Parallele Algorithmen

Prof. Dr. Hans-Dietrich Hecker

SS 2005 / 2007, WS 2010

Vorwort

Dieses Skript ist im Rahmen des Projekts „Vorlesungsskripte der Fakultät für Mathematik und Informatik“ entstanden und wird im Rahmen dieses Projekts weiter betreut. Das Skript ist nach bestem Wissen und Gewissen entstanden. Dennoch garantiert weder der auf der Titelseite genannte Dozent, noch die Mitglieder des Projekts für dessen Fehlerfreiheit. Für etwaige Fehler und dessen Folgen wird von keiner der genannten Personen eine Haftung übernommen. Es steht jeder Person frei, dieses Skript zu lesen, zu verändern oder auf anderen Medien verfügbar zu machen, solange die Adresse der Internetseiten des Projekts `http://www.minet.uni-jena.de/~joergs/skripte/` *genannt wird.*

Diese Ausgabe trägt die Versionsnummer 3237 und ist vom 7. Februar 2011. Eine neue Ausgabe könnte auf der Webseite des Projekts verfügbar sein.

Jeder ist aufgerufen, Verbesserungen, Erweiterungen und Fehlerkorrekturen für das Skript einzureichen bzw. zu melden oder selbst einzupflegen – einfach eine E-Mail an die Mailingliste `<uni-skripte@lug-jena.de>` *senden. Weitere Informationen sind unter der oben genannten Internetadresse des Projekts verfügbar.*

Hiermit möchten wir allen Personen, die an diesem Skript mitgewirkt haben, vielmals danken:

- *Jörg Sommer* `<joerg@alea.gnuu.de>` *(2004, 2005, 2007)*
- *Fred Thiele (2005)*
- *Christian Raschka (2005)*
- *Michael Preiss (2007)*
- *Jörg Bader* `<joerg.bader@wurzel.org>` *(2011)*

Inhaltsverzeichnis

1 Einleitung **8**
- 1.1 Modelle für parallele Algorithmen . . . 8
  - 1.1.1 DAG – Modell eines gerichteten, kreisfreien Graphen . . . 8
  - 1.1.2 Netzwerkmodell . . . 10
  - 1.1.3 Synchrones Shared-Memory-Modell . . . 10

2 Die Güte von parallelen Algorithmen und das Theorem von Brent **17**

3 Die sieben Paradigmen zum Entwurf paralleler Algorithmen **21**
- 3.1 Binärbaumparadigma . . . 21
  - 3.1.1 Rekursiver Ansatz . . . 21
  - 3.1.2 Nichtrekursiver Ansatz . . . 22
- 3.2 Pointer Jumping . . . 24
  - 3.2.1 Parallel Prefix . . . 25
- 3.3 Teile und Herrsche . . . 26
- 3.4 Partitioning . . . 30
- 3.5 Pipeline-Verfahren . . . 33
- 3.6 Accelarated Cascading . . . 34
- 3.7 Aufbrechen von Symmetrien . . . 37

4 Listen und Bäume **39**
- 4.1 List-Ranking . . . 39
- 4.2 optimales Listranking . . . 39
  - 4.2.1 Pointer Jumping . . . 39
  - 4.2.2 Optimales Listranking . . . 41
- 4.3 Eulertour-Technik . . . 42
- 4.4 Baumkontraktion . . . 43
- 4.5 Das LCA-Problem . . . 45
- 4.6 Das Range-Minima-Problem . . . 46

5 Suchen, Mischen, Sortieren **49**
- 5.1 Suchen in einer sortierten Menge nach Kruskal . . . 49
- 5.2 Mischen . . . 50
- 5.3 Sortieren . . . 55
  - 5.3.1 Merge with the help of a cover . . . 55
  - 5.3.2 Optimales Sortieren - Pipeline Merge Sort . . . 56
- 5.4 Selektion . . . 57

6 Parallelisierbarkeit **60**
6.1 P-vollständige Probleme 61
6.1.1 Maximaler Fluss in einem Netzwerk 61
6.1.2 Lineare Optimierungsprobleme 61
6.1.3 Das circuit value problem 62
6.2 P-Vollständigkeit von CVP 62
6.3 DFS-Theorem (geordnete Tiefensuche) 63

Auflistung der Theoreme

Sätze

Satz 5 Satz von Brent 19
Satz 6 Satz von Eckstein 20
Satz 18 Satz von Kruskal (1984) 50
Satz 24 Hauptsatz über die Reduktion 64
Satz 25 Satz über Transitivität 64

Definitionen und Festlegungen

Definition 4 Speed-Up 17
Definition 5 Effizienz 18

Literaturverzeichnis

[1] Joseph Ja'Ja: Introduction to parallel algorithms

[2] AKL: Introduction to parallel algorithms

[3] Gibbon and Rytter: parallel algorithms

[4] Andreas Goerdt: „Skript zur Vorlesung Parallele Algorithmen“, Technische Universität Chemnitz, 1994, http://www.tu-chemnitz.de/informatik/TI/paralg_ws20022003/skript.ps.gz, Vorsicht: Es sind einige Fehler im Skript!

[5] Markus Krebs, Andreas Horstmann: „Zusammenfassung der Vorlesung ‚Parallele Algorithmen‘ von Prof. Dr. Amitava Datta“, 2002, http://www.informatixx.de/privat/files/informatik/pdf/Parallele_Algorithmen_Komplett.pdf, Vorsicht: Es sind einige Fehler im Skript!

1 Einleitung

Die Paradigmen (Strategien) für den Entwurf serieller Algorithmen werden durch die Kriterien Speicher- (SPACE) und Zeitkomplexität (TIME), sowie ihrer einfachen Formulierbarkeit (Beschreibungskomplexität) bestimmt. Ein serieller Algorithmus soll bezüglich dieser Kriterien „gut“ sein.

Für die Bewertung serieller Algorithmen ist ein allgemein anerkanntes Modell die random access machine (**RAM**). Die RAM verfügt über eine Recheneinheit und einen beliebig großen Speicher, auf dessen Inhalt wahlfrei zugegriffen werden kann. Alle Basisoperationen können in $O(1)$ TIME berechnet werden.

Für parallele Algorithmen hingegen gibt es verschiedene Modelle, die wesentlich abhängiger vom konkreten Rechner sind. Einige (spezielle) Kriterien für Modelle sind:

- Geringe Anzahl leistungsfähiger Prozessoren
- Hohe Anzahl „einfacher“ Prozessoren
- Verteilte Systeme (nicht Gegenstand der Vorlesung, siehe Rechnerarchitektur)

Bei der Betrachtung von parallelen Algorithmen treten aber auch neue Fragen auf, die sich für den seriellen Fall nicht ergeben:

- Computational Concurrency: Wie stark lässt sich eine Problem parallelisieren? Bringt der Einsatz weiterer Prozessoren eine weitere Beschleunigung?
- Processor Allocation: Welcher Prozessor soll wann welche Operation ausführen?
- Scheduling: Welche Operationen können parallel ausgeführt werden? Welche Operationen hängen von Ergebnissen anderer Operationen ab?
- Communication: Wie kommen die Ergebnisse von einem Prozessor zu den anderen Prozessoren?
- Synchronization: Arbeiten alle Prozessoren unabhängig voneinander oder arbeiten sie alle im gleichen Rythmus? Wie synchronisieren sie die Prozessoren?

1.1 Modelle für parallele Algorithmen

1.1.1 DAG – Modell eines gerichteten, kreisfreien Graphen

Mit Hilfe eines gerichteten, kreisfreien Graphen $DAG = (E, V)$ (directed acyclic graph; **DAG**) beschreibt man eine Berechnung, indem man jedem inneren Knoten $v \in V$ einen Berechnungsschritt

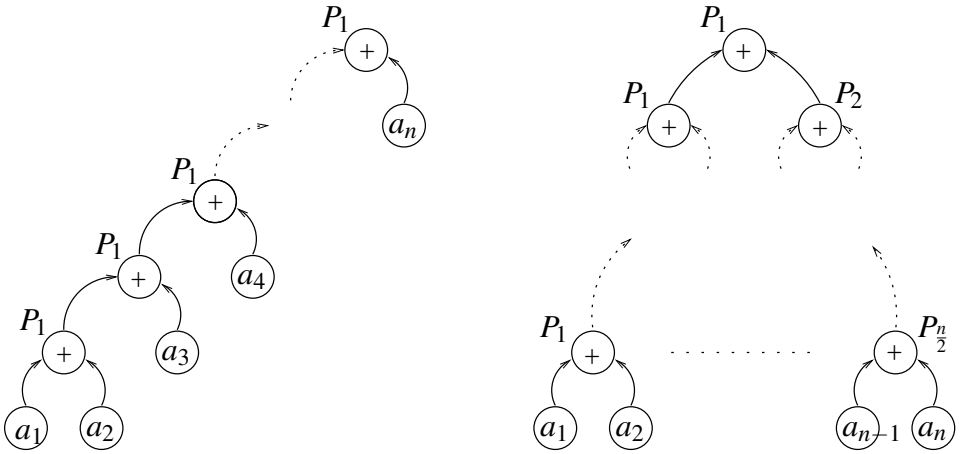


Abbildung 1.1: Zwei mögliche DAGs zur Berechnung der Summe von n Zahlen

zuweist und über die Kanten $e \in E$ die Abhängigkeiten der Operanten der Berechnungsschritte untereinander modelliert. Eine Kante e von v_1 nach v_2 besagt, dass die Ausgabe von v_1 als Eingabe von v_2 verwendet wird. Einen Knoten mit Indegree null bezeichnet man als Eingabeknoten der Berechnung, einen Knoten mit Outdegree null als einen Ausgabeknoten. Alle anderen Knoten bezeichnet man als innere Knoten.

In Abbildung 1.1 sind zwei verschiedene Graphen für die parallele Berechnung der Summe von n Zahlen dargestellt, wobei eine der beiden Darstellungen keine Parallelität aufweist.

Definition 1
Gegeben seien p Prozessoren und ein gerichter, kreisfreier Graph $DAG = (E,V)$, der eine Berechnung beschreibt. Jedem inneren Knoten $v \in V$ ordnet man ein Paar (j_v,t_v) zu, wobei t_v einen Zeitpunkt bestimmt, an dem der Prozessor j_v ($1 \leq j_v \leq p$) diesen Knoten berechnet. Zudem gelten folgende Bedingungen:

1. ein Prozessor kann zu einem Zeitpunkt nur an einem Knoten aktiv sein; wenn $t_u = t_v$ für $u,v \in V$ und $u \neq v$, so gilt: $j_u \neq j_v$

2. der Prozessor kann für einen Knoten v erst dann aktiv werden, wenn die Prozessoren der Vorgängerknoten fertig sind; wenn $(u,v) \in V^2$ eine gerichtete Kante (von u nach v) ist, so ist $t_v \geq t_u + 1$.

Eine solche Zuordnung $S\colon V \to (\{1,\ldots,p\} \times \mathbb{N})$ bezeichnet man als **Ablaufplan** oder **Schedule**.

Als die Zeit T eines Ablaufplans S bezeichnet man $T(S) = \max\{t\colon \exists v \in V\colon S(v) = (j_v,t_v)\}$.

Man erkennt an dem DAG leicht, welche Berechnungen voneinander abhängen und welche unabhängig voneinander sind, also parallel zueinander ausgeführt werden können. Jedoch wird nicht auf die Speicherung der Daten und die Kommunikation der Prozessoren untereinander eingegangen, weswegen das Modell des gerichteten, kreisfreien Graphen für unsere Bedürfnisse zu allgemein ist.

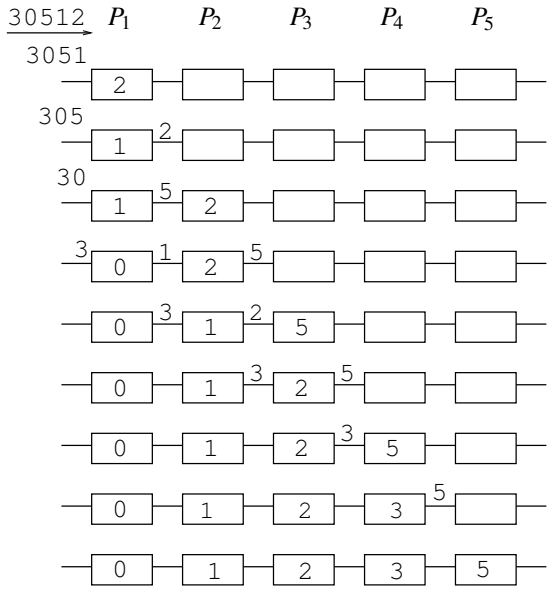


Abbildung 1.2: Sortierung der Zahlen 2, 1, 5, 0 und 3 in einem linearen Feld von Prozessoren, ein sogenanntes Sortiernetzwerk

1.1.2 Netzwerkmodell

Das **Netzwerkmodell** beschreibt durch einem ungerichteten Graphen eine Menge von Prozessoren (= Knoten) und deren Vernetzung (= Kanten) untereinander. Die Prozessoren verfügen über einen *lokalen* Speicher und tauschen ihre Daten mit den Operationen **receive** und **send** aus.

Mögliche Arten der Vernetzung, sogenannte Topologien, sind:

- ein linearen Feld mit p Prozessoren, in dem jeder innere Knoten genau zwei Nachbarn hat. Dabei können noch der erste und der letzte Knoten miteinander verbunden werden, so dass ein Ring entsteht. Abbildung 1.2
- ein zweidimensionales Gitter, in dem jeder innere Knoten genau vier Nachbarn hat.
- oder ein d-dimensionaler Würfel (Hypercube) mit 2^d Prozessoren, in dem die Nachbarschaftsverhältnisse über binäre Zahlen beschrieben werden: Zwei Prozessoren sind benachbart, wenn sich die Binärdarstellungen ihre Prozessornummern nur in genau einer Stelle unterscheiden.

Für unsere Bedürfnisse ist das Netzwerkmodell zu speziell, da für jede Problemstelle die jeweilige Topologie betrachtet werden muss.

