F. Parallele Algorithmen

Autor: Markus Möller

- Bezug: Vorlesung bei Herrn Monien und die dort angegebene Referenzliteratur.
- Dieser Extrakt entstand als Vorbereitung auf meine Diplomprüfung (Teil: Theoretische Informatik). Er faßt einige Themen einfach zusammen und mag etwas unorthodox erscheinen.
- Dieser Text ist unvollständig. Fehlende wichtige Themen:
  - Sortieralg. von Cole und
  - Färben von Graphen.
- Auf dem Uni-WWW-Server (AG Monien) liegt die aktuellste Version dieser Vorlesung.
- Erstellt auf Apple Macintosh.

1. Minimum auf Q mit Ascend

Die Idee ist, bei jedem Schritt die Informationen auf die Hälfte der Knoten zu verdichten. Im dreidimensionalen Fall sieht das so aus: Erst schicken die vier rechten Knoten ihre Information zu ihren linken Nachbarn, von denen sie sich –per Numerierung– nur in der letzten Stelle unterscheiden. Im zweiten Schritt arbeiten nur noch die linken vier Knoten. Die hinteren beiden, die sich von den vorderen in der zweiten Stelle unterscheiden, senden ihre Information zu ihrem vorderen Nachbarn jeweils. Im dritten Schritt (drei Dimensionen!) schickt der obere linke vordere Knoten seine Information zum unteren linken vorderen; sie differieren in der vordersten Stelle.

Bei einem Ascend-Lauf werden jedoch alle Knoten in die –je nach Problem differierende– Operation einbezogen, die alle Kanten der aktuellen Dimension betrachtet. Hier ist es „Minimum“ und es ist nicht nötig, daß Knoten, die ihre Info verschickt haben nochmal aktiv werden. Also baut man in die von Ascend aufgerufene Min-Prozedur eine Abfrage ein, ob der Knoten eine Nummer hat, die am Ende aktuelle-Dimension-viele Nullen aufweist (ein Ascend-Lauf geht von 0 bis k-1). Ist das der Fall, dann bestimmt er das Minimum von sich und seinem Nachbarn, der auch so viele Nullen hinten hat und sich nur in der darauffolgenden Stelle unterscheidet.

Das Problem ist einfacher, als es sich erklären läßt.

2. (Hamilton-) Kreise in Netzen

2.1 Kreise im Hypercube

Warum enthält der Hypercube der Dimension k alle Kreise der Länge $r \quad \{2, 4, 6, \quad , 2^k\}$ **?**

Liegt an der rekursiven Struktur des Hypercube. Er besteht jeweils aus zwei kleineren, deren äquivalente Knoten miteinander verbunden sind:

- Induktionsvoraussetzung:
  Q(1) mit 2 Knoten und Q(2) mit 4 Knoten haben einen Kreis der Länge 2 bzw. 2 und 4.
- Induktionsannahme:
  Q(k) hat alle geraden Kreise der Länge $r \quad \{2, 4, 6, \quad , 2^k\}$.
- Induktionsschritt:
  Q(k+1) ist aus zwei Q(k) zusammengesetzt, deren äquivalente Knoten verbunden sind, enthält also alle Kreise der Länge $r \quad \{2, 4, 6, \quad , 2^k\}$. Einen Kreis der Länge $r \quad \{2^k + 2, \quad , 2^{k+1}\}$ erhält man, indem man in einem Q(k) einen Kreis der Länge 2^k nimmt, diesen an einer Stelle auftrennt, in den anderen Q(k) geht, dort einen Kreis der Länge nimmt die noch fehlt (zwischen $2, \quad , 2^k$), diesen auch auftrennt an der äquivalenten Stelle und zurück in den ersten Teil-Q(k) geht.
  Die beiden Verbindungen zwischen den aufgetrennten Teilkreisen gibt es, weil die Knoten äquivalent sind in den beiden Teilcubes und somit per Definition des Q über Kanten verbunden sind. Die beiden Kanten (betrifft Gesamtkreislänge), die aufgetrennt werden, sind durch die beiden neu verwendeten Verbindungen wieder ausgeglichen.

Warum gibt es nur Kreis gerader Länge in Q (und keine ungeraden)?

Da man zum Startknoten zurückgehen muß bei einem Kreis und Nachbarknoten sich in genau einem Bit der Numerierung unterscheiden beim Hypercube, hat man keine andere Wahl, als jedes Bit, was beim „Weglaufen“ umgekippt wurde beim „Zurückkommen“ wieder umzusetzen. Bei einem ungeraden Kreis würde man nicht zum Startknoten zurückgelangen können, da man zuviel (oder zu wenig) Bits zurückkippt. Nur bei geraden Wegen hat man die Chance, einen Kreis zu laufen.

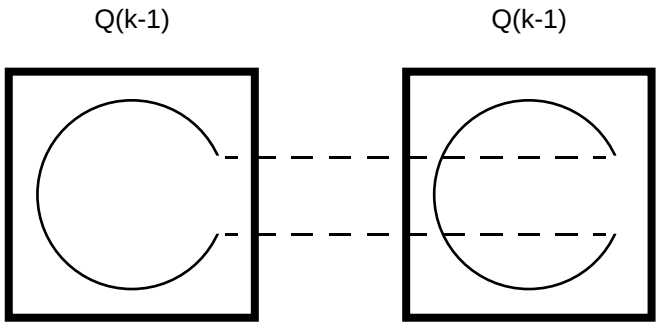
Siehe dazu auch Routingstrategie im Hypercube (zur Not im Skript).

Warum gibt es einen Hamiltonkreis in Q?

Ein Hamiltonkreis ist ein Kreis, der alle Knoten genau einmal durchläuft. (Ein Eulerkreis dagegen durchläuft alle Kanten genau einmal). Der Hamiltonkreis darf demzufolge Kanten auslassen.

