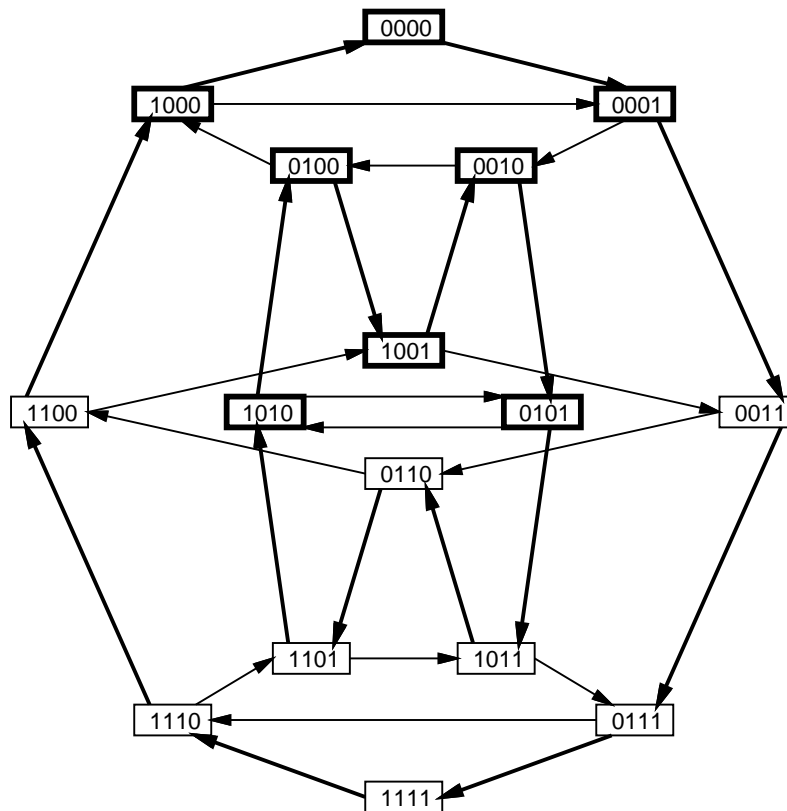


Skript zur Vorlesung Algorithmen für synchrone Rechnernetze

Version 1.34



Prof. Dr. Burkhard Monien, Sven Grothklags, Henning Meyerhenke
Universität Paderborn
Fakultät für Elektrotechnik, Informatik und Mathematik
D-33102 Paderborn

Inhaltsverzeichnis

1	Einleitung	2
1.1	Parallelrechnerarchitekturen	2
1.2	Parallelisierung von Algorithmen am Beispiel der Matrizenmultiplikation . . .	4
1.3	Inhalt der Vorlesung	5
1.4	Literaturangaben und Lehrbücher	6
2	Die Parallele Random Access Maschine	7
2.1	Von der RAM zur PRAM	7
2.2	Die verschiedenen PRAM Modelle	9
3	Netzwerke der Hypercube-Familie	11
3.1	Der Hypercube	13
3.2	Das Butterfly und das Cube-Connected-Cycles Netzwerk	23
3.3	Das DeBruijn und das Shuffle-Exchange Netzwerk	27
4	Algorithmen für synchrone Rechnernetze	34
4.1	ASCEND/DESCEND Algorithmen für den Hypercube	36
4.2	Bitones Sortieren auf der Hypercube	38
4.3	ASCEND/DESCEND Algorithmen auf anderen Netzen der Hypercube-Familie	43
4.3.1	ASCEND/DESCEND auf $SE(k)$	43
4.3.2	ASCEND/DESCEND auf $CCC(k)$	46
4.4	Fast-Fourier-Transformation (FFT)	50
4.5	Broadcasting und Gossiping	54
4.5.1	Broadcasting	55
4.5.2	Gossiping	58
5	Bisektionsweite und Partitionierung	69
5.1	Untere Schranken für die Bisektionsweite	69
5.1.1	Verfahren von Leighton	69
5.1.2	Spektrale untere Schranke	71
5.2	Obere Schranken für die Bisektionsweite	73
	Literatur	87

Kapitel 1

Einleitung

In den letzten Jahren ist der Bedarf an Rechenleistung so stark angestiegen, daß er von sequentiellen Computern nicht mehr erfüllt werden kann. Ein elektrischer Impuls kann sich in Silizium mit einer Geschwindigkeit von 30.000 Kilometern pro Sekunde fortbewegen. Betrachtet man einen Chip mit 3 cm Durchmesser, der zur Verarbeitung einer Operation nur ein Signal benötigt, so kann selbst dieser idealisierte Chip pro Sekunde höchstens 10^9 Operationen verarbeiten. Dies entspricht einer Leistung von einem Giga-Flop. Für die Lösung vieler praxisrelevanter Probleme benötigt man jedoch eine Rechenleistung von einigen hundert Giga-Flops bis zu mehreren Tera-Flops.

Beispielsweise muß für die Strömungssimulation in einem Flugzeugtriebwerk zu jedem Simulationszeitpunkt ein Gleichungssystem mit mehreren 100.000 Unbekannten gelöst werden. In der Wirtschaft benötigt man solch hohe Rechenleistungen, um die Optimierung von komplexen Produktionsprozessen voranzutreiben, und im Bereich der virtuellen Realität müssen zur Echtzeit ganze Räume mit darin enthaltenen 3-dimensionalen Objekten bewegt werden. Die Aufzählung ließe sich beliebig fortsetzen und zeigt den immensen Bedarf an Rechenleistung in allen Anwendungsbereichen der Informatik. Diese Rechenleistung kann nur erreicht werden, wenn mehrere Prozessoren gleichzeitig an der Problemlösung arbeiten. Die Erforschung von Algorithmen für solche Parallelrechner nimmt daher eine Schlüsselposition innerhalb der Informatik ein.

Diese Vorlesung führt in konkrete Parallelrechnerarchitekturen und ihre Algorithmen ein. Sie soll damit neben *Parallele Algorithmen I* einen weiteren Einblick in die Welt der Parallelverarbeitung geben, welche eine der großen Zukunftstechnologien darstellt.

In dem ersten Abschnitt der Einleitung werden die verschiedenen Parallelrechnerarchitekturen kurz vorgestellt und untereinander abgegrenzt. Danach wird anhand der Matrizenmultiplikation gezeigt, wie ein sequentieller Algorithmus in einen parallelen Algorithmus überführt werden kann. Abschnitt 1.3 gibt einen kurzen Überblick über die Inhalte der Vorlesung. Am Schluß der Einleitung findet man eine Liste von Lehrbüchern, die den Stoff der Vorlesung ergänzen.

1.1 Parallelrechnerarchitekturen

In einem Parallelrechner arbeiten mehrere Prozessoren gleichzeitig an der Lösung eines Problems. Daher muß ein Mechanismus bereitgestellt werden, mit dessen Hilfe die Prozessoren untereinander kommunizieren können. Je nachdem, wie dieser Mechanismus realisiert ist, un-

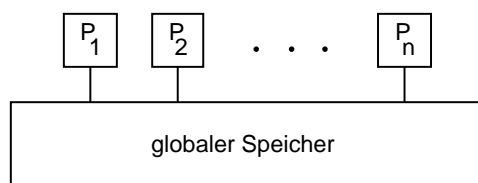


Abbildung 1: *Shared Memory Maschine mit n Prozessoren P_1, \dots, P_n*

terscheidet man zwischen Parallelrechnern, die über einen gemeinsamen Speicher und solchen, die über ein Verbindungsnetzwerk kommunizieren.

Im ersten Fall ist jeder Prozessor, z.B. über einen Bus, an einen großen globalen Speicher angeschlossen. Jeder Prozessor kann auf jede beliebige Speicherzelle lesend oder schreibend zugreifen. Man nennt einen solchen Rechner deshalb auch *Shared Memory Maschine* oder *parallele Registermaschine*. Abbildung 1 zeigt eine Shared Memory Maschine mit n Prozessoren P_1, \dots, P_n . Ein großer Vorteil dieser Architektur besteht darin, daß Anfragen der Prozessoren untereinander direkt über Speicherzugriffe realisiert werden können. Damit verursacht die Kommunikation keine zusätzlichen Kosten – sie ist quasi umsonst. Die Komplexität von Algorithmen kann wie im sequentiellen Fall über die Anzahl der Speicherzugriffe analysiert werden. Ein großer Nachteil der Architektur besteht darin, daß Lese- und Schreibkonflikte auftreten können. Insbesondere für den Fall, daß mehrere Prozessoren gleichzeitig auf dieselbe Speicherzelle schreibend zugreifen, muß genau definiert werden, wie die Shared Memory Maschine den Konflikt auflöst. Physikalisch ist ein solcher Schreibkonflikt nur unter Inkaufnahme erheblicher Effizienzverluste auflösbar.

In einem Prozessornetzwerk (oder kurz Netzwerk) besitzt jeder Prozessor einen eigenen lokalen Speicher. Die Prozessoren sind untereinander durch Kommunikationskanäle verbunden. Man nennt diese Kommunikationskanäle auch *Links*. Prozessornetze werden häufig als ungerichtete Graphen dargestellt. Dabei entspricht jedem Knoten des Graphen ein Prozessor mit seinem lokalen Speicher und jeder Kante des Graphen ein Link. Abbildung 2 zeigt ein Prozessornetzwerk bestehend aus vier Prozessoren P_1, \dots, P_4 mit lokalen Speichern M_1, \dots, M_4 und seine Darstellung als Graph. In der Regel ist es nicht möglich, jeden Prozessor mit jedem anderen durch einen Link zu verbinden. Bei einem Netzwerk bestehend aus n Prozessoren wären hierfür nämlich $\binom{n}{2}$ Leitungen notwendig. In der Praxis beschränkt man sich deshalb auf Netzwerke mit konstantem Grad, d.h. jeder Prozessor ist nur mit konstant vielen anderen Prozessoren direkt durch einen Link verbunden. Hieraus leitet sich unmittelbar ein großer Nachteil der Architektur ab: Möchte ein Prozessor P_i eine Nachricht zu einem Prozessor P_j schicken und sind die beiden Prozessoren nicht durch einen Link verbunden, so muß die Nachricht über alle Prozessoren, die auf einem Weg von P_i nach P_j liegen, geroutet werden. Damit ist die Laufzeit eines Algorithmus entscheidend vom Kommunikationsbedarf und von der Länge der Kommunikationswege abhängig.

Beispiele für real existierende Prozessornetze sind die verschiedenen an der Universität Paderborn existierenden Cluster-Systeme. Bei einem Cluster-System wird Standard-Workstation-Hardware durch ein Hochleistungskommunikationsnetzwerk miteinander verbunden. Beispielsweise besteht der hpcLine-Cluster aus 192 Intel Pentium-III-850 MHz CPUs, die auf 96 Doppelprozessor-Mainboards mit je 512 MByte RAM untergebracht sind. Verbun-

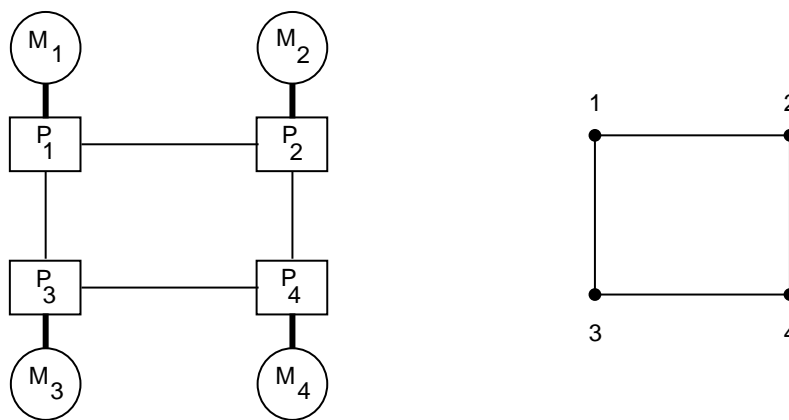


Abbildung 2: Prozessornetzwerk und seine Darstellung als Graph

den werden die Mainboards durch einen 12x8-Torus, der von 96 Dolphin PCI/SCI-Interface-Karten mit einer Linkbandbreite von 500 MByte/s aufgebaut wird. Eine genauere Beschreibung der Architekturen, insbesondere des Nachfolgersystems hpcLine2, findet man auf dem Paderborner WWW-Server unter <http://www.uni-paderborn.de/pc2>.

Parallelrechner können auch dadurch unterschieden werden, ob sie die Instruktionen eines parallelen Programms *synchron* oder *asynchron* abarbeiten. Bei der synchronen Abarbeitung existiert ein globaler Takt, der garantiert, daß die jeweils nächste Instruktion eines Programms von allen Prozessoren gleichzeitig ausgeführt wird. Insbesondere Shared Memory Maschinen arbeiten nach diesem Prinzip. Führen darüberhinaus alle Prozessoren in einem Takt die gleiche Instruktion aus, so spricht man von einer *single instruction multiple data (SIMD)* Architektur. Bei der asynchronen Abarbeitung existiert kein globaler Takt. Werden von den Prozessoren darüberhinaus unterschiedliche Instruktionen abgearbeitet, so spricht man von einer *multiple instruction multiple data (MIMD)* Architektur. Transputersysteme sind typische Vertreter dieser Architektur. Natürlich wären auch asynchrone SIMD Architekturen und synchrone MIMD Architekturen vorstellbar, in der Praxis spielen sie jedoch keine Rolle.

1.2 Parallelisierung von Algorithmen am Beispiel der Matrizenmultiplikation

Viele praxisrelevante Probleme besitzen Lösungsverfahren, die direkt in einen parallelen Algorithmus umgesetzt werden können. Man sagt in diesem Fall, daß das Problem eine natürliche Parallelität besitzt. Ein klassisches Beispiel hierfür ist die Multiplikation zweier Matrizen $A, B \in M(n, \mathbb{R})$. Ist $A = (a_{ij})$ und $B = (b_{ij})$, so sind die Koeffizienten der Matrix $C = A \cdot B$ definiert durch $c_{ij} = \sum_{k=1}^n a_{ik} \cdot b_{kj}$. Ein einfacher sequentieller Algorithmus benötigt zur Berechnung von C $O(n^3)$ Zeiteinheiten. Stehen $1 \leq p \leq n^2$ Prozessoren zur Verfügung, wobei angenommen sei, daß p ein Teiler von n^2 ist, so kann jeder Prozessor $\frac{n^2}{p}$ Koeffizienten berechnen. Dabei wird vorausgesetzt, daß jeder Prozessor auf die Einträge der Matrizen A und B zugreifen kann, wie dies zum Beispiel bei einer Shared Memory Maschine der Fall ist. Da jeder Prozessor nur $\frac{n^2}{p}$ Koeffizienten berechnen muß, und die Berechnung eines Koeffizienten $O(n)$ Zeiteinheiten kostet, beträgt der Gesamtaufwand des parallelen Algorithmus $O(\frac{n^3}{p})$. Im

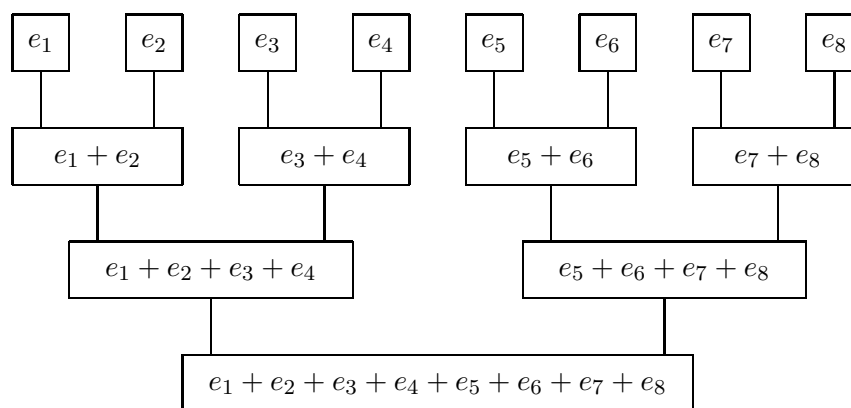


Abbildung 3: Addition von 8 Zahlen nach der Binärbaum Methode

Bereich von $1 \leq p \leq n^2$ Prozessoren hat der parallele Algorithmus einen linearen Speedup, d.h. mit jeder Verdoppelung der Prozessorenanzahl halbiert sich die Laufzeit.

Es stellt sich die Frage, ob und wie bei einem Einsatz von mehr als n^2 Prozessoren die Laufzeit weiter verbessert werden kann. Zur Beantwortung der Frage betrachten wir zunächst die Berechnung der Summe von n reellen Zahlen e_1, \dots, e_n . O.B.d.A sei angenommen, daß n eine Zweierpotenz ist. Addiert man die Zahlen nacheinander von links nach rechts auf, so ergibt sich ein streng sequentielles Verfahren. Die Addition stellt jedoch eine assoziative Operation auf der Menge der reellen Zahlen dar. Daher können die einzelnen Additionen in jeder beliebigen Reihenfolge ausgeführt werden. Abbildung 3 zeigt, wie mit Hilfe eines binären Baumes die Summe von n Zahlen gebildet werden kann. In den Blättern des Baumes sind die Zahlen e_1, \dots, e_n eingetragen. Die innerhalb einer Schicht des Baumes ausgeführten Additionen können parallel abgearbeitet werden. Die Summe von n Zahlen kann dann mit $\frac{n}{2}$ Prozessoren in $O(\log n)$ Zeiteinheiten berechnet werden. Übertragen auf die Matrizenmultiplikation bedeutet dies, daß ein Koeffizient c_{ij} mit $O(n)$ Prozessoren in Zeit $O(\log n)$ berechnet werden kann. Die Matrix C kann also mit $O(n^3)$ Prozessoren in Zeit $O(\log n)$ berechnet werden. Im Bereich von $n^2 < p \leq n^3$ Prozessoren ergibt sich jedoch kein linearer Speedup mehr.

Natürlich kann man nicht erwarten, für jedes Problem so leicht einen effizienten parallelen Algorithmus zu finden, wie das bei der Matrizenmultiplikation der Fall ist. Hier ist es möglich ein sequentielles Lösungsverfahren direkt in ein paralleles zu überführen. In der Regel wird man für die Lösung eines Problems eine ganz neue Vorgehensweise entwickeln müssen, die dann leicht in einen parallelen Algorithmus umgesetzt werden kann. Die Parallelisierung setzt also nicht beim sequentiellen Algorithmus, sondern beim Problem selbst an.

1.3 Inhalt der Vorlesung

Kenntnisse paralleler Algorithmenmodelle sind zum Verständnis der Vorlesung hilfreich, aber nicht zwingend notwendig. Das gängige PRAM-Modell, das man üblicherweise als synchron arbeitend ansieht, wird als Hilfe daher in Kapitel 2 vorgestellt, obwohl es nicht expliziter Teil der Vorlesung ist. Die Vorlesung beginnt mit Kapitel 3. Hier werden die Netzwerke der Hypercube-Familie vorgestellt. Wie eng die Netze miteinander verwandt sind, wird anhand

verschiedener Einbettungen gezeigt. In Kapitel 4 wird eine spezielle Algorithmenklasse – die sogenannten ASCEND/DESCEND Algorithmen – vorgestellt. Zur Veranschaulichung wird ein bitoner Sortieralgorithmus als ASCEND/DESCEND Programm formuliert. Anschließend wird gezeigt, daß ASCEND/DESCEND Algorithmen auf allen Netzwerken der Hypercube-Familie effizient implementiert werden können. Es schließt sich die Beschreibung der Fast-Fourier-Transformation und deren Parallelisierung an. Abschließend wird die Informationsverbreitung in Rechnernetzen durch Broadcasting und Gossiping aus einer mehr theoretischen Sicht untersucht. In Kapitel 5 wird eine wichtige Netzwerkeigenschaft, die Bisektionsweite, genauer untersucht. Es werden obere und untere Schranken für die Bisektionsweite vorgestellt und Anwendungsmöglichkeiten aufgezeigt.

1.4 Literaturangaben und Lehrbücher

Die Erforschung paralleler Algorithmen hat sich zu einem eigenständigen und schnell wachsenden Gebiet der Informatik entwickelt. Neue Forschungsergebnisse werden zunächst auf Konferenzen und danach in Fachzeitschriften der Öffentlichkeit vorgestellt. Im Anschluß an die Vorlesung ist eine Liste von Publikationen angegeben, die den Stoff der Vorlesung vertiefen. In den letzten Jahren sind zudem zahlreiche Lehrbücher erschienen, die sich mit parallelen Algorithmen und Architekturen befassen und für die Vorlesung besonders interessant sind:

1. Pranay Chaudhuri, *Parallel Algorithms, Design and Analysis*, Prentice Hall, 1992.
2. Alan Gibbons, Wojciech Rytter, *Efficient Parallel Algorithms*, Cambridge University Press, 1988.
3. Joseph JáJá, *An Introduction to Parallel Algorithms*, Addison-Wesley, 1992.
4. Marek Karpinski, Wojciech Rytter, *Fast Parallel Algorithms for Graph Matching Problems*, Oxford University Press, 1998
5. V. Kumar, A. Grama, A. Gupta, G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*, Benjamin Cummings Publishing Company, Redwood City, CA, 1994.
6. F.T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann, 1992.
7. F.T. Leighton, *Einführung in Parallele Algorithmen und Architekturen*, Deutsche Übersetzung und Bearbeitung von B. Monien, M. Röttger und U.-P. Schroeder, International Thomson Publishing, 1997.
8. Jan van Leuwen (editor), *Handbook of Theoretical Computer Science, Chapter 17: Parallel Algorithms for Shared-Memory Machines*, Elsevier Science Publishers, 1990.
9. Nancy A. Lynch, *Distributed Algorithms*, Morgan Kaufmann, 1996.
10. Ian Parberry, *Parallel Complexity Theory*, Pitman-Wiley, 1987.
11. M.J. Quinn, *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, 1987.
12. John H. Reif (editor), *Synthesis of Parallel Algorithms*, Morgan Kaufmann, 1993.

Kapitel 2

Die Parallele Random Access Maschine

2.1 Von der RAM zur PRAM

In diesem Abschnitt wird zunächst das bekannteste parallele Rechnermodell, die *Parallel Random Access Machine*, kurz **PRAM** vorgestellt. So wie die RAM ein idealisiertes Modell des von Neumann Rechners darstellt, ist die PRAM ein idealisiertes Modell einer shared memory Maschine. Vergleichbar der RAM verfügt die PRAM über eine abzählbar unendliche Menge von einzeln adressierbaren Speicherzellen. Jede Speicherzelle kann eine beliebig große natürliche Zahl aufnehmen. Darüberhinaus besitzt die PRAM eine unbeschränkte Anzahl von Prozessoren, die von 1 bis n durchnummeriert sind. Jeder Prozessor kann auf jede Speicherzelle in konstanter Zeit zugreifen. Die von einem Prozessor ausgeführten Instruktionen entsprechen dem Befehlsrepertoire einer RAM. Typische Befehle sind ($x_i =$ Inhalt von Speicherzelle i):

$x_i := x_j \otimes x_k$	arithmetische Operationen, $\otimes \in \{+, -, *, /\}$
$x_{x_i} := x_j$	Speichern in indirekt adressierten Speicherzellen
$x_i := x_{x_j}$	Laden von indirekt adressierten Speicherzellen
if $x_i = 0$ then	bedingte Verzweigungen
for $i := r$ to s do	for-Schleifen
while $i \neq r$ do	while-Schleifen

Die Ausführung elementarer Operationen kostet unabhängig von der Größe der involvierten Operanden konstante Zeit (uniformes Kostenmaß). Alle Prozessoren arbeiten synchron dasselbe Programm ab. Zur Formulierung eines parallelen Algorithmus benutzen wir die bekannte *PASCAL*-Syntax, welche um die Definition paralleler Unterprogramme erweitert wird. Es gibt zwei ausgezeichnete Variablen p und N . Die Variable p enthält die Prozessornummer, und die Variable N gibt an, wie viele Prozessoren zur Ausführung des Algorithmus benötigt werden. Jeder Prozessor kennt seine Prozessornummer p . Schematisch läßt sich ein PRAM-Algorithmus wie folgt darstellen:

```

Paralleler Algorithmus <Name: > <PRAM Modell>
Globale Definitionen (Variablen, Funktionen, Prozeduren)
  Par.Proc. <Name: > (p : integer);
  Lokale Definitionen (Variablen, Funktionen, Prozeduren)
  begin
    Programmtext
  end;
begin /* Hauptprogramm */
  N := /* N: Anzahl der benötigten Prozessoren */
  for p := 1 to N parallel do <Name: > (p);
end.

```

Auf die global definierten Variablen können alle Prozessoren zugreifen. Sie heißen deshalb auch *shared variables*. Die lokalen Variablen gehören exklusiv nur einem Prozessor (*private variables*). Jeder Prozessor muß daher Speicherplatz für sie allokalieren. Der Programmtext eines parallelen Unterprogramms könnte beispielsweise wie folgt aussehen:

```

if p ≤ a then
  { Programmteil 1 }
else
  { Programmteil 2 }
x := 0;

```

Alle Prozessoren, deren Nummer kleiner oder gleich a ist, arbeiten Programmteil 1 ab, alle anderen Prozessoren Programmteil 2. Unabhängig von der Länge der beiden Programmteile wird die Zuweisung $x := 0$ von allen Prozessoren gleichzeitig ausgeführt. Die Prozessoren synchronisieren sich also bei jedem Speicherzugriff. Bei dem von Goldschlager in [23] vorgestellten PRAM Modell ist dies nicht der Fall. Dort muß der jeweils kürzere Programmteil durch „leere“ Befehle aufgefüllt werden, damit die Zuweisung von allen Prozessoren gleichzeitig ausgeführt wird.

Das obige Beispiel zeigt, daß die PRAM keine reine SIMD-Architektur ist, denn unterschiedliche Prozessoren können unterschiedliche Programmteile und damit auch unterschiedliche Instruktionen ausführen. In der Literatur wird die PRAM deswegen auch als synchrone *single program multiple data* (SPMD) Architektur bezeichnet.

Die PRAM ist, wie bereits eingangs erwähnt, ein sehr idealisiertes Modell und in der Praxis kaum zu realisieren. Sie ermöglicht jedoch eine Formulierung paralleler Algorithmen frei von einschränkenden Hardware-Restriktionen. In den Mittelpunkt rückt so das Studium des in dem betrachteten Problem enthaltenen Parallelisierungspotentials. Ein weiterer Vorteil besteht in der relativ einfachen Laufzeitanalyse von PRAM-Algorithmen. Wie im sequentiellen Fall wird die Laufzeit bestimmt durch die Anzahl der Speicherzugriffe. Desweiteren ist es möglich, PRAM-Algorithmen auf real existierenden Prozessornetzwerken zu simulieren [1]. Wir beenden den Abschnitt mit einigen Definitionen. Dazu wird im folgenden durch die Abbildungen $p, t : \mathbb{N} \rightarrow \mathbb{N}$ immer die Prozessorenanzahl bzw. die Berechnungszeit einer PRAM bezeichnet.

Definition 2.1 (optimaler PRAM-Algorithmus)

Ein PRAM-Algorithmus heißt optimal, wenn sein Prozessor-Zeit-Produkt $p(n) \cdot t(n)$ bis auf einen konstanten Faktor gleich der Laufzeit des besten sequentiellen Algorithmus ist.

Definition 2.2 (effizienter PRAM-Algorithmus)

Ein PRAM-Algorithmus heißt effizient, wenn er mit $p(n) = n^{O(1)}$ Prozessoren in Zeit $t(n) = (\log n)^{O(1)}$ ausgeführt werden kann.

2.2 Die verschiedenen PRAM Modelle

Je nachdem, ob eine PRAM den gleichzeitigen Lese- bzw. Schreibzugriff mehrerer Prozessoren auf dieselbe Speicherzelle erlaubt oder nicht, können verschiedene PRAM Modelle unterschieden werden.

EREW: *exclusive read, exclusive write*

Bei einer EREW-PRAM sind weder gleichzeitige Lese-, noch gleichzeitige Schreibzugriffe mehrerer Prozessoren auf dieselbe Speicherzelle erlaubt. Tritt während der Programmabarbeitung ein Lese- oder Schreibkonflikt auf, so bricht die Maschine ihre Berechnung ab.

CREW: *concurrent read, exclusive write*

Bei einer CREW-PRAM dürfen verschiedene Prozessoren gleichzeitig auf dieselbe Speicherzelle lesend zugreifen. Ein gleichzeitiger Schreibzugriff ist jedoch nicht erlaubt und führt zum Abbruch der Berechnung.

CRCW: *concurrent read, concurrent write*

Bei einer CRCW-PRAM ist es verschiedenen Prozessoren erlaubt, sowohl lesend als auch schreibend auf dieselbe Speicherzelle zuzugreifen. Auftretende Schreibkonflikte können dabei auf unterschiedliche Arten gelöst werden:

- COMMON-CRCW: Alle Prozessoren, die in eine Speicherzelle schreiben wollen, müssen dasselbe schreiben.
- PRIORITY-CRCW: Der Prozessor mit der kleinsten Nummer darf schreiben.
- ARBITRARY-CRCW: Der Prozessor, der schreiben darf, wird zufällig ausgewählt.

Die einzelnen CRCW-PRAM Modelle sind unterschiedlich mächtig, d.h. es gibt Probleme, die auf einem CRCW-PRAM Modell mit asymptotisch weniger Prozessoren oder mit einer asymptotisch geringeren Laufzeit berechnet werden können, als auf einem anderen CRCW-PRAM Modell. Eine ausführliche Gegenüberstellung der einzelnen CRCW-PRAM Modelle kann man in dem Artikel von Fich, Radge und Widgerson [19] finden.

Wird im folgenden von einer CRCW-PRAM gesprochen, so ist immer eine COMMON-CRCW-PRAM gemeint. Der Vollständigkeit halber sei angemerkt, daß auch eine PRAM vom Typ ERCW (*exclusiv read, concurrent write*) denkbar wäre. Die Realisierung gleichzeitiger Lesezugriffe ist jedoch nicht schwieriger als die Realisierung gleichzeitiger Schreibzugriffe. Aus diesem Grund wird die ERCW-PRAM nicht weiter betrachtet.

Definition 2.3 (PRAM-Klassen)

$CRCW(p(n), t(n))$ bezeichnet die Klasse aller Funktionen, die auf einer CRCW-PRAM mit $O(p(n))$ Prozessoren in Zeit $O(t(n))$ berechenbar sind. Analog definiert man die Klassen $CREW(p(n), t(n))$ und $EREW(p(n), t(n))$.

Bezeichnet man mit $SEQ(t(n))$ die Klasse aller Funktionen, die auf einer RAM in Zeit $O(t(n))$ berechenbar sind, so kann nachfolgender Satz formuliert werden:

Satz 2.1

Für beliebige Funktionen $p, t : \mathbb{N} \rightarrow \mathbb{N}$ gilt:

$$EREW(p(n), t(n)) \subseteq CREW(p(n), t(n)) \subseteq CRCW(p(n), t(n)) \subseteq SEQ(p(n) \cdot t(n))$$

Beweis:

Die ersten zwei Inklusionen folgen unmittelbar aus der wachsenden Mächtigkeit der Maschinenmodelle. Eine sequentielle RAM kann eine CRCW-PRAM simulieren, indem sie die Anweisungen, die die $p(n)$ Prozessoren parallel ausführen, nacheinander ausführt. ■

Die nachfolgenden Beispiele veranschaulichen die unterschiedliche Mächtigkeit der einzelnen Maschinenmodelle. Auf einen formalen Beweis der unteren Schranken soll an dieser Stelle verzichtet werden.

 $EREW(p(n), t(n))$ vs. $CREW(p(n), t(n))$

Die Prozessoren der CREW-PRAM können im Gegensatz zu denen der EREW-PRAM gleichzeitig auf eine Speicherzelle lesend zugreifen. Daher ist es möglich, eine Information von einem Prozessor auf alle Prozessoren in konstanter Zeit $O(1)$ zu verteilen. Die EREW-PRAM benötigt hierfür $\Omega(\log n)$ Zeiteinheiten (untere Schranke).

 $CREW(p(n), t(n))$ vs. $CRCW(p(n), t(n))$

Die CREW-PRAM benötigt zur Feststellung, ob mehrere Prozessoren das gleiche schreiben wollen, $\Omega(\log n)$ Zeiteinheiten (untere Schranke).

Kapitel 3

Netzwerke der Hypercube-Familie

In diesem Kapitel wollen wir einige der bekanntesten und leistungsstärksten Prozessornetzwerke vorstellen. Wie bereits in der Einleitung beschrieben, werden Prozessornetzwerke als ungerichtete Graphen dargestellt. Jedem Knoten des Graphen entspricht ein Prozessor mit seinem lokalen Speicher und jeder Kante ein Kommunikationskanal (Link). Im folgenden werden wir daher die Begriffe Prozessornetzwerk und Graph synonym verwenden.

Möchte ein Prozessor P_i auf eine Variable zugreifen, die in dem lokalen Speicher eines nicht benachbarten Prozessors P_j abgelegt ist, so muß die Variable auf einem möglichst kurzen Weg zu Prozessor P_i geroutet werden. Hieraus leiten sich eine Reihe von Eigenschaften ab, die ein gutes Prozessornetz charakterisieren:

1. kleiner Durchmesser

Um Routing-Wege möglichst kurz zu halten, sollte das Prozessornetz einen kleinen Durchmesser besitzen. Es kann jedoch nicht einfach jeder Prozessor mit jedem anderen Prozessor verbunden werden, da sich ein derartig dichtes Netzwerk in der Praxis nicht realisieren läßt.

2. gute Simulationseigenschaften

Ein Algorithmus, der effizient auf einem Rechnernetz G_1 ausgeführt werden kann, sollte auch auf einem Netzwerk G_2 schnell laufen. Dies gewährleistet die Portierbarkeit von Algorithmen. Dazu muß das Netzwerk G_2 die Kommunikationsstruktur des Netzwerks G_1 effizient simulieren können.

3. einfache Routing-Algorithmen

Falls eine Nachricht von einem Prozessor P_i zu einem Prozessor P_j geroutet werden muß, so sollte jeder Prozessor auf dem Weg von P_i nach P_j möglichst in konstanter Zeit entscheiden, über welche Kante die Nachricht weitergeschickt wird. Dies verringert die Kosten eines nicht lokalen Speicherzugriffs.

4. keine Flaschenhälse

In einem guten Netzwerk sollten nicht nur alle Routing-Wege kurz sein, sie sollten sich auch gleichmäßig über die Knoten und Kanten des Graphen verteilen. Dies ist beispielsweise in einem sternförmigen Netzwerk nicht der Fall.

5. hohe Fehlertoleranz

Kommt ein Netzwerk in der Praxis zum Einsatz, so muß gewährleistet sein, daß auch nach Ausfall eines Prozessors oder eines Links auf dem restlichen Netzwerk weitergearbeitet werden kann. Der Graph sollte eine hohe Konektivität besitzen, so daß selbst bei einem Ausfall mehrerer Links das Netzwerk zusammenhängend bleibt.

In den nachfolgenden Abschnitten werden wir der Reihe nach den *Hypercube*, das *Butterfly* und das *Cube Connected Cycles* Netzwerk, sowie das *DeBruijn* und das *Shuffle Exchange* Netzwerk vorstellen. Diese Netzwerke bilden zusammen die Hypercube-Familie. Zunächst benötigen wir jedoch noch einige graphentheoretische Definitionen:

Definition 3.1 (bipartiter Graph)

Ein ungerichteter Graph $G = (V, E)$ heißt bipartite (paar), falls eine disjunkte Zerlegung $V = V_1 \dot{\cup} V_2$ der Knotenmenge existiert mit $\{u, v\} \in E \Rightarrow u \in V_1, v \in V_2$.

