** und nennen f eine ***p*-*p*-Funktion**.
Für $p = 1$ und bijektive Funktionen $f \in F_1$ haben wir das in Kapitel 2 betrachtete Permutationsrouting.
Falls M das Butterfly-Netzwerk BF($\log N$) ist, besteht das Problem, eine Funktion $f \in F_p$ zu routen, darin, das j-te Paket von **Quelle** i zur **Senke** $f(i, j)$ zu schicken.

6.2 Beispiel:
Das Permutationsrouting können wir mit Hilfe eines Sortieralgorithmus einfach durchführen, indem wir die Pakete $(i, \pi(i), x_i)$ nach der zweiten Komponente sortieren. Dies ist ein Verfahren ohne Preprocessing, das z. B. auf dem Cube-Connected Cycles-Netzwerk und dem Shuffle-Exchange-Netzwerk mit n Prozessoren eine Laufzeit von $O((\log n)^2)$ hat. Gegenüber dem Durchmesser von $O(\log n)$ ist dies ein Verlust eines Faktors von $O(\log n)$. Dieses Verfahren ist nicht oblivious.

6.3 Beispiel:
Betrachten wir den folgenden Algorithmus für das Routing auf $M(n, 2)$, dem 2-dimensionalen Gitter der Kantenlänge n, für eine Permutation $\pi : [n]^2 \to [n]^2$. Sei dazu $\pi(i, j) = (i', j')$.
Sende Paket $x_{i,j}$ nach Prozessor (i', j), dann nach Prozessor (i', j'). Wenn ein Puffer dabei

mehr als ein Paket enthält, so verschicke dasjenige, das am weitesten von seinem Ziel entfernt ist.
Wenn $N = n^2$ die Anzahl der Prozessoren ist, so ist die Laufzeit dieses Algorithmus $O(n) = O(\sqrt{N})$. Denn während des ersten Teils werden genau n Pakete in jede der n Prozessorzeilen mit dem bereits aus Satz 2.7 bekannten Algorithmus der Laufzeit $O(n)$ auf $M(n, 1)$ geroutet; um diese Pakete in den Zeilen ans Ziel zu bringen, benötigen wir dann auch wieder $O(n)$ Schritte, da in jeder Spalte wieder nur n Pakete sind. Da diese Laufzeit von der gleichen Ordnung wie der Durchmesser ist, ist dieser Algorithmus für das 2-dimensionale Gitter optimal.

Wenn wir uns die Wege der Pakete in Beispiel 6.3 anschauen, so stellen wir fest, daß der Weg, den das Paket $x_{i,j}$ nimmt, nur von den beiden Paaren (i, j) und (i', j') abhängt, aber nicht von der Permutation π, d. h. von den Wegen und Zielen anderer Pakete. Wir können also direkt aus den Zahlen i, j, i', j' den Weg des Paketes $x_{i,j}$ ausrechnen.
Vergleichen wir dies z. B. mit dem Preprocessing der Permutationsnetzwerke, so sehen wir, daß dort der Weg eines Paketes in Abhängigkeit von den Wegen der anderen Pakete gewählt wurde.
Auch für Beispiel 6.2 können wir die Wege nicht aus einem einzelnen Paar aus Start- und Zieladresse ausrechnen, da in einem zum Routen abgewandelten Sortieralgorithmus der Weg eines Paketes vom Ausgang der Vergleiche, also den anderen Start-/Zieladressen abhängt.
Diese Überlegungen führen zu folgender Definition:

6.4 Definition:
Sei M ein Netzwerk mit Prozessormenge $[n]$. Für $i, k \in [n]$ sei $w(i, k)$ ein Weg von i nach k in M. Eine Menge $\mathcal{W} = \{w(i, k) \mid i, k \in [n]\}$ heißt **Wegesystem** in M. $w(i, k)$ beschreibt einen Weg von i nach k in M. Ein Routing-Protokoll für M heißt **oblivious** (= vergeßlich, „etwas außer acht lassend"), falls es ein Wegesystem $\mathcal{W}$ in M gibt, so daß das Protokoll ein Paket von i nach k immer entlang dem Weg $w(i, k)$ schickt.
Sei $f : [n] \times [p] \to [n]$ eine Routingfunktion, d. h. jeder Prozesor i hat p Pakete $(i, 0), \ldots, (i, p-1)$. (i, j) muß zu Prozessor $f(i, j)$ geschickt werden.
Die **Congestion** (= Stau) $C_f(i)$ von f in Prozessor i ist die Anzahl der Wege $w(i', f(i', \cdot))$, $i' \in [n]$, die durch Prozessor i führen. $C_f := \max\{C_f(i) \mid i \in [n]\}$ ist die Congestion von f.
Die Länge D_f eines längsten Weges in $\{w(i, f(i, \cdot)) \mid i \in [n]\}$ ist die **Dilation** (= Kantenstreckung) von f.

6.5 Beispiel:
Wollen wir in BF($\log N$) Pakete von der Quelle $(0, \bar{a})$ zur Senke $(\log N, \bar{b})$ auf dem kürzesten Weg versenden, so muß dies über den (eindeutigen) Weg

$$(0, a_{\log N-1} \ldots a_2 a_1 a_0) \to (1, a_{\log N-1} \ldots a_2 a_1 b_0) \to$$
$$(2, a_{\log N-1} \ldots a_2 b_1 b_0) \to \cdots \to (\log N, b_{\log N-1} \ldots b_2 b_1 b_0)$$

geschehen. Damit ist ein Wegesystem beschrieben, das die Quellen mit den Senken verbindet.

6.6 Bemerkung:
a) *Jedes Oblivious Routing-Protokoll benötigt zum Routen von f mindestens* $\max\{C_f, D_f\}$ *Schritte.*

b) *Jedes Oblivious Routing-Protokoll, in dem jeder Prozessor in jedem Schritt, in dem sein Puffer nicht leer ist, ein Paket weiterleitet, (also jedes „vernünftige“ Protokoll) routet f in höchstens $C_f \cdot D_f$ Schritten.*

Als „Freiheit“ eines Prozessors in einem Oblivious Routing-Algorithmus zur Laufzeit bleibt nur die Auswahl desjenigen Paketes aus dem Puffer, das im aktuellen Schritt verschickt wird (das hatten wir in Beispiel 6.3 in der zweiten Phase ja gemacht).

6.1 Analyse der Congestion: Worst-Case-Untersuchungen

Gefühlsmäßig läßt sich vermuten, daß das Oblivious Routing nicht sehr effizient durchgeführt werden kann, da bei Unkenntnis der Routingfunktion „natürlich“ hohe Congestion auftritt. Tatsächlich gilt die folgende untere Schranke aus[BH85] (dort wird ein etwas anderes Modell untersucht, darum sehen die Ergebnisse dort wie auch in[KKT91] und[Lei92a] etwas anders aus).

6.7 Satz: (Borodin/Hopcroft)
In jedem Netzwerk M mit n Prozessoren und Grad c gibt es für jedes Oblivious Routingverfahren eine Permutation $\pi : [n] \to [n]$ mit $C_\pi \geq \sqrt{n/c}$.

Beweis:
Sei M ein beliebiges Netzwerk mit Prozessormenge $[n]$ und vom Grad c, und sei ein Algorithmus für das Oblivious Routing auf M gegeben. $\mathcal{W} = \{w(i,j) \mid i,j \in [n]\}$ sei das zugrundeliegende Wegesystem. Mit G_k (vgl. Abbildung 6.1) bezeichnen wir den Teilgraphen von M, der aus den Wegen $w(0,k), \ldots, w(n-1,k)$ besteht. Beachte dabei, daß Wege, die sich einmal getroffen haben, nicht unbedingt gemeinsam weiterlaufen müssen.

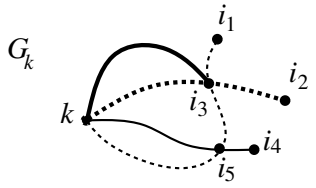


Abbildung 6.1: Der Teilgraph G_k.

Ein Knoten i heißt **Quelle** für Knoten j bzgl. k, falls $w(i,k)$ durch j geht. In Abbildung 6.1 ist z. B. Knoten i_2 eine Quelle für i_3 bzgl. k, da $w(i_2,k)$ durch i_3 verläuft.
Im folgenden werden wir eine Permutation π mit hoher Congestion konstruieren. Zuerst einmal zeigen wir, daß es „viele“ Knoten gibt, die bzgl. k „viele“ Quellen haben.
Sei $A(k,z) := \{j \in [n] \mid j$ hat bzgl. k mindestens z Quellen$\}$ die Menge aller Knoten, durch die mindestens z verschiedene Wege aus $\mathcal{W}$ nach k führen. Dann gilt die folgende Aussage:

6.8 Lemma:
Für alle $k \in [n]$ gilt: $|A(k,z)| \geq \dfrac{n}{c \cdot z}$

Beweis von Lemma 6.8:
Für beliebiges, aber festes k sei $L := \Big|\{w(i,k) \mid i \in [n],\ i \notin A(k,z)\}\Big|$, also die Zahl der Wege, die außerhalb von $A(k,z)$ starten, und B die Menge der Nachbarn von Knoten in $A(k,z)$, die nicht in $A(k,z)$ selbst liegen.

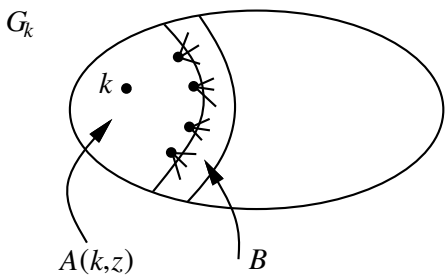


Abbildung 6.2:

Es ist $L = n - |A(k,z)|$. Wegen der Gradbeschränkung gilt weiter, daß $|B| \leq |A(k,z)| \cdot c$ ist. Da $B \cap A(k,z) = \emptyset$ ist, laufen über jeden Knoten aus B höchstens $z-1$ Wege, also gilt:

$$\begin{array}{rrcl} & |A(k,z)| \cdot c \cdot (z-1) \ \geq\ |B| \cdot (z-1) & \geq & L \\ \Rightarrow & |A(k,z)| \cdot c \cdot (z-1) & \geq & n - |A(k,z)| \\ \Rightarrow & |A(k,z)| \cdot [c \cdot (z-1) + 1] & \geq & n \\ \Rightarrow & |A(k,z)| & \geq & \dfrac{n}{c \cdot (z-1) + 1} \geq \dfrac{n}{c \cdot z} \end{array}$$

□ (Lemma 6.8)

Sei $X(z) := \{(j,k) \mid j,k \in [n],\ j \in A(k,z)\} = \bigcup_{k \in [n]} \Big(A(k,z) \times \{k\}\Big)$. Wenn wir die Menge $A(k,z)$ als Markierung der Knoten in G_k ansehen, durch die mindestens z verschiedene Wege laufen, so kann man $X(z)$ als Markierung aller Knoten j von M auffassen, und zwar eine Markierung mit all den Zieladressen, für die jeweils mindestens z Wege durch j gehen. Für die Gesamtzahl dieser Markierungen gilt dann:

$$|X(z)| = \sum_{k \in [n]} |A(k,z)| \overset{\text{Lemma 6.8}}{\geq} \frac{n}{c \cdot z} \cdot n = \frac{n^2}{c \cdot z}$$

Zum Knoten j sei $S_j := \{k \mid (j,k) \in X(z)\}$ die Menge all der Ziele, für die mindestens z Wege durch j gehen.
Da $\sum_{j \in [n]} |S_j| = |X(z)| \geq \dfrac{n^2}{c \cdot z}$ ist, es aber nur n Mengen S_j gibt, gibt es mindestens einen Knoten $\hat{\jmath}$ mit der Eigenschaft $|S_{\hat{\jmath}}| \geq \dfrac{n}{c \cdot z}$. Wähle z so, daß z und $|S_{\hat{\jmath}}|$ gleich groß sind. Dies ist der Fall, wenn $z = \dfrac{n}{c \cdot z}$ ist, also $z = \sqrt{n/c}$. Beachte, daß dann $|S_{\hat{\jmath}}| \geq \sqrt{n/c}$ ist. $\hat{\jmath}$ ist also ein Knoten, der für mindestens $\sqrt{n/c}$ Ziele jeweils mindestens $\sqrt{n/c}$ Quellen hat.
Für das Ziel $k \in S_{\hat{\jmath}}$ bezeichne $Y_k := \{i \mid w(i,k)$ läuft über $\hat{\jmath}\}$ die Menge aller Startadressen, für die die Wege nach k über den Knoten $\hat{\jmath}$ verlaufen. Da für jedes $k \in S_{\hat{\jmath}}$ mindestens z Wege über $\hat{\jmath}$ laufen, gilt für alle $k \in S_{\hat{\jmath}}$: $|Y_k| \geq z = \sqrt{n/c}$.

Wir können nun einen Algorithmus angeben, der die gesuchte Permutation π bestimmt. Wir brauchen jetzt nämlich nur noch für z viele (also nicht unbedingt für alle) Zieladressen $k \in S_{\hat{\jmath}}$ eine Startadresse $i_k \in Y_k$ (davon haben wir ja sehr „viele“) derart zu wählen, daß alle i_k verschieden sind. Alle z Wege $w(i_k, k)$ für diese Paare (i_k, k) laufen über $\hat{\jmath}$. Also benötigt das Oblivious Routingverfahren auch mindestens z Schritte.

Eingabe: $Y_{\ell_1}, \ldots, Y_{\ell_z}$ mit $|Y_{\ell_i}| \geq z$

1. *frei* $:= [n]$;
2. `for` $k := 1$ `to` z `do`
3. wähle $i_{\ell_k} \in Y_{\ell_k}$;
4. { also $\pi(i_{\ell_k}) := \ell_k$ }
5. *frei* $:=$ *frei* $\setminus \{\ell_k\}$;
6. `for` $k' := k+1$ `to` z `do`
7. $Y_{\ell_{k'}} := Y_{\ell_{k'}} \setminus \{i_k\}$
8. `done`
9. `done`
10. für alle $i \notin \{i_{\ell_1}, \ldots, i_{\ell_z}\}$:
11. wähle $k \in$ *frei* ;
12. $\pi(i) := k$;
13. *frei* $:=$ *frei* $\setminus \{k\}$;
14. `done`

Das Verfahren liefert eine Permutation, da nach k Durchläufen der äußeren Schleife (Zeile 2.) noch für jedes $Y_{\ell_{k'}}$ gilt: $|Y_{\ell_{k'}}| \geq z - k > 0$.

Beim Routing von π in M gilt nun, daß die Wege $w(i_k, k)$ für mindestens z viele verschiedene $k \in S_{\hat{\jmath}}$ über den Knoten $\hat{\jmath}$ verlaufen, also, da ja $|S_{\hat{\jmath}}| \geq \sqrt{n/c}$ ist, daß $\sqrt{n/c}$ Wege durch $\hat{\jmath}$ gehen. D. h.: $C_\pi \geq \sqrt{n/c}$. □

6.9 Korollar:
Das Oblivious Routing auf einem beliebigen Netzwerk der Größe n mit konstantem Grad benötigt $\Omega(\sqrt{n})$ Zeit.

Das bedeutet, daß auch auf Netzwerken aus n Prozessoren mit geringem Durchmesser wie z. B. dem Cube-Connected Cycles-Netzwerk, dem Butterfly-Netzwerk oder dem Shuffle-Exchange-Netzwerk das Oblivious Routing gar nicht effizient durchgeführt werden kann. Ein Ansatz, um das Routing schneller durchführen zu können, muß also für ein Paket zur Laufzeit den Weg in Abhängigkeit von den anderen Paketen wählen.

Ein derartiges Verfahren für ein Netzwerk mit konstantem Grad werden wir im nächsten Abschnitt untersuchen.

Die folgende Bemerkung faßt noch einige interessante Aussagen zum Oblivious Routing zusammen.

6.10 Bemerkung:

a) *In einem Netzwerk M mit n Prozessoren, Grad c, $m \leq n$ festen Eingabeprozessoren (Quellen), und m festen Ausgabeprozessoren (Senken) benötigt jedes Oblivious Routingverfahren für Permutationen im worst case $\Omega\Big(\frac{m}{\sqrt{c \cdot n}}\Big)$ Schritte. Dies kann mit einer etwas modifizierten Analyse, folgend dem Beweis von Satz 6.7, gezeigt werden.*

b) *Oblivious Routing für Permutationen auf dem Hypercube $M(2, \log N)$ mit N Prozessoren ist in Zeit $O(\sqrt{N})$ möglich. Da die untere Schranke eine Mindestzeit von $\Omega(\sqrt{N/\log N})$ liefert, bleibt hier eine Lücke.*

c) *Es gibt ein Netzwerk mit N Prozessoren und Grad $O(\log N)$, auf dem Oblivious Routing für Permutationen in Zeit $O(\sqrt{N/\log N})$ durchgeführt werden kann ([KKT91]).*
D. h., daß bei Betrachtung von Prozessorzahl und -grad alleine die untere Schranke nicht vergrößert werden kann, und daß das Oblivious Routing auf dem Hypercube entweder schneller durchgeführt werden kann, oder daß es eine Struktureigenschaft des Hypercubes gibt, die zusätzlich betrachtet werden muß, um zu zeigen, daß die obere Schranke aus b) nicht verbessert werden kann.

Satz 6.7 zeigt, daß Oblivious Routing in Netzwerken mit kleinem Durchmesser im worst case sehr schlecht ist. Im Butterfly-Netzwerk etwa ist die Dilation nur $\log N$, falls nur von Quellen zu Senken geroutet wird. Wir würden also hoffen, in Zeit $O(\log N)$ routen zu können. Es gilt jedoch:

6.11 Korollar: (zu Satz 6.7 und Bemerkung 6.10 a))

a) *Soll jeder Prozessor von $BF(\log N)$ ein Paket verschicken und eines erhalten, so ist die Laufzeit im worst-case $\Omega(\sqrt{N \log N})$.*

b) *Sollen nur die Prozessoren der Ebenen 0 und $\log N$ Daten verschicken bzw. empfangen, so ist die worst-case-Laufzeit $\Omega(\sqrt{N/\log N})$.*

Diese unteren Schranken machen aber nur Aussagen über sehr schlechte Fälle, die eintreten können, sagen aber nichts darüber aus, wie sich ein Oblivious Routing-Algorithmus in den meisten Fällen auf dem Butterfly-Netzwerk verhalten wird.
Wir werden uns im folgenden Abschnitt davon überzeugen, daß die worst-case-Permutationen „Ausreißer“ sind: Für die „meisten“ Routingprobleme ist die Congestion im Butterfly-Netzwerk viel kleiner als $\sqrt{N/\log N}$.

6.2 Analyse der Congestion: Average-Case-Untersuchungen für das Butterfly-Netzwerk

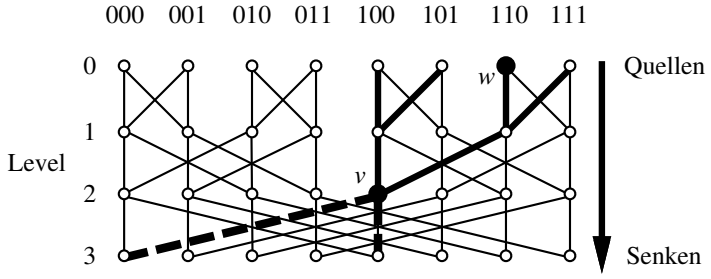
Im vorigen Abschnitt haben wir u. a. gesehen, daß beim Butterfly-Netzwerk einige Permutationen Congestion $\Omega(\sqrt{N/\log N})$ haben. Somit gibt es in F_p Funktionen mit Congestion $\Omega(p \cdot \sqrt{N/\log N})$. In diesem Abschnitt zeigen wir, daß diese „schlechten“ Funktionen Ausreißer sind: Fast alle $f \in F_p$ haben im Butterfly-Netzwerk sehr viel kleinere Congestion.

6.12 Satz:
Sei $\ell > 0$ fest. Für einen Anteil von $1 - 1/N^\ell$ der Funktionen $f \in F_p$ ist die Congestion C_f von f im Butterfly-Netzwerk höchstens

$$2e \cdot p + (\ell + 1) \log N + \operatorname{loglog} N = O(\log N + p) \ .$$

Beweis:
Sei $f \in F_p$ eine zufällig gewählte Funktion. Die Wahrscheinlichkeiten in diesem Beweis beziehen sich immer auf die zufällige Wahl von f. Sei $v = (i, \alpha)$ ein Knoten auf Level i. Mit $P_r(v)$

bezeichnen wir die Wahrscheinlichkeit, daß mindestens r Pakete beim Routing von f durch v gehen, d. h. die Congestion in v mindestens r ist.



Von v aus können $2^{\log N - i} = N/2^i$ Senken erreicht werden. Sei w eine Quelle, von der aus v erreichbar ist. Wenn ein Weg von w aus über v geht, können nur $N/2^i$ der N Senken Ziel sein. Also folgt:

$$\text{Prob(In } w \text{ startender Weg geht über } v) = \frac{N/2^i}{N} = \frac{1}{2^i}$$

Da in jeder Quelle p Pakete starten und v von 2^i Quellen aus erreichbar ist, gibt es $\binom{p\cdot 2^i}{r}$ Möglichkeiten, r Pakete zu wählen, die über v verschickt werden können.
Da die Wahl der Ziele der Pakete unabhängig ist, ist die Wahrscheinlichkeit, daß r Pakete durch v gehen $(\frac{1}{2^i})^r$.
Insgesamt haben wir für $P_r(v)$:

$$P_r(v) \le \binom{p\cdot 2^i}{r} \cdot \left(\frac{1}{2^i}\right)^r \le \left(\frac{p\cdot 2^i \cdot e}{r}\right)^r \cdot \left(\frac{1}{2^i}\right)^r = \left(\frac{pe}{r}\right)^r$$

Bemerkenswert an dieser Abschätzung ist, daß die Wahrscheinlichkeit nicht von v abhängt, wir diese Abschätzung also für jeden Knoten (der keine Quelle ist) benutzen dürfen. D. h.:

$$\begin{aligned}&\text{Prob(Es gibt einen Knoten } v, \text{ über den mindestens } r \text{ Pakete gehen)}\\ &\qquad \le \sum_{v \text{ aus } \mathrm{BF}(\log N)} P_r(v) \;\le\; N\log N \cdot \left(\frac{pe}{r}\right)^r\end{aligned}$$

Diese Wahrscheinlichkeit ist also eine Funktion in p und r. Abbildung 6.3 zeigt exemplarisch für $N = 8$, $p \in [5]$, $r \in [41]$ den Graphen der Funktion; da Wahrscheinlichkeiten nicht größer als 1 sind, werden keine größeren Werte als 1 angezeigt.
Wie man sieht, liegt die gesuchte Wahrscheinlichkeit sehr schnell in der Nähe von 0. (Das Bild zeigt, daß das bei ungefähr $r \approx 6p$ passiert). Wir wählen r so, daß sie unter $1/N^\ell$ sinkt, d. h. $r := 2e\cdot p + (\ell+1)\log N + \mathrm{loglog}\, N$, und haben somit:

$$\begin{aligned}&\text{Prob(Es gibt einen Knoten } v, \text{ über den mindestens } r \text{ Pakete gehen)}\\ &\le\; N\log N \cdot \left(\frac{pe}{r}\right)^r\\ &\le\; N\log N \cdot \left(\frac{1}{2}\right)^{(\ell+1)\log N + \mathrm{loglog}\, N} = N\log N \cdot \frac{1}{N^{\ell+1}\log N} = \frac{1}{N^\ell}\end{aligned}$$

□

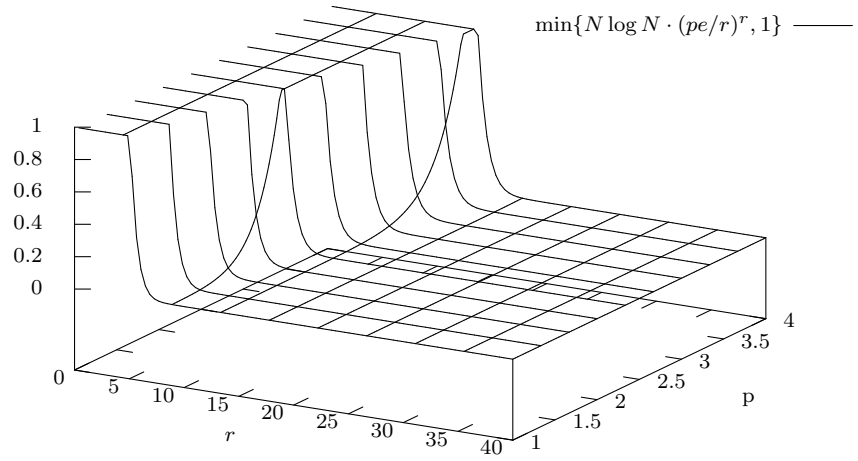


Abbildung 6.3: Der Funktionsgraph für $\min\{N \log N \cdot (pe/r)^r, 1\}$.

6.3 Das Random-Rank-Protokoll

In diesem Abschnitt beschreiben wir ein Routing-Protokoll für Oblivious Routing von Quellen zu Senken (entlang der Wege des Wegesystems aus Beispiel 6.5) des Butterfly-Netzwerks $\mathrm{BF}(\log N)$.

Wir betrachten dazu ein beliebiges Routingproblem, etwa eine Funktion $f \in F_p$. Durch f und das Wegesystem ist die Congestion C_f festgelegt. Da in $\mathrm{BF}(\log N)$ die Dilation immer $\log N$ ist, gilt für die Routingzeit T für f (vgl. Bemerkung 6.6):

$$\max\{C_f, \log N\} \leq T \leq C_f \cdot \log N$$

Wo T in diesem Intervall anzusiedeln ist, hängt vom Routing-Protokoll ab, d. h. von Regeln, die für jeden Prozessor festlegen, welches Paket aus seinem Puffer er im nächsten Schritt weiter senden wird. Solche Protokolle werden auch **Contention-Resolution-Protokolle**, im Deutschen auch einfach **Vorrangsregeln** genannt. Ziel dieses Abschnitts ist es, ein solches Routing-Protokoll zu beschreiben und zu analysieren.

Ein Routing-Verfahren, das diese Wege nimmt, nennt man auch ein **Greedy** (gierig) **Routing**. Es ist leicht zu zeigen, daß es bei dieser Wahl der Wege sogar Permutationen gibt, bei denen $\Theta(\sqrt{N})$ viele Pakete durch einen einzigen Prozessor laufen. Eine dieser Permutationen ist die aus Abschnitt 4.2 bekannte Bit-Reversal-Permutation (vgl. S. 31).

Um das Verfahren vollständig zu beschreiben, benötigen wir noch zusätzlich eine Regel, die, wenn ein Puffer mehr als ein Paket enthält, festlegt, welches davon in einem nächsten Schritt verschickt wird. Dieser Vorgang wird auch als **Contention-Resolution** („einen Streit schlichten“) bezeichnet. Diese Freiheit haben wir ja in einem Oblivious Algorithmus (vgl. S. 58). Wir werden in diesem Abschnitt eine solche Vorrangsregel für die Pakete angeben.