1.1.3 Synchrones Shared-Memory-Modell

Eine direkte Erweiterung der random access machine ist die parallel random access machine (**PRAM**), bei der mehrere RAMs über einen unbegrenzt großen, gemeinsamen Speicher (**globaler**

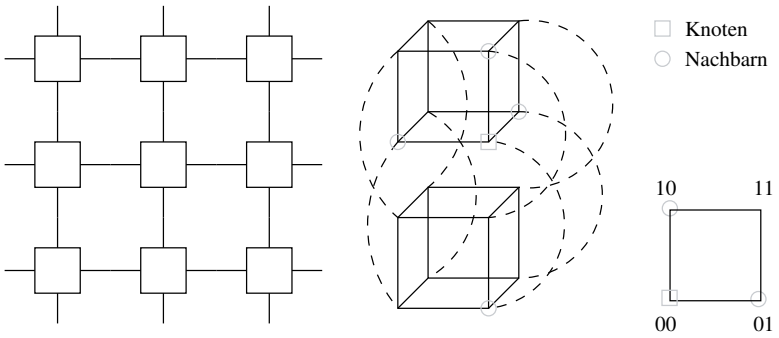


Abbildung 1.3: Ein zweidimensionales Gitter und ein Hypercube für $d = 4$ und $d = 2$ mit der Kennzeichnung der Nachbarschaften

todo: Das Bild trennen in zwei Grafiken. Das Gitter hat nicht direkt etwas mit dem Hypercube zu tun.

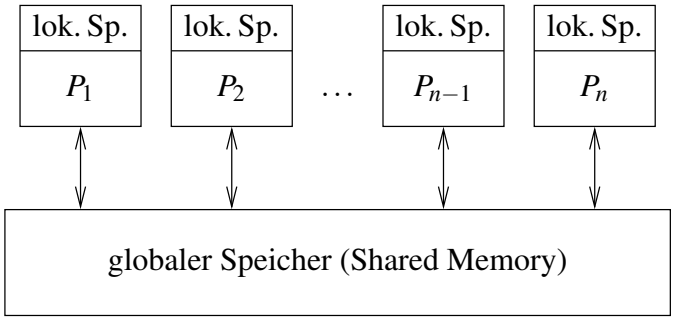


Abbildung 1.4: Modell der parallel random access machine

Speicher oder **shared memory**) verfügen, den sie unabhängig voneinander in einer Zeiteinheit lesen und verändern können. Dabei wird in unserem abstrakten Modell die Zugriffszeit auf den globalen Speicher vernachlässigt. Ein Forscherteam in Ontario, Kanada, hat für spezielle Probleme und Algorithmen die Zugriffszeit mitberücksichtigt und keine wesentlich anderen Ergebisse erhalten. Untereinander sind die Prozessoren nicht verbunden, so dass jegliche Kommunikation über den globalen Speicher erfolgen muss. Die Prozessoren arbeiten mit einem gemeinsamen Zeitsignal (**synchron**), d. h. jede Anweisung wird von allen Prozessoren zum gleichen Zeitpunkt ausgeführt. Abbildung 1.4

todo: einarbeiten: PRAM berücksichtigt Zuordnung der Aufgaben an die Prozessoren und Kommunikation der Prozessoren untereinander.

Der Zugriff auf den globalen Speicher erfolgt mit den Operationen **globalRead**(A,x) zum Lesen der globalen Speicherzelle A in die lokale Speicherzelle x und **globalWrite**(x,A) zum Schreiben der lokalen Speicherzelle x in die globale Speicherzelle A. Zur Vereinfachung der Schreibweise verwenden wir große Buchstaben für globale Speicherzellen und kleine Buchstaben für lokale Speicherzellen, so dass man für einen globalRead $x := A$ und für einen globalWrite $A := x$ schreiben kann.

Beim simultanen Zugriff auf den globalen Speicher kann es zu Konflikten kommen, wenn mehrere Prozessoren auf die gleiche Speicherzelle zugreifen. Man unterscheidet die verschiedenen PRAMs anhand der Art, welche Zugriffe sie zulassen:

- **EREW** (exclusive read and exclusive write): eine Speicherzelle kann zu einem Zeitpunkt nur von einem Prozessor gelesen oder beschrieben werden.
- **CREW** (concurrent read and exclusive write): eine Speicherzelle kann zu einem Zeilpunkt von beliebig vielen Prozessoren gelesen, aber nur von einem Prozessor beschrieben werden.
- **common CRCW** (concurrent read and concurrent write): eine Speicherzelle kann zu einem Zeitpunkt von beliebig vielen Prozessoren gelesen und beschrieben werden. Beim Schreibzugriff müssen alle Prozessoren den gleichen Wert schreiben.
- **arbitrary CRCW**: eine Speicherzelle kann zu einem Zeitpunkt von beliebig vielen Prozessoren gelesen und beschrieben werden. Wenn die Prozessoren unterschiedliche Werte schreiben, kann der tatsächlich geschriebene Wert einer der Werte oder die Summe/Minumim/Maximum der Werte sein.
- **priority CRCW**: eine Speicherzelle kann zu einem Zeitpunkt von beliebig vielen Prozessoren gelesen und beschrieben werden. Welcher Wert in die Speicherzelle tatsächlich geschrieben wird, hängt von einer vorgegebenen Priorisierung der Prozessoren ab; z. B. könnte der Wert des Prozessors mit der kleinsten Prozessornummer geschrieben werden.

Beispiel 1 (Matrixmultiplikation)
Um die Multiplikation einer $n \times n$-Matrix A mit einem n-dimensionalen Spaltenvektor x parallel auf p Prozessoren auszuführen, muss man die Berechnung in p unabhängige Probleme zerlegen. Dazu eignet sich die Zerlegung des Ergebnisvektors y, da die Einträge unabhängig von den anderen Einträgen in y bestimmt werden können. Jeder Prozessor übernimmt $r = \frac{n}{p}$ Zeilen

(o. B. d. A. ist n ein Vielfaches von p) von y

$$\begin{pmatrix} y_1 \\ \vdots \\ y_r \\ \hline y_{r+1} \\ \vdots \\ y_n \end{pmatrix} = \left(\begin{array}{ccc} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{r1} & \dots & a_{rn} \\ \hline a_{(r+1)1} & \dots & a_{(r+1)n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nn} \end{array}\right) \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_r \\ \\ x_{r+1} \\ \vdots \\ x_n \end{pmatrix}$$

Input : $A_{(n,n)}, \vec{x}$, Prozessorzahl i, p Anzahl der Prozessoren ($r := \frac{n}{p}$)
Output : Die Komponenten $[(i-1)\,r+1,\dots,i\,r]$ von $y = A \cdot x$

```
1 globalread(x, z)            /* Alle Proz. lesen x zur gleichen Zeit */
2 globalread(A[(i - 1) · r + 1 : ir , 1 : n], B)
3 Berechne w := B · z                          /* Arbeit der Prozessoren */
4 globalwrite(w, y[(i - 1) · r + 1 : i · r])
```

Algorithmus 1.1: Das Programm für den i. Prozessor zur Berechnung von $A \cdot x$

Bemerkung 1

- In der ersten Zeile lesen allen Prozessoren zur selben Zeit aus dem selben Speicherbereich. Die PRAM muss also concurrent read (CR) unterstützen.
- In den vierten Zeile schreiben alle Prozessoren in unterschiedliche Speicherbereiche. Die PRAM muss also nur exclusive write (EW) unterstützen.
- Da kein Prozessor die Ergebnisse eines anderen benötigt, also kein Datenaustausch untereinander statt findet, ist keine Synchronisation nötig.
- Alternativ könnte man die Spalten von A zu Blöcken zusammenzufassen, diese mit den entsprechenden Zeilen von x multiplizieren und danach die Summe bilden. Allerdings ist dann eine Synchronisation notwendig!

Entsprechend der obigen Vereinbarung zur Vereinfachung der Schreibweise lässt sich der Algorithmus 1.1 auch als Algorithmus 1.2 schreiben. Da die Ein- und Ausgabe nur jeweils

Input : $A_{(n,n)}, \vec{x}$, Prozessorzahl i, p Anzahl der Prozessoren ($r := \frac{n}{p}$)
Output : Die Komponenten $[(i-1)\,r+1,\dots,i\,r]$ von $y = A \cdot x$

```
1 z := X;
2 b := A[(i-1) r + 1 : ir , 1 : n];
3 w := b + z;
4 Y[(i - 1) · r + 1 : i · r] := w;
```

Algorithmus 1.2: Vereinfachte Darstellung von Algorithmus 1.1

$O(1)$ Schritte dauert, kann sie ignoriert werden, um die Schreibweise noch weiter zu verkürzen:

$$Y[(i-1)\cdot r+1 : i\cdot r] := A[(i-1)r+1 : ir, 1 : n]\cdot X$$

Man kann die Beschreibung der Algorithmen in zwei unterschiedlichen Abstraktionsschichten betrachten: Die obere Stufe (upper level) beschreibt nur die generelle Vorgehensweise, wärend auf der unteren Stufe (lower level) die Prozessorallokation berücksichtigt wird.

Beispiel 2
Als Beispiel soll die Summation von n Zahlen betrachtet werden. Die Beschreibung auf dem hohen Abstraktionsniveau (Algorithmus 1.3) arbeitet mit der **for-pardo**-Schleife, die angibt, dass der Schleifenrumpf für jedem Wert der Laufvariablen parallel auf so viel wie nötig Prozessoren ausgeführt werden soll. Jede Anweisung stellt eine **Zeiteinheit** – oder auch **Takt** oder **Time-unit** genannt – dar.

Input : Ein Feld A mit $n = 2^k$ Zahlen
Output : Summer S der Zahlen im Feld
1 **for** *i := 1* **to** n **pardo**
2 | $B[i] := A[i]$
3 **endfor**
4 **for** *h := 1* **to** *log(n)* **do**
5 | **for** *i := 1* **to** $\frac{n}{2^h}$ **pardo**
6 | | $B[i] := B[2i-1] + B[2i]$
7 | **endfor**
8 **end**
9 $S := B[1]$;

Algorithmus 1.3: Summation von Zahlen – auf hohem Abstraktionsniveau

Auf der unteren Stufe geht es um das konkrete Programm, das auf den Prozessoren ausgeführt wird. Algorithmus 1.4 Dabei wird auf die Aufteilung der einzelnen Schritte auf die Prozessoren – **Allokation** genannt – eingegangen. Es muss z. B. auf den Fall geachtet werden, dass weniger Prozessoren als Eingabedaten vorhanden sind, einige Prozessoren also mehrere Datensätze bearbeiten müssen.

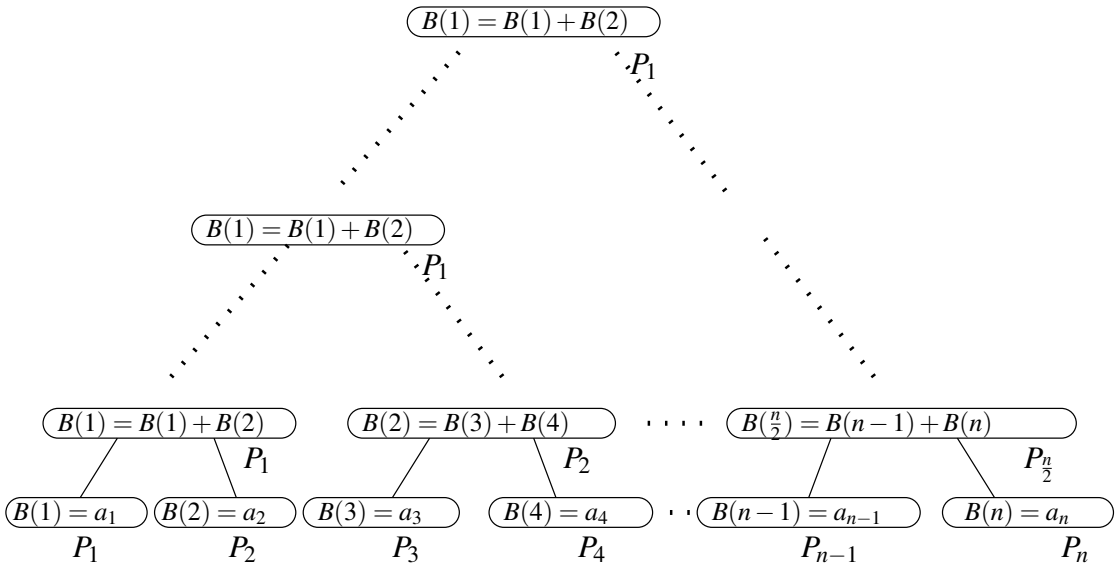
Wie der Ablauf und die Zuordnung der Prozessoren bei $n = 8$ Werten für Algorithmus 1.3 ist, ist in Abbildung 1.5 dargestellt. Stehen für Algorithmus 1.4 nur $p = 4$ Prozessoren zur Verfügung, ist die Zuordnung die in Abbildung 1.6.

Die Allokation ist technisch aufwendig, ohne dass neue Ideen gefragt sind. Deshalb werden wir uns darauf konzentrieren, die Operationen zu beschreiben, die gleichzeitig möglich sind (obere Stufe) und die Allokation über das Theorem von Brent (Satz 5) zu behandeln.

Input : Ein Feld A mit n Zahlen
Output : Die Summe S der Zahlen in A

```
1  Es sei r die Prozessornummer und l = n/p (= 2^k) die Anzahl der Einträge aus A, für die der
   Prozessor verantwortlich ist, wobei p die Anzahl der Prozessoren ist
2  for j := 1 to l do
3  |  B[l·(r−1)+j] := A[l·(r−1)+j]
4  end
5  for h := 1 to log(n) do
6  |  if 2^h ≤ l then
7  |  |  for j := 2^{−h}·l·(r−1)+1 to 2^{−h}·l·r do
8  |  |  |  B[j] := B[2·j−1]+B[2·j]
9  |  |  end
10 |  else if r·2^h ≤ n then
11 |  |  B[r] := B[2·r−1]+B[2·r]
12 |  end
13 end
14 if r = 1 then
15 |  S := B[1]
16 end
```

Algorithmus 1.4: Summation von Zahlen – auf niedrigem Abstraktionsniveau



todo: Die Grafik muss im Original mal verkleinert werden, damit die Schritt bzw. die Boxgröße mit angepasst wird.