Definition 3.2 (Durchmesser)

Sei $G = (V, E)$ ein ungerichteter Graph und $u, v \in V$. Bezeichne $\text{dist}(u, v)$ die Länge des kürzesten Weges von u nach v . Dann ist der Durchmesser $d(G)$ des Graphen G definiert als:

$$d(G) = \max\{\text{dist}(u, v); u, v \in V\}$$

Definition 3.3 (Partition, Schnitt)

Sei $G = (V, E)$ ein ungerichteter Graph. Dann ist $V_1, V_2 \subset V$ mit $V_1 \dot{\cup} V_2 = V$ eine Partition des Graphen und $\text{Ext}(V_1, V_2) = \{\{u, v\} \in E; u \in V_1, v \in V_2\}$, die Menge der Kanten, die zwischen V_1 und V_2 verlaufen, heißt Schnitt der Partition. $\text{ext}(V_1, V_2) = |\text{Ext}(V_1, V_2)|$ stehe für die Kantenanzahl des Schnitts. Eine Partition V_1, V_2 heißt balanciert, wenn $||V_1| - |V_2|| \leq 1$ gilt.

Definition 3.4 (Bisektionsweite)

Sei $G = (V, E)$ ein ungerichteter Graph. Die Bisektionsweite $\sigma(G)$ ist definiert als:

$$\sigma(G) = \min\{\text{ext}(V_1, V_2); V_1, V_2 \text{ ist eine balancierte Partition von } G\}$$

Definition 3.5 (Isomorphismus, Automorphismus)

Seien $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ ungerichtete Graphen. G_1 und G_2 heißen isomorph, wenn es eine bijektive Abbildung $\varphi : V_1 \rightarrow V_2$ gibt mit $\{u, v\} \in E_1 \Leftrightarrow \{\varphi(u), \varphi(v)\} \in E_2$. In diesem Fall heißt φ Isomorphismus zwischen G_1 und G_2 . Gilt $G_1 = G_2$, so heißt φ Automorphismus.

Definition 3.6 (Knoten-Symmetrie)

Sei $G = (V, E)$ ein ungerichteter Graph. G heißt knotensymmetrisch (vertex-transitiv), wenn für alle $u, v \in V$ ein Automorphismus $\varphi_{u,v}$ existiert mit $\varphi_{u,v}(u) = v$.

Aus der Definition des Automorphismus folgt sofort, daß jeder knotensymmetrische Graph regulär sein muß (d.h. alle Knoten besitzen gleichen Knotengrad).

Definition 3.7 (Kanten-Symmetrie)

Sei $G = (V, E)$ ein ungerichteter Graph. G heißt kantensymmetrisch, falls für je zwei Kanten $\{u, v\}, \{u', v'\} \in E$ ein Automorphismus φ existiert mit $\varphi(u) = u'$ und $\varphi(v) = v'$.

Definition 3.8 (Einbettung)

Seien $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ ungerichtete Graphen. Eine injektive Abbildung $f : V_1 \rightarrow V_2$ heißt Einbettung von G_1 in G_2 .

Im allgemeinen werden benachbarte Knoten aus G_1 nicht auf benachbarte Knoten in G_2 abgebildet. Um eine Einbettung von G_1 nach G_2 vollständig zu beschreiben, muß daher noch für jede Kante $\{u, v\}$ aus G_1 ein Weg $P_f(u, v)$ angegeben werden, der in G_2 die Knoten $f(u)$ und $f(v)$ miteinander verbindet. Dieser Weg muß nicht unbedingt der kürzeste Weg zwischen $f(u)$ und $f(v)$ sein.

Definition 3.9 (Kantenstreckung, Kantenauslastung, Knotenauslastung)

Seien $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ ungerichtete Graphen. Sei f eine Einbettung von G_1 in G_2 . Dann ist die Kantenstreckung $KS(f)$ von f definiert als die maximale Länge eines Weges aus $\{P_f(u, v); \{u, v\} \in E_1\}$. Für eine Kante $e \in E_2$ bezeichne $\rho(e)$ die Anzahl der Wege aus $\{P_f(u, v); \{u, v\} \in E_1\}$, die e enthalten. Dann ist die Kantenauslastung $KA(f)$ von f definiert als $\max\{\rho(e); e \in E_2\}$. Sei $\nu(u)$ mit $u \in V_2$ die Anzahl der Knoten, die aus G_1 unter f auf u abgebildet werden. Die Knotenauslastung $KA(f)$ ist dann definiert als $\max\{\nu(u) \mid u \in V_2\}$.

Kantenstreckung (Dilation) und Kantenauslastung (Congestion) sind wichtige Maße zur Beurteilung der Güte einer Einbettung. Kann G_1 in G_2 mit konstanter Kantenstreckung und konstanter Kantenauslastung eingebettet werden, so ist jeder Algorithmus für G_1 auf G_2 mit konstantem Zeitverlust simulierbar.

Definition 3.10 (Hamilton-Kreis)

Ein einfacher Kreis in einem Graphen $G = (V, E)$, der alle Knoten $v \in V$ durchläuft, heißt *Hamilton-Kreis*.

3.1 Der Hypercube

Der Hypercube gehört zu den leistungsstärksten Verbindungsnetzwerken, die bis heute bekannt sind. Viele andere Netzwerke können von dem Hypercube effizient, d.h. mit konstantem Zeitverlust, simuliert werden. Der Hypercube stellt daher eine vielseitig einsetzbare parallele Maschine dar. In diesem Abschnitt wollen wir den Hypercube und seine wichtigsten Eigenschaften vorstellen.

Definition 3.11 (Hypercube)

Der Hypercube der Dimension k wird mit $Q(k)$ bezeichnet und ist wie folgt definiert:

$$\begin{aligned} Q(k) &= (V_k, E_k) \text{ mit} \\ V_k &= \{0, 1\}^k \\ E_k &= \{\{u, v\}; u, v \in \{0, 1\}^k \text{ unterscheiden sich in genau einem Bit}\} \end{aligned}$$

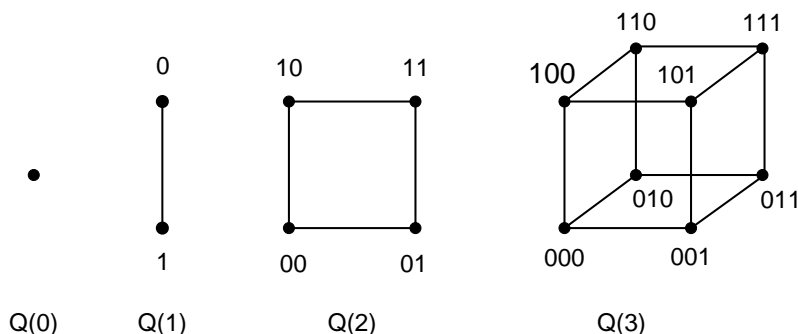


Abbildung 4: Der Hypercube der Dimension 0,1,2 und 3

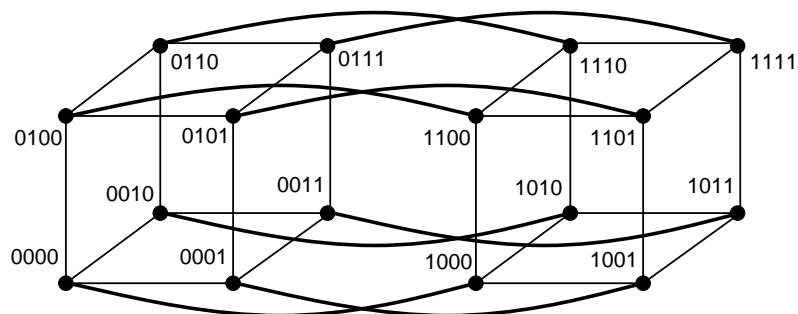


Abbildung 5: Der rekursive Aufbau des Q(4)

Abbildung 4 zeigt den Hypercube der Dimension 0,1,2 und 3. Der Hypercube besitzt eine rekursive Struktur, d.h. der Hypercube der Dimension k , $k \geq 2$, enthält 2 Kopien des Hypercube der Dimension $k - 1$. Abbildung 5 zeigt dies beispielhaft an $Q(4)$. Die hervorgehobenen Kanten verbinden die zwei Kopien des $Q(3)$.

Satz 3.1 (Eigenschaften des Hypercube)

Der Hypercube $Q(k)$ der Dimension k besitzt die folgenden Eigenschaften:

- 1.) $Q(k)$ besitzt 2^k Knoten und $k \cdot 2^{k-1}$ Kanten.
- 2.) $Q(k)$ ist regulär vom Grad k .
- 3.) Der Durchmesser des $Q(k)$ ist k .
- 4.) Die Bisektionsweite des $Q(k)$ ist 2^{k-1} .

Die ersten 3 Aussagen des Satzes sind klar. Der Beweis der vierten Aussage folgt später in Satz 5.2. Für den Hypercube kann ein einfacher Routing-Algorithmus angegeben werden. Seien dazu $u, v \in \{0, 1\}^k$ zwei beliebige Knoten des $Q(k)$ mit $u = (u_{k-1} \dots u_0)$ und $v = (v_{k-1} \dots v_0)$. Desweiteren bezeichne $u(i)$, $0 \leq i < k$, das Binärwort, das sich von u genau im i -ten Bit unterscheidet. Dann kann wie folgt eine Nachricht von u nach v geroutet werden:

```

for  $i := 0$  to  $k - 1$  do begin
    if  $u_i \neq v_i$  then begin

```

schicke Nachricht von Knoten u nach Knoten $u(i)$;
 $u := u(i)$;
 end;
 end;

Die Anzahl der Kanten, die der Routing-Algorithmus durchläuft, entspricht der Anzahl der Bits, in denen sich die Binärdarstellungen der Knoten u und v unterscheiden. Damit findet der Routing-Algorithmus immer den kürzesten Weg zwischen u und v . Der Algorithmus ist also optimal. Im folgenden wollen wir weitere Eigenschaften des Hypercube untersuchen.

Satz 3.2

Der Hypercube $Q(k)$ ist knotensymmetrisch.

Beweis:

Für $x \in \{0, 1\}^k$ bezeichne wieder $x(i)$, $0 \leq i < k$, das Binärwort, das sich von x genau im i -ten Bit unterscheidet. Sei $\delta_i : V_k \mapsto V_k$ definiert durch $\delta_i(x) = x(i)$. Dann ist δ_i ein Automorphismus, denn für $\{u, v\} \in E_k$ mit $v = u(j)$, $0 \leq j < k$, gilt:

$$\{\delta_i(u), \delta_i(u(j))\} = \begin{cases} \{u(i), u(i)(j)\}, & \text{falls } i \neq j \\ \{u(i), u\}, & \text{falls } i = j \end{cases}$$

In beiden Fällen ist $\{\delta_i(u), \delta_i(u(j))\}$ eine Kante des $Q(k)$. Da die Konkatenation von Automorphismen selbst wieder einen Automorphismus ergibt, kann der gesuchte Automorphismus $\varphi_{u,v}$ mit $\varphi_{u,v}(u) = v$ definiert werden durch:

$$\varphi_{u,v} = \prod_{\substack{i \\ u_i \neq v_i}} \delta_i$$

■

Satz 3.3

Der Hypercube $Q(k)$ ist kantensymmetrisch.

Beweis:

Seien $\{u, v\}, \{u', v'\}$ zwei Kanten des $Q(k)$. Sei i die Dimension, in der $\{u, v\}$ verläuft, d.h.

$$\begin{aligned} u &= u_{k-1} \dots u_i \dots u_0 \\ v &= u_{k-1} \dots \bar{u}_i \dots u_0 \end{aligned}$$

und sei j die Dimension, in der $\{u', v'\}$ verläuft, d.h.

$$\begin{aligned} u' &= u'_{k-1} \dots u'_j \dots u'_0 \\ v' &= u'_{k-1} \dots \bar{u}'_j \dots u'_0 \end{aligned}$$

Sei $\pi : \{0, \dots, k-1\} \rightarrow \{0, \dots, k-1\}$ eine Permutation mit $\pi(j) = i$ und bezeichne \oplus die logische XOR-Verknüpfung. Dann kann der gesuchte Automorphismus $\varphi : V_k \rightarrow V_k$ definiert werden durch:

$$\varphi(x_{k-1} \dots x_0) = ((x_{\pi(k-1)} \oplus u_{\pi(k-1)} \oplus u'_{k-1}) \dots (x_{\pi(0)} \oplus u_{\pi(0)} \oplus u'_0))$$

Wir müssen zeigen:

- (i) $\varphi(u) = u'$ und $\varphi(v) = v'$
- (ii) φ ist Automorphismus, d.h. $\{z_1, z_2\} \in E_k \Leftrightarrow \{\varphi(z_1), \varphi(z_2)\} \in E_k$

ad(i):

Für $\varphi(u)$ ergibt sich:

$$\begin{aligned} \varphi(u_{k-1} \dots u_0) &= ((u_{\pi(k-1)} \oplus u_{\pi(k-1)} \oplus u'_{k-1}) \dots (u_{\pi(0)} \oplus u_{\pi(0)} \oplus u'_0)) \\ &= u'_{k-1} \dots u'_0 = u' \end{aligned}$$

und $\varphi(v)$ berechnet sich zu:

$$\varphi(v_{k-1} \dots v_0) = ((v_{\pi(k-1)} \oplus u_{\pi(k-1)} \oplus u'_{k-1}) \dots (v_{\pi(0)} \oplus u_{\pi(0)} \oplus u'_0))$$

dabei gilt für $0 \leq h < k$:

$$v_{\pi(h)} = \begin{cases} u_{\pi(h)}, & \text{falls } \pi(h) \neq i, \text{ d.h. } h \neq j \\ \bar{u}_i, & \text{falls } h = j \end{cases}$$

daraus folgt: $\varphi(v_{k-1} \dots v_0)$

$$\begin{aligned} &= ((u_{\pi(k-1)} \oplus u_{\pi(k-1)} \oplus u'_{k-1}) \dots (\bar{u}_i \oplus u_i \oplus u'_j) \dots (u_{\pi(0)} \oplus u_{\pi(0)} \oplus u'_0)) \\ &= u'_{k-1} \dots \bar{u}'_j \dots u'_0 = v' \end{aligned}$$

ad (ii):

Sei $\{z_1, z_2\}$ eine Kante des $Q(k)$. Sei r das Bit, in dem sich die Binärworte z_1 und z_2 unterscheiden. Sei $\pi(s) = r$. Nach (i) unterscheiden sich $\varphi(z_1)$ und $\varphi(z_2)$ im Bit s . Daraus folgt: $\{\varphi(z_1), \varphi(z_2)\} \in E_k$. ■

Lemma 3.1

Der Hypercube $Q(k)$, $k \geq 2$, enthält einen Hamilton-Kreis.

Beweis: (durch Induktion nach k)

Ind. Anf.: $k = 2$

Der gesuchte Hamilton-Kreis lautet: $00 \rightarrow 01 \rightarrow 11 \rightarrow 10 \rightarrow 00$.

Ind. Schritt: $k \mapsto k + 1$

Der Hypercube $Q(k + 1)$ ist aus zwei disjunkten Kopien des $Q(k)$ zusammengesetzt. Nach Ind.Vor. enthalten beide Kopien einen Hamilton-Kreis C_k . Durch Verschmelzen der beiden Kreise kann ein Hamilton-Kreis C_{k+1} in $Q(k + 1)$ konstruiert werden. Sei dazu $\{u, v\} \in E_k$ eine beliebige Kante auf C_k . Der Kreis C_{k+1} wird wie folgt gebildet:

- 1.) Streichen der Kanten $\{0u, 0v\}, \{1u, 1v\}$
- 2.) Hinzufügen der Kanten $\{0u, 1u\}, \{0v, 1v\}$

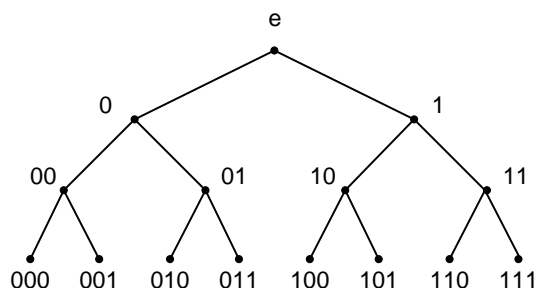


Abbildung 6: Der vollständige binäre Baum der Höhe 3

■

Eine weitere Möglichkeit zur Generierung von Hamilton-Kreisen im Hypercube besteht in der Benutzung von *Gray-Codes*. Ein Gray-Code besteht aus einer Folge von Binärwörtern, wobei sich zwei aufeinanderfolgende Worte in genau einem Bit unterscheiden. Mit Hilfe des nachfolgend beschriebenen Verfahrens kann ein Gray-Code und damit ein Hamilton-Kreis im Hypercube bestimmt werden.

Sei dazu $g : \{0, 1\}^k \rightarrow \{0, 1\}^k$ eine Funktion, die eine Binärzahl $x_{k-1} \dots x_0$ abbildet auf $y_{k-1} \dots y_0$ mit

$$\begin{aligned} y_{k-1} &= x_{k-1} \\ y_j &= x_j \oplus x_{j+1}, \quad 0 \leq j < k-1 \end{aligned}$$

Bezeichne $(z)_2, z \in \{0, \dots, 2^k - 1\}$, die k -stellige Binärdarstellung der Zahl z . Dann ist $(g((0)_2), g((1)_2), \dots, g((2^k - 1)_2))$ ein Gray-Code der Länge 2^k und damit ein Hamilton-Kreis in $Q(k)$. Sei beispielsweise $k = 3$. Dann berechnet sich $(g((0)_2), \dots, g((7)_2))$ zu:

$$\begin{aligned} g((0)_2) &= g(000) = 000 & g((4)_2) &= g(100) = 110 \\ g((1)_2) &= g(001) = 001 & g((5)_2) &= g(101) = 111 \\ g((2)_2) &= g(010) = 011 & g((6)_2) &= g(110) = 101 \\ g((3)_2) &= g(011) = 010 & g((7)_2) &= g(111) = 100 \end{aligned}$$

Die Folge $(000, 001, 011, 010, 110, 111, 101, 100)$ ist ein Hamilton-Kreis in $Q(3)$. Im folgenden wollen wir anhand zweier Beispiele die guten Simulationseigenschaften des Hypercube aufzeigen. Wir betrachten dazu den vollständigen binären Baum der Höhe k , bestehend aus $2^{k+1} - 1$ Knoten, und das d -dimensionale Gitter. Wir untersuchen zunächst, wie der vollständige binäre Baum der Höhe k in den $Q(k+1)$ eingebettet werden kann (vgl. auch [31]).

Definition 3.12 (vollständiger binärer Baum der Höhe k)

Der vollständige binäre Baum der Höhe k wird mit $B(k)$ bezeichnet. Seine Knoten sind binäre Zeichenketten der Länge $\leq k$. Seine Kanten verbinden Zeichenketten u der Länge $i, 0 \leq i < k$, mit Zeichenketten $ua, a \in \{0, 1\}$, der Länge $i+1$. Die Wurzel des Baumes ist mit dem „leeren“ Wort ϵ beschriftet.

Abbildung 6 zeigt den vollständigen binären Baum der Höhe 3.

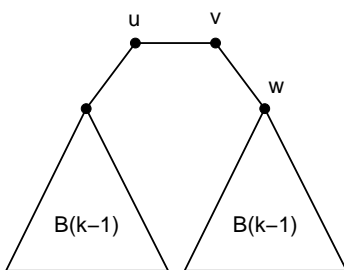


Abbildung 7: Der Doppelwurzelbaum der Höhe k

Satz 3.4

Für $k \geq 2$ ist $B(k)$ kein Teilgraph des $Q(k + 1)$.

Beweis:

Ann.: $B(k)$ ist Teilgraph des $Q(k + 1)$

Die Knotenmenge V des $Q(k + 1)$ kann wie folgt in zwei disjunkte Teilmengen aufgeteilt werden:

$$V_1 = \{v \in V; \text{ die Anzahl der Einsen in } v \text{ ist gerade}\}$$

$$V_2 = \{v \in V; \text{ die Anzahl der Einsen in } v \text{ ist ungerade}\}$$

Dabei gilt: $|V_1| = |V_2| = \frac{1}{2} \cdot |V|$. Die Knoten aus V_1 heißen *gerade Knoten* und die aus V_2 *ungerade Knoten*. O.B.d.A. sei angenommen, daß die Wurzel ϵ des $B(k)$ auf einen geraden Knoten des $Q(k + 1)$ abgebildet wird. Dann müssen die Söhne von ϵ auf ungerade Knoten abgebildet werden. Allgemein gilt also, daß alle Knoten einer Schicht entweder vollständig auf gerade oder auf ungerade Knoten des $Q(k + 1)$ abgebildet werden. Insbesondere werden die 2^k Knoten in Schicht k (also die Blätter des Baumes) und die 2^{k-2} Knoten in Schicht $k - 2$ entweder auf gerade oder auf ungerade Knoten des $Q(k + 1)$ abgebildet. Wegen $2^k + 2^{k-2} > 2^k = \frac{1}{2} \cdot |V|$ für $k \geq 2$, ist dies jedoch nicht möglich.

Fazit: $B(k)$ ist kein Teilgraph des $Q(k + 1)$

■

Definition 3.13 (Doppelwurzelbaum)

Der Doppelwurzelbaum der Höhe k , kurz $DWB(k)$, ist ein $B(k)$, dessen Wurzel durch einen Weg der Länge 2 ersetzt wurde. Damit besitzt $DWB(k)$ genau 2^{k+1} Knoten.

Abbildung 7 veranschaulicht die Definition.

Satz 3.5

$DWB(k)$ ist Teilgraph des $Q(k + 1)$.

Beweis: (durch Induktion nach k)

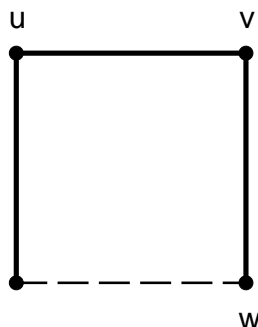


Abbildung 8: Einbettung des DWB(1) in $Q(2)$

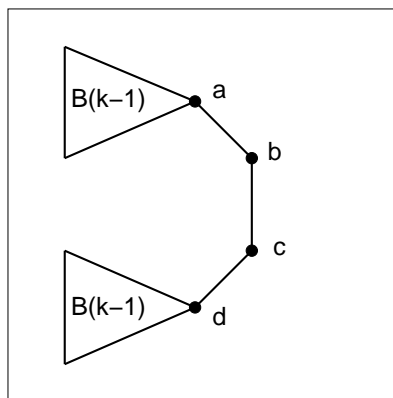


Abbildung 9: Einbettung des DWB(k) in $Q(k + 1)$

Ind. Anf.: $k = 1$

DWB(1) ist ein Weg bestehend aus vier Knoten. Dieser ist in $Q(2)$ enthalten (vgl. Abbildung 8).

Ind. Schritt: $k \mapsto k + 1$

Wir betrachten die Einbettung des DWB(k) in $Q(k + 1)$ (vgl. Abbildung 9). O. B. d. A. können die Knoten b, c und d durch folgende Bitdarstellung im Hypercube repräsentiert werden:

$$\begin{aligned} b &= 000^{k-1} \\ c &= 010^{k-1} \\ d &= 110^{k-1} \end{aligned}$$

Der Grund hierfür ist folgender: b und c können wegen der Kantensymmetrie so gewählt werden. d ist dann ein Nachbar von $c \neq b$, d.h. $d = c(j)$ mit $j \neq k$. Anschließend tauscht man noch Bit j mit Bit k+1 und erhält d.

Wir wollen nun DWB(k + 1) in $Q(k + 2)$ einbetten. Dazu benutzen wir die Induktionsvoraussetzung, nämlich daß es für beide Teilcubes der Dimension $k + 1$ einen Teilgraph DWB(k) gibt. Sei die Einbettung des DWB(k) in den linken Teilcube so wie oben definiert. Die Ein-

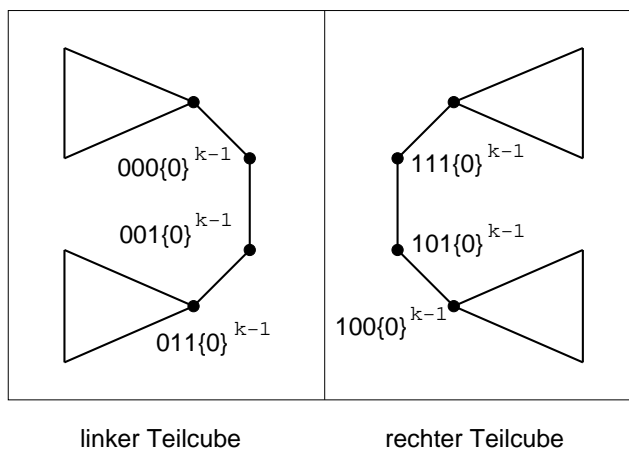


Abbildung 10: Einbettung des $DWB(k)$ in linken und in rechten Teilcube des $Q(k+2)$

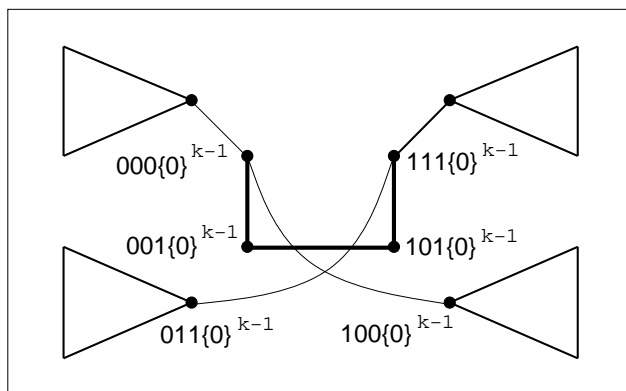


Abbildung 11: $DWB(k+1)$ -Teilgraph im $Q(k+2)$

bettung in den rechten Teilcube erhalten wir durch folgenden Automorphismus:

$$\varphi(x_k \dots x_0) = \bar{x}_{k-1} \bar{x}_k x_{k-2} \dots x_0$$

Es ergeben sich folgende Transformationen durch φ :

$$\begin{aligned} 00 \dots &\rightarrow 11 \dots \\ 01 \dots &\rightarrow 01 \dots \\ 10 \dots &\rightarrow 10 \dots \\ 11 \dots &\rightarrow 00 \dots \end{aligned}$$

Wir setzen nun vor die $(k+1)$ -stellige Bitdarstellung der Knoten des linken Teilcubes eine 0, vor die Knoten des rechten Teilcubes eine 1. Somit erhalten wir für jeden Knoten des $Q(k+2)$ eine zulässige $(k+2)$ -stellige Bitdarstellung (siehe Abbildung 10). Verbinden wir dann die drei ausgezeichneten Knoten des linken Teilcubes mit ihrem jeweiligen Nachbar im rechten Teilcube, so erhalten wir einen Doppelwurzelbaum der Höhe $k+1$ (siehe Abbildung 11).

■

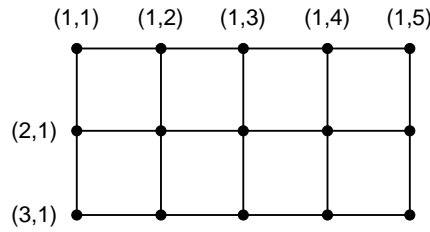


Abbildung 12: Das 2-dimensionale Gitter $M[3, 5]$

Korollar 3.1

$B(k)$ ist Teilgraph des $Q(k + 2)$, denn $B(k) \subset DWB(k + 1) \subset Q(k + 2)$.

Korollar 3.2

$B(k)$ lässt sich in $Q(k + 1)$ mit Kantenstreckung 2 und Kantenauslastung 1 einbetten. Dabei wird nur eine Kante auf die Länge 2 gestreckt.

Im folgenden werden einige Ergebnisse bezüglich der Einbettung eines d -dimensionalen Gitters in den Hypercube vorgestellt. Das d -dimensionale Gitter ist wie folgt definiert:

Definition 3.14 (d -dimensionales Gitter)

Das d -dimensionale Gitter wird mit $M[n_1, \dots, n_d]$ bezeichnet und ist wie folgt definiert:

$$M[n_1, \dots, n_d] = (V, E) \text{ mit}$$

$$V = \{(i_1, \dots, i_d); 1 \leq i_\nu \leq n_\nu \text{ f\u00fcr } 1 \leq \nu \leq d\}$$

$$E = \{(i_1, \dots, i_d), (j_1, \dots, j_d)\}; \exists 1 \leq \kappa \leq d \text{ mit } j_\kappa = i_\kappa + 1 \text{ und } i_\nu = j_\nu \text{ f\u00fcr } \nu \neq \kappa\}$$

Abbildung 12 zeigt das 2-dimensionales Gitter $M[3, 5]$.

Satz 3.6

- 1.) $M[n_1, \dots, n_d]$ ist Teilgraph des $Q(\sum_{i=1}^d \lceil \log n_i \rceil)$
- 2.) $M[n_1, \dots, n_d]$ ist Teilgraph des $Q(\lceil \log(n_1 \cdot \dots \cdot n_d) \rceil) \Leftrightarrow \sum_{i=1}^d \lceil \log n_i \rceil = \lceil \log(n_1 \cdot \dots \cdot n_d) \rceil$

Beweis:

O.B.d.A. sei $d = 2$.

ad 1.)

Sei $k_1 = \lceil \log n_1 \rceil$ und $k_2 = \lceil \log n_2 \rceil$. Seien $\alpha(i)$, $1 \leq i \leq 2^{k_1}$, und $\beta(j)$, $1 \leq j \leq 2^{k_2}$, Hamilton-Wege in $Q(k_1)$ bzw. $Q(k_2)$. Ein Knoten (i, j) des 2-dimensionalen Gitters $M[n_1, n_2]$, $1 \leq i \leq n_1, 1 \leq j \leq n_2$, wird abgebildet auf den Knoten $\alpha(i) \circ \beta(j)$ des $Q(k_1 + k_2)$ (im folgenden sei durch \circ die Konkatenation zweier Bitstrings bezeichnet). Die beschriebene Abbildung ist injektiv. Eine Kante $\{(i, j), (i + 1, j)\}$ des Gitters wird abgebildet auf die Kante $\{\alpha(i) \circ \beta(j), \alpha(i + 1) \circ \beta(j)\}$ des Hypercube und eine Kante $\{(i, j), (i, j + 1)\}$ auf die Kante $\{\alpha(i) \circ \beta(j), \alpha(i) \circ \beta(j + 1)\}$.

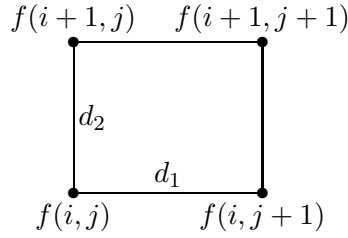


Abbildung 13: Das Bild $f(i + 1, j + 1)$ ist eindeutig bestimmt

ad 2.)

Die Richtung „ \Leftarrow “ folgt unmittelbar aus Teil 1. Wir müssen daher nur noch „ \Rightarrow “ zeigen. Sei dazu f eine Einbettung des $M[n_1, n_2]$ in den $Q(\lceil \log(n_1 \cdot n_2) \rceil)$ mit Kantenstreckung 1. Dann ist für jedes i, j mit $1 \leq i \leq n_1$ und $1 \leq j \leq n_2$ das Bild $f(i + 1, j + 1)$ durch Angabe von $f(i, j), f(i + 1, j), f(i, j + 1)$ eindeutig bestimmt. Dies sieht man wie folgt:

Bezeichne d_1 das Bit, in dem sich $f(i, j)$ und $f(i, j + 1)$ unterscheiden und d_2 das Bit, in dem sich $f(i, j)$ und $f(i + 1, j)$ unterscheiden (vgl. Abbildung 13). Da $f(i + 1, j + 1)$ adjazent sein muß zu $f(i + 1, j)$ und $f(i, j + 1)$, darf sich $f(i + 1, j + 1)$ von $f(i, j)$ nur in den Bits d_1 und d_2 unterscheiden. Daraus folgt: Die Einbettung f ist durch $f(1, j), 1 \leq j \leq n_2$, und $f(i, 1), 1 \leq i \leq n_1$, eindeutig festgelegt. Sei nun d_i das Bit, in dem sich $f(i, 1)$ und $f(i + 1, 1)$ unterscheiden und e_j das Bit, in dem sich $f(1, j)$ und $f(1, j + 1)$ unterscheiden. Sei $I_1 = \{d_i; 1 \leq i < n_1\}$ und $I_2 = \{e_j; 1 \leq j < n_2\}$. Dann gilt: $I_1 \cap I_2 = \emptyset$.

Ann.: $d_k = e_l \in I_1 \cap I_2$

Dann gilt:

$$\begin{aligned} f(k, y) \text{ und } f(k + 1, y) \text{ unterscheiden sich im Bit } d_k, 1 \leq y \leq n_2 \\ f(x, l) \text{ und } f(x, l + 1) \text{ unterscheiden sich im Bit } e_l = d_k, 1 \leq x \leq n_1 \end{aligned}$$

Für $x = k$ und $y = l$ folgt dann: $f(k + 1, l) = f(k, l + 1) \Rightarrow$ Widerspruch zur Injektivität von f .

Fazit: $I_1 \cap I_2 = \emptyset$

Für die Einbettung f werden also $|I_1| + |I_2|$ Bits benötigt. Da in $Q(\lceil \log(n_1 \cdot n_2) \rceil)$ die Adresse jedes Knotens nur aus $\lceil \log(n_1 \cdot n_2) \rceil$ Bits besteht, folgt: $|I_1| + |I_2| \leq \lceil \log(n_1 \cdot n_2) \rceil$. Weiter gilt: $n_1 \leq 2^{|I_1|}, n_2 \leq 2^{|I_2|}$ und damit $\lceil \log n_1 \rceil \leq |I_1|, \lceil \log n_2 \rceil \leq |I_2|$.

Insgesamt folgt: $\lceil \log n_1 \rceil + \lceil \log n_2 \rceil \leq |I_1| + |I_2| \leq \lceil \log(n_1 \cdot n_2) \rceil$

■

Satz 3.6 besagt, daß $Q(\sum_{i=1}^d \lceil \log n_i \rceil)$ der kleinste Hypercube ist, der $M[n_1, \dots, n_d]$ als Teilgraph enthält (vgl. [35]). Nach [9] können alle 2-dimensionalen Gitter $M[n_1, n_2]$ in $Q(\lceil \log(n_1 \cdot n_2) \rceil)$ mit Kantenstreckung ≤ 2 und alle d -dimensionalen Gitter $M[n_1, \dots, n_d], d \geq 3$, in $Q(\lceil \log(n_1 \cdot \dots \cdot n_d) \rceil)$ mit Kantenstreckung $\leq 4d + 1$ eingebettet werden. Für $d = 3$ kann bezüglich der Kantenstreckung eine schärfere obere Schranke angegeben werden. Nach [5] ist es möglich, jedes 3-dimensionale Gitter $M[n_1, n_2, n_3]$ in $Q(\lceil \log(n_1 \cdot n_2 \cdot n_3) \rceil)$ mit Kantenstreckung ≤ 5 einzubetten. Gilt zusätzlich $n_1 \cdot n_2 \cdot n_3 \leq 2^9 - 18$ so kann nach [33] das 3-dimensionale Gitter mit Kantenstreckung ≤ 2 eingebettet werden.