Wir werden zeigen, daß die durchschnittliche Laufzeit des Oblivious Algorithmus $O(\log N)$ ist. Dabei nehmen wir den Durchschnitt über alle Funktionen $f : [N] \times [p] \to [N]$, die wir routen wollen. Unser Algorithmus hat natürlich trotzdem weiterhin eine worst-case-Laufzeit von $\Theta(\sqrt{N})$, da sich die Wege, die die Pakete z. B. bei der Bit-Reversal-Permutation nehmen, nicht ändern.

Hat BF($\log N$) beim Routing der Funktion f eine Congestion von C, so ist

$$\max\{C, \text{dist}((0,\bar{a}),(\log N,\bar{b}))\} = \Omega(C + \log N)$$

eine untere Schranke für die Laufzeit. Wir wollen das Verfahren jetzt so organisieren, daß diese Schranken (bis auf konstante Faktoren) nachweisbar sogar im Durchschnitt erreicht werden.

Dazu wird jedem Paket D_i ein zufällig ausgewählter Rang $\mathit{rang}(D_i) \in [k]$ zugeordnet. Den Wert für k werden wir später genauer spezifizieren.

Wir schreiben $D_i \prec D_j$, falls $\mathit{rang}(D_i) < \mathit{rang}(D_j)$ oder $\mathit{rang}(D_i) = \mathit{rang}(D_j)$ und $i < j$ gilt. „$\prec$“ liefert eine totale Ordnung auf allen Paketen.

Der Routing-Algorithmus sieht nun wie folgt aus. Jedes Paket wandert über den eindeutigen Weg von seiner Quelle zu seiner Senke. Falls in einem Prozessor mehrere Pakete sind, so werden sie nach der folgenden Vorrangsregel weitergeleitet: Das bezüglich der Ordnung „$\prec$“ kleinste Paket wird ausgewählt und verschickt. Diese so beschriebene Vorrangsregel wird **random rank protocol** genannt.

Um die Analyse technisch zu vereinfachen, gehen wir davon aus, daß ein Prozessor auf Level i in einem Schritt von seinen beiden Vorgängern auf Level $i-1$ ein Paket erhalten kann. Dies erhöht die „Schnelligkeit“ des Algorithmus nur um einen konstanten Faktor. In einem Schritt wird er ja weiterhin nur ein Paket los.

6.13 Satz:
Sei $\ell > 0$ fest. Falls mit dem oben beschriebenen Greedy Algorithmus bezüglich der Funktion $f \in F_p$ geroutet wird, und BF($\log N$) bei f die Congestion C hat, reichen $O(\log N + C)$ Schritte mit Wahrscheinlichkeit $1 - 1/N^{\ell}$. Die Wahrscheinlichkeit wird gemessen bezüglich der zufälligen Wahl der Ränge.

Beweis:
Seien die Ränge $\mathit{rang}(D_i)$ für die Vorrangsregel zufällig gewählt. Das Routing von f möge $T = \log N + s$ Schritte dauern. s ist die Gesamtverzögerung. Wir zeigen, daß es sehr unwahrscheinlich ist, daß s groß wird.

$D^{(0)}$ bezeichne ein in Schritt T an seiner Senke v_0 ankommendes Paket. $D^{(0)}$ muß unterwegs s-mal aufgrund der Vorrangsregel nicht weitergeschickt worden sein. Dies soll zum letzten Mal zum Zeitpunkt $T-\ell_0$ geschehen sein ($\ell_0 \geq 1$). Die Verzögerung habe sich in Knoten v_1 auf Level $\log N - \ell_0$ ereignet. D. h., daß im Schritt $T-\ell_0$ ein anderes Paket $D^{(1)}$ den Knoten v_1 verlassen hat. Wegen der Vorrangsregel gilt $D^{(1)} \prec D^{(0)}$ und damit auch $\mathit{rang}(D^{(1)}) \leq \mathit{rang}(D^{(0)})$.

$D^{(1)}$ ist bislang, ebenso wie $D^{(0)}$, $(s-1)$-mal aufgehalten worden, da es zum gleichen Zeitpunkt $T - \ell_0$ im gleichen Level $\log N - \ell_0$ wie $D^{(0)}$ ist.

Wir verfolgen $D^{(1)}$ nun auf seinem Weg zur Quelle soweit zurück, bis es auf Level $\log N - \ell_0 - \ell_1$ in Knoten v_2 in Schritt $T - \ell_0 - \ell_1 - 1$ zuletzt aufgehalten worden ist. Es ist $\ell_1 \geq 0$. Es kann $\ell_1 = 0$ sein, wenn beispielsweise $D^{(0)}$ in v_1 zweimal, $D^{(1)}$ in v_1 einmal aufgehalten worden ist. Da zum Zeitpunkt $T - \ell_0 - \ell_1 - 1$ in v_2 anstatt $D^{(1)}$ das Paket $D^{(2)}$ verschickt wurde, ist $D^{(2)} \prec D^{(1)}$, also wieder insbesondere $\mathit{rang}(D^{(2)}) \leq \mathit{rang}(D^{(1)})$.

Wenn wir dieses Verfahren fortsetzen, landen wir schließlich bei einer Quelle v_{s+1} auf Level 0. Von dort wurde im ersten Schritt das Paket $D^{(s)}$ verschickt. Auf diese Weise erhalten wir eine Folge $v_0, v_1, \ldots, v_{s+1}$ aus nicht notwendig verschiedenen Knoten, die auf einem Weg $\mathcal{W}$ der Länge $\log N$ im Butterfly-Netzwerk BF($\log N$) liegen. Beachte, daß die Pakete $D^{(0)}$ und $D^{(s)}$ eine Sonderstellung einnehmen: $D^{(0)}$ verzögert in v_0 kein anderes Paket, $D^{(s)}$ wird von keinem

anderen Paket verzögert. Ansonsten wird jedes Paket $D^{(i)}$ in v_{i+1} von $D^{(i+1)}$ aufgehalten, und selbst hält es in v_i das Paket $D^{(i-1)}$ auf. In Abbildung 6.4 ist für $s = 3$ ein solcher Weg dargestellt. Diese Betrachtungen führen uns zu folgender Definition:

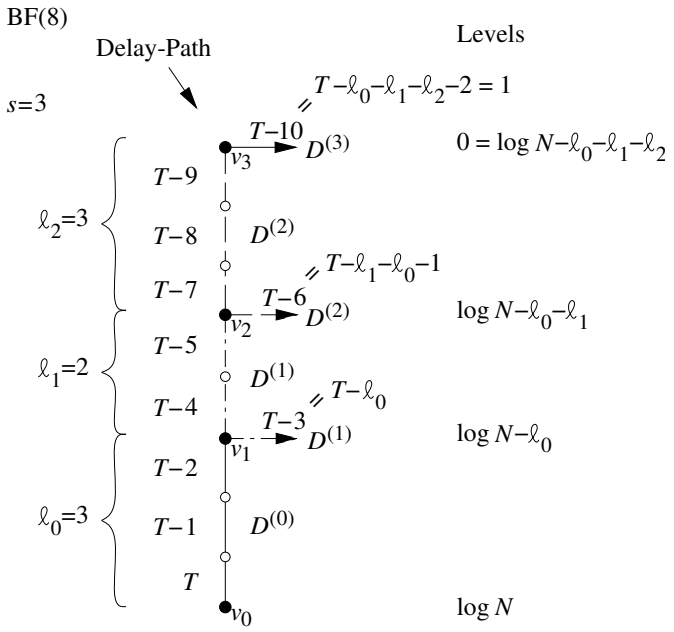


Abbildung 6.4: Zur Konstruktion einer Delay-Sequence.

6.14 Definition: (Delay-Sequence)
Mit den Bezeichnungen aus der obigen Diskussion ist die **Delay-Sequence DS(*rang*) der Länge *s*** bei fester Wahl der Ränge *rang* beschrieben durch

- den **Delay-Path** $\mathcal{W}$,
- die Längen $\ell_0, \ell_1, \ldots, \ell_s$, mit $\ell_0 \geq 1$, $\ell_1, \ldots, \ell_s \geq 0$, der **verzögerungsfreien Teilwege**,
- die **Kollisionsknoten** $v_0, v_1, \ldots, v_{s+1}$,
- und die **Kollisionspakete** $D^{(0)}, \ldots, D^{(s)}$.

Die Eigenschaften der Delay-Sequence, die wir in unserer obigen Diskussion gesehen haben, fassen wir in der folgenden Bemerkung zusammen.

6.15 Bemerkung:
Für die Delay-Sequence DS(rang) gilt:

a) $rang(D^{(0)}) \geq rang(D^{(1)}) \geq \cdots \geq rang(D^{(s)})$

b) $\sum_{i=0}^{s} \ell_i = \log N$

c) *Falls das Routing* $\log N + s$ *Schritte benötigt, hat die Delay-Sequence DS(rang) die Länge* s.

Im folgenden fragen wir danach, wieviele Möglichkeiten es gibt, die Ränge *rang* zu wählen, so daß die korrespondierende Delay-Sequence mindestens die Länge s haben kann.
Dazu „vergessen“ wir zuerst die Bedeutung der Delay-Sequence und definieren uns einfach ein Gerüst, in dem auch die oben beobachteten Delay-Sequences vorkommen.

6.16 Definition: (Formale Delay-Sequence)
Eine **formale Delay-Sequence DS der Länge** s besteht aus

- einem Weg $\mathcal{W}$ der Länge $\log N$ von einer Quelle zu einer Senke;
- $s+1$ ganzen Zahlen mit $\ell_0, \ldots, \ell_s$, mit $\ell_0 \geq 1$, $\ell_1, \ldots, \ell_s \geq 0$ und $\sum_{i=0}^{s} \ell_i = \log N$;
- Knoten $v_0, v_1, \ldots, v_{s+1}$ auf $\mathcal{W}$, wobei v_i auf Level $\log N - \ell_0 - \ell_1 - \ldots - \ell_{i-1}$ liegt;
- $s+1$ Paketen $D^{(0)}, \ldots, D^{(s)}$;
- Zahlen $k_0, \ldots, k_s \in [k]$ mit $k_s \leq k_{s-1} \leq \ldots \leq k_0$.

Eine formale Delay-Sequence DS ist **aktiv**, falls DS = DS(*rang*) ist, so daß für alle $i \in [s+1]$ gilt: $\mathit{rang}(D^{(i)}) = k_i$

6.17 Lemma:
Sei N_s die Zahl der formalen Delay-Sequences der Länge mindestens s. Dann ist

$$\text{Prob}(\textit{Routing von } f \textit{ benötigt mindestens } \log N + s \textit{ Schritte}) \leq \frac{N_s}{k^{s+1}}.$$

Beweis:
In einer Delay-Sequence der Länge s sind $s+1$ Pakete gekennzeichnet.
Es gibt $|\{r \mid r : [N] \times [p] \to [k]\}| = k^{N \cdot p}$ Möglichkeiten, *rang* zu wählen.
Eine formale Delay-Sequence der Länge mindestens s kann für höchstens $k^{N \cdot p - (s+1)}$ viele *rang*-Zuteilungen aktiv sein, da sie $s+1$ viele $\mathit{rang}(D_i)$ festlegt.
Also gibt es für höchstens $N_s \cdot k^{N \cdot p - (s+1)}$ viele *rang*-Zuteilungen aktive formale Delay-Sequences der Länge mindestens s, d. h.

$$\begin{aligned}
&\text{Prob(Routing von } f \text{ benötigt mindestens } \log N + s \text{ Schritte)} \\
&\quad \leq \text{ Prob(es gibt eine aktive formale Delay-Sequence der Länge mindestens } s) \\
&\quad \leq \frac{N_s \cdot k^{N \cdot p - (s+1)}}{k^{N \cdot p}} = \frac{N_s}{k^{s+1}}.
\end{aligned}$$

□ (Lemma 6.17)

Nun haben wir den Wert N_s abzuschätzen.

6.18 Lemma:

$$N_s \leq N^3 \cdot \left(\frac{2e \cdot C(s+k)}{s+1} \right)^{s+1}$$

Beweis:
Wir beginnen damit, abzuzählen, wieviele Möglichkeiten es gibt, die Objekte der einzelnen Punkte in der Definition formaler Delay-Sequences zu wählen:

- Es gibt N^2 Möglichkeiten, den Delay-Path $\mathcal{W}$ zu wählen, da die Wahl von Quelle und Senke den Weg $\mathcal{W}$ bereits festlegt.
- Es gibt $\binom{s+\log N-1}{s}$ Möglichkeiten, die Längen ℓ_i der verzögerungsfreien Teilwege derart zu wählen, daß $\sum_{i=0}^{s} \ell_i = \log N$ ist.

  Dies kann man folgendermaßen zeigen: Wir kodieren die Zahl ℓ_i unär, d. h. durch eine 1-Folge der Länge ℓ_i. Zusätzlich fügen wir an jede Kodierung vorne eine 0 an. Alle diese Zahlen kodieren wir nun mittels einer einzigen 0-1-Folge, indem wir die einzelnen Kodierungen konkatenieren. Da $\sum \ell_i = \log N$ ist, bekommen wir somit die folgende 0-1-Folge der Länge $s + 1 + \log N$:

  $$\underbrace{0\overbrace{1\ldots}^{\ell_0}0\overbrace{\ldots}^{\ell_1}0\ldots0\overbrace{\ldots}^{\ell_s}}_{s+1+\log N}$$

  Die erste 1 ist notwendig, da ja $\ell_0 \geq 1$ gilt.

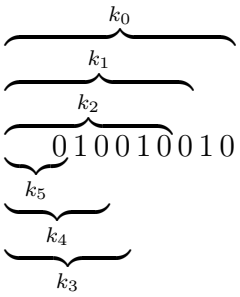
  In dieser Zahlenfolge der Länge $s + 1 + \log N$ sind also genau $s + 1$ viele 0en. Zwei der Positionen sind von vornherein festgelegt, nämlich die erste durch eine 0 und die zweite durch eine 1, in der Restfolge der Länge $s + \log N - 1$ können somit die restlichen s Positionen der 0en frei gewählt werden, jede Wahl entspricht genau einer Festlegung der ℓ_i. Also haben wir für die Anzahl der Möglichkeiten, die ℓ_i zu wählen, $\binom{s+\log N-1}{s}$.

  Im folgenden Beispiel sind $s = 4$, $\log N = 10$, $\ell_0 = 5$, $\ell_1 = 0$, $\ell_2 = 2$, $\ell_3 = 3$:

  $$0\underbrace{11111}_{\ell_0}0\underbrace{}_{\ell_1}0\underbrace{11}_{\ell_2}0\underbrace{111}_{\ell_3}$$

- Die Kollisionsknoten $v_0, \ldots, v_{s+1}$ sind durch $\mathcal{W}$ und $\ell_0, \ldots, \ell_s$ bereits eindeutig bestimmt: v_j ist der Knoten von $\mathcal{W}$ auf Level $\log N - \sum_{i=0}^{j-1} \ell_i$.
- Es gibt für jedes Kollisionspaket $D^{(i)}$ höchstens C Kandidaten, da höchstens C der Pakete über den Knoten v_i laufen. Damit haben wir maximal C^{s+1} Möglichkeiten, $D^{(0)}, \ldots, D^{(s)}$ zu wählen.
- Es gibt höchstens $\binom{s+k}{s+1}$ Möglichkeiten, die k_i zu wählen mit $0 \leq k_s \leq k_{s-1} \leq \ldots \leq k_0 < k$. Auch hier können wir dieses Zählproblem auf 0-1-Folgen reduzieren. Diesmal kodieren wir die Zahlen wie folgt: Wir schreiben $k-1$ viele 1en hin und markieren in dieser Folge durch Dazwischenschreiben von $s + 1$ vielen 0en, welche der 1en zur unären Kodierung von k_i gehören. Seien z. B. $k = 4$ und $s = 5$, sowie $k_5 = 0$, $k_4 = 1$, $k_3 = 1$, $k_2 = 2$, $k_1 = 2$,

$k_0 = 3$. Dies kodieren wir durch den folgenden String:



Die Zahl der „umklammerten“ 1en, startend bei der $(i+1)$-ten 0 von rechts, ist k_i. Die Länge der Folge ist $k-1+s+1 = k+s$, und jede Auswahl von $s+1$ Positionen, an die 0en geschrieben werden, liefert eine andere Folge der k_i. Damit gibt es $\binom{k+s}{s+1}$ Möglichkeiten.

An dieser Stelle geht die Vorrangsregel, durch die die k_i eine monoton fallende Folge sind, in die Analyse ein. Würde jedesmal ein weiterzuleitendes Paket zufällig ausgewählt, so wäre $N_s \geq k^{s+1}$, also nach Lemma 6.17 Prob(Routing von f benötigt mindestens $\log N+s$ Schritte) nur durch einen Wert größer als 1 abgeschätzt. Wir würden also keine Aussage über die Güte des Algorithmus erhalten.

Insgesamt bekommen wir damit:

$$\begin{aligned} N_s &\leq N^2 \cdot \binom{s+\log N-1}{s} \cdot C^{s+1} \cdot \binom{s+k}{s+1} \\ &\leq N^2 \cdot 2^{s+\log N-1} \cdot C^{s+1} \cdot \left(\frac{(s+k)\cdot e}{s+1}\right)^{s+1} \\ &\leq N^3 \cdot \left(\frac{2e\cdot C\cdot(s+k)}{s+1}\right)^{s+1} \end{aligned}$$

□ (Lemma 6.18)

Die beiden letzten Lemmata liefern uns:

$$\text{Prob(Routing von } f \text{ benötigt mindestens } \log N + s \text{ Schritte)} \leq \frac{N_s}{k^{s+1}} \leq N^3 \cdot \left(\frac{2e\cdot C\cdot(s+k)}{(s+1)\cdot k}\right)^{s+1}$$

Wir wählen nun $s := 8e\cdot C - 1 + (\ell+3)\log N$ und $k := s+1$. Eingesetzt ergibt das:

$$\begin{aligned} &\text{Prob(Routing von } f \text{ benötigt mindestens } (\ell+4)\log N + 8eC - 1 \text{ Schritte)} \\ &\overset{k=s+1}{\leq} N^3 \cdot \left(\frac{2e\cdot C\cdot 2(s+1)}{(s+1)^2}\right)^{s+1} \\ &= N^3 \cdot \left(\frac{4e\cdot C}{s+1}\right)^{s+1} \leq N^3 \cdot (\tfrac{1}{2})^{s+1} \\ &\leq N^3 \cdot (\tfrac{1}{2})^{(\ell+3)\log N} = \frac{1}{N^\ell} \end{aligned}$$

□ (Satz 6.13)

Die Sätze 6.12 und 6.13 ergeben dann den folgenden Satz.

6.19 Satz:
Für eine zufällig gewählte Funktion $f \in F_p$ benötigt der Greedy Routing-Algorithmus mit zufälliger Wahl der Ränge mit Wahrscheinlichkeit von mindestens $(1-\frac{1}{N^\ell})^2 \geq 1-\frac{2}{N^\ell}$ höchstens Zeit $O(\log N + p)$. $\ell > 0$ kann beliebig gewählt werden.

6.20 Bemerkung:

a) *In der Praxis ist das hier beschriebene Verfahren sehr schnell. Varianten des Verfahrens wurden auf den Parallelrechnern der Universität Paderborn implementiert.*

b) *In all unseren Analysen haben wir nie benötigt, daß die Quellen auf Level 0 und die Senken auf Level $\log N$ sind, außer daß diese Sichtweise die Rechnung etwas vereinfacht hat. Darum gelten alle Resultate auch für den Fall, daß von Level $\log N$ nach Level 0 geroutet werden soll.*

c) *Auch für die Wahl anderer Vorrangsregeln (z. B. first-in first-out, FIFO) können die gleichen Wahrscheinlichkeiten für eine Laufzeit von $O(\log N + p)$ berechnet werden.*

6.4 Valiants Trick

Der in Abschnitt 6.3 vorgestellte Greedy Algorithmus mit zufällig gewählten Rängen hat ja noch immer eine worst-case-Laufzeit von $\Theta(\sqrt{N})$. Wir werden den Algorithmus jetzt so modifizieren, daß er jedes p-p-Routing-Problem mit hoher Wahrscheinlichkeit schnell löst.
Die algorithmische Idee ist äußerst einfach. Sie wird nach ihrem Erfinder **Valiants Trick** genannt.
Sei $\pi \in F_p$ eine p-p-Funktion. Das j-te Paket in Quelle i, also $D_{i\cdot p+j}$, soll nach Senke $\pi(i,j)$ geschickt werden.

1. Schicke $D_{i\cdot p+j}$ zu einer zufällig gewählten Senke $f(i,j)$ auf Level $\log N$, d. h. route die Pakete gemäß einer zufällig gewählten Funktion $f \in F_p$.
2. Route $D_{i\cdot p+j}$ von dort aus nach Quelle $f(i,j)$ auf Level 0.
3. Route $D_{i\cdot p+j}$ von Quelle $f(i,j)$ nach Senke $\pi(i,j)$.

Satz 6.19 sagt uns, daß Phase 1. mit Wahrscheinlichkeit mindestens $1-\frac{2}{N}$ in Zeit $O(\log N+p)$ durchgeführt wird.
Phase 2. geht in Zeit $O(\log N + p)$, da jede Senke wegen der Phase 1. höchstens $O(\log N + p)$ Pakete bekommt (in t Schritten können nicht mehr als $O(t)$ Pakete in einen Prozessor hinein); diese Pakete werden nun direkt senkrecht auf das Level 0 zurückgeroutet.
3. Die Analyse wollen wir hier nur kurz andeuten. Wir haben hier ein Routing-Problem, das gewissermaßen eine Umkehrung des Problems ist, das wir bislang betrachtet haben. Diesmal sind die Startadressen zufällig und die Zieladressen sind vorgegeben. D. h.: Würden wir die Pakete von den Senken $\pi(i,j)$ zu den Quellen $f(i,j)$ routen wollen, so hätten wir die Ausgangslage, wie wir sie im vorigen Abschnitt betrachtet haben, da wir wieder gemäß einer zufälligen Funktion zu routen hätten. Dieses Routing würde mit Wahrscheinlichkeit $1-\frac{2}{N}$ in Zeit $O(\log N + p)$ beendet sein. Eine nahezu identische Untersuchung wie die im Beweis zu Satz 6.13 zeigt, daß auch dann mit hoher Wahrscheinlichkeit Zeit $O(\log N+p)$ ausreicht, wenn wir „vorwärts“, d. h. von $f(i,j)$ nach $\pi(i,j)$ routen.

Zu beachten ist hierbei noch, daß die Phasen nicht sauber voneinander getrennt laufen können. Phase 1. ist z. B. nur mit hoher Wahrscheinlichkeit nach $O(\log N + p)$ Schritten beendet. Aber diese Phaseneinteilung muß natürlich nicht beibehalten werden, die Phase 2. kann beginnen, sobald das erste Paket aus Phase 1. einen Knoten auf Level $\log N$ erreicht hat. Die Phasen werden „ineinander verschoben“ ausgeführt.

Somit haben wir:

6.21 Satz:
Für jede p-p-Funktion $f \in F_p$ gilt: Mit Wahrscheinlichkeit $(1-\frac{2}{N^\ell})^2 \geq 1-\frac{4}{N^\ell}$ reichen $O(\log N + p)$ Schritte zum Routen von f. $\ell > 0$ kann beliebig gewählt werden.

D. h., daß dieser Algorithmus keine schlechten Eingaben mehr hat. Das ist so zu verstehen: Zu jeder p-p-Funktion f gibt es bei den vielen Möglichkeiten, die es gibt, sie in den drei Phasen durch das Netzwerk zu routen, nur sehr wenige, die eine lange Laufzeit erfordern. Und da alle diese Wege gleich wahrscheinlich sind, ist die erwartete Laufzeit auch für die ungünstige Funktion noch immer $O(\log N+p)$. Der Algorithmus arbeitet also immer korrekt, was passieren kann, wenn man „Pech“ hat, ist, daß er länger benötigt als erwartet.

„Erkauft“ haben wir diese Eigenschaft damit, daß dieser Algorithmus im Durchschnitt ungefähr doppelt so viele Schritte braucht wie der Greedy Algorithmus, da ja in jedem Fall zwei Routingphasen durchgeführt werden, und da ja eben die meisten Routingprobleme schon durch den Greedy Algorithmus effizient gelöst werden können.

Die Phase 2. entfällt, wenn wir das Verfahren auf dem Permutationsnetzwerk $(2, d)$-PN oder einem Butterfly-Netzwerk mit wrap-around-Kanten implementieren.

6.5 Zusammenfassung und abschließende Bemerkungen

Puffergröße. Das Verfahren, das wir in diesem Kapitel kennengelernt haben, erfordert noch immer einen großen Puffer, d. h. an einem Prozessor eine erwartete Puffergröße von $O(C) = O(\log N)$. Der Algorithmus kann so verändert werden, daß er bei gleicher Laufzeit mit konstanter Puffergröße auskommt[Ran91].

Andere Netzwerke. Die in diesem Kapitel vorgestellten Analyse-Methoden für Routing-Algorithmen können auch auf andere Netzwerkfamilien wie z. B. Gitter (u. a. auch dem Hypercube) und Distributoren übertragen werden. Leighton[Lei90] wendet sie beispielsweise auf $M(n, 2)$ und $M(n, 3)$ an; in[LMRR94] wird gezeigt, daß in jedem „Schicht-Netzwerk“ (fast alle interessanten Netzwerke sind von diesem Typ) die Laufzeit $O(\text{Congestion} + \text{Durchmesser})$ mit hoher Wahrscheinlichkeit erreicht wird. In[MV95] wird ein erweitertes Routing-Protokoll eingeführt, daß eine Laufzeit von $O(\text{Congestion} + \text{Dilation})$ auf beliebigen Netzwerken erzielt.

PRAM-Simulationen. Wie man die hier vorgestellten Techniken anwendet, um auf dem Butterfly-Netzwerk das mächtigste Modell für paralleles Rechnen, die PRAM, effizient zu simulieren, werden wir in Kapitel 8.2 untersuchen.

Entwicklung. Die Idee des Routings mit zufälligen Zwischenzielen stammt von Valiant/Brebner[VB81]. Die Analysetechnik mit Hilfe von Delay-Sequences wurde von Upfal eingeführt[Upf84]. Daß man mit konstanter Puffergröße auskommt, hat Ranade[Ran91] gezeigt.

6.6 Literatur zu Kapitel 6

[BH85] A. Borodin and J. E. Hopcroft. Routing, merging, and sorting on parallel models of computation. *Journal of Computer and System Sciences*, 30:130–145, 1985.

[HY98] F. K. Hwang and Y. C. Yao. Comments on the oblivious routing algorithm of Kaklamanis, Krizanc, and Tsantilas in the hypercube. *Theory of Computing Systems*, 31:63–66, 1998.

[KKT91] C. Kaklamanis, D. Krizanc, and T. Tsantilas. Tight bounds for oblivious routing in the hypercube. *Mathematical Systems Theory*, 24:223–232, 1991.

[Lei90] F. T. Leighton. Average case analysis of greedy routing algorithms on arrays. In *Proceedings of 2nd SPAA*, pages 2–10, 1990.

[Lei92a] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[Lei92b] T. Leighton. Methods for message routing in parallel machines. In *Proceedings of the 24th ACM Symposium on Theory of Computing (STOC)*, pages 77–96, 1992.

[LMRR94] T. Leighton, B. M. Maggs, A. G. Ranade, and S. B. Rao. Randomized routing and sorting on fixed connection networks. *Journal of Algorithms*, 17:157–205, 1994.