Der Hamiltonkreis ist ein Spezialfall des oben Geschilderten. Er ist ein Kreis der Länge 2^k, der keinen Knoten doppelt besucht. Man benutzt in beiden Teilcubes folglich einen Hamiltonkreis der Länge 2^{k-1} und nutzt die Verbindung an einer beliebigen äquivalenten Stelle, um den Gesamt-Hamiltonkreis zu konstruieren. (Induktion; im Q(2) hat man einen Hamiltonkreis mit Länge 4 und gemäß Beweis oben dann auch im Q(k).)

ABBILDUNG 4: Hamiltonkreis im Q(k)



Bemerkung: Es gibt mehr als einen Hamiltonkreis im Hypercube. Dies ist leicht am dreidimensionalen Fall zu sehen.

2.2 Hamiltonkreis im CCC(k) und BF(k)

Definition der beiden Netztypen >Bereich 7.1 auf Seite 10 und >Bereich 8.2 auf Seite 14. Für k≥2 enthalten beide einen Hamiltonkreis.

Laut Vorlesung ohne Beweis.

2.3 Hamiltonkreis in SE(k)

Laut Buch.

3. Bitones Sortieren auf dem Hypercube

Das bitone Mischen sortiert eine bitone Folge korrekt. Das wird mit einem Descendlauf realisiert. Die Vertauschungen finden also zuerst auf den größeren Distanzen statt.

Nach der ersten Vertauschungsrunde auf höchster Dimension stehen in der linken Hälfte die kleineren Zahlen, in der rechten die größeren. Die linke und die rechte Folge sind wieder jeweils biton. Die nächste Iteration von Descend vertauscht Zahlen halber Distanz im Vergleich zum vorherigen Durchgang. Diesmal sind die beiden linken Viertel kleiner als die beiden rechten. In der letzten Iteration von Descend werden Zahlenpaare getauscht. Betrachtet man das Gesamtergebnis, dann gilt: Alle Paare (disjunkt) sind sortiert, denn bitones Sortieren einer Zweierfolge sortiert im herkömmlichen Sinne. Und: rekursiv ist jeweils die linke Folge kleiner als die rechte Hälfte. Damit ist die Gesamtfolge sortiert.

Um jedoch so vorgehen zu können, muß die Gesamtfolge biton sein. Ist sie aber noch nicht.

Deswegen geht man rekursiv vor: Man teilt die zu sortierende Folge in eine linke und eine rechte. Die linke wird wieder rekursiv aufsteigend, die rechte rekursiv absteigend sortiert. Dann ist nach Beendigung der Rekursion die Gesamtfolge biton. Darauf läßt man dann Descend los.

Man macht insgesamt mehrere Descendläufe, die auf linken Folgen aufsteigend, auf rechten Folgen absteigend sind.: Descend(Bitonic Merge) auf 0ter Dimension. Dann ab Dimension 1 (abwärts „descend“), dann ab Dimension 2, …, dann ab oberster Dimension. Der Aufruf von Descend mit Dimension 0 liefert jeweils zwei sortierte Zahlen, die abwechselnd aufsteigend und absteigend sortiert sind. Der Aufruf von Descend mit Dimension 1 benutzt jeweils zwei benachbarte sortierte Paare, die insgesamt eine bitone Folge darstellen. Nach der ersten Runde stehen links die beiden kleineren Zahlen. Nach der zweiten Runde werden die Paare evtl.

nochmal intern getauscht. Der dritte Descendlauf benutzt jeweils zwei benachbarte sortierte Quadrupel, die wieder eine bitone Folge darstellen. Usw.

Der rekursive Alg. für Sortiere alle Zahlen in A auf-/absteigend:

1. Sortiere alle Zahlen A[z0x] (x hat Länge k-1) aufsteigend.
2. Sortiere alle Zahlen A[z1x] (x hat Länge k-1) absteigend.
3. Descend(bitonic merge) ab Dimension k.
   (Die Zahlen A[y] (y hat Länge k) sind bitone Folge!)

Aufruf mit maximalem k= Anzahl der bits.

Sobald sich die Rekursion bis auf die unterste Dimension runtergeschraubt hat, wird zum ersten Male Descend aufgerufen. Der hat nicht viel zu tun und sortiert entweder auf- oder absteigend jeweils (log n mal insgesamt, parallel) zwei Zahlen. Das ist der Punkt, wo das bitone Mischen identisch ist mit Sortieren: Bei Folgen aus zwei Zahlen. Die kleinere wird nach links getauscht.

Dann kommt man eine Rekursionsebene zurück. Dimension 1. Dort sind nun die ersten beiden Schritte abgearbeitet. D.h. man hat log n / 2 bitone Folgen mit jeweils 4 Elementen. Man macht also zwei Descendrunden. Eine auf Dimension 1, eine auf 0. Da die 4er Folgen biton sind, weil die 2er Folgen auf- und absteigen sind, bringt Descend nun sortierte 4er Folgen. Runde 1 sorgt dafür, daß im linken Zweierpack die kleineren Zahlen stehen. Runde 2 auf Dimension 0 sorgt dafür, daß in jedem Zweierpack links die kleinere Zahl steht. So ist jede Viererfolge sortiert: Die zweier sind sortiert. Und der linke zweier ist kleiner als der rechte. Bei der rechten Viererfolge andersrum.

Nach log n Absätzen ist alles sortiert. In jedem Absatz werden maximal log n Descendschritte gemacht. Daher die log^2 Laufzeit

Man nutzt also aus, daß auf Paarebene bitones Mischen und Sortieren identisch sind. Dann wird die Folgenlänge verdoppelt und wieder biton sortiert. Usw. Die jeweils höhere Rekursionsebene nutzt die bitone Sortierung der unteren Rekursion.