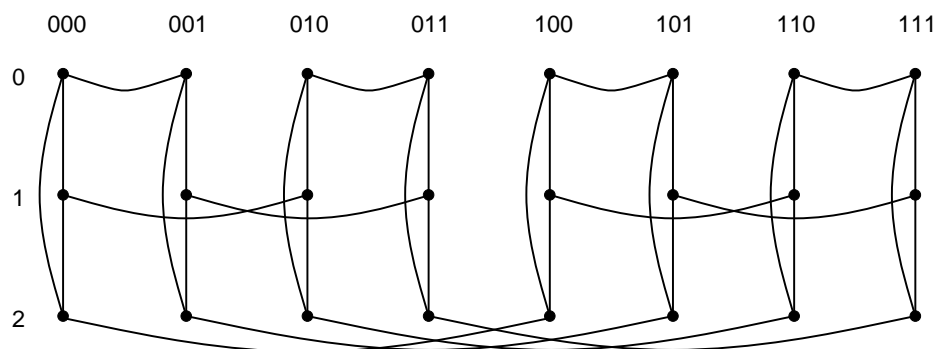


Abbildung 14: Das Cube-Connected-Cycles Netzwerk der Dimension 3

3.2 Das Butterfly und das Cube-Connected-Cycles Netzwerk

Ein Nachteil der Hypercube-Architektur besteht darin, daß mit wachsender Knotenzahl auch der Knotengrad zunimmt. So kann beispielsweise auf dem frei konfigurierbaren Parallelrechner SC320 höchstens ein Hypercube der Dimension vier geschaltet werden. Man kann dieses Problem umgehen, indem jeder Knoten des $Q(k)$ durch einen Kreis der Länge k ersetzt wird. Man erhält so das Cube-Connected-Cycles Netzwerk, das in jeder Dimension den konstanten Knotengrad 3 besitzt. Das Netzwerk kann formal wie folgt definiert werden:

Definition 3.15 (Cube-Connected-Cycles Netzwerk)

Das Cube-Connected-Cycles Netzwerk der Dimension k wird mit $CCC(k)$ bezeichnet und ist wie folgt definiert:

$$\begin{aligned}
 CCC(k) &= (V_k, E_k) \text{ mit} \\
 V_k &= \{(i, u); 0 \leq i < k, u \in \{0, 1\}^k\} \\
 E_k &= \{ \{(i, u), ((i + 1) \bmod k, u)\}; 0 \leq i < k, u \in \{0, 1\}^k \} \\
 &\quad \cup \{ \{(i, u), (i, u(i))\}; 0 \leq i < k, u \in \{0, 1\}^k \}
 \end{aligned}$$

Dabei bezeichnet wieder $u(i)$ das Binärwort, das sich von u genau im i -ten Bit unterscheidet. Die Kanten der ersten Menge heißen *Kreiskanten* und die Kanten der zweiten Menge werden *Hypercube-Kanten* genannt.

Abbildung 14 zeigt das Cube-Connected-Cycles Netzwerk der Dimension 3.

Satz 3.7 (Eigenschaften des Cube-Connected-Cycles Netzwerks)

Das Cube-Connected-Cycles Netzwerk der Dimension k besitzt die folgenden Eigenschaften:

- 1.) $CCC(k)$ besitzt $k \cdot 2^k$ Knoten und $3 \cdot k \cdot 2^{k-1}$ Kanten.
- 2.) $CCC(k)$ ist regulär vom Grad 3.
- 3.) Der Durchmesser des $CCC(k)$ ist $2k - 2 + \lfloor \frac{k}{2} \rfloor = \lfloor \frac{5k}{2} \rfloor - 2$ für $k > 3$.
- 4.) Die Bisektionsweite des $CCC(k)$ ist 2^{k-1} .

Die ersten zwei Aussagen des Satzes sind klar. Der Beweis der dritten Aussage sei dem Leser als Übungsaufgabe empfohlen. Auf den Beweis der 4. Aussage sei an dieser Stelle verzichtet.

Für das Cube-Connected-Cycles Netzwerk kann ebenfalls ein einfacher Routing-Algorithmus angegeben werden. Seien dazu $(i, u), (j, v)$ zwei beliebige Knoten des $\text{CCC}(k)$. Dann kann wie folgt eine Nachricht von (i, u) nach (j, v) geroutet werden:

```

repeat  $k$  times
  if  $u_i \neq v_i$  then begin                               /* laufe Hypercube-Kante entlang */
    sende Nachricht von  $(i, u)$  nach  $(i, u(i))$ ;
     $u := u(i)$ ;
  end;
  if  $u \neq v$  then begin                                   /* laufe Kreiskante entlang */
    sende Nachricht von  $(i, u)$  nach  $((i + 1) \bmod k, u)$ ;
     $i := (i + 1) \bmod k$ ;
  end;
end of repeat;
laufe im Kreis des Knotens  $v$  auf kürzestem Weg bis zum entsprechenden  $j$ ;

```

Der Algorithmus zeigt, daß für zwei Knoten $(i, u), (j, v)$ des $\text{CCC}(k)$ gilt: $\text{dist}((i, u), (j, v)) \leq 2k - 1 + \lfloor \frac{k}{2} \rfloor$. Der Routing-Algorithmus ist jedoch nicht optimal, denn beispielsweise kann der Knoten $(2, 111)$ des $\text{CCC}(3)$ von dem Knoten $(0, 011)$ über einen Weg der Länge 2 erreicht werden (vgl. Abbildung 14). Dieser optimale Weg wird von dem Algorithmus nicht gefunden.

Satz 3.8

Das Cube-Connected-Cycles Netzwerk $\text{CCC}(k)$ ist knotensymmetrisch.

Beweis:

Für $x \in \{0, 1\}^k$, $x = x_{k-1}x_{k-2} \dots x_0$, definieren wir die Funktion $s : \{0, 1\}^k \rightarrow \{0, 1\}^k$ durch $s(x_{k-1}x_{k-2} \dots x_0) = x_{k-2} \dots x_0x_{k-1}$. Die Funktion s beschreibt also einen zyklischen Linksshift der binären Zeichenkette x um eine Stelle. Man nennt s auch *Shuffle-Funktion*. Bezeichne $V_k = \{(i, u); 0 \leq i < k, u \in \{0, 1\}^k\}$ die Knotenmenge des $\text{CCC}(k)$. Für einen Knoten $(i, u) \in V_k$ definieren wir die Funktion $\psi : V_k \rightarrow V_k$ durch $(i, u) \mapsto ((i + 1) \bmod k, s(u))$. Die Funktion ψ ist ein Automorphismus, der die Kreiskante $\{(i, u), ((i + 1) \bmod k, s(u))\}$ abbildet auf die Kreiskante $\{((i + 1) \bmod k, s(u)), ((i + 2) \bmod k, s(u))\}$ und die Hypercube-Kante $\{(i, u), (i, u(i))\}$ auf die Hypercube-Kante $\{((i + 1) \bmod k, s(u)), ((i + 1) \bmod k, s(u(i)))\}$. Darüberhinaus definieren wir die Funktion $\varphi_j : V_k \rightarrow V_k$ durch $(i, u) \mapsto (i, u(j))$. Man sieht leicht, daß auch φ_j ein Automorphismus ist. Durch eine geeignete Konkatenation beider Automorphismen kann ein Knoten (i, u) auf jeden beliebigen Knoten (j, v) des $\text{CCC}(k)$ abgebildet werden. ■

Ersetzt man in dem Cube-Connected-Cycles Netzwerk jeden Weg, der über eine Hypercube- und eine Kreiskante führt durch eine direkte Verbindung und eine Kreiskante, so erhält man das Butterfly Netzwerk.

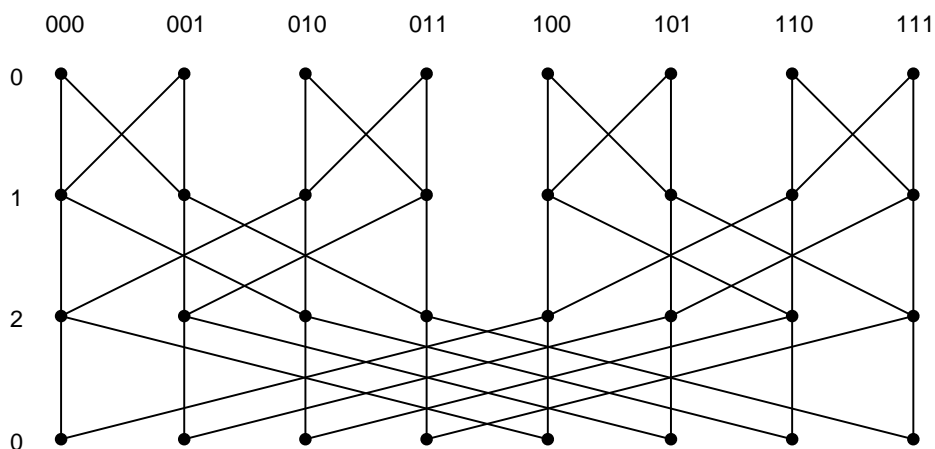


Abbildung 15: Das Butterfly Netzwerk der Dimension 3

Definition 3.16 (Butterfly Netzwerk)

Das Butterfly Netzwerk der Dimension k wird mit $\text{BF}(k)$ bezeichnet und ist wie folgt definiert:

$$\begin{aligned} \text{BF}(k) &= (V_k, E_k) \text{ mit} \\ V_k &= \{(i, u); 0 \leq i < k, u \in \{0, 1\}^k\} \\ E_k &= \{ \{(i, u), ((i + 1) \bmod k, u)\}; 0 \leq i < k, u \in \{0, 1\}^k\} \\ &\quad \cup \{ \{(i, u), ((i + 1) \bmod k, u(i))\}; 0 \leq i < k, u \in \{0, 1\}^k\} \end{aligned}$$

Die Kanten der ersten Menge heißen wieder *Kreiskanten* und die Kanten der zweiten Menge werden *Kreuzkanten* genannt.

Wie in Abbildung 15 dargestellt, wird Schicht 0 des Butterfly Netzwerks doppelt dargestellt. Dies erleichtert das Einzeichnen der Kreuzkanten von Schicht $k - 1$ nach Schicht 0.

Satz 3.9 (Eigenschaften des Butterfly Netzwerks)

Das Butterfly Netzwerk der Dimension k besitzt die folgenden Eigenschaften:

- 1.) $\text{BF}(k)$ besitzt $k \cdot 2^k$ Knoten und $4 \cdot k \cdot 2^{k-1} = k \cdot 2^{k+1}$ Kanten.
- 2.) $\text{BF}(k)$ ist regulär vom Grad 4.
- 3.) Der Durchmesser des $\text{BF}(k)$ ist $k + \lfloor \frac{k}{2} \rfloor = \lfloor \frac{3k}{2} \rfloor$.
- 4.) Die Bisektionsweite des $\text{BF}(k)$ ist 2^k .

Auch hier ist lediglich der Beweis der vierten Aussage schwierig. Wir verzichten auf einen Beweis und wenden uns direkt dem Routing-Problem im Butterfly Netzwerk zu. Seien (i, u) , (j, v) zwei beliebige Knoten des $\text{BF}(k)$. Dann kann wie folgt eine Nachricht von (i, u) nach (j, v) geroutet werden:

```
repeat k times
  if  $u_i \neq v_i$  then begin
    /* laufe Kreuzkante entlang */
    sende Nachricht von  $(i, u)$  nach  $((i + 1) \bmod k, u(i))$ ;
     $i := (i + 1) \bmod k$ ;  $u := u(i)$ ;
```

```

end
else begin
    sende Nachricht von  $(i, u)$  nach  $((i + 1) \bmod k, u)$ ;
     $i := (i + 1) \bmod k$ ;
end;
end of repeat;
laufe im Kreis des Knotens  $v$  auf kürzestem Weg bis zum entsprechenden  $j$ ;

```

Auch dieser Routing-Algorithmus ist nicht optimal. Betrachte wieder den Startknoten $(0, 011)$ und den Zielknoten $(2, 111)$. Diese sind in $\text{BF}(3)$ direkt durch eine Kante verbunden (vgl. Abbildung 15). Der Routing-Algorithmus findet diese Kante jedoch nicht. Analog zu Satz 3.8 beweist man:

Satz 3.10

Das Butterfly Netzwerk $\text{BF}(k)$ ist knotensymmetrisch.

Das Butterfly und das Cube-Connected-Cycles Netzwerk besitzen eine sehr ähnliche Struktur. Aus der Definition des Butterfly Netzwerks folgt unmittelbar:

Satz 3.11

$\text{BF}(k)$ läßt sich in $\text{CCC}(k)$ mit Kantenstreckung 2 und Kantenauslastung 2 einbetten.

Damit kann jeder Algorithmus des $\text{BF}(k)$ auf dem $\text{CCC}(k)$ mit konstantem Zeitverlust simuliert werden. Noch einfacher ist die Simulation eines Cube-Connected-Cycles Algorithmus auf dem Butterfly Netzwerk. Nach [18] gilt nämlich:

Satz 3.12

$\text{CCC}(k)$ ist Teilgraph des $\text{BF}(k)$.

Beweis:

Sei $V_k = \{(i, u); 0 \leq i < k, u \in \{0, 1\}^k\}$ die Knotenmenge des $\text{CCC}(k)$ bzw. des $\text{BF}(k)$. Definiere $f : V_k \rightarrow V_k$ durch: $(i, u) \mapsto ((i + g(u)) \bmod k, u)$ mit

$$g(u) = \begin{cases} 0, & \text{falls die Anzahl der Einsen in } u \text{ gerade} \\ 1, & \text{sonst} \end{cases}$$

Dann ist f eine bijektive Funktion. Es verbleibt zu zeigen: $\{(i, u), (j, v)\}$ Kante in $\text{CCC}(k) \Rightarrow \{f(i, u), f(j, v)\}$ Kante in $\text{BF}(k)$.

1.Fall: $(j, v) = ((i + 1) \bmod k, u)$ (Kreiskante)

Es gilt: $\{f(i, u), f((i + 1) \bmod k, u)\}$

$$= \begin{cases} \{(i, u), ((i + 1) \bmod k, u)\}, & \text{falls } g(u) = 0 \\ \{((i + 1) \bmod k, u), ((i + 2) \bmod k, u)\}, & \text{falls } g(u) = 1 \end{cases}$$

In beiden Fällen ist $\{f(i, u), f((i + 1) \bmod k, u)\}$ eine Kante des $\text{BF}(k)$.

2.Fall: $(j, v) = (i, u(i))$ (Hypercube-Kante)

Es gilt: $\{f(i, u), f(i, u(i))\}$

$$= \begin{cases} \{(i, u), ((i + 1) \bmod k, u(i))\}, & \text{falls } g(u) = 0 \text{ und damit } g(u(i)) = 1 \\ \{((i + 1) \bmod k, u), (i, u(i))\}, & \text{falls } g(u) = 1 \text{ und damit } g(u(i)) = 0 \end{cases}$$

In beiden Fällen ist $\{f(i, u), f(i, u(i))\}$ eine Kante des $\text{BF}(k)$.

■

Wir beenden diesen Abschnitt mit Satz 3.13. Der Beweis des Satzes sei dem Leser als Übungsaufgabe empfohlen.

Satz 3.13

$\text{CCC}(k)$ und $\text{BF}(k)$ enthalten für $k \geq 2$ einen Hamilton-Kreis.

3.3 Das DeBruijn und das Shuffle-Exchange Netzwerk

In diesem Abschnitt wollen wir zwei weitere Vertreter der Hypercube-Familie vorstellen. Es handelt sich dabei um das DeBruijn und das Shuffle-Exchange Netzwerk. Auf den ersten Blick scheinen diese Netzwerke nicht viel mit den Butterfly oder Cube-Connected-Cycles Netzwerken gemeinsam zu haben. Wir werden jedoch sehen, daß das DeBruijn Netzwerk sehr eng mit dem Butterfly und das Shuffle-Exchange Netzwerk sehr eng mit dem Cube-Connected-Cycles Netzwerk verwandt ist. Zunächst wollen wir jedoch die Netzwerke definieren und einige grundlegende Eigenschaften vorstellen.

Definition 3.17 (Shuffle-Exchange Netzwerk)

Das Shuffle-Exchange Netzwerk der Dimension k wird mit $\text{SE}(k)$ bezeichnet und ist wie folgt definiert:

$$\begin{aligned} \text{SE}(k) &= (V_k, E_k) \text{ mit} \\ V_k &= \{0, 1\}^k \\ E_k &= \{\{a\alpha, \alpha a\}; \alpha \in \{0, 1\}^{k-1}, a \in \{0, 1\}\} \\ &\quad \cup \{\{\alpha a, \alpha \bar{a}\}; \alpha \in \{0, 1\}^{k-1}, a \in \{0, 1\}\} \end{aligned}$$

Die Kanten der ersten Menge heißen *Shuffle-Kanten* und die Kanten der zweiten Menge werden *Exchange-Kanten* genannt. Alle Kreise, die ausschließlich aus Shuffle-Kanten bestehen, werden *Shuffle-Kreise* genannt.

Abbildung 16 zeigt das Shuffle-Exchange Netzwerk der Dimension 3. Alle Exchange-Kanten sind fett eingezeichnet.

Satz 3.14 (Eigenschaften des Shuffle-Exchange Netzwerks)

Das Shuffle-Exchange Netzwerk der Dimension k besitzt die folgenden Eigenschaften:

- 1.) $\text{SE}(k)$ besitzt 2^k Knoten und $3 \cdot 2^{k-1}$ Kanten.
- 2.) $\text{SE}(k)$ ist regulär vom Grad 3.
- 3.) Der Durchmesser des $\text{SE}(k)$ ist $2 \cdot k - 1$.
- 4.) Die Bisektionsweite des $\text{SE}(k)$ ist $\Theta(\frac{2^k}{k})$.

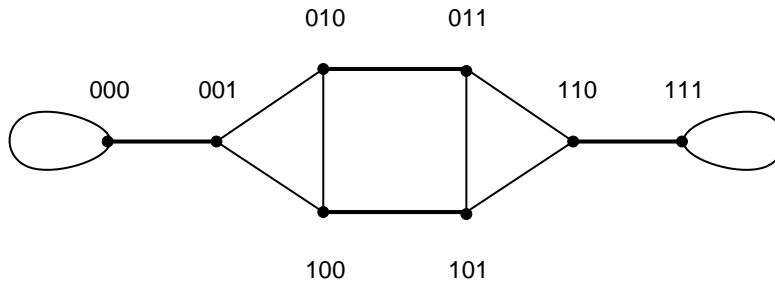


Abbildung 16: Das Shuffle-Exchange Netzwerk der Dimension 3

Wieder wollen wir auf den Beweis der 4. Aussage verzichten. Für das Shuffle-Exchange Netzwerk kann ein einfacher Routing-Algorithmus angegeben werden. Seien dazu $u, v \in \{0, 1\}^k$ zwei beliebige Knoten des $SE(k)$ mit $u = (u_{k-1} \dots u_0)$ und $v = (v_{k-1} \dots v_0)$. Sei weiter $s : \{0, 1\}^k \rightarrow \{0, 1\}^k$ die Shuffle-Funktion aus dem Beweis von Satz 3.8. Dann kann wie folgt eine Nachricht von u nach v geroutet werden.

```

for  $i := k - 1$  downto 0 do begin
  if  $u_0 \neq v_i$  then begin
    /* laufe Exchange-Kante entlang */
    schicke Nachricht von Knoten  $u$  nach Knoten  $u(0)$ ;
     $u := u(0)$ ;
  end;
  if  $i > 0$  then begin
    /* laufe Shuffle-Kante entlang */
    schicke Nachricht von Knoten  $u$  nach Knoten  $s(u)$ ;
     $u := s(u)$ ;
  end;
end;
end;
    
```

Der Algorithmus zeigt, daß für zwei Knoten u, v des $SE(k)$ gilt: $\text{dist}(u, v) \leq 2 \cdot k - 1$. Der Routing-Algorithmus ist jedoch nicht optimal, denn beispielsweise verschickt der Algorithmus eine Nachricht von Knoten 001 nach Knoten 100 über einen Weg der Länge 2. Beide Knoten sind jedoch in $SE(3)$ benachbart.

Definition 3.18 (DeBruijn Netzwerk)

Das DeBruijn Netzwerk der Dimension k wird mit $DB(k)$ bezeichnet und ist wie folgt definiert:

$$\begin{aligned}
 DB(k) &= (V_k, E_k) \text{ mit} \\
 V_k &= \{0, 1\}^k \\
 E_k &= \{(a\alpha, \alpha a); \alpha \in \{0, 1\}^{k-1}, a \in \{0, 1\}\} \\
 &\quad \cup \{(a\alpha, \alpha \bar{a}); \alpha \in \{0, 1\}^{k-1}, a \in \{0, 1\}\}
 \end{aligned}$$

Die Kanten der ersten Menge heißen wieder *Shuffle-Kanten* und die Kanten der zweiten Menge werden *Shuffle-Exchange-Kanten* genannt. Neben den vom $SE(k)$ bekannten Shuffle-Kreisen gibt es im $DB(k)$ zusätzlich *Shuffle-Exchange-Kreise*.

Im Gegensatz zu den bisher vorgestellten Netzwerken der Hypercube-Familie ist das DeBruijn Netzwerk als gerichteter Graph definiert. Dies erleichtert die Darstellung von Shuffle-

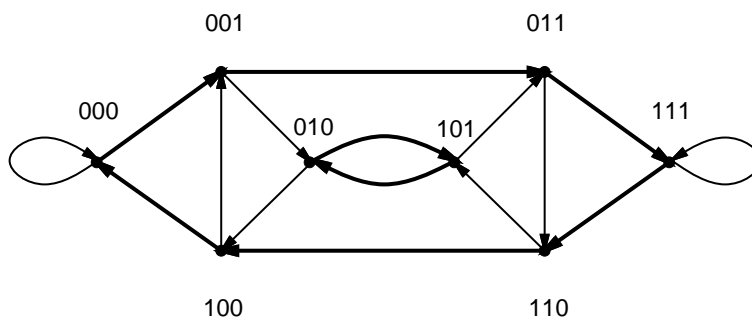


Abbildung 17: Das DeBruijn Netzwerk der Dimension 3

bzw. Shuffle-Exchange-Kreisen der Länge 2. Zudem ist die gerichtete Darstellung des $DB(k)$ Voraussetzung für die in Lemma 3.2 beschriebene Konstruktion. Auch das $SE(k)$ kann Shuffle-Kreise der Länge 2 enthalten (z.B. enthält das $SE(4)$ den Shuffle-Kreis $1010 \rightarrow 0101 \rightarrow 1010$). Folgerichtig müßte auch das $SE(k)$ als gerichteter Graph definiert werden. In diesem Fall lassen sich jedoch die Exchange-Kanten nicht „schön“ darstellen. Man geht daher den Kompromiß ein, daß $SE(k)$ als ungerichteten Graphen zu definieren, wobei Shuffle-Kreise der Länge 2 durch zwei ungerichtete Kanten dargestellt werden. Abbildung 17 zeigt das DeBruijn Netzwerk der Dimension 3. Alle Shuffle-Exchange-Kanten sind fett eingezeichnet.

Satz 3.15 (Eigenschaften des DeBruijn Netzwerks)

Das DeBruijn Netzwerk der Dimension k besitzt die folgenden Eigenschaften:

- 1.) $DB(k)$ besitzt 2^k Knoten und $4 \cdot 2^{k-1} = 2^{k+1}$ Kanten.
- 2.) $DB(k)$ ist regulär vom Grad 4.
- 3.) Der Durchmesser des $DB(k)$ ist k .
- 4.) Die Bisektionsweite des $DB(k)$ ist $\Theta(\frac{2^k}{k})$.

Auch für das DeBruijn Netzwerk kann ein einfacher Routing-Algorithmus angegeben werden. Seien dazu wieder $u, v \in \{0, 1\}^k$ zwei beliebige Knoten des $DB(k)$ mit $u = (u_{k-1} \dots u_0)$ und $v = (v_{k-1} \dots v_0)$. Sei weiter $s : \{0, 1\}^k \rightarrow \{0, 1\}^k$ die bekannte Shuffle-Funktion und $q : \{0, 1\}^k \rightarrow \{0, 1\}^k$ mit $q(x_{k-1} \dots x_0) = x_{k-2} \dots x_0 \bar{x}_{k-1}$ die *Shuffle-Exchange-Funktion*. Dann kann wie folgt eine Nachricht von u nach v geroutet werden.

```

for  $i := k - 1$  downto 0 do
  if  $u_{k-1} \neq v_i$  then begin
    /* laufe Shuffle-Exchange-Kante entlang */
    schicke Nachricht von Knoten  $u$  nach Knoten  $q(u)$ ;
     $u := q(u)$ ;
  end
  else begin
    /* laufe Shuffle-Kante entlang */
    schicke Nachricht von Knoten  $u$  nach Knoten  $s(u)$ ;
     $u := s(u)$ ;
  end;
end;

```

Der Leser macht sich leicht klar, daß auch dieser Routing-Algorithmus nicht optimal ist. Das Shuffle-Exchange und das DeBruijn Netzwerk besitzen – vergleichbar dem Cube-Connected-

Cycles und dem Butterfly Netzwerk – eine sehr ähnliche Struktur. Definiert man das $DB(k)$ als ungerichteten Graphen, so gilt analog zu den Sätzen 3.11 und 3.12:

Satz 3.16

Das ungerichtete $DB(k)$ läßt sich in $SE(k)$ mit Kantenstreckung 2 und Kantenauslastung 2 einbetten.

Satz 3.17

$SE(k)$ ist Teilgraph des ungerichteten $DB(k)$.

Beweis:

Sei $V_k = \{0, 1\}^k$ die Knotenmenge des Shuffle-Exchange bzw. des DeBruijn Netzwerks. Desweiteren seien $s, q : \{0, 1\}^k \rightarrow \{0, 1\}^k$ die bekannten Shuffle- bzw. Shuffle-Exchange-Funktionen. Die beiden Funktionen besitzen die folgenden Eigenschaften:

- 1.) s, q sind bijektiv
- 2.) $(s \circ q^{-1})(u) = (q \circ s^{-1})(u) = u(0)$

Die gesuchte Einbettung $f : V_k \rightarrow V_k$ kann jetzt wie folgt definiert werden:

$$f(u) = \begin{cases} u, & \text{falls die Anzahl der Einsen in } u \text{ gerade} \\ s^{-1}(u), & \text{sonst} \end{cases}$$

Man sieht leicht, daß f injektiv ist. Es verbleibt daher zu zeigen: $\{u, v\}$ Kante in $SE(k) \Rightarrow \{f(u), f(v)\}$ Kante in dem ungerichteten $DB(k)$.

1.Fall: $v = u(0)$ (Exchange-Kante)

Ist die Anzahl der Einsen in u gerade, so gilt: $\{f(u), f(u(0))\} = \{u, s^{-1}(u(0))\} \stackrel{2.)}{=} \{u, q^{-1}(u)\}$.

Anderenfalls gilt: $\{f(u), f(u(0))\} = \{s^{-1}(u), u(0)\} \stackrel{2.)}{=} \{s^{-1}(u), q(s^{-1}(u))\}$. In beiden Fällen ist $\{f(u), f(u(0))\}$ eine Kante des $DB(k)$.

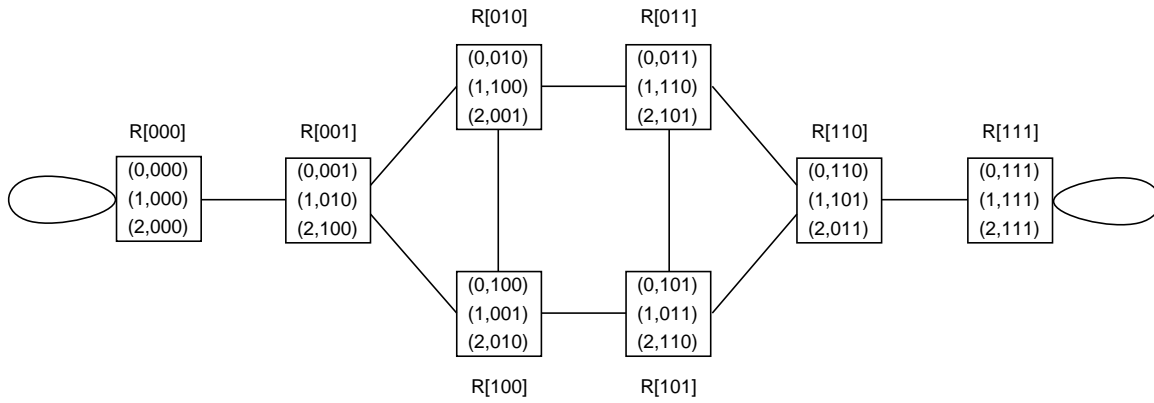
2.Fall: $v = s(u)$ (Shuffle-Kante)

Ist die Anzahl der Einsen in u gerade, so gilt: $\{f(u), f(s(u))\} = \{u, s(u)\}$. Anderenfalls gilt: $\{f(u), f(s(u))\} = \{s^{-1}(u), s^{-1}(s(u))\} = \{s^{-1}(u), u\}$. In beiden Fällen ist $\{f(u), f(s(u))\}$ eine Kante des ungerichteten $DB(k)$.

■

Im folgenden wollen wir zeigen, wie man durch geschicktes „Zusammenfalten“ der Knoten des Cube-Connected-Cycles Netzwerks das Shuffle-Exchange Netzwerk erhält. Analog erhält man aus dem Butterfly Netzwerk das ungerichtete DeBruijn Netzwerk. Für $\alpha \in \{0, 1\}^k$, $\alpha = (b_{k-1} \dots b_0)$, definieren wir dazu die Menge $R[\alpha]$ durch:

$$\begin{aligned} R[\alpha] &= \{(0, b_{k-1} \dots b_0), (1, b_{k-2} \dots b_0 b_{k-1}), \dots, (k-1, b_0 b_{k-1} \dots b_1)\} \\ &= \{(i, b_{k-i-1} \dots b_0 b_{k-1} \dots b_{k-i}); 0 \leq i < k\} \end{aligned}$$


 Abbildung 18: Der Faktorgraph $G_{CCC}(3)$

Aus der Definition folgt sofort: $R[\alpha_1] \cap R[\alpha_2] = \emptyset$ für $\alpha_1 \neq \alpha_2$. Man kann daher wie folgt einen Graphen $G_{CCC}(k) = (U, F_{CCC})$ und einen Graphen $G_{BF}(k) = (U, F_{BF})$ definieren:

$$\begin{aligned}
 U &= \{R[\alpha]; \alpha \in \{0, 1\}^k\} \\
 F_{CCC} &= \{\{R[\alpha_1], R[\alpha_2]\}; \exists \text{ Kante } \{(i, u), (j, v)\} \text{ des } CCC(k) \text{ mit} \\
 &\quad (i, u) \in R[\alpha_1] \text{ und } (j, v) \in R[\alpha_2]\} \\
 F_{BF} &= \{\{R[\alpha_1], R[\alpha_2]\}; \exists \text{ Kante } \{(i, u), (j, v)\} \text{ des } BF(k) \text{ mit} \\
 &\quad (i, u) \in R[\alpha_1] \text{ und } (j, v) \in R[\alpha_2]\}
 \end{aligned}$$

$G_{CCC}(k)$ heißt *Faktorgraph* des $CCC(k)$ und $G_{BF}(k)$ heißt *Faktorgraph* des $BF(k)$. Abbildung 18 zeigt den Faktorgraphen $G_{CCC}(3)$.

Satz 3.18

Die Abbildung $f : \{0, \dots, k-1\} \times \{0, 1\}^k \mapsto \{0, 1\}^k$ definiert durch

$$f(i, \alpha) = s^{-i}(\alpha) \quad \forall \alpha \in \{0, 1\}^k, 0 \leq i \leq k-1$$

hat Knotenauslastung k und ist Einbettung des $CCC(k)$ in $SE(k)$ mit Kantenauslastung 1 und Einbettung des $BF(k)$ in $DB(k)$ mit Kantenauslastung 1.

Beweis:

Setze $R(\alpha) = \{(i, s^i(\alpha)) \mid 0 \leq i \leq k-1\}$, wobei $(i, u) \in R(\alpha) \Leftrightarrow f(i, u) = \alpha$. Wir zeigen:

- $\{(i, u), (i+1, u)\}$ ist Kreiskante im $CCC(k) \Rightarrow \{f(i, u), f(i+1, u)\}$ ist Shuffle-Kante im $SE(k)$.
- $\{(i, u), (i, u(i))\}$ ist Querkante im $CCC(k) \Rightarrow \{f(i, u), f(i, u(i))\}$ ist Exchange-Kante im $SE(k)$.

zu (i): Die Beobachtungen, dass $f(i, u) = s^{-i}(u)$ und $f(i+1, u) = s^{-(i+1)}(u)$ liefern die gewünschte Aussage.

zu (ii): Wir haben wieder, dass gilt: $f(i, u) = s^{-i}(u)$ und $f(i, u(i)) = s^{-i}(u(i))$. Für beliebiges, aber festes $u = a_{k-1} \dots a_0$ gilt somit, dass

$$u(i) = a_{k-1} \dots a_{i+1} \bar{a}_i a_{i-1} \dots a_0,$$

$$s^{-i}(u) = a_{i-1} \dots a_0 a_{k-1} \dots a_i,$$

und

$$s^{-i}(u(i)) = a_{i-1} \dots a_0 a_{k-1} \dots a_{i+1} \bar{a}_i.$$

Somit wird jede Querkante im CCC(k) auf eine Exchange-Kante im SE(k) abgebildet. ■

Das DeBruijn Netzwerk besitzt eine ganze Reihe interessanter Eigenschaften. An dieser Stelle wollen wir nur auf die hierarchische Struktur des Netzwerks eingehen. So wie der Hypercube $Q(k+1)$ aus $Q(k)$ konstruiert werden kann, kann auch das DeBruijn Netzwerk $DB(k+1)$ aus $DB(k)$ hergeleitet werden. Wir definieren dazu:

Definition 3.19 (Kantengraph)

Sei $G = (V, E)$ ein gerichteter Graph. Der Kantengraph von G ist definiert als $\widehat{G} = (E, \widehat{E})$ mit $\widehat{E} = \{(e_1, e_2); e_1, e_2 \in E \text{ und } \exists u, v, w \in V \text{ mit } e_1 = (u, v) \text{ und } e_2 = (v, w)\}$.

Lemma 3.2

$DB(k+1)$ ist isomorph zum Kantengraph des $DB(k)$.

Beweis:

Im folgenden sei $\alpha \in \{0, 1\}^{k-1}$ und $a, b \in \{0, 1\}$. Jede Kante $(a\alpha, \alpha a')$, $a' \in \{a, \bar{a}\}$, des $DB(k)$ wird in dem zugehörigen Kantengraph dargestellt als Knoten $a\alpha a'$. Betrachte nun zwei beliebige aufeinanderfolgende Kanten $e_1 = (ab\alpha, b\alpha a')$, $e_2 = (b\alpha a', \alpha a' b')$, $b' \in \{b, \bar{b}\}$, des $DB(k)$. Die Kante e_1 wird in dem Kantengraph dargestellt als Knoten $ab\alpha a'$ und die Kante e_2 als Knoten $b\alpha a' b'$. In dem Kantengraph von $DB(k)$ gibt es daher eine Kante $(ab\alpha a', b\alpha a' b')$. Im Fall $a = b'$ ist dies eine Shuffle-Kante in $DB(k+1)$, im Fall $a \neq b'$ eine Shuffle-Exchange-Kante. Es folgt: der Kantengraph des $DB(k)$ ist ein Teilgraph des $DB(k+1)$. Analog zeigt man, daß der $DB(k+1)$ ein Teilgraph des Kantengraphen von $DB(k)$ ist. ■

Mit Hilfe der in Lemma 3.2 beschriebenen hierarchischen Struktur des DeBruijn Netzwerks kann leicht nachgewiesen werden, daß $DB(k)$ einen Hamilton-Kreis besitzt.