[MV95] F. Meyer auf der Heide and B. Vöcking. A packet routing protocol for arbitrary networks. In *Proc. 12th Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 291–302, 1995.

[MW96] F. Meyer auf der Heide and R. Wanka. Kommunikation in parallelen Rechnernetzen. In I. Wegener, editor, *Highlights aus der Informatik*, pages 177–198. Springer-Verlag, 1996.

[Ran91] A. G. Ranade. How to emulate shared memory. *Journal of Computer and System Sciences*, 42:307–326, 1991.

[Sch98] C. Scheideler. *Universal Routing Strategies for Interconnection Networks*. Springer-Verlag, Berlin, 1998.

[Upf84] E. Upfal. Efficient schemes for parallel communication. *Journal of the ACM*, 31:507–517, 1984.

[VB81] L. Valiant and G. Brebner. Universal schemes for parallel communication. In *Proceedings of the 13th ACM-STOC*, pages 263–277, 1981.

[Wac90] A. Wachsmann. *Eine theoretische und praktische Untersuchung von Emulationsalgorithmen eines gemeinsamen Speichers auf einem Butterfly-Netzwerk*. Diplomarbeit, Universität Dortmund, 1990.

Kapitel 7

Das Multibutterfly-Netzwerk

Nachdem wir im vorhergehenden Abschnitt gesehen haben, daß der Ansatz über Oblivious Routing immer zu einem schlechten Laufzeitverhalten im worst case führt, werden wir in diesem Abschnitt ein Netzwerk mit konstantem Grad und $N = n(\log n+1)$ Prozessoren vorstellen: das Multibutterfly-Netzwerk. Es kann mit folgenden Eigenschaften beliebige Permutationen routen:

- Es hat n Eingabeprozessoren und n Ausgabeprozessoren;
- Es benötigt kein Preprocessing;
- Die Puffergröße ist 1;
- Die Routing-Zeit ist $O(\log n)$.

Zur Konstruktion dieses Netzwerkes benötigen wir einen sehr interessanten Typ von Netzwerken, nämlich sog. Konzentratoren. Ihnen ist der folgende Abschnitt gewidmet. Sie gehören zu der Klasse der sog. **Expander**.

7.1 Konzentratoren

7.1 Definition:
Seien $\alpha, \beta \in \mathbb{R}^+$, $m, c \in \mathbb{N}$, und sei m eine gerade Zahl. Ein bipartiter Graph $G = (A \cup B, E)$ heißt **(α, β, m, c)-Konzentrator**, falls gilt:

(i) $|A| = m$, $|B| = m/2$;

(ii) Knoten aus A haben maximalen Grad c, Knoten aus B haben maximalen Grad $2c$;

(iii) **(Expansionseigenschaft)**:

$$\text{Für alle } X \subseteq A, |X| \leq \alpha \cdot m, \text{ gilt: } |\Gamma(X)| \geq \beta \cdot |X|$$

D. h.: Jede Knotenmenge aus A (bis zu einer bestimmten Größe) hat sehr viele Nachbarn in B. Wir sagen auch, daß A **expandiert**, und nennen β den **Expansionsfaktor**.

Wegen der Eigenschaft (iii) gilt $\alpha \cdot \beta \leq \frac{1}{2}$, denn sonst wäre Eigenschaft (i) verletzt.

Aus solchen Konzentratoren werden wir unser effizientes Routing-Netzwerk konstruieren. Dazu benötigen wir Konzentratoren mit konstantem Grad und Konstanten α und β, wobei $\beta > 1$ sein muß.

Es ist möglich, Konzentratoren zu konstruieren[LPS88, GS92] (sie sind sogar recht einfach), allerdings sind die Korrektheitsbeweise methodisch sehr aufwendig (Funktionalanalysis und algebraische Graphentheorie spielen eine wesentliche Rolle), und der resultierende Expansionsfaktor β zwar größer als 1, aber doch nur sehr klein, relativ zu c. Wir werden nun die *Existenz* guter Konzentratoren nachweisen.

7.2 Satz:
Für $\alpha \leq \frac{1}{2\beta}\left(4\beta \cdot e^{1+\beta}\right)^{-\frac{1}{c-\beta-1}}$ *gibt es* (α, β, m, c)*-Konzentratoren.*

Beweis:
Betrachte die im folgenden definierte Klasse $\mathcal{R}$ von bipartiten Graphen $G = (A \cup B, E)$, $A = [m]$, $B = [m/2]$.

Für eine beliebige Permutation $\pi : A \to A$ sei $E_\pi = \{(i, j) \in A \times B \mid \pi(i) \in \{j, j + m/2\}\}$. Wir erzeugen also durch die Permutation π ein Matching auf einem bipartiten Graphen mit jeweils m Knoten auf beiden Seiten und „klappen“ dann auf einer Seite eine Hälfte der Knoten auf die andere Hälfte der Knoten. Sei $\mathcal{R} = \{G = (A \cup B, E) \mid E = E_{\pi_1} \cup \cdots \cup E_{\pi_c}$ für Permutationen $\pi_1, \ldots, \pi_c : A \to A\}$.

Jeder Graph $G \in \mathcal{R}$ erfüllt durch seine Definition bereits die Bedingungen (i) und (ii) aus Definition 7.1. Wir zeigen nun, daß es mindestens einen Graphen $G \in \mathcal{R}$ gibt, so daß für G auch Bedingung (iii) erfüllt ist.

Dazu betrachten wir das Zufallsexperiment, zufällig unabhängig c Permutationen $\pi_1, \ldots, \pi_c$ zu wählen. Wir sagen dann: Wir wählen zufällig einen Graphen $G \in \mathcal{R}$ mit Kantenmenge $E_{\pi_1} \cup \cdots \cup E_{\pi_c}$.

7.3 Lemma:
Sei G ein zufällig aus $\mathcal{R}$ gewählter Graph. Dann gilt:

$$\text{Prob}(G \textit{ ist kein } (\alpha, \beta, m, c)\textit{-Konzentrator}) < 1$$

Vor dem Beweis dieses Lemmas folgern wir zuerst daraus den Satz 7.2:

Wegen Lemma 7.3 ist die Wahrscheinlichkeit, daß ein zufällig aus $\mathcal{R}$ gewählter Graph ein (α, β, m, c)-Konzentrator ist, echt größer als 0. Das heißt aber, daß es in $\mathcal{R}$ mindestens einen (α, β, m, c)-Konzentrator gibt, denn sonst wäre die oben genannte Wahrscheinlichkeit ja 0.

Diese Argumentation ist ein sogenanntes **probabilistisches Existenz-Argument**. Es ist sehr elegant, wenn es in erster Linie um die Existenz und nicht um die Konstruktion gewisser Objekte geht. Wie bereits erwähnt, ist es äußerst schwierig, für Graphenfamilien die Konzentratoren-Eigenschaft nachzuweisen, dagegen ist der Beweis von Lemma 7.3 technisch relativ einfach. Es wird nur elementares Rechnen benötigt. □ (Satz 7.2)

Beweis von Lemma 7.3:
In der folgenden Abschätzung wird die Eigenschaft, kein Konzentrator zu sein, genauer formalisiert. Wir berechnen in der Wahrscheinlichkeit gerade das Verhältnis der Zahl an „Nicht-Konzentratoren“ zu der Zahl der Graphen in $\mathcal{R}$ überhaupt.

$$\text{Prob}(G \text{ ist kein } (\alpha, \beta, m, c)\text{-Konzentrator})$$

$$
\begin{aligned}
&\leq \text{Prob}(\exists X \subseteq A, |X| \leq \alpha \cdot m : |\Gamma(X)| < \beta \cdot |X|) \\
&\leq \text{Prob}(\exists \mu \leq \alpha \cdot m, \exists X \subseteq A, \exists Y \subseteq B : |X| = \mu \wedge |Y| = \lfloor \beta \cdot \mu \rfloor \wedge \Gamma(X) \subseteq Y) \\
&\leq \sum_{\mu=1}^{\alpha \cdot m} \sum_{\substack{X \subseteq A \\ |X|=\mu}} \sum_{\substack{Y \subseteq B \\ |Y|=\lfloor \beta \cdot \mu \rfloor}} \text{Prob}(\Gamma(X) \subseteq Y)
\end{aligned}
$$

Seien $X \subseteq A$ und $Y \subseteq B$ mit $|X| = \mu$ und $|Y| = \lfloor \beta \cdot \mu \rfloor$ beliebig aber fest. Wir schätzen jetzt $\text{Prob}(\Gamma(X) \subseteq Y)$ ab.

$$
\begin{aligned}
&\text{Prob}(\Gamma(X) \subseteq Y) \\
&\quad = \text{Prob}\Big(\bigwedge_{i=1}^{c} \pi_i(X) \subseteq Y \cup \underbrace{\{j + m/2 \mid j \in Y\}}_{=:\, Y'} \Big) \\
&\quad = \prod_{i=1}^{c} \text{Prob}(\pi_i(X) \subseteq Y \cup Y') \qquad \text{(da die } \pi_1, \ldots, \pi_c \text{ unabhängig gewählt sind)} \\
&\quad \leq \prod_{i=1}^{c} \frac{\mu! \cdot \binom{2\lfloor \beta\mu \rfloor}{\mu} \cdot (m-\mu)!}{m!} \qquad (7.1) \\
&\quad \leq \prod_{i=1}^{c} \left(\frac{2 \cdot \beta \cdot \mu}{m} \cdot \frac{2 \cdot \beta \cdot \mu - 1}{m-1} \cdot \frac{2 \cdot \beta \cdot \mu - 2}{m-2} \cdot \ldots \cdot \frac{2 \cdot \beta \cdot \mu - \mu + 1}{m - \mu + 1} \right) \\
&\quad \leq \prod_{i=1}^{c} \left(\frac{2 \cdot \beta \cdot \mu}{m} \right)^{\mu} = \left(\frac{2 \cdot \beta \cdot \mu}{m} \right)^{c \cdot \mu}
\end{aligned}
$$

Die Abschätzung (7.1) begründet sich folgendermaßen: Da X und Y fest gewählt sind, fragen wir nach der Anzahl der Permutationen über $A = [m]$, die X nach $Y \cup Y'$ abbilden. Beachte, daß wir hier das Hineinklappen der Permutation π_i wieder rückgängig gemacht haben, und daß $|Y \cup Y'| \leq 2\lfloor \beta\mu \rfloor$ ist. Es gibt $\binom{2\lfloor \beta\mu \rfloor}{\mu}$ Möglichkeiten, Teilmengen der Größe μ in $Y \cup Y'$ zu wählen, und damit $\mu! \cdot \binom{2\lfloor \beta\mu \rfloor}{\mu}$ Möglichkeiten, die Werte aus X nach $Y \cup Y'$ abzubilden. Zusätzlich haben wir $(m-\mu)!$ Möglichkeiten, die restlichen Funktionswerte zu wählen. Diese Zahl wird ins Verhältnis zur Gesamtzahl $m!$ der möglichen Permutationen gesetzt.

Die so berechnete Abschätzung für $\text{Prob}(\Gamma(X) \subseteq Y)$, die ja nicht von der Wahl von X und Y abhängt, setzen wir nun ein.

$$
\begin{aligned}
&\text{Prob}(G \text{ ist kein } (\alpha, \beta, m, c)\text{-Konzentrator}) \\
&\quad \leq \sum_{\mu=1}^{\lfloor \alpha \cdot m \rfloor} \sum_{\substack{X \subseteq A \\ |X|=\mu}} \sum_{\substack{Y \subseteq B \\ |Y|=\lfloor \beta \cdot \mu \rfloor}} \left(\frac{2 \cdot \beta \cdot \mu}{m} \right)^{c \cdot \mu} \\
&\quad \leq \sum_{\mu=1}^{\lfloor \alpha \cdot m \rfloor} \binom{m}{\mu} \cdot \binom{m/2}{\lfloor \beta\mu \rfloor} \cdot \left(\frac{2 \cdot \beta \cdot \mu}{m} \right)^{c \cdot \mu} \\
&\quad \leq \sum_{\mu=1}^{\lfloor \alpha \cdot m \rfloor} \left(\frac{e \cdot m}{\mu} \right)^{\mu} \cdot \left(\frac{e \cdot m/2}{\beta \cdot \mu} \right)^{\beta \cdot \mu} \cdot \left(\frac{2 \cdot \beta \cdot \mu}{m} \right)^{c \cdot \mu} \qquad \text{(s. Lemma 7.4)} \\
&\quad = \sum_{\mu=1}^{\lfloor \alpha \cdot m \rfloor} \left[\left(\frac{m}{\mu} \right)^{1+\beta-c} \cdot e^{1+\beta} \cdot (2 \cdot \beta)^{c-\beta} \right]^{\mu}
\end{aligned}
$$

$$\begin{aligned} &< \sum_{\mu=1}^{\infty} \Big(\underbrace{\alpha^{c-1-\beta} \cdot e^{1+\beta} \cdot (2\beta)^{c-\beta}}_{=:\ r} \Big)^{\mu} \qquad (\text{da } \mu \le \alpha \cdot m) \\ &\le 1 \end{aligned}$$

Das letzte „$\le$“ gilt, wenn $r \le \frac{1}{2}$ ist, denn es gilt $\sum_{\mu=1}^{\infty} (\frac{1}{2})^{\mu} = 1$ (unendliche geometrische Reihe).

$r \le \frac{1}{2}$ gilt aber wegen der Voraussetzung des Satzes, denn:

$$\begin{aligned} \alpha &\le \frac{1}{2\beta} \left(4\beta \cdot e^{1+\beta}\right)^{-\frac{1}{c-\beta-1}} = \left(\left(\frac{1}{2\beta}\right)^{c-\beta-1} \cdot \frac{1}{2} \cdot \frac{1}{2\beta} \cdot e^{-(1+\beta)} \right)^{\frac{1}{c-\beta-1}} \\ &= \left(\frac{1}{2} \cdot \left(\frac{1}{2\beta}\right)^{c-\beta} \cdot e^{-(1+\beta)} \right)^{\frac{1}{c-\beta-1}} = \left(\tfrac{1}{2} \cdot (2\beta)^{-(c-\beta)} \cdot e^{-(1+\beta)}\right)^{\frac{1}{c-\beta-1}} \end{aligned}$$

$$\iff \qquad \tfrac{1}{2} \ge \alpha^{c-\beta-1} \cdot (2\beta)^{c-\beta} \cdot e^{1+\beta} = r$$

□ (Lemma 7.3)

Die oben benutzte Abschätzung für Binomialkoeffizienten wird oft angewandt. Darum soll sie hier nicht unbewiesen bleiben.

7.4 Lemma:
Für $k, \ell \in \mathbb{N}$, $0 \le \ell \le k$, gilt: $\binom{k}{\ell} \le \left(\frac{k \cdot e}{\ell}\right)^{\ell}$

Beweis:
$\binom{k}{\ell} = \frac{k \cdot (k-1) \cdots (k-\ell+1)}{\ell!} \le \frac{k^{\ell}}{\ell!} = \frac{k^{\ell}}{\ell^{\ell}} \cdot \frac{\ell^{\ell}}{\ell!} \le \frac{k^{\ell}}{\ell^{\ell}} \cdot e^{\ell}$, da $e^{\ell} = \sum_{i=0}^{\infty} \frac{\ell^i}{i!} \ge \frac{\ell^{\ell}}{\ell!}$ gilt. □

Man beachte, daß in der Bedingung „$\alpha \le \frac{1}{2\beta} \overbrace{\left(4\beta \cdot e^{1+\beta}\right)^{-\frac{1}{c-\beta-1}}}^{(*)}$“ der Term $(*)$ bei konstantem β für wachsenden Grad c (sehr schnell) gegen 1 geht, also $\alpha \cdot \beta$ sehr nah an den größtmöglichen Wert $\frac{1}{2}$ herankommt.

7.5 Beispiel:
Für $\beta = 2$ und $c = 4$ hat zu gelten $\alpha \le \frac{1}{32e^3}$. Das ist für $\alpha = \frac{1}{643}$ der Fall. Satz 7.2 garantiert uns dann die Existenz eines $(\frac{1}{643}, 2, m, 4)$-Konzentrators.

7.2 Definition des Multibutterfly-Netzwerks

Aus den gerade vorgestellten Konzentratoren bauen wir jetzt unser Netzwerk, auf dem wir schnell routen können. Dazu „verdoppeln“ wir zuerst die Konzentratoren zu sog. Teilern.

7.6 Definition:
Ein **(α, β, m, c)-Teiler** (oder auch **-Splitter**) ist ein bipartiter Graph $(A \dot\cup (B_0 \cup B_1), E_0 \cup E_1)$, wobei $(A \dot\cup B_0, E_0)$ und $(A \dot\cup B_1, E_1)$ jeweils (α, β, m, c)-Konzentratoren sind. Der Grad eines (α, β, m, c)-Teilers ist $2c$. Eine Kante nach B_0 heißt **0-Kante**, eine nach B_1 **1-Kante**.

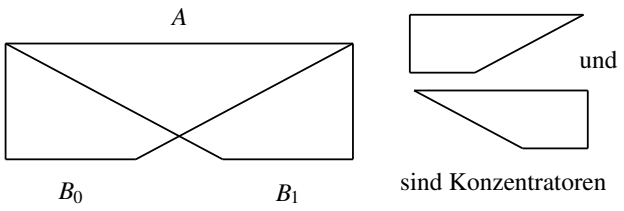


Abbildung 7.1: Schematischer Aufbau eines Teilers.

Aus den Teilern setzen wir jetzt das Multibutterfly-Netzwerk zusammen:

7.7 Definition:
Das **Multibutterfly-Netzwerk MBN(d, α, β, c)** hat $N = 2^d \cdot (d+1)$ Knoten und Grad $4c$. Abbildung 7.2 zeigt seine rekursive Defintion.

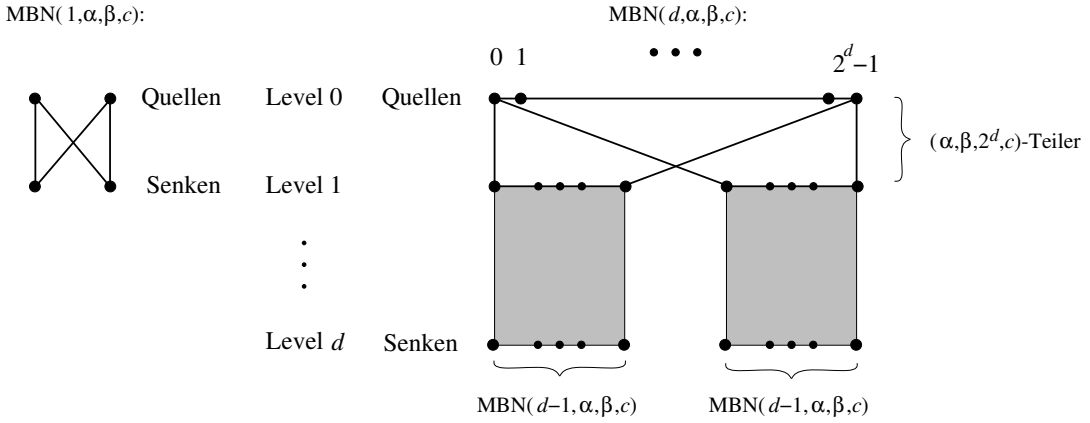


Abbildung 7.2: Rekursive Definition von MBN(d, α, β, c).

Die Knoten auf Level 0 heißen **Quellen**, die auf Level d **Senken** von MBN(d, α, β, c). Das Routing-Problem, das wir betrachten, besteht darin, von jeder Quelle zu genau jeder Senke genau ein Paket bzgl. einer beliebigen Permutation zu versenden. Der Abstand zwischen einer beliebigen Quelle und einer beliebigen Senke ist d.

Beachte dabei, daß das Butterfly-Netzwerk BF(d) ein spezielles Multibutterfly-Netzwerk ist (die Levels sind hier aus technischen Gründen andersherum numeriert), in dem die Teiler spezielle $(1, \frac{1}{2}, m, 1)$-Teiler sind, wobei die beiden Konzentratoren je bezüglich der identischen Permutation gebildet sind. Der Expansionsfaktor ist $\frac{1}{2}$. In unserer Analyse für das Multibutterfly-Netzwerk werden wir den Expansionsfaktor $\beta > 1$ benötigen.

7.3 Routing im Multibutterfly-Netzwerk

Seien α, β, c nun feste Konstanten. $n = 2^d$ sei die Zahl der Quellen. Wir werden diese Parameter später genauer spezifizieren. Im folgenden benuzten wir MBN(d) als Abkürzung für MBN(d, α, β, c). Sei $L = \lceil \frac{1}{2\alpha} \rceil$, und bezeichne A_i die Menge der Pakete mit Zieladresse j, wobei $j \bmod L = i$ ist.

Dann haben $A_0, \ldots, A_{L-1}$ je etwa $\frac{n}{L}$ Elemente. In ein Submultibutterfly-Netzwerk MBN(d'), $d' < d$, gehören darum höchstens $\frac{2^{d'}}{L}$ viele Pakete, jeder der zugehörigen Konzentratoren muß also nur $\frac{1}{2} \cdot \frac{2^{d'}}{L} \approx \alpha \cdot 2^{d'}$ Pakete weiterleiten. Das sind aber nach Definition gerade so wenige, daß sie mit dem Faktor β expandiert werden können.

Wir beschreiben zunächst, wie der Weg von den Quellen zu den Senken ungefähr aussieht: Der **Grobe Weg** eines Pakets von $i = (i_{d-1}, \ldots, i_0)_2$ auf Level 0 nach $j = (j_{d-1}, \ldots, j_0)_2$ auf Level d geht zuerst über eine j_{d-1}-Kante, dann über eine j_{d-2}-Kante usw. Abbildung 7.3 gibt die Groben Wege in MBN(3) von den Quellen zur Senke $(100)_2$ an.

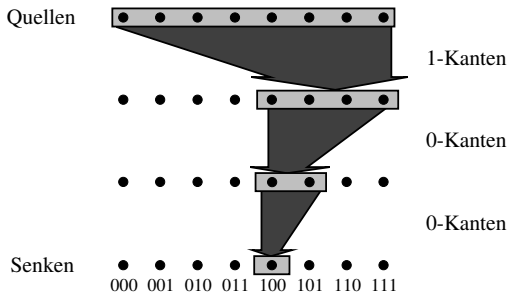


Abbildung 7.3: Ein Grober Weg.

Im Butterfly-Netzwerk ist der Grobe Weg gemäß Bemerkung 2.17 (S. 10) und Bespiel 6.5 (S. 57) der einzige Weg von Quelle i nach Senke j, im Multibutterfly-Netzwerk dagegen gibt es viele mögliche Wege, da es von jedem Knoten aus c viele 0-Kanten und c viele 1-Kanten gibt. Diese mögliche Auswahl der Wege ist im Butterfly-Netzwerk nicht möglich, weshalb der Algorithmus über die Groben Wege dort oblivious ist. Sie macht den Algorithmus im MBN nicht-oblivious.

Routing-Algorithmus für MBN(d):

- Der Algorithmus arbeitet in L Schüben, ein Schub besteht aus einzelnen Runden. Pro Schub werden ausschließlich die maximal $\frac{n}{L}$ Elemente aus A_k verschickt. Sei k nun fest, $|A_k| \leq \frac{n}{L}$.
- In einer Runde des Routing-Algorithmus für A_k wird folgendes gemacht:
  - Jeder Prozessor, der ein Paket hat, ist **blockiert**. Die Kanten jedes Teilers seien in $2c$ Matchings $\mathcal{M}_1, \ldots, \mathcal{M}_{2c}$ zerlegt (Diese Matchings gibt es wegen des Korollars 2.14 zum Heiratssatz).
  - `for` $j := 1$ `to` $2c$ `do`
    Jeder Prozessor, der ein Paket hat, das über eine s-Kante muß für $s \in \{0, 1\}$, sendet es über eine s-Kante im Matching $\mathcal{M}_j$, falls dort eine solche dabei ist, und sofern der darüber erreichbare Prozessor nicht bereits blockiert ist.
  - Ein Prozessor, der ein Paket bekommt, ist für den Rest der Runde blockiert.

Eine Runde dauert $O(c)$, also konstant viele Schritte.

Wieviele Runden werden benötigt, bis alle Pakete aus A_k bei den Senken auf Level d angekommen sind, d. h. ein Schub beendet ist? Im folgenden werden wir die Zahl der Runden für einen Schub analysieren.

Sei dazu ein Zeitpunkt $t \geq 0$ fest. Wir untersuchen nun die Verteilung der Pakete im Netzwerk, wir machen also eine Momentaufnahme davon, wo welches Paket nach Runde t ist, und wir untersuchen, was in der Runde $t+1$ mit den Paketen passiert: Nach t Runden seien K_i Pakete auf Level i von MBN(d), $0 \leq i \leq d-1$, angekommen. Nach der $(t+1)$-ten Runde seien dann K'_i Pakete auf Level i, davon mögen b_i viele in dieser Runde liegengeblieben (blockiert) und a_i viele hinzugekommen (aktiv) sein. Also ist $K'_i = a_i + b_i$.
Es gelten folgende Beziehungen:

7.8 Lemma:

a) $a_0 = 0$, *da auf Level 0 keine neuen Pakete auftauchen können.*

b) $a_i = K_{i-1} - b_{i-1}$ *für* $i > 0$, *da auf Level* i *alle* K_{i-1} *vielen neu hinzu kommen, die vorher auf Level* $i-1$ *waren, bis auf die* b_{i-1} *vielen, die dort liegengeblieben sind.*

c) $b_i \leq \frac{1}{\beta+1} \cdot (K_{i+1} + K_i)$ *für* $i \leq d-2$.

d) $b_{d-1} = 0$, *da alle Pakete auf Level* $d-1$ *sofort weitergeleitet werden, und somit keines liegen bleibt.*

Beweis:
Bis auf die Aussage c) sind alle Punkte bereits begründet.

Zu c):
Betrachte ein Submultibutterfly-Netzwerk MBN($d-i$). Seine $m = 2^{d-i}$ Quellen liegen auf Level i.

Sei z die Zahl der in der Runde $t+1$ liegengebliebenen Pakete, die über eine 0-Kante (bzw. eine 1-Kante) des durch die 0-Kanten (bzw. 1-Kanten) spezifizierten (α, β, m, c)-Konzentrators das Level verlassen müssen. z ist höchstens αm, da in das entsprechende MBN($d-i-1$) nur $\frac{1}{2}m/L \approx \alpha m$ Pakete hineinwollen. Diese Pakete werden durch mindestens βz Pakete auf Level $i+1$ blockiert, da hier die Expansionseigenschaft gilt. Es gibt in Runde $t+1$ höchstens $K_{i+1} + a_{i+1}$ blockierende Pakete auf Level $i+1$, also: $\beta \cdot b_i \leq K_{i+1} + a_{i+1}$.
Zusammen ergibt das:

$$\begin{aligned} & \beta \cdot b_i && \leq && K_{i+1} + a_{i+1} \overset{\text{b)}}{=} K_{i+1} + K_i - b_i \\ \Rightarrow\ & b_i \cdot (1+\beta) && \leq && K_{i+1} + K_i \\ \Rightarrow\ & b_i && \leq && \frac{1}{1+\beta} \cdot (K_{i+1} + K_i) \end{aligned}$$

□

Mit Hilfe dieses Lemmas wollen wir die Laufzeitschranke folgern. Dazu benutzen wir eine **Potentialfunktion**, die der Situation zur Zeit t, beschrieben durch $K_0, \ldots, K_{d-1}$, ein Gewicht zuordnet, oder mit anderen Worten, die die Situation nach der Runde t in einer Zahl kodiert. Sei $\omega \in (0,1)$ beliebig aber fest. Wir definieren $\Phi_t := \sum_{i=0}^{d-1} K_i \cdot \omega^i$. Dadurch, daß $\omega < 1$ gilt, wird einer Situation nach Runde t eine um so kleinere Zahl Φ_t zugeordnet, je näher die Pakete an ihr Ziel gekommen sind. Φ_t ist eine Bewertung der Situation.