Abbildung 1.5: todo: ausfüllen

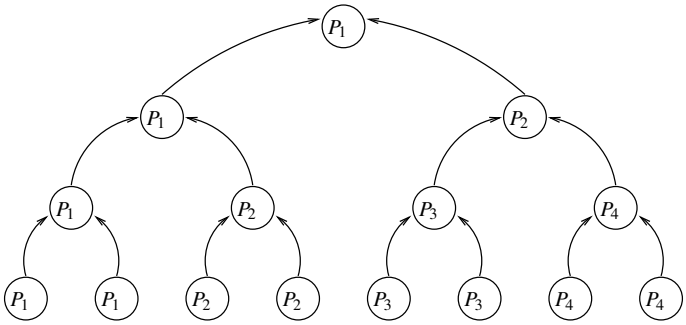


Abbildung 1.6: todo: ausfüllen

2 Die Güte von parallelen Algorithmen und das Theorem von Brent

Für die Bewertung serieller Algorithmen ist entscheident, wie viele Rechenoperationen (TIME) und wie viel Speicherplatz (SPACE) sie zur Berechnung benötigen. Bei parallelen Algorithmen muss zusätzlich noch sie Anzahl der verwendeten Prozessoren bzw. der sich daraus ergebende Aufwand betrachtet werden.

Definition 2
Für ein Problem $\mathscr{P}$ bezeichnet $T^*(n)$ die beste komplexitätstheoretische Abschätzung für die Rechenzeit einen seriellen Algorithus, der das Problem $\mathscr{P}$ löst.

Da bei parallelen Algorithmen mehrere Rechenschritte in einer Zeiteinheit ausgeführt werden – bis zu p Schritten bei p Prozessoren in einer Zeiteinheit – trifft man eine Unterscheidung bezüglich dieser beiden Größen:

Definition 3
Die **Rechenzeit** $T_p(n)$ (**TIME**) eines parallelen Algorithmus' mit p Prozessoren ist die Mindestanzahl an Zeiteinheiten, die der Algorithmus zur Lösung des Problems benötigt.

Der **Arbeitsaufwand** $W(n)$ (**WORK**) eines parallelen Algorithmus' ist die Anzahl der insgesamt ausgeführten Einzeloperationen (aller Prozessoren zusammen).

Wenn man $W_i(n)$ als die Anzahl der Operationen, die in einem Schritt durchgeführt werden, bzw. als die Anzahl der Prozessoren, die in einem Schritt aktiv sind, betrachtet, so ist der Arbeitsaufwand des gesamten Algorithmus' die Summe davon:

$$W(n) = \sum_{i=1} W_i(n)$$

Definition 4 (Speed-Up)
Für ein Problem $\mathscr{P}$ definiert man als ein Maß für die Verbesserung (**Speed-up**) eines parallelen Algorithus' gegenüber dem besten seriellen Algorithmus

$$S_p(n) := \frac{T^*(n)}{T_p(n)}$$

Satz 1
Der **Speed-up** eines parallelen Algorithmus' kann nie größer sein als die Anzahl der Prozessoren oder anders gesagt, die p-fache Zeit eines parallelen Algorithmus' kann nie kleiner sein als die Zeit des seriellen Algorithmus'.

$$S_p(n) \leq p \qquad\qquad T^*(n) \leq pT_p(n)$$

BEWEIS:
Simuliert man die Berechnungen der p Prozessoren schrittweise nacheinander auf einer Einprozessormaschine, ergibt dies einen neuen seriellen Algorithmus mit $p \cdot T_p(n)$ Schritten. Dies kann aber nicht weniger als $T^*(n)$ sein, da $T^*(n)$ bereits das serielle Optimum ist. ∎

Bemerkung 2
Wenn der Speed-up $S_p(n)$ das Maximum p erreicht, braucht ein paralleler Algorithmus den p. Teil der Rechenzeit des seriellen Algorithmus'. Insbesondere kann man mit einem parallelen Algorithmus und polynomial vielen Prozessoren kein NP-schweres Problem in Polynomialzeit lösen!

$$S_p(n) = \frac{T^*(n)}{T_p(n)} = p \quad \Rightarrow \quad T_p(n) = \frac{T^*(n)}{p}$$

Definition 5 (Effizienz)
Die **Effizienz** eines parallelen Algorithmus' beschreibt das Verhältnis von Speed-up zur Anzahl der eingesetzten Prozessoren.

$$E_p(n) := \frac{S_p(n)}{p} = \frac{T^*(n)}{p \cdot T_p(n)} \leq 1$$

Definition 6
Als die **Kosten** $C_p(n)$ eines parallelen Algorithmus' bezeichnet man das Produkt aus Anzahl der eingesetzten Prozessoren und der benötigten Rechenzeit.

$$C_p(n) := p \cdot T_p(n)$$

Satz 2
Die Kosten $C_p(n)$ eines parallelen Algorithmus' können nicht unter der seriellen Komplexität liegen.

$$T^*(n) \leq C_p(n)$$

BEWEIS:
Nach Satz 1 ist

$$S_p(n) = \frac{T^*(n)}{T_p(n)} \leq p$$

also ist $T^*(n) \leq p \cdot T_p(n) = C_p(n)$. ∎

Satz 3
Die Kosten $C_p(n)$ eines parallelen Algorithmus' sind mindestens in der Größenordnung von WORK bzw. der Arbeitsaufwand überseigt nie die Kosten: $W(n) \leq C_p(n)$.

BEWEIS:
Es seien $W_i(n)$ die Anzahl der Schritte, die der Algorithmus in Schritt i macht, d. h. $W(n) = \sum_{i=1}^{T_p(n)} W_i$. In einem Schritt können nicht mehr Operationen ausgeführt werden als Prozessoren da sind: $W_i(n) \leq p$

$$W(n) = \sum_{i=1}^{T_p(n)} W_i(n) \leq \sum_{i=1}^{T_p(n)} p = p \cdot T_p = C_p(n)$$ ∎

Satz 4
Beim Einsatz von $p = O(\frac{W(n)}{T_p(n)})$ Prozessoren liegen WORK und Kosten in der gleich Größenordnung:

$$C_p(n) = p \cdot T_p(n) = O(\frac{W(n)}{T_p(n)} \cdot T_p(n)) = O\big(W(n)\big)$$

Ein erster Ansatz für ein Gütekriterium eines Algorithmus wäre:

Definition 7
Ein paralleler Algorithmus heißt *kostenoptimal*, wenn seine Kosten der seriellen Komplexität entsprechen: $C_p(n) = T^*(n)$.

Bewerten wir mit dieser Maßgabe die Summation von n Zahlen, so zeigt sich:

- Der serielle Algorithmus mit einem Prozessor ist *optimal*: $C_1(n) = 1 \cdot T^*(n) = O(n)$.
- Der Algorithmus 1.3 mit n Prozessoren ist *nicht optimal*, denn wie man in Abbildung 1.5 sieht, ist die Laufzeit in $O(\log n)$ und damit sind die Kosten $C_p(n) = n \cdot \log n \neq O(n)$.
- Berechnet man in einem Vorschritt mit $\frac{n}{\log n}$ Prozessoren jeweils sequentiell die Summe von $(\log n)$ Zahlen (Laufzeit: $O(\log n)$) und berechnet dann mit Algorithmus 1.3 die Summe der verbleibenden $\frac{n}{\log n}$ Zahlen (Laufzeit: $O(\log \frac{n}{\log n})$ mit $\frac{n}{\log n}$ Prozessoren), ist der Algorithmus *optimal*, da die Kosten $C_{\frac{n}{\log n}}(n) = O(\frac{n}{\log n}) \cdot \big(O(\log n) + O(\log \frac{n}{\log n})\big) = O(n)$ betragen. Dieses Vorgehen bezeichnet man als Accelarated Cascading, was im Abschnitt 3.6 noch einmal genauer besprochen wird.

Die Definition ist also unzureichend. Eine bessere Definition ergibt sich mit dem Arbeitsaufwand $W(n)$ eines Algorithmus'.

Definition 8
Ein paralleler Algorithmus heißt **optimal** (WORK-optimal), wenn der Arbeitsaufwand der besten seriellen Laufzeit entspricht: $W(n) = T^*(n)$. Ein paralleler Algorithmus heißt **WT-optimal** (WORK-TIME-optimal oder **streng optimal**), wenn er optimal ist und wenn es keinen schnelleren optimalen Algorithmus gibt.

Bemerkung 3
Diese Definition wird auch relativ zu einem speziellen Modell des Maschinentyps gebraucht, z. B. WT-optimal für die common CRCW.

Satz 5 (Satz von Brent)
Ein paralleler Algorithmus, der mit p Prozessoren eine TIME von $T_p(n)$ und einen WORK von $W(n)$ hat, kann auf $p' < p$ Prozessoren in $T_{p'}(n) = T_p(n) + \left\lfloor \frac{W(n)}{p'} \right\rfloor$ ausgeführt werden.

BEWEIS:
In der $i.$ Zeiteinheit werden mit p Prozessoren $W_i(n)$ Operationen ausgeführt. Mit p' Prozessoren benötigt man zur Berechnung dieser $W_i(n)$ Operationen höchstens $t^i_{p'}(n) \leq \left\lceil \frac{W_i(n)}{p'} \right\rceil$ Zeiteinheiten.

Um die gesamte Arbeit ($W(n) = \sum_{i=1}^{T_p(n)} W_i(n)$) zu verrichten, muss man alle Schritte der p-Prozessor-PRAM nachvollziehen.

$$T_{p'}(n) = \sum_{i=1}^{T_p(n)} t^i_{p'}(n) \leq \sum_{i=1}^{T_p(n)} \left\lceil \frac{W_i(n)}{p'} \right\rceil \leq \sum_{i=1}^{T_p(n)} \left\lfloor \frac{W_i(n)}{p'} \right\rfloor + 1$$
$$\leq \left\lfloor \frac{1}{p'} \sum_{i=1}^{T_p(n)} W_i(n) \right\rfloor + T_p(n) = \left\lfloor \frac{W(n)}{p'} \right\rfloor + T_p(n) \qquad \blacksquare$$

Dieser Satz liefert die Grundlage für die allgemeine Betrachtung der Algorithmen mit einer nicht konkreten Anzahl an Prozessoren, denn jeder parallele Algorithmus kann im Normalfall auf jede Anzahl an Prozessoren angepasst werden.

Satz 6 (Satz von Eckstein)
Eine EREW-PRAM mit p Prozessoren kann eine priority CRCW-PRAM so simulieren, dass die Laufzeit nur um den Faktor $O(\log p)$ wächst.

BEWEIS:
Um das konkurierende Lesen der CRCW zu realisieren, kann ein vorbestimmter Prozessor die Zelle lesen und die Information dann baumartig verteilen – jeder Prozessor gibt die Information an zwei andere Prozessor weiter. so können alle Prozessoren in $\log p$ TIME benachrichtigt werden.

Für das konkurierende Schreiben wird der Baum von den Blättern aus abgearbeitet. Jeder Prozessor verständigt sich mit einem Nachbarn über den Wert, der geschrieben werden soll. Nach $\log p$ Schritten ist ein zu schreibender Wert ermittelt. ■

Bemerkung 4
Es gilt die folgende Ordnung der verschiedenen PRAM-Typen, wobei $X \preceq Y$ bedeutet: Algorithmen, die für X erstellt wurden, können auf X in der gleichen Zeit ausgeführt werden.

$$\text{EREW} \preceq \text{CREW} \preceq \text{common CRCW} \preceq \text{arbitrary CRCW} \preceq \text{priority CRCW}$$

3 Die sieben Paradigmen zum Entwurf paralleler Algorithmen

3.1 Binärbaumparadigma

todo: Allgemeine Idee des Binärbaumparadigmas beschreiben

Gegeben sei eine Folge $(x_i)_{i\in\mathbb{N}}$ von n Zahlen. Als **Präfixsumme** S des k. Gliedes bezeichnet man die Summe aller Glieder bis zum k. Glied. Die Präfixsumme des n. Gliedes ist die Summe aller Folgenglieder.

$$s_k = x_1 + x_2 + \cdots + x_{k-1} + x_k = \sum_{j=1}^{k} x_j$$

Wir betrachten im Folgenden sowohl den rekursiven als auch den nichtrekursiven Ansatz zur Berechnung der Präfixsummen.