Satz 3.19

$DB(k)$ enthält für $k \geq 2$ einen Hamilton-Kreis.

Beweis:

$DB(k)$ ist regulär vom Grad 4, wobei jeweils zwei Kanten in einen Knoten hinein- und zwei Kanten aus einem Knoten herausführen. Daher besitzt $DB(k)$ einen Euler-Kreis. Dieser wird in dem Kantengraphen des $DB(k)$ – und damit nach Lemma 3.2 in $DB(k + 1)$ – zu einem Hamilton-Kreis.

■

Das Shuffle-Exchange Netzwerk $SE(k)$ enthält keinen Hamilton-Kreis, denn nach Entfernen der Shuffle-Kanten an den Knoten $\{0\}^k$ und $\{1\}^k$ besitzen beide Knoten den Grad 1. Nach [17] gilt jedoch:

Satz 3.20

$SE(k)$ enthält einen Hamilton-Pfad.

Kapitel 4

Algorithmen für synchrone Rechnernetze

In diesem Kapitel wollen wir einige grundlegende Algorithmen für die in Kapitel 3 vorgestellten Rechnernetze entwickeln. Zur Vereinfachung des Algorithmenentwurfs gehen wir davon aus, daß es in dem Rechnernetz einen globalen Takt gibt (synchrones Rechnernetz), und daß alle Prozessoren das gleiche Programm abarbeiten. Im Gegensatz zum PRAM-Modell besitzen die Prozessoren des synchronen Rechnernetzes nur einen lokalen Speicher. Jeder Prozessor kann auf seine eigenen und auf die Variablen seiner Nachbarn zugreifen. Nachbarn sind all diejenigen Prozessoren, zu denen eine direkte Verbindung im Netzwerk (Link) existiert.

Speicherzugriffe nicht benachbarter Prozessoren können z.B. realisiert werden durch Kommunikationsoperationen (Send, Receive) auf *virtuellen* Links. Aufgabe der Routing-Software ist es dann, die über einen virtuellen Link verschickte Nachricht auf den tatsächlich vorhandenen physikalischen Links zum Zielprozessor weiterzuleiten. Das Betriebssystem PARIX benutzt dieses Konzept der virtuellen Links. Da sich der Anwender nicht um das Routing kümmern muß, ist so eine komfortable Programmierung des Prozessornetzwerks möglich.

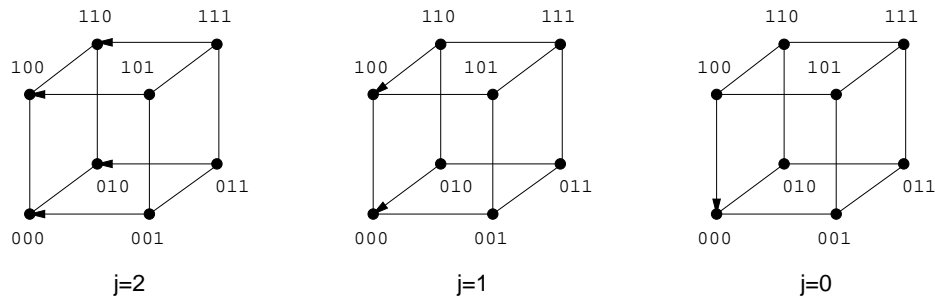
Um im folgenden die Algorithmen unabhängig von bestimmten Kommunikations-Routinen formulieren zu können, werden Sende- bzw. Empfangsoperationen implizit durch Lese- bzw. Schreiboperationen ausgedrückt. Dabei darf jedoch nicht vergessen werden, daß – im Gegensatz zum PRAM-Modell – nur Speicherzugriffe auf den eigenen lokalen Speicher und den eines direkt benachbarten Prozessors erlaubt sind. Die Programme für synchrone Rechnernetze können dann wieder in der bekannten PASCAL-Notation geschrieben werden.

Zur Bestimmung des Aufwands eines Algorithmus werden lediglich die Zugriffe auf den Speicher eines benachbarten Prozessors gezählt. Diese externen Speicherzugriffe werden im folgenden *Kommunikationsschritte* genannt. Weder interne Operationen, noch lokale Speicherzugriffe werden bei der Aufwandsbestimmung berücksichtigt. In Abhängigkeit von dem zugrunde liegenden Kommunikationsmodus kann ein Kommunikationsschritt unterschiedlich definiert werden. Im folgenden betrachten wir nur die Modi *telegraph-* und *telephone-mode*.

Definition 4.1 (Kommunikationsschritt im telegraph-mode)

Ein Kommunikationsschritt ist im telegraph-mode wie folgt definiert:

1. In einem Kommunikationsschritt kann ein Prozessor nicht gleichzeitig Senden und Empfangen.

Abbildung 19: Bestimmung des Minimums auf $Q(3)$

2. Jeder Kanal darf pro Kommunikationsschritt nur in einer Richtung benutzt werden.
3. Jeder Prozessor darf pro Kommunikationsschritt nur einen seiner Kanäle benutzen.

Definition 4.2 (Kommunikationsschritt im telephone-mode)

Ein Kommunikationsschritt ist im telephone-mode wie folgt definiert:

1. In einem Kommunikationsschritt kann ein Prozessorenpaar Nachrichten untereinander austauschen.
2. Bei dem Nachrichtenaustausch können die Kanäle in beiden Richtungen benutzt werden.
3. Jeder Prozessor darf pro Kommunikationsschritt nur einen seiner Kanäle benutzen.

Es folgt unmittelbar, daß jeder Algorithmus, der im telephone-mode n Kommunikationsschritte benötigt, im telegraph-mode durch einen Algorithmus mit $\leq 2n$ Kommunikationsschritten simuliert werden kann.

Zur Einführung wollen wir für den Hypercube und für das Cube-Connected-Cycles Netzwerk einen Algorithmus zur Bestimmung des Minimums angeben. Bei beiden Algorithmen gehen wir davon aus, daß jedem Prozessor des Netzwerks eine Zahl zugeordnet ist. Im Hypercube $Q(k)$ sei die dem Prozessor $\alpha \in \{0,1\}^k$ zugeordnete Zahl in $A[\alpha]$, $A : \text{array}[0..2^k - 1]$ of integer, gespeichert. Nach Termination des Algorithmus soll das Minimum aller Zahlen in $A[0^k]$ stehen, d.h. Prozessor 0^k kennt das Minimum. Die Bestimmung des Minimums geschieht mit Hilfe des folgenden Algorithmus.

```

for  $j := k - 1$  downto 0 do
  for all  $\alpha \in \{0,1\}^j$  parallel do
     $A[\alpha 00^{k-1-j}] := \min\{A[\alpha 00^{k-1-j}], A[\alpha 10^{k-1-j}]\}$ 
  od
od

```

Abbildung 19 veranschaulicht die Vorgehensweise des Algorithmus an $Q(3)$. Während der ersten Iteration ($j = 2$) wird das Minimum in den Teilcube bestehend aus den Knoten $\{0,1\}^2$ geschoben, in der zweiten Iteration in den Teilcube $\{0,1\}00$. Nach der dritten Iteration kennt Prozessor 0^3 das Minimum und der Algorithmus terminiert. Im telegraph-mode werden so insgesamt k Kommunikationsschritte benötigt. Bei dem Algorithmus für das Cube-Connected-Cycles Netzwerk $CCC(k)$ sei angenommen, daß Prozessor (i, α) , $0 \leq i < k$, $\alpha \in \{0,1\}^k$, seine Zahl in $A[(i, \alpha)]$ speichert. Im folgenden bezeichne α_i das i -te Bit von α .

1. Verschiebe auf den Kreisen das Minimum zu den Knoten $(k - 1, \alpha)$
2. for $i := k - 1$ downto 1 do
 - for all $\alpha \in \{0, 1\}^k$ parallel do
 - if $\alpha_i = 0$ then
 - $A[(i, \alpha)] := \min\{A[(i, \alpha)], A[(i, \alpha(i))]\}$
 - $A[(i - 1, \alpha)] := A[(i, \alpha)]$
 - fi
 - od
- od
3. for all $\alpha \in \{0, 1\}^k$ parallel do
 - if $\alpha_0 = 0$ then
 - $A[(0, \alpha)] := \min\{A[(0, \alpha)], A[(0, \alpha(0))]\}$
 - fi
- od

Abbildung 20 veranschaulicht an CCC(3) den Kommunikationsfluß in Schritt 2 des Algorithmus. Im telegraph-mode werden insgesamt $2k - 1 + \lfloor \frac{k}{2} \rfloor$ Schritte benötigt, denn es gilt:

- Schritt 1 benötigt $\lfloor \frac{k}{2} \rfloor$ Kommunikationsschritte
- Schritt 2 benötigt $2(k - 1)$ Kommunikationsschritte
- Schritt 3 benötigt 1 Kommunikationsschritt

Der Rest des Kapitels ist wie folgt aufgebaut: In Abschnitt 4.1 werden wir eine ganze Klasse von Algorithmen – die sogenannten *ASCEND/DESCEND Algorithmen* – für den Hypercube vorstellen. Abschnitt 4.2 zeigt, wie mit Hilfe eines ASCEND/DESCEND Algorithmus 2^k Zahlen auf einem Hypercube der Dimension k sortiert werden können. In Abschnitt 4.3 wird gezeigt, wie ASCEND/DESCEND Algorithmen auf anderen Netzwerken der Hypercube-Familie implementiert werden können. Abschnitt 4.4 stellt die Fast-Fourier-Transformation und die Rechnernetze, auf denen sie gut parallelisierbar ist, vor. Abschließend wird in Abschnitt 4.5 auf einer abstrakteren Ebene die Verbreitung von Informationen in Rechnernetzen untersucht.

4.1 ASCEND/DESCEND Algorithmen für den Hypercube

Die ASCEND/DESCEND Algorithmen stellen eine direkte Umsetzung des Divide & Conquer Paradigmas auf Parallelrechner dar. Die Vorgehensweise eines ASCEND/DESCEND Algorithmus kann wie folgt charakterisiert werden:

- Rekursive Aufteilung der Daten in zwei gleich große Teilprobleme.
- Zusammenführung der Ergebnisse durch Operationen auf Paaren von Daten.

Aufgrund der rekursiven Struktur des Hypercube stellen ASCEND/DESCEND Algorithmen eine natürliche Art und Weise zur Programmierung der Maschine dar. ASCEND/DESCEND Algorithmen zeichnen sich dadurch aus, daß sie die Dimensionen des Hypercube in einer bestimmten Reihenfolge durchlaufen. Ein ASCEND Algorithmus durchläuft die Dimensionen aufsteigend und sieht wie folgt aus:

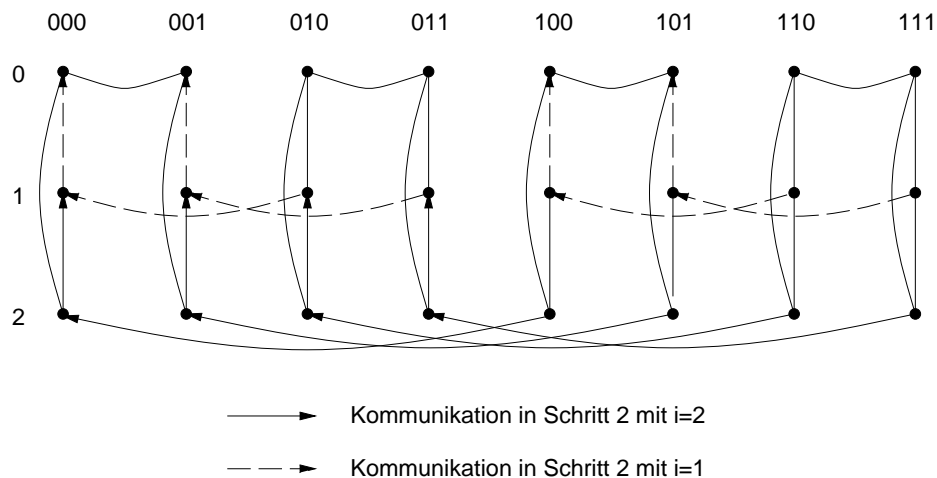


Abbildung 20: Bestimmung des Minimums auf $CCC(3)$

```

Procedure ASCEND (procedure OPER)
var  $\alpha : \{0, 1\}^k$ ; dim :  $0 \dots k - 1$ ;
begin
  for dim := 0 to  $k - 1$  do
    for all  $\alpha \in \{0, 1\}^k$  parallel do
      OPER( $\alpha$ , dim)
    od
  od
end
    
```

In der als Parameter übergebenen Prozedur OPER werden Operationen mit Datenpaaren über Kanten der Dimension dim ausgeführt. Der DESCEND Algorithmus unterscheidet sich nur dadurch, daß er die Dimensionen des Hypercube absteigend durchläuft. Bezeichnet t_{OPER} die Ausführungszeit der Prozedur OPER, so beträgt die Laufzeit eines ASCEND/DESCEND Programms auf $Q(k)$ $k \cdot t_{\text{OPER}}$. Anhand zweier Beispiele wollen wir den Gebrauch der ASCEND/DESCEND Prozeduren veranschaulichen. Der Hypercube-Algorithmus zur Bestimmung des Minimums kann wie folgt als ASCEND Programm formuliert werden:

```

procedure MIN ( $\alpha : \{0, 1\}^k$ ; dim :  $0 \dots k - 1$ )
var  $\alpha' : \{0, 1\}^k$ ;
begin
   $\alpha' = \alpha(\text{dim})$ ;
  if  $\alpha = \beta 00^{\text{dim}}$ ,  $\beta \in \{0, 1\}^{k-1-\text{dim}}$ , then  $A[\alpha] := \min\{A[\alpha], A[\alpha']\}$ ;
end;
begin /* Hauptprogramm */
  ASCEND(MIN);
end.
    
```

Für das zweite Beispiel bezeichne wieder $A[\alpha]$ die von Prozessor $\alpha \in \{0, 1\}^k$ gespeicherte Zahl. Sei weiter π_{Rev} eine Permutation auf den Zahlen $0, \dots, 2^k - 1$ definiert durch $\pi_{\text{Rev}}(i) =$

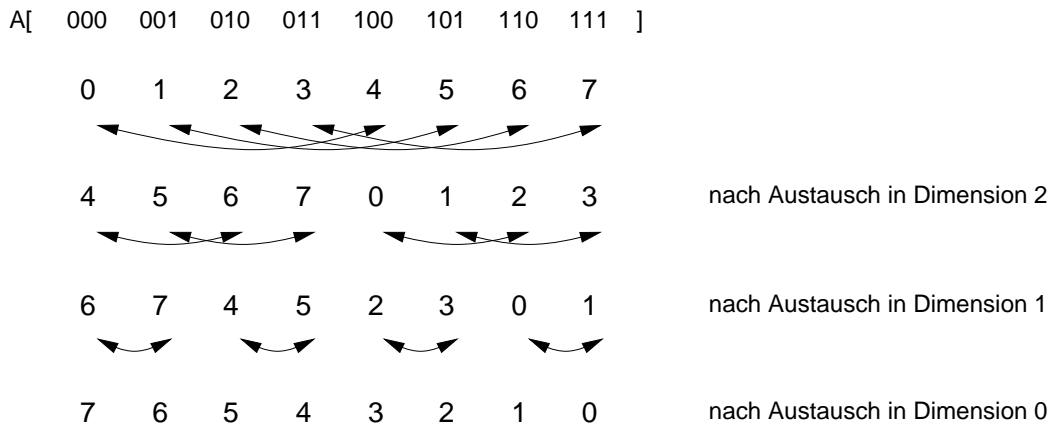


Abbildung 21: Vertauschungen in $Q(3)$ zum Routen von π_{Rev}

$2^k - 1 - i$. Aufgabe ist es nun, die Variable $A[i]$ von Prozessor i nach Prozessor $\pi_{Rev}(i)$ zu routen. Der folgende DESCEND Algorithmus leistet das Gewünschte.

```

procedure REV ( $\alpha : \{0, 1\}^k$ ; dim :  $0 \dots k - 1$ )
var  $\alpha' : \{0, 1\}^k$ ;
begin
   $\alpha' = \alpha(\text{dim})$ ;
  if (bit( $\alpha$ , dim) = 0) then tausche( $A[\alpha]$ ,  $A[\alpha']$ );
end;
begin /* Hauptprogramm */
  DESCEND(REV);
end.

```

Um den Variablen austausch in Prozedur REV in einem Schritt durchführen zu können, muß im telephone-mode gearbeitet werden. Abbildung 21 zeigt die in den einzelnen Dimensionen durchgeführten Vertauschungen für $k = 3$.

4.2 Bitones Sortieren auf der Hypercube

Die zu sortierenden Daten seien in $A : \text{array}[\{0, 1\}^k]$ of integer abgespeichert. Aufgabe ist es, A so umzuordnen, daß gilt: $\forall \alpha, \beta \in \{0, 1\}^k : \alpha < \beta \Rightarrow A[\alpha] \leq A[\beta]$. Wir zeigen zunächst, wie spezielle Zahlenfolgen sortiert werden können.

Definition 4.3 (bitone Zahlenfolge)

Eine Zahlenfolge (a_0, \dots, a_{n-1}) heißt biton, falls gilt:

- a) Es gibt einen Index j , $0 \leq j < n$, so daß (a_0, \dots, a_j) monoton steigt und (a_j, \dots, a_{n-1}) monoton fällt.
- b) Ist a) nicht erfüllt, so gibt es einen Index i , $0 \leq i < n$, so daß $(a_i, \dots, a_{n-1}, a_0, \dots, a_{i-1})$ die Bedingung a) erfüllt.

Beispielsweise ist $(0, 2, 4, 5, 6, 7, 3, 1)$ biton nach a) und $(6, 7, 5, 3, 0, 1, 4, 5)$ biton nach b) mit $i = 4$. Einige einfache Eigenschaften bitoner Folgen faßt Lemma 4.1 zusammen.

Lemma 4.1

1. Biton ist gegen zyklisches Shiften abgeschlossen.
2. Jede Teilfolge einer bitonen Folge ist biton.
3. Ist die Folge (a_0, \dots, a_i) aufsteigend und die Folge $(b_{i+1}, \dots, b_{n-1})$ absteigend sortiert, dann ist die zusammengesetzte Folge $(a_0, \dots, a_i, b_{i+1}, \dots, b_{n-1})$ biton.

Definition 4.4 (eingabeunabhängiger Comparison-Exchange-Algorithmus)

Ein Algorithmus im Telephon-Modus heißt Comparison-Exchange-Algorithmus, wenn in einem Schritt die Prozessorpaare ihre Zahlen miteinander vergleichen und in Abhängigkeit des Vergleichs eventuell austauschen können. Ein solcher Vergleich/Austausch heißt Comparison-Exchange-Operation. Ein Comparison-Exchange-Algorithmus heißt eingabeunabhängig (oblivious), wenn die Wahl der Prozessorpaare pro Runde nur von der Länge der Eingabe abhängt.

Eine Comparison-Exchange-Operation läßt sich durch ein Tupel von Prozessornummern (α, β) beschreiben; der Effekt einer solchen Operation ist, daß Prozessor α anschließend das Minimum und Prozessor β das Maximum der beiden Zahlen $A[\alpha]$ und $A[\beta]$ speichert.

Um zu beweisen, daß ein Algorithmus ein Sortieralgorithmus ist, muß i.A. gezeigt werden, daß er jede beliebige Folge von Zahlen korrekt sortiert. Für eingabeunabhängige Comparison-Exchange-Algorithmen reicht es aus zu zeigen, daß sie jede beliebige 0-1-Folge (Folgen, die nur aus 0 und 1 bestehen) korrekt sortieren, wie das folgende Lemma zeigt:

Lemma 4.2 (0-1-Prinzip)

Sei CE ein eingabeunabhängiger Comparison-Exchange-Algorithmus. CE sortiert genau dann alle Folgen ganzer Zahlen korrekt, wenn er alle 0-1-Folgen korrekt sortiert.

Beweis:

\implies : Offensichtlich.

\impliedby : Widerspruchsbeweis.

Ann.: CE sortiert jede 0-1-Folge korrekt, aber es existiert eine Folge ganzer Zahlen, die von CE nicht korrekt sortiert wird.

Sei $A = (a_0, \dots, a_{n-1})$ eine ganzzahlige Folge, die von CE nicht korrekt sortiert wird. Jeder Comparison-Exchange-Algorithmus führt eine Permutation auf seiner Eingabefolge aus. Sei $\pi : \{0, \dots, n-1\} \rightarrow \{0, \dots, n-1\}$ die Permutation, die CE auf der Folge A durchführt, d.h. die Ausgabefolge von CE lautet $(a_{\pi(0)}, \dots, a_{\pi(n-1)})$. Da CE A nicht korrekt sortiert, existiert ein $p \in \{1, \dots, n-1\}$ mit $a_{\pi(p-1)} > a_{\pi(p)}$. Wir konstruieren aus A und $a_{\pi(p)}$ wie folgt eine 0-1-Folge $B = (b_0, \dots, b_{n-1})$:

$$\forall i \in \{0, \dots, n-1\} : b_i = \begin{cases} 0 & , \text{ falls } a_i \leq a_{\pi(p)} \\ 1 & , \text{ sonst} \end{cases}$$

Da A und B Folgen gleicher Länge sind und CE eingabeunabhängig ist, wendet CE auf A und B dieselben Comparison-Exchange-Operationen an. Sei c die Anzahl der Comparison-Exchange-Operationen, die CE auf A bzw. B durchführt, und $(\alpha_1, \beta_1), \dots, (\alpha_c, \beta_c)$ eine sequentielle Folge dieser Operationen, d.h. wenn (α_i, β_i) im Kommunikationsschritt s_i und (α_j, β_j) im Kommunikationsschritt s_j durchgeführt werden, so gilt $s_i < s_j \Rightarrow i < j$. Ferner stehe π_i^A (π_i^B) für die Permutation, die sich aus den Comparison-Exchange-Operationen 1 bis i auf A (B) ergibt.

Durch Induktion über die Anzahl der Comparison-Exchange-Operationen zeigen wir im folgenden, daß $\pi_i^A = \pi_i^B$ für alle $i \in \{0, \dots, c\}$ gilt. Da dann insbesondere $\pi = \pi_c^A = \pi_c^B$ gilt und aus der Konstruktion von B $1 = b_{\pi(p-1)} > b_{\pi(p)} = 0$ folgt, wird auch die 0-1-Folge B von CE nicht korrekt sortiert, was den gewünschten Widerspruch liefert.

Ind. Anf.: $s = 0$

Vor der ersten Comparison-Exchange-Operation gilt $\pi_0^A = \pi_0^B = id$.

Ind. Vor.:

Nach den Comparison-Exchange-Operationen 1 bis $s-1$ gelte $\pi_{s-1}^A = \pi_{s-1}^B$.

Ind. Schritt: $s-1 \mapsto s$

Im Schritt s wird die Comparison-Exchange-Operation (α_s, β_s) durchgeführt. Nach Induktionsvoraussetzung arbeitet die Operation dabei auf den Elementen a_i, a_j bzw. b_i, b_j mit $i = \pi_{s-1}^A(\alpha_s) = \pi_{s-1}^B(\alpha_s)$ und $j = \pi_{s-1}^A(\beta_s) = \pi_{s-1}^B(\beta_s)$.

Fall 1: $a_i \leq a_j$

Es folgt $b_i \leq b_j$. Somit wird weder auf A noch auf B ein Tausch ausgeführt und es gilt $\pi_s^A = \pi_s^B (= \pi_{s-1}^A)$.

Fall 2: $a_i > a_j$

Es folgt $b_i \geq b_j$. Auf A wird somit ein Tausch vorgenommen. Für den Fall $b_i > b_j$ findet derselbe Tausch auch auf B statt. Im Fall von $b_i = b_j$ können wir ferner o.B.d.A. annehmen, daß auch hier die beiden Elemente ausgetauscht werden. Also gilt wieder $\pi_s^A = \pi_s^B$.

Fazit: CE sortiert alle Folgen ganzer Zahlen korrekt. ■

Bitone Folgen können auf der Hypercube mittels eines DESCEND Durchlaufs sehr einfach sortiert werden:

```

procedure bitonic_merge ( $\alpha : \{0, 1\}^k$ ; dim :  $0 \dots k-1$ )
var  $\alpha' : \{0, 1\}^k$ ;
begin
   $\alpha' = \alpha(\text{dim})$ ;
  if (bit( $\alpha$ , dim) = 0) then
    if  $A[\alpha] > A[\alpha']$  then tausche( $A[\alpha]$ ,  $A[\alpha']$ );
  end;
begin      /* Hauptprogramm */
  DESCEND(bitonic_merge);
end.
```

Das folgende Lemma zeigt, daß die Folge nach dem DESCEND Durchlauf sortiert ist.

Lemma 4.3

Ist A zu Beginn biton, so ist A nach dem Aufruf DESCEND(bitonic_merge) aufsteigend sortiert.

Beweis:

Offensichtlich handelt es sich bei DESCEND(bitonic_merge) um einen eingabeunabhängigen Comparison-Exchange-Algorithmus, d.h. nach Lemma 4.2 genügt es zu zeigen, daß der Algorithmus jede bitone 0-1-Folge korrekt sortiert.

Für $\beta \in \{0, 1\}^{k-1}$ setze $D[\beta] := \min\{A[0\beta], A[1\beta]\}$ und $E[\beta] := \max\{A[0\beta], A[1\beta]\}$. Wir beweisen zunächst:

Zwischenbehauptung:

Ist A eine bitone 0-1-Folge, so sind D und E jeweils bitone 0-1-Folgen und es gilt: $\max(D) \leq \min(E)$.

Beweis der Zwischenbehauptung:

Es sei $n = 2^k$ die Anzahl der Elemente von A. Bitone 0-1-Folgen sind entweder von der Form $(0, \dots, 0, 1, \dots, 1, 0, \dots, 0)$ oder $(1, \dots, 1, 0, \dots, 0, 1, \dots, 1)$. Wir unterscheiden 6 Fälle:

Fall 1: $A = (\underbrace{0, \dots, 0}_i, \underbrace{1, \dots, 1}_j \mid \underbrace{1, \dots, 1}_l, \underbrace{0, \dots, 0}_m)$ mit $i + j = l + m = n/2$

a) $i < l$

Es sind $D = (\underbrace{0, \dots, 0}_i, \underbrace{1, \dots, 1}_{l-i}, \underbrace{0, \dots, 0}_m)$ und $E = (1, \dots, 1)$ biton mit $\max(D) \leq \min(E)$.

b) $i \geq l$

Es folgt $j \leq m$ und deshalb $D = (0, \dots, 0)$ und $E = (\underbrace{1, \dots, 1}_l, \underbrace{0, \dots, 0}_{m-j}, \underbrace{1, \dots, 1}_j)$

Fall 2: $A = (\underbrace{0, \dots, 0}_{n/2} \mid \underbrace{0, \dots, 0, 1, \dots, 1, 0, \dots, 0}_{n/2})$

Es folgt $D = (0, \dots, 0)$ und $E = (0, \dots, 0, 1, \dots, 1, 0, \dots, 0)$.

Fall 3: $A = (\underbrace{0, \dots, 0, 1, \dots, 1, 0, \dots, 0}_{n/2} \mid \underbrace{0, \dots, 0}_{n/2})$

Es folgt $D = (0, \dots, 0)$ und $E = (0, \dots, 0, 1, \dots, 1, 0, \dots, 0)$.

Fall 4-6: $A = (1, \dots, 1, 0, \dots, 0, 1, \dots, 1)$

Die Fallunterscheidungen entsprechen denen der Fälle 1 bis 3 und die Beweise verlaufen analog.

□

Der Beweis des Lemmas kann jetzt durch Induktion nach k geführt werden.

Ind. Anf.: $k = 1$

In diesem Fall wird durch die if-Abfrage in `bitonic_merge` die korrekte Sortierung der beiden Zahlen hergestellt.

Ind. Vor.:

Auf $Q(k - 1)$ wird jede bitone Folge durch den Algorithmus korrekt sortiert.

Ind. Schritt: $k - 1 \mapsto k$

Der Aufruf `bitonic_merge` ($\alpha, k - 1$) hat die folgende Wirkung:

$$\forall \beta \in \{0, 1\}^{k-1} : A[0\beta] := D[\beta] \text{ und } A[1\beta] := E[\beta]$$

$D[\beta]$ und $E[\beta]$ sind nach der Zwischenbehauptung `biton` und werden anschließend nach der Ind. Vor. auf $Q(k - 1)$ korrekt sortiert. Wegen $\max(D) \leq \min(E)$ ist dann ganz A sortiert. ■

Mit Hilfe des bitonen Sortierers kann nun ein rekursiver Algorithmus formuliert werden, der beliebige Zahlenfolgen korrekt sortiert.

```
for all  $\alpha \in \{0, 1\}^{k-1}$  do
  sortiere die Zahlen  $A[0\alpha]$  aufsteigend
  sortiere die Zahlen  $A[1\alpha]$  absteigend
  /*  $A$  ist jetzt biton nach Lemma 4.1 Teil 3 */
  DESCEND(bitonic_merge)
od
```

Der Algorithmus besitzt eine Rekursionstiefe von k . Da in jeder Rekursionsstufe ein `DESCEND`(`bitonic_merge`) Aufruf abgearbeitet werden muß, ergibt sich ein Gesamtaufwand von k^2 . Der Algorithmus kann iterativ wie folgt formuliert werden:

```
procedure bitonic_sort ( $\alpha : \{0, 1\}^k$ ;  $\text{dim} : 0 \dots k - 1$ )
var  $\alpha' : \{0, 1\}^k$ ; mode : {aufsteigend, absteigend};
begin
  if  $\text{dim} \leq r$  then begin /* DESCEND durch die Dimensionen  $r \dots 0$  */
     $\alpha' = \alpha(\text{dim})$ ;
    if  $r = k - 1$  then mode:=aufsteigend; /* letzter DESCEND Lauf sortiert aufsteigend */
    else if ( $\text{bit}(\alpha, r + 1) = 0$ ) then mode:=aufsteigend;
    else mode:=absteigend;
    case mode of
      aufsteigend :
        if ( $\text{bit}(\alpha, \text{dim}) = 0$  and  $A[\alpha] > A[\alpha']$ ) then tausche( $A[\alpha], A[\alpha']$ );
      absteigend :
        if ( $\text{bit}(\alpha, \text{dim}) = 0$  and  $A[\alpha] < A[\alpha']$ ) then tausche( $A[\alpha], A[\alpha']$ );
    end;
  end;
end;
begin /* Hauptprogramm */
  for  $r := 0$  to  $k - 1$  do DESCEND(bitonic_sort);
end.
```

A[000	001	010	011	100	101	110	111]			
	7	3	1	6	2	5	4	8				
	3	7		6	1		2	5		8	4	Vertauschungen des 1. Aufrufs (r=0)
	3	1	6	7		8	5	2	4			Vertauschungen des 2. Aufrufs (r=1)
	1	3	6	7		8	5	4	2			
	1	3	4	2	8	5	6	7				
	1	2	4	3	6	5	8	7				Vertauschungen des 3. Aufrufs (r=2)
	1	2	3	4	5	6	7	8				

Abbildung 22: *Vertauschungen beim bitonen Sortieren auf $Q(3)$*

Wir fassen die Ergebnisse des Abschnitts zusammen und erhalten:

Satz 4.1 (bitones Sortieren auf der Hypercube)

2^k Zahlen können auf der Hypercube $Q(k)$ in $O(k^2)$ Kommunikationsschritten sortiert werden.

Abbildung 22 veranschaulicht die Vorgehensweise des Algorithmus für $k = 3$. Beim ersten Aufruf von `DESCEND(bitonic_sort)` gilt $r = 0$. Wegen der Abfrage „if $\dim \leq r$ “ ist der `DESCEND` Algorithmus nur in Dimension 0 aktiv. Beim zweiten Aufruf (es gilt also $r = 2$) durchläuft der `DESCEND` Algorithmus die Dimensionen 0 und 1 und beim letzten Aufruf ($r = 3$) die Dimensionen 2, 1 und 0. Abbildung 22 zeigt die in den einzelnen Aufrufen durchgeführten Vertauschungen.

4.3 ASCEND/DESCEND Algorithmen auf anderen Netzen der Hypercube-Familie

In diesem Abschnitt wollen wir untersuchen wie `ASCEND/DESCEND` Algorithmen auf dem $CCC(k)$ und auf dem $SE(k)$ implementiert werden können. Hierbei wird ein großer Vorteil der `ASCEND/DESCEND` Algorithmen offensichtlich. Aufgrund der einheitlichen Struktur dieser Algorithmen reicht es nämlich aus zu zeigen, wie die Prozeduren `ASCEND` und `DESCEND` auf dem $CCC(k)$ bzw. $SE(k)$ simuliert werden können. Da $CCC(k)$ nach Satz 3.12 Teilgraph des $BF(k)$ und $SE(k)$ nach Satz 3.17 Teilgraph des $DB(k)$ ist, können wir nach Ende des Abschnitts einen `ASCEND/DESCEND` Algorithmus auf jedem Netzwerk der Hypercube-Familie mit nur konstantem Zeitverlust ausführen. Wir beginnen unsere Untersuchungen mit der Simulation eines `ASCEND/DESCEND` Laufs auf dem $SE(k)$.