7.9 Lemma:

a) $\Phi_0 = \frac{n}{L}$.

b) *Falls $\Phi_T < \omega^{d-1}$ ist, sind alle Pakete bei den Senken angekommen, d. h. der Schub ist beendet.*

c) $\Phi_{t+1} \leq \Phi_t \cdot \left(\frac{\beta}{\beta+1}\omega + \frac{1}{(\beta+1)\cdot\omega}\right)$.

Beweis:

Zu a)
Zu Beginn gibt es nur die $\frac{n}{L}$ Pakete in Level 0, d. h. $K_0 = \frac{n}{L}$, $K_i = 0$ für $i = 1, \ldots, d-1$. Also ist $\Phi_0 = \frac{n}{L} \cdot \omega^0 = \frac{n}{L}$.

Zu b)
Falls $\Phi_T < \omega^{d-1}$ ist, ist $\Phi_T = 0$, und somit alle $K_i = 0$, d. h. alle Pakete sind auf Level d angekommen, also am Ziel.

zu c)

$$
\begin{aligned}
\Phi_{t+1} = \sum_{i=0}^{d-1} K'_i\omega^i \;&=\; \sum_{i=0}^{d-1}(b_i + a_i)\omega^i \\
&= b_0 + \underbrace{a_0}_{=0} + \sum_{i=1}^{d-1}(b_i + K_{i-1} - b_{i-1})\omega^i \\
&= \sum_{i=1}^{d-1} K_{i-1}\omega^i + \sum_{i=0}^{d-2} b_i(\underbrace{\omega^i - \omega^{i+1}}_{>0}) + \underbrace{b_{d-1}\omega^{d-1}}_{=0} \\
&\leq \sum_{i=1}^{d-1} K_{i-1}\omega^i + \sum_{i=0}^{d-2} \frac{1}{\beta+1}(K_{i+1} + K_i)(\omega^i - \omega^{i+1}) \\
&= K_0 \cdot \left(\omega^1 + \frac{1}{\beta+1}(\omega^0 - \omega^1)\right) \\
&\quad + \; K_1 \cdot \left(\omega^2 + \frac{1}{\beta+1}(\omega^0 - \omega^1 + \omega^1 - \omega^2)\right) \\
&\quad + \; K_2 \cdot \left(\omega^3 + \frac{1}{\beta+1}(\omega^1 - \omega^2 + \omega^2 - \omega^3)\right) \\
&\quad \vdots \\
&\quad + \; K_i \cdot \left(\omega^{i+1} + \frac{1}{\beta+1}(\omega^{i-1} - \omega^i + \omega^i - \omega^{i+1})\right) \\
&\quad \vdots \\
&\quad + \; K_{d-2} \cdot \left(\omega^{d-1} + \frac{1}{\beta+1}(\omega^{d-3} - \omega^{d-2} + \omega^{d-2} - \omega^{d-1})\right) \\
&\quad + \; K_{d-1} \cdot \left(\frac{1}{\beta+1}(\omega^{d-2} - \omega^{d-1})\right) \\
&= K_0 \cdot \omega^0 \left(\omega + \frac{1}{\beta+1}(1-\omega)\right) \qquad (*)
\end{aligned}
$$

$$
\begin{aligned}
&+ K_1 \cdot \omega^1 \left(\omega + \frac{1}{\beta+1}\left(\frac{1}{\omega} - \omega\right)\right) \\
&+ K_2 \cdot \omega^2 \left(\omega + \frac{1}{\beta+1}\left(\frac{1}{\omega} - \omega\right)\right) \\
&\vdots \\
&+ K_i \cdot \omega^i \left(\omega + \frac{1}{\beta+1}\left(\frac{1}{\omega} - \omega\right)\right) \\
&\vdots \\
&+ K_{d-2} \cdot \omega^{d-2} \left(\omega + \frac{1}{\beta+1}\left(\frac{1}{\omega} - \omega\right)\right) \\
&+ K_{d-1} \cdot \omega^{d-1} \left(\frac{1}{\beta+1}\left(\frac{1}{\omega} - 1\right)\right) && (**) \\
&\leq \sum_{i=0}^{d-1} K_i \omega^i \left(\omega + \frac{1}{\beta+1}\left(\frac{1}{\omega} - \omega\right)\right) && \text{(wegen } \tfrac{1}{\omega} > 1 \text{ für } (*) \text{ und } \omega > 0 \text{ für } (**)\text{)} \\
&= \Phi_t \cdot \left(\left(1 - \frac{1}{\beta+1}\right)\omega + \frac{1}{(\beta+1)\omega}\right) \\
&= \Phi_t \cdot \left(\frac{\beta}{\beta+1}\omega + \frac{1}{(\beta+1)\omega}\right)
\end{aligned}
$$

Das bedeutet, daß, wenn $\frac{\beta}{\beta+1}\omega + \frac{1}{(\beta+1)\omega} < 1$ ist, die Folge der Φ_t gegen 0 konvergiert. □

7.10 Satz:
Sei $\beta > 1$ beliebig, seien α und c so gewählt, daß für jedes d MBN(d, α, β, c) existiert. Dann kann MBN(d, α, β, c) beliebige Permutationen von $n = 2^d$ Paketen in Zeit $O(\log n)$ mit Puffergröße 1 und ohne Preprocessing routen. MBN(d, α, β, c) hat $n(\log n + 1)$ Prozessoren und Grad $4c$.

Beweis:
Sei $\delta := \frac{\beta}{\beta+1}\omega + \frac{1}{(\beta+1)\omega}$.
Wir müssen gemäß Lemma 7.9 $\omega \in (0,1)$ und $\delta \in (0,1)$ wählen. Aus der Definition von δ können wir folgende Beziehung zwischen δ und ω bestimmen:

$$
\begin{aligned}
& \frac{\beta}{\beta+1}\omega + \frac{1}{(\beta+1)\omega} = \delta \\
\iff \quad & \omega^2 - \delta \cdot \frac{\beta+1}{\beta} \cdot \omega + \frac{1}{\beta} = 0 \\
\iff \quad & \omega_{1/2} = \frac{1}{2}\frac{\delta(\beta+1)}{\beta} \pm \sqrt{\left(\frac{\delta(\beta+1)}{2\beta}\right)^2 - \frac{1}{\beta}}
\end{aligned}
$$

Wähle $\delta = \frac{2\sqrt{\beta}}{\beta+1}$. Da $\beta > 1$ ist, gilt $\delta < 1$.

Dann ist $\omega = \frac{1}{2}\frac{\beta+1}{\beta}\left(\frac{2\sqrt{\beta}}{\beta+1}\right) = \frac{1}{\sqrt{\beta}}$ die einzige Lösung, da der Wurzelterm zu 0 wird. Zudem ist $\omega < 1$, da $\beta > 1$ ist.

Aus Lemma 7.9 folgt:

- Zu Beginn ist $\Phi_0 = \frac{n}{L}$;
- in jeder Runde wird Φ_t um den Faktor δ kleiner, also: $\Phi_t \leq \frac{n}{L} \cdot \delta^t$;
- für das erste T mit $\Phi_T < \omega^{d-1}$ gilt: T Runden reichen für einen Schub.

Wir müssen also $\min\{T \mid \Phi_T < \omega^{d-1}\}$ bestimmen.

$$\begin{aligned} & \frac{n}{L} \cdot \delta^T < \omega^{d-1} \\ \iff \quad & \delta^T < \omega^{d-1} \cdot \frac{L}{n} \\ \iff \quad & \left(\frac{1}{\delta}\right)^T > \left(\frac{1}{\omega}\right)^{d-1} \cdot \frac{n}{L} \\ \iff \quad & T \cdot \log\left(\frac{1}{\delta}\right) > (d-1) \cdot \log\left(\frac{1}{\omega}\right) + \log\left(\frac{n}{L}\right) \end{aligned}$$

Also reichen

$$T = \left\lceil \frac{(d-1) \cdot \log\left(\frac{1}{\omega}\right) + \log\left(\frac{n}{L}\right)}{\log\left(\frac{1}{\delta}\right)} \right\rceil = O\left(d + \log\left(\frac{n}{L}\right)\right) = O(\log n)$$

Runden für einen Schub.
Da wir L Schübe durchführen, ist die Gesamtlaufzeit $O(L \cdot \log n) = O(\frac{\log n}{\alpha}) = O(\log n)$. □

7.11 Beispiel:
In Beispiel 7.5 haben wir gesehen, daß es $(\frac{1}{643}, 2, 2^d, 4)$-Konzentratoren für beliebige d gibt. Unsere Parameter wählen wir wie folgt:

$$L := 322 \qquad \delta := \tfrac{2}{3}\sqrt{2} \approx 0,9428 \qquad \omega := \tfrac{1}{\sqrt{2}} \approx 0,7072$$

Dann haben wir in $\mathrm{MBN}(\log n, \frac{1}{643}, 2, 4)$ ein Netzwerk mit Grad 16 und für die Zahl T der Runden je Schub: $T \approx 17,6549 \cdot \log n - 102$ bei $L = 322$ Schüben und einer Dauer von 8 Schritten für eine Runde.

7.12 Bemerkung:

a) *Sei $N = 2^d(d+1)$. Enthält jeder Prozessor von MBN(d) ein Paket, so können auch noch N Botschaften gemäß beliebiger Permutationen $\pi : [N] \to [N]$ in Zeit $O(\log N)$ geroutet werden; $p > N$ viele Botschaften benötigen (optimale) Zeit $O(\frac{p}{N} + \log N)$.*

b) *Experimente zeigen, daß das Routing im MBN(d) sehr schnell ist. Bei einer Implementierung kann man ja auch darauf verzichten, das Verfahren in Schüben arbeiten zu lassen. Die Schübe haben wir nur eingeführt, um die Analyse zu vereinfachen.*

c) *Ein weiterer Vorteil besteht darin, daß MBN(d) hochgradig fehlertolerant ist. D. h.: Falls einige Links oder Prozessoren ausfallen, ist die Leistungsfähigkeit des Netzwerks immer noch sehr gut.*

d) *Probleme mit der Akzeptanz bei Praktikern: (z. B.: Ingenieure, die Routing Chips bauen) Die Konzentratoren sind unregelmäßig und unstrukturiert; man kann nicht aus kleinen Konzentratoren einen Großen bauen. Die besten Konzentratoren erhält man, indem man zufällig Graphen mit passendem Grad „auswürfelt“.*

7.4 Literatur zu Kapitel 7

[GS92] M. Gruß and H. Selke. Über die explizite Konstruktion von Expandergraphen. Diplomarbeit, Universität-GH Paderborn, 1992.

[Lei92] T. Leighton. Methods for message routing in parallel machines. In *Proceedings of the 24th ACM Symposium on Theory of Computing (STOC)*, pages 77–96, 1992.

[LM92] F. T. Leighton and B. M. Maggs. Fast algorithms for routing around faults in multibutterflies and randomly-wired splitter networks. *IEEE Transactions on Computers*, 41:578–587, 1992.

[LPS88] A. Lubotzky, R. Phillips, and P. Sarnak. Ramanujan graphs. *Combinatorica*, 8:261–277, 1988.

[MV97] B. M. Maggs and B. Vöcking. Improved routing and sorting on multibutterflies. In *Proc. 29th ACM Symposium on Theory of Computing (STOC)*, pages 517–530, 1997.

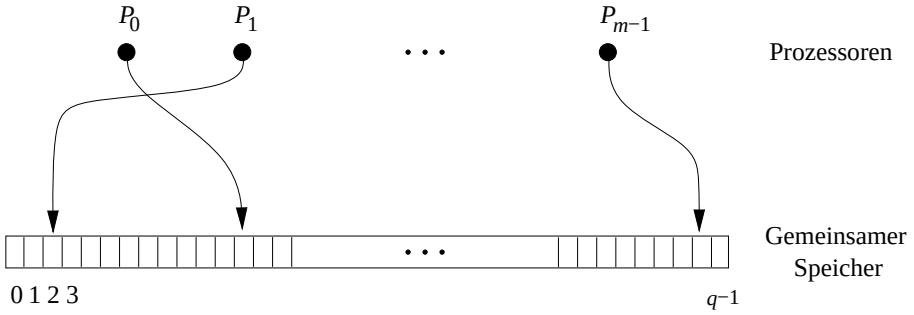
[Par90] I. Parberry. An optimal time bound for oblivious routing. *Algorithmica*, 5:243–250, 1990.

[Upf92] E. Upfal. An $O(\log N)$ deterministic packet-routing scheme. *Journal of the ACM*, 39:55–70, 1992.

Kapitel 8

Simulationen von PRAMs durch Distributed Memory Machines

In Abschnitt 1.1 der Einleitung wurde ein spezieller Parallelrechner, die **parallele Registermaschine** (**Parallel random access machine**, **PRAM**) vorgestellt. Die PRAM besteht aus den Prozessoren $P_0, \ldots, P_{m-1}$ und einem gemeinsamen Speicher aus q Registerzellen $0, \ldots, q-1$.



Jeder PRAM-Prozessor kann pro Schritt auf eine beliebige Zelle des gemeinsamen Speichers lesend oder schreibend zugreifen. Die Prozessoren arbeiten synchron, wobei ein Rechenschritt aus drei Phasen besteht. In der ersten Phase wird eine interne Berechnung, in der zweiten wird ein Leseschritt, und in der dritten ein Schreibschritt ausgeführt.

Wenn mehrere Prozessoren in einem Schritt auf die gleiche Zelle zugreifen wollen, so gibt es die folgenden Möglichkeiten, diese **Zugriffskonflikte** zu regeln:

- **ER** (*Exclusive Read*): das gleichzeitige Lesen in derselben Zelle ist verboten.
- **CR** (*Concurrent Read*): gleichzeitiges Lesen in derselben Zelle ist erlaubt.
- **EW** (*Exclusive Write*): das gleichzeitige Schreiben in dieselbe Zelle ist verboten.
- **CW** (*Concurrent Write*): gleichzeitiges Schreiben in dieselbe Zelle ist erlaubt.

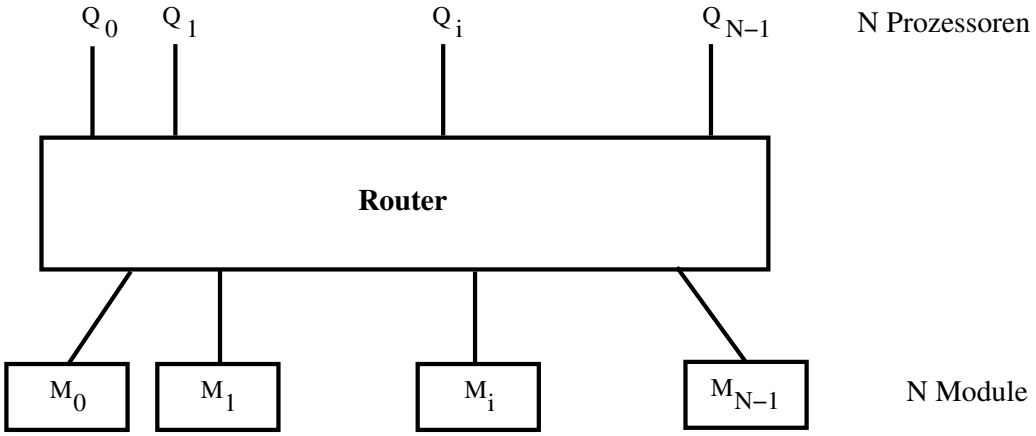
Die PRAMs werden durch die entsprechenden Abkürzungen bezeichnet, also z. B. „EREW-PRAM“ für eine exclusive-read-exclusive-write-PRAM. In einer CRCW-PRAM können also Konflikte beim gleichzeitigen Schreiben durch mehrere Prozessoren in der gleichen Zelle auftreten. Diese können durch die sogenannten **Schreibkonfliktregeln** aufgelöst werden. Eine

dieser Regeln ist die ARBITRARY-Regel. Bei ihr „gewinnt“ ein beliebiger der Prozessoren P_i, die gleichzeitig in die Zelle u schreiben wollen. Diese Regel werden wir in Abschnitt 8.3 in etwas anderem Rahmen benutzen. Es gibt noch weitere mögliche Regeln, die wir hier aber nicht vorstellen werden (vgl. Abschnitt 8.5). Alle Resultate in diesem Kapitel beschreiben wir für EREW-PRAMs, sie gelten, zum Teil auch für z.B. ARBITRARY-CRCW-PRAMs.

Wie bereits in Abschnitt 1.1 erwähnt, besteht der Vorteil der PRAM darin, daß sie eine sehr mächtige Kommunikationsmethode benutzt, so daß Algorithmen einfach beschrieben werden können. Man sagt auch häufig, daß man beim Entwurf von PRAM-Algorithmen nur auf die „reine“ Parallelität zu achten habe, ohne sich dabei allzu große Gedanken zur Kommunikation machen zu müssen. Allerdings ist eine PRAM technisch nicht realisierbar. Technisch einfacher realisierbar (und zum Teil schon realisiert) sind **Parallelrechner mit verteiltem Speicher**, *distributed memory machines*, kurz **DMM**s.

Eine DMM der Größe N besteht aus N Prozessoren $Q_0, \ldots, Q_{N-1}$ und N Speichermodulen $M_0, \ldots, M_{N-1}$. Ein Speichermodul M_i besteht aus q Registerzellen und einem Kommunikationsfenster KF_i.

Die Prozessoren sind mit den Modulen durch ein Netzwerk verbunden.



Jeder Prozessor kann auf jede Zelle eines jeden Moduls zugreifen, **aber:** die Anfragen an ein Modul werden sequentialisiert.

Falls das Netzwerk ein Butterfly-Netzwerk oder Multibutterfly-Netzwerk ist, reden wir von **Butterfly-DMM** oder **Multibutterfly-DMM**. Falls es vollständig ist, d. h. (vollständig bipartit Prozessoren und Module verbindet), müssen wir wie bei PRAMs Kollisionsregeln definieren.

ARBITRARY**-DMM:** Falls mindestens eine Anfrage (Lese(x), Schreibe z nach x) an M_i gestellt wird, wird irgendeine beantwortet.

***c*-Collision-DMM:** Falls höchstens c Anfragen an M_i gestellt werden, werden alle beantwortet, falls mehr als c Anfragen gestellt werden, wird keine Anfrage beantwortet.

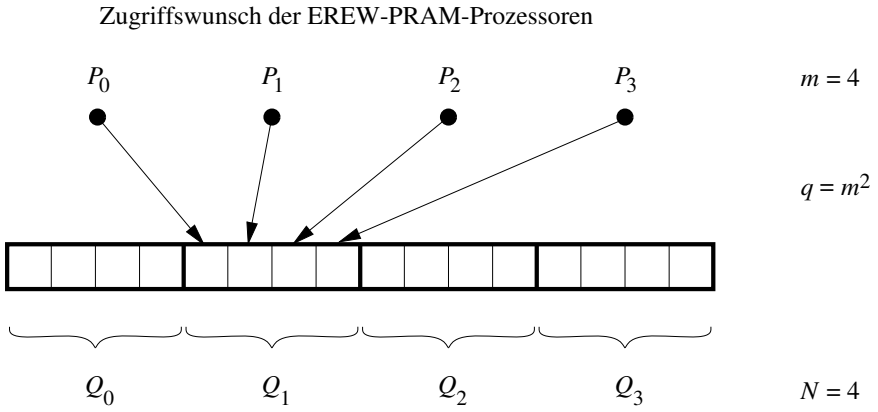
Motivation für die 1-Collision-Regel. Zur Zeit gibt es Forschung und Entwicklung auf dem Gebiet der *Optical Crossbars* (OC). Ein OC realisiert Kommunikation zwischen Q_i und M_j, indem Q_i einen Laserstrahl auf den Rezeptor ($= KF_j$) von M_j richtet, über den die Kommunikation läuft. Falls nun mehrere Q_i ihren Strahl auf KF_j richten, überlagern sich die

Lichtstrahlen und die Anfragen werden unlesbar für KF_j. Somit kann M_j keine der Anfragen verstehen, also auch keine beantworten.

Die Grundidee der in diesem Kapitelbeschriebenen Simulationen von PRAMs durch DMMs besteht darin, den gemeinsamen Speicher der PRAM geschickt auf die Module der DMM zu verteilen. Dabei können mehrere Kopien einer PRAM-Speicherzelle angelegt werden, d. h. diese Zelle wird in mehreren Modulen verwaltet.

Welche Probleme können auftauchen, wenn wir eine PRAM aus m Prozessoren auf einer DMM mit N Prozessoren simulieren wollen? Wir gehen zuerst davon aus, daß jede Zelle nur in einem Modul verwaltet wird.

Da die Zellen des gemeinsamen Speichers auf die Speichermodule der DMM aufgeteilt werden müssen, kann es passieren, daß alle PRAM-Prozessoren auf Zellen zugreifen wollen, die im gleichen Modul verwaltet werden, ein Zugriff dauert dann m Zeiteinheiten. Ist die Aufteilung des gemeinsamen Speichers auf die Module bekannt, so ist es sogar recht einfach, ein PRAM-Programm zu schreiben, bei dem die Simulation **jedes** PRAM-Schrittes $\Theta(m)$ Schritte auf der DMM benötigt. Aber diesen Zeitverlust weist sogar eine sequentielle PRAM-Simulation durch nur einen einzelnen Prozessor auf. Eine solche Situation zeigt das folgende Bild:



Man verfolgt bei der Simulation von PRAMs zwei Ziele:

- Den Entwurf von Simulationsalgorithmen mit möglichst geringem Zeitverlust.
- Den Entwurf von Zeit-Prozessor-effizienten, schnellen Simulationen, d. h. Simulationsalgorithmen, bei denen die Simulation einer PRAM aus m Prozessoren auf einer DMM der Größe N einen Zeitverlust möglichst nah bei $\frac{m}{N}$ hat.

Tabelle 8.1 faßt die erzielten Resultate zusammen.

8.1 Simulation einer CRCW-PRAM mittels einer EREW-PRAM

Wir werden uns bei den folgenden Simulationen von PRAMs mittels DMMs auf EREW-PRAMs beschränken. In diesem Abschnitt wollen wir zeigen, wie man mit einer EREW-PRAM eine CRCW-PRAM simulieren kann, hierbei werden Lese- und Schreibzugriffe geschickt koordiniert, so daß keine Zugriffskonflikte auftreten.

Tabelle 8.1: Resultate bei PRAM-Simulationen

	N: Größe der DMM				
	PRAM-Größe	DMM	Zeitverlust	Bemerkung	Literatur
1.	N	ARBITRARY	$O\left(\frac{\log N}{\log\log N}\right)$	probabilistisch	[DM90]
2.	N	ARBITRARY	$O(\log N)$	deterministisch	[AHMP87], 8.3
3.	$N \log N$	ARBITRARY	$O(\log N)$	probabilistisch (Zeit-Prozessor-optimal)	[Val90] [DM90]
4.	N	c-COLLISION	$O(\log\log N)$	probabilistisch	[DM93], 8.4
5.	$N \log\log N \log^* N$	ARBITRARY	$O(\log\log N \log^* N)$	probabilistisch (Zeit-Prozessor-optimal)	[KLM92]
6.	N	Multibutterfly	$O((\log N)^2)$	deterministisch	aus 2. und Satz 7.10, 8.3
7.	N	Butterfly	$O(\log N)$	probabilistisch	[KU88], 8.2
8.	$N(\log N)^2$	Butterfly	$O((\log N)^2)$	probabilistisch	aus 7., 8.2
9.	N	ARBITRARY	$O\left(\frac{\log\log N}{\log\log\log N}\right)$	probabilistisch	[MSS94]
10.	N	ARBITRARY	$O(\log\log\log N \log^* N)$	probabilistisch	[CMS95]

8.1 Satz:
Eine m Prozessor CRCW-PRIORITY-PRAM mit q Speicherzellen kann mittels einer m Prozessor EREW-PRAM mit $q \cdot m$ Speicherzellen mit Zeitverlust $O(\log m)$ simuliert werden.

Beweis:
Wir zeigen, wie ein Schritt der CRCW-PRAM simuliert wird:

- Jeder Prozessor P_i der CRCW-PRAM wird durch den Prozessor P'_i der EREW-PRAM simuliert.
- Jede Registerzelle R_i, $i = 0, \ldots, q-1$ der CRCW-PRAM wird durch ein Array von m Registerzellen $R'_{i,j}$, $j = 0, \ldots, m-1$ der EREW-PRAM simuliert. In $R'_{i,0}$ wird der Wert von R_i gespeichert, die anderen Zellen sind Hilfszellen und dienen der Zugriffskontrolle. Diese Zellen bilden die inneren Knoten eines vollständigen, binären Baumes, dessen Blätter die Prozessoren bilder, und enthalten zu Beginn der Simulation alle den Wert *frei*.
- Simulation eines Schreib-Schrittes:
  Jede Prozessor muss herausfinden, ob er in der Gruppe der Prozessoren, die auf die

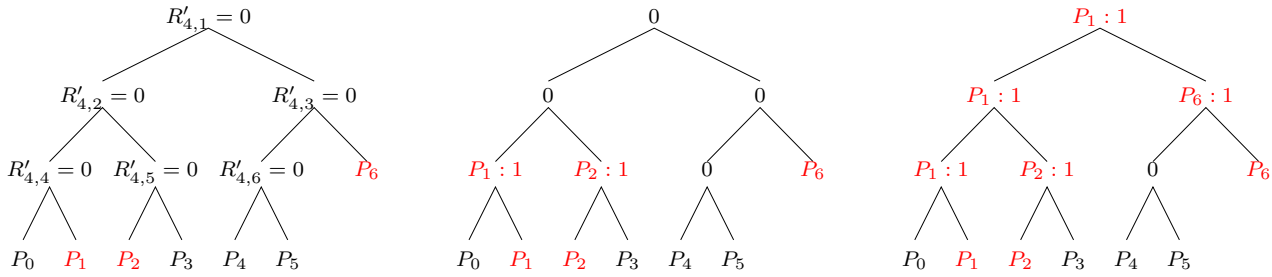


Abbildung 8.1: Beispiel für einen Schreibzugriff der Prozessoren P_1, P_2 und P_6 auf R_4

gleiche Registerzelle schreiben wollen, derjenige mit kleinster Id ist, d.h. ob er nach der PRIORITY-Regel schreiben darf. Dies wird mit folgendem Algorithmus bestimmt:

1. Falls P_k auf R_i schreiben möchte, wird P'_k im zur Speicherzelle i gehörigen Baum T_i aktiv (als k-tes Blatt). Der Baum ist bekannt, d. h. insbesondere ist bekannt, ob P'_k ein linker oder rechter Nachfahre seines Vaters ist.
2. Wiederhole $\log m$ mal
   (a) Jeder aktive linke Prozessor speichert seine Id in die Registerzelle des Vaters im Baum und markiert diese damit als belegt.
   (b) Jeder aktive rechte Prozessor liest die Registerzelle des Vaters im Baum. Ist diese als belegt gekennzeichnet, gibt es einen Prozessor weiter links, d. h. mit kleinerer Id, der die selbe Registerzelle schreiben möchte. Daher wird der rechte Prozessor inaktiv. Falls die Vaterzelle nicht belegt war, bleibt der Prozessor aktiv und ist nun für diesen Knoten im Baum zuständig.
3. Der im Wurzelknoten aktive Prozessor schreibt die Daten nach $R'_{i,0}$
4. Alle an T_i beteiligten Prozessoren setzen alle von ihnen als belegt gekennzeichneten Registerzellen wieder auf *frei*.