3.1.1 Rekursiver Ansatz

Input : Array X der Länge $n = 2^k$
Output : Feld S der Länge n mit Partialsummen s_i

1 **if** *n=1* **then**
2 | $s_1 := x_1$ und exit
3 **for** *i:= 1* **to** $\frac{n}{2}$ **pardo**
4 | $y_i := x_{2i-1} + x_{2i}$
5 **endfor**
6 Berechne rekursiv Präfixsummen $(y_1, \ldots, y_{\frac{n}{2}})$ und speichere sie in z_1, ..., $z_{\frac{n}{2}}$
7 **for** *i := 1* **to** *n* **pardo**
8 | **if** *i gerade* **then**
9 | | $s_i := z_{\frac{i}{2}}$
10 | **else if** *i=1* **then**
11 | | $s_1 := z_1$
12 | **else**
13 | | $s_i := z_{\frac{i-1}{2}} + x_i$
14 | **end**
15 **endfor**

Algorithmus 3.1: Präfixsumme

3.1.2 Nichtrekursiver Ansatz

Input : Feld A der Länge $n = 2^r$
Output : Feld S der Länge n mit $s_k = \sum_{j=1}^{k} a_j$ – Präfixsummen

```
1  for j := 1 to n pardo
2  |  b_{0,j} := a_j
3  endfor
4  for h := 1 to log(n) do
5  |  for j := 1 to n · 2^{-h} pardo
6  |  |  b_{h,j} := b_{h-1,2·j-1} + b_{h-q,2·j}
7  |  endfor
8  end
9  for h := log(n) to 1 do
10 |  for j := 1 to n · 2^{-h} pardo
11 |  |  if j gerade then
12 |  |  |  c_{h,j} := c_{h+1,j/2}
13 |  |  else if j=1 then
14 |  |  |  c_{h,1} := b_{h,1}
15 |  |  else
16 |  |  |  c_{h,j} := c_{h+1,j-1/2} + b_{h,j}
17 |  |  end
18 |  endfor
19 end
20 for j := 1 to n pardo
21 |  if j gerade then
22 |  |  s_j := c_{1,j/2}
23 |  else if j=1 then
24 |  |  s_1 := b_{0,1}
25 |  else
26 |  |  s_j := c_{1,j-1/2} + b_{0,j}
27 |  end
28 endfor
```

Algorithmus 3.2: Präfixsumme

Der nichtrekursive Algorithmus zur Berechnung aller Präfixsummen einer Folge (Algorithmus 3.2) läuft in zwei Schritten ab (Abbildung 3.1):

1. Zuerst wird von den Blattknoten aus ein Baum erzeugt, so dass in desses inneren Knoten jeweils die Summe ihrer Söhne steht, wobei die Blattknoten die Glieder der Folge sind. Im Wurzelknoten steht dann die Summe aller Folgenglieder.

   Es sei h die Blattebene im Baum, wobei die Blätter die Höhe 0 haben und die Wurzel die Höhe $\log n$ hat, und j die Position (beginnend bei eins) des Knotens in der Blattebene von

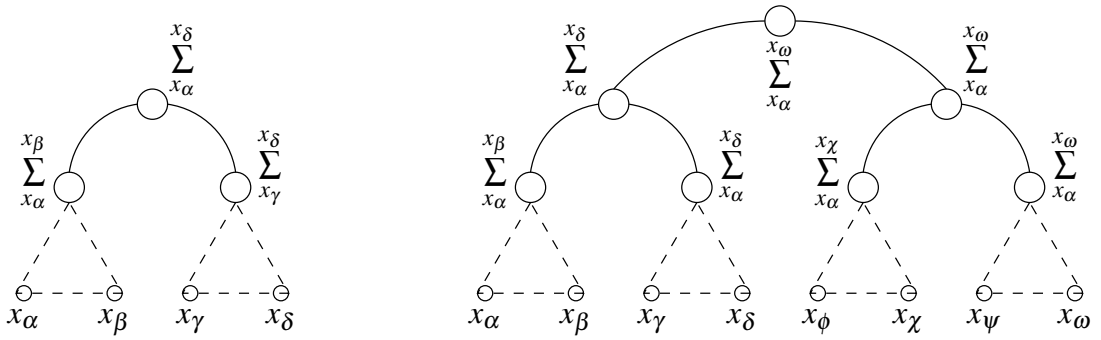


Abbildung 3.1: Die zwei Schritte des Präfixsummenalgorithmus': links die Berechnung der Teilsummen und rechts die Berechnung der Präfixsummen

links gesehen ist.

$$B_{h,j} = B_{h-1,2j-1} + B_{h-q,2j}$$

2. Danach wird vom Wurzelknoten aus für alle Teilbäume die Präfixsumme des am weitesten rechts stehenden Blattknotens bestimmt (und im Wurzelknoten des Teilbaums gespeichert). Dabei sind drei Fälle zu unterscheiden:

   - Der Teilbaum ist rechter Sohn, d. h. j ist gerade: Dann ist der äußere rechte Blattknoten des Teilbaums derselbe Knoten, der auch im Vaterbaum rechts außen steht. Also wird der Eintrag des Vaters übernommen.

     $$C_{h,j} = C_{h+1,\frac{j}{2}}$$

   - Der Teilbaum ist linker Sohn und steht ganz links, d. h. $j = 1$: Dann ist die Präfixsumme des äußeren rechten Knotens die Summe aller Blattknoten.

     $$C_{h,j} = B_{h,j}$$

   - Der Teilbaum ist linker Sohn und steht nicht ganz links, d. h. $j > 1$ und ungerade: Dann ist die Präfixsumme des äußeren rechten Blattknotens gleich der Summe aller Blattknoten des Teilbaums plus der Präfixsumme des Blattknotens, der links vom am weitesten links stehenden Blattknoten in diesem Teilbaum steht. Diese Präfixsumme steht bereits im linken Onkelknoten.

     $$C_{h,j} = B_{h,j} + C_{h+1,\frac{j-1}{2}}$$

Der Algorithmus lässt sich für beliebige zweistellige, assoziative Operationen $\star$ verwenden und kann für beliebige n-stellige, assoziative Operationen erweitert werden.

$$s_k = x_1 \star x_2 \star \cdots \star x_{k-1} \star x_k$$

todo: Bild aus Beispiel 2.9 übernehmen

Abbildung 3.2: Ein Wald mit drei wurzelgerichteten Bäumen

Beide Schritte benötigen jeweils $O(\log n)$ TIME und $O(n)$ WORK. Der Algorithmus braucht also insgesamt $O(\log n)$ TIME und $O(n)$ WORK und ist daher optimal. Da ein serieller Algorithmus für die Berechnung der n. Präfixsumme mindestens einmal alle Folgenglieder betrachten muss, benötigt er $O(n)$ TIME. Der parallele Präfixsummenalgorithmus ist also auch streng optimal. todo: Prüfen! Diese Aussage stimmt IMO nicht, da wir später noch einen schnelleren Algorithmus bekommen.

Wenn die Zahlen in Form einer verketteten Liste gegeben sind, kann dieser Algorithmus nicht verwendet werden, da die Zuordnung des n. Element auf den n. Prozessor, wie sie für den Aufbau des Baums notwendig ist, nicht in $O(1)$ durchgeführt werden kann. Dazu später mehr unter dem Thema Parallel Prefix im Unterabschnitt 3.2.1.

3.2 Pointer Jumping

Um die Idee des **Pointer jumpings** (oder **path doubling**) zu verdeutlichen, soll in einem Wald (Disjoint Set Forest) zu jedem Knoten der Wurzelknoten des Baums, in dem der Knoten hängt, bestimmt werden.

Ein **Wald** ist eine Menge von paarweise disjunkten Bäumen, die durch ihre Wurzel gekennzeichnet sind. Im Speziellen geht es um wurzelgerichtete Bäume, d. h. jeder Knoten des Baums, außer der Wurzel r, hat genau einen Vorgängerknoten und der Baum ist in der Form gegeben, dass jeder Knoten auf seien Vorgängerknoten verweist. Der Vorgängerknoten des Wurzelknotens ist er selbst. Abbildung 3.2

Input : n-dimensionales Feld F, wobei $F[i] = j$, wenn j der Vater von i ist
Output : n-dim. Feld S, wobei $S[i] = j$, wenn j die Wurzel des Baums ist, in dem i steht

```
1 for i := 1 to n pardo
2 |   S[i] := F[i]
3 |   while S[i] ≠ S[S[i]] do
4 |   |   S[i] := S[S[i]]
5 |   end
6 endfor
```

Algorithmus 3.3: Pointer jumping

Der Wald ist als ein n-dimensionales Feld F gegeben, in dem der i. Eintrag $F[i]$ den Vorgängerknoten des i. Knotens beinhaltet. Jedem Knoten wird ein Prozessor zugewiesen, der den Wurzelknoten dadurch findet, dass er in jedem Schritt den Vorgängerknoten seines Vorgängerknotens ermittelt. Algorithmus 3.3

Nach dem *ersten* Schritt zeigen alle Knoten auf den Vorgänger (2. Generation) ihres direkten Vorgängers. Im zweiten Schritt greifen die Prozessoren auf diese Ergebnisse zu, so dass sie nicht den Vorgänger der 3. Generation bestimmen, wenn sie den Vorgänger des Vorgängers der

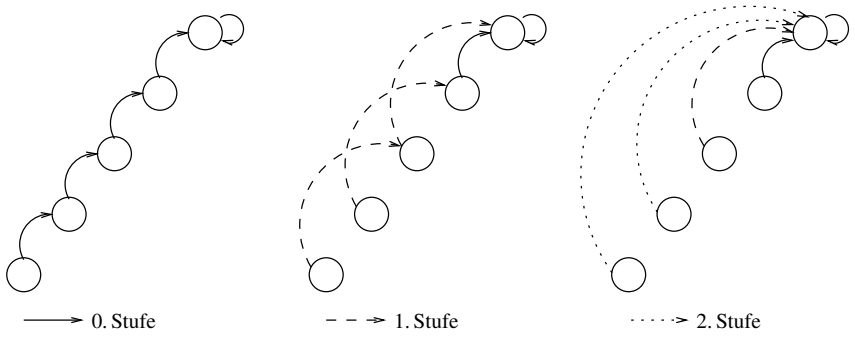


Abbildung 3.3: Veranschaulichung des Pointer jumpings

2. Generation bestimmen, sondern den der 4. Generation. Nach dem *zweiten* Schritt verweisen also alle Knoten auf den Vorgänger der 4. Generation. Abbildung 3.3 Im *dritten* Schritt ist dann der Vorgänger des Vorgängers (der 4. Generation) der Vorgänger der 8. Generation. So springt der Zeiger durch den Baum und bewegt sich nicht von Knoten zu Knoten. Da die Anzahl der Zugriffe auf eine einzelne Zelle des Vorfahrenfeldes S nicht durch eine Konstante beschränkt ist, wird für den Algorithmus echtes concurrent read benötigt.

h sei die maximale Höhe eines Baums im Wald. Auf einer CREW-PRAM liefert der Algorithmus in $O(\log h)$ TIME und $O(n \cdot \log h)$ WORK die Wurzel zu jeden Knoten. Da sich das Problem im Seriellen in $\theta(n)$ lösen lässt, ist der Algorithmus nicht optimal.

3.2.1 Parallel Prefix

Entartet man den Wald zu einem Baum, der auch keine Verzweigungen hat, kann man damit die **Präfixsummen** für Zahlen, die in einer verketteten Liste gegeben sind, berechnen. Dieses Problem bezeichnet man als **parallel Prefix**. Die Zahlen sind in einer weiteren Liste W gegeben, die allen Knoten im Wald F ein Gewicht zuordnet: $W[i]$ ist das Gewicht am Knoten i. Der Wald selbst beschreibt die Ordnung der Zahlen, wobei der Wurzelknoten die erste Zahl und der Blattknoten die letzte Zahl ist.

Jeder Prozessor übernimmt wieder einen Knoten und springt auf die oben beschriebene Weise durch den Baum (die Liste). Dabei berechnet er jeweils die Summe der Gewichte von ihm bis zu seinem Vorgängerknoten, auf den er gerade springt/verweist.

Mit parallel Prefix kann man also in $O(\log n)$ TIME und $O(n \log n)$ WORK sie Präfixsummen einer verketteten Liste berechnen. Da die Präfixsummenberechnung im Seriellen mit $O(n)$ TIME funktioniert, ist der Algorithmus nicht optimal.

In einem nicht entarteten Wald kann mit diesem Algorithmus für jeden Knoten die Entfernung zu seinem Wurzelknoten bestimmen, wenn man für die Knoten das Gewicht $w_i = 1$ und für den Wurzelknoten r das Gewicht $w_r = 0$ wählt.

1 todo: Algorithmus auf Feldschreibweise ([] statt Index) umstellen
Input : zwei n-dim. Felder F, W, wobei w_i das Gewicht am Knoten i ist und $f_i = j$, wenn j der Vater von i ist
Output : n-dim. Feld P, wobei p_i die Summe der Gewichte auf dem Pfad von Knoten i zur Wurzel ist
2 **for** $i := 1$ **to** n **pardo**
3 $\quad s_i := f_i$ `/* S ist die Liste der Vorgängerknoten */`
4 $\quad p_i := w_i$
5 $\quad$ **while** $s_i \neq s_{s_i}$ **do**
6 $\quad\quad p_i := p_i + p_{s_i}$
7 $\quad\quad s_i := s_{s_i}$
8 $\quad$ **end**
9 **endfor**

Algorithmus 3.4: Pointer Jumping mit Knotengewichten

3.3 Teile und Herrsche

Die Strategie **Teile und Herrsche** (engl. **Divide and Conquer**) ist bereits von den seriellen Algorithmen bekannt. Ein Problem wird in Teilprobleme zerlegt, die untereinander möglichst gleich groß, aber kleiner als das ursprüngliche Problem sind. Diese Probleme werden unabhängig voneinander gelöst und aus deren Lösung die Lösung des Gesamtproblems konstruiert.

Die Zerlegung in Teilprobleme sollte dabei „einfach“ sein, das Zusammenfügen der Gesamtlösung kann „kompliziert“ sein. – Der umgekehrte Fall heißt Partitioning (Abschnitt 3.4). – Durch die Teilung in *unabhängige* Teilprobleme bietet sich eine Verteilung auf mehrere Prozessoren direkt an.

Zur Erläuterung des Paradigmas soll das Problem der Bestimmung der konvexen Hülle einer Punktmenge im $\mathbb{R}^2$ aus der algorithmischen Geometrie dienen: Die **konvexe Hülle** einer Punktmenge $S \subset \mathbb{R}^2$ ist das kleinste Polygon, dass alle Punkte der Menge und die Strecke zwischen ihnen[1] umfasst.

Das Problem lässt sich im Seriellen in $O(n \log n)$ TIME lösen, wobei $\Omega(n \log n)$ die untere Schranke für die Laufzeit ist, da man aus der konvexen Hülle, die Sortierung der Punkte herleiten kann und dafür ist $\Omega(n \log n)$ bekanntlich die untere Schranke.