4.3.1 ASCEND/DESCEND auf $SE(k)$

Wir betrachten zunächst die Prozedur `ASCEND`. Eine Operation auf dem Datenpaar $A[\alpha]$ und $A[\alpha(\dim)]$ wird immer durch die Kommunikation über eine Exchange-Kante des $SE(k)$

realisiert. In Dimension 0 ist dies ohne weiteres möglich, da sich die Datenpaare passend an den Exchange-Kanten gegenüber stehen. Bevor die Kommunikation in höheren Dimensionen ausgeführt werden kann, müssen die Prozessoren über die Shuffle-Kanten des $SE(k)$ ihre Information austauschen. Dadurch ändert sich auch der Hypercube-Prozessor, der von dem Shuffle-Exchange-Prozessor simuliert wird. Deshalb merken sich in allen folgenden Simulationen die Prozessoren im Array HC, welchen Hypercube-Prozessor sie gerade simulieren. Sei s die bekannte Shuffle-Funktion. Die Prozedur ASCEND kann wie folgt auf dem $SE(k)$ implementiert werden:

```

Procedure ASCEND_SE (procedure OPER)
var  $\alpha : \{0, 1\}^k$ ;  dim : 0 ...  $k - 1$ ;  HC : array[ $\{0, 1\}^k$ ] of  $\{0, 1\}^k$ ;
begin
  for all  $\alpha \in \{0, 1\}^k$  parallel do
    HC[ $\alpha$ ] =  $\alpha$ 
  od
  for dim := 0 to  $k - 1$  do
    for all  $\alpha \in \{0, 1\}^k$  parallel do
      OPER(HC[ $\alpha$ ], dim)
    od
    for all  $\alpha \in \{0, 1\}^k$  parallel do
      schicke die Daten von Prozessor  $\alpha$  nach Prozessor  $s^{-1}(\alpha)$ 
      HC[ $\alpha$ ] =  $s(\text{HC}[\alpha])$ 
    od
  od
end

```

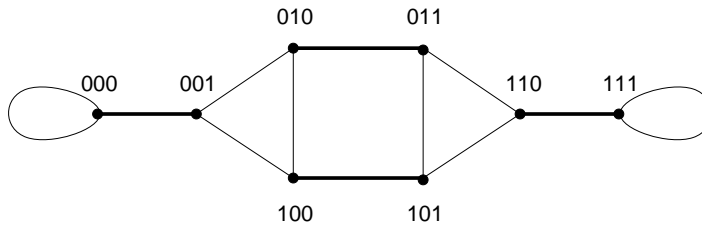
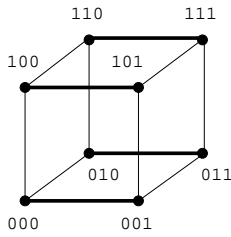
Abbildung 23 verdeutlicht die Vorgehensweise anhand der Simulation eines ASCEND Laufs des $Q(3)$ auf dem $SE(3)$. Dabei ist jede Prozessornummer des $SE(3)$ mit einem weiteren Bitstring versehen, der den Inhalt des HC-Arrays darstellt, also den simulierten Hypercube-Prozessor ausweist. Die Kanten, über die in den einzelnen Dimensionen kommuniziert wird, sind fett dargestellt. Die Prozedur DESCEND kann man ganz ähnlich implementieren. Es ändert sich lediglich die Richtung, in der die Daten über die Shuffle-Kanten ausgetauscht werden. Darüberhinaus erfolgt der Datenaustausch über die Shuffle-Kanten bevor die Prozedur OPER aufgerufen wird.

```

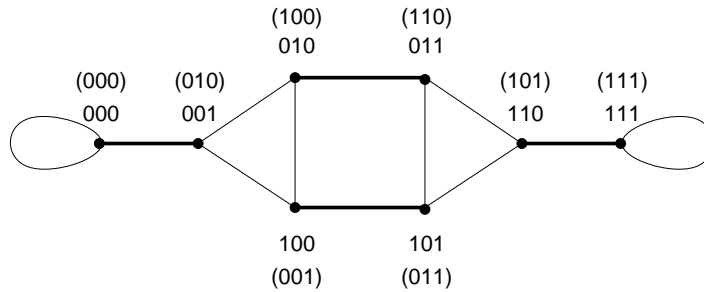
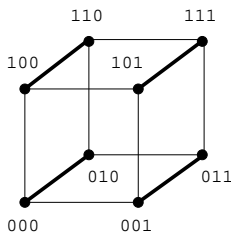
Procedure DESCEND_SE (procedure OPER)
var  $\alpha : \{0, 1\}^k$ ;  dim : 0 ...  $k - 1$ ;  HC : array[ $\{0, 1\}^k$ ] of  $\{0, 1\}^k$ ;
begin
  for all  $\alpha \in \{0, 1\}^k$  parallel do
    HC[ $\alpha$ ] =  $\alpha$ 
  od
  for dim :=  $k - 1$  downto 0 do
    for all  $\alpha \in \{0, 1\}^k$  parallel do
      schicke die Daten von Prozessor  $\alpha$  nach Prozessor  $s(\alpha)$ 
      HC[ $\alpha$ ] =  $s^{-1}(\text{HC}[\alpha])$ 
    od
  od
  for all  $\alpha \in \{0, 1\}^k$  parallel do

```

Kommunikation in Dimension 0



Kommunikation in Dimension 1



Kommunikation in Dimension 2

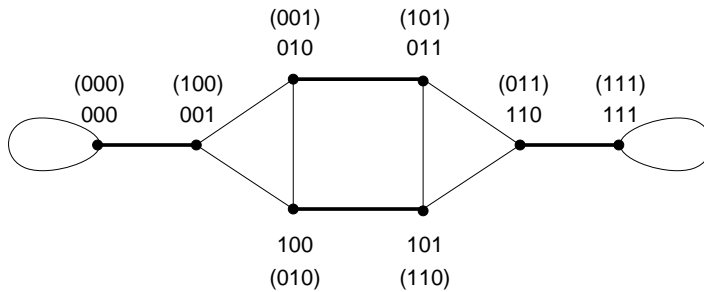
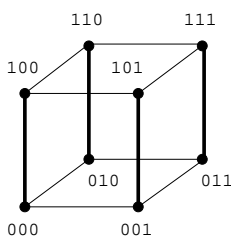


Abbildung 23: Simulation eines ASCEND Laufs auf dem SE(3)

```

OPER(HC[α], dim)
od
od
end
    
```

Korrektheit und Laufzeit der Algorithmen ASCEND_SE und DESCEND_SE zeigt der folgende Satz.

Satz 4.2 (ASCEND/DESCEND auf SE(k))

Ein ASCEND/DESCEND Lauf des $Q(k)$ kann auf dem $SE(k)$ in Zeit $O(k)$ abgearbeitet werden.

Beweis:

Im folgenden betrachten wir nur die Prozedur ASCEND_SE. Der Satz kann für DESCEND_SE analog bewiesen werden.

zur Korrektheit:

Bevor in Iteration dim , $0 \leq \text{dim} < k$, die Prozedur OPER aufgerufen wird, wurde bereits dim -mal die Anweisung

```

for all  $\alpha \in \{0, 1\}^k$  parallel do
  schicke die Daten von Prozessor  $\alpha$  nach Prozessor  $s^{-1}(\alpha)$ 
   $\text{HC}[\alpha] = s(\text{HC}[\alpha])$ 
od

```

ausgeführt. Der Hypercube-Prozessor $\alpha = \alpha_{k-1} \dots \alpha_0$ wird also vom Shuffle-Exchange-Prozessor $\alpha' = s^{-\text{dim}}(\alpha) = \alpha_{\text{dim}-1} \dots \alpha_0 \alpha_{k-1} \dots \alpha_{\text{dim}}$ simuliert und es gilt $\text{HC}[\alpha'] = \alpha$. Eine Operation auf dem Datenpaar $A[\text{HC}[\alpha']]$, $A[\text{HC}[\alpha'](\text{dim})]$ kann also durch eine Kommunikation über die Exchange-Kante $\{\alpha', \alpha'(0)\}$ realisiert werden.

zur Laufzeit:

In Prozedur ASCEND_SE findet zusätzlich in jeder Iteration ein Datenaustausch über die Shuffle-Kanten statt. Daher beträgt die Laufzeit $2k$. ■

4.3.2 ASCEND/DESCEND auf CCC(k)

Wir untersuchen zunächst, wie ein ASCEND/DESCEND Algorithmus auf einem linearen (Prozessoren-) Array implementiert werden kann. Ein lineares Array der Länge n ist ein ungerichteter Graph $G = (V, E)$ mit $V = \{1, \dots, n\}$ und $E = \{\{i, i+1\}; 1 \leq i < n\}$.

Sei π eine beliebige Permutation auf den Zahlen $1, \dots, n$. Unter dem Begriff Permutationsrouting versteht man die Aufgabe, für alle $1 \leq i \leq n$ eine Nachricht von Prozessor i nach Prozessor $\pi(i)$ zu routen. Wir haben bereits gesehen, wie die Permutation π_{REV} auf einem Hypercube geroutet werden kann. Für das Routen beliebiger Permutationen auf dem linearen Array gilt:

Lemma 4.4

Auf einem linearen Array der Länge n kann Permutationsrouting in Zeit $2(n-1)$ und Puffergröße 3 durchgeführt werden.

Beweis:

Sei π eine beliebige Permutation auf den Zahlen $1, \dots, n$. Zuerst schicken alle Prozessoren i mit $\pi(i) < i$ ihre Nachricht über Pipelining nach links zum Zielprozessor $\pi(i)$. Danach schicken alle Prozessoren j mit $j < \pi(j)$ ihre Nachricht über Pipelining nach rechts zum Zielprozessor $\pi(j)$. Das Pipelining ist in beiden Fällen nach $n-1$ Schritten abgeschlossen. Zu jedem Zeitpunkt ist pro Prozessor i maximal die Nachricht von Prozessor i selbst, die Nachricht von Prozessor j mit $\pi(j) = i$ und die an einen anderen Prozessor weiterzuleitende Nachricht abgespeichert. Daher ist ein Puffer der Größe 3 ausreichend. ■

Definition 4.5 (unshuffle-Permutation)

Sei $i_{k-1} \dots i_0$ die Binärdarstellung der Zahl i . Dann ist für $0 \leq l < k$ $\text{unshuffle}_l(i)$ definiert durch:

$$\text{unshuffle}_l(i) = \text{unshuffle}_l(i_{k-1} \dots i_{k-l} i_{k-l-1} \dots i_0) = i_{k-1} \dots i_{k-l} i_0 i_{k-l-1} \dots i_1$$

Mit Hilfe der Permutation unshuffle kann jetzt ein ASCEND Lauf des $Q(k)$ auf einem linearen Array der Länge $n = 2^k$ simuliert werden. Der Algorithmus arbeitet wie folgt:

```

Procedure ASCEND_lin_Array (procedure OPER)
var  $\alpha : \{0, 1\}^k$ ;   dim :  $0 \dots k - 1$ ;   HC : array[ $\{0, 1\}^k$ ] of  $\{0, 1\}^k$ ;
begin
  for all  $\alpha \in \{0, 1\}^k$  parallel do
    HC[ $\alpha$ ] =  $\alpha$ 
  od
  for dim := 0 to  $k - 1$  do
    for all  $\alpha \in \{0, 1\}^k$  parallel do
      OPER(HC[ $\alpha$ ], dim)
    od
    for all  $\alpha \in \{0, 1\}^k$  parallel do
      schicke die Daten von Prozessor  $\alpha$  nach Prozessor  $\text{unshuffle}_{\text{dim}}(\alpha)$ 
      HC[ $\alpha$ ] = HC[ $\text{unshuffle}_{\text{dim}}^{-1}(\alpha)$ ]
    od
  od
end

```

Abbildung 24 verdeutlicht die Vorgehensweise anhand der Simulation eines ASCEND Laufs des $Q(3)$ auf dem linearen Array der Länge 8. Die Prozessornummern des linearen Arrays sind binär dargestellt. Jede Prozessornummer ist mit einem weiteren Bitstring versehen, der den Inhalt des HC-Arrays darstellt, also den simulierten Hypercube-Prozessor ausweist. Die Kanten, über die in den einzelnen Dimensionen kommuniziert wird, sind fett dargestellt. Nach Aufruf der Prozedur OPER in Dimension 1 schickt beispielsweise der Prozessor 010 die von Prozessor 100 übernommene Nachricht weiter an Prozessor $\text{unshuffle}_1(010) = 001$. Vor Aufruf der Prozedur OPER in Dimension 2 ist damit auf Prozessor 001 die Nachricht von Prozessor 100 gespeichert.

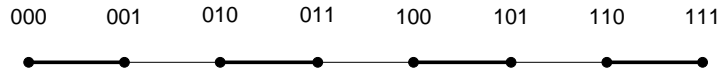
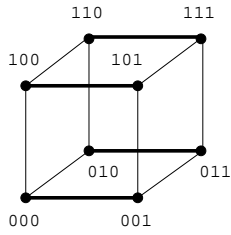
Die neue Prozessornummer in $\text{HC}[\alpha]$ kann auch direkt aus dem alten Wert berechnet werden. Hierfür benötigt man die Funktion $\text{xshuffle}_l(i)$, die (ähnlich wie unshuffle_l) wie folgt definiert ist:

$$\text{xshuffle}_l(i) = \text{xshuffle}_l(i_{k-1} \dots i_l i_{l-1} \dots i_0) = i_{k-2} \dots i_l i_{k-1} i_{l-1} \dots i_0$$

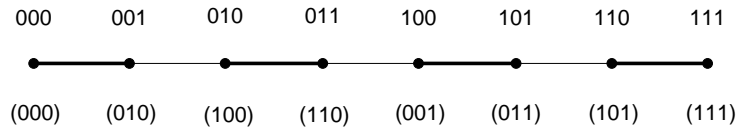
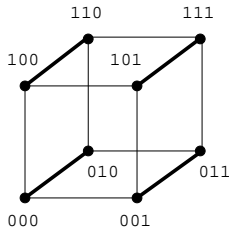
Dann lässt sich $\text{HC}[\alpha] = \text{HC}[\text{unshuffle}_{\text{dim}}^{-1}(\alpha)]$ durch $\text{HC}[\alpha] = \text{xshuffle}_{\text{dim}}(\text{HC}[\alpha])$ ersetzen.

Am Ende der ASCEND-Simulation entspricht die Prozessorzuordnung nicht mehr der am Anfang der Simulation; Prozessor $\alpha = \alpha_{k-1} \dots \alpha_0$ des linearen Arrays hat die Rolle des Hypercube-Prozessors $\alpha_0 \dots \alpha_{k-1}$ inne (Bit-Reversal-Permutation). Ist dies nicht erwünscht, muß nach Abschluß des eigentlichen ASCEND-Laufs einmal die Bit-Reversal-Permutation auf dem Array ausgeführt werden, was auf die asymptotische Laufzeit keinen Einfluß hat.

Kommunikation in Dimension 0



Kommunikation in Dimension 1



Kommunikation in Dimension 2

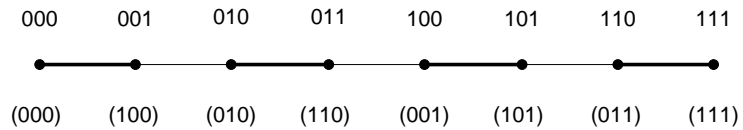
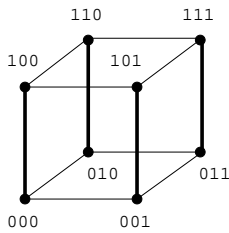


Abbildung 24: Simulation eines ASCEND Laufs auf dem linearen Array der Länge 8

Ein DESCEND Lauf kann ähnlich simuliert werden. Die Einzelheiten seien dem Leser überlassen. Korrektheit und Laufzeit des Algorithmus ASCEND_lin_Array zeigt der folgende Satz.

Satz 4.3 (ASCEND/DESCEND auf linearem Array)

Ein ASCEND/DESCEND Lauf des $Q(k)$ kann auf dem linearen Array der Länge $n = 2^k$ in Zeit $O(n)$ abgearbeitet werden.

Beweis:

Im folgenden betrachten wir nur die Prozedur ASCEND_lin_Array. Der Satz kann für DESCEND_lin_Array analog bewiesen werden.

zur Korrektheit:

Bevor in Iteration dim , $0 \leq \text{dim} < k$, die Prozedur OPER aufgerufen wird, wurde bereits dim -mal die Anweisung

for all $\alpha \in \{0, 1\}^k$ parallel do
 schicke die Daten von Prozessor α nach Prozessor $\text{unshuffle}_{\text{dim}}(\alpha)$
 $\text{HC}[\alpha] = \text{HC}[\text{unshuffle}_{\text{dim}}^{-1}(\alpha)]$
 od

ausgeführt. Sei $\alpha = \alpha_{k-1} \dots \alpha_0$. Es gilt:

$$\text{unshuffle}_{\text{dim}-1}(\text{unshuffle}_{\text{dim}-2}(\dots(\text{unshuffle}_0(\alpha))\dots)) = \alpha_0 \dots \alpha_{\text{dim}-1} \alpha_{k-1} \dots \alpha_{\text{dim}}$$

Also wird der Hypercube-Prozessors α vom Prozessor $\alpha' = \alpha_0 \dots \alpha_{\text{dim}-1} \alpha_{k-1} \dots \alpha_{\text{dim}}$ des linearen Arrays simuliert und es gilt $\text{HC}[\alpha'] = \alpha$. Eine Operation auf dem Datenpaar $A[\text{HC}[\alpha']]$, $A[\text{HC}[\alpha'](\text{dim})]$ kann also durch eine Kommunikation über Kante $\{\alpha', \alpha'(0)\}$ des linearen Arrays realisiert werden.

zur Laufzeit:

Die Gesamtlaufzeit wird bestimmt von der Zeit, die zum Routen von $\text{unshuffle}_{\text{dim}}$ benötigt wird. Für $\alpha \in \{0, 1\}^k$ gilt, daß $\text{unshuffle}_{\text{dim}}(\alpha)$ die obersten dim bits festhält. In dem linearen Array beträgt daher der Abstand zwischen den Prozessoren α und $\text{unshuffle}_{\text{dim}}(\alpha)$ höchstens $2^{k-\text{dim}}$. Nach Lemma 4.4 kann das Routing der Permutation $\text{unshuffle}_{\text{dim}}$ in Zeit $O(2 \cdot (2^{k-\text{dim}} - 1))$ durchgeführt werden. Die Gesamtlaufzeit berechnet sich damit zu:

$$O\left(2 \cdot \sum_{\text{dim}=0}^{k-1} (2^{k-\text{dim}} - 1)\right) = O(2^k) = O(n)$$

■

Satz 4.4 (ASCEND/DESCEND auf CCC(k))

Sei $k = 2^i, i \in \mathbb{N}$. Ein ASCEND/DESCEND Lauf des $Q(k + \log k)$ kann auf $\text{CCC}(k)$ in Zeit $O(k)$ abgearbeitet werden.

Beweis:

Der Hypercube $Q(k + \log k)$ besteht aus den Teilcubes $Q(k)$ und $Q(\log k)$. Der Bitstring $x \in \{0, 1\}^{k+\log k}$ eines Knotens aus $Q(k + \log k)$ setzt sich zusammen aus dem Bitstring $u \in \{0, 1\}^k$ eines Knotens aus $Q(k)$ und dem Bitstring $v \in \{0, 1\}^{\log k}$ eines Knotens aus $Q(\log k)$. Wir schreiben daher für x im folgenden uv . Der Knoten uv wird nun abgebildet auf den Knoten (i, u) , $(i)_2 = v$, des $\text{CCC}(k)$. Dadurch haben wir eine Einbettung des $Q(k + \log k)$ in den $\text{CCC}(k)$ definiert. Die Simulation eines ASCEND Laufs in $\text{CCC}(k)$ wird in zwei Phasen durchgeführt.

1.Phase: ASCEND entlang der Dimensionen $0, \dots, \log k - 1$

Die Kommunikation entlang der Dimensionen $0, \dots, \log k - 1$ des $Q(k + \log k)$ verläuft innerhalb der Kreise $(0, u), (1, u), \dots, (k-1, u)$ des $\text{CCC}(k)$. Mit Hilfe von `ASCEND_lin_Array` kann der Lauf in Zeit $O(k)$ abgearbeitet werden.

2.Phase: ASCEND entlang der Dimensionen $\log k, \dots, \log k + k - 1$

Es verbleibt noch ein ASCEND Lauf entlang der Dimensionen $\log k, \dots, \log k + k - 1$. Die Kanten der Dimensionen $\log k, \dots, k + \log k - 1$ sind auf die Hypercube-Kanten $\{(i, u), (i, u(i))\}$,

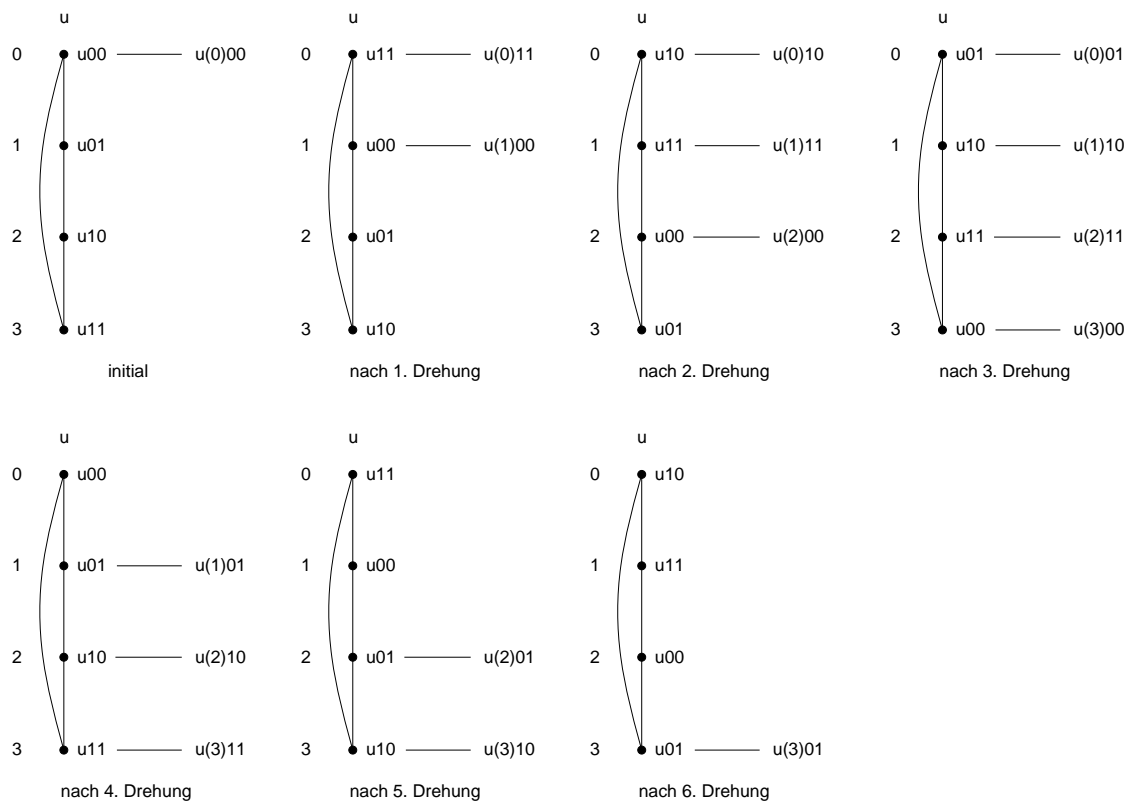


Abbildung 25: *Simulation eines ASCEND Laufs auf dem CCC(3)*

$0 \leq i < k$ des $CCC(k)$ abgebildet. Wir drehen die auf den Kreis $(0, u), (1, u), \dots, (k-1, u)$ abgebildeten Knoten $u0 \dots 0, u0 \dots 1, u1 \dots 1$ des $Q(k + \log k)$ so an den Hypercube-Kanten des $CCC(k)$ vorbei, daß die in Dimension $\dim, \log k \leq \dim < k + \log k$, des ASCEND Laufs kommunizierenden Prozessoren entlang der Hypercube-Kante $\dim - \log k$ des $CCC(k)$ ihre Nachricht austauschen können. Die Drehrichtung ist auf allen Kreisen $(0, u), (1, u), \dots, (k-1, u)$ des $CCC(k)$ gleich. Der auf Knoten $(1, u)$ abgebildete Knoten $u0 \dots 1$ beendet den ASCEND Lauf als letzter. Es sind zunächst $k - 1$ Drehungen notwendig bis er den Knoten $(0, u)$ erreicht. Erst jetzt kann er mit seinem ASCEND Lauf beginnen. Dafür sind noch einmal $k - 1$ Drehungen notwendig. Insgesamt ergibt sich so für Phase 2 eine Laufzeit von $2 \cdot (k - 1)$.

■

Sei $u \in \{0, 1\}^4$. Initial kann $u00$ mit $u(0)00$ über die in Dimension 0 verlaufende Hypercube-Kante kommunizieren. Nach der ersten Drehung kann $u00$ mit $u(1)00$ über die in Dimension 1 und $u11$ mit $u(0)11$ über die in Dimension 0 verlaufende Hypercube-Kante kommunizieren. Abbildung 25 zeigt die nach den einzelnen Drehungen stattfindende Kommunikation.

4.4 Fast-Fourier-Transformation (FFT)

In diesem Abschnitt befassen wir uns mit der schnellen Berechnung der (diskreten) Fourier-Transformation und den Netzwerken, auf denen diese Berechnung effizient parallel ausgeführt

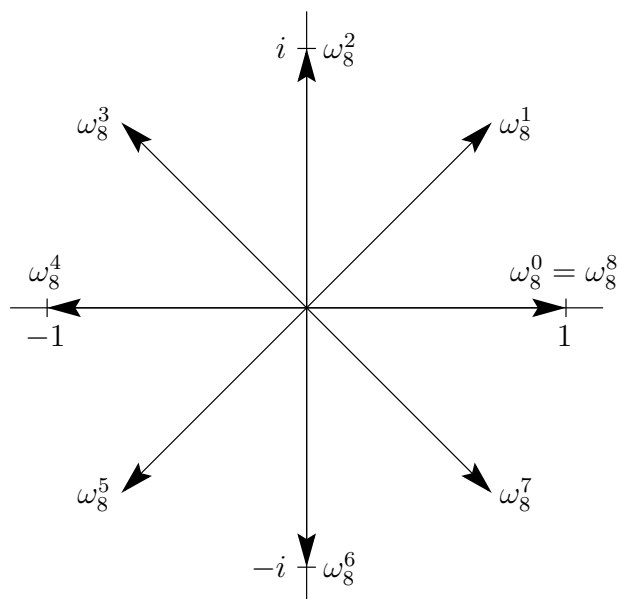


Abbildung 26: Die 8-ten Einheitswurzeln und ihre Lage in der komplexen Ebene

werden kann. Bei der Fourier-Transformation handelt es sich um eine grundlegende Operation, die in vielen Bereichen, wie der Signalverarbeitung, dem Lösen von Differentialgleichungen, der Polynom-Multiplikation, usw., Anwendung findet.

Bei der Fourier-Transformation handelt es sich, allgemein gesprochen, einfach um eine Matrix-Vektor-Multiplikation, wobei die Matrix eine sehr spezielle Form aufweist. Die in diesem Abschnitt vorgestellte Fast-Fourier-Transformation ist ein Algorithmus, der die Fourier-Transformation unter Ausnutzung der speziellen Matrix-Eigenschaften in sequentieller Zeit $O(n \log n)$ berechnen kann.

Definition 4.6 (*n*-te Einheitswurzel)

Sei $n \in \mathbb{N}$. $\omega \in \mathbb{C}$ heißt *n*-te Einheitswurzel, wenn $\omega^n = 1$ gilt.

Ohne Beweis geben wir folgenden Satz an:

Satz 4.5

Sei $n \in \mathbb{N}$. Es gibt genau *n* *n*-te Einheitswurzeln, nämlich $e^{2\pi il/n}$ für $l = 0, \dots, n - 1$.

Definition 4.7 (*n*-te Haupteinheitswurzel)

Sei $n \in \mathbb{N}$. $\omega_n = e^{2\pi i/n}$ heißt *n*-te Haupteinheitswurzel.

Somit gilt für alle *n*-ten Einheitswurzeln, daß sie eine Potenz der *n*-ten Haupteinheitswurzel sind. Unter Verwendung der Definition $e^{iu} = \cos(u) + i \sin(u)$ sieht man, daß die *n*-ten Einheitswurzeln gleichmäßig auf dem Einheitskreis mit Zentrum im Ursprung der komplexen Ebene angeordnet sind (Abbildung 26).

Definition 4.8 (Fourier-Transformation)

Sei $n \in \mathbb{N}$ und $F_n = (\omega_n^{ij})_{0 \leq i, j < n} \in \mathbb{C}^{n \times n}$. Die (diskrete) Fourier-Transformation eines Vektors $x \in \mathbb{C}^n$ ist definiert durch $y = F_n x \in \mathbb{C}^n$, dem Matrix-Vektor-Produkt von F_n und x .

y_i läßt sich also mittels $y_i = \sum_{0 \leq j < n} \omega_n^{ij} x_j$ berechnen. Für $n = 4$ ergibt sich beispielsweise:

$$\omega_4 = i \quad \text{und} \quad F_4 = \begin{pmatrix} \omega_4^0 & \omega_4^0 & \omega_4^0 & \omega_4^0 \\ \omega_4^0 & \omega_4^1 & \omega_4^2 & \omega_4^3 \\ \omega_4^0 & \omega_4^2 & \omega_4^4 & \omega_4^6 \\ \omega_4^0 & \omega_4^3 & \omega_4^6 & \omega_4^9 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{pmatrix}$$

Der triviale Ansatz zur Berechnung der Fourier-Transformation, einfach die Matrix-Vektor-Multiplikation auszuführen, benötigt folglich sequentielle Zeit $O(n^2)$. Unter Ausnutzung der speziellen Struktur von F_n läßt sich diese Laufzeit durch einen Divide & Conquer-Ansatz auf $O(n \log n)$ reduzieren. Die Divide & Conquer-Idee beschreibt das folgende Lemma.

Lemma 4.5

Sei $n \in \mathbb{N}$ gerade und $x = (x_0, \dots, x_{n-1}) \in \mathbb{C}^n$. Die Fourier-Transformation $y = F_n x$ läßt sich wie folgt berechnen. Seien $u = F_{n/2}(x_0, x_2, \dots, x_{n-2})$ und $v = F_{n/2}(x_1, x_3, \dots, x_{n-1})$. Dann gilt für alle $0 \leq i < n$:

$$y_i = \begin{cases} u_i + \omega_n^i v_i & , \text{ falls } 0 \leq i < n/2 \\ u_{i-n/2} - \omega_n^{i-n/2} v_{i-n/2} & , \text{ sonst} \end{cases}$$

Beweis:

Es ist $\omega_n^2 = (e^{2\pi i/n})^2 = e^{2\pi i/(n/2)} = \omega_{n/2}$ und deshalb:

$$\begin{aligned} y_i &= \sum_{0 \leq j < n} \omega_n^{ij} x_j = \sum_{0 \leq j < n/2} (\omega_n^{2ij} x_{2j} + \omega_n^{i(2j+1)} x_{2j+1}) \\ &= \sum_{0 \leq j < n/2} \omega_n^{2ij} x_{2j} + \sum_{0 \leq j < n/2} \omega_n^{i(2j+1)} x_{2j+1} \\ &= \sum_{0 \leq j < n/2} \omega_{n/2}^{ij} x_{2j} + \omega_n^i \sum_{0 \leq j < n/2} \omega_{n/2}^{ij} x_{2j+1} \end{aligned}$$

Für $i < n/2$ folgt also unmittelbar $y_i = u_i + \omega_n^i v_i$. Sei nun $i \geq n/2$ und $r = i - n/2$. Es ist $0 \leq r < n/2$ und wegen $\omega_{n/2}^{n/2} = 1$ ist $\omega_n^{i-n/2} = \omega_{n/2}^{n/2+r} = \omega_{n/2}^r$. Ferner gilt $\omega_n^{n/2} = \omega_2 = -1$ und folglich:

$$\begin{aligned} y_i &= \sum_{0 \leq j < n/2} \omega_{n/2}^{ij} x_{2j} + \omega_n^i \sum_{0 \leq j < n/2} \omega_{n/2}^{ij} x_{2j+1} \\ &= \sum_{0 \leq j < n/2} \omega_{n/2}^{rj} x_{2j} + \omega_n^i \sum_{0 \leq j < n/2} \omega_{n/2}^{rj} x_{2j+1} \\ &= u_r + \omega_n^i v_r = u_{i-n/2} + \omega_n^i v_{i-n/2} = u_{i-n/2} + \omega_n^{n/2+(i-n/2)} v_{i-n/2} \\ &= u_{i-n/2} - \omega_n^{i-n/2} v_{i-n/2} \end{aligned}$$

■

Die Berechnung der Fourier-Transformation eines n -elementigen Vektors läßt sich somit auf die Berechnung von zwei Fourier-Transformationen mit $(n/2)$ -elementigen Vektoren zurückführen. Dieses Verfahren läßt sich rekursiv solange fortführen, bis man zu n Transformationen mit 1-elementigen Vektoren gelangt. Es ist $F_1 = (1)$ und man erhält:

```

procedure FFT_rekursiv (var X : array[0..2k - 1] of complex)
var U, V : array[0..2k-1 - 1] of complex;  i : 0...2k-1 - 1;  o : complex;
begin
  if k = 0 then return
  for i := 0 to 2k-1 - 1 do
    U[i] := X[2*i]
    V[i] := X[2*i+1]
  od
  FFT_rekursiv(U)
  FFT_rekursiv(V)
  o := 1
  for i := 0 to 2k-1 - 1 do
    X[i] := U[i] + o*V[i]
    X[i+2k-1] := U[i] - o*V[i]
    o := o*ω2k
  od
end

```

Satz 4.6 (Fast-Fourier-Transformation)

Sei $n = 2^k$, $k \in \mathbb{N}$, eine Zweierpotenz und $x \in \mathbb{C}^n$. $F_n x$ ist mittels FFT_rekursiv in Zeit $O(n \log n)$ berechenbar.

Beweis:

Die Korrektheit von FFT_rekursiv folgt unmittelbar aus Lemma 4.5. Wir zeigen durch Induktion nach k , daß $y = F_{2^k} x$ in $T(k) \leq c \cdot 2^k \cdot k$ Schritten (bei passend gewähltem c) berechenbar ist:

Ind. Anf.: $k = 1$

y ist in konstant vielen Schritten berechenbar: $T(1) = c' \leq c \cdot 2^1 \cdot 1$.

Ind. Vor.:

Sei $F_{2^k} x$ in $T(k) \leq c \cdot 2^k \cdot k$ Schritten berechenbar.

Ind. Schritt: $k \mapsto k + 1$

Für die Anzahl benötigter Schritte von FFT_rekursiv bei Eingabe eines 2^{k+1} -elementigen Vektors gilt:

$$T(k+1) = 2 \cdot T(k) + c'' \cdot 2^k \leq 2 \cdot c \cdot 2^k \cdot k + c \cdot 2^{k+1} = c \cdot 2^{k+1} \cdot (k+1)$$

■

Der FFT-Algorithmus läßt sich auch graphisch als eine Art arithmetisches Schaltnetzwerk beschreiben. Die grundlegende Operation ist dabei in Abbildung 27 dargestellt, die Berechnung

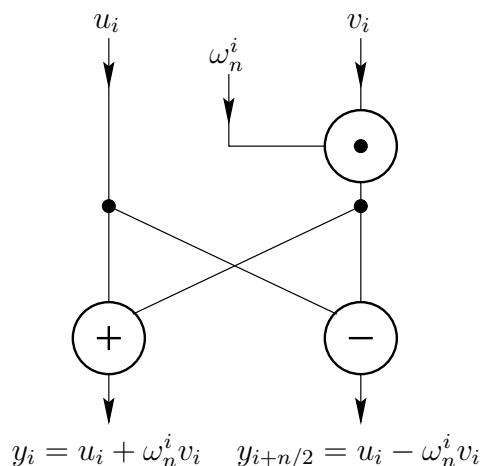


Abbildung 27: Basisoperation des FFT-Algorithmus

von y_i und $y_{i+n/2}$ mittels u_i und v_i . Abbildung 28 zeigt das sich daraus ergebende komplette Netzwerk für $n = 8$.

Der dem FFT-Algorithmus für $n = 2^k$ zugrundeliegende Graph, FFT-Netzwerk genannt, entspricht dabei dem Butterfly-Netzwerk $BF(k)$ „ohne Wrap-Around-Kanten“. Auf diesem Graphen kann der FFT-Algorithmus parallel in k Runden ausgeführt werden; dabei sind pro Runde nur die n Knoten einer Schicht aktiv. Bei der Ausführung ist zu beachten, daß der Eingabevektor $x = (x_0, \dots, x_{n-1})$ permutiert werden muß: Die Aufteilung des Eingabevektors in FFT_rekursiv erfolgt gemäß des letzten Bits (gerade/ungerade) der Indexposition im Vektor. Diese Aufteilung wird rekursiv fortgesetzt, und man sieht daher leicht, daß man den Eingabevektor mittels π_{BRP} permutieren muß. π_{BRP} ist die sogenannte bit-reversal-Permutation auf $\{0, \dots, 2^k - 1\}$ mit $\pi_{BRP}((i_{k-1}, \dots, i_0)_2) = (i_0, \dots, i_{k-1})_2$.