> Grundgedanke: Bitones Mischen einer bitonen Folge mit Descend sortiert diese korrekt. Die Ausgangsfolge muß aber biton sein! Also fängt man bei den kleinsten Folgen (Länge 2) an. Diese –biton gemischt– sind dann sortiert und können paarweise als bitone Inputfolge weiterverwendet werden, die ihrerseits biton sortiert wird. Die

Ergebnisfolgen sind zu Paaren gebündelt wieder bitone Folgen…

Exponentiell viele Zahlen werden mit quadratischer Anzahl Prozessoren und Zeit sortiert.

4. Simulation CRCW auf CREW

Mit Hilfe des Sortieralg. von Cole –siehe unten– kann eine CREW-PRAM jeden Schritt einer CRCW mit logarithmischem Zeitverlust simulieren.

D.h. CRCW(p(n), t(n)) CREW(p(n), t(n) log p(n)).

Einer speziellen Simulation bedürfen nur die Schreibzugriffe innerhalb einer parallelen for-Schleife. Sei N die Prozessorenanzahl, S die Größe des globalen Speichers und B, H, W integer-arrays mit N Einträgen.

Möchte also ein Prozessor p_i schreibend auf die Speicherzelle s_i des globalen Speichers A zugreifen, so wird die Anweisung

0. for $0 \le i < N$ parallel do
1. $A[s_i] := x_i$

von der EW wie folgt abgearbeitet:

0. for $0 \le i < N$ parallel do
1. $H[i] := x_i$
2. $W[i] := (s_i, i)$
3. Sort(W)
4. Check(W, H, B)
5. Oder(B)
6. if B[0] = 0 then SWrite(W, H)
7. else Abbruch

Erläuterungen:

1. Im Array **H** wird gespeichert, **was** die Prozessoren schreiben wollen. Dabei ist Zelle i für Prozessor i zuständig. *Konstante Zeit.*
2. Im Array **W** wird gespeichert, **wohin** die Prozessoren schreiben wollen. Das Tupel wird durch s_i S + i codiert. O.B.d.A. ist $S \ge N$ und die Codierung eindeutig somit, da $i < N$. *Konstante Zeit*.

3. Mit **Cole** wird W sortiert. Dann stehen die Tupel aller Prozessoren, die auf die gleiche Zelle s_i zugreifen wollen, hintereinander. *Logarithmische Zeit.*
4. Das Unterprogramm Check testet, ob Schreibkonflikte bestehen. Dazu wird parallel für alle **aufeinanderfolgenden Paare** in W nachgesehen, ob dasselbe geschrieben wird, falls die Zieladresse identisch ist. Testergebnisse landen jeweils im Array B.
   Das genügt, weil W nach Zieladressen sortiert ist. *Konstante Zeit!*
5. Logisches Oder in *logarithmischer Zeit* über das Array B. Damit findet man heraus ob überhaupt ein Schreibkonflikt entsteht.
6. In *konstanter Zeit* schreibt jeweils der Prozessor, dessen Tupel-Nachfolger in W nicht dasselbe Ziel hat seinen Wert.
   Auch hier ist die Sortierung unbedingt notwendig.

> Die Idee war hierbei die geschickte Sortierung der Informationen, wodurch wichtige Operationen in konstanter Simulationslaufzeit möglich wurden. Durch die Sortierung und das Oder auf der CREW fällt man dann jedoch auf „nur“ logarithmischen Zeitaufwand zurück.

5. Abgrenzende Beispiele CREW-CRCW

5.1 „Oder“

ODER EREW(n, $\log n$), also auch CREW

ODER EREW($\frac{n}{\log n}$, $\log n$), also auch CREW

ODER CRCW(n, 1)

1. Balancierte Binärbaummethode
2. Eingabe in log(n) Teile splitten. <Konstante Zeit>
   Mit n/log(n) Prozessoren ODER über die einzelnen Teile sequentiell berechnen. <log(n) Zeit>
   Mit den Prozessoren Binärbaummethode über die n/log(n) Teilergebnisse. <log(n/log(n)) = log(n)-loglog(n) Zeit>
3. Jeder Prozessor, der eine „1“ liest, schreibt diese in die Ergebniszelle. Da alle dasselbe schreiben, wenn sie schreiben, tritt kein Konflikt auf.

5.2 „Minimum“

Min CREW$(n, \log n)$ geht wie oben mit ODER, nur Ändern der assoziativen ODER-Operation in „Kleiner-gleich“. Binärbaummethode.

Min CRCW$(n^2, 1)$ geht mit einer $n \times n$ Matrix von Prozessoren und einer Boolschen Matrix B (Die Zahlen liegen in A(i)):

- Prozessor(i,j) bemerkt, ob A(i)≤A(j), indem er B(i,j) dann auf „true“ setzt. Concurrent Read!
- A(i) ist Min $\Leftrightarrow$ Zeile i von B komplett „true“, d.h. i≤j für alle j.
  Bilde UND über jede Zeile von B, um dies zu testen:
  - Prozessor(i,1) setzt C(i):=true (als Hilfsvariable)
  - Prozessor(i,j): if not B(i,j) then C(i):=false
    Concurrent Write!
- Da C(i) mehrere trues enthalten kann, finde eines:
  Prozessor(i,1): if C(i) then Minimum:= A(i).
  Concurrent Write!