O. B. d. A. seien die x- und die y-Koordinaten der Punkte von S paarweise verschieden. Die Punktmenge S sei aufsteigend nach der x-Koordinate geordnet. Der Punkt mit der kleinsten und der mit der größten x-Koordinate gehören zur konvexen Hülle und teilen sie in eine obere und eine untere konvexe Hülle. Wir betrachten nur die Konstruktion der oberen konvexen Hülle *UCH* (engl. upper convex hull), die der unteren konvexen Hülle *LCH* verläuft analog.

Der Algorithmus (Algorithmus 3.5) verläuft in zwei Phasen:

- in der Top-Down-Phase zerlegt man die Menge S solange in zwei gleichgroße Teilmengen $S_1 = \{p_1, \ldots, p_{\frac{n}{2}}\}$ und $S_2 = \{p_{\frac{n}{2}+1}, p_n\}$ bis die Konstruktion der oberen konvexen Hülle

[1] Man nennt eine Menge M **konvex**, wenn für alle Punkte x_1 und x_2 aus M auch die Strecke zwischen x_1 und x_2 zu M gehört: $\forall x_1, x_2 \in M, \forall \lambda \in [0,1] : (\lambda x_1 + (1-\lambda) x_2) \in M$

Input : Punktmenge $S \subset \mathbb{R}^2$, nach x sortiert
Output : Obere konvexe Hülle von S
1 **if** $n \leq 4$ **then**
2 | bestimme $UCH(S)$ durch einfaches Durchsuchen aller Tupel
3 **else**
4 | teile S in $S_1 = \{p_1, \ldots, p_{\frac{n}{2}}\}$ und $S_2 = \{p_{\frac{n}{2}+1}, \ldots, p_n\}$;
5 | bestimme $UCH(S_1)$ und $UCH(S_2)$ rekursiv und parallel;
6 | bestimme die obere Tangente von $UCH(S_1)$ und $UCH(S_2)$ und konstruiere $UCH(S)$ durch Verbinden der beiden höchsten Punkte
7 **end**

Algorithmus 3.5: Bestimmen der oberen konvexen Hülle einer Punktmenge S – $UCH(S)$

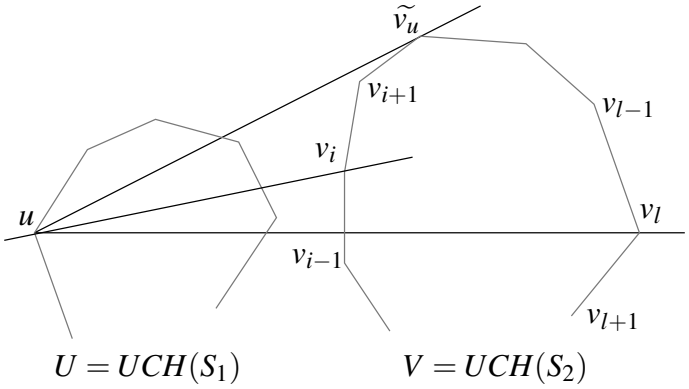


Abbildung 3.4: Bestimmung der Tangente vom Punkt u aus an V

trivial ist. Das Bestimmen von $UCH(S_2)$ wird dabei an einen anderen Prozessor abgegeben.

- in der Bottom-Up-Phase wird für je zwei oberen konvexen Hüllen die gemeinsame obere konvexe Hülle bestimmt, indem man die gemeinsame obere Tangente an beide Hüllen findet und sie so verbindet.

Der Rekursionsbaum hat also die Tiefe $O(\log n)$. Damit der Algorithmus streng optimal ist, d. h. $O(\log n)$ TIME und $O(n \log n)$ WORK, darf jeder Rekursionsschritt nur $O(1)$ TIME und $O(n)$ WORK benötigen. Die Bestimmung der Tangente muss also in $O(1)$ TIME erfolgen.

Zur Vereinfachung der Schreibung sei $U := UCH(S_1)$ und $V := UCH(S_2)$. Der Punkt $\widetilde{v}_u \in V$ sei der Berührungspunkt der Hülle, so dass die Gerade durch u und $\widetilde{v}_u$ eine Tangente an V ist. Da V konvex ist, liegen alle Punkte unterhalb oder auf der Geraden $\overline{u\widetilde{v}_u}$.

Um für einen Punkt $u \in U$ die relative Lage des Punkts $\widetilde{v}_u \in V$ zu $v_i \in V$ zu bestimmen, verbindet man die beiden Punkte durch eine Gerade $\overline{uv_i}$ und bestimmt, ob der Vorgänger $v_{i-1} \in V$ und der Nachfolger $v_{i+1} \in V$ von v nicht auf der gleichen Seite der Gerade $\overline{uv_i}$ liegen. Ist dies der Fall, so liegt der Punkt $\widetilde{v}_u$ oberhalb von v_l, da die Eigenschaft der Tangente eben besagt, dass kein Punkt über ihr liegt. Sind also die Punkte v_{i-1}, v_i, v_{i+1} mit wachsender y-Koordinate, so ist $j > i$ für $v_j = \widetilde{v}_u$. Andernfalls ist $j \leq i$ für $v_j = \widetilde{v}_u$. Siehe Abbildung 3.4. Diese Entscheidung lässt sich

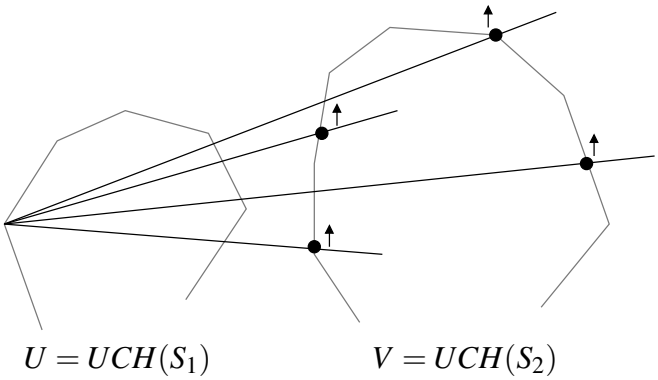


Abbildung 3.5: Einschränken des Intervalls, in dem der Punkt für die Tangente liegt

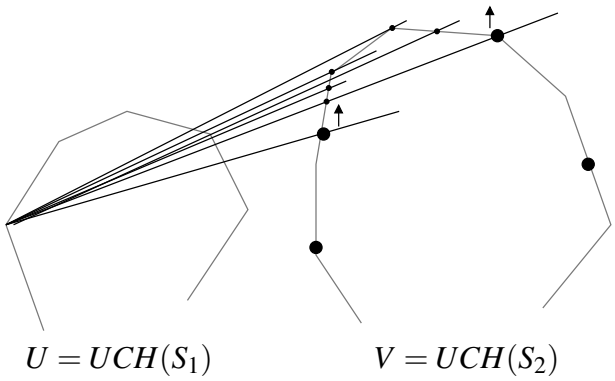


Abbildung 3.6: Bestimmen des Punkts für die Tangente im einschränkten Intervall

in $O(1)$ TIME treffen.

Zerlegt man V in $d_V = \sqrt{\text{card}\,V}$ gleichgroße Segmente $V^1, \ldots, V^{d_V}$, so kann man für den Punkt u in $O(1)$ TIME mit d_V Prozessoren ein Segment bestimmen, in dem der Punkt $\widetilde{v}_u$ liegt. Jeder Prozessor P_k bestimmt für die beiden Endpunkte von V^k, ob $\widetilde{v}_u$ zwischen ihnen liegt – also rechts vom linken und links vom rechten Punkt aus. Siehe Abbildung 3.5. Das geht in $O(1)$ TIME. Da das Intervall d_V Punkte enthält, kann man mit den d_V Prozessoren in $O(1)$ TIME darin den Punkt $\widetilde{v}_u$ finden, für den beide Nachbarn unterhalb der Geraden $\overline{u\widetilde{v}_u}$ liegen. Siehe Abbildung 3.6. Aufwand wiederum $O(1)$ TIME.

Zerlegt man ebenfalls U in $d_U = \sqrt{\text{card}\,U}$ gleichgroße Segmente $U^1, \ldots, U^{d_U}$ und führt mit d_U Prozessoren die Bestimmung der Tangenten von den Endpunkten der U^j aus, so bekommt man in $O(1)$ TIME über die gleiche Beziehung wie oben für $\widetilde{v}_u$ ein Segment $\hat{U}$, in dem der Berührungspunkt $\hat{u} \in U$ der gemeinsamen Tangente an U liegt. Siehe Abbildung 3.7. Dafür werden insgesamt $d_U \cdot d_V = \sqrt{U} \cdot \sqrt{V} \leq \sqrt{\frac{n}{2}} \cdot \sqrt{\frac{n}{2}} = O(n)$ Prozessoren eingesetzt.

Da in dem Segment $\hat{U}$, das $\hat{u}$ enthält, d_U Punkte liegen, kann man mit d_V Prozessoren für jeden Punkt $u_i \in U^k$ in $O(1)$ TIME den Berührungspunkt $\widetilde{v}_{u_i}$ an V bestimmen. Die gemeinsame Tangente zeichnet sich dadaruch aus, dass die Nachbarpunkte u_{i-1} und u_{i+1} unterhalb der Geraden $\overline{u_i\widetilde{v}_{u_i}}$

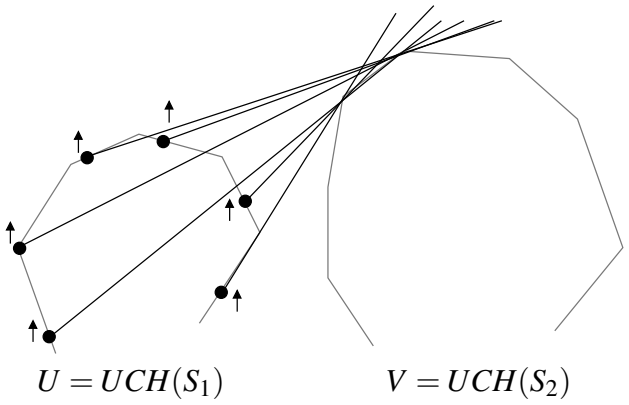


Abbildung 3.7: Parallele Bestimmung der Tangente für ausgewählte Punkte von U

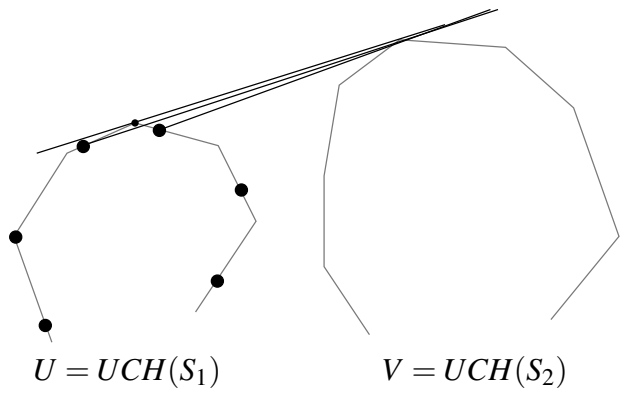


Abbildung 3.8: Bestimmung der oberen Tangente durch betrachten aller Tangenten aus dem verbleibenden Intervall

liegen. Siehe Abbildung 3.8.

3.4 Partitioning

Partitioning oder auch **Zerlegunsstrategie** genannt, ähnelt dem Teile-und-Herrsche-Ansatz (Abschnitt 3.3). Ein Problem wird in unabhängige Teilprobleme zerlegt, diese werden einzeln gelöst und aus deren Teillösungen die Gesamtlösung konstruiert. Der Unterschied zum Teile-und-Herrsche-Ansatz ist, dass beim Partitioning die Zerlegung in Teilprobleme kompliziert, aber der Aufbau der Gesamtlösung einfach ist.

Die Idee des Partitionings soll an der Vereinigung von zwei sortierten Folgen A und B mit m bzw. n Elementen zu einer sortierten Folge C mit $m+n$ Elementen erläutert werden. Dieses Problem wird als **Merge** oder **Mischen** bezeichnet.

Definition 9
Eine Folge $A = (a_1, \ldots, a_m)$ heißt genau dann **sortiert**, wenn alle Folgenglieder größer oder gleich ihren Vorgängern sind: $\forall i \in \{1, \ldots, m\} \forall j \in \{1, \ldots, i\} : a_j \leq a_i$.

Definition 10
Als den **Rang** eines Elements x bezüglich einer Menge $A = \{a_1, \ldots, a_m\}$ bezeichnet man die Anzahl der Elemente aus A, die kleiner oder gleich x sind.

$$Rang(x : A) := \text{card}\{a : a \in A \wedge a \leq x\}$$

Als den **Rang** einer Folge $B = (b_1, \ldots, b_n)$ bezüglich einer Menge $A = \{a_1, \ldots, a_m\}$ bezeichnet man die Folge der Ränge der Elemente von B

$$Rang(B : A) := \big(Rang(b_1 : A), Rang(b_2 : A), \ldots, Rang(b_{n-1} : A), Rang(b_n : A)\big)$$

Beispiel 3

$$Rang(4 : \{7,25,23,5\}) = 0 \qquad Rang(12 : \{5,7,23,25\}) = 2$$
$$Rang(4 : \emptyset) = 0 \qquad Rang(12 : \{1,3,2\}) = 3$$
$$Rang((4,12) : \{5,7,23,25\}) = (0,2) \qquad Rang((12,4) : \{7,23,3,25,1,2,5\}) = (5,3)$$

Bemerkung 5
Im Folgenden seien zur Vereinfachung die Elemente aus A und B paarweise verschieden.

Satz 7
Seien A und B zwei Folgen mit m bzw. n Elementen. Die Folge $C = (c_1, \ldots, c_{m+n})$ mit den Gleidern c_i ist die gemischte, sortierte Folge von A und B, wobei $c_i = x \in A \cup B$ genau dann, wenn $i = Rang(x : A \cup B)$ ist.