Wie gesagt sind im FFT-Netzwerk pro Runde nur die Knoten einer Schicht aktiv. Dieses Netzwerk bietet sich daher besonders dann zur Berechnung der FFT an, wenn viele Fourier-Transformationen ausgeführt werden müssen. Diese können dann mittels Pipelining bearbeitet werden; in jeder Runde kann mit der Berechnung einer neuen Fourier-Transformation begonnen werden.

Hat man nur eine Fourier-Transformation durchzuführen, ist die Berechnung auf einem FFT-Netzwerk nicht besonders effizient, da pro Runde nicht alle Knoten aktiv sind. Bildet man das FFT-Netzwerk (bzw. das zugehörige Butterfly-Netzwerk) gemäß Satz 3.18 auf das DeBruijn-Netzwerk $DB(k)$ ab, erreicht man, daß bei der Ausführung der FFT alle 2^k Knoten des $DB(k)$ in jeder Runde aktiv sind und die Anzahl der Runden weiterhin k beträgt; die parallele Implementierung der FFT auf dem $DB(k)$ ist also optimal.

4.5 Broadcasting und Gossiping

Dieser Abschnitt befaßt sich mit einem abstrakteren Bereich des parallelen Rechnens, der Verbreitung von Informationen in Rechnernetzen. Dabei werden zwei Arten von Problemstellungen, das Broadcast- und das Gossip-Problem, genauer untersucht. Eine ausführliche

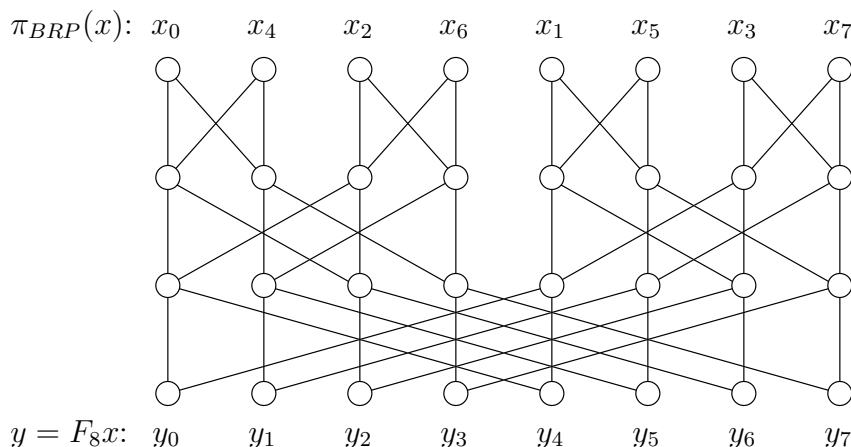


Abbildung 28: Schaltnetzwerk des FFT-Algorithmus für 8-elementige Vektoren

Einführung in dieses Themengebiet findet sich in [26].

Definition 4.9 (Broadcast-Problem)

Sei $G = (V, E)$ ein zusammenhängender Graph und $v \in V$ ein Knoten des Graphen. Im Knoten v ist eine Information $I(v)$ gespeichert, die allen anderen Knoten $u \in V \setminus \{v\}$ des Graphen unbekannt ist. Das Broadcast-Problem besteht darin, die Information $I(v)$ an alle Knoten des Graphen zu senden.

Definition 4.10 (Gossip-Problem)

Sei $G = (V, E)$ ein zusammenhängender Graph. Jeder Knoten $v \in V$ habe eine Information $I(v)$ gespeichert, die allen übrigen Knoten des Graphen unbekannt ist. Das Gossip-Problem besteht darin, allen Knoten des Graphen die kumulierte Information $I(G) = \bigcup_{v \in V} I(v)$ bekannt zu machen.

Beide Problemstellungen sind in parallelen Rechnernetzen häufig anzutreffen. Von besonderem Interesse ist in beiden Fällen, die minimale Anzahl von Kommunikationsschritten zu ermitteln, die für bestimmte Graphen benötigt werden. Diese Schrittzahl hängt neben dem Graphen auch von dem verwendeten Kommunikationsmodus ab. Wir betrachten hier, wie im ganzen Kapitel 4, nur den Telegraph- und Telephon-Modus.

4.5.1 Broadcasting

Definition 4.11

Sei $G = (V, E)$ ein zusammenhängender Graph und $v \in V$ ein Knoten des Graphen. Dann bezeichnet $b(G, v)$ die minimale Anzahl von Kommunikationsschritten im Telegraph-Modus, um das Broadcast-Problem für G von Knoten v aus zu lösen.

Wie man leicht zeigen kann, bringt es für einem Broadcast keine Vorteile, wenn man den Telephon-Modus verwenden darf; die minimale Schrittzahl ist die gleiche wie im Telegraph-Modus. Der Beweis sei dem Leser als Übungsaufgabe empfohlen. Für $b(G, v)$ lassen sich zwei einfache untere Schranken angeben:

Satz 4.7 (untere Schranken für Broadcasting)

Sei $G = (V, E)$ ein zusammenhängender Graph und $v \in V$. Dann gilt:

- 1.) $b(G, v) \geq \lceil \log |V| \rceil$
- 2.) $b(G, v) \geq \max\{\text{dist}(v, u); u \in V\}$, wobei $\text{dist}(v, u)$ für die Länge des kürzesten Weges von Knoten v nach Knoten u steht.

Beweis:

ad 1.)

Sei $r(k)$ die Anzahl der Knoten, denen von einem Broadcast-Algorithmus während der ersten k Runden die Information $I(v)$ zugesickt worden ist. Wir zeigen durch Induktion nach k , daß $r(k) \leq 2^k$ gilt, was 1.) beweist.

Ind. Anf.: $k = 0$

Vor der ersten Runde kennt nur v die Information und es gilt $r(0) = 1 \leq 2^0$.

Ind. Vor.:

Nach den ersten k Runden gelte $r(k) \leq 2^k$.

Ind. Schritt: $k \mapsto k + 1$

In Runde $k+1$ kann höchstens jeder der $r(k)$ Knoten, denen $I(v)$ bekannt ist, einen weiteren Knoten informieren. Also:

$$r(k+1) \leq 2 \cdot r(k) \leq 2 \cdot 2^k = 2^{k+1}$$

ad 2.)

Durch einen Induktionsbeweis kann man leicht zeigen, daß nach k Runden nur solche Knoten die Information $I(v)$ erhalten haben können, deren Abstand zu v kleiner gleich k ist. ■

Im folgenden wollen wir uns noch kurz mit der Eigenschaft beschäftigen, die ein Graph haben muß, um einen Broadcast in optimaler Zeit $\lceil \log |V| \rceil$ durchführen zu können. Satz 4.7 hat gezeigt, daß es schneller nicht gehen kann. Andererseits gibt es Graphen, in denen man in $\lceil \log |V| \rceil$ Runden einen Broadcast durchführen kann. Dazu muß man gewährleisten können, daß sich die Zahl der informierten Knoten in jeder Runde verdoppelt. Auf dem Hypercube ist dies möglich und für $Q(k)$ sieht ein solcher optimaler Broadcast-Algorithmus beispielsweise wie folgt aus:

```

Procedure Broadcast ( $v : \{0, 1\}^k$ )
var  $\alpha : \{0, 1\}^k$ ;  $i : 0 \dots k - 1$ ;
begin
  for  $i := 0$  to  $k - 1$  do
    for all  $\alpha \in \{v_{k-1} \dots v_i \beta; \beta \in \{0, 1\}^i\}$  parallel do
      Schicke  $I(v)$  von  $\alpha$  nach  $\alpha(i)$ 
    od
  od
end

```

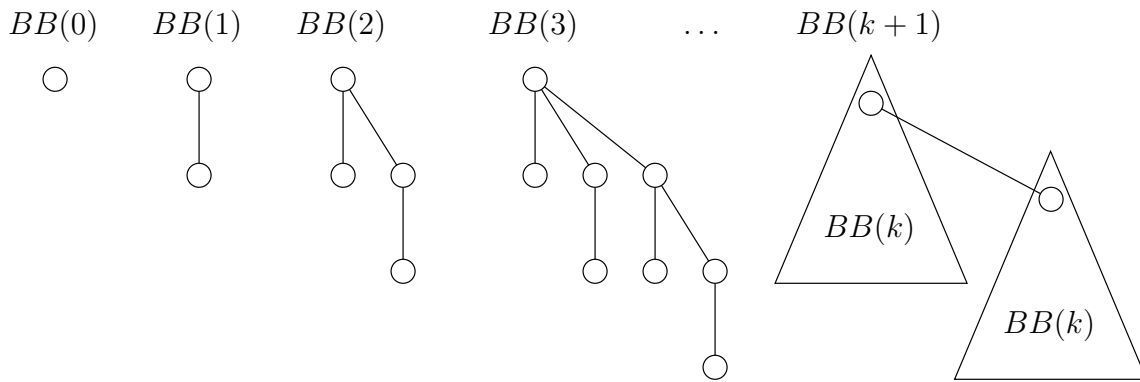


Abbildung 29: Optimale Broadcast-Bäume

Die Laufzeit beträgt offensichtlich $k = \log |V|$ und die Korrektheit folgt aus einem einfachen Induktionsbeweis, auf den an dieser Stelle verzichtet werden soll. Die einem optimalen Broadcast zugrundeliegende Struktur läßt sich als Baum beschreiben:

Definition 4.12 (optimaler Broadcast-Baum)

Die optimalen Broadcast-Bäume $BB(k)$ sind rekursiv wie folgt definiert:

- $BB(0)$ ist der Baum, der nur aus einem einzigen Knoten besteht.
- $BB(k+1)$, $k \in \mathbb{N}_0$, setzt sich aus zwei optimalen Broadcast-Bäumen der Größe k zusammen, deren Wurzeln durch eine Kante miteinander verbunden sind. Die Wurzel einer dieser beiden Teilbäume bildet die Wurzel des $BB(k+1)$.

Abbildung 29 zeigt einige optimale Broadcast-Bäume und deren rekursive Konstruktion. Aus der Konstruktion der optimalen Broadcast-Bäume folgt unmittelbar folgendes Lemma:

Lemma 4.6

Sei $G = (V, E)$ ein beliebiger Graph und $v \in V$ ein Knoten von G . Dann gilt: Von v aus lassen sich in $k \in \mathbb{N}_0$ Kommunikationsschritten im Telegraph-Modus genau dann 2^k viele Knoten aus G informieren, wenn $BB(k)$ mit Wurzel v ein Teilgraph von G ist.

Für Graphen, deren Knotenanzahl eine Zweierpotenz ist, folgt daraus:

Korollar 4.1

Sei $G = (V, E)$ ein Graph mit $|V| = 2^k$, $k \in \mathbb{N}_0$, und $v \in V$ ein Knoten von G . Folgende Aussagen sind äquivalent:

- $b(G, v) = k$
- $BB(k)$ mit Wurzel v ist Teilgraph von G .

4.5.2 Gossiping

Definition 4.13 (Gossip-Komplexität)

Sei $G = (V, E)$ ein zusammenhängender Graph. Dann stehe $r_1(G)$ ($r_2(G)$) für die minimale Anzahl von Kommunikationsschritten im Telegraph-Modus (Telephon-Modus), um das Gossip-Problem auf G zu lösen. $r_1(G)$ ($r_2(G)$) heißt Gossip-Komplexität von G im Telegraph-Modus (Telephone-Modus).

Wie wir im Verlauf des Abschnitts sehen werden, ist der Telephon-Modus für das Gossip-Problem mächtiger als der Telegraph-Modus. Auch für das Gossip-Problem lassen sich ein paar einfache untere Schranken angeben:

Satz 4.8 (untere Schranken für Gossiping)

Sei $G = (V, E)$ ein zusammenhängender Graph. Dann gilt:

- 1.) $r_2(G) \leq r_1(G) \leq 2 \cdot r_2(G)$
- 2.) $r_2(G) \geq \lceil \log |V| \rceil$
- 3.) $r_2(G) \geq d(G)$

Beweis:

ad 1.)

Ist klar, da jeder Algorithmus im Telegraph-Modus auch einen Algorithmus im Telephon-Modus darstellt und jeder Algorithmus im Telephon-Modus von einem Algorithmus im Telegraph-Modus mit doppelter Laufzeit simuliert werden kann.

ad 2.)

Folgt unmittelbar aus Satz 4.7, da ein Gossip nichts anderes ist als ein gleichzeitiger Broadcast von allen Knoten des Graphen aus.

ad 3.)

Da der Durchmesser eines Graphen durch $d(G) = \max\{\text{dist}(u, v); u, v \in V\}$ definiert ist, folgt auch diese Aussage unmittelbar aus Satz 4.7. ■

Im folgenden werden wir nach oberen Schranken für die Gossip-Komplexität im Cliques-Netzwerk suchen und die untere Schranke für $r_1(G)$ verbessern. Die oberen Schranken im Cliques-Netzwerk zeigen uns die Güte unserer unteren Schranken, da man das Gossip-Problem in keinem Graphen schneller als in einer Clique lösen kann. Zu diesem Zweck definieren wir:

Definition 4.14

Sei $Clique(n)$, $n \in \mathbb{N}$, der vollständige Graph mit n Knoten. Die Gossip-Komplexitäten von $Clique(n)$ $r_1(n)$ und $r_2(n)$ sind dann definiert durch $r_1(n) = r_1(Clique(n))$ und $r_2(n) = r_2(Clique(n))$.

Die untere Schranke für $r_2(G)$ ist für Graphen mit gerader Knotenanzahl scharf, wie der folgenden Satz zeigt:

Satz 4.9 (Satz von Knödel [29])

Ist $n \in \mathbb{N}$ gerade, so gilt $r_2(n) = \lceil \log n \rceil$.

Beweis:

$r_2(n) \geq \lceil \log n \rceil$ gilt nach Satz 4.8. Wir zeigen $r_2(n) \leq \lceil \log n \rceil$, indem wir einen Gossip-Algorithmus mit Laufzeit $\lceil \log n \rceil$ angeben. Sei dazu $n = 2m$ und die Knotenmenge der Clique $V = \{Q[i], R[i]; 0 \leq i < m\}$. $u \leftrightarrow v$ steht im folgenden dafür, daß die Knoten u und v ihre aktuellen Informationen austauschen; dies funktioniert mittels eines Kommunikationsschritts im Telephone-Modus.

```

procedure GossipTelephone
var i : 0...m - 1;  t : integer;
begin
  for t := 0 to  $\lceil \log m \rceil$  do
    for all i  $\in \{0, \dots, m - 1\}$  parallel do
       $Q[i] \leftrightarrow R[(i + 2^t - 1) \bmod m]$ 
    od
  od
end

```

Die Laufzeit des Algorithmus beträgt offensichtlich $\lceil \log m \rceil + 1 = \lceil \log n \rceil$. Die Korrektheit zeigen wir durch Induktion nach t . Stehe dazu $Q(i, t)$ bzw. $R(i, t)$ für die Informationen, die die Knoten $Q[i]$ bzw. $R[i]$ nach Runde t des Algorithmus kennen. Ferner sei $\alpha(i) = I(R[i]) \cup I(Q[i])$ und $[i, l] = \{\alpha((i + j) \bmod m); 0 \leq j < l\}$. Es gilt:

$$Q(i, t) = R((i + 2^t - 1) \bmod m, t) = [i, 2^t],$$

was den Satz beweist, da $[i, 2^{\lceil \log m \rceil}] = [i, m] = I(Clique(n))$ ist.

Ind. Anf.: $t = 0$

Nach Runde 0 haben $Q[i]$ und $R[i]$ ihre Informationen ausgetauscht und es gilt $Q(i, 0) = R(i, 0) = \alpha(i) = [i, 1]$.

Ind. Vor.:

Nach Runde t gelte $Q(i, t) = R((i + 2^t - 1) \bmod m) = [i, 2^t]$.

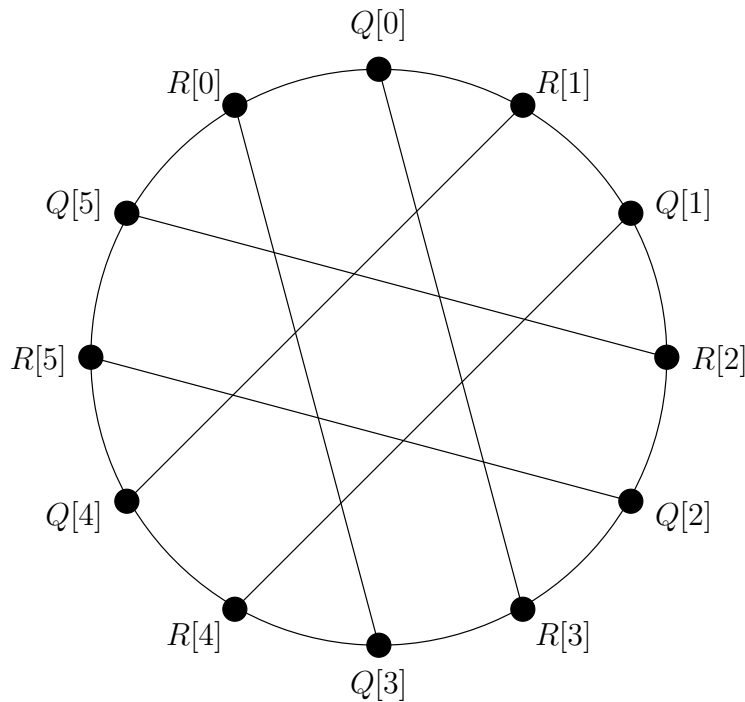


Abbildung 30: Gossip-Graph $Gos(12)$

Ind. Schritt: $t \mapsto t + 1$

In Runde $t + 1$ tauschen die Knoten $Q[i]$ und $R[(i + 2^{t+1} - 1) \bmod m]$ ihre Informationen aus. Also:

$$\begin{aligned}
 Q(i, t + 1) &= R((i + 2^{t+1} - 1) \bmod m, t + 1) \\
 &= Q(i, t) \cup R((i + 2^{t+1} - 1) \bmod m, t) \\
 &= [i, 2^t] \cup [(i + 2^{t+1} - 1 - (2^t - 1)) \bmod m, 2^t] \\
 &= [i, 2^t] \cup [(i + 2^t) \bmod m, 2^t] \\
 &= [i, 2^{t+1}]
 \end{aligned}$$

■

Der Algorithmus GossipTelephon benutzt nicht alle Kanten der Clique; von jedem Knoten werden höchstens $\lceil \log n \rceil$ Kanten verwendet. Der durch den Algorithmus definierte Graph wird Gossip-Graph $Gos(n)$ genannt und es gilt offensichtlich $r_2(Gos(n)) = \lceil \log n \rceil$. Abbildung 30 zeigt $Gos(12)$.

Kommen wir nun zur Gossip-Komplexität von Graphen im Telegraph-Modus. Wir zeigen zunächst eine obere Schranke für $r_1(n)$, der Gossip-Komplexität der Clique im Telegraph-Modus. Dazu benötigen wir folgende Definition:

Definition 4.15 (Fibonacci-Zahlen)

Die Fibonacci-Zahlen $F(i)$, $i \in \mathbb{N}$, sind rekursiv wie folgt definiert:

- 1.) $F(1) = F(2) = 1$
- 2.) $F(i + 1) = F(i - 1) + F(i)$, für $i \geq 2$

Lemma 4.7

Sei $b = (1 + \sqrt{5})/2$. Dann gilt $b^{i-2} < F(i) < b^{i-1}$ für $i \geq 3$.

Beweis:

Es ist $b \approx 1.618$ und b erfüllt die Gleichung $1 + b = b^2$. Induktion nach i liefert:

Ind. Anf.: $i = 3, i = 4$

Es gilt $b^1 < F(3) = 2 < b^2$ und $b^2 < F(4) = 3 < b^3$.

Ind. Vor.:

Es gelte $b^{j-2} < F(j) < b^{j-1}$ für $j \in \{i - 1, i\}$.

Ind. Schritt: $i - 1, i \mapsto i + 1$

Es gilt:

$$F(i + 1) = F(i - 1) + F(i) < b^{i-2} + b^{i-1} = b^{i-2}(1 + b) = b^{i-2}b^2 = b^i$$

Analog zeigt man $F(i + 1) > b^{i-1}$.

■

Mit Hilfe der Fibonacci-Zahlen läßt sich jetzt ein guter Gossip-Algorithmus für die Clique im Telegraph-Modus entwickeln.

Satz 4.10 (obere Schranke für Gossiping im Telegraph-Modus)

Sei $n = 2m$, $m \in \mathbb{N}$, und k so gewählt, daß $F(k) \geq m$ gilt. Dann folgt $r_1(n) \leq k + 1$.

Beweis:

Wir geben einen Gossip-Algorithmus im Telegraph-Modus mit Laufzeit $k + 1$ an. $R[i]$, $Q[i]$, $R(i, t)$, $Q(i, t)$ und $[i, l]$ seien wie im Beweis zu Satz 4.9 definiert. $u \rightarrow v$ bedeutet, daß Knoten u seine Informationen an Knoten v schickt.

```

procedure GossipTelegraph
var i : 0 ... m - 1;  t : integer;
begin
  for all i ∈ {0, ..., m - 1} parallel do
    Q[i] → R[i];  R[i] → Q[i]
  od
  t := 1
  while F(2t - 1) < m do
    for all i ∈ {0, ..., m - 1} parallel do
      R[(i + F(2t - 1)) mod m] → Q[i]
    if F(2t) < m then

```

```

    Q[(i + F(2t)) mod m] → R[i]
  fi
od
t := t + 1
od
end

```

Um die Korrektheit zu beweisen, zeigen wir induktiv, daß nach Runde t

$$Q(i, t) = [i, F(2t + 1)] \quad \text{und} \quad R(i, t) = [i, F(2t + 2)]$$

gilt.

Ind. Anf.: $t = 0$

Vor der ersten Runde gilt $Q(i, 0) = R(i, 0) = \alpha(i) = [i, F(1)] = [i, F(2)]$.

Ind. Schritt: $t - 1 \mapsto t$

Nach Runde t gilt

$$\begin{aligned}
 Q(i, t) &= Q(i, t - 1) \cup R((i + F(2t - 1)) \bmod m, t - 1) \\
 &= [i, F(2t - 1)] \cup [(i + F(2t - 1)) \bmod m, F(2t)] \\
 &= [i, F(2t - 1) + F(2t)] = [i, F(2t + 1)]
 \end{aligned}$$

und

$$\begin{aligned}
 R(i, t) &= R(i, t - 1) \cup Q((i + F(2t)) \bmod m, t) \\
 &= [i, F(2t)] \cup [(i + F(2t)) \bmod m, F(2t + 1)] \\
 &= [i, F(2t) + F(2t + 1)] = [i, F(2t + 2)]
 \end{aligned}$$

Man beachte, daß $R[i]$ in Runde t bereits die Informationen des Konten $Q[i + F(2t)]$ aus Runde t zugeschickt bekommt.

Nach $\lceil (k - 1)/2 \rceil$ Runden wird die while-Schleife beendet, da dann $t = \lceil (k - 1)/2 \rceil + 1$ und folglich $F(2t - 1) = F(2\lceil (k - 1)/2 \rceil + 1) \geq F(k) \geq m$ gilt. Alle Knoten sind dann auch im Besitz der kumulierten Information $I(\text{Clique}(n))$. (Für gerade k wird den Knoten $R[i]$ in der letzten Runde keine Information mehr gesendet, da diese bereits nach der vorletzten Runde die kumulierte Information besaßen.)

Daraus folgt auch, daß in der while-Schleife insgesamt $k - 1$ Kommunikationsschritte durchgeführt werden. Mit den beiden Schritten vor der Schleife ergibt sich also eine Laufzeit von $k + 1$. ■

Die gefundene obere Schranke für Gossiping im Telegraph-Modus auf der Clique ist relativ weit von der trivialen unteren Schranke $r_1(n) \geq \lceil \log n \rceil$ entfernt. Im folgenden wollen wir eine bessere untere Schranke entwickeln, allerdings nicht direkt für Gossiping sondern eine etwas andere Problemstellung:

Definition 4.16 (Zählproblem, NCP)

Das Zählproblem (Network Counting Problem, NCP) besteht in folgendem: In der Clique $Clique(n)$ speichert jeder Knoten zu Beginn die Zahl 1. Es werden solange Kommunikationsschritte im Telegraph-Modus ausgeführt, bis jeder Prozessor eine Zahl $\geq n$ speichert. Dabei hat die Kommunikation $u \rightarrow v$ die Wirkung, daß Knoten v anschließend die Summe der Zahlen von u und v speichert.

Das Zählproblem ist eine einfachere Variante des Gossip-Problems. Jeder Knoten muß hier zwar auch insgesamt n Informationen erhalten, diese n Informationen müssen aber nicht mehr paarweise verschieden sein. Wie man leicht sieht, löst jeder Gossip-Algorithmus auch das Zählproblem, also stellt eine untere Schranke für das Zählproblem auch eine untere Schranke für das Gossip-Problem dar. Bevor wir zu einer unteren Schranke für das NCP kommen, wollen wir noch eine obere Schranke angeben:

Satz 4.11 (obere Schranke für NCP)

Sei $n \in \mathbb{N}$ gerade und $k \in \mathbb{N}$ so gewählt, daß $F(k) \geq n$ gilt. Dann kann das Zählproblem in $k - 1$ Runden gelöst werden.

Beweis:

Es sei $n = 2m$ und $V = \{Q[i], R[i]; 0 \leq i < m\}$. Folgender Algorithmus löst das NCP:

```

procedure NCPTelegraph
var i : 0 ... m - 1;
begin
  while „Es gibt Knoten mit Zahl < n“ do
    for all i in {0, ..., m - 1} parallel do
      R[i] → Q[i]
    if „Es gibt Knoten mit Zahl < n“ then
      Q[i] → R[i]
    fi
  od
od
end

```

Betrachtet man die Zahlen, die ein Knotenpaar $R[i], Q[i]$ nach jedem Kommunikationsschritt speichern, so ergibt sich die Folge:

$$(1, 1) \rightarrow (1, 2) \rightarrow (3, 2) \rightarrow (3, 5) \rightarrow (8, 5) \rightarrow (8, 13) \rightarrow \dots$$

Das heißt, nach Kommunikationsschritt t werden die Zahlen $F(t+2)$ und $F(t+1)$ gespeichert. Nach $k - 1$ Schritten besitzt also jeder Knoten eine Zahl $\geq F(k) \geq n$.

■

Für den Beweis der unteren Schranke für das NCP benötigen wir einige Aussagen der linearen Algebra.

Definition 4.17

Seien im folgenden $n \in \mathbb{N}$, $x, y \in \mathbb{R}^n$ und $A = (a_{ij}) \in \mathbb{R}^{n \times n}$.

- a) Das Skalarprodukt zweier Vektoren x und y ist definiert durch $(x, y) = \sum_{i=1}^n x_i \cdot y_i$.
- b) Als Norm eines Vektors x verwenden wir die euklidische Norm $\|x\| = \sqrt{(x, x)}$.
- c) Die Matrix $A^T = (a_{ji}) \in \mathbb{R}^{n \times n}$ heißt transponierte Matrix von A .
- d) Eine Matrix A heißt symmetrisch, wenn $A = A^T$ gilt.
- e) Das Matrix-Vektorprodukt $y = Ax$ ist definiert durch $y_i = \sum_{j=1}^n a_{ij} \cdot x_j = (A_{i\bullet}, x)$.
- f) Die Norm einer Matrix A ist durch $\|A\| = \sup_{x \neq 0} \frac{\|Ax\|}{\|x\|}$ definiert.
- g) λ heißt Eigenwert der Matrix A , wenn $Au = \lambda \cdot u$ für ein $u \in \mathbb{R}^n$, $u \neq 0$, gilt. Alle Vektoren u mit $Au = \lambda \cdot u$ heißen dann Eigenvektoren zum Eigenwert λ von A .
- h) Das charakteristische Polynom der Matrix A ist definiert durch $f_A(\lambda) = \det(A - \lambda E_n)$.

Ohne Beweis geben wir die folgenden drei Sätze an:

Satz 4.12

Seien $A, B \in \mathbb{R}^{n \times n}$ und $x \in \mathbb{R}^n$. Dann gilt:

- 1.) $\|A \cdot B\| \leq \|A\| \cdot \|B\|$
- 2.) $\|Ax\| \leq \|A\| \cdot \|x\|$

Satz 4.13

Sei $A \in \mathbb{R}^{n \times n}$. λ ist genau dann ein Eigenwert von A , wenn λ eine Nullstelle des charakteristischen Polynoms f_A ist.

Satz 4.14

Sei $A \in \mathbb{R}^{n \times n}$ symmetrisch. Dann besitzt A nur reelle Eigenwerte und die Eigenvektoren zu verschiedenen Eigenwerten bilden eine Orthogonal-Basis des \mathbb{R}^n .

Satz 4.15

Sei $A \in \mathbb{R}^{n \times n}$ symmetrisch und $\lambda_1 \leq \lambda_2 \leq \dots \leq \lambda_n$ die Eigenwerte von A . Dann gilt:

$$\lambda_1 = \min_{x \neq 0} \frac{(Ax, x)}{(x, x)} \quad \text{und} \quad \lambda_n = \max_{x \neq 0} \frac{(Ax, x)}{(x, x)}$$

Beweis:

Sei u_1, \dots, u_n eine Orthogonal-Basis aus Eigenvektoren von A zu den Eigenwerten $\lambda_1, \dots, \lambda_n$. Jedes $x \in \mathbb{R}^n$ läßt sich als Linearkombination der u_i schreiben:

$$\begin{aligned} x &= \sum_{i=1}^n \alpha_i u_i, \quad \alpha_i \in \mathbb{R} \\ \implies Ax &= \sum_{i=1}^n \alpha_i A u_i = \sum_{i=1}^n \alpha_i \lambda_i u_i \\ \implies (Ax, x) &= \left(\sum_{i=1}^n \alpha_i \lambda_i u_i, \sum_{j=1}^n \alpha_j u_j \right) = \sum_{i=1}^n \sum_{j=1}^n \alpha_i \lambda_i \alpha_j (u_i, u_j) \end{aligned}$$

Da die Eigenvektoren paarweise senkrecht aufeinander stehen, gilt $(u_i, u_j) = 0$ für $i \neq j$:

$$\begin{aligned} (Ax, x) &= \sum_{i=1}^n \lambda_i \alpha_i^2 (u_i, u_i) \leq \lambda_n \sum_{i=1}^n (\alpha_i u_i, \alpha_i u_i) = \lambda_n (x, x) \\ \implies \frac{(Ax, x)}{(x, x)} &\leq \lambda_n \quad \forall x \neq 0 \end{aligned}$$

Für $x = u_n$ gilt ferner $\frac{(Ax, x)}{(x, x)} = \lambda_n$. Die Aussage für λ_1 zeigt man entsprechend, indem man (Ax, x) mit λ_1 nach unten abschätzt. ■

Korollar 4.2

Sei $A \in \mathbb{R}^{n \times n}$ und λ_n der größte Eigenwert der Matrix $A^T A$. Dann gilt:

$$\|A\| = \sqrt{\lambda_n}$$

Beweis:

$A^T A$ ist symmetrisch und aus dem Beweis zu Satz 4.15 folgt für beliebige $x \in \mathbb{R}^n$:

$$\begin{aligned} \|Ax\|^2 &= (Ax, Ax) = (A^T Ax, x) \leq \lambda_n (x, x) = \lambda_n \|x\|^2 \\ \implies \|A\| &= \sqrt{\lambda_n}, \end{aligned}$$

da für $x = u_n$ die Gleichung $\|Ax\|^2 = \lambda_n \|x\|^2$ erfüllt ist. ■

Die Berechnung der unteren Schranke für das NCP wird mittels linearer Algebra geführt. Zu diesem Zweck betrachten wir einen (optimalen) Algorithmus für das NCP auf $Clique(n)$. Der Algorithmus benötige insgesamt r Kommunikationsschritte und $a^t \in \mathbb{Z}^n$ beschreibe die Belegung der Prozessoren nach Runde t . Es ist $a^0 = (1 \dots 1)$ und für alle $1 \leq i \leq n$ gilt $a_i^r \geq n$, also $\|a^0\| = \sqrt{n}$ und $\|a^r\| \geq n^{3/2}$.

Für Runde t definieren wir eine Matrix A^t , so daß $a^t = A^t a^{t-1}$ gilt, d.h. die Matrix A^t beschreibt die in Runde t stattfindenden Kommunikationen. Wenn Prozessor i seine Zahl an Prozessor j schickt, können wir dies wie folgt in A^t abbilden:

$$\begin{aligned}
 A^t &= \begin{pmatrix} 0 & & 0 & & \\ \vdots & & \vdots & & \\ 0 \cdots 0 & a_{ii} = 1 & 0 \cdots 0 & a_{ij} = 0 & 0 \cdots 0 \\ & 0 & & 0 & \\ \vdots & & \vdots & & \\ 0 \cdots 0 & a_{ji} = 1 & 0 \cdots 0 & a_{jj} = 1 & 0 \cdots 0 \\ & 0 & & 0 & \\ \vdots & & \vdots & & \\ 0 & & 0 & & \end{pmatrix} \\
 \Rightarrow A^t \begin{pmatrix} \vdots \\ a_i^{t-1} \\ \vdots \\ a_j^{t-1} \\ \vdots \end{pmatrix} &= \begin{pmatrix} \vdots \\ a_i^{t-1} \\ \vdots \\ a_j^{t-1} + a_i^{t-1} \\ \vdots \end{pmatrix} = \begin{pmatrix} \vdots \\ a_i^t \\ \vdots \\ a_j^t \\ \vdots \end{pmatrix}
 \end{aligned}$$

Man beachte, daß die Prozessoren i und j an keiner anderen Kommunikation beteiligt sein können, da wir den Telegraph-Modus verwenden. Für die Berechnung der unteren Schranke müssen wir die Norm der A^t -Matrizen bestimmen. Die Norm einer Matrix ändert sich nicht, wenn man Zeilen und Spalten permutiert, wir können also die Matrix A^t so umordnen, daß kommunizierende Prozessorpaaire aufeinanderfolgende Indizes besitzen. O.B.d.A nehmen wir ferner an, daß alle Prozessoren in jeder Runde kommunizieren (dies macht die Zahlen der beteiligten Prozessoren nur größer), und erhalten eine modifizierte Matrix

$$\tilde{A}^t = \begin{pmatrix} B & & 0 \\ & B & \\ & & \ddots \\ 0 & & & B \end{pmatrix} \quad \text{mit} \quad B = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$$

Ohne Beweis gilt $\|A^t\| = \|\tilde{A}^t\| = \|B\|$ und mittels Korollar 4.2 folgt:

$$\begin{aligned}
 B^T B &= \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix} \\
 \Rightarrow f_{B^T B}(\lambda) &= \det \begin{pmatrix} 2-\lambda & 1 \\ 1 & 1-\lambda \end{pmatrix} = (2-\lambda)(1-\lambda) - 1 = \lambda^2 - 3\lambda + 1 \\
 \Rightarrow f_{B^T B}(\lambda) \stackrel{!}{=} 0 &\Rightarrow \lambda_{1,2} = \frac{3}{2} \pm \frac{\sqrt{5}}{2} \\
 \Rightarrow \|A^t\| = \|B\| &= \sqrt{\lambda_2} = \sqrt{\frac{3+\sqrt{5}}{2}} = \frac{1+\sqrt{5}}{2} =: b
 \end{aligned}$$

Wir sind nun bereit, untere Schranken für das NCP zu berechnen. Es folgen zwei leicht unterschiedliche Schranken, wobei der Beweis für die zweite, bessere Schranke technisch aufwendiger ist.