Min CRCW$(n, 1)$ funktioniert, wenn die Zahlen, unter denen das Minimum bestimmt werden soll, alle zwischen 1 und n liegen:

- Prozessor i schreibt „true“ in A(a(i))
  In diesem Schritt wird festgestellt, welche Zahlen in der Eingabe a vorkommen.
  A repräsentiert alle n Zahlen. A ist *kein Array*, sondern eine $\sqrt{n} \times \sqrt{n}$ Matrix mit $i = j\sqrt{n} + k, 0 \le j, k < \sqrt{n}$.
- Bilde ODER über jede Zeile von A. Ergebnis in Speicher B mit $\sqrt{n}$ Elementen. Geht so:
  Prozessor i $(i = j\sqrt{n} + k)$ schreibt „true“ in B(j), falls A(i)=true; $0 \le j, k < \sqrt{n}$.
- Berechne λ, so daß $\lambda = \text{Min}(j, B(j) = \text{true})$.
  In konstanter Zeit mit n Prozessoren möglich, weil Min CRCW$(n^2, 1)$ und „n“ in dieser Formel $\sqrt{n}$ entspricht, da B diese Länge hat.
  Dabei muß die „≤“-Operation erweitert werden um: „und B(A(i))=true“.
- Berechne μ, so daß $\mu = \text{Min}(j, A(\lambda\sqrt{n} + j) = \text{true})$.
  Aufwand wie im Schritt vorher.

Dann ist $\lambda\sqrt{n} + \mu$ das Minimum. Es geht also um folgendes: Man hat die Zahlen 1 bis n fest vorgegeben und stellt nur fest, welche die erste ist, die tatsächlich verwendet wurde. Dazu schreibt man die n Zahlen in eine quadratische Matrix. Nun genügt es, den richtigen Zeilen- und Spaltenindex zu finden. Im zweiten Schritt werden die Zeilen markiert, die eine Eingabe enthalten. Im dritten wird die erste

Zeile ausgegeben, die eine Eingabe enthält. Im letzten Schritt wird der Spaltenindex ausgeben, der die erste Eintragung in der eben gefundenen Zeile bezeichnet. Da die Zeilenlänge und die Zahlen fest sind, kann mit obiger Formel die gefundene Zahl berechnet werden.

6. Listranking

Listranking ist eine Methode, um parallel die Entfernung jedes Elementes einer linearen Liste zur Wurzel zu berechnen.

Listranking	$\mathrm{CREW}(n, \log n)$
Listranking	$\mathrm{CREW}(\frac{n}{\log n}, \log n)$

Das Concurrent Read wird spätestens dann benötigt, wenn einige Prozessoren immer wieder gleichzeitig die Wurzel lesen.

Das zweite (optimale) Ergebnis bekommt man durch das (vom logischen ODER bekannte) geschickte Aufteilen des Datenbereichs in Portionen der Länge log(n).

Verfahren:

- Initialisiere jedes Element mit 1 und die Wurzel mit Entfernung 0.
- Wiederhole für jedes Element ceil(log(n))-mal:
  - Entfernung:= Entfernung + Entfernung des Nachfolgers
  - Nachfolger:= Nachfolger des Nachfolgers

7. Einbettungen

Eine Einbettung von A in B bildet alle Knoten von A injektiv auf Knoten in B ab. Außerdem muß für jede Kante in A ein zugehöriger Weg in B angegeben werden, weil die zu der Kante in A gehörigen Kanten-Knoten in B nicht unbedingt Nachbarn sind.

7.1 CCC(k) in Q(k+⌈log k⌉)

7.1.1 1.Fall: k gerade Zahl

Sei C_k ein Kreis der Länge k in $Q(\lceil \log k \rceil)$. Dieser Kreis existiert; siehe "Kreise im Hypercube" auf Seite 2. Sei ferner $\beta(i)$ die binäre Adresse des i-ten Knoten auf dem Kreis in $Q(\lceil \log k \rceil)$ mit $0 \le i \le k-1$.

Konstruiere Einbettung $\delta((i, \alpha)) = (\beta(i)\alpha)$. Ein CCC(k)-Kreis[1] der Länge k (es gibt nur solche!) wird also auf einen Kreis der (geraden) Länge[2] $k \le 2^{\lceil \log k \rceil}$ in einem Teilcube $Q(\lceil \log k \rceil)$ des $Q(\lceil \log k \rceil + k)$ abgebildet. α hat dabei die Länge k.

Noch einzusehen: Jede Kante in CCC hat einen Weg in Q:

- Kreiskanten in CCC:
  i wird also zu i+1 modulo k und α bleibt. Ist also $\beta((i+1) \bmod k)\alpha$ ein Nachbarknoten von $\beta(i)\alpha$ in $Q(\lceil \log k \rceil)$? Ja, denn die beiden Knoten sind Nachbarn auf dem Kreis C_k.
- Cubekanten in CCC:
  i bleibt gleich und α kippt i-tes Bit um. Ist also $\beta(i)\alpha$ Nachbarknoten von $\beta(i)\alpha(i)$? Ja, denn die Knoten unterscheiden sich nur in genau einem Bit.

7.1.2 2.Fall: k ungerade Zahl

Bette die Kreise der Länge k des CCC(k) in Kreise der Länge k+1 des $Q(\lceil \log k \rceil)$ ein. Rest wie bei 1.Fall. Allerdings hat man hier eine Kantenstreckung von 2 an genau einer Stelle des Kreises, weil ein Knoten im geraden Kreis des Q übersprungen werden muß, um auf das ungerade k zu kommen.

7.2 Gitter in Hypercube

Man kann ein d-dimensionales Gitter $M[n_1, \ldots, n_d]$ in den $Q(\sum_{i=1}^{d} \lceil \log n_i \rceil)$ einbetten. Man benutzt dazu die Eigenschaft des Hypercube(k), Hamiltonkreise der Länge 2^k zu besitzen. Jede „Richtung" des Gitters (bei 2 Dimensionen sind es 2)

1. Jeder Knoten eines CCC befindet auf genau einem Kreis des CCC.
2. siehe "Kreise im Hypercube" auf Seite 2.

wird auf einen Hamiltonkreis eines anderen Teilcubes abgebildet. Man betrachtet bei d Dimensionen d Teilcubes $Q_i(\lceil \log n_i \rceil)$, $1 \le i \le d$.