Ein Beispiel für einen Schreibzugriff zeigt Abbildung 8.1. Die Prozessoren P_1, P_2 und P_6 wollen in R_4 schreiben. Mit dem oben genannten Algorithmus wird festgelegt, daß Prozessor P_1 die höchste Priorität hat und die Registerzelle R_4 – simuliert in $R'_{4,0}$ beschreiben darf.

- Simulation eines Lese-Schrittes: Ähnlich eines Schreib-Schrittes
  1. Analog zum Schreiben wird bestimmt, welcher Prozessor der „Gewinner“ ist
  2. Der Gewinner liest die Registerzelle $R_{i,0}$
  3. Während des Aufräumens wird der gelesene Wert an alle beteiligten Prozessoren verteilt.

□

Man kann den Speicherverbrauch von $O(m \cdot p)$ auf $O(m + p)$ reduzieren, falls man paralleles Sortieren benutzt:

8.2 Satz:
Eine p Prozessor CRCW-PRIORITY-PRAM mit m Speicherzellen kann mittels einer p Prozessor EREW-PRAM mit $m + p$ Speicherzellen mit Zeitverlust $O(\log p)$ simuliert werden.

Beweis:
Wir beschreiben im folgenden, wie wir einen Lese- oder Schreibschritt der CRCW-PRAM simulieren:

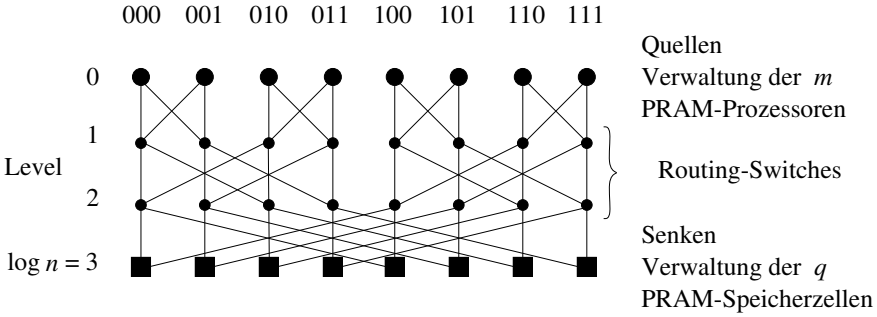
- Jeder Prozessor P_i der CRCW-PRAM wird durch den Prozessor P'_i der EREW-PRAM simuliert.
- Jede Registerzelle R_i der CRCW-PRAM wird durch eine Registerzelle R'_i der EREW-PRAM simuliert. Zusätzlich benutzen wir ein Hilfsarray A_j, $j = 1, \ldots, p$ um Zugriffe zu koordinieren.
- Wenn P_i auf R_j zugreifen will, schreibt er (j, i) in das Hilfsarray an die Position A_i, falls ein Prozessor keinen Zugriff macht, schreibt er $(0, i)$ an A_i.
- Alle Prozessoren sortieren das Hilfsarray in Zeit $O(\log p)$ mittels eines parallelen Sortierverfahrens (z. B. ColeSort).
- Jeder Prozessor P_i fügt in A_i ein Flag f an, mit $f = 0$, falls die erste Komponente von A_i gleich 0 (Prozessor möchte keinen Zugriff) oder gleich der ersten Komponente von $Ai - 1$ (Prozessor hat nicht die kleinste Id für diesen Zugriff) oder $f = 1$ sonst.
- Schreib-Schritt:
  1. Jeder Prozessor P_k liest (i, j, f) aus A_k und schreibt den Wert nach A_j.
  2. Jeder Prozessor P_k liest (i, k, f) aus A_k und schreibt nach R_i, falls $f = 1$ ist
- Lese-Schritt
  1. Jeder Prozessor P_k liest (i, j, f) aus A_k.
  2. Falls $f = 1$, liest P_k den Wert aus R_i und schreibt in die Stelle der 3. Komponente in A_k
  3. In $\log p$ Schritten wird dieser Wert nun auf die folgenden Elemente von A verteilt, bis entweder das Ende von A erreicht ist, oder sich das Element in A in der ersten Komponente unterscheidet (Zugriff auf eine andere Speicherzelle).
  4. Jeder Prozessor P_k liest (i, j, f) aus A_k und schreibt den Wert nach A_j.
  5. Jeder Prozessor P_k, der R_i lesen wollte, liest den Wert nun aus der 3. Komponente von A_k

□

8.2 Probabilistische PRAM-Simulation auf der Butterfly-DMM

In diesem Abschnitt werden wir eine EREW-PRAM mit $m = p \cdot N$ Prozessoren, $P_{i,j}$, $i \in [N]$, $j \in [p]$ auf einer Butterfly-DMM mit N Prozessoren, simulieren.
Sei $[q]$ der Adreßraum des gemeinsamen Speichers der PRAM.
Wir bilden die PRAM folgendermaßen auf die DMM ab:

- Die PRAM-Prozessoren werden gleichmäßig auf die DMM-Prozessoren verteilt. Q_i übernimmt damit die Simulation von $P_{i,0}, \ldots, P_{i,p-1}$.
- Für die Speicherverwaltung benutzen wir eine Hash-Funktion $h : [q] \to [N]$. Speicherzelle $x \in [q]$ wird in Modul $M_{h(x)}$ verwaltet.



Der Simulationsalgorithmus sieht dann so aus:

- Simuliere die interne Berechnung.
- Die **Lesephase** des PRAM-Schritts wird wie folgt simuliert: Falls für $i \in [N]$ und $s \in [p]$ der Prozessor $P_{i,s}$ aus der Zelle $u_s^{(i)}$ lesen will, so führen wir ein Greedy Routing gemäß der Funktion $f \in F_p$ (vgl. Abschnitt 6.2) mit $f(i,s) := h(u_s^{(i)})$ zwischen den Prozessoren und Modulen der DMM durch. Wir schicken zu der Senke ein leeres Paket. Dieses wird dort mit dem Inhalt von Zelle $u_s^{(i)}$ gefüllt und zu Q_i zurückgeschickt.
- Die **Schreibphase** des PRAM-Schritts wird nun wie folgt simuliert: Will für $i \in [N]$ und $s \in [p]$ der Prozessor $P_{i,s}$ etwas in die Zelle $v_s^{(i)}$ schreiben, so wird gemäß der gleichen Funktion $f \in F_p$ wie oben geroutet, da in $M_{h(v_s^{(i)})}$ die Speicherzelle $v_s^{(i)}$ verwaltet wird. Diesmal enthält das Paket den Wert, der geschrieben werden soll. Das Modul $M_{h(v_s^{(i)})}$ führt nach Erhalt des Paketes damit die Aktualisierung des Inhalts von $v_s^{(i)}$ durch.

Man beachte, wie ähnlich dieses Vorgehen zu dem in Kapitel 6.4 untersuchten ist. Die Kosten der Simulation eines PRAM-Schrittes bestehen jetzt darin,

- die interne Berechnung auszuführen,
- $h(u_s^{(i)})$ und $h(v_s^{(i)})$ zu berechnen, und
- $f \in F_p$ zweimal „vorwärts“ und einmal „rückwärts“ zu routen.

Wie wir in Abschnitt 6.4 gesehen haben, sollte, damit das Routing schnell vonstatten geht, die Funktion $f \in F_p$ „zufällig gewählt“ sein. Dieses ist sicherlich erfüllt, wenn h eine zufällig gewählte Funktion aus $\mathcal{A} = \{g \mid g : [q] \to [N]\}$ ist. Ein zufälliges $g \in A$ hat aber die Eigenschaft, daß die platzeffizienteste Methode, sie abzuspeichern, darin besteht, eine Funktionstabelle aufzustellen. D. h., daß ihre Erzeugung $\Theta(q)$ Zeit und $\Theta(q)$ Platz benötigt. Also hätte jede einzelne Quelle so viel Speicher zu verwalten, wie die gesamte PRAM zur Verfügung hat, und das Preprocessing wird indiskutabel viel Zeit benötigen.

Eine Klasse von Funktionen, die sich „zufällig“ genug verhalten, ohne dabei zu viele Ressourcen zur Erzeugung und zum Abspeichern zu benötigen, werden wir im folgenden Abschnitt kennenlernen.

8.2.1 Exkurs: Universelles Hashing

Wir benötigen für das Routing eine Klasse von Funktionen, die sich zumindest dann wie die Klasse $\mathcal{A}$ verhält, wenn wir uns nur wenige Eingaben anschauen. Die folgende Definition formalisiert diesen Wunsch[CW79].

8.3 Definition: (Universelle Hash-Funktionen)
Sei $\mathcal{H} \subseteq \mathcal{A} = \{g \mid g : [q] \to [n]\}$, $k \in I\!N$. $\mathcal{H}$ ist eine ***k*-universelle** Klasse von (Hash-)Funktionen, falls für alle $\ell \leq k$, alle paarweise verschiedenen $x_1, \ldots, x_\ell \in [q]$ und alle $j_1, \ldots, j_\ell \in [n]$ gilt:

$$\text{Prob}(h(x_i) = j_i \text{ für alle } i \in \{1, \ldots, \ell\}) \leq (\tfrac{2}{n})^\ell$$

Die Wahrscheinlichkeit wird gemessen relativ zur zufälligen Wahl von $h \in \mathcal{H}$.

Ist h eine beliebige zufällige Funktion aus $\mathcal{A}$, dann ist die obige Wahrscheinlichkeit $(\frac{1}{n})^\ell$. D. h. wenn wir nur $\ell \leq k$ viele Eingaben betrachten, verhält sich eine zufällig aus $\mathcal{H}$ gewählte Funktion h fast wie eine „richtige“ zufällige Funktion.
Im Rest dieses Abschnitts weisen wir jetzt nach, daß spezielle Klassen von Polynomen universelle Hash-Funktionen sind. Sei dazu q eine fest gewählte Primzahl. Warum q eine Primzahl sein soll, wird die nachfolgende Analyse zeigen.

8.4 Definition:
Sei $d \in I\!N$. Für $\bar{a} = (a_d, \ldots, a_0) \in [q]^{d+1}$ sei

$$h_{\bar{a}}(x) := \left(\Big(\sum_{i=0}^{d} a_i \cdot x^i \Big) \bmod q \right) \bmod n, \text{ für } x \in [q].$$

$\bar{a}$ ist das Koeffizienten-Tupel. Sei $\mathcal{H}_n^d := \{h_{\bar{a}} : [q] \to [n] \mid \bar{a} \in [q]^{d+1}\}$.

$\mathcal{H}_n^d$ besteht also aus Polynomen vom Grad d im Körper $\mathbb{Z}_q$, bei denen die Bilder noch auf $[n]$ „zurechtgestutzt“ werden.

8.5 Satz:
$\mathcal{H}_n^d$ hat die folgenden Eigenschaften:

a) *Jede Funktion $h_{\bar{a}} \in \mathcal{H}_n^d$ kann in Zeit $O(d)$ ausgewertet werden.*

b) *$\mathcal{H}_n^d$ ist $(d+1)$-universell.*

Beweis:
a) ist offensichtlich.

Zu b)
Für das Koeffizienten-Tupel $\bar{a} \in [q]^{d+1}$ sei $f_{\bar{a}} : [q] \to [q]$ das Polynom

$$f_{\bar{a}}(x) = \Big(\sum_{i=0}^{d} a_i \cdot x^i \Big) \bmod q.$$

Da q eine Primzahl ist, ist $f_{\bar{a}}$ ein Polynom vom Grad d über dem Körper $\mathbb{Z}_q$. In Körpern ist ein Polynom vom Grad d durch $d+1$ verschiedene Punkte eindeutig bestimmt (Fundamentalsatz der Algebra).

Sei l, $1 \leq l \leq d+1$, beliebig aber fest. Seien $x_1, \ldots, x_l \in [q]$ verschiedene Eingaben für das Polynom, seien $j_1, \ldots, j_l \in [n]$ mögliche Ausgaben „modulo n“, und sei

$$A^l := \{h_{\bar{a}} \in \mathcal{H}_n^d \mid h_{\bar{a}}(x_i) = j_i \text{ für alle } i \in \{1, \ldots, l\}\}.$$

Dann gilt:

$$h_{\bar{a}} \in A^l \iff h_{\bar{a}} = f_{\bar{a}} \bmod n \text{ und } f_{\bar{a}}(x_i) \in \underbrace{\{j_i + \alpha \cdot n \mid \alpha \in [\lceil \tfrac{q}{n} \rceil]\}}_{=:\, B_i} \text{ für alle } i \in \{1, \ldots, l\}$$

Da die $f_{\bar{a}}$ Polynome vom Grad d über dem Körper $\mathbb{Z}_q$ sind, werden sie durch Funktionswerte an $d+1$ unterschiedlichen Stellen eindeutig bestimmt, d.h. die Koeffizienten $\bar{a}$ von $f_{\bar{a}}$ werden eindeutig bestimmt. Wir legen durch die Wahlen der x_i und j_i schon l Punkte fest, um ein Polynom eindeutig zu bestimmen, müssen wir noch an $d-l+1$ unterschiedlichen Stellen Funktionswerte festlegen. Wir können also zu jedem Tupel $(k_1, \ldots, k_l) \in B_1 \times \ldots \times B_l$ mit $f_{\bar{a}}(x_i) = k_i$ noch $[q]^{d-l+1}$ unterschiedliche Funktionswerte bestimmen, da der Wertebereich von $f_{\bar{a}}$ gerade $[q]$ ist und $d-l+1$ Punkte noch zu wählen sind.
Damit folgt für die Anzahl der Hash-Funktionen, die genau auf die gewählten Ziele abbilden: $|A^l| = |B_0| \cdot \ldots \cdot |B_d| \cdot [q]^{d-l+1} \leq \lceil \frac{q}{n} \rceil^l \cdot [q]^{d-l+1}$. Da weiterhin $|\mathcal{H}_n^d| = q^{d+1}$ gilt (denn jede Folge $\bar{a} \in [q]^{d+1}$ definiert ein anderes Polynom über dem Körper $\mathbb{Z}_q$), folgt:

$$\begin{aligned} &\text{Prob}(h_{\bar{a}}(x_i) = j_i \text{ für alle } i \in \{1, \ldots, l\}) \\ = \; &\frac{|A^l|}{q^{d+1}} \leq \frac{\lceil \frac{q}{n} \rceil^l \cdot [q]^{d-l+1}}{q^{d+1}} \leq \left(\frac{2}{n}\right)^l \end{aligned}$$

$\mathcal{H}_n^d$ ist damit $(d+1)$-universell. □

eine

8.2.2 Analyse der PRAM-Simulation durch universelle Hashfunktionen

Man betrachte nun den folgendermaßen abgewandelten Simulationsalgorithmus, wobei $\mathcal{H}$ als k-universell vorausgesetzt ist. Wie üblich, werden wir k erst später spezifizieren.

- Wähle zufällig eine Funktion $h \in \mathcal{H}$ und
- simuliere damit Schritt für Schritt den PRAM-Algorithmus.

8.6 Satz:
Sei $\alpha > 11$ eine Konstante, $k = \alpha(\log N + p)$, und sei $\mathcal{H} \subseteq \{g \mid g : [q] \to [N]\}$ eine k-universelle Klasse von Hash-Funktionen. Jede Funktion $h \in \mathcal{H}$ sei in Zeit T auswertbar. Dann hat die obige Simulation einer PRAM aus $m = p \cdot N$ Prozessoren auf einer Butterfly-DMM mit N Prozessoren folgende Eigenschaften:

a) *Sei $\ell > 0$ beliebig. Für jeden PRAM-Schritt gilt:*

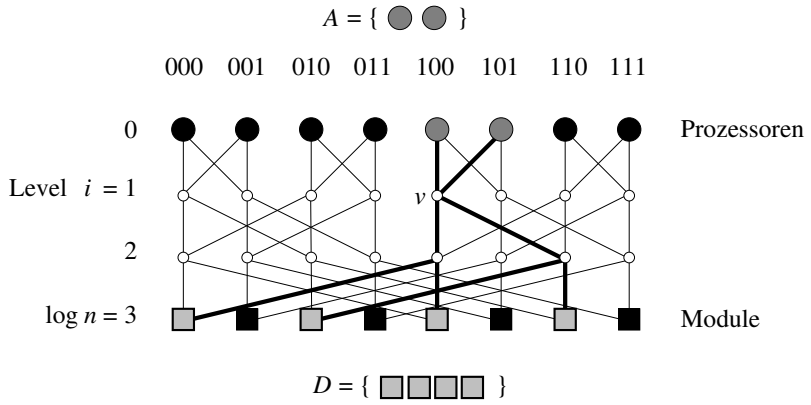
$$\text{Prob}(\textit{Simulation des Schrittes benötigt Zeit höchstens } c \cdot (\log N + p + T \cdot p)) \geq 1 - \frac{2}{N^\ell}$$

für genügend große Konstanten c.

b) Die erwartete Simulationszeit für einen PRAM-Schritt ist $O(\log N + p + Tp)$.

Die Wahrscheinlichkeiten beziehen sich auf die zufällige Wahl von $h \in \mathcal{H}$ und die zufällige Wahl der Ränge im Random Rank-Protokoll.

Beweis:
Zu a)
Wir müssen den Algorithmus für das Greedy Routing aus Abschnitt 6.2 unter der Annahme analysieren, daß $f \in F_p$ nicht zufällig gewählt ist, sondern mit Hilfe von h definiert wurde, d. h. daß das Paket $u_s^{(i)}$ das Ziel $h(u_s^{(i)})$ hat. Dazu reicht es, die Analyse für die Congestion aus Satz 6.12 (S. 61) nachzuvollziehen.
Sei v wiederum ein Knoten in BF($\log N$) (welches Prozessoren und Module verbindet) auf Level i mit $i > 0$, und sei $P_r(v)$ nochmals die Wahrscheinlichkeit, daß beim Routing mindestens r Pakete durch Knoten v gehen. Diese Wahrscheinlichkeit ist relativ zu den festen Paketen $u_s^{(j)}$, aber einer zufällig gewählten Funktion $h \in \mathcal{H}$ definiert.
Wir bestimmen nun $P_r(v)$ für $r \leq k$. Dazu sei A die Menge der Pakete, die von ihren Quellen v erreichen können. Es ist $|A| = p \cdot 2^i$. D sei die Menge der Ziele, die von Quellen aus über Wege durch v erreichbar sind, also $|D| = 2^{\log N - i} = \frac{N}{2^i}$.



Sei $L = \{u_1, \ldots, u_r\}$ mit $L \subseteq A$, $r \leq k$ fest. Dann ist

$$\begin{aligned}
&\text{Prob(Die Pakete } u_1, \ldots, u_r \text{ laufen über } v) \\
&= \text{Prob(Die Pakete } u_1, \ldots, u_r \text{ haben Ziele in } D) \\
&= \text{Prob}(\exists j_1, \ldots, j_r \in D : h(u_i) = j_i \text{ für alle } i \in \{1, \ldots, r\}) \\
&\leq \sum_{j_1, \ldots, j_r \in D} \text{Prob}(h(u_i) = j_i \text{ für alle } i \in \{1, \ldots, r\}) \\
&\leq |D|^r \cdot \left(\frac{2}{N}\right)^r \qquad (r \leq k, \text{ und } \mathcal{H} \text{ ist } k\text{-universell}) \\
&= \left(\frac{N}{2^i}\right)^r \cdot \left(\frac{2}{N}\right)^r \leq \left(\frac{2}{2^i}\right)^r .
\end{aligned}$$

Analog zu Abschnitt 6.2 folgt:

$$P_r(v) \leq \binom{p \cdot 2^i}{r} \cdot \left(\frac{2}{2^i}\right)^r \leq \left(\frac{2ep}{r}\right)^r$$

Wir wählen r wieder so, daß die gesuchte Wahrscheinlichkeit höchstens $\frac{1}{N^\ell}$ ist. Somit ist dann für $r = 4e \cdot p + (\ell+1)\log N + \text{loglog}\, N$:

$$\begin{aligned}&\text{Prob(Es gibt einen Knoten } v\text{, über den mindestens } r \text{ Pakete gehen)}\\ &\leq \quad N \log N \cdot \left(\frac{2ep}{r}\right)^r \leq \frac{1}{N^\ell}\end{aligned}$$

Also gilt: Falls für $k = 4e \cdot p + (\ell+1)\log N + \text{loglog}\, N$ die Klasse $\mathcal{H}$ k-universell ist, so ist

$$\text{Prob(Über alle Knoten gehen } \textbf{höchstens } k \text{ Pakete)} \geq 1 - \frac{1}{N^\ell} \ .$$

Zusammen mit Satz 6.13 (S. 64) können wir damit h mit Wahrscheinlichkeit $(1-\frac{1}{N^\ell})\cdot(1-\frac{1}{N^\ell}) \geq (1-\frac{2}{N^\ell})$ in Zeit $O(\log N + p)$ routen. Bei der PRAM-Simulation muß jeder Prozessor zusätzlich die Hash-Funktion h für p Werte auswerten, d. h. es kommt zusätzlich die Zeit $p \cdot T$ hinzu.

Zu b)
Wir nehmen nun $\ell = 1$ an. Wie wir in a) gesehen haben, ist die Simulationszeit für einen Schritt für einen Anteil von mindestens $1 - \frac{2}{N}$ aller möglichen Probleme höchstens $O(\log N + p + p \cdot T)$. Sie ist im worst-case, d. h. wenn alle Pakete zum gleichen Ziel geroutet werden sollen, höchstens $\underbrace{p \cdot N}_{=\,m} + \log N + p \cdot T$. Dies kann aber für nicht mehr als einen Anteil von $\frac{2}{N}$ aller Probleme der Fall sein. Damit ergibt sich für den Erwartungswert:

$$\begin{aligned}&E(\text{Laufzeit für die Simulation eines Schrittes})\\ &= \ O\left((1-\tfrac{2}{N})\cdot(\log N + p + p\cdot T) + \tfrac{2}{N}\cdot(p\cdot N + \log N + p\cdot T)\right)\\ &= \ O(\log N + p\cdot T).\end{aligned}$$

□

Zusammen mit der $(d+1)$-universellen Klasse $\mathcal{H}_n^d$ mit Auswertungszeit $O(d)$ aus Abschnitt 8.2.1 ergibt sich damit:

8.7 Satz:
Es gibt eine probabilistische Simulation einer EREW-PRAM aus $m = p \cdot N$ Prozessoren auf der Butterfly-DMM mit N Prozessoren mit erwartetem Zeitverlust $O(p \cdot \log N + p^2)$. Für $p = 1$ ergibt sich also der Zeitverlust $O(\log N)$.

8.8 Bemerkung:
Es gibt $O(\log N + p)$-universelle Hash-Klassen, deren Funktionen sich sogar in Zeit $O(1)$ auswerten lassen[Sie89]. Diese liefern damit den erwarteten Zeitverlust $O(\log N + p)$, also für $p = \log N$ eine Zeit-Prozessor optimale Simulation.

8.3 Deterministische Simulation einer PRAM durch eine ARBITRARY-DMM

In diesem Abschnitt wollen wir auf einer ARBITRARY-DMM M_0 mit N Prozessoren eine EREW-PRAM M gleicher Größe mit $q = N^k$ gemeinsamen Speicherzellen, k eine beliebige Konstante, deterministisch simulieren.

Verwalten wir jede Speicherzelle der PRAM in genau einem Modul, so haben wir, wie wir schon einmal auf Seite 85 bemerkt hatten, im worst-case einen Zeitverlust von N. Diese Zeit wird benötigt, wenn bei der Simulation eines PRAM-Schrittes alle Netzwerk-Prozessoren gleichzeitig auf Zellen zugreifen wollen, die dasselbe Speichermodul verwaltet. Wir benötigen daher eine bessere Idee.

Diese besteht darin, jede Speicherzelle der PRAM nicht nur in einem, sondern in mehreren Modulen zu speichern, d. h. wir benutzen eine Speicherabbildung mit **Redundanz**. Dadurch wird einerseits das Lesen einfacher, da wir dann viele Module zur Auswahl haben, in denen wir nach dem Inhalt der gewünschten Speicherzelle anfragen können. Andererseits wird das Schreiben dadurch aufwendiger, weil wir nun viele Kopien schreiben müssen.

Den Aufwand für das Lesen und Schreiben balancieren wir, indem wir bei einem Schreibschritt nicht alle Kopien auf den neuesten Stand bringen. Deshalb benötigen die Kopien einer Zelle u zusätzlich zum Inhalt der Speicherzelle einen Zeitstempel. Diese Kopien bestehen also aus einem Tupel $\langle$Inhalt von u, Zeitstempel$\rangle$, d. h. dem Inhalt, den die Speicherzelle u zum durch den Zeitstempel gegebenen Zeitpunkt (= Rechenschritt der PRAM) hat. Im Normalfall werden somit nicht alle Kopien **aktuell** sein, d. h. den neuesten (= Rechenschritt, zu dem die Zelle zuletzt beschrieben wurde) Zeitstempel enthalten. Man beachte, daß zwischen der Nummer des aktuellen Schrittes und dem neuesten Zeitstempel einer Speicherzelle eine beliebig große Differenz sein kann. Wir sorgen dafür, daß immer mehr als die Hälfte aller Kopien auf dem neuesten Stand sind. Genauer sieht das folgendermaßen aus:

Sei $U = [q]$ der Adreßraum der Speicherzellen der PRAM M. Sei $\Gamma(u)$ die Menge der Module von M_0, die eine Kopie von Zelle $u \in U$ enthalten. Es ist $|\Gamma(u)| = 2c - 1$. c werden wir, wie üblich, erst später geeignet festlegen. Wir untersuchen den folgenden Algorithmus:

Makroalgorithmus der PRAM-Simulation

- **Zu Beginn:**
  Für jedes $u \in U$ enthält jedes Modul $P \in \Gamma(u)$ das Tupel $\langle$Inhalt von u in M zu Beginn, $0\rangle$, d. h. der Zeitstempel steht auf 0.
- Der Prozessor Q_i von M_0 simuliert den Prozessor P_i der PRAM M.
- Es seien $t \geq 0$ Schritte simuliert. Der PRAM-Prozessor P_i will im Schritt $t + 1$ in der Zelle u_i lesen oder den Wert v_i nach Zelle u_i schreiben.
  - **Beim Lesen:**
    Q_i betrachtet die Kopien von Zelle u_i in c beliebigen Modulen aus $\Gamma(u_i)$ und liest denjenigen Wert ein, dessen Zeitstempel am größten ist.
  - **Beim Schreiben:**
    Q_i ersetzt den Inhalt der Kopie von Zelle u_i in c beliebigen Modulen aus $\Gamma(u_i)$ durch das Tupel $\langle v_i, t + 1\rangle$.