Satz 4.16 (untere Schranke I für NCP)

Ein Algorithmus löse das NCP auf $Clique(n)$ in r Runden. Dann gilt:

$$r \geq \lceil \log_b n \rceil$$

Beweis:

Gemäß obigen Überlegungen und mit den dort verwendeten Bezeichnungen gilt:

$$\begin{aligned} a^r &= A^r \cdot \dots \cdot A^1 a^0 \\ \implies n^{3/2} \leq \|a^r\| &= \|A^r \cdot \dots \cdot A^1 a^0\| \leq \prod_{t=1}^r \|A^t\| \cdot \|a^0\| = b^r \sqrt{n} \\ \implies r &\geq \log_b n \end{aligned}$$

■

Satz 4.17 (untere Schranke II für NCP)

Ein Algorithmus löse das NCP auf $Clique(n)$ in r Runden. Dann gilt:

$$r \geq 2 + \left\lceil \log_b \frac{n}{2} \right\rceil$$

Beweis:

In Runde r empfangen höchstens $n/2$ Prozessoren eine Nachricht, d.h. nach Runde $r - 1$ müssen bereits mindestens $n/2$ Prozessoren eine Zahl $\geq n$ speichern. Wir betrachten nun die Situation nach Runde $r - 2$ und setzen der besseren Lesbarkeit halber $\alpha = a^{r-2}$. Wir unterscheiden drei Gruppen von Prozessoren i :

- 1.) $\alpha_i \geq n$
- 2.) $\alpha_i < n$ und j mit $\alpha_j \geq n$ schickt in Runde $r - 1$ seine Zahl an i
- 3.) $\alpha_i < n$ und j mit $\alpha_j < n$ und $\alpha_i + \alpha_j \geq n$ schickt in Runde $r - 1$ seine Zahl an i

Sei c_1 , c_2 bzw. c_3 die Anzahl der Prozessoren, für die 1.), 2.) bzw. 3.) gilt. Offensichtlich gilt $c_1 \geq c_2$. Die drei Gruppen sind disjunkt und bilden zusammen die Prozessoren, die nach Runde $r - 1$ eine Zahl $\geq n$ speichern. Also:

$$\begin{aligned} c_1 + c_2 + c_3 &\geq \frac{n}{2} \quad \text{und} \quad c_1 \geq c_2 \\ \implies 2c_1 + c_3 &\geq \frac{n}{2} \end{aligned}$$

Für ein Prozessorkpaar (i, j) aus 3.) gilt $\alpha_i + \alpha_j \geq n$ und somit $\alpha_i^2 + \alpha_j^2 \geq n^2/2$.

$$\begin{aligned} \|\alpha\|^2 &= \sum_{i=1}^n \alpha_i^2 \geq \sum_{i \text{ aus 1.})} \alpha_i^2 + \sum_{(i,j) \text{ aus 3.})} (\alpha_i^2 + \alpha_j^2) \\ &\geq c_1 n^2 + c_3 \frac{n^2}{2} = \frac{n^2}{2} (2c_1 + c_3) \geq \frac{n^3}{4} \\ \implies \|\alpha\| &\geq \frac{n^{3/2}}{2} \end{aligned}$$

Wie im Beweis zu Satz 4.16 zeigt man, daß $\|\alpha\| \leq b^{r-2} \sqrt{n}$ ist. Zusammen folgt schließlich $r \geq 2 + \log_b \frac{n}{2}$. ■

Der Unterschied der beiden gefundenen unteren Schranken ist gering, denn wegen $(2 + \log_b \frac{n}{2}) - \log_b n = 2 - \log_b 2 \approx 0.5596$ unterscheiden sie sich höchstens um 1. Wie die folgende Tabelle zeigt, liegen die in diesem Abschnitt gefundenen unteren und oberen Schranken für das Gossip- bzw. Zählproblem allerdings so dicht beieinander, daß sich der zusätzliche Aufwand aus Satz 4.17 lohnt.

n	2	4	6	8	10	12	14	16	18	20	22
Obere Schranke Gossip (4.10)	2	4	5	6	6	7	7	7	8	8	8
Obere Schranke NCP (4.11)	2	4	5	5	6	6	7	7	7	7	8
Untere Schranke NCP (4.17)	2	4	5	5	6	6	7	7	7	7	7
Untere Schranke NCP (4.16)	2	3	4	5	5	6	6	6	7	7	7

Es gilt nämlich:

Satz 4.18

Bezeichne $o(n)$ die obere Schranke für das Gossip-Problem im Telegraph-Modus auf $Clique(n)$ aus Satz 4.10 und $u(n)$ die untere Schranke für das Zählproblem (und damit auch das Gossip-Problem) aus Satz 4.17. Es gilt:

- 1.) $o(n) = u(n)$ für unendlich viele $n \in \mathbb{N}$
- 2.) $u(n) \leq o(n) \leq u(n) + 1$ für alle $n \in \mathbb{N}$, n gerade

Beweis:

ad 1.)

Sei $n = 2 \cdot F(k)$ für ein $k \geq 3$. Dann ist $o(n) = k + 1$ und $u(n) = 2 + \lceil \log_b F(k) \rceil$. Wegen $b^{k-2} < F(k) < b^{k-1}$ folgt $u(n) = 2 + (k - 1) = k + 1$.

ad 2.)

Sei $n = 2m$ beliebig und i so gewählt, daß $b^{i-1} < m \leq b^i$ gilt. Dann ist $u(n) = 2 + \lceil \log_b m \rceil = 2 + i$. Sei k die kleinste Zahl mit $F(k) \geq m$. Weil in jedem Intervall $]b^{j-1}, b^j]$, $j \in \mathbb{N}$, eine Fibonacci-Zahl liegt, muß $F(k)$ entweder im Intervall $]b^{i-1}, b^i]$ oder $]b^i, b^{i+1}]$ liegen. Wegen $b^{k-2} < F(k) < b^{k-1}$ folgt $k - 1 = i$ oder $k - 1 = i + 1$ und letztlich $o(n) = k + 1 = i + 2 = u(n)$ oder $o(n) = k + 1 = i + 3 = u(n) + 1$. ■

Kapitel 5

Bisektionsweite und Partitionierung

Dieses Kapitel beschäftigt sich intensiver mit der Bisektionsweite von Graphen. Die Bisektionsweite ist eine wichtige Eigenschaft eines parallelen Rechnernetzes, da eine kleine Bisektionsweite auf mögliche Engpässe bei der Informationsverteilung im Netz hinweist.

In Abschnitt 5.1 werden Verfahren zur Berechnung von unteren Schranken für die Bisektionsweite vorgestellt. In Abschnitt 5.2 werden obere Schranken für die Bisektionsweite von regulären Graphen bewiesen und auf eine Anwendung der Ergebnisse beim Bau von Transputer-Systemen hingewiesen.

5.1 Untere Schranken für die Bisektionsweite

In diesem Abschnitt werden zwei Verfahren vorgestellt, mit denen sich untere Schranken für die Bisektionsweite von Graphen berechnen lassen. Solche unteren Schranken sind insbesondere in den Fällen nützlich, in denen sich auch eine Partition mit entsprechender Größe finden läßt, da so die Bisektionsweite eines Graphen bestimmt werden kann. In Satz 5.2 wird beispielsweise bewiesen, daß die Bisektionsweite des Hypercube $Q(k)$ eine Größe von 2^{k-1} besitzt. Allerdings liefert keines der beiden hier vorgestellten Verfahren für alle Graphen scharfe untere Schranken.

5.1.1 Verfahren von Leighton

Das Verfahren von Leighton verwendet ein vollständiges Wegesystem und die sich daraus ergebende maximale Kantenbelastung, um die Bisektionsweite eines Graphen nach unten abzuschätzen.

Definition 5.1 (Wegesystem)

Sei $G = (V, E)$ ein ungerichteter Graph. Ein Wegesystem \mathcal{W} definiert zu jedem Knotenpaar $(u, v) \in V^2$ einen Weg von u nach v . Für $e \in E$ stehe $c_{\mathcal{W}}(e)$ für die Anzahl der Wege von \mathcal{W} , in denen Kante e enthalten ist, und $c_{\mathcal{W}}^* = \max\{c_{\mathcal{W}}(e); e \in E\}$ für die maximale Kantenbelastung durch das Wegesystem.

Man kann ein Wegesystem auch als eine Einbettung der Clique mit $|V|$ Knoten in G ansehen. $c_{\mathcal{W}}^*$ entspricht dabei der Kantenauslastung der Einbettung.

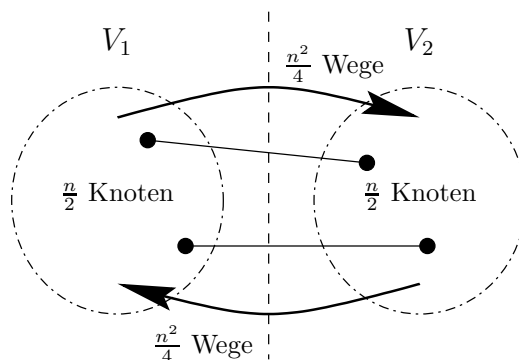


Abbildung 31: $\frac{n^2}{2}$ Wege laufen über den Schnitt der Partition V_1, V_2

Satz 5.1 (Satz von Leighton)

Sei $G = (V, E)$ ein ungerichteter Graph mit gerader Knotenanzahl $n = |V|$ und \mathcal{W} ein Wegesystem für G mit maximaler Kantenbelastung $c_{\mathcal{W}}^*$. Dann gilt für die Bisektionsweite von G :

$$\sigma(G) \geq \frac{n^2}{2c_{\mathcal{W}}^*}$$

Beweis:

Sei $V_1, V_2 \subset V$ mit $V_1 \dot{\cup} V_2 = V$ und $|V_1| = |V_2| = n/2$ eine balancierte Partition von G mit minimalem Schnitt $\text{ext}(V_1, V_2) = \sigma(G)$. Insgesamt gibt es $2 \cdot |V_1| \cdot |V_2| = n^2/2$ viele Wege, die zwischen Knoten aus V_1 und V_2 verlaufen ($|V_1| \cdot |V_2|$ viele von Knoten aus V_1 zu Knoten aus V_2 und genauso viele in die andere Richtung), und jeder dieser Wege muß dabei (mindestens) eine der $\sigma(G)$ vielen Kanten des Schnitts benutzen (Abbildung 31). Wir wissen aber ferner, daß jede Kante vom Wegesystem höchstens $c_{\mathcal{W}}^*$ -mal verwendet wird, es laufen also höchstens $c_{\mathcal{W}}^* \cdot \sigma(G)$ Wege über den Schnitt. Daraus folgt:

$$\frac{n^2}{2} \leq c_{\mathcal{W}}^* \cdot \sigma(G) \iff \frac{n^2}{2c_{\mathcal{W}}^*} \leq \sigma(G)$$

■

Wir wollen den Satz von Leighton verwenden, um die Bisektionsweite des Hypercube zu bestimmen.

Satz 5.2 (Bisektionsweite des Hypercube)

Die Bisektionsweite des Hypercube $Q(k)$ beträgt 2^{k-1} .

Beweis:

Wir bestimmen zuerst mittels des Satzes von Leighton und des in Abschnitt 3.1 angegebenen Routing-Algorithmus eine untere Schranke für die Bisektionsweite des $Q(k)$. Jeder Routing-Algorithmus definiert ein Wegesystem \mathcal{W} und zur Bestimmung der maximalen Belastung $c_{\mathcal{W}}^*$ der Kanten durch das Wegesystem betrachten wir eine beliebige Kante $\{u, u(i)\}$,

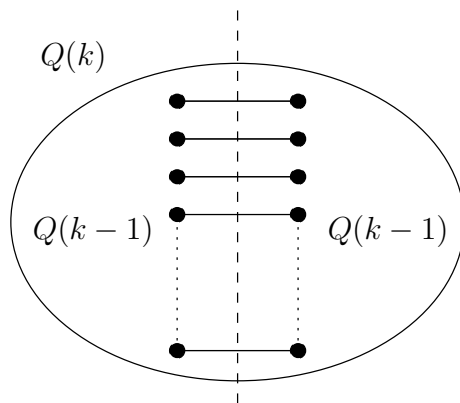


Abbildung 32: Bisektion des $Q(k)$ mit 2^{k-1} Kanten

$u = u_{k-1} \dots u_0 \in \{0, 1\}^k$ und $0 \leq i < k$, des $Q(k)$. Die Kante kann vom Routing-Algorithmus in beiden Richtungen durchlaufen werden. Bestimmen wir zuerst die Anzahl der Wege, die die Kante von u nach $u(i)$ durchlaufen. Alle Knoten der Form $u_{k-1} \dots u_i \{0, 1\}^i$ können mittels des Routing-Algorithmus den Knoten u erreichen, von $u(i)$ aus können alle Knoten der Form $\{0, 1\}^{k-i-1} \bar{u}_i u_{i-1} \dots u_0$ erreicht werden (Man beachte, daß die Kante vom Routing-Algorithmus nur im i -ten Schleifendurchlauf verwendet wird). Das bedeutet, daß von u nach $u(i)$ insgesamt $2^i \cdot 2^{k-i-1} = 2^{k-1}$ Wege laufen. Für die Richtung von $u(i)$ nach u verläuft die Analyse entsprechend, also wird die Kante von $2 \cdot 2^{k-1} = 2^k$ Wegen verwendet. Damit ist auch die maximale Belastung $c_{\mathcal{V}}^* = 2^k$ und für die Bisektionsweite des $Q(k)$ folgt:

$$\sigma(Q(k)) \geq \frac{(2^k)^2}{2 \cdot 2^k} = 2^{k-1}$$

Für den $Q(k)$ läßt sich ferner einfach ein Schnitt mit 2^{k-1} Kanten finden. Wähle beispielsweise $V_1 = \{0u; u \in \{0, 1\}^{k-1}\}$ und $V_2 = \{1u; u \in \{0, 1\}^{k-1}\}$ (Abbildung 32). Insgesamt folgt also $\sigma(Q(k)) = 2^{k-1}$. ■

5.1.2 Spektrale untere Schranke

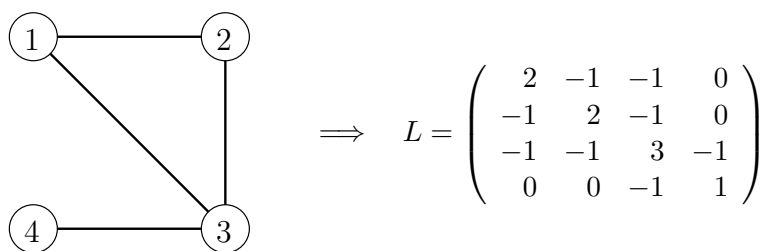
Das in diesem Abschnitt beschriebene Verfahren verwendet einen gänzlich anderen Ansatz als das Verfahren von Leighton. Die algebraischen Eigenschaften der Laplace-Matrix eines Graphen werden hier für die Bestimmung einer unteren Schranke der Bisektionsweite ausgenutzt.

Definition 5.2 (Laplace-Matrix)

Sie $G = (V, E)$ ein beliebiger Graph mit $n = |V|$ Knoten, $V = \{v_1, \dots, v_n\}$. Die Laplace-Matrix $L_G = (l_{ij}) \in \mathbb{Z}^{n \times n}$ ist definiert durch:

$$l_{ij} = \begin{cases} \text{grad}(v_i) & , \text{ falls } i = j \\ -1 & , \text{ falls } (v_i, v_j) \in E \\ 0 & , \text{ sonst} \end{cases}$$

Für den folgenden Graphen ergibt sich so beispielsweise:



Die Zeilensumme einer Laplace-Matrix L_G ist nach Konstruktion immer 0. Es gilt also:

$$L_G \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix} = 0 \cdot \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}$$

Somit ist 0 ein Eigenwert jeder Laplace-Matrix und $(1 \dots 1)$ ein zugehöriger Eigenvektor. Offensichtlich ist die Laplace-Matrix von ungerichteten Graphen symmetrisch. Nach Satz 4.14 sind daher alle Eigenwerte der Matrix reell und die zugehörigen Eigenvektoren bilden eine Orthogonal-Basis. Man kann ferner zeigen, daß alle Eigenwerte größer gleich Null sind, 0 ist also der kleinste Eigenwert einer Laplace-Matrix. Ferner gilt, daß ein ungerichteter Graph genau dann zusammenhängend ist, wenn alle anderen Eigenwerte echt größer als Null sind.

Satz 5.3 (spektrale untere Schranke)

Sei $G = (V, E)$ ein ungerichteter Graph mit gerader Knotenanzahl $n = |V|$ und λ_2 der zweitkleinste Eigenwert der Laplace-Matrix L_G . Dann gilt:

$$\sigma(G) \geq \lambda_2 \cdot \frac{n}{4}$$

Beweis:

Da $(1 \dots 1)$ Eigenvektor zum kleinsten Eigenwert 0 von L_G ist, müssen alle Eigenvektoren x von λ_2 nach Satz 4.14 senkrecht auf $(1 \dots 1)$ stehen, d.h. es muß $(x, (1 \dots 1)) = \sum_{i=1}^n x_i = 0$ gelten. Wie im Beweis zu Satz 4.15 zeigt man dann, daß

$$\lambda_2 = \min_{\substack{x \neq 0 \\ \sum x_i = 0}} \frac{(L_G x, x)}{(x, x)}$$

gilt. Wir betrachten nun eine minimale balancierte Partition V_1, V_2 von G , d.h. es ist $|V_1| = |V_2| = n/2$ und $\text{ext}(V_1, V_2) = \sigma(G)$. Zu dieser Partition definieren wir wie folgt einen Vektor $y \in \mathbb{Z}^n$:

$$y_i = \begin{cases} -1 & , \text{ falls } v_i \in V_1 \\ +1 & , \text{ falls } v_i \in V_2 \end{cases}$$

Offensichtlich ist $\sum_{i=1}^n y_i = 0$ und deshalb gilt:

$$\lambda_2 \leq \frac{(L_G y, y)}{(y, y)}$$

Der Wert von (y, y) ist $\sum_{i=1}^n y_i^2 = n$. Sei im folgenden $N(i)$ die Menge der Nachbarn von Knoten v_i . Für einen beliebigen Vektor $x \in \mathbb{Z}^n$ gilt dann:

$$\begin{aligned}
 (L_G x, x) &= \sum_{i=1}^n \left(\sum_{j=1}^n l_{ij} x_j \right) x_i \\
 &= \sum_{i=1}^n \left(l_{ii} x_i - \sum_{v_j \in N(i)} x_j \right) x_i \\
 &= \sum_{i=1}^n \left(|N(i)| \cdot x_i - \sum_{v_j \in N(i)} x_j \right) x_i \\
 &= \sum_{i=1}^n \left(\sum_{v_j \in N(i)} (x_i - x_j) \right) x_i \\
 &= \sum_{\{v_i, v_j\} \in E} ((x_i - x_j)x_i + (x_j - x_i)x_j) \\
 &= \sum_{\{v_i, v_j\} \in E} (x_i - x_j)^2
 \end{aligned}$$

Für alle Kanten, die nicht zum Schnitt der optimalen Partition V_1, V_2 gehören, ist $y_i = y_j$; für eine Kante des Schnitts gilt $|y_i - y_j| = 2$ und deshalb

$$(L_G y, y) = \sum_{\{v_i, v_j\} \in E} (y_i - y_j)^2 = \sum_{\substack{\{v_i, v_j\} \in \\ \text{Ext}(V_1, V_2)}} (y_i - y_j)^2 = \sum_{\substack{\{v_i, v_j\} \in \\ \text{Ext}(V_1, V_2)}} 2^2 = 4 \cdot \sigma(G)$$

Zusammen mit $(y, y) = n$ erhalten wir schließlich:

$$\lambda_2 \leq \frac{(L_G y, y)}{(y, y)} = \frac{4 \cdot \sigma(G)}{n} \implies \sigma(G) \geq \lambda_2 \cdot \frac{n}{4}$$

■

5.2 Obere Schranken für die Bisektionsweite

Nachdem wir im letzten Abschnitt Verfahren zur Berechnung von unteren Schranken der Bisektionsweite vorgestellt haben, dreht sich dieser Abschnitt um obere Schranken der Bisektionsweite.

Für einen einzelnen Graphen läßt sich eine obere Schranke einfach durch Angabe einer balancierten Partition berechnen. Daher geht es in diesem Abschnitt um die Bestimmung von oberen Schranken für ganze Klassen von Graphen. Beginnen wollen wir zunächst mit dem Spezialfall des Shuffle-Exchange-Netzwerkes, bevor wir ganz allgemein reguläre Graphen betrachten.

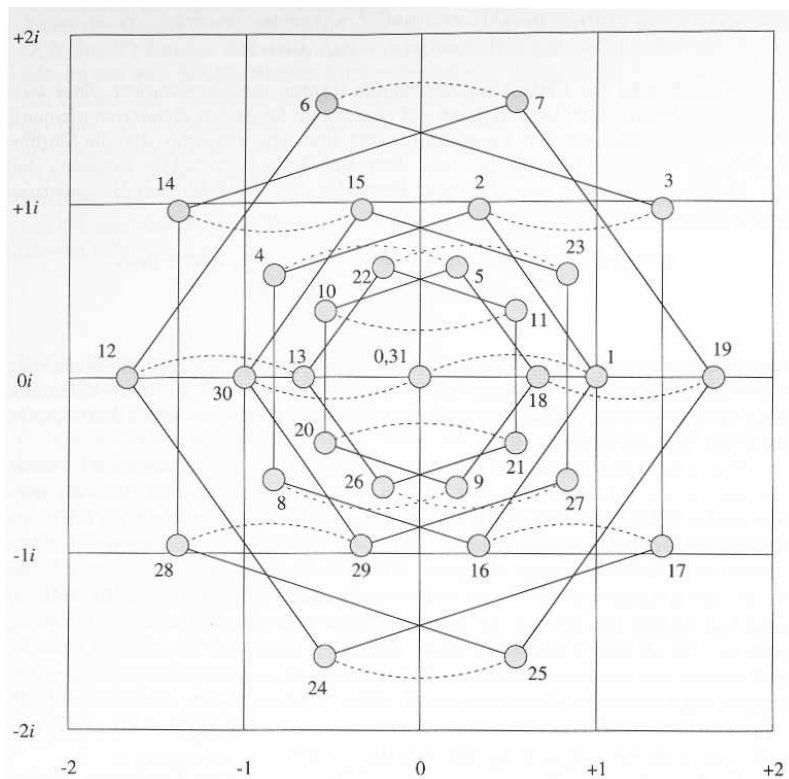


Abbildung 33: Abbildung des $SE(5)$ auf die Gaußsche Zahlenebene. Unterbrochene Linien bezeichnen Exchange-Kanten, durchgezogene Linien stellen Shuffle-Kanten dar. Quelle: [30, S. 367]

Satz 5.4 (Bisektionsweite des $SE(k)$)

Die Bisektionsweite des k -dimensionalen Shuffle-Exchange-Netzwerkes $SE(k)$ kann wie folgt nach oben beschränkt werden: $\sigma(SE(k)) \leq 4 \cdot \frac{2^k}{k}$.

Beweis:

Seien dazu $n = 2^k$, $\omega_k = e^{\frac{2\pi i}{k}}$ die k -te Einheitswurzel (d. h. $\omega_k^k = 1$) und die Abbildung ϕ der Knoten des $SE(k)$ in die Gaußsche Zahlenebene folgendermaßen definiert:

$$\begin{aligned} \phi : \{0, 1\}^k &\rightarrow \mathbb{C} \\ \phi(u_{k-1} \dots u_0) &= \sum_{j=0}^{k-1} \omega_k^j u_j \\ &= \omega_k^{k-1} u_{k-1} + \omega_k^{k-2} u_{k-2} + \dots + \omega_k u_1 + u_0 \end{aligned}$$

Diese Einbettung ist in Abbildung 33 am Beispiel des Shuffle-Exchange-Netzwerkes mit 32 Knoten verdeutlicht. Dort wird bereits ersichtlich, dass ϕ nicht injektiv ist.

1. Exchange-Kanten:

Die Abbildung 33 zeigt auch, dass Exchange-Kanten als horizontale Linien der Länge 1 abgebildet werden, denn es gilt: $\phi(u_{k-1} \dots u_1 1) = \phi(u_{k-1} \dots u_1 0) + 1$.

2. Shuffle-Kanten:

Shuffle-Kanten treten in symmetrischen Kreisen um den Ursprung auf, weil folgender Zusammenhang besteht:

$$\omega_k \phi(u) = \omega_k \left(\sum_{j=0}^{k-1} \omega_k^j u_j \right) = \omega_k^{k-1} u_{k-2} + \dots + \omega_k^2 u_1 + \omega_k u_0 + u_{k-1} = \phi(u_{k-2} \dots u_0 u_1) = \phi(s(u)).$$

Somit entspricht die Verwendung einer Shuffle-Kante einer Drehung um $\frac{2\pi}{k}$ in der Gaußschen Zahlenebene.

Der größte und der kleinste Knoten bleiben bei $k = 5$ außerhalb der Shuffle-Kreise (sie werden auf den Ursprung abgebildet):

3. Degenerierte Shuffle-Kreise (d. h. mit Länge $< k$) werden auf den Nullpunkt abgebildet:

Für Shuffle-Kreise mit weniger als k Knoten gilt: $s^r(u) = u$ für ein $1 \leq r < k$ (dann $r \mid k$). Hieraus folgt $\phi(u) = \phi(s^r(u))$ und dies entspricht nach Induktion $\omega_k^r \phi(u)$ wegen Punkt 2.

Weiterhin gilt wegen $1 \leq r < k$: $\omega_k^r \neq 1 \Rightarrow \phi(u) = 0$.

Ohne Beweis stellen wir fest, dass nicht-degenerierte Shuffle-Kreise nicht auf den Nullpunkt abgebildet werden. Im folgenden zeigen wir, dass höchstens $O(n \log n)$ Knoten auf den Ursprung abgebildet werden.

4. $|\{u \mid \phi(u) = 0\}| = O\left(\frac{n}{\log n}\right)$:

$$\phi(u) = 0 \Rightarrow \phi(x(u)) = 1, x(u) = u_{k-1} \dots u_1 \bar{u}_0$$

Jeder Knoten u des Ursprungs kann eindeutig mit $(-1, 0)$ oder $(1, 0)$ assoziiert werden, da eines von beiden $x(u)$ darstellt. Hieraus folgt, dass alle Knoten des Shuffle-Kreises, auf dem $x(u)$ liegt, nicht auf 0 abgebildet werden. Da dieser Kreis nicht degeneriert ist, besteht er aus $k = \log n$ Knoten. Folglich existieren höchstens $O\left(\frac{n}{\log n}\right)$ vollständige Shuffle-Kreise, so daß $(-1, 0)$ oder $(1, 0)$ höchstens $\frac{2n}{\log n}$ Knoten zugeordnet werden können. Damit können höchstens genauso viele Knoten auf den Ursprung abgebildet werden.

5. $|\{u \mid \phi(u) \in \mathbb{R}\}| = O\left(\frac{n}{\log n}\right)$:

Für u mit $\phi(u) = 0$ definiert $x(u)$ einen nicht degenerierten Shuffle-Kreis. Jeder solche Shuffle-Kreis hat höchstens zwei reelle Knoten, da jeder Shuffle-Kreis die reelle Achse entweder gar nicht oder in genau zwei Knoten trifft.

6. $|\{u \mid \Im(\phi(u)) < 0\}| = |\{u \mid \Im(\phi(u)) > 0\}|$:

Sei u hierfür beliebig, $\bar{u} = \overline{u_{k-1} \dots u_1 \bar{u}_0}$. Dann muss gelten: $\phi(u) + \phi(\bar{u}) = 0$, also: $\phi(u) + \phi(\bar{u}) = \sum_{j=0}^{k-1} \omega_k^j =: a$ und $a = 0$, denn $a \cdot \omega_k = a$ und $\omega_k \neq 1 \Rightarrow a = 0$. Anschaulich gesprochen teilt die reelle Achse den Graphen somit in zwei gleich große Teile.

Nun haben wir alles Nötige zur Hand, um die ursprüngliche Behauptung zu beweisen:

7. Berechnung einer Bisektion mit Bisektionsweite $O\left(\frac{n}{\log n}\right)$:

- U_1, U_2 seien die beiden Partitionen einer exakten Bisektion der Knoten u mit $\phi(u) \in \mathbb{R}$.
- Seien $V_1 = U_1 \cup \{u; \Im(\phi(u)) > 0\}$ und $V_2 = U_2 \cup \{u; \Im(\phi(u)) < 0\}$.

Zu zeigen: $|\text{ext}(V_1, V_2)| = O\left(\frac{n}{\log n}\right)$

- Die Knoten aus U_1, U_2 erzeugen höchstens $O\left(\frac{n}{\log n}\right)$ externe Kanten.

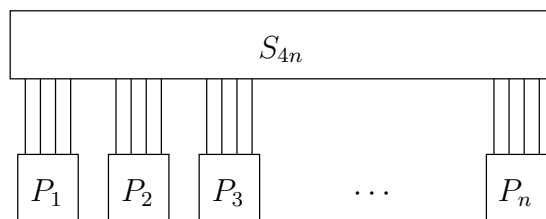


Abbildung 34: Ein Transputer-System mit n Prozessoren und einem Switch

- Aus jedem degenerierten Shuffle-Kreis kreuzen maximal zwei Knoten die reelle Achse. Es gibt weiterhin höchstens $\frac{n}{\log n}$ solche Kreise.

$$\Rightarrow \sigma(SE(k)) = O\left(\frac{n}{\log n}\right), \text{ genauer } \sigma(SE(k)) \leq 4 \cdot \frac{2^k}{k}.$$

■

Die untere Schranke für $\sigma(SE(k))$ liegt bei $\frac{2^k}{k}$ (ohne Beweis). Diese Lücke zwischen oberer und unterer Schranke konnte bisher für allgemeines k nicht geschlossen werden.

Nun wenden wir uns dem allgemeineren Fall der regulären Graphen zu und verwenden im folgenden ausschließlich konstruktive Verfahren. Dies bedeutet, dass diese Verfahren es ermöglichen, zu jedem Graphen aus der betrachteten Klasse einen Schnitt zu erzeugen, der höchstens so viele Kanten enthält, wie die obere Schranke vorgibt. Dazu definieren wir:

Definition 5.3 (maximale Bisektionsweite regulärer Graphen)

Für $d \in \mathbb{N}$ sei $\sigma_d(n)$ die maximale Bisektionsweite aller regulären Graphen vom Grad d , d.h.

$$\sigma_d(n) = \max\{\sigma(G); G \text{ hat } n \text{ Knoten und ist regulär vom Grad } d\}$$

Einleitend wollen wir uns kurz mit einer Anwendung von oberen Schranken für die Bisektionsweite beschäftigen, dem Bau von Transputer-Systemen.

Definition 5.4 (Transputer)

Ein Transputer-System ist ein Parallelrechner mit verteiltem Speicher, in dem spezielle Prozessoren, Transputer genannt, zum Einsatz kommen. Jeder Prozessor ist mit vier Kommunikationskanälen (Links) ausgerüstet, über die er direkt mit anderen Transputern verbunden werden kann. Der Datenaustausch zwischen den Prozessoren findet ausschließlich über diese Links statt.

Die Art und Weise, in der ein Transputer-System verschaltet ist, wird dabei normalerweise nicht im voraus festgelegt, damit man (nach Möglichkeit) jeden beliebigen 4-regulären Graphen als Netzwerk-Topologie verwenden kann. Dabei kommen sogenannte Switches zum Einsatz; ein Switch mit n Kanälen kann seine Kanäle beliebig paarweise miteinander verschalten. Abbildung 34 zeigt, wie man mit einem Switch mit $4n$ Kanälen ein Transputer-System mit n Prozessoren realisieren kann.

Switches mit vielen Kanälen sind technisch sehr aufwendig und teuer. Der folgende Satz zeigt eine Möglichkeit auf, wie man Transputer mit kleineren Switches bauen kann, ohne dabei die Flexibilität einzuschränken.

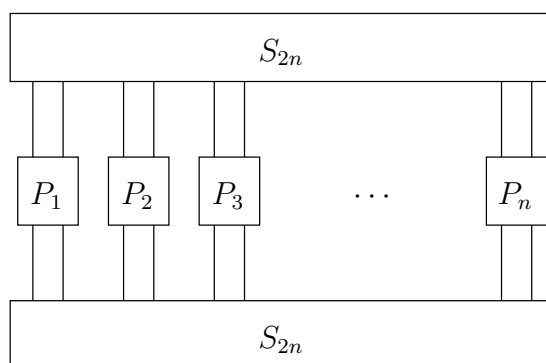


Abbildung 35: Ein Transputer-System mit n Prozessoren und zwei Switches

Satz 5.5 (Satz von Petersen [32])

Sei $G = (V, E)$ ein regulärer Graph vom Grad $2d$. Dann kann man G in d Graphen vom Grad 2 zerlegen, d.h. es existiert eine d -teilige Kantenpartition $E_1 \dot{\cup} E_2 \dot{\cup} \dots \dot{\cup} E_d = E$ mit (V, E_i) ist regulär vom Grad 2 für alle $1 \leq i \leq d$.

Beweis: (für $d = 2$)

Sei K ein Eulerkreis des 4-regulären Graphen G . K besteht aus $|E| = 2 \cdot |V|$ vielen Kanten. Da die Kantenanzahl gerade ist, können die Kanten von K abwechselnd rot und blau gefärbt werden. Eine gesuchte Kantenpartition für G ist dann $E_1 = \{e \in E; e \text{ ist rote Kante}\}$ und $E_2 = \{e \in E; e \text{ ist blaue Kante}\}$, da jedem Knoten in K immer paarweise eine rote und eine blaue Kante zugeordnet wird. ■

Auf 4-reguläre Graphen angewandt bedeutet dies, daß die Kanten in zwei Mengen aufgeteilt werden können und die so entstehenden Teilgraphen 2-regulär sind. Also kann man ein Transputer-System auch wie in Abbildung 35 aufbauen; jeder 4-reguläre Graph kann weiterhin direkt als Netzwerk-Topologie verwendet werden, indem die Kanten in zwei Partition aufgeteilt werden und die erste Kantenpartition dem oberen Switch und die zweite dem unteren Switch zugewiesen wird.

Will man mit noch kleineren Switches ein Transputer-System aufbauen, kann man dies wie in Abbildung 36 tun (siehe [27]). Die Frage, die sich dabei stellt, ist, wie viele Kanten man zwischen den beiden Blöcken vorsehen muß, um gewährleisten zu können, daß man immer noch jeden 4-regulären Graphen direkt auf dem Transputer-System abbilden kann. Man kann zeigen, daß man mit $\sigma_4(n) + 1$ Kanten auskommt. Die oberen Switches sind dabei durch $\lfloor \sigma_4(n)/2 \rfloor$ Kanten miteinander verbunden, die unteren Switches durch $\lfloor \sigma_4(n)/2 \rfloor - 1$ Kanten; ferner benötigt man noch zwei Kanten, die die oberen mit den unteren Switches verbinden. Es reichen also vier Switches mit je mindestens $n + \lfloor \sigma_4(n)/2 \rfloor + 1$ Kanälen, um das Transputer-System zu bauen.