Der Knoten (i,j) eines zweidimensionalen Gitters(3,5) wird dann auf den Knoten Nummer(i-ter Knoten auf Hamiltonweg in Q(2)) konkateniert mit Nummer(j-ter Knoten auf Hamiltonweg in Q(3)) abgebildet. Da sich die Gitterkanten bilden, indem man genau eine Koordinate um 1 ändert, unterscheiden sich die Abbildungsknoten auch nur in einer Stelle, denn sie sind Nachbarn auf dem betroffenen Hamiltonweg.

7.3 B(k) kein Teilgraph von Q(k+1)

Vollständiger binärer Baum der Höhe k. Seine Knoten sind binäre Zeichenketten der Länge≤k. Seine Kanten verbinden Zeichenketten u der Länge $0 \le i < k$ mit Zeichenketten ua, a ist 0 oder 1, der Länge i+1. Die Wurzel ist mit dem „leeren“ Wort e beschriftet.

Ein Graph ist bipartite, wenn seine Kanten zwischen zwei disjunkten Knotenmengen verlaufen. Q(k+1) ist bipartite mit Knoten, die gerade bzw. ungerade Anzahl von 1en haben. In beiden Mengen sind 2^k Knoten. B(k) ist bipartite mit Knoten, die gerade bzw. ungerade Nummernlänge haben. Abhängig davon, ob k gerade oder ungerade ist, ist entweder die eine oder die andere Menge diejenige, die alle Blätter enthält. Sie hat dann mehr als 2^k Knoten, da soviele allein schon in den Blättern sind. Da alle Kanten aber zwischen diesen Mengen verlaufen (in beiden Netzen jeweils) kann dieses größere Menge nicht eingebettet werden. Also ist B(k) kein Teilgraph von Q(k+1). Es gibt nämlich keine injektive Abbildung für diese Menge in eine der beiden von Q. Es sind zu viele Knoten, die eine Kante nach außen haben. Wenn man die große Menge auf die zwei im Q verteilen würde, dann hätte ein aus der großen Menge herausgenommener Knoten keine Kante mehr, da diese nur zwischen den Mengen verläuft.

Ich versuchs nochmal anders: In B(k) gibt es mehr als 2^k verschiedene Knoten, die disjunkte Kante haben. In Q(k+1) sind aber nur maximal 2^k verschiedene Knoten enthalten, die disjunkte Kanten haben. Es ist also keine injektive Abbildung möglich, es sei denn, man verzichtet auf Kanten. Die verschiedenen Knoten verlieren ihre Kante, wenn sie auf verschiedene Knoten abgebildet werden.

Teilgraph heißt auch Kantenstreckung 1. Man kann aber den vielen Knoten keine direkte Verbindung aus ihrer eigenen Menge heraus für jeden geben in Q.

7.4 DWB(k) Teilgraph von Q(k+1)

Ein Doppelwurzelbaum(k) hat die Höhe k und –verglichen mit dem normalen Baum– eine Kante anstelle der einen Wurzel. Man ersetzt die normale Wurzel also durch zwei mit einer Kante verbundene Knoten. Folglich hat er 2^{k+1} Knoten.

Induktiver Beweis. Mit Automorphismus werden die jeweiligen Doppelwurzelbäume in den Teilcubes passend verschoben und anschließend verschmolzen mit den obersten vier Kanten.

8. Ascend/Descend auf anderen Netzen

8.1 Descend auf SE(k)

Ein Ascend/Descend-Lauf des Q(k) kann auf dem SE(k) in Zeit O(k) abgearbeitet werden.

Was ist SE(k)?

Shuffle-Exchange-Netzwerk. Es hat 2^k Knoten –wie der Hypercube(k)– und zwei Arten von (gerichteten!) Kanten:

- Shuffle:
  Wenn die Knotennummer links raus und rechts rein geshiftet wird um eine Stelle, dann sind sie in dieser Folge verbunden: $a\alpha \rightarrow \alpha a$, mit $\alpha \in \{0,1\}^{k-1}$ und $a \in \{0,1\}$
- Exchange:
  Wenn das letzte Bit der Knotennummer invertiert ist, dann sind sie verbunden: $\alpha a \rightarrow \alpha \bar{a}$, mit $\alpha \in \{0,1\}^{k-1}$ und $a \in \{0,1\}$

>>Routing: Im Prinzip guckt der Algorithmus sich die Bits in der Reihenfolge 0, n-1, n-2,…, 1 an (wegen „Dauer-Shuffle“) und vergleicht sie mit den Zielbits n-1, n-2,…, 0. Wenn sie nicht stimmen jagt er sie über eine Exchangekante. Das Nacheinanderansehen läuft über Shufflekanten (k-1 mal bei k Dimensionen).

Man rotiert sozusagen die Quelle und kippt bei Bedarf die Bits. Beim Routen von 100 nach 101, was über eine einzige Exchangekante ginge, zeigt sich die Nicht-Optimalität des Algorithmus: Er benötigt 5 Kanten und besucht sein Ziel sogar zweimal! Bei 001 nach 011 ist es noch witziger: Er läuft erstmal zurück dreht ‘nen Kreis und routet dann direkt.

Zum Amüsieren bitte Seite 102 des Skriptes studieren!

Wie geht Ascend/Descend auf SE(k)?