BEWEIS:
trivial ∎

Um also zwei Folgen A und B zu einer sortierten Folge C zu vereinen, muss man für alle $x \in A \cup B$ den Rang von x in $A \cup B$ bestimmen. Dabei gilt die folgenden Beziehung:

$$Rang(x : A \cup B) = Rang(x : A) + Rang(x : B)$$

Sind A und B bereits sortiert, so ist der Rang von $a \in A$ bezüglich A bzw. der Rang von $b \in B$ bezüglich B die Postition in der Folge. Bestimmt man dann noch $Rang(A : B)$ und $Rang(B : A)$, so ist die sortierte, gemischte Folge von A und B leicht zu bestimmen.

Bemerkung 6
Im Folgenden seien A und B immer sortiert.

Um den Rang eines Elements x bezüglich einer sortierten Folge B mit n Elementen zu bestimmen, kann man auf die sequentielle, binäre Suche zurückgreifen, womit sich $Rang(x : B)$ in $O(\log n)$ TIME bestimmen lässt.

Da die Aufgaben ($Rang(A : B)$ und $Rang(B : A)$ bestimmen) symmetrisch zueinander sind, genügt es den Vorgang $Rang(A : B)$ zu analysieren: Da die Folge B sortiert ist, kann man für ein Element $x \in A$ mit binärer Suche in $O(\log n)$ TIME mit einem Prozessor $Rang(x : B)$ bestimmen. Da alle Operationen parallel auf einer CREW-PRAM ausgeführt werden können, erhält man so für die gesamte Folge $O(\log n)$ TIME und $O(m \log n)$ WORK.

Analog erhält man für $Rang(B : A)$ $O(\log m)$ TIME und $O(n \log m)$ WORK. Für den gesamten Algorithmus ergibt sich somit $O(\log(m+n))$ TIME und $O((m+n)\log(m+n))$ WORK. Da das Mischen im Seriellen in Linearzeit $T^*(n) = \Theta(n)$ TIME geht, ist der Algorithmus nicht optimal.

Um einen optimalen Algorithmus zu erhalten, kann man das Problem mit der **Partitioning**-Strategie angehen:

1. Zerlegung (Partitioning) von A in $\frac{m}{\log m}$ Abschnitte der Länge $\log m$: $A_1, A_2, \ldots, A_{m/\log m}$.
2. Für die Elemente an den Abschnittsgrenzen $a_{\log m}, a_{2\log m}, \ldots, a_{1+m-\log m}$ ihre Positionen $j(1), j(2), \ldots, j(m/\log m - 1)$ innerhalb der Folge B bestimmen.
3. Damit ergeben sich Paare von Abschnitten von A und B die unabhängig voneinander mit einem sequentiellen Algorithmus, der in Linearzeit arbeitet, gemischt werden können. Das Mischen der Abschnitte (A_i, B_i), $i = 1, 2, \ldots, m/\log m$, geschieht parallel.
4. Aneinanderhängen der gemischten Abschnitte zur Folge C.

Beim Mischen der Paare (A_i, B_i) muss darauf geachtet werden, dass beide Abschnitte höchstens $\log m$ bzw. $\log n$ Elemente enthalten, damit der Mischvorgang nicht länger als $(\log m + \log n)$ TIME braucht. Unter ungünstigsten Umständen sind alle Elemente aus A größer als alle Elemente aus B, so dass der Prozessor für das Paar (A_1, B_1) alle Elemente von B behandelt, weil $B_1 = B$ ist.

Für die Abschnitte A_i ist die Größenbeschränkung bereits gegeben. Sollte ein Abschnitt B_i mehr als $\log n$ Elemente enthalten, so teilt man ihn in Abschnitte der Länge $\log n$ und bestimmt für diese die zugehörigen (Unter)-Abschnitte innerhalb von A_i. Dadurch werden es zwar mehr Paare, aber es ist sichergestellt, dass sie in $O(\log m + \log n)$ TIME gemischt werden können, und es werden nie mehr als $m + n + 1$ Paare. Siehe Abbildung 3.9.

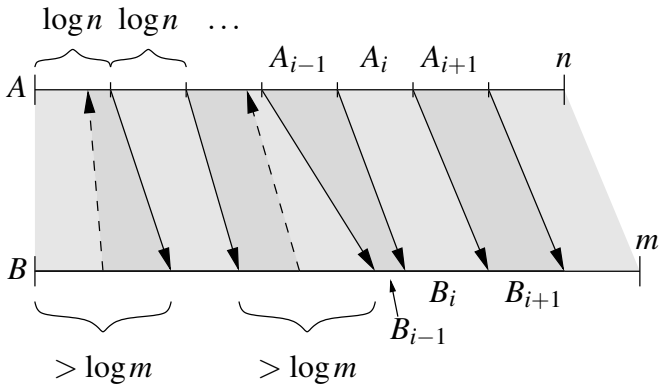


Abbildung 3.9: Zerlegung von zwei Folgen in kleine Abschnitte (Streifen) zum direkten Mischen

Für das Eingliedern der neuen Abschnitte von B_i ist es ausreichend den Abschnitt A_i zu betrachten, da alle Elemente aus B_i größer sind als das erste Element im Abschnitt und dieses so gewählt wurde, dass es kleiner oder gleich dem ersten Element von A_i ist. Ebenso sind alle Elemente kleiner als das letzte, welches so gewählt wurde, dass es kleiner oder gleich dem letzten Element in A_i ist. Anschaulich gesprochen: Die Pfeile können sich nicht überschneiden.

Input : Zwei sortierte Folgen A und B mit m bzw. n Gliedern.
Output : Menge von Paaren von Teilfolgen (A_i, B_i) für die gemischte Gesamtfolge

1 $k := \frac{m}{\log m}$
2 $j[0] := 0$
3 $j[k] := n$
4 **for** $i := 1$ **to** $k-1$ **pardo**
5 $\quad j[i] := Rang(A[i \cdot \log m] : B)$ `/* mit binärer Suche */`
6 **endfor**
7 **for** $i := 0$ **to** $k-1$ **pardo**
8 $\quad A_{i+1} := \left[A[(i \cdot \log m)+1], \ldots, A[(i+1) \cdot \log m]\right]$
9 $\quad B_{i+1} := \left[B[j[i]+1], \ldots, B[j[i+1]]\right]$
10 **endfor**

Algorithmus 3.6: Zerlegen zweier sortierten Folgen in Paare von Teilfolgen zum Mischen

Satz 8
Das Mischen von zwei sortierten Folgen A und B mit m bzw. n Elementen benötigt $O(\log m + \log n)$ TIME und $O(m+n)$ WORK.

BEWEIS:
Die Zerlegung von A in Teilabschnitte $O(1)$ TIME und $O(1)$ WORK.

Die binäre Suche benötigt $\log n$ Schritte für die Bestimmung von $Rang(A[i \cdot \log m] : B)$. Da dies für $\frac{m}{\log m}$ Elemente gleichzeitig geschieht, ist der Aufwand für den zweiten Schritt $O(\frac{m}{\log m} \cdot \log n) = O(m+n)$ WORK. Die Rechenzeit ist $O(\log n)$ TIME.

Da $\frac{z}{\log z}$ für $z > 2$ monton wachsend ist, gilt für $m < m+n$

$$\frac{m}{\log m} < \frac{m+n}{\log(m+n)} \quad \Rightarrow \quad \frac{m}{\log m}\log n < \frac{m}{\log m}\log(m+n) < m+n$$

Das Mischen der Teilfolgen mit einem sequentiellen Algorithmus geht in $O(\log m + \log n)$ TIME und benötigt $O(m+n)$ WORK. ∎

Der parallele Algorithmus ist aber nicht streng optimal, da es einen Algorithmus gibt, der in $O(\log\log n)$ TIME mit $O(n)$ WORK funktioniert. Dazu später noch einmal mehr im Abschnitt 5.2.

3.5 Pipeline-Verfahren

Unter dem **Pipeline-Verfahren** versteht man folgendes Vorgehen: Zwei Aufgaben $\mathscr{A}$ und $\mathscr{B}$ werden in eine Folge von paralleilierbaren Teilaufgaben $\mathscr{A} = (a_1, \ldots, a_n)$ und $\mathscr{B} = (b_1, \ldots, b_n)$ aufgeteilt, so dass nach der Beendigung einer Teilaufgabe a_i die folgende Teilaufgabe a_{i+1} begonnen und parallel dazu die neue Teilaufgabe b_i abgearbeitet werden kann. Dieses Vorgehen ist nur dann lohnenswert, wenn mehrere Aufgaben erfüllt werden müssen.

Beispiel 4
Das Pipeline-Verfahren existiert auch in der Wirklichkeit und kann z. B. beim täglichen Gang in die Mensa beobachtet werden. Die Aufgabe „Essensausgabe“ für viele Gäste ist in die Teilprobleme „Hauptspeiße auf den Teller legen“, „Sättigungsbeilage auf den Teller legen“ und „Gemüsebeilage auf den Teller legen“ geteilt. Dafür sind drei Angestellte (Prozessoren) zuständig.

Der erste Angestellte nimmt einen Teller und legt die Hauptspeiße darauf. Danach gibt er den Teller dem zweiten Angestellten und nimmt sich wieder einen neuen Teller, auf den er die Hauptspeiße legt. Der nächste Angestellte legt die Sättigungsbeilage auf den Teller, gibt ihn an den dritten Angestellten und nimmt den nächsten Teller in Empfang und befüllt ihn. Der dritte Angestellte legt noch das Gemüse auf den Teller und reicht ihn dem Gast und nimmt den neuen Teller an.

Nach einer kurzen Anlaufphase von drei Takten, wird in jedem Takt ein Teller ausgegeben.

2-3-Bäume (ähnlich Top-Down-2-3-4-Bäumen, ausbalancierter Suchbaum mit Knoten aus zwei oder drei Elementen)

Gegeben: $A = (a_1, a_2, \ldots, a_n) \quad : \quad a_1 < a_2 < \ldots < a_n$ Blattsuchbaum
Innere Knoten: Pfad-Infos $(L[v], M[v], R[v])$ - jeweils größter Knoten im linken (bzw. mittleren oder rechten) Teilbaum

Wir wollen nun in diesen Baum $B = b_1, \ldots, b_k \quad : \quad b_1 < b_2 < \ldots < b_k \quad k << n$ einfügen:

Definition 11
$B_i \subseteq B$ - Werte zwischen a_i und a_{i+1}
$|B_i| := k_i \quad (\rightarrow$ Summe der k_i sind ergibt im wesentlichen $k-2$, nach dem ersten Schritt)

Vorschritt: Füge b_1, b_k ($O(\log n)$ TIME) (Damit alle Elemente zwischen zwei Elementen im Baum eingefügt werden und nicht vor dem linken oder nach dem rechten)

1. Fall $|B_i| \leq 1 \quad \forall i = 1, \ldots, n-1$

   WORSTCASE todo: Bild: Worstcase bei Fall1 einfügen

   $\Rightarrow$ bei pardo insert $(b_2, \ldots, b_{k-1})$ im Fall 1 entstehen pro Knoten über der Blattebene max. 6 Söhne

2. Fall ($\neg$ 1. Fall) $|B_i| = k_i$ ($\sum k_i = k-2$) möglich für ein i: $|B_i|=k_i=\Omega(k)$

   $b_{i_1}, \ldots, b_{i_k} \rightarrow$ mittleres Element mit Index $z := \lceil \frac{1+k_i}{2} \rceil$ also b_{i_z} das für alle i

   insert alle b_{i_z} = 1. Fall

   Damit reduzieren wir die Maximallänge der Folge der einzufügenden Elemente zwischen zwei Elementen auf die Hälfte. Das wird fortgesetzt, bis Fall 1 erreicht.

Höhe des Baumes: $O(\log n)$ Im 1. Fall: parallel (pardo) : $O(\log n)$ TIME, $O(k \log n)$ WORK 2. Fall: ($\log k$)-mal durchzuführen, jedesmal Fall 1 $\Rightarrow O(\log k \log n)$ TIME, $O(k \log n)$ WORK

Aufgabe: INSERT z. B. $b_{i_1}, \ldots, b_{i_{k_i}}$

1. Teilaufgabe INSERT: INSERT jeweils das erste gelbe Element 2. Teilaufgabe: INSERT jeweils die beiden nächsten gelben Elemente $\Rightarrow O(\log n + \log k) = O(\log n)$ TIME

Bemerkung 7
Das erste Element braucht $\log n$ Schritte um an seiner Stelle zu sein. Das letzte Element wird nach $\log k$ Schritten losgeschickt und kommt nach weiteren $\log n$ Schritten an, also $\log k + \log n$

3.6 Accelarated Cascading

Im Kapitel 2 kam bereits die Strategie des **Accelarated Cascading** schon einmal zum Einsatz. Die Idee ist, einen langsamen, optimalen und einen schnellen, nicht optimalen Algorithmus so zu kombinieren, dass ein schneller(er) und optimaler Algorithmus entsteht.

1. Verkleinere das Problem mit einem optimalen, aber langsamen Algorithmus, bis durch die Anwendung des nicht optimalen, aber schnellen Algorithmus' sich der WORK nicht erhöht.

2. Wende auf das verkleinerte Problem aus dem ersten Schritt den nicht optimalen, aber schnellen Algorithmus an.

Ein Spezialfall des Accelarated Cascading ist das **Sequential Subset**, bei dem die zwei Algorithmen nicht nacheinander ausgeführt werden, sondern ineinander verschachtelt sind, so dass einer der beiden den globalen Ablauf bestimmt und der andere für die lokale Arbeit eingesetzt wird.

Durch den Einsatz von Sequential Subset und Accelarated Cascading kann man einen Algorithmus bekommen, der in $O(\log \log n)$ TIME mit $O(n)$ WORK das Maximum bestimmt. Dies gilt aber nur für eine CRCW-PRAM. Auf einer CREW-PRAM gilt prinzipell $\Omega(\log n)$ TIME unabhängig von WORK.

Satz 9
Auf einer CREW-PRAM ist $\Omega(\log n)$ die untere Schranke für das bestimmen des Maximums von n Elementen.

BEWEIS:
Das Problem ist äquivalent zur Berechnung einer n-stelligen boolschen Funktion $f\colon \{0,1\}^n \to \{0,1\}$.