Zwei Dinge bleiben in diesem Makroalgorithmus offen:

a) Die Verteilung der Kopien auf die Module und

b) die Durchführung der Lese- und Schreibzugriffe, da es ja nicht ausgeschlossen ist, daß es beim Zugriff auf einen Modul zu Kollisionen kommen kann.

In den beiden nachfolgenden Unterabschnitten werden wir diese beiden Punkte untersuchen und zeigen, daß der Makroalgorithmus eine effiziente PRAM-Simulation liefert. Zu Beginn zeigen wir erst einmal die Korrektheit des Verfahrens.

8.9 Lemma:
Falls M_0 nach dem obigen Makroalgorithmus vorgeht, simuliert M_0 die PRAM M.

Beweis:
Behauptung: Zu jedem Zeitpunkt t (der PRAM) sind für jede Zelle $u \in U$ mindestens c der $2c-1$ Kopien von u aus $\Gamma(u)$ aktuell, d. h. sie enthalten den Wert, den u zur Zeit t enthalten hat, und den Zeitstempel, der den Zeitpunkt des letzten Schreibzugriffs auf u angibt.

Beweis der Behauptung:
Wir führen eine Induktion nach t:

Für $t = 0$ braucht nichts gezeigt zu werden.

Sei nun $t > 0$: Zur Zeit $t-1$ waren von jeder Zelle u mindestens c Kopien aktuell. Eine Kopie von u kann nur dann seine Aktualität verlieren, wenn u durch M neu beschreiben wird. In diesem Fall werden aber im Makroalgorithmus c Kopien von u aktualisiert, also sind auch nach der Simulation von Schritt t von den Kopien c aktuell. □ (Behauptung)

Behauptung ⇒ Lemma 8.9:
Q_i liest c Kopien von u, es gibt insgesamt nur $2c-1$ Kopien, wovon c Kopien aktuell mit Zeitstempel vom letzten Schreibzugriff sind. Also ist mindestens eine der gelesenen Kopien eine aktuelle. □ (Lemma 8.9)

8.3.1 Die Speicherverwaltung

Das erste Problem, das sich einstellt, wenn wir die Simulation schnell machen wollen, ist das folgende: Wie haben die Modulmengen $\Gamma(u)$ auszusehen? M_0 darf nämlich nicht „zu lange“ herumsuchen müssen, um die c Kopien schließlich lesen zu können. Das Problem läßt sich leicht in graphentheoretischem Kontext formulieren. Antwort auf die Frage gibt dann das folgende Lemma:

8.10 Lemma:
Sei $b = \max\{(2e)^4 \cdot N^{2k/c}, 4\}$, $c \leq \log N$. Es gibt einen bipartiten Graphen $G = (\mathcal{U} \cup \mathcal{P}, E)$, $|\mathcal{U}| = N^k$, $|\mathcal{P}| = N$, mit den Eigenschaften:

a) Jeder Knoten $u \in \mathcal{U}$ hat Grad $2c-1$.

b) Wie üblich bezeichne $\Gamma(u)$ die Menge der Nachbarn von $u \in \mathcal{U}$ in G.
Zu jedem u sei $\Gamma'(u) \subseteq \Gamma(u)$ eine beliebige Teilmenge von $\Gamma(u)$ mit $|\Gamma'(u)| \geq c$. Es gilt: Jede Menge $A \subseteq \mathcal{U}$ mit $|A| \leq \frac{N}{2c-1}$ erfüllt die Bedingung:

$$|\Gamma'(A)| = \Big|\bigcup_{u \in A} \Gamma'(u)\Big| \geq \frac{2c-1}{b} \cdot |A| \qquad (*)$$

Die Existenz dieser bipartiten Graphen, die wieder zur Familie der Expander gehören, kann wie die der Konzentratoren in Satz 7.2 technisch relativ einfach durch ein probabilistisches

Existenz-Argument nachgewiesen werden. Explizite Konstruktionen sind in diesem Fall bisher nicht bekannt.

$\mathcal{U}$ entspricht natürlich der Menge der Speicherzellen und $\mathcal{P}$ der Menge der Netzwerk-Prozessoren. Die Mengen $\Gamma(u)$ sind die Zuordnung der Kopien der Speicherzelle u auf die Prozessoren des Netzwerks. Die Mengen $\Gamma'(A)$ entsprechen Mengen von Kopien, auf die trotz Zugriffswunsch während des Makroalgorithmus noch nicht zugegriffen worden ist. Die Bedingung $(*)$ sagt aus, daß es, so lange A nicht „sehr groß“ ist, noch immer „viele verschiedene“ Modulen gibt, in denen gewünschte Kopien zu finden sind. Der Faktor $\frac{2c-1}{b}$ ist wiederum ein Expansionsfaktor (vgl. die Definition 7.1 (iii) der Konzentratoren).

Wir gehen im Rest dieses Kapitels davon aus, daß die Modulmengen $\Gamma(u)$, die die Kopien von Speicherzelle u enthalten, wie in Lemma 8.10 festgelegt sind.

8.3.2 Organisation von Lese- und Schreibphase

Nachdem wir im vorhergehenden Unterabschnitt die Zuordnung Γ der Kopien der Speicherzellen festgelegt haben, zeigen wir nun, daß es damit möglich ist, einen PRAM-Schritt schnell zu simulieren. Dazu müssen wir jetzt das zweite Problem lösen, nämlich zeigen, daß es möglich ist, die Speicherzugriffe garantiert schnell zu realisieren.

Wir gehen also davon aus, daß vor dem zu simulierenden Schritt mindestens c der $2c-1$ Kopien aller Zellen u in $\Gamma(u)$ aktuell sind. Jetzt will für $i \in [n]$ der PRAM-Prozessor P_i den Inhalt von Zelle u_i lesen bzw. den Wert v_i in die Zelle u_i schreiben. Dies soll von Prozessor Q_i gemäß dem Makroalgorithmus simuliert werden.

Um die in Lemma 8.10 aufgeführte Expansionseigenschaft nutzen zu können, simulieren wir die Speicherzugriffe der N PRAM-Prozessoren nicht alle gleichzeitig, sondern wir führen in $2c-1$ vielen Phasen jeweils $\ell := \frac{N}{2c-1}$ der Speicherzugriffe durch.

Wir sagen, der DMM-Prozessor „Q_i ist fertig“, falls er c Kopien von u_i gelesen hat.

$c_P(u_i)$

Wir betrachten jetzt die folgende τ-te Phase der Simulation eines PRAM-Schrittes, in der die Schreib- und Lesewünsche der DMM-Prozessoren $Q_{\tau\cdot\ell}, \ldots, Q_{(\tau+1)\cdot\ell-1}$ erfüllt werden.

Die τ-te Phase (für $Q_{\tau\cdot\ell}, \ldots, Q_{(\tau+1)\cdot\ell-1}$)

`while` $\exists i \in \{\tau\cdot\ell, \ldots, (\tau+1)\cdot\ell-1\}$ mit „Q_i nicht fertig“ `do`

- Für jeden nicht-fertigen Prozessor Q_i sei $\Gamma'(u_i) \subseteq \Gamma(u_i)$ die Menge der Module, die Kopien von u_i enthalten, und auf die Q_i bislang nicht mit Erfolg zugegriffen hat.
- Jeder nicht-fertige Prozessor Q_i versucht, nacheinander in den Modulen aus $\Gamma'(u_i)$ jeweils Kopien von u_i zu lesen bzw. mit $\langle v_i, t\rangle$ zu überschreiben. Jedes Modul beantwortet eine der Anfragen (falls er mindestens eine bekommt. (Dies ist die ARBITRARY-Regel.)
- Falls Q_i insgesamt auf c Kopien von u_i erfolgreich zugegriffen hat, ist Q_i fertig. Falls Q_i die Lesephase von P_i simuliert, übernimmt Q_i denjenigen der c gelesenen Werte, der den höchsten Zeitstempel trägt.

`done`

Die Dauer eines Durchlaufs der `while`-Schleife ist offensichtlich $O(c)$.

Führen wir die obige Simulationsphase für alle $\tau \in [2c-1]$ durch, d. h. für die Prozessoren $\{P_0, \ldots, P_{\ell-1}\}, \{P_\ell, \ldots, P_{2\ell-1}\}, \ldots$, so haben wir einen Schritt aller PRAM-Prozessoren simuliert.

Wir müssen jetzt zeigen, daß diese Phasen korrekt arbeiten, d. h. halten und die richtigen Werte gelesen bzw. geschrieben haben.

Unter der Maßgabe, daß alle Phasen schließlich halten, ist es wie bei Lemma 8.9 einfach, die Korrektheit der Speicherzugriffe zu beweisen:

8.11 Lemma:
Falls der Algorithmus hält, simuliert er einen Schritt von M.

Beweis:
Wenn der Algorithmus hält, sind alle Q_i fertig, d. h. jeder Prozessor Q_i hat auf c Kopien von u_i zugegriffen und (beim Schreiben) alle aktualisiert bzw. (beim Lesen) den Wert mit dem neuesten Zeitstempel gelesen. Nach Lemma 8.9 ist damit ein Schritt simuliert. □

Jetzt geht es darum zu zeigen, wie oft die `while`-Schleife höchstens durchlaufen werden muß.

8.12 Lemma:
Jede Phase hält nach maximal $\frac{\log N}{\log((1-\frac{1}{b})^{-1})} + 1$ *Durchläufen der* `while`*-Schleife.*

Beweis:
Wir nennen eine Kopie einer Zelle u_i, $i \in \{\tau\ell, \ldots, (\tau+1)\ell - 1\}$, lebendig nach s Durchläufen der `while`-Schleife, wenn der Prozessor Q_i noch nicht fertig ist und auf diese Kopie bisher nicht zugegriffen hat.

Wir zeigen zuerst die folgende Aussage:

Behauptung:
Nach s Durchläufen der `while`-Schleife gibt es noch maximal $(1-\frac{1}{b})^s \cdot N$ viele lebendige Kopien.

Beweis der Behauptung:
Diesen Beweis führen wir durch Induktion nach s:

Sei $s = 0$:
Die $\ell = \frac{N}{2c-1}$ vielen Zellen u_i haben je $2c-1$ Kopien, also zusammen höchstens $\ell \cdot (2c-1) = N = (1-\frac{1}{b})^0 \cdot N$ viele Kopien.

Sei nun $s > 0$:
Nach $s-1$ Durchläufen gebe es h nicht-fertige Prozessoren Q_i. Jeder besitzt noch eine Menge $\Gamma'(u_i)$ von mindestens c Modulen, die Kopien von u_i enthalten, auf die bislang noch nicht zugegriffen worden ist.

Nach Lemma 8.10 sind diese auf mindestens $h \cdot \frac{2c-1}{b}$ verschiedene Module verteilt, d. h. mindestens in jedem dieser Module wird in dem s-ten Durchgang auf eine lebendige Kopie zugegriffen. Das bedeutet, daß mindestens $h \cdot \frac{2c-1}{b}$ Zugriffe auf lebendige Kopien erfolgreich sind.

Da die Gesamtzahl der lebendigen Kopien nach s Durchläufen höchstens $h \cdot (2c-1)$ ist, wird auf einen Anteil von mindestens $\frac{1}{b}$ der lebendigen Kopien zugegriffen, d. h. nur ein Anteil von höchstens $1-\frac{1}{b}$ der vor Beginn des Durchlaufs lebendigen Kopien bleibt lebendig.

Nach Induktionsvoraussetzung waren vor dem s-ten Durchlauf weniger als $(1-\frac{1}{b})^{s-1} \cdot N$ Kopien lebendig. Wir verringern diese Zahl um den Faktor $1 - \frac{1}{b}$, es bleiben also nach s Durchläufen höchstens $(1 - \frac{1}{b})^s \cdot N$ viele Kopien lebendig. □ (Behauptung)

Behauptung ⇒ Lemma 8.12:
Der Algorithmus stoppt, sobald die Anzahl lebendiger Kopien kleiner als 1, also 0 ist. Wir müssen somit $\min\{s \mid (1 - \frac{1}{b})^s \cdot N < 1\}$ bestimmen.

$$\begin{aligned} & (1 - \tfrac{1}{b})^s \cdot N &< 1 \\ \iff & \left((1 - \tfrac{1}{b})^{-1}\right)^s &> N \\ \iff & s &> \frac{\log N}{\log((1 - \frac{1}{b})^{-1})} \end{aligned}$$

Darum reichen

$$s = \frac{\log N}{\log((1 - \frac{1}{b})^{-1})} + 1$$

Durchläufe aus. □ (Lemma 8.12)

Die „richtige“ Wahl von c liefert jetzt den folgenden Satz:

8.13 Satz:
Ein vollständiges Netzwerk mit N Prozessoren kann eine PRAM mit N Prozessoren und N^k gemeinsamen Speicherzellen mit Zeitverlust $O((\log N)^3)$ simulieren.

Beweis:
Wir benötigen zur Simulation eines Schrittes der PRAM

$$O\left(\underbrace{(2c-1)}_{\substack{\text{Anzahl}\\ \text{der Phasen}}} \cdot \underbrace{c}_{\substack{\text{Schritte je}\\ \text{Schleifendurchlauf}}} \cdot \underbrace{\frac{\log N}{\log((1 - \frac{1}{b})^{-1})}}_{\substack{\text{Anzahl der}\\ \text{Schleifendurchläufe}}}\right)$$

Schritte auf der DMM. Wähle $c = \log N$. Nach Lemma 8.10 ist

$$b = \max\{N^{2k/c} \cdot (2e)^4, 4\} = \max\{2^{2k} \cdot (2e)^4, 4\} = 2^{2k} \cdot (2e)^4$$

eine Konstante größer als 1. Somit ist $(1 - \frac{1}{b})^{-1} > 1$, also $\log((1 - \frac{1}{b})^{-1}) > 0$.
Es ergibt sich ein Zeitverlust von $O((\log N)^3)$. □

8.14 Bemerkung:

a) *In der Originalarbeit von Upfal/Wigderson[UW87] wird durch einen schlaueren Algorithmus ein Zeitverlust von $O(\log N \cdot (\text{loglog } N)^2)$ erreicht. Geschickte algorithmische Tricks[AHMP87] reduzieren den Zeitverlust sogar auf den Wert von $O(\log N)$. Upfal und Wigderson zeigen zusätzlich in ihrer Arbeit eine untere Schranke für den Zeitverlust von $\Omega(\frac{\log N}{\text{loglog } N})$ für Simulationen von PRAMs, bei denen nicht vorab bekannt ist, auf welche Zellen die Prozessoren in jedem Schritt zugreifen wollen.*

b) *Wir können sowohl die Aussagen aus Bemerkung 5.27 (S. 49) über die Existenz von $(N, \log N)$-Simulatoren der Größe N wie auch die Fähigkeit des Multibutterfly-Netzwerks, in Zeit $O(\log N)$ routen zu können, mit der Aussage a) kombinieren. Somit kann die Multibutterfly-DMM jede PRAM der Größe N mit polynomiell in N vielen Speicherzellen mit Zeitverlust $O((\log N)^2)$ simulieren.*

c) *Das hier vorgestellte Verfahren besitzt, ähnlich wie das Routing auf dem Multibutterfly-Netzwerk, gute Fehlertoleranz-Eigenschaften, die zu einem guten Teil auf die benutzten Expander-Graphen zurückgehen.*

d) *Das deterministische Verfahren ist leider in hohem Maße unrealistisch und sehr speicherintensiv (es werden ja $\Theta(\log N)$ Kopien jeder Speicherzelle verwaltet), außerdem nichtuniform, da die Prozessormengen $\Gamma(u)$ nicht durch einfache Regeln beschrieben werden können, sondern explizit abgespeichert werden müssen.*

8.4 Probabilistische Shared Memory Simulationen auf ARBITRARY- und c-COLLISION DMMs

Wir kommen jetzt zurück auf die Methode, den gemeinsamen Speicher der PRAM mittels einer oder mehrerer, zufällig aus einer $O(\log N)$-universellen Klasse gewählten Hashfunktion(en) auf die Module zu verteilen. H bezeichne im folgenden immer eine derartige Klasse von Hashfunktionen $h : U \to [N]$. Wir gehen davon aus, daß jedes $h \in H$ in konstanter Zeit ausgewertet werden kann (vgl. Bemerkung 8.8).

8.4.1 Simulationen, die eine Hash Funktion benutzen.

Wir simulieren zuerst PRAMs auf N-Prozessor ARBITRARY-DMMs.

Wir nehmen an, daß $h \in H$ zufällig gewählt ist, und Zelle x des gemeinsamen Speichers der PRAM in Modul $M_{h(x)}$ verwaltet wird.

Sei $m = N \cdot p$, M eine m-Prozessor PRAM, deren Prozessoren mit $P_{i,j}$, $i = 1, \ldots, N$; $j = 1, \ldots, p$ bezeichnet werden.

Wir wollen auf einer N-Prozessor ARBITRARY-DMM simulieren. Dazu simuliert Prozessor Q_i der DMM die PRAM Prozessoren $P_{i,1}, \ldots, P_{i,p}$. Wir geben einen Algorithmus, an das Standard-Zugriffsschema, welches die Zugriffe auf den gemeinsamen Speicher in der offensichtlichen Weise durch Zugriffe auf die Module ersetzt.

Standard-Zugriffsschema ($P_{i,j}$ will $x_{i,j}$ lesen/beschreiben.)

Jedes Q_i führt folgende Schritte aus.

```
for ℓ := 1 to p do
    status(x_{i,ℓ}) := not ready
    while status(x_{i,ℓ}) not ready do
        stelle Lese-/Schreibanfrage für x_{i,ℓ} an M_{h(x_{i,ℓ})} ;
        if M_{h(x_{i,ℓ})} Anfrage beantwortet/akzeptiert, then status(x_{i,ℓ}) := ready ;   } Runde
    done
done
```

Jedes Modul beantwortet/akzeptiert eine Anfrage pro Runde falls es mindestens eine bekommt. Obiges Schema arbeitet also in der offensichtlichen Weise die Anfragen ab.

- Wie lange dauert es für $p = 1$, d. h. für eine Simulation einer N-Prozessor PRAM auf einer N-Prozessor DMM?
- Für welche p (wenn überhaupt) erhalten wir Zeit-Prozessor-optimale Strategien?

Im folgenden liegt den Wahrscheinlichkeiten und Erwartungswerten immer das Zufallsexperiment „Wähle $h \in H$ zufällig“ zugrunde.

8.15 Satz:
*Mit dem Standard-Zugriffsschema kann eine N-Prozessor EREW-PRAM auf einer N-Prozessor-*ARBITRARY*-DMM mit erwartetem Zeitverlust $O(\log N / \operatorname{loglog} N)$ simuliert werden. Die Zeitschranke ist sehr zuverlässig: zu jedem $\ell > 0$ gibt es ein $c > 0$ mit*

$$\text{Prob}\Big(\textit{Zeitverlust } > c \cdot \frac{\log N}{\operatorname{loglog} N}\Big) \leq \frac{1}{N^\ell} \ .$$

Beweis:
Wir müssen das Standard-Zugriffsschema für $p = 1$ analysieren. Eine Runde kostet konstante Zeit.
Wieviele Runden sind nötig?
Wir schreiben kurz $x_1, \ldots, x_N$ für die N Zellen des gemeinsamen Speichers, auf die zugegriffen werden soll.
Sei $h : U \to [N]$, $S = \{x_1, \ldots, x_N\}$. Wir sagen: h zerlegt S für $i = 1, \ldots, N$ in die **Buckets** $B_i^h = h^{-1}(i) \cap S$ der Größe $b_i^h := |B_i^h|$.

8.16 Lemma:
Für $p = 1$ benötigt das Standard-Zugriffsschema genau $C^h := \max\{b_i^h \mid i = 1, \ldots, N\}$ Runden.

Beweis:
Jedes Modul M_i bekommt insgesamt b_i^h verschiedene Anfragen. Diese beantwortet es in den ersten b_i^h Runden, in jeder Runde eine Anfrage. Somit ist auch das am stärksten, mit C^h Anfragen belastete Modul, nach C^h Runden fertig. □

Folgendes Lemma impliziert somit Satz 8.15.

8.17 Lemma:
Sei $S \subseteq U$, $|S| \leq N$.

a) Sei $i \in \{1, \ldots, N\}$ fest. Für $u = O(\log N)$ ist

$$\text{Prob}(b_i^h \geq u) \leq \binom{N}{u} \cdot \Big(\frac{2}{N}\Big)^u \leq \Big(\frac{2e}{u}\Big)^u \ .$$

b) Für genügend große N und $\ell > 0$ beliebig ist

$$\text{Prob}\Big(C^h \geq 2(\ell+1)\frac{\log N}{\operatorname{loglog} N}\Big) \leq \frac{1}{N^\ell} \ .$$

c) $E(C^h) = O\Big(\frac{\log N}{\operatorname{loglog} N}\Big)$

Beweis:
Zu a)

$$\begin{aligned}
&\operatorname{Prob}(b_i^h \geq u) = \operatorname{Prob}(\exists A \subseteq S,\ |A| = u \forall x \in A : h(x) = i) \\
&\leq \sum_{\substack{A \subseteq S \\ |A|=u}} \operatorname{Prob}(h(x) = i \text{ für alle } x \in A) \\
&\overset{(*)}{\leq} \binom{N}{u} \cdot \Big(\frac{2}{N}\Big)^u \overset{(**)}{\leq} \Big(\frac{Ne}{u}\Big)^u \cdot \Big(\frac{2}{N}\Big)^u = \Big(\frac{2e}{u}\Big)^u \ .
\end{aligned}$$

Die Abschätzung $(*)$ gilt, da H $O(\log N)$-universell ist (vgl. 8.2.1), und $(**)$ gilt wegen Lemma 7.4.

Zu b)

$$\operatorname{Prob}(C^h \geq u) \leq \operatorname{Prob}(\exists i \in \{1, \ldots, N\} \text{ mit } b_i^h \geq u) \leq N \cdot \Big(\frac{2e}{u}\Big)^u$$

Für $u = 2(\ell+1)\frac{\log N}{\operatorname{loglog} N}$ folgt:

$$\operatorname{Prob}(C^h \geq u) \leq N \cdot \left(\frac{2e}{u}\right)^u \overset{(*)}{\leq} N \left(\frac{1}{\sqrt{\log N}}\right)^{2(\ell+1)\log N/\operatorname{loglog} N} = \frac{1}{N^\ell}$$

Die Abschätzung $(*)$ gilt, da für genügend große N gilt:

$$\frac{\log N}{\operatorname{loglog} N} \geq \sqrt{\log N}$$

Zu c)
Es gilt: $C^h \leq N$. Sei $u = 2(\ell+1)\log N/\operatorname{loglog} N$. Damit folgt:

$$\begin{aligned}
E(C^h) &\leq \operatorname{Prob}(C^h < u) \cdot u + \operatorname{Prob}(C^h \geq u) \cdot N \\
&\leq \Big(1 - \frac{1}{N^\ell}\Big) \cdot u + \frac{1}{N^\ell} \cdot N < u + 1
\end{aligned}$$

□

8.18 Satz:
*Mit dem Standard-Zugriffsschema kann eine $N \log N$-Prozessor-EREW-PRAM auf einer N-Prozessor-*ARBITRARY*-DMM mit erwartetem (Zeit-Prozessor optimalem) Zeitverlust $O(\log N)$ simuliert werden. Die Zeitschranke ist sehr zuverlässig.*

Den Beweis für diesen Satz wollen wir hier nicht führen, er ist recht schwierig. Wir wollen nur die Schwierigkeiten herausarbeiten.
Analog zu Lemma 8.17 kann man zeigen:

8.19 Lemma:
Sei $S \subseteq U$, $|S| \leq N \log N$. Dann ist $E(C^h) = O(\log N)$.

Der Beweis dieser Aussage bleibt zur Übung überlassen.
Somit bekommt (im Sinne des Erwartungswertes) kein Modul mehr als $O(\log N)$ viele Anfragen. Daraus folgt allerdings noch nicht die Schranke $O(\log N)$ für den erwarteten Zeitverlust. Denn es können nicht alle $N \log N$ Anfragen gleichzeitig gestellt werden, sondern es sind zu jedem Zeitpunkt höchstens N Anfragen unterwegs. Dadurch kann es passieren, daß zu vielen Zeitpunkten viele Module keine Anfragen bekommen, obwohl noch Anfragen für sie ausstehen. Man kann Beispiele der Form: „Q_i greift auf $M_{i_1}, \ldots, M_{i_{\log N}}$ zu, $i = 1, \ldots, N$“ konstruieren, so daß jedes Modul nur $\log N$ Anfragen bekommt, das Standard-Zugriffsschema aber Zeit $\Theta(\log N^2)$ benötigt (Übung).
Zum Beweis des obigen Satzes nutzt man implizit aus, daß derartige „gemeine Beispiele“ recht unwahrscheinlich sind.

8.20 Bemerkung:

a) In Ergänzung zu den Lemmata 8.17 und 8.19 kann man auch untere Schranken zeigen:

(i) Für $S \subseteq U$, $|S| = N$, ist

$$E(C^h) = \Omega\Big(\frac{\log N}{\operatorname{loglog} N}\Big) \ .$$

(ii) $E(C^h) = O(t)$ gilt erst für $S \subseteq U$, $|S| = N \cdot t$ für $p = \Omega(\log N)$.

Daraus ergibt sich für PRAM-Simulationen auf der Basis einer Hashfunktion:

- *Der erwartete Zeitverlust bei Simulationen einer N-Prozessor-PRAM auf einer N-Prozessor-DMM ist $\Omega(\log N / \operatorname{loglog} N)$, d. h. Satz 8.15 ist bestmöglich.*
- *Zeit-Prozessor-optimale Simulationen haben einen erwarteten Zeitverlust von $\Omega(\log N)$, d. h. Satz 8.18 ist bestmöglich.*

b) Satz 8.18 gilt nur für EREW-PRAMs. Es ist nicht bekannt, ob das Standard-Zugriffsschema oder Varianten davon auch für CRCW-PRAMs funktioniert. Das Ergebnis aus Satz 8.18 kann allerdings mit einem sehr komplizierten Zugriffsschema erzielt werden ([DM90]).

c) Eine kleine Übungsaufgabe: In Kapitel 8.2 haben wir eine Zeit-Prozessor-optimale PRAM-Simulation auf Butterfly-DMMs mit erwartetem Zeitverlust $O(\log N)$ vorgestellt. Wieso impliziert dieses Ergebnis kein Resultat für ARBITRARY-DMMs?

8.4.2 Simulationen, die mehrere Hashfunktionen benutzen.

In diesem Abschnitt gehen wir von einer redundanten Speicherabbildung aus, wie wir es auch in Abschnitt 8.3 für die deterministische Simulation getan haben. Nun benutzen wir aber an Stelle der in Lemma 8.10 beschriebenen deterministischen Zuordnung der Zellen des gemeinsamen Speichers auf je $\log N$ Module a viele zufällig und unabhängig aus H gezogene Hash-Funktionen $h_1, \ldots, h_a$. Jede Zelle x des gemeinsamen Speichers wird in den Modulen $M_{h_1(x)}, \ldots, M_{h_a(x)}$ verwaltet. Wir benutzen den gleichen Trick wie in Abschnitt 8.3: Um Lesen und Schreiben gleich schwer zu machen, verlangen wir beim Beschreiben von x, daß mehr als die Hälfte der Kopien von x aktualisiert werden. Der Aktualisierung wird ein Zeitstempel hinzugefügt. Beim Lesen von x reicht es nun, irgendwelche, Hauptsache mehr als die Hälfte der a Kopien von x zu lesen. Die mit größtem Zeitstempel sind nun sicher aktuell.