Man simuliert einen Ascend-Lauf des Q(k). Die Kommunikation über Dimension 0 geht direkt über die Exchange-Kanten. Das ist so einfach, weil die ungerichtete Kante des Hypercube zwei gerichteten Kanten im SE entspricht auf Dimension 0. Wie sieht es in den höheren Dimensionen aus? Erst rotieren die Prozessoren ihre Informationen über die Shufflekanten, dann tauschen sie sie über Exchangekanten aus.

Der Austausch, der dem Hypercube entspricht findet also immer über dieselben Exchangekanten statt. Nur bei höheren Dimensionen muß vorher über Shufflekanten rotiert werden.

Die Kernfrage ist also: Wenn man zwei Bitstrings hat, die sich nur an der i-ten Stelle unterscheiden; kann man diese Strings durch zyklisches Shiften so umformen, daß sie sich nur an der rechtesten Stelle unterscheiden? Geht. Das machen die Shufflekanten. Dann stehen sich die Daten per Definition SE an Exchangekanten gegenüber, die sich an der untersten Stelle unterscheiden.

Erinnerung: Beim Hypercube findet die Operation immer zwischen Knoten statt, die sich an der i-ten Stelle unterscheiden.

Anschaulich: Wenn sich zwei Knoten an der i-ten Stelle ($i \neq 0$) unterscheiden, dann werden sie über k-i Shufflekanten geshiftet (k ist die Stellenanzahl). Dann ist der Unterschied ganz rechts (Stelle 0). Somit sind sie dann automatisch Nachbarn auf Tauschkanten!

Bei Ascend wird jeweils erst die Operation über die Exchangekanten ausgeführt und danach schicken alle Prozessoren ihre Daten „links“ herum über Shufflekanten. Bei Descend erst „rechts“ herum und dann Operation. Durch dieses Rotieren

der Infos stehen sich immer die Knotendaten in Exchangekanten gegenüber, die sich in der gewünschten Dimension ihrer Nummer unterscheiden.

8.2 Descend auf DB(k)

Was ist DB(k)?

DeBruijn Netzwerk, das dem SE(K) sehr ähnlich ist. Es besitzt dieselben Shufflekanten und anstatt der Exchangekanten sog. Shuffle-Exchange-Kanten. Also folgendes:

- Shuffle: $a\alpha \rightarrow \alpha a$, mit $\alpha \in \{0,1\}^{k-1}$ und $a \in \{0,1\}$
- Shuffle-Exchange $a\alpha \rightarrow \alpha \bar{a}$, mit $\alpha \in \{0,1\}^{k-1}$ und $a \in \{0,1\}$

Es besitzt neben den vom SE bekannten Shufflekreisen noch Shuffle-Exchange-Kreise.

>>Routing: Ähnlich dem Routing bei SE(k). Man checkt die k Stellen der Quelle, indem immer die vorderste mit dem i-ten Bit des Zieles verglichen wird. Bei Abweichung macht man den Shuffle-Exchange-Übergang, sonst den Shuffle. Damit wird der String k-mal (SE: k-1 mal) geshiftet und die Reihenfolge ist wie am Start.

Das Routing im DB ist also etwas intuitiver als beim SE, weil „komplett“ (k-mal) rotiert wird.

Wie geht Ascend/Descend auf DB(k)?

Genau wie auf dem SE(k) –siehe oben–, weil das SE(k) Teilgraph des *ungerichteten* DB(k) ist, d.h. es kann in das DB eingebettet werden.

„Teilgraph“ bedeutet, daß die Knoten nur andere Bezeichnungen haben und evtl. zusätzliche Knoten oder Kanten vorkommen. In diesem Fall ist die Knotenanzahl gleich, es existieren „überflüssige“ Kanten im DB und die Knoten haben andere „Namen“. Die Topologie des SE kommt jedoch 1:1 im DB vor.

Einbettung SE(k) in DB(k)

Seien s die Shuffle- und q die Shuffleexchangefunktion des DB-Netzes. Es gilt:

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

Dabei ist u(i) der Knoten, der sich von u an der i-ten Stelle unterscheidet. Die Einbettungsfunktion lautet dann:

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

Man sieht leicht, daß f injektiv ist. Es muß also nur noch gezeigt werden, daß jede in SE(k) auf eine Kante des DB(k) abgebildet wird. Siehe dazu Skript Seite 105. Im Prinzip geht das so: Fallunterscheidung nach Exchange- und Shufflekanten im SE(k) und Unterfallunterscheidung, ob u gerade Anzahl Einsen hat. Dann Durchrechnen von f bis man einen Wert kriegt, dem man ansieht, daß er eine Kante in DB(k) ist. Dazu wird der Punkt 2 von oben zur Umformung verwendet.

Im Skript ist da ein Fehler: Die Exchangekanten werden auf ungerichtete Kanten abgebildet, weil im „geraden“ Fall umgedrehte Kanten herauskommen. Die anderen drei Fälle liefern original ungerichtete Kanten des DB(k). Im Skript wird fälschlicherweise behauptet, die Shufflekanten würden ungerichtete DB-Kanten benötigen und die Exchangekanten die Originale nur. Richtig ist es natürlich andersrum, wie man leicht an einem Beispiel sieht.

8.3 Ascend/Descend auf linearem Array

Der Trick besteht auch hier wieder darin, daß im Endeffekt immer nur über dieselben Kanten die parallele Operation stattfindet. Die Arbeit ist, die nötigen Informationen dort geschickt hinzubringen. Die „Operationskanten“ des Array sind die erste, dritte, fünfte,…, $2^n - 1$ te. Insgesamt sind es also bei 2^n Knoten 2^{n-1} Kanten, da immer aufeinanderfolgende Paare gebildet werden.