Es verbleibt, den Wert von $\sigma_4(n)$ möglichst genau zu ermitteln. Mit dieser Aufgabe beschäftigen wir uns im restlichen Kapitel. Wir beweisen zunächst eine obere Schranke für die Bisektionsweite von regulären Graphen mit geradem Grad. Das dabei verwendete Verfahren

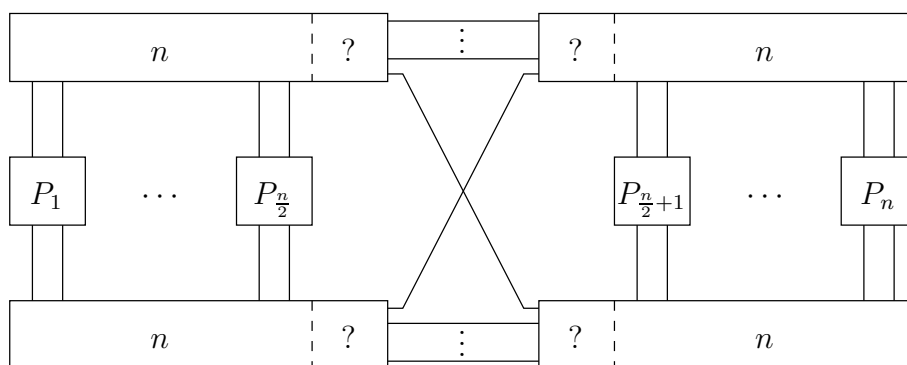


Abbildung 36: Ein Transputer-System mit n Prozessoren und vier Switches

wird anschließend verfeinert, um eine schärfere Schranke für reguläre Graphen vom Grad 4 zu beweisen.

Definition 5.5 (externer/interner Knotengrad)

Sei $G = (V, E)$ ein ungerichteter Graph und $V_1 \dot{\cup} V_2 = V$ eine Partition von G . Für einen Knoten $v \in V_1$ ist sein externer bzw. interner Knotengrad $\text{ext}_{V_1, V_2}(v)$ bzw. $\text{int}_{V_1, V_2}(v)$ definiert durch:

$$\begin{aligned} \text{ext}_{V_1, V_2}(v) &= |\{ \{v, w\} \in E; w \in V_2 \}| \\ \text{int}_{V_1, V_2}(v) &= |\{ \{v, w\} \in E; w \in V_1 \}| \end{aligned}$$

Für einen Knoten $v \in V_2$ ist sein externer bzw. interner Knotengrad entsprechend definiert.

Wenn klar ist, auf welche Partition wir uns beziehen, schreiben wir für den externen bzw. internen Knotengrad eines Knotens v einfach $\text{ext}(v)$ bzw. $\text{int}(v)$. Offensichtlich gilt für alle Knoten v eines Graphen $\text{ext}(v) + \text{int}(v) = \text{grad}(v)$.

Satz 5.6 (obere Schranke für reguläre Graphen mit geradem Grad)

Seien $d, n \in \mathbb{N}$, n gerade. Dann gilt:

$$\sigma_{2d}(n) \leq d \cdot \frac{n}{2}$$

Beweis:

Sei $G = (V, E)$ ein beliebiger regulärer Graph vom Grad $2d$ mit n Knoten und $V_1 \dot{\cup} V_2 = V$ eine beliebige balancierte Partition von G . Wir geben einen einfachen Hill-Climbing-Algorithmus an, der die Partition V_1, V_2 in eine balancierte Partition mit einem Schnitt kleiner gleich $d \cdot n/2$ umwandelt.

Wir teilen dazu die Knoten des Graphen bezüglich der aktuellen Partition in drei Klassen A, B und C ein. Ein Knoten $v \in V$ ist ein

- A-Knoten, falls $\text{ext}(v) > d$
- B-Knoten, falls $\text{ext}(v) = d$
- C-Knoten, falls $\text{ext}(v) < d$

gilt. Beim Verschieben eines Knotens auf die andere Seite der Partition ändert sich der Schnitt in Abhängigkeit von der Klasse, zu der der Knoten gehört. Das Verschieben von einem

- A-Knoten verkleinert den Schnitt
- B-Knoten läßt den Schnitt unverändert
- C-Knoten vergrößert den Schnitt

Der Hill-Climbing-Algorithmus sieht nun wie folgt aus:

```
while „Es gibt A-Knoten in  $V_1$  und  $V_2$ “ do
  Verschiebe A-Knoten von  $V_1$  nach  $V_2$ 
  Verschiebe A- oder B-Knoten von  $V_2$  nach  $V_1$ 
od
```

Die Verschiebe-Operationen im Schleifenrumpf sind wohldefiniert. Das erste Verschieben eines A-Knotens von V_1 nach V_2 ist offensichtlich möglich. Dadurch kann sich der externe Knotengrad von Knoten in V_2 höchstens um 1 verringern, d.h. danach existiert mindestens noch ein A- oder B-Knoten in V_2 .

In jedem Schleifendurchlauf wird ein Knoten von V_1 nach V_2 und ein Knoten von V_2 nach V_1 verschoben, die Partition bleibt also balanciert.

Ferner wird der Schnitt der Partition in jedem Schleifendurchlauf echt kleiner, da mindestens ein A- und ein B-Knoten verschoben werden. Daraus folgt auch, daß der Algorithmus terminiert, da ein Schnitt nur endlich oft verkleinert werden kann.

Nach der Terminierung enthält V_1 oder V_2 keine A-Knoten mehr. O.B.d.A. enthalte V_1 keine A-Knoten mehr. Für den Schnitt gilt dann:

$$\text{ext}(V_1, V_2) = \sum_{v \in V_1} \text{ext}(v) \leq \sum_{v \in V_1} d = d \cdot \frac{n}{2}$$

■

Die gerade beschriebene Idee wollen wir verwenden, um für reguläre Graphen vom Grad 4 eine bessere obere Schranke der Bisektionsweite anzugeben. Dabei werden (ausgehend von einer beliebigen balancierten Partition) pro Runde nicht mehr nur einzelne Knoten, sondern Knotenmengen verschoben.

Definition 5.6 (hilfreiche Menge)

Sei $G = (V, E)$ ein ungerichteter Graph, $V_1 \dot{\cup} V_2 = V$ eine Partition von G und $k \in \mathbb{Z}$. Eine Knotenmenge $S \subseteq V_1$ ($S \subseteq V_2$) heißt k -hilfreiche Menge, wenn das Verschieben von S nach V_2 (V_1) den Schnitt $\text{ext}(V_1, V_2)$ um k vermindert.

Abbildung 37 zeigt Beispiele für hilfreiche Mengen. Der sich für reguläre Graphen vom Grad 4 ergebende Algorithmus zur Bestimmung einer Partition mit kleinem Schnitt sieht grob wie folgt aus:

```
„Solange wie möglich“ do
  1.) Verschiebe 4-hilfreiche Menge von  $V_1$  nach  $V_2$ 
  2.) Balanciere die Partition mittels einer  $(-2)$ -hilfreichen Menge
od
```

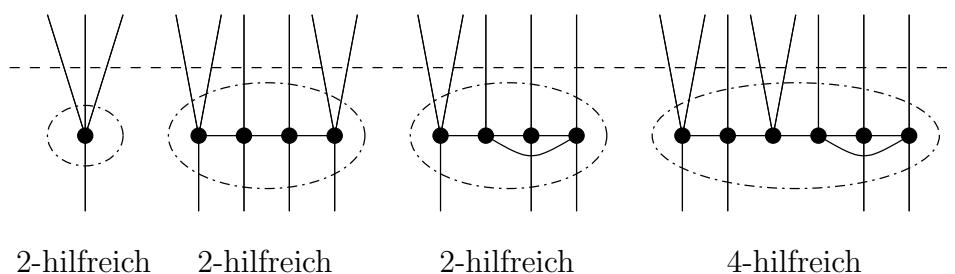


Abbildung 37: Beispiele für hilfreiche Mengen von 4-regulären Graphen

Insgesamt wird der Schnitt dadurch in jedem Schleifendurchlauf um 2 Kanten kleiner. Das Finden einer 4-hilfreichen Menge wird später in Satz 5.8 behandelt. Kommen wir zunächst dazu, wie man Partitionen balancieren kann, ohne den Schnitt um mehr als 2 Kanten zu vergrößern.

Satz 5.7

Sei $G = (V, E)$ ein ungerichteter Graph vom Grad 4 mit gerader Knotenanzahl $n = |V|$. Zu einer Partition $V_1 \dot{\cup} V_2 = V$ mit $|V_1| \leq |V_2|$ und $4 \cdot |V_2| < 5 \cdot \text{ext}(V_1, V_2) + 2$ läßt sich eine balancierte Partition $U_1 \dot{\cup} U_2 = V$ mit $\text{ext}(U_1, U_2) \leq \text{ext}(V_1, V_2) + 2$ konstruieren.

Beweis:

Sei $e = \text{ext}(V_1, V_2)$ die Größe des ursprünglichen Schnitts. Für den Beweis teilen wir die Knoten bezüglich der aktuellen Partition in vier Klassen ein. Ein Knoten $v \in V$ ist ein

- A-Knoten, falls $\text{ext}(v) \geq 3$
- B-Knoten, falls $\text{ext}(v) = 2$
- C-Knoten, falls $\text{ext}(v) = 1$
- D-Knoten, falls $\text{ext}(v) = 0$

Zuerst modifizieren wir die Partition derart, daß in V_2 nur noch C- und D-Knoten vorkommen und die Bedingung $4 \cdot |V_2| < 5 \cdot \text{ext}(V_1, V_2) + 2$ weiterhin erfüllt ist.

```

while  $|V_1| < |V_2|$  do
  if „Es existiert A- oder B-Knoten in  $V_2$ “ then
    Verschiebe A- bzw. B-Knoten von  $V_2$  nach  $V_1$ 
  elseif  $\text{ext}(V_1, V_2) < e$  then
    Verschiebe C-Knoten von  $V_2$  nach  $V_1$ 
  else
    Breche Schleife ab
fi
od
    
```

Es werden alle A- und B-Knoten von V_2 nach V_1 verschoben. Dadurch kann sich die Knotenanzahl von V_2 und die Größe des Schnitts verkleinern. Hat sich der Schnitt verkleinert, ist unter Umständen die Bedingung $4 \cdot |V_2| < 5 \cdot \text{ext}(V_1, V_2) + 2$ nicht mehr erfüllt. Deshalb

werden dann so lange C-Knoten von V_2 nach V_1 verschoben, bis die ursprüngliche Schnittgröße wieder erreicht ist; in V_2 kann es nicht nur D-Knoten geben, da der Graph sonst nicht zusammenhängend wäre. Im Verlauf dieser Modifikationen kann es passieren, daß bereits eine balancierte Partition (mit einem Schnitt $\leq e$) gefunden wird. In diesem Fall ist der Satz bewiesen.

Wir können nun also annehmen, daß $|V_1| < |V_2|$ und $4 \cdot |V_2| < 5 \cdot \text{ext}(V_1, V_2) + 2$ gilt. Ferner enthält V_2 nur C- und D-Knoten. Im folgenden betrachten wir die Zusammenhangskomponenten, die nur aus C-Knoten bestehen, genauer. Sei dazu $C \subseteq V_2$ eine beliebige Zusammenhangskomponente, die nur aus C-Knoten in V_2 besteht, mit $c = |C|$ Knoten. Als Zusammenhangskomponente hat C mindestens $c - 1$ Kanten, die innerhalb von C verlaufen; ferner besitzt C genau c externe Kanten zu Knoten aus V_1 . Es verbleiben also höchstens $4c - 2(c - 1) - c = c + 2$ Kanten, die C mit den restlichen Knoten aus V_2 verbinden. Somit ist C mindestens (-2) -hilfreich, d.h. das Verschieben von C nach V_1 vergrößert den Schnitt höchstens um 2; enthält C gar einen Kreis, so ändert sich der Schnitt nicht.

Wir zeigen im folgenden, daß es in V_2 immer eine Zusammenhangskomponente aus C-Knoten mit einem Kreis gibt. Unter dieser Voraussetzung läßt sich die Partition offensichtlich wie folgt balancieren, ohne den Schnitt um mehr als 2 Kanten zu vergrößern:

```
while  $|V_1| < |V_2|$  do
  Sei  $C$  eine Zusammenhangskomponente mit Kreis aus C-Knoten in  $V_2$ 
  if  $|C| < (|V_2| - |V_1|)/2$  then
    Verschiebe alle Knoten aus  $C$  von  $V_2$  nach  $V_1$ 
    /* Der Schnitt bleibt unverändert. */
  else
    Sei  $C' \subseteq C$  eine Zusammenhangskomponente mit  $(|V_2| - |V_1|)/2$  Knoten
    /*  $C'$  enthält nicht unbedingt einen Kreis. */
    Verschiebe alle Knoten aus  $C'$  von  $V_2$  nach  $V_1$ 
    /* Die Partition ist balanciert; maximal 2 zusätzliche Kanten im Schnitt. */
  fi
od
```

Es verbleibt zu zeigen, daß es in V_2 immer eine Zusammenhangskomponente aus C-Knoten mit einem Kreis gibt.

Ann.: Es gibt keine Zusammenhangskomponente aus C-Knoten mit Kreis.

Sei C (D) die Menge aller C-Knoten (D-Knoten) in V_2 , $c = |C|$ ($d = |D|$) deren Anzahl und $\text{ext}(C, D)$ die Anzahl der Kanten zwischen Knoten aus C und D . Offensichtlich gilt $\text{ext}(C, D) = 4d - 2 \cdot \text{int}(D) \leq 4d$. Ferner bilden nach unserer Annahme die C-Knoten eine Wald, d.h. es verlaufen maximal $c - 1$ Kanten innerhalb von C . Also:

$$\begin{aligned} \text{ext}(C, D) &= 4c - 2 \cdot \text{int}(C) - \text{ext}(C, V_1) \geq 4c - 2(c - 1) - c = c + 2 \\ \implies 4d &\geq \text{ext}(C, D) \geq c + 2 \end{aligned}$$

Zusammen mit $c + d = |V_2|$ und $c = \text{ext}(V_1, V_2)$ folgt:

$$4 \cdot |V_2| = 4(c + d) = 4c + 4d \geq 4c + c + 2 = 5c + 2 = 5 \cdot \text{ext}(V_1, V_2) + 2$$

Dies ist offensichtlich ein Widerspruch zu der Voraussetzung $4 \cdot |V_2| < 5 \cdot \text{ext}(V_1, V_2) + 2$.

Fazit: Es gibt in V_2 eine Zusammenhangskomponente aus C-Knoten mit Kreis. ■

Nachdem wir gesehen haben, wie sich Partitionen balancieren lassen, folgt nun ein Verfahren, mit dem sich (kleine) 4-hilfreiche Mengen in balancierten Partitionen finden lassen.

Satz 5.8

Sei $G = (V, E)$ ein ungerichteter Graph vom Grad 4 mit gerader Knotenanzahl $n = |V|$ und $V_1 \dot{\cup} V_2 = V$ eine balancierte Partition mit $\text{ext}(V_1, V_2) > n/2 + 1$. Dann läßt sich entweder eine balancierte Partition mit kleinerem Schnitt oder eine 4-hilfreiche Menge H mit $|H| \leq 6\lceil \log n \rceil + 2$ finden.

Beweis:

Wir unterscheiden zwei Fälle.

1. Fall: V_1 und V_2 enthalten je mindestens einen A-Knoten

Durch das Vertauschen von einem A-Knoten aus V_1 mit einem A-Knoten aus V_2 wird der Schnitt um mindestens 2 kleiner und die resultierende Partition ist balanciert.

2. Fall: V_1 oder V_2 enthält keine A-Knoten

O.B.d.A. enthalte V_1 keine A-Knoten. Wir zeigen, daß V_1 eine hinreichend kleine 4-hilfreiche Menge enthält. b, c bzw. d stehe für die Anzahl der B-, C- bzw. D-Knoten in V_1 . Es gilt $b + c + d = |V_1| = n/2$ und $\text{ext}(V_1, V_2) = 2b + c > n/2 + 1$. Also:

$$2b + c \geq n/2 + 2 = b + c + d + 2 \implies b \geq d + 2$$

Das heißt, es gibt (mindestens zwei) B-Knoten in V_1 . Wir führen von jedem B-Knoten v eine Baumsuche durch. Dabei führen wir die Suche an einem Knoten w nicht weiter fort, wenn es sich bei w um einen D-Knoten handelt oder der Abstand von w zu v $\lceil \log n \rceil$ beträgt. Für jeden B-Knoten v stehe $T(v)$ für den sich aus der Baumsuche ergebenden Teilgraph von V_1 und $s(v)$ für die Anzahl der D-Knoten in $T(v)$.

Nach Konstruktion haben alle Knoten in $T(v)$ einen Abstand $\leq \lceil \log n \rceil$ zur Wurzel und innerhalb des kürzesten Weges von einem Knoten zur Wurzel liegt kein D-Knoten. Bei den inneren Knoten eines kürzesten Weges handelt es sich also nur um B- oder C-Knoten. Wie Abbildung 38 zeigt, hat ein solcher innerer Pfad, unabhängig von seiner Länge, keinen Einfluß auf die „Hilfreichheit“; wie hilfreich ein kürzester Weg ist, entscheiden die beiden Endknoten. Kommt im inneren Pfad ein B-Knoten vor, erhöht sich die „Hilfreichheit“ des gesamten Weges dadurch sogar noch; für die folgenden Argumentationen genügt es zu wissen, daß ein innerer Pfad die „Hilfreichheit“ nicht verschlechtert. In den weiteren Abbildungen wird deshalb ein solcher innerer Pfad nur gepunktet dargestellt.

a) Es ist $s(v) \geq 2$ für alle B-Knoten v .

In allen $T(v)$ -Teilgraphen zusammen tauchen mindestens $2b$ Kanten zu D-Knoten auf. Wegen $b \geq d + 2$ sind das mindestens $2d + 4$ Kanten. Da andererseits insgesamt nur $4d$ D-Kanten existieren, gibt es entweder einen D-Knoten, der zu 4 Teilgraphen gehört oder 2 D-Knoten, die in jeweils 3 Teilgraphen vorkommen. Im ersten Fall haben wir eine $(4\lceil \log n \rceil + 1)$ -elementige und im zweiten Fall eine $(6\lceil \log n \rceil + 2)$ -elementige 4-hilfreiche Menge gefunden, wie die Abbildungen 39 und 40 zeigen.

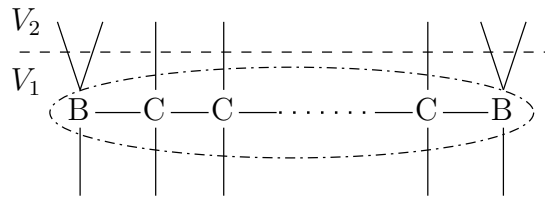


Abbildung 38: Innere Knoten haben keinen Einfluß darauf, ob ein Pfad hilfreich ist

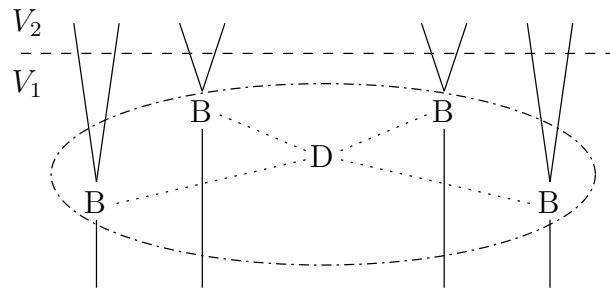


Abbildung 39: 4-hilfreiche Menge: D-Knoten kommt in 4 Suchbäumen vor

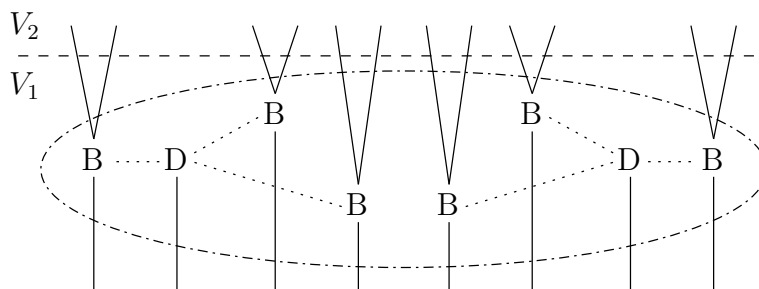


Abbildung 40: 4-hilfreiche Menge: zwei D-Knoten kommen in je 3 Suchbäumen vor

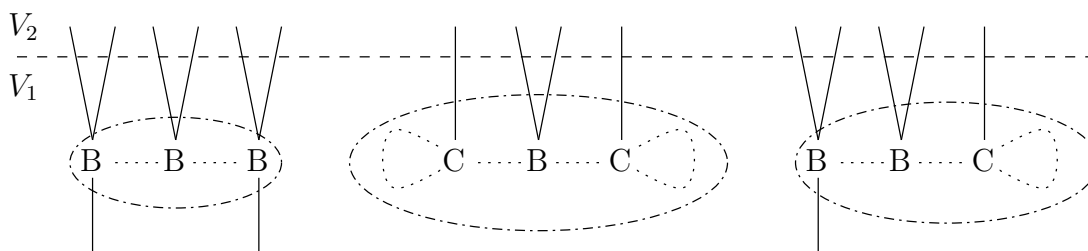


Abbildung 41: Mögliche 4-hilfreiche Mengen in einem Suchbaum

b) Es gibt einen B-Knoten v mit $s(v) \leq 1$.

Im Suchbaum von v kommt eine der drei 4-hilfreichen Mengen aus Abbildung 41 vor, d.h. $T(v)$ enthält entweder zwei weitere B-Knoten, zwei Kreise oder einen weiteren B-Knoten und einen Kreis. Die Größe der 4-hilfreichen Menge ist dabei maximal $4\lceil \log n \rceil + 1$. Daß eine dieser drei Konstellationen auftreten muß, zeigen wir mittels eines Widerspruchsbeweises.

Ann.: $T(v)$ enthält höchstens einen weiteren B-Knoten oder einen Kreis, aber nicht beides.

Wir zeigen, daß $T(v)$ dann aus mehr als $|V_1| = n/2$ Knoten bestehen muß. Wir modifizieren $T(v)$, indem wir etwaige Kreiskanten und D-Knoten löschen und anschließend alle (inneren) Knoten mit Grad 2 (außer der B-Wurzel v) durch eine entsprechende Kante ersetzen. Der resultierende Graph $T'(v)$ ist ein binärer Baum, in dem alle inneren Knoten zwei Söhne besitzen.

Welche Knoten werden bei der Modifikation durch eine Kante ersetzt? Zum einen der eventuell vorhandene B-Knoten bzw. ein C-Knoten, dessen Kreiskante gelöscht worden ist (nur einer der beiden Fälle ist nach unserer Annahme möglich). Zum anderen ein Knoten, der Nachbar eines D-Knotens war. Davon kann es nach Voraussetzung ebenfalls maximal einen geben.

In $T'(v)$ ist also der Abstand eines jeden Knotens zur Wurzel höchstens um 2 kleiner geworden. Da wir $T(v)$ bis zu einer Tiefe von $\lceil \log n \rceil$ aufgebaut haben, muß es deshalb in $T'(v)$ einen vollständigen binären Baum der Tiefe $\lceil \log n \rceil - 2$ geben. $T'(v)$ enthält also mindestens $2^{\lceil \log n \rceil - 1} - 1 \geq n/2 - 1$ Knoten. $T'(v)$ ist aus $T(v)$ durch Entfernen zweier Knoten hervorgegangen, also enthält $T(v)$ mindestens $n/2 + 1 > |V_1|$ Knoten, was den gewünschten Widerspruch liefert.

Fazit: In $T(v)$ ist eine der drei 4-hilfreichen Mengen aus Abbildung 41 enthalten. ■

Mit den Ergebnissen der Sätze 5.7 und 5.8 läßt sich nun eine obere Schranke für die Bisektionsweite von 4-regulären Graphen angeben.

Satz 5.9 (obere Schranke für reguläre Graphen mit Grad 4)

Sei $n \geq 466$, n gerade. Dann gilt:

$$\sigma_4(n) \leq \frac{n}{2} + 1$$

Beweis:

Sei $G = (V, E)$ ein beliebiger regulärer Graph vom Grad 4 mit $n = |V|$ Knoten und $V_1 \dot{\cup} V_2 = V$ eine beliebige balancierte Partition. Folgender Algorithmus berechnet eine balancierte Partition mit $\leq n/2 + 1$ Kanten:

```

while ext( $V_1, V_2$ ) >  $n/2 + 1$  do
  Verkleinere Schnitt oder bestimme 4-hilfreiche Menge  $H$  (Satz 5.8)
  if „4-hilfreiche Menge  $H$  gefunden“ then
    Verschiebe alle Knoten aus  $H$  auf die andere Seite der Partition
    Balanciere Partition mit (-2)-hilfreicher Menge (Satz 5.7)
  fi
od

```

Korrektheit und Terminierung des Algorithmus sind klar, wenn wir zeigen können, daß Satz 5.7 angewendet werden darf. Sei dazu U_1, U_2 mit $|U_1| \leq |U_2|$ die Partition, die sich nach dem Verschieben von H ergibt. Es gilt $|U_2| \leq n/2 + 6\lceil \log n \rceil + 2$ und $\text{ext}(U_1, U_2) \geq n/2 - 2$. Um Satz 5.7 anwenden zu können, muß gelten:

$$\begin{aligned}
& 4 \cdot |U_2| < 5 \cdot \text{ext}(U_1, U_2) + 2 \\
\implies & 4\left(\frac{n}{2} + 6\lceil \log n \rceil + 2\right) = 2n + 24\lceil \log n \rceil + 8 < 5\left(\frac{n}{2} - 2\right) + 2 = \frac{5n}{2} - 8 \\
\implies & \frac{n}{2} - 24\lceil \log n \rceil > 16 \\
\implies & n \geq 466
\end{aligned}$$

■

Durch eine etwas genauere Analyse läßt sich Satz 5.8 noch verschärfen und Satz 5.9 gilt dann für $n \geq 340$. Ohne Beweis geben wir im folgenden Satz weitere Ergebnisse zu der maximalen Bisektionsweite von regulären Graphen mit Grad 4 an.

Satz 5.10

Sei $n \in \mathbb{N}$ gerade. Dann gilt:

- 1.) $\sigma_4(n) \leq n/2 + 4$ für $n \bmod 4 = 0$
- 2.) $\sigma_4(n) \leq n/2 + 3$ für $n \bmod 4 = 2$
- 3.) $\sigma_4(n) \leq n/2 + 1$ für $n \geq 340$
- 4.) $\sigma_4(n) \leq (0.4 + \epsilon)n$ für $n \geq n(\epsilon)$

Für $n \leq 20$ sind die Schranken 1.) und 2.) scharf. Abbildung 42 zeigt beispielsweise einen Graphen mit 12 Knoten und Bisektionsweite 10.

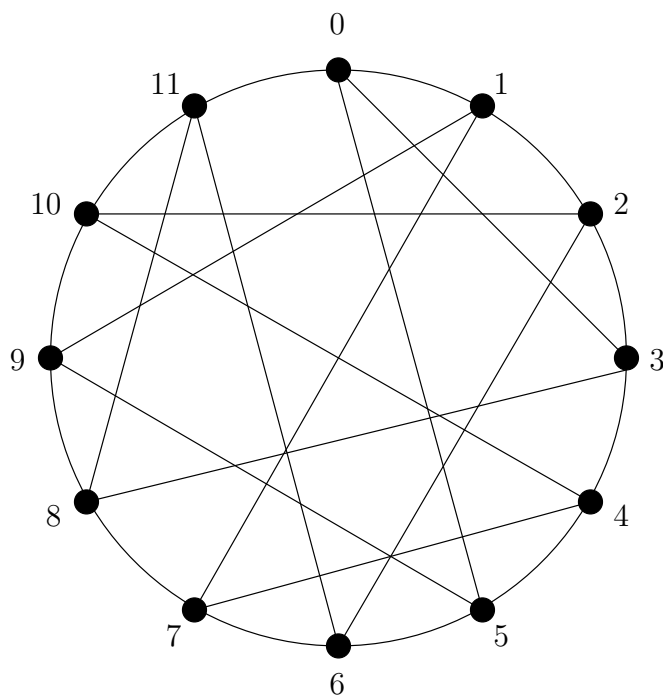


Abbildung 42: Graph mit 12 Knoten und Bisektionsweite 10

Literaturverzeichnis

- [1] H. Alt, T. Hagerup, K. Mehlhorn, F.P. Preparata, *Deterministic simulation of idealized parallel computers on more realistic ones*, Proc. of 12th Symposium on Mathematical Foundation of Computer Science, pages 199-208, 1986. Lecture Notes in Computer Science 233 (eds. J. Grunski, B. Rovan, J. Wiedermann), Springer Verlag.
- [2] K. Appel, W. Haken, *Every planar map is four colorable. Part 1: Discharging and Part 2: Reducibility*, Illinois Journal of Mathematics 21, pages 429-567, 1977.
- [3] Y. Azar, U. Vishkin, *Tight comparison bounds on the complexity of parallel sorting*, SIAM Journal on Computing 16, pages 458-464, 1987.
- [4] I. Bar-On, U. Vishkin, *Optimal parallel generation of a computation tree form*, ACM Transactions on Programming Languages and Systems 7, pages 348-357, 1985.
- [5] S. Bettayeb, Z. Miller, I.H. Sudborough, *Embedding grids into hypercubes*, Proc. of Aegean Workshop on Computing, Lecture Notes in Computer Science, Springer Verlag, Vol. 319, pages 200-211, 1988.
- [6] A. Borodin, J.E. Hopcroft, *Routing, merging and sorting on parallel models of computation*, J. Computer and System Sciences 30, pages 130-145, 1985.
- [7] R.P. Brent, *Parallel evaluation of general arithmetic expressions*, Journal of the ACM 21, pages 201-208, 1974.
- [8] R.L. Brooks, *On colouring the nodes of a network*, Proc. Cambridge Philos. Soc. 37, pages 194-197, 1941.
- [9] M.Y. Chan, *Embedding of grids into optimal hypercubes*, SIAM Journal on Computing, Vol. 20, No 5, pages 834-864, 1991.
- [10] F.Y. Chin, J. Lam, I. Chen, *Efficient parallel algorithms for some graph problems*, Communications of the ACM 25, 1982.
- [11] M. Chrobak, K. Diks, T. Hagerup, *Parallel 5-coloring of planar graphs*, SIAM Journal on Computing 18, pages 288-300, 1989.
- [12] R. Cole, *Parallel merge sort*, SIAM Journal on Computing 17, pages 770-785, 1988.
- [13] R. Cole and U. Vishkin, *Approximate and exact parallel scheduling with applications to list, tree and graph problems*, Proc. of 27th IEEE Symposium on Foundations of Computer Science (FOCS), pages 478-491, 1986.

- [14] R. Cole and U. Vishkin, *Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms*, Proc. of 18th ACM Symposium on Theory of Computing (STOC), pages 206-219, 1986.
- [15] S.A. Cook *The complexity of theorem-proving procedures*, Proc. 3rd Ann. ACM Symposium on Theory of Computing, pages 151-158, 1971.
- [16] T.H. Cormen, C.E. Leiserson, R.L. Rivest, *Introduction to algorithms*, McGraw-Hill (1991).
- [17] R. Feldmann, P. Mysliewitz, *The shuffle exchange network has a Hamiltonian path*, Proc. of 17th Math. Foundations of Computer Science (MFCS 1992), Springer 629, pages 246-254.
- [18] R. Feldmann, W. Unger, *The cube-connected-cycles network is a subgraph of the butterfly network*, Parallel Processing Letters Vol. 2, No. 1, World Scientific Publishing Company, pages 13-19, 1992.
- [19] F.E. Fich, P. Radge, A. Widgerson, *Relations between concurrent-write models of parallel computation*, SIAM Journal on Computing, Vol. 17, No. 3, pages 606-627, 1988.
- [20] M.R. Garey, D.S. Johnson, *Computers and intractability: a guide to the theory of NP-completeness*, Freeman, 1979.
- [21] A.M. Gibbons, W. Rytter, *An optimal parallel algorithm for dynamic evaluation and its applications*, 6th Conference on Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science 241, pages 453-469, Springer Verlag, 1986.
- [22] A.V. Goldberg, S.A. Plotkin, G.E. Shannon, *Parallel symmetry-breaking in sparse graphs*, Journal of Discrete Mathematics, No. 1, pages 434-446, 1988.
- [23] L.M. Goldschlager, *A unified approach to models of synchronous parallel machines*, Symposium on Theory of Computing, pages 89-95, 1978.
- [24] R. Greenlaw, H.J. Hoover, W.L. Ruzzo, *Limits to parallel computation: P-completeness theory*, Oxford University Press, 1995.
- [25] D.S. Hirschberg, A.K. Chandra, D.V. Sarvate, *Computing connected components on parallel computers*, Communications of the ACM 22, pages 461-464, 1979.
- [26] J. Hromkovič, R. Klasing, B. Monien, R. Peine, *Dissemination of Information in Interconnection Networks (Broadcasting & Gossiping)*, Combinatorial Network Theory (eds. D.-Z. Du, D.F. Hsu), Kluwer Academic Publishers, pages 125-212, 1996
- [27] J. Hromkovič, B. Monien, *The Bisection Problem for Graphs of Degree 4 (Configuring Transputer Systems)*, Mathematical Foundations of Computer Science 1991 (MFCS91), Lecture Notes in Computer Science 520, pages 211-220, Springer Verlag, 1991
- [28] D.B. Johnson, P. Metaxas, *Connected components in $O(\log n)^{3/2}$ parallel time for the CREW-PRAM*, FOCS 1992.

- [29] W. Knödel, *New gossips and telephones*, Discrete Math. 13, page 95, 1975.
- [30] F. T. Leighton: *Einführung in parallele Algorithmen und Architekturen: Gitter, Bäume und Hypercubes*, Int. Thomson Publishing, 1997.
- [31] B. Monien, I.H. Sudborough, *Embedding one interconnection network in another*, Computing Suppl. 7, pages 257-282, 1990.
- [32] J. Petersen, *Die Theorie der regulären Graphs*, Acta Mathematica 15, pages 193-220, 1891
- [33] M. Röttger, U.-P. Schroeder, W. Unger, *Embedding 3-dimensional grids into optimal hypercubes*, Proc. of the 1st Canada-France Conference on Parallel Computing (CFPC 1994), Springer LNCS 805, pages 81-94.
- [34] W. Rytter, *The complexity of two way pushdown automata and recursive programs*, Combinatorial Algorithms on Words (eds. A. Apostolica, Z. Galil), NATO ASI Series F:12, Springer Verlag, 1985.
- [35] D.S. Scott, J. Brandenburg, *Minimal mesh embeddings in binary hypercubes*, IEEE Transaction on Computers, Vol. 37, No. 10, 1988.
- [36] R. Sedgewick, *Algorithms*, Addison-Wesley (1984).
- [37] Y. Shiloach, U. Vishkin, *An $O(\log n)$ parallel connectivity algorithm*, Journal of Algorithms 2, pages 57-63, 1981.