Sei wieder $m = N \cdot p$, M eine m-Prozessor-PRAM mit Prozessoren $P_{i,j}$, $i = 1, \ldots, N$, $j = 1, \ldots, p$.
Wir wollen M auf einer N-Prozessor-c-COLLISION-DMM simulieren. Q_i simuliert $P_{i,1}, \ldots, P_{i,p}$. Um den Zugriff von $P_{i,j}$ auf $x_{i,j}$ zu simulieren, muß Q_i auf irgendwelche b Kopien von x_{ij} zugreifen, $b > \frac{1}{2}a$.
Das folgende (a, b, c)-Schema erledigt diese b Zugriffe auf der Basis von a Hashfunktionen $h_1, \ldots, h_a$ auf einer c-COLLISION-DMM.

(a, b, c)-Schema ($P_{i,j}$ will x_{ij} lesen/beschreiben)

Jedes Q_i führt folgende Schritte aus:

```
for ℓ := 1 to p do
    Count := 0
    while Count < b do
      for d := 1 to a do
        Stelle Lese/Schreibanfrage für x_{i,ℓ} an M_{h_d(x_{i,ℓ})},
          falls dieser Zugriff nicht schon früher erfolgreich war;
        if M_{h_d(x_{i,ℓ})} Anfrage beantwortet/akzeptiert then erhöhe Count;
      done
    done
done
```

Jedes Modul beantwortet in jedem Schleifendurchlauf jeder Runde alle Anfragen, falls höchstens c viele gestellt werden, sonst beantwortet es keine Anfrage.
Es ist zu beachten, daß dieses Schema nicht immer hält. Man beachte etwa den Fall $p = 1$ und nehme an, daß etwa die ersten $c + 1$ Anfragen bzgl. jeder Hashfunktion kollidieren. Sie würden dann nie auch nur ein einziges Mal beantwortet werden. Wir werden u. a. sehen, daß es sehr unwahrscheinlich ist, daß soetwas passiert. Es ist einfach, das Schema so zu ergänzen, daß es immer hält.

8.21 Satz:
Sei $p = 1$. Für $a \geq 2$, $b < a$, $c \geq \max\{4, 2\frac{a-1}{a-b}\}$ benötigt das (a, b, c)-Schema erwartet $O(\log\log N)$ Runden. Diese Schranke ist sehr zuverlässig.

Wie wir oben bereits erwähnt haben, liefert das (a, b, c)-Schema für $b > \frac{1}{2}a$ PRAM-Simulationen auf c-Collision DMMs.

8.22 Satz:
Mit dem $(3, 2, 4)$-Schema kann man eine N-Prozessor-EREW-PRAM mit erwartetem Zeitverlust $O(\log\log N)$ simulieren. Die Zeitschranke ist sehr zuverlässig.

Der Beweis folgt direkt aus Satz 8.21, da eine Runde $O(a)$ Schritte, also konstante Zeit, dauert.
Wir werden Satz 8.21 nicht allgemein beweisen, sondern nur das $(2, 1, c)$-Schema analysieren. Wir werden später auch hieraus eine PRAM-Simulation entwickeln (obwohl hier $a = 2$, $b = 1$, also nicht $b > \frac{1}{2}a$ gilt).

Analyse des $(2, 1, c)$-Schemas, für $p = 1$, falls nicht N, sondern $n = \frac{1}{128}N$ Anfragen (für Zellen $x_1, \ldots, x_n$) gestellt werden.

Zur Analyse betrachten wir folgenden bipartiten Graphen $G = (V_1 \cup V_2, E)$. $V_1 = \{M_1, \ldots, M_N\}$, $V_2 = \{M'_1, \ldots, M'_N\}$.

Für jedes x_i, $i = 1, \ldots, n$, ist $(M_{h_1(x_i)}, M'_{h_2(x_i)}) \in E$. Diese Kante trägt das Label x_i. (Beachte: Mehrfachkanten (mit verschiedenen Labels) sind erlaubt.)

Das $(2, 1, c)$-Schema übersetzt sich direkt in folgendes Spiel auf G:

Es spielen abwechselnd die Knoten aus V_1 und V_2. Spiel von V_1 (bzw. V_2): Jedes M_j (bzw. M'_j) mit Grad $\leq c$ entfernt alle inzidenten Kanten (d. h. gibt für jede u_n einer solchen Kante notierten Anfrage x_k die (eine benötigte) Antwort.) Das Spiel endet, sobald alle Kanten entfernt sind.

Wieviele Runden sind notwendig?

8.23 Lemma:
(Beachte: Die unten aufgeführten Wahrscheinlichkeiten sind relativ zur zufälligen, unabhängigen Wahl von $h_1, h_2 \in H$).
Für den oben definierten Graphen G gilt: Für jedes $\ell \geq 1$ gibt es $\beta, s \geq 1$ mit:

a) Prob(G *hat eine ZHK der Größe mindestens* $\beta \log N$) $\leq N^{-\ell}$

b) Prob(G *hat eine ZHK A mit mindestens* $|A| - 1 + s$ *Kanten*) $\leq N^{-\ell}$.

Beachte: b) besagt, daß jede Zusammenhangskomponente ein Baum mit höchstens s zusätzlichen Kanten ist.

Beweis:
Wir zeigen:

Behauptung:
Sei $k \geq 2$, $s \geq 0$, $k + s - 1 = O(\log N)$.

$$\text{Prob}(G \text{ enthält Subgraph } G' \text{ mit } k \text{ Knoten und mindestens } k - 1 + s \text{ Kanten}) \leq N^{-s+1} \cdot k^{s-1} \cdot 2^{-k-\frac{7}{2}s+\frac{7}{2}}$$

Beweis der Behauptung:
Sei $G_{k,s}$ die Menge aller bipartiten Graphen mit k Knoten aus $V_1 \cup V_2$ und $k + s - 1$ Kanten (Mehrfachkanten erlaubt), jede mit einem Label aus $x_1, \ldots, x_n$. Wir wollen $|G_{k,s}|$ abschätzen.

- Es gibt $\binom{2N}{k}$ Wahlen für die Knotenmenge $V'_1 \cup V'_2 \subseteq V_1 \cup V_2$ der Größe k.
- Es gibt
$$\binom{|V'_1| \cdot |V'_2| + k + s - 1}{k + s - 1}$$
Wahlen der $k + s - 1$ Kanten zwischen V'_1 und V'_2. (Vergleiche Kapitel 6.4 als wir Delay-Sequences gezählt haben.) Diesen Term können wir wie folgt abschätzen: Da $|V'_1| + |V'_2| = k$ ist, ist $|V'_1| \cdot |V'_2| \leq \frac{1}{4}k^2$.

  Also ist für $k > 2$:
$$\begin{aligned}\binom{|V'_1| \cdot |V'_2| + k + s - 1}{k + s - 1} &\leq \left(\frac{(\frac{1}{4}k^2 + k + s - 1) \cdot e}{k + s - 1}\right)^{k+s-1} \\ &\leq e^{k+s-1} \cdot \left(\frac{k^2}{4(k + s - 1)} + 1\right)^{k+s-1} \\ &\leq e^{k+s-1} k^{k+s-1}\end{aligned}$$

- Es gibt höchstens $n^{k+s-1} = (\frac{1}{128}N)^{k+s-1}$ Wahlen für die Labels der Kanten aus $x_1, \ldots, x_n$.

Also folgt:

$$\begin{aligned} |G_{k,s}| &\leq \binom{2N}{k} \cdot e^{k+s-1} \cdot k^{k+s-1} \cdot \left(\frac{N}{128}\right)^{k+s-1} \\ &\leq N^{2k+s-1} \cdot 2^k \cdot e^{2k+s-1} \cdot k^{s-1} \cdot \left(\frac{1}{128}\right)^{k+s-1} \end{aligned}$$

Nun betrachte festes $G' \in G_{k,s}$ mit Knotenmenge $V_1' \cup V_2'$ und Kanten (a_i, b_i) mit Label x_{j_i}, $i = 1, \ldots, k+s-1$.
Da h_1, h_2 unabhängig aus einer $O(\log N)$-universellen Klasse von Hashfunktionen gezogen wurde, und $k+s-1 = O(\log N$ ist, folgt:

$$\begin{aligned} &\text{Prob}(G' \text{ ist Subgraph von } G) \\ &\leq \text{Prob}(h_1(x_{j_i}) = a_i,\ h_2(x_{j_i}) = b_i \text{ für } i = 1, \ldots, k+s-1) \\ &\leq \left(\frac{2}{N}\right)^{2(k+s-1)} . \end{aligned}$$

Somit folgt:

$$\begin{aligned} &\text{Prob}(\exists G' \in G_{k,s}, \text{ so daß } G' \text{ Subgraph von } G \text{ ist}) \\ &\leq |G_{k,s}| \cdot \left(\frac{2}{N}\right)^{2(k+s-1)} \\ &\leq N^{2k+s-1} \cdot 2^k \cdot e^{2k+s-1} \cdot k^{s-1} \cdot \left(\frac{1}{128}\right)^{k+s-1} \cdot \left(\frac{2}{N}\right)^{2(k+s-1)} \\ &\leq N^{-s+1} \cdot k^{s-1} \cdot 2^{-7(k+s-1)+2(k+s-1)+k+\frac{3}{2}(2k+s-1)} \\ &\leq N^{-s+1} \cdot k^{s-1} \cdot 2^{-k-\frac{7}{2}s+\frac{7}{2}} \end{aligned}$$

Somit ist die Behauptung bewiesen. □
Um Lemma 8.23 a) zu beweisen, benutzen wir die obige Behauptung mit $s = 0$.

$$\begin{aligned} &\text{Prob}(G \text{ hat Zusammenhangskomponente der Größe} \geq \beta \log N) \\ &\leq \text{Prob}(G \text{ hat Subgraph mit } k = \beta \log N \text{ Knoten und mindestens } k-1 \text{ Kanten}) \\ &\leq N \cdot k^{-1} \cdot 2^{-k+7/2} \\ &\leq N \cdot 2^{-k} \leq \frac{1}{N^\ell} \text{ für } k \geq (\ell+1) \log N \end{aligned}$$

D. h. $\beta = \ell + 1$ ist o.k.
Um Lemma 8.23 b) zu beweisen, benutzen wir die Behauptung und Lemma 8.23 a) für ein genügend großes l' wie folgt:

$$\begin{aligned} &\text{Prob}(G \text{ hat Zusammenhangskomp. } A \text{ mit mind. } |A| + s - 1 \text{ Kanten}) \\ &\leq \sum_{k=1}^{N} \text{Prob}(G \text{ enthält } G' \in G_{k,s}) \end{aligned}$$

$$
\begin{aligned}
&\leq \sum_{k=1}^{\beta \log N} \text{Prob}(G \text{ enthält } G' \in G_{k,s}) + N \cdot N^{-\ell'} \\
&\leq N^{-\ell'+1} + \sum_{k=1}^{\beta \log N} N^{-s+1} \cdot k^{s-1} 2^{-k-\frac{7}{2}s+\frac{7}{2}} \\
&\leq N^{-\ell+1} + N^{-s+1} \cdot 2^{-\frac{7}{2}s+\frac{7}{2}} \cdot \sum_{k=1}^{\beta \log N} k^{s-1} \cdot 2^{-k} \\
&\leq N^{-\ell'+1} + N^{-s+1} \cdot 2^{-\frac{7}{2}s+\frac{7}{2}} \cdot (\beta \log N)^{s-1} \\
&\leq N^{-\ell}
\end{aligned}
$$

für genügend große ℓ', s. □

Wir kommen nun zurück zur Analyse des $(2,1,c)$-Schemas mit $\frac{N}{128}$ Anfragen.

8.24 Lemma:
Sei G, wie oben definiert, der zum $(2,1,c)$-Schema gehörige Graph. G erfülle die in Lemma 8.23 definierten Eigenschaften mit Parametern β und s. Für $c = s+2$ benötigt dann das $(2,1,c)$-Schema nur $O(\log\log N)$ Runden.

Beweis:
Betrachte eine Zusammenhangskomponente A von G, $|A| \leq \beta \log N$.
Wir nehmen zuerst an, daß A ein Baum ist, und $c = 2$. Das $(2,1,c)$-Schema übersetzt sich, wie schon oben erklärt, in ein Spiel auf G, in dem in jeder Runde jeder Knoten vom Grad höchstens $c(=2)$ alle seine inzidenten Kanten streicht. Für A bedeutet dieses: Es werden in der ersten Runde alle zu Blättern und zu Knoten von Grad zwei inzidenten Kanten bestrichen. Da über die Hälfte der Knoten eines Baumes Grad 1 oder 2 haben (man überlege sich, wieso?), bleiben von A nur Komponenten der Größe $\leq |A|/2$ über. Induktiv folgt, daß nach $\log(|A|)$ Runden keine Kante in A übrig ist. Durch die Wahl $c = s+2$ wird sichergestellt, daß auch die übrigen s Kanten während der $\log(|A|)$ Runden gestrichen werden.
Also folgt:
Rundenzahl $\leq \max\{\log(|A|) \mid A \text{ ist Zusammenhangskomponente von } G) \leq \text{loglog}(\beta N)) = O(\log\log N)$. □

8.25 Satz:
Für jedes $\ell \geq 1$ gibt es ein $c \geq 1$, so daß das $(2,1,c)$-Schema für $\frac{N}{128}$ Anfragen mit Wahrscheinlichkeit $1 - N^{-\ell}$ in Zeit $O(\log\log N)$ beendet ist.

Der Beweis folgt direkt aus den Lemmata 8.23 und 8.24.
Wir bekommen wie folgt eine PRAM-Simulation mit Hilfe des obigen Satzes.

- Benutze 3 Hashfunktionen h_1, h_2, h_3 unabhängig aus H gezogen.
- Ein Zugriff auf $x_1, \ldots, x_N$ wird in 128 Schüben mit je $\frac{N}{128}$ Zugriffen realisiert.
- Pro Schub wird das obige $(2,1,c)$-Schema 3 mal verwandt, und zwar bezüglich h_1, h_2; h_2, h_3; h_3, h_1.

Mit Wahrscheinlichkeit $1 - 3N^{-\ell}$ benötigt jede Anwendung des Schemas Zeit $O(\text{loglog}\, N)$. Also wird insgesamt Zeit $128 \cdot 3 \cdot O(\text{loglog}\, N) = O(\text{loglog}\, N)$ benötigt.
Zusätzlich gilt: Für jeden Schlüssel wird mindestens auf zwei Kopien zugegriffen (warum?).
Somit erhalten wir eine PRAM-Simulation, da wir auf mehr als die Hälfte der Kopien zugreifen.

8.26 Satz:
*Obiges Verfahren liefert eine Simulation einer N-Prozessor-EREW-PRAM auf einer c-*Collision*-DMM mit erwartetem Zeitverlust $O(\text{loglog}\, N)$, falls c eine genügend große Konstante ist.*

8.27 Bemerkung:

a) *Bessere Analysen, wie sie in [Sch93] oder [MSS94] zu finden sind, erlauben es, alle Anfragen gleichzeitig zu bearbeiten und $c = 2$ zu benutzen, oder in Schüben zu arbeiten, und $c = 1$ zu benutzen (mit $a = 9$, $b = 5$).*

b) *Simulationen zeigen, daß bei $a = 3$, $b = 2$, $c = 2$ und „alle Anfragen gleichzeitig“ bis zu $N = 1.000.000$ Prozessoren 3 Runden reichen.*

c) *Für $p = \log N$ liefert das $(3, 2, 3)$-Schema eine Zeit-Prozessor-optimale Simulation mit erwartetem Zeitverlust $O(\log N)$.*

d) *Techniken aus [KLM92] liefern Simulationen, die für das Lesen ein $(a, 1, c)$-Schema nutzen, und durch „verzögertes Schreiben“ den Schreibzugriff in erwarteter Zeit $O(\log^* N \cdot a)$ ermöglichen. Diese Techniken liefern u. a. Zeit-Prozessor-optimale Simulationen auf* Arbitrary*-DMMs mit erwartetem Zeitverlust $O(\text{loglog}\, N \cdot \log^* N)$.*

e) *Verallgemeinerte Varianten der oben vorgestellten Algorithmen erlauben Simulationen mit erwartetem Zeitverlust $O(\frac{\text{loglog}\, N}{\text{logloglog}\, N})$ ([MSS94]).*

f) *In[CMS95] wird ein komplizierteres Schema vorgestellt, das eine erwartete Simulationszeit von $O(\text{logloglog}\, N \cdot \log^* N)$ erlaubt.*

8.5 Abschließende Bemerkungen

Schreibkonfliktlösungen. Neben der c-Collision- und Arbitrary-Regel gibt es noch die folgenden, häufig benutzten Schreibkonfliktlösungen:

- **Priority**: Falls mehrere Prozessoren gleichzeitig in die Zelle u schreiben wollen, erscheint in Zelle u der Wert, den der Prozessor mit kleinster Prozessornummer geschrieben hat.
- **COMMON**: Falls mehrere Prozessoren gleichzeitig in dieselbe Zelle schreiben wollen, so müssen sie alle den gleichen Wert schreiben wollen.
- **COLLISION**: Falls mehrere Prozessoren gleichzeitig in die Zelle u schreiben wollen, erscheint in Zelle u ein spezielles Kollisionssymbol.
- **ROBUST**: Falls mehrere Prozessoren gleichzeitig in die Zelle u schreiben wollen, erscheint in Zelle u irgendein (eventuell gemein gewähltes) Symbol.

Eine Übersicht über die Fähigkeiten dieser PRAMs mit verschiedenen Schreibkonfliktlösungen, sich gegenseitig zu simulieren, und deren Grenzen gibt[Fic93]. Einen zusammenfassenden Überblick über Simulationstechniken gibt Harris in[Har94].

Es gibt noch eine ganze Reihe von weiteren Schreibkonfliktlösungen, auf die wir hier nicht eingehen wollen.

Komplexitätstheorie. In der Komplexitätstheorie gibt es die Klasse $\mathcal{NC}$ der Probleme der Größe n, die auf einer CRCW-PRAM in sog. polylogarithmischer Zeit, d. h. in Zeit $O((\log n)^{k_1})$ für irgendeine Konstante k_1, mit polynomiell vielen Prozessoren, d. h. $O(n^{k_2})$ vielen für eine weitere Konstante k_2, gelöst werden können. Unsere Untersuchungen in diesem Kapitel zeigen, daß in der Definition von $\mathcal{NC}$ die Benutzung von PRAMs keine Einschränkung darstellt, und daß $\mathcal{NC}$ eben auch über Netzwerke definiert werden könnte. Beachte die Analogie zur sequentiellen Klasse $\mathcal{P}$ der in Polynomialzeit lösbaren Probleme, die invariant dagegen ist, ob man sie über Turingmaschinen oder Registermaschinen definiert.

8.6 Literatur zu Kapitel 8

[AHMP87] H. Alt, T. Hagerup, K. Mehlhorn, and F. Preparata. Deterministic simulation of idealized parallel computers on more realistic ones. *SIAM Journal on Computing*, 16:808–835, 1987.

[CMS95] A. Czumaj, F. Meyer auf der Heide, and V. Stemann. Shared memory simulations with triple-logarithmic delay. In *ESA '95: Proceedings of the Third Annual European Symposium on Algorithms*, pages 46–59, London, UK, 1995. Springer-Verlag.

[Col88] R. Cole. Parallel merge sort. *SIAM Journal on Computing*, 17:770–785, 1988.

[CW79] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

[DM90] M. Dietzfelbinger and F. Meyer auf der Heide. How to distribute a dictionary in a complete network. In *Proceedings of the 22nd ACM-STOC*, pages 117–127, 1990.

[DM93] M. Dietzfelbinger and F. Meyer auf der Heide. Simple, efficient shared memory simulations. In *Proceedings of the 5th ACM-SPAA*, pages 110–119, 1993.

[Fic93] F. E. Fich. The complexity of computation on the parallel random access machine. In J. H. Reif, editor, *Synthesis of Parallel Algorithms*, pages 843–899. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[Har94] T. J. Harris. A survey of PRAM simulation techniques. *ACM Computing Surveys*, 26:187–206, 1994.

[KLM92] R. M. Karp, M. Luby, and F. Meyer auf der Heide. Efficient PRAM simulation on a distributed memory machine. In *Proceedings of the 24th ACM Symposium on Theory of Computing (STOC)*, pages 318–326, 1992.

[KU88] A. R. Karlin and E. Upfal. Parallel hashing: An efficient implementation of shared memory. *Journal of the ACM*, 35:876–892, 1988.

[MSS94] F. Meyer auf der Heide, C. Scheideler, and V. Stemann. *Fast, Simple Dictionary and Shared Memory Simulations on DMM; Upper and Lower Bounds.* Preprint, Universität-GH Paderborn, 1994.

[Sch93] C. Scheideler. *Analyse von Shared-Memory Simulationen, die mehrere Hashfunktionen benutzen.* Diplomarbeit, Universität-GH Paderborn, 1993.

[Sie89] A. Siegel. On universal classes of fast high performance hash functions, their time-space tradeoff, and their applications. In *Proceedings of the 30th IEEE-FOCS*, pages 20–25, 1989.

[UW87] E. Upfal and A. Wigderson. How to share memory in a distributed system. *Journal of the ACM*, 34:116–127, 1987.

[Val90] L. G. Valiant. General purpose parallel architecures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume A – Algorithms and Complexity*, Chapter 18, pages 945–971. Elsevier, Amsterdam, 1990.

Kapitel 9

Datenverwaltung in Netzwerken mit beschränkter Bandbreite

Bei unseren bisherigen Überlegungen zur Datenverwaltung haben wir die DMM zugrunde gelegt und angenommen, daß ein Router mit sehr hoher Bandbreite (d.h. sehr guten Kommunikationsfähigkeiten) die Kommunikation übernimmt. Diese Modellierung paßt sehr gut zu dem Konzept eines verteilten Datenservers: Die Speichermodule sind hier typischweise Plattenspeicher, die über ein Netzwerk mit den Nutzern verbunden sind. Nutzer haben Anfragen an Daten, die vom Datenserver verwaltet werden. Diese Daten sind typischerweise große Blöcke (z.B. Bilder oder Filmsequenzen) und werden oft gelesen aber selten aktualisiert.

Der Flaschenhals in diesen Systemen sind die Platten, da die Geschwindigkeit, mit der Daten von einer Platte gelesen werden können, deutlich geringer ist als die Geschwindigkeit des Netzwerks. Somit beschreibt unser Ansatz, die Contention der Platten, d.h. die maximale Zahl von Anfragen, die bei n Anfragen an die gleiche Platte gerichtet werden, zu minimieren, eine wesentliche Aufgabe bei der Entwicklung eines verteilten Datenservers. Das $(2, 1, c)$-Schema kann so modifiziert werden, daß Contention höchstens 3 mit großer Wahrscheinlichkeit nicht überschritten wird.

Bei Parallelrechnern ist der Flaschenhals allerdings typischerweise nicht die Contention der Knoten des Netzwerkes. Vielmehr ergibt sich, etwa bei einem Gitternetzwerk $M(n, 2)$, daß die Congestion den Flaschenhals bildet. Als Congestion einer Kante bezeichnen wir dabei die Zahl der Botschaften, die im Verlauf der gesamten Anwendung über diese Kante laufen, als Congestion (der Anwendung) bezeichnen wir das Maximum der Congestions aller Kanten.

Wenn wir diese Congestion minimieren wollen, sollten wir die Plazierung der globalen Variablen (mit oder ohne Redundanz) nicht unabhängig von der Anwendung machen, vielmehr sollten wir die in der Anwendung enthaltene "Lokalität" ausnutzen.

9.1 Datenverwaltung für Bäume

Wir nehmen in diesem Kapitel an, daß unser Netzwerk ein Baum T ist. Die Kanten des Baumes sind mit natürlichen Zahlen $g(e)$ gewichtet. $g(e)$ ist die Kapazität oder Bandbreite von e, d.h. über e können bis zu $g(e)$ Botschaften gleichzeitig versandt werden. Durch Simulation der Strategie für Bäume auf Gittern werden wir später auch für Gitter Strategien entwickeln.

9.1.1 Das statische Modell

Sei X die Menge der globalen Variablen, die in unserer Anwendung benutzt werden. Sei $\mathcal{P}$ die Prozessormenge unseres Netzwerks, $h_r : \mathcal{P} \times X \to \mathbb{N}$ ($h_w : \mathcal{P} \times X \to \mathbb{N}$) beschreibt die Lese- (Schreib-) Häufigkeiten: $h_r(P, x)$ ($h_w(P, x)$) gibt an, wie oft während der Anwendung Prozessor P lesend (schreibend) auf Variable x zugreift.

Die Aufgabe besteht darin, die Variablen auf die Knoten (ggf. redundant) zu plazieren, so daß die Congestion minimiert wird.

Genauer: Betrachte eine Plazierung der Variablen auf den Knoten des Baumes, so daß jede Variable mindestens einmal plaziert ist.

Prozessor P wird $h_r(P, x)$ mal lesend auf A zugreifen. Dafür liest er die ihm am nächsten liegende Kopie, sagen wir im Prozessor P'. Somit ist jede Kante auf dem eindeutigen Weg im Baum zwischen P und P' $h_r(P, x)$ mal belastet. Prozessor P wird zudem $h_w(P, x)$ mal schreibend auf x zugreifen. Dafür muß er **alle** Kopien von x beschreiben. D.h., alle Kanten in dem kleinsten Subbaum von T, der P und alle Prozessoren enthält, die Kopien von x haben (der sog. Steiner Baum dieser Prozessormenge) werden $h_w(P, x)$ mal belastet. Wenn wir die so entstehenden Lasten pro Kante e über alle (P, x) aufsummieren, erhalten wir die Last $l(e)$, d.h. die Anzahl $l(e)$ von Botschaften, die während der Anwendung bei der vorgegebenen Plazierung der Variablen über e laufen. Da e in einem Schnitt $g(e)$ Botschaften weiterleiten kann, ist die Congestion $C(e) := \frac{l(e)}{g(e)}$, und die (Gesamt-) Congestion $C = max\{C(e), e \in E\}$.

Eine optimale Plazierung zu suchen bedeutet nun, eine Plazierung der Variablen zu suchen, die minimale Congestion hat.

9.1.2 Das dynamische Modell

Die Lese- und Schreibhäufigkeiten sind für manche Algorithmen durch theoretische oder experimentelle Analyse des Algorithmus recht gut abzuschätzen bzw. sogar genau zu berechnen. Das geht immer dann, wenn das Kommunikationsmuster des Algorithmus (wer liest/beschreibt wann welche Variable) unabhängig vom Input ist. Beispiele sind Matrixmultiplikation oder Batchers Bitones Sortieren. Für viele andere Anwendungen ist allerdings das Kommunikationsmuster hochgradig vom Input abhängig, wie z.B. bei vielen auf Adjazenzlisten basierenden Graphalgorithmen (für Zusammenhang, Spannbäume, etc.). In diesen Fällen ist es nicht möglich, die Funktionen h_r und h_w auch nur annähernd anzugeben. Trotzdem steckt häufig für viele Inputs z eine gewisse Lokalität im bei z auftretenden Kommunikationsmuster; wir kennen nur leider diese Lokalität nicht im voraus. Können wir sie trotzdem ausnutzen, um die Congestion zu reduzieren? Wenn ja, wie können wir die Qualität eines solchen Algorithmus bewerten?