Zum Routen/Rotieren der Zahlen zu den richtigen Knoten überlegt man sich:

0. Auf Dimension 0 kann alles so bleiben. Die Knoten, die sich in der letzten Stelle unterscheiden, stehen per Definition schon nebeneinander und führen die gewünschte Operation durch.
1. Auf Dimension 1 müssen nun die Knoteninfos, die sich in der zweitletzten Stelle unterscheiden, zu Knoten transferiert werden, die sich in der letzten Stelle unterscheiden, da nur diese die 2^{n-1} parallelen Operation

durchführen können.
Wie bekommt man das hin? Das zweitletzte Bit muß an die letzte Stelle. Also steckt man das letzte ganz nach links, wodurch die anderen „rutschen“.

2. Dimension 2: Nimm das unterste Bit und stecke es hinter das erste von links.
3. …hinter das zweite von links
4. …hinter das dritte von links

Einfaches Shiften hätte auch genügt, besitzt aber den Nachteil, daß viel mehr Kommunikation nötig wird und –s.u.– die Laufzeit sabotiert. Das obige Verfahren wird

>>Unshuffle-Operation: $i_{k-1} \quad i_{k-l} i_{k-l-1} \quad i_0 \rightarrow i_{k-1} \quad i_{k-l} i_0 i_{k-l-1} \quad i_1$

genannt. Dies ist z.B. die $\text{Unshuffle}_l(i)$-Funktion: Stecke das unterste Bit an die l-te Stelle!

Der Ascendlauf sieht dann so aus: Mach die gewünschte Operation auf Dimension 0 parallel, mache ein $\text{Unshuffle}_{\text{dimension}}(i)$ für alle Knoten „i“. Gehe dann eine Dimension höher.

Mit dem Unshuffle ist gewährleistet, daß sich immer die Knoten der aktuellen Dimension im untersten Bit unterscheiden.

Die Laufzeit wird durch die Zeit bestimmt, die Unshuffle zum Routen braucht. Das ist in O(n) möglich. Geht, weil das Routen von Permutationen in Zeit 2(n-1) und Puffergröße 3 durchgeführt werden kann:

1. Alle routen, die nach links müssen (Pipelining, n-1 Schritte max.),
2. Alle routen, die nach rechts müssen (Pipelining, n-1 Schritte max.),
3. Max. eigener Wert, zu speichernder Wert, aktuell gerouteter Wert zu speichern!

Da Unshuffle die obersten dim Bits in Ruhe läßt ist der Abstand zwischen Ziel und Quelle jeweils höchstens $2^{k-\dim}$.

Gesamtlaufzeit: $O\left(2 \sum_{\dim=0}^{k-1} (2^{k-\dim} - 1)\right) = O(2^k) = O(n)$.

8.4 Ascend/Descend auf CCC(k)

Was ist CCC(k)?

Cube-Connected-Cycles Netz der Dimension k. Man gewinnt es aus dem Q(k), indem man jeden Knoten durch einen Kreis mit k Knoten ersetzt. Jeder neue Knoten hat eine Verbindung „nach außen“, wenn man es vom ursprünglichen Q(k) aus betrachtet. Diese Verbindung findet sich also auch im Q(k). Das CCC hat den Vorteil, daß es immer den konstanten Knotengrad 3 hat. 2 Kanten für den Kreis und 1 Kante als Hypercubekante bei jedem Knoten. Es hat $2^k \ k = 2^{k+1}$ Knoten.

Die Knoten bekommen einen Doppelindex, bei dem der erste Teil einen Knoten im Kreis bezeichnet (0,…,k-1) und der zweite Teil den „Kreis“ des Hypercube $(0, \ , 2^k - 1)$. Also (integer, binär).

>>Routing: Laufe wie vom Hypercube bekannt die „Hypercubekanten“ entlang. Dazu ist es nötig, in jedem „Knoten-Kreis“ diesen Kreis bis zum richtigen Ausgangskantenknoten zu durchlaufen. Angekommen im richtigen Kreis, flitzt man den Kreis entlang bis man da ist. Alles völlig intuitiv!

Wie geht Ascend/Descend auf CCC(k)?

>>Einbettung des Q(k+logk) in den CCC(k):[3] Man bildet den Knoten uv des Hypercube auf den Knoten (i,u) des CCC ab, wobei die Binärdarstellung von i dem v entspricht. Wie simuliert man den Ascendlauf CCC:

1. Phase: Ascend über die Dimensionen 0,…,logk-1
   Diese Dimensionen entsprechen den Kreisen im CCC. Der Ascendlauf des Hypercube wird hier durch den Ascendlauf auf dem Kreis als Array – s.o.– simuliert. Zeit O(k).
2. Phase: Ascend über die Dimensionen logk,…,logk+k-1
   Kommunikation über eine Kante dieser Dimension entspricht im CCC Kommunikation zwischen Knoten, die auf benachbarten Kreisen liegen. Benachbart, weil die zweite Koordinate wie beim Hypercube definiert ist. Hier muß Knoten i des Kreises x mit dem Knoten i des Kreises y kommunizieren. X und y unterscheiden sich dabei in genau einem Bit.
   Man dreht also die Knoten der Kreise so, daß sie über die geeigneten Kanten kommunizieren können. Zeit O(k).
   Knoten x,0 hat die 0-Dimension-Hypercubekante. Er macht seinen

3. Macht nur Sinn, wenn k Zweierpotenz ist

Ascendlauf und die anderen auf dem Kreis „rutschen“ ihm hinterher. Da der Kreis k Knoten hat, kommt x,1 nach k-1 Rutschern bei Dimension (Hypercubedimension!) 0 an und beginnt auch seinen Ascendlauf, der nach weiteren k-1 Schritten beendet ist. Zeit O(k)

Bemerkenswert ist im zweiten Schritt, daß der Hypercubeascendlauf nicht in der richtigen Reihenfolge stattfindet. Die Knoten auf einem Ring kommunizieren jeweils in verschiedenen Dimensionen, abhängig von ihrer momentanen Position, auf die sie gedreht wurden. D.h. es findet immer in allen diesen Dimensionen eine Kommunikation statt – gleichzeitig.