Für eine Eingabe $I = (x_1,\ldots,x_n) \in \{0,1\}^n$ ist $I(i) = (x_1,\ldots,\neg x_i,\ldots,x_n) \in \{0,1\}^n$. Eine Eingabe heißt genau dann kritisch, wenn für alle i $(1 \leq i \leq n)$ gilt: $f(I) \neq f(I(i))$. (Bezogen auf das Maximumproblem wäre die Folge $(0,\ldots,0)$ eine kritische Eingabe.)

Irgendwo kommt ein Haupttheorem her, das besagt: Besitzt eine Funktion $f\colon \{0,1\}^n \to \{0,1\}$ eine kritische Eingabe, dann sind auf der CREW-PRAM $\Omega(\log n)$ Schritte zur Berechnung von f nötig. help: Warum? Wieso? Weshalb? Können wir das beweisen oder dauert das zu lange? ■

Die Bestimmung des Maximums einer Folge von n Elementen geht mit dem Binärbaumparadigma in $O(\log n)$ TIME mit $O(n)$ WORK. Das ist optimal, da das serielle Optimum $T^*(n) = \Theta(n)$ ist, aber es geht schneller. Algorithmus 3.7 bestimmt das Maximum in konstanter Zeit mit $O(n^2)$ WORK und kann auf einer common CRCW-PRAM implementiert werden, da in Zeile 5 alle Prozessoren das gleiche schreiben. Dieser Algorithmus ist aber nicht optimal.

Input : Ein Feld A mit n verschiedenen Elementen
Output : Boolsches Feld M der Länge an, für das gilt $M[i] = 1 \Leftrightarrow A[i] = \max A$

```
1 for i := 1 to n pardo
2 |  M[i] := 1
3 |  for j := 1 to n pardo
4 |  |  if A[i] < A[j]) then
5 |  |  |  M[i] := 0
6 |  |  end
7 |  endfor
8 endfor
```

Algorithmus 3.7: schnelle, nicht optimale Bestimmung des Maximums von n Elementen

Das Problem an Algorithmus 3.7 ist, dass er zu viele Prozessoren verwendent, wärend beim Binärbaum von Stufe zu Stufe weniger Prozessoren verwendet werden. Kombiniert man beide Algorithmen in dem Sinne, dass man die Berechnung stufenweise ablaufen lässt und immer größere Gruppen von Prozessoren mit dem nicht optimalen Algorithmus zusammenfasst, so bekommt man die gewünschte Beschleunigung mit konstantem Aufwand.

Die Gruppengröße soll doppeltexponentiell wachsen, d. h. in der ersten Stufe sind es Zweiergruppen, in der zweiten Vierergruppen, in der dritten Sechzehnergruppen usw. Es entsteht ein

sogenannter Baum mit doppellogarithmischer Tiefe.

Stufe	#Eingabe	Gr.größe	#Gr.	Proz. pro Gr.	Proz. insg.	#Ausgabe
1.	$n = \frac{n}{2^0}$	$2 = 2^1$	$\frac{n}{2}$	2^2	$2n$	$\frac{n}{2}$
2.	$\frac{n}{2} = \frac{n}{2^1}$	$4 = 2^2$	$\frac{n}{8}$	4^2	$\frac{16n}{8} = 2n$	$\frac{n}{8}$
3.	$\frac{n}{8} = \frac{n}{2^3}$	$16 = 2^4$	$\frac{n}{128}$	16^2	$\frac{256n}{128} = 2n$	$\frac{n}{128}$
4.	$\frac{n}{128} = \frac{n}{2^7}$	$256 = 2^8$	$\frac{n}{2^{15}}$	2^{16}	$\frac{2^{16}n}{2^{15}} = 2n$	$\frac{n}{2^{15}}$
$\vdots$						
$i.$	$n \cdot 2^{1-2^{i-1}}$	$2^{2^{i-1}}$	$n \cdot 2^{1-2^i}$ $(\star)$	$(2^{2^{i-1}})^2 = 2^{2^i}$	$2n$ $(\star\star)$	$n \cdot 2^{1-2^i}$

Für die Rechnung $(\star)$:

$$n \cdot 2^{1-2^{i-1}} \cdot 2^{-2^{i-1}} = n \cdot 2^{1-2^i}$$

Für die Rechnung $(\star\star)$:

$$n \cdot 2^{1-2^{i-1}} \cdot 2^{2^{i-1}} = n \cdot 2$$

Das Maximum ist bestimmt, wenn die Ausgabe einer Stufe nur noch ein Element umfasst:

$$\begin{aligned} n \cdot 2^{1-2^i} = 1 \quad &\Leftrightarrow \quad 2^{2^i-1} = n \\ &\Leftrightarrow \quad 2^i - 1 = \log n \\ &\Leftrightarrow \quad i = \log(1 + \log n) \end{aligned}$$

Der Algorithmus braucht also $O(\log\log n)$ TIME und $O(n \cdot \log\log n)$ WORK und ist damit nicht optimal. Aber durch den Einsatz von Accelarated Cascading kann ein optimaler Algorithmus entworfen werden:

1. Mit dem optimalen (Binärbaum-)Algorithmus reduziert man die Elemente auf $\frac{n}{\log\log n}$ Stück und

2. arbeitet dann mit dem nicht optimalen, doppeltexponentiellen Algorithmus weiter.

Der erste Schritt braucht $O(\log\log\log n)$ TIME und verwendet $O(n)$ WORK (da der Algorithmus optimal; grobe Abschätzung).

$$n \cdot 2^{-i} = \frac{n}{\log\log n} \quad \Leftrightarrow \quad 2^i = \log\log n \quad \Leftrightarrow \quad i = \log\log\log n$$

Im zweiten Schritt gehen $\frac{n}{\log\log n}$ Elemente ein. Das Ergebnis wird also in $O(\log\log \frac{n}{\log\log n}) = O(\log\log n)$ TIME mit $O(\frac{n}{\log\log n} \cdot \log\log \frac{n}{\log\log n}) = O(n)$ WORK bestimmt. Damit ist der Algorithmus optimal und alle sind glücklich.

3.7 Aufbrechen von Symmetrien

Einfärben der Knoten, so dass eine Kante nicht zwei Knoten gleicher Farbe verbindet. Geht seriell in O(n) TIME

Basisalgortihmus. INPUT: Array der n Knoten (Kreis), zulässige Farbung c_n OUTPUT: c' neue Färbung, zulässig begin

```
for 1 $\leq$ i $\leq$ n pardo
setze k gleich der kleinsten singifikanten Position, wo sich c(i)
und c(s(i)) als Binärzahlen sich unterscheiden
c'(i) := 2k+c(i)
```

end

Basisoperationen: $i = i_{t-1}i_{t-2}\dots i_1 i_0$ k-t-kleinstes signifikantes Bit := i_k

– vierstellige: 0001 Vereinbarung: möglichst kurz

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
*	3		7				14							2	

v	Basisfärbung	k	c'
1	0001	1	2
3	0011	2	4
7	0111	0	1
14	1110	2	5
2	0010	0	0
15	1111	0	1
4	0100	0	0
5	0101	0	1
6	0110	0	0

$$c'(1) = 2k + c(1) = 2 + 0 = 2 \tag{3.1}$$

$$c'(2) = 2 \cdot 0 + 0 = 0 \tag{3.2}$$

Lemma 1
c zulässig → c' ist auch zulässig, T(n)=O(1), W(n)=O(n)

BEWEIS:
(indirekt) Ann.: c'(i)=c'(j) für einen Kante $(i,j) \in E$, d. h. j=S(i) $\Rightarrow c'(i) = 2k + c(i)_k = c'(j) = 2l + c(j)_l$ Wegen Faktor 2 und da $c(i)_k, c(j)_l \in \{0,1\} \Rightarrow k = l \Rightarrow c(i)_k = c(j)_l$ Widerspruch zur Wahl von $k \Rightarrow c'(i) \neq c'(j)$

Die Aussagen zu TIME und WORK gelten offensichtlich. ■

Sei $t > 3$. Dafür reichen für c' $\lceil \log t \rceil + 1$ Binärplätze
$\Rightarrow c$ q Farben hatte $\Rightarrow 2^{t-1} < q \leq 2^t$, so braucht c' $2^{\lceil \log t \rceil + 1} = O(t) = O(\log g)$ Farben
$t > 3 \Rightarrow \lceil \log t \rceil + 1 < t$
Iterative Anwendung des Basisalgo. bis $t = 3(\lceil \log t \rceil + 1 = 2 + 1 = 3)$
Farben: {0,1,2,3,4,5}=6 Eliminiere: 3;4;5 → 3 O(1)-Schritte parallel mit O(n) WORK

Satz 10
Wir können einen Kreis mit n Knoten mit 3 Farben färben in $O(\log^* n)$ TIME mit $O(n \log^* n)$ WORK

BEWEIS:
Basisalgo. iterativ bis auf $t = 3$ anwenden. ■

Bemerkung 8
Dieser Algorithmus ist ohne Mühe auf der EREW implementierbar.

Bemerkung 9
Algo nicht Optimal.

Ziel: optimaler Algrotihmus in $O(\log n)$ Zeit.

Satz 11
(o.Bew.) Man kann ganze Zahlen aus $[0, \log n]$ in $O(\log n)$ TIME mit $O(n)$ WORK sorierten.

Optimale Färbung: INPUT: Di-Kreis mit n Knoten, S OUTPUT: 3-Färbung begin

```
for i \le i \le n pardo C(i) := i
Wende den Basisalgo genau einmal an
Sortiere die Ecken nach Farbe
for i=3 to \lseil\log n\rseil do für alle Ecken der Farbe i
pardo Färbe v mit kleinster farbe aus \{0,1,2\}, die vom Vorgänger
und Nachfolger verschieden ist.
```

end.

Satz 12
3-Färben geht in $O(\log n)$ TIME optimal.

4 Listen und Bäume

4.1 List-Ranking

verkettete Liste L , Nachfolgerarray $S\ i \to S(i)$, $S(i) = 0$ Ende der Liste bei Knoten i

List-Ranking-Problem: $\forall i$ bestimme den Abstand von i zum Ende der Liste. (Erinnerung: Pointerjumping: $O(\log n)$ TIME, WORK: $O(n \log n)$)

Idee: sequential subset $\cdot \to \cdot \to \cdot \to \ldots \to \cdot$

Aufteilen des Arrays S in Abschnitte der Länge $\log n$.

Leider funktioniert die Idee nicht, da die Elemente in den Abschnitten sich auf Elemente außerhalb des Abschnitts beziehen, was das spätere mischen verkompliziert macht.

4.2 optimales Listranking

Ergebnis: Listranking geht in $O(\log n)$ TIME mit $O(n)$ WORK

Wir beweisen: $O(\log n \log\log n)$ geht optimal

4.2.1 Pointer Jumping

$O(\log n)$ TIME, $O(n \log n)$ WORK

Unterteilung des Arrays in Blöcke der Größe $\log n$ funktioniert nicht, da die Zeiger in den Blöcken auch außerhalb des Blocks zeigen können – Block nicht abgeschlossen.

Strategie trotzdem

1. Reduziere die Startliste von n auf $\frac{n}{\log n}$ Knoten.
2. Pointer Jumping auf red. Liste anwenden
3. Stelle die Ausgangsliste wieder her.

Aufgabe für Listranking: INPUT Liste, OUTPUT für jeden Knoten den Abstand zum Ende.

Algo. Pointer Jumping

i	1	2	3	4	5	6	7	8	9
$S(i)$	2	6	1	5	7	8	3	9	0

Definition 12
Eine Teilmenge I aller Knoten heiße unabhängig, wenn für alle i: $[i \in I \to S(i) \notin I]$

begin end.

Input : Nachfolgerarray S
Output : $\forall i : R(i) \ldots$ Abstand zum Ende

```
1  for i := 1 to n pardo
2  |  if S(i) ≠ 0 then R(i) := 1
3  |  else R(i) := 0 ;
4  endfor
5  for i := 1 to n pardo
6  |  Q(i) := S(i)
7  |  while Q(i) ≠ 0 and Q(Q(i)) ≠ 0 do
8  |  |  R(i) := R(i) + R(Q(i)) ; Q(i) := Q(Q(i)) ;
9  |  end
10 endfor
```

Algorithmus 4.1: Algorithmus: List-Ranking

Input : Nachfolgerarray S, Vorgängerarray P, Teilliste I, $\forall i : R(i)$
Output : Gesamtliste ohne I (independent set)

```
1  Bestimme Seriennummern N(i) ∀i ∈ I: 1 ≤ N(i) ≤ |I| =: n' geht in O(log n) TIME
   optimal mit Hilfe des Präfixsummen-Algo über den Array
2  for i ∈ I pardo
3  |  U(N(i)) := (i, S(i), R(i)) – Info, die sonst verlohren geht!
4  |  R(P(i)(Vorgnger)) := R(P(i)) + R(i)
5  |  S(P(i)) := S(i)
6  |  P(S(i)) := P(i)
7  endfor
```

Algorithmus 4.2: Algorithmus: Entferne I

Beispiel 5
$I = \{1,5,6\}$, $i = 6$: $U(N(6)) = U(3) = (6,8,2)$ R(2)=4, S(2)=8, P(8)=2 Da U(3)=(6,8,2) $\rightarrow$ 8 war Nachfolger von 6 $\rightarrow R(6) = R(8) + 2 = 3$ Ausserdem kann jetzt 6 wieder eingefügt werden

Ziel: große unabhängige Menge; k-Färbung der Knoten

Die Knoten seien k-gefärbt: Farben {0,...,k-1} Knoten heißt lokales Minimum $\Leftrightarrow$ Farbe(i) = $\min\{\text{Farbe}(i), \text{Farbe}(\text{Vorgänger}(i)), \text{Farbe}(\text{Nachfolger}(i))\}$

Lemma 2
Bei gegebener k-Färbung in einer Liste L^n ist die Mende der lokalen Minima unabh. Menge von size $\Omega(\frac{n}{k})$ (Finde sie in O(1) TIME, O(n) WORK)

BEWEIS:
u,v lok. Minima, benachbert in dieser Eigenschaft max. Anzahl von Knoten zwischen zwei Minima u und v = 2k-2-1=2k-3 $\Rightarrow \Omega(\frac{n}{k})$
Bsp.: k=3: 2k-3=3

$\frac{1}{5}$ Elemente kommen raus (es bleiben $\frac{4}{5}$ Rest) Wir reduzieren von n Elementen auf höchstens $\frac{4}{5}n$ Elemente

4.2.2 Optimales Listranking

Beim Listranking mittels Pointer Jumping haben wir $O(\log n)$ TIME, $O(n \log n)$ WORK bewiesen. Da Listranking seriell in $O(n)$ läuft, ist dies nicht optimal. Der folgende Algorithmus benötigt $O(\log n \log\log n)$ TIME, $O(n)$ WORK (optimal), ist allerdings nicht W-T optimal.