Wir werden dynamische Strategien betrachten. Hier ist es erlaubt, daß während der Anwendung Kopien von Variablen gelöscht bzw. bei Prozessoren (durch Routing von Prozessoren aus, die schon eine Kopie haben) neu erzeugt werden. Wir gehen davon aus, daß beim Lesen die aktuell nächste Kopie gelesen wird, beim Schreiben der schreibende Prozessor eine neue Kopie bei sich anlegt, und alle anderen löschen muß. Zudem dürfen beliebig Kopien erzeugt werden (wobei allerdings auch Congestion erzeugt wird!).

Wie bewerten wir die Qualität einer solchen Strategie? Durch competitive Analyse! Bei dieser Analyse vergleichen wir die Congestion unserer Lösung mit der Congestion einer Lösung, die man erreichen könnte, wenn man das gesamte Kommunikationsmuster (wer greift wann le-

send/schreibend auf welche Variable zu) kennen würde, und die optimale dynamische Strategie hierfür nutzen würde. Dabei ist es egal, wie aufwendig es wäre, diese optimale (sogenannte off-line) Strategie zu berechnen. Der Faktor, um den unsere (sogenannte on-line) Strategie höhere Congestion aufweist als die optimale off-line Strategie, ist die competitive Ratio R. Kleines R bedeutet, daß unser on-line Strategie die vorhandene Lokalität (die ja von der off-line Strategie optimal ausgenutzt wird) zumindest recht gut ausnutzt. Wir werden on-line Strategien mit competitive Ratio 3 für Bäume entwickeln.

9.1.3 Die statische Strategie für Bäume

Sei unser Netzwerk ein Baum $T = (\mathcal{P}, E)$ mit Kantenkapazitäten $g(e), e \in E$.
Unsere Anwendung benutze die Variablenmenge X, und die Lese/Schreibhäufigkeiten seien durch die Funktionen $h_r, h_w : \mathcal{P} \times X \to \mathbb{N}$ gegeben.
Betrachte eine feste Variable $x \in X$.

Bezeichnungen:

Für $P \in \mathcal{P}$ sei $r(P) := h_r(P, x),\ w(\mathcal{P}) := h_w(P, x),\ h(P) := r(P) + w(P)$. Für einen Subbaum $T' = (\mathcal{P}', E')$ von T sei $r(T') := \sum_{P \in \mathcal{P}'} r(P),\ w(T') := \sum_{P \in \mathcal{P}'} w(P),\ h(T') := r(T') + w(T')$.
Wo, in wievielen Prozessoren sollen Kopien von x liegen?
Zuerst untersuchen wir, wo wir eine Kopie plazieren sollten, falls nur eine einzige erlaubt ist. Man überlege sich: Wenn wir die Kopie an Prozessor P geben, erhalten wir als Congestion $C_p = \max\{h(T_i), i = 1, \ldots, r\}$, wobei $T_1, \ldots, T_r$ die Bäume sind, in die T durch Entfernen von P zerfällt. P nennen wir *Gravitationszentrum*, falls $C_p = \min\{C_{p'}, P' \in \mathcal{P}\}$ gilt.

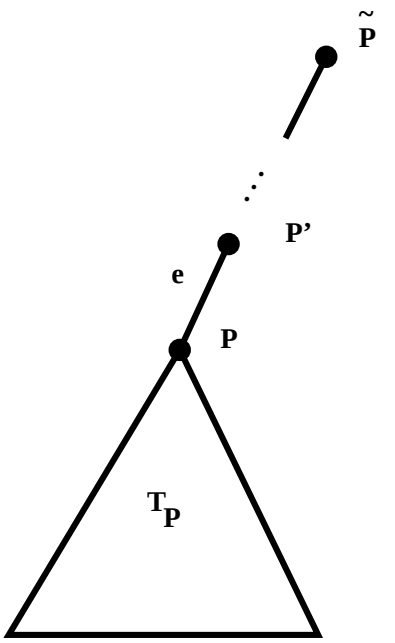
9.1 Lemma:
P ist Gravitationszentrum genau dann, wenn $C_p \leq h(T)/2$ gilt.

Somit haben wir eine optimale Plazierung für Variable x, falls wir nur eine Kopie anlegen dürfen. Wenn wir mehrere Kopien anlegen dürfen, nutzen wir folgendes Protokoll:

Statisches Protokoll

1. Ein Gravitationszentrum $\tilde{P}$ bekommt eine Kopie.
2. Wir fassen $\tilde{P}$ als Wurzel auf und richten alle Kanten in Richtung $\tilde{P}$. Für einen Knoten $P \neq P'$ betrachten wir den Subbaum T_P von T mit Wurzel P. P bekommt eine Kopie, genau dann wenn $h(T_P) \geq w(T)$ ist.

Warum ist diese Strategie gut?
Wir führen hier keinen formalen Beweis, sondern erklären nur intuitiv:



Falls P keine Kopie bekommt, kann man sich überlegen, daß nirgendwo in T_p eine Kopie plaziert wird. Also laufen Zugriffe auf x von Prozessoren aus T_p über e, d.h. e bekommt Last $h(T_p)$.
Falls aber P eine Kopie bekommt, läuft kein Lese-Befehl aus T_p über e, dafür aber alle Schreibbefehle aus T, d.h. e bekommt Last $w(T)$. Also: Falls $w(T) < h(T_p)$ gilt, lohne es sich, auf P eine Kopie zu legen.
Man kann zeigen:

9.2 Satz:
Das statische Protokoll erzeugt für Bäume minimale Last auf jeder Kante.

Da nicht nur die maximale Last, sondern die Last jeder Kante minimiert wird, folgt, daß wir zu einer optimalen Plazierung für alle Variablen kommen, wenn wir jede einzeln mit der statischen Strategie plazieren. Es ist einfach, dieses auf dem Baumnetzwerk in Zeit $O(|X| +$ [Durchmesser von T] $\cdot$ [Grad von T]) durchzuführen.

9.1.4 Die dynamische Strategie für Bäume

Sei wiederum $T = (\mathcal{P}, E)$ ein Baum mit Kantenkapazitäten $g(e)$ für $e \in E$, X die Menge der bei unserer Anwendung benutzten Variablen, $x \in X$. Das folgende einfache Protokoll beschreibt, wie Zugriffe auf Variable x ausgeführt werden, wie dabei Kopien von x gelöscht bzw. neu angelegt werden. Prozessoren, die eine Kopie von x haben, nennen wir *Vertreter von* x.

Dynamisches Protokoll

- Falls Prozessor P die Variable x lesen will, sendet P eine Botschaft zum nächstgelegenen Vertreter P' von x. P' schickt den aktuellen Inhalt von x auf dem (eindeutigen) Weg von P' nach P, und erzeugt in jedem Knoten auf diesem Weg eine Kopie von x.

- Falls Prozessor P die Variable x beschreiben will, führt P ein Multicast zu allen Vertretern von x aus, und löscht deren Kopien von x. Die nächstgelegene Kopie wird dabei zu P geschickt und dort aktualisiert. P hat danach die einzige Kopie von x.

Bemerkung: Wir benötigen zusätzlich Protokolle, die jedem Prozessor sagen, wo die nächste Kopie von x ist. Da sich deren Position ändern kann, müssen wir zur Laufzeit diese Information aktualisieren. Man nennt diese Aufgabe Data Tracking, im allgemeinen ist sie recht aufwendig. Für unsere Strategie auf Bäume ist sie jedoch einfach.

Man überlege sich: Die Vertreter von x bilden immer einen Teilbaum von T.

Vertreter von x halten immer Zeiger auf benachbarte Vertreter von x, andere Prozessoren halten Zeiger auf den Nachbarn in Richtung des nächsten Vertreters von x. Man überlege sich, daß ohne zusätzliche Kommunikation während des dynamischen Protokolls diese Zeiger aktualisiert werden können. (Beachte: Das ist nur in Bäumen einfach, da hier ja nur ein Weg zwischen zwei Prozessoren existiert.)

9.3 Satz:
Das dynamische Protokoll ist für Bäume 3-competitive.

Beweis:
Betrachte eine Folge von Zugriffen auf Variable x, die nach einem Schreiben beginnt und mit dem nächsten Schreiben endet:
$P_1, P_2, \ldots, P_{k-1}$ lesen x, P_k beschreibt x. Zuvor habe P_o die Variable x beschrieben, d.h. zu Beginn hat nur P_o eine Kopie von x.

Betrachte den durch $P_o, \ldots, P_k$ induzierten Steiner Baum T' von T, d.h. den kleinsten Subbaum von T, der $P_o, \ldots, P_k$ enthält. Offensichtlich muß jede, also auch die optimale off-line Strategie, jede Kante von T' mindestens einmal durchlaufen. Unser Strategie durchläuft jede dieser Kanten höchstens dreimal: (i) höchstens einmal, um eine Leseanfrage abzuschicken, (ii) höchstens einmal, um die Variable zurückzuschicken ((i) und (ii) inklusive der Anfrage von P_k an die nächste Kopie), und (iii) einmal zum Löschen durch den Multicast von P_k aus.

Beachte:

In (i) und (ii) wird jede Kante jeweils nur einmal benutzt, da beim Lesen durch P_1 auf dem "Rückweg" an jedem Knoten des Weges von P_o nach P_1 eine Kopie angelegt wird. P_2 wird also seine nächste Kopie finden, sobald er auf diesen Weg stößt, und keine der Kanten dieses Weges von P_o nach P_1 nochmal benutzen. Analoges gilt für $P_2, \ldots, P_{k-1}$. □

9.2 Datenverwaltung für die Gitter-Netzwerke

Für Gitternetzwerke ist es nicht mehr so einfach wie für Bäume, effiziente Datenverwaltung durchzuführen. Wir haben für Bäume eine optimale statische Plazierung mit der statischen Strategie entwickelt, die sehr effizient, nämlich verteilt im Baumnetzwerk in Zeit $O([\text{Durchmesser}]\cdot[\text{Grad}]+|X|)$ durchgeführt werden kann. Sequentiell kann man die Plazierung einfach in $O([\text{ Größe von } T] \cdot |X|)$, also in linearer Zeit berechnen. (Beachte: Input sind der gewichtete Baum T (Größe $O(|T|)$) sowie die Tabellen der Funktionen h_w und h_r (Größe $O([\text{Größe von } T]|X|)$.) Für Gitter wird das Problem sehr viel schwieriger.

9.4 Satz:
Die Berechnung einer optimalen statischen Plazierung ist schon auf dem 3×3*-Gitter NP-schwer. (ohne Beweis)*

Wir versuchen es deshalb mit Approximationen. Dabei werden wir einen geschickt gewählten Baum geschickt in das Gitter einbetten, und die Baumstrategie simulieren. Der Einfachheit halber nehmen wir an, daß wir ein $n \times n$-Gitter M mit $n = 2^k$ vorliegen haben. Dann betrachten wir folgenden Baum T_k vom Grad 4 mit Tiefe k. Für $k = 0$ ist $T(0)$ ein Knoten, der mit dem 1 x 1-Gitter assoziiert wird. Für $k > 0$ assoziieren wir zur Wurzel des Baumes das gesamte Gitter, zu den Wurzeln der vier darunter hängenden Subbäume die vier $2^{k-1} \times 2^{k-1}$ -Subgitter, in die wir M zerlegen. In jedem Subgitter werden die Subbäume induktiv definiert.

Wir werden die Datenverwaltung pro Variable im Gitter dadurch realisieren, daß wir den Baum für jede Variable in das Gitter einbetten, und zwar jeden Baumknoten in einen zufälligen Knoten seines assoziierten Subgitters, wobei die zufälligen Wahlen unabhängig sind, auch bzgl. verschiedener Variablen. Wir stellen fest, daß nun große Subgitter gegenüber kleinen im Vorteil sind, da sie über viel mehr Möglichkeiten verfügen, Daten in ihr übergeordnetes Subgitter zu schicken; ein $2^k \times 2^k$-Subgitter hat hierfür zwischen 2^{k+1} und 2^{k+2} viele Möglichkeiten, da es soviele Kanten nach aussen hat. Wir werden deshalb die Kanten des Baumes entsprechend mit Kapazität versehen:

Eine Kante, zu deren Endknoten ein $2^{k+1} \times 2^{k+1}$- und ein $2^k \times 2^k$-Gitter assoziiert sind, bekommt Kapazität 2^k.

Wir werden nun auf den so eingebetteten 4-ären Bäumen mit den o.g. Kapazitäten die Datenverwaltungsstrategien aus Kapitel 2 laufen lassen. Die dabei notwendige Kommunikation zwischen benachbarten Baumknoten wird durch Routing zwischen den Gitterknoten durchgeführt, in die die Baumknoten eingebettet sind. Als Routingweg wird der eindeutige Dimension-Order-Weg (erst waagerecht, dann senkrecht) genutzt.

Mit einigem technischen Aufwand, den wir hier nicht vorführen, läßt sich zeigen:

9.5 Satz:
Die durch obige Baum-Einbettung und die statische Strategie entstehende Datenverwaltung auf $n \times n$*-Gittern approximiert die Congestion einer optimalen Strategie mit großer Wahrscheinlichkeit bis auf einen Faktor* $O(\log(n))$*.*

9.6 Satz:
Die durch obige Baum-Einbettung und die dynamische Strategie entstehende Datenverwaltung auf $n \times n$*-Gittern ist mit großer Wahrscheinlichkeit* $O(\log(n))$ *competitive.*

9.3 Abschließende Bemerkungen

- Varianten der Strategie sind implementiert und getestet worden, sie liefern sehr gute Ergebnisse. Informationen dazu gibt es z.B. unter
  http://wwwcs.uni-paderborn.de/SFB376/a2/diva.html.
- Erweiterungen sind gezeigt worden z.B. für sog. Cluster-Netzwerke, die recht gut die Struktur des Internet wiederspiegeln, und für die Behandlung unterschiedlich großer Datenobjekte, deren Zugriff entsprechend unterschiedlich hohe Congestion verursacht.

Literaturverzeichnis

[AHMP87] H. Alt, T. Hagerup, K. Mehlhorn, and F. Preparata. Deterministic simulation of idealized parallel computers on more realistic ones. *SIAM Journal on Computing*, 16:808–835, 1987.

[AKS83] M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in $c \cdot \log n$ parallel steps. *Combinatorica*, 3:1–19, 1983.

[BA91] M. Baumslag and F. Annexstein. A unified framework for off-line permutation routing in parallel networks. *Mathematical Systems Theory*, 24:233–251, 1991.

[Bat68] K. E. Batcher. Sorting networks and their applications. In *AFIPS Conf. Proc. 32*, pages 307–314, 1968.

[BCH+96] S. N. Bhatt, F. R. K. Chung, J.-W. Hong, F. T. Leighton, B. Obrenić, A. L. Rosenberg, and E. J. Schwabe. Optimal emulations by Butterfly-like networks. *Journal of the ACM*, 43:293–330, 1996.

[Ben64] V. Beneš. Permutation groups, complexes, and rearrangeable multistage connecting networks. *Bell System Technical Journal*, 43:1619–1640, 1964.

[Ben65] V. Beneš. *Mathematical Theory of Connecting Networks and Telephone Traffic*. Academic Press, New York, NY, 1965.

[BH85] A. Borodin and J. E. Hopcroft. Routing, merging, and sorting on parallel models of computation. *Journal of Computer and System Sciences*, 30:130–145, 1985.

[Bol98] B. Bollobás. *Modern Graph Theory*. Springer-Verlag, Heidelberg, 1998.

[Chv92] V. Chvátal. Lecture notes on the new AKS sorting network. Technical Report DCS-TR-294, Rutgers University, Computer Science Department, November 1992.

[CMS95] A. Czumaj, F. Meyer auf der Heide, and V. Stemann. Shared memory simulations with triple-logarithmic delay. In *ESA '95: Proceedings of the Third Annual European Symposium on Algorithms*, pages 46–59, London, UK, 1995. Springer-Verlag.

[Col88] R. Cole. Parallel merge sort. *SIAM Journal on Computing*, 17:770–785, 1988.

[CP93] R. Cypher and C. G. Plaxton. Deterministic sorting in nearly logarithmic time on the hypercube and related computers. *Journal of Computer and System Sciences*, 47:501–548, 1993.

[CW79] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

[DM90] M. Dietzfelbinger and F. Meyer auf der Heide. How to distribute a dictionary in a complete network. In *Proceedings of the 22nd ACM-STOC*, pages 117–127, 1990.

[DM93] M. Dietzfelbinger and F. Meyer auf der Heide. Simple, efficient shared memory simulations. In *Proceedings of the 5th ACM-SPAA*, pages 110–119, 1993.

[DPSR89] M. Dowd, Y. Perl, M. Saks, and L. Rudolph. The periodic balanced sorting network. *Journal of the ACM*, 36:738–757, 1989.

[Fic93] F. E. Fich. The complexity of computation on the parallel random access machine. In J. H. Reif, editor, *Synthesis of Parallel Algorithms*, pages 843–899. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[Fig89] M. Figge. *Permutationsrouting auf hochdimensionalen Gittern.* Diplomarbeit, Universität Dortmund, 1989.

[FM96] R. Feldmann and P. Mysliwietz. The Shuffle Exchange network has a Hamiltonian path. *Mathematical Systems Theory*, 29:471–485, 1996.

[GP83] Z. Galil and W. Paul. An efficient general-purpose parallel computer. *Journal of the ACM*, 30:360–387, 1983.

[GR88] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, Cambridge, 1988.

[GS92] M. Gruß and H. Selke. Über die explizite Konstruktion von Expandergraphen. Diplomarbeit, Universität-GH Paderborn, 1992.

[Har94] T. J. Harris. A survey of PRAM simulation techniques. *ACM Computing Surveys*, 26:187–206, 1994.

[HHL88] S. M. Hedetniemi, S. T. Hedetniemi, and A. L. Liestman. A survey of gossiping and broadcasting in communication networks. *Networks*, 18:319–349, 1988.

[HY98] F. K. Hwang and Y. C. Yao. Comments on the oblivious routing algorithm of Kaklamanis, Krizanc, and Tsantilas in the hypercube. *Theory of Computing Systems*, 31:63–66, 1998.

[JáJ92] J. JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, Reading, MA, 1992.

[KKT91] C. Kaklamanis, D. Krizanc, and T. Tsantilas. Tight bounds for oblivious routing in the hypercube. *Mathematical Systems Theory*, 24:223–232, 1991.

[KLM92] R. M. Karp, M. Luby, and F. Meyer auf der Heide. Efficient PRAM simulation on a distributed memory machine. In *Proceedings of the 24th ACM Symposium on Theory of Computing (STOC)*, pages 318–326, 1992.

[KLM+97] R. R. Koch, F. T. Leighton, B. M. Maggs, S. B. Rao, A. L. Rosenberg, and E. J. Schwabe. Work-preserving emulations of fixed-connection networks. *Journal of the ACM*, 44:104–147, 1997.

[Knu98] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, Reading, Massachusetts, 2nd edition, 1998.

[KU88] A. R. Karlin and E. Upfal. Parallel hashing: An efficient implementation of shared memory. *Journal of the ACM*, 35:876–892, 1988.

[Lei85] T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, 34:344–354, 1985.

[Lei90] F. T. Leighton. Average case analysis of greedy routing algorithms on arrays. In *Proceedings of 2nd SPAA*, pages 2–10, 1990.

[Lei92a] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[Lei92b] T. Leighton. Methods for message routing in parallel machines. In *Proceedings of the 24th ACM Symposium on Theory of Computing (STOC)*, pages 77–96, 1992.

[LM92] F. T. Leighton and B. M. Maggs. Fast algorithms for routing around faults in multibutterflies and randomly-wired splitter networks. *IEEE Transactions on Computers*, 41:578–587, 1992.

[LMRR94] T. Leighton, B. M. Maggs, A. G. Ranade, and S. B. Rao. Randomized routing and sorting on fixed connection networks. *Journal of Algorithms*, 17:157–205, 1994.

[LP90] T. Leighton and C. G. Plaxton. A (fairly) simple circuit that (usually) sorts. In *Proceedings of the 31st IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 264–274, 1990.

[LPS88] A. Lubotzky, R. Phillips, and P. Sarnak. Ramanujan graphs. *Combinatorica*, 8:261–277, 1988.

[LPV81] G. F. Lev, N. Pippenger, and L. G. Valiant. A fast parallel algorithm for routing in permutation networks. *IEEE Transactions on Computers*, 30:93–100, 1981.

[Mey83a] F. Meyer auf der Heide. Efficiency of universal parallel computers. *Acta Informatica*, 19:269–296, 1983.

[Mey83b] F. Meyer auf der Heide. Infinite Cube-Connected Cycles. *Information Processing Letters*, 16:1–2, 1983.

[Mey86] F. Meyer auf der Heide. Efficient simulations among several models of parallel computers. *SIAM Journal on Computing*, 15:106–119, 1986.

[MS90] B. Monien and H. Sudborough. Embedding one interconnection network in another. *Computing*, 7:257–282, 1990.

[MSS94] F. Meyer auf der Heide, C. Scheideler, and V. Stemann. *Fast, Simple Dictionary and Shared Memory Simulations on DMM; Upper and Lower Bounds.* Preprint, Universität-GH Paderborn, 1994.

[MSW97] F. Meyer auf der Heide, M. Storch, and R. Wanka. Optimal tradeoffs between size and slowdown for universal parallel networks. *Theory of Computing Systems*, 30:627–644, 1997.

[MV95] F. Meyer auf der Heide and B. Vöcking. A packet routing protocol for arbitrary networks. In *Proc. 12th Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 291–302, 1995.

[MV97] B. M. Maggs and B. Vöcking. Improved routing and sorting on multibutterflies. In *Proc. 29th ACM Symposium on Theory of Computing (STOC)*, pages 517–530, 1997.

[MW89] F. Meyer auf der Heide and R. Wanka. Time-optimal simulations of networks by universal parallel computers. In *Proc. 6th Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 120–131, 1989.

[MW96] F. Meyer auf der Heide and R. Wanka. Kommunikation in parallelen Rechnernetzen. In I. Wegener, editor, *Highlights aus der Informatik*, pages 177–198. Springer-Verlag, 1996.

[NS80] D. Nassimi and S. Sahni. Data broadcasting in SIMD computers. *IEEE Transactions on Computers*, 30:101–106, 1980.

[NS82a] D. Nassimi and S. Sahni. Parallel algorithms to set up the Benes permutation network. *IEEE Transactions on Computers*, 31:148–154, 1982.

[NS82b] D. Nassimi and S. Sahni. Parallel permutation and sorting algorithms and a new generalized connection network. *Journal of the ACM*, 29:642–667, 1982.

[Par80] D. Parker. Notes on shuffle/exchange-type switching networks. *IEEE Transactions on Computers*, C-29:213–222, 1980.

[Par86] I. Parberry. On recurrent and recursive interconnection patterns. *Information Processing Letters*, 22:285–289, 1986.

[Par87] I. Parberry. *Parallel Complexity Theory.* Pitman/Wiley, London, New York, Toronto, 1987.

[Par90] I. Parberry. An optimal time bound for oblivious routing. *Algorithmica*, 5:243–250, 1990.

[Pat90] M. S. Paterson. Improved sorting networks with $O(\log n)$ depth. *Algorithmica*, 5:75–92, 1990.

[Pla92] C. G. Plaxton. A hypercubic network with nearly logarithmic depth. In *Proceedings of the 24th ACM Symposium on Theory of Computing (STOC)*, pages 405–416, 1992.

[Pra71] V. R. Pratt. *Shellsort and Sorting Networks.* PhD thesis, Stanford University, 1971.

[PV81] F. Preparata and J. Vuillemin. The Cube-Connected Cycles: a versatile network for parallel computation. *Communications of the ACM*, 24:300–309, 1981.

[Qui87] M. J. Quinn. *Designing Efficient Algorithms for Parallel Computers*. McGraw-Hill, 1987.

[Qui88] M. J. Quinn. *Algorithmenbau und Parallelcomputer*. McGraw-Hill, 1988.

[Ran91] A. G. Ranade. How to emulate shared memory. *Journal of Computer and System Sciences*, 42:307–326, 1991.

[Rei93] J. H. Reif, editor. *Synthesis of Parallel Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[RV87] J. H. Reif and L. G. Valiant. A logarithmic time sort for linear size networks. *Journal of the ACM*, 34:60–76, 1987.

[Sav98] J. E. Savage. *Models of Computation: Exploring the Power of Computing*. Addison-Wesley, Reading, 1998.

[Sch80] J. T. Schwartz. Ultracomputers. *ACM Transactions on Programming Languages and Systems*, 2:484–521, 1980.

[Sch93a] C. Scheideler. *Analyse von Shared-Memory Simulationen, die mehrere Hashfunktionen benutzen*. Diplomarbeit, Universität-GH Paderborn, 1993.

[Sch93b] E. J. Schwabe. Constant-slowdown simulations of normal hypercube algorithms on the butterfly network. *Information Processing Letters*, 45:295–301, 1993.

[Sch98] C. Scheideler. *Universal Routing Strategies for Interconnection Networks*. Springer-Verlag, Berlin, 1998.

[Sie89] A. Siegel. On universal classes of fast high performance hash functions, their time-space tradeoff, and their applications. In *Proceedings of the 30th IEEE-FOCS*, pages 20–25, 1989.

[SL87] A. M. Schwartz and M. C. Loui. Dictionary machines on cube-class networks. *IEEE Transactions on Computers*, 36:100–105, 1987.

[Sto71] H. Stone. Parallel processing with the perfect shuffle. *IEEE Transactions on Computers*, 20:153–161, 1971.

[Upf84] E. Upfal. Efficient schemes for parallel communication. *Journal of the ACM*, 31:507–517, 1984.

[Upf92] E. Upfal. An $O(\log N)$ deterministic packet-routing scheme. *Journal of the ACM*, 39:55–70, 1992.

[UW87] E. Upfal and A. Wigderson. How to share memory in a distributed system. *Journal of the ACM*, 34:116–127, 1987.

[Val90] L. G. Valiant. General purpose parallel architecures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume A – Algorithms and Complexity*, Chapter 18, pages 945–971. Elsevier, Amsterdam, 1990.

[VB81] L. Valiant and G. Brebner. Universal schemes for parallel communication. In *Proceedings of the 13th ACM-STOC*, pages 263–277, 1981.

[vL90] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science: Volume A – Algorithms and Complexity*. Elsevier, Amsterdam, 1990.

[Wac90] A. Wachsmann. *Eine theoretische und praktische Untersuchung von Emulationsalgorithmen eines gemeinsamen Speichers auf einem Butterfly-Netzwerk*. Diplomarbeit, Universität Dortmund, 1990.

[Wak68] A. Waksman. A permuting network. *Journal of the ACM*, 15:159–163, 1968.

[Wan88] R. Wanka. *Über die Effizienz von Simulationen eingeschränkter Netzwerkklassen auf universellen parallelen Rechnern*. Diplomarbeit, Universität Dortmund, 1988.

Index