8.5 Ascend auf Butterfly (BF)

Was ist Butterfly?

Es hat bei Dimension k die Knoten (i,u); $0 \leq i < k$, u ist k-stellige Binärzahl. Es gibt wie beim CCC Kreiskanten, die das „u“ beibehalten. Und es gibt „Kreuzkanten“, die i um 1 modulo k erhöhen (wie Kreiskante) und u an der Stelle i invertieren.

Wie geht Ascend/Descend auf BF?

Das CCC(k) ist Teilgraph des BF(k), d.h. es läßt sich mit Kantenstreckung 1 und Kantenauslastung 1 einbetten in BF.

Einbettungsfunktion: $((i, u) \rightarrow ((i + g(u)) \bmod k, u))$

mit $g(u) = \begin{cases} 0, \text{falls die Anzahl der Einsen in u gerade} \\ 1, \text{sonst} \end{cases}$. Siehe dazu Seite 101 im Skript.

Damit läßt sich jeder Ascendlauf des CCC auf dem BF simulieren. CCC simuliert seinerseits einen Hypercubelauf s.o.

9. Markierungsspiel

Wenn A ein arithmetischer Ausdruck (nur: +,-,*,/) ist, der durch einen Binärbaum – jeder Knoten hat genau 0 oder 2 Söhne– gegeben ist, der n numerische Konstanten hat, dann kann A in Zeit O(logn) mit O(n) Prozessoren auf einer CREW-PRAM ausgewertet werden.

Jeder Knoten bekommt einen Prozessor, der logn-mal die folgenden Anweisungen bearbeitet:

0. Initialisierung –wird nur einmal ausgeführt!–
   Setze x (cond) für diesen Knoten auf sich selbst.
   Setze „ausgerechnet“ und „Wert“, wenn Knoten Blatt ist.
   Setze Funktion auf „Identität“ für innere Knoten.
1. Aktiviere
   Wenn Knoten nicht aktiv ist –x (cond) zeigt auf ihn selbst– dann: Wenn einer der beiden Söhne berechnet ist, bilde die neue Funktion „berechneter Sohn“ op x (oder umgekehrt) im Knoten und nimm den zweiten als neues x. Der rechte wird als cond x bevorzugt.
2. Komprimiere (2mal ausführen!)
   Wenn x nicht berechnet ist, dann bilde die neue Funktion „Funktion(Funktion von x)“ –Einsetzen des untern Knotens in „x“ des oberen– und setze x auf x von x –also den cond-Zeiger auf cond(cond)–
3. Markiere
   Wenn x berechnet ist, dann rechne den Knoten aus und setze ihn auf „ausgerechnet“.

Aktiviert werden dabei nur Knoten, die also mindestens einen berechneten –direkten– Sohn haben. Damit nehmen inaktive auch nicht an der Komprimierung teil.

Auch bei diesem Alg. läßt sich die Anzahl der Prozessoren wieder auf $O(\frac{n}{\log n})$ senken. Desweiteren ist es möglich, den Ausdrucksbaum mit nur $O(\frac{n}{\log n})$ Prozessoren in Zeit $O(\log n)$ auf einer CREW-PRAM zu konstruieren nach Buch. Siehe Skript Seite 42.

9.1 Anwendung: Wieviele Blätter hat der Baum?

Nimm einen arithmetischen Ausdruck, dessen Blätter alle eine 1 enthalten und dessen Knoten + als Operator verwenden.

9.2 Anwendung: Wieviele Knoten hat ein Teilbaum mit Wurzel v?

Wie oben, nur addiere für jeden Operator zusätzlich 1 dazu. Also als Operation ein „addiere-inkrementiere“.

10.Zusammenhangskomponenten

Eine weitere Anwendung für die Verdopplungstechnik, die auch beim Listranking eingesetzt wird.

Man speichert jede Zusammenhangskomponente in einem sog. Wurzelstern: Jeder Knoten einer ZHK zeigt auf den kleinsten Knoten seiner ZHK.

Zu Beginn bildet jeder Knoten einen eigenen Wurzelstern der Größe 1. Nach r Runden besteht jeder Wurzelstern aus mindestens 2^r Knoten oder er ist schon eine fertige ZHK, wo eh nichts mehr dazukommen kann.

Bei jeder Iteration werden mindestens zwei Wurzelsterne miteinander verschmolzen, wobei die kleinere Wurzelnummer die neue gemeinsame Wurzel bildet. Die „unterlegene“ Wurzel (oder mehrere) zeigen nicht mehr auf sich selbst, sondern auf die neue Wurzel. Durch Pfadkomprimierung erhält man den neuen Stern.

Um das zu erreichen guckt sich jeder Knoten an, welcher Nachbar von ihm die kleinste Wurzel hat. Diese merkt er sich. Danach sucht sich die Wurzel den kleinsten gemerkten Nachbarstern seiner „Kinder“. Damit weiß jede Wurzel, an welche Wurzel sie sich hängen muß: Die kleinste Wurzel, die die Nachbarn all ihrer Kinder haben.

Nach log n Runden (n Anzahl der Knoten) ist man fertig. Eine EREW braucht zur Minimumsbestimmung (wie ODER) nur n/log n Prozessoren und log n Zeit. Da aber in n Knoten das Min gleichzeitig bestimmt wird, benötigt man eine CREW und insgesamt $\frac{n^2}{\log n}$ Prozessoren und log Runden mit Min ergibt $\log^2 n$ Zeit.