Input : S^n
Output : $\forall$ Knoten Abstand zur Wurzel
1 $n_0 := n$
2 $k := 0$
3 **while** $n_k > \frac{n}{\log n}$ **do**
4 $\quad k := k + 1$
5 $\quad$ Färbe Liste mit 3 Farben, Bestimme I (lok. Minima)
6 $\quad$ Entferne die Knoten aus I (wie gehabt)
7 $\quad$ n_k sei die Zahl der Restknoten, die wir in aufeinanderfolgende Speicherplätze komprimieren
8 **end**
9 Pointer Jumping auf weißen Rest - optimal in $O(\log n)$ TIME
10 Nutze die Informationen in den Arrays U, um die Gesamtausgabe zu erhalten ;
Algorithmus 4.3: Algorithmus: Allgemeines einfaches optimales Listranking

Analyse $n_k \leq \left(\frac{4}{5}\right)^k \cdot n \leq \frac{n}{\log n} \Rightarrow \left(\frac{4}{5}\right)^k \leq \frac{1}{\log n}((\frac{4}{5})^{\log\log n} = O(\log n))k = O(\log\log n)$ reicht while-schleife $O(\log\log n)$ mal. $\Rightarrow$ TIME: $O(\log n \log\log n)$, WORK: $O(n)$

WORK des Algorithmus.: n_k Elemente bleiben nach Iteration k, damit

$$\text{WORK} = O(\sum_k n_k) = O(\sum_k \left(\frac{4}{5}\right)^k \cdot n) = O(n) \tag{4.1}$$

I
3-Färbung
R: [1]

Nach dem Löschen der Elemente

help: Kann das jemand entschlüsseln?

R:
3-Farben

$\frac{8}{\log 8} = \frac{8}{3} > 2 \rightarrow$ Abbruch

---

einfügen der Knoten in umgekehrter Reichenfolge ihres Entnehmens
R_{Ende}

Bemerkung 10
Analog dazu ist: Wende Parallel-Prefix auf die umkehrte Liste (Nachfolger = Vorgänger) an, um die Entfernung zum Ende zu berechnen.

Bemerkung 11
Listranking geht auch in $O(\log n)$, Zeit optimal (Lit.: Ja Ja)

4.3 Eulertour-Technik

$\Rightarrow$ Kreise sind immer zusammenhängende Graphen.

Ein gerichteter Graph G heißt Eulergraph, wenn es eine Anordnung (Permutation) aller seiner Kanten gibt, die einen geschlossenen Weg in G bildet.

Satz 13
Ein zusammenhängender gerichteter Graph $G = (V,E)$ ist genau dann ein Eulergraph, wenn für alle Knoten $v \in V$: $Indegree(v) = Outdegree(v)$ gilt.

Ein ungerichteter Baum $T = (V,E)$ kann leicht in einen gerichteten Baum $T' = (V,E')$ überführt werden, indem man jede Kante $e = (u,v) \in E$ durch zwei gerichtete Kanten $\{< u,v >, < v,u >\} \subset E'$ ersetzt. Dann ist T' ein Eulergraph.

ungerichteter Graph gegeben als Menge von Adjazenzlisten, $< v, L[v] > \in V \times \mathfrak{P}(V)$, wobei $L[v]$ die Nachbarn von v sind.

todo: Grafik vom Eulergraph einfügen

z. B. $L[1] =< 2,3,4 >$

Eulerkreis wird als Menge von Kanten dargestellt.

Beispiel für Eulerkreis:

$$< 1,2 > \rightarrow < 2,5 > \rightarrow < 5,2 > \rightarrow < 2,6 > \rightarrow < 6,2 > \rightarrow < 2,7 > \ldots < 3,1 > \rightarrow < 1,4 > \rightarrow < 4,1 >$$

Ziel: parallele Berechnung des Eulerkreises mit $S(< u,v >) =< v,w >$ (Nachfolger einer Kante)

Die Nachbarn $L[v]$ von $v \in V$ seien durchnummeriert: $L[v] =< u_0, \ldots, u_{d-1} >$. Ein Knoten v hat genau dann den Grad d, wenn er d Nachbarn hat: $|L[v]| = d$.

Satz 14
Die Funktion S: $E' \rightarrow E'$ beschreibt einen Eulerkreis in $T' = (V,E')$

$$S(< u_i, v >) :=< v, u_{(i+1) \bmod d} > \tag{4.2}$$

Beispiel 6
Setzen das Gewicht einer Kante vom Vater zum Sohn $w(< p(v), v >) := +1$ und vom Sohn zum Vater $w(< v, p(v) >) := -1$

Damit lässt sich das Level eines jeden Knoten bei Wahl eines Knotens als Root in $O(\log n)$ TIME mit $O(n)$ WORK berechnen. (Vor.: Rooting geht in diesen Grenzen und Eulertour geht so)

Beispiel 7
ROOTING: Eulertour-Technik, alle Kanten erhalten Gewicht 1 + Parallel Prefix

Der Vorgänger eines Knoten hat ein kleineres Gewicht

Beispiel 8
POST-Order, PRE-Order w(<v,p(v)>)=1, w(<p(v),v>)=0

Ergebnis: O(1) TIME mit O(n) WORK kann eine Eulertour berechnet werden

äquvalent: Angabe der Nachfolgerfunktion s: $s(e) = e'(e,e' \in E')T = (V,E), T' = (V,E')$

Vor.:

adj(v) = <u_0, …, u_{d-1}> Liste aller Pfade, die von v zu u_i führen.

adj als Ringliste implementieren.

Ringlisten für Standardgraph:

1	
2	(1,) (5,) (6,) (7,Zeiger auf (2,) in Liste 7)
7	(2,) (8,) (9,)
8	(7,)

Beipiel: wähle $u_i = 7$, v=2; bestimme den Index j: L[v] = <$u_1, \ldots, u_j, \ldots >$

Damit lässt sich der Nachfolger in O(1) berechnen.

4.4 Baumkontraktion

Was ist eine Harke (engl. Rake)? Entfernen des Vaters und eines Sohnes, der andere Sohn wird mit dem Großvater verbunden.

Ziel: Schnelle parallele Auswertung arithm. Ausdrücke (+,·)

Ausgehend von einem Binärbaum sollen nur noch 3 Knoten übrig bleiben, d.h. der Restbaum wird entfernt und die Auswertung des Baumes kann auf den 3 Restknoten (Vater, linker und rechter Sohn) erfolgen, was in einem Schritt geht. Es ergibt sich also das Problem, dass die Informationen der entfernten Knoten im Restbaum gespeichert werden müssen, damit die arithmetische Auswertung kein falsches Ergebnis liefert.

Von unten nach oben führt in zweitem Fall zu O(n), was keine Verbesserung gegenüber dem Seriellen darstellt.

Input : Binärbaum T, jeder innere Knoten hat genau 2 Söhne und für alle Knoten v: parent(v) und brother(v)

Output : Reduktion auf Wurzel und linkestes und rechtestes Blatt

1 markiere alle Blätter von A (Array der Blätter) (ohne linkestes und rechtestes) wachsend von links nach rechts; über parallel Prefix mit gewicht w(v) = 0 wenn v innerer Knoten, w(v)=1 wenn v Blatt
2 **for** $\lceil \log(n+1) \rceil$ **do**
3 | RAKE auf $A_{odd} = \{v_i : iodd\}$, die linke Söhne sind
4 | RAKE auf Rest von A_{odd}
5 | $A := A_{even}$;
6 **end**

Algorithmus 4.4: Algorithmus: Baumkontraktion

Satz 15
Die Kontraktion funktioniert auf einer EREW-PRAM in $O(\log n)$ TIME und $O(n)$ WORK.

BEWEIS:

- Es gibt m Blätter $\rightarrow$ nach einer Iteration $\lfloor \frac{m}{2} \rfloor$ Blätter $\rightarrow$ nach $\lceil \log(n+1) \rceil$ Iterationen fertig.
- Zeile 1 mit Eulertour in $O(\log n)$ TIME
- Zeilen 3 und 4 mit Eulertour in $O(1)$ TIME und $O(n)$ WORK
- Zeile 5 in $O(1)$ TIME
- Anzahl der Operationen: $O(|A|)$, $A_{even} \leq \frac{|A|}{2}$, $O(\sum_i \frac{n}{2^i}) = O(n)$ ■

help: RAKE-Beispiel aus Vorlesung (21.5.07) einfügen

Satz 16
Bei gegebenem Binärbaum, der Konstanten in den Blättern hält und '+' und '*' in den inneren Knoten, kann die Auswertung in $O(\log n)$ TIME und $O(n)$ WORK erfolgen. Denn: Ummarkierung bei RAKE in $O(1)$ TIME; Baumkontraktion erfüllt diese Schranken

Bemerkung 12
Gegeben sei ein Wurzelbaum T, der in den Knoten Zahlen enthält. Dann kann das Maximum / Minimum (oder jede andere angewendete binäre Operation) in $O(\log n)$ TIME optimal gewonnen werden.

BEWEIS:
Umformung in Binärbaum und Baumkontraktion mit Markierung ■

4.5 Das LCA-Problem

LCA steht für "lowest common ancestor"(jüngster gemeinsame Vorfahre) und bezieht sich auf den tiefsten Knoten in einem Binärbaum, der Vater der Teilbäume ist, in denen die Knoten u und v vorkommen. Das Problem ist ein Preprocessing-Problem, welches den Baum so vorbereitet, dass die Anfrage LCA(u, v) möglichst schnell beantworten werden kann (man besteht auf $O(1)$ TIME).

INPUT: Wurzelbaum T OUTPUT: Baum, für den einzelne LCA-Anfragen schnell beantwortet werden können

Man unterscheidet 2 Fälle, für die LCA-Anfragen trivial sind:

1. T ist ein einfacher Pfad
2. T ist ein vollständiger Binärbaum

Achtung: Der Algorithmus im Ja'Ja ist falsch! In der Praxis ist dieser Umstand lange nicht aufgefallen, da der Algorithmus wohl nur für den Spezialfall angewendet wird, für den er zufälligerweise richtig ist. Bisher gab es wohl keine Neuauflage des Ja'Ja und es ist nicht bekannt, ob der Fehler an den Autor bzw. den Verleger weitergereicht worden ist. Der Ansatz ist aber wohl brauchbar.

Algorithmus von Ja'Ja: Der Algorithmus hat als INPUT zwei Knoten u und v, die entsprechend ihrer INORDER-Nummer innerhalb des Baums bezeichnet sind. Laut Ja'Ja wird der $LCA(u,v)$ so ermittelt, dass die Knotennummern (binär) von links nach rechts durchlaufen werden und an der ersten sich unterscheidenden Stelle wird eine '1' gesetzt, der Rest ist '0'.

Gegenbeispiel: Man nehme zwei Knoten, deren Nummerierung $code(u) = 1100$ und $code(v) = 1101$ ist. Da die Nummerierung wie erwähnt in INORDER erfolgt, ist 1100 der Vater von 1101, jedoch würde der Ja'Ja-Algorithmus die letzte Stelle als die erste sich unterscheidende Stelle erkennen, diese 1 setzen und den Wert $'1101'$ ausgeben; das ist falsch. Die Lösung des Problems erfolgt mittels des Algorithmus von Spillner.

Vereinbarung: Sei v ein Knoten und l(v) das linkeste Vorkommen im Level- bzw. Eulerarray. Analog r(v) als rechtestes Vorkommen.

Lemma 3
Sei T=(V,E) ein Wurzelbaum Eulerarray A, Levelarray B, r(v) und l(v) seien definiert seien u und v zwei verschiedene Knoten Dann gilt:

1. u ist Vorfahre von v $\Leftrightarrow$ l(u) < l(v) < r(u)
2. u,v sind nicht direkt verwandt $\Leftrightarrow$ r(u) < l(v) oder r(v) < l(u)

Input : u, v
Output : $LCA(u,v)$
1 Berechne k_u, k_v: letzte '1' von links
2 **while** *Durchlaufe die Ziffern von u, v von links nach rechts* **do**
3 | Falls k_u oder k_v noch nicht erreicht, aber Ja'Ja hat Ergebnis, dann OUTPUT Ja'Ja
4 | Falls vor k_u erreicht, dann OUTPUT bis k_u, Rest '0'en
5 | analog k_v
6 **end**

Algorithmus 4.5: Algorithmus von Spillner

3. falls r(u)<l(v), so ist LCA (u,v) der Knoten im Eulerarray mit minimalen Level im Intervall [r(u), l(v)] im Levelarray

BEWEIS:
(1) $\rightarrow$ Eulertour entspricht Tiefensuche, Beginn ist Wurzel. Es folgt: u wird vor v besucht; ausserdem wird der ganze Teilbaum mit Wurzel v komplett durchsucht, bevor u ein letztes Mal besucht wird. $\leftarrow$ Annahme: u sein kein Vorfahre von v. Da l(u) < l(v), wird der Teilbaum mit Wurzel u komplett durchsucht, bevor v durchsucht wird $\rightarrow$ auch r(u) < l(u) $\rightarrow$ Widerspruch zu Voraussetzung! (2,3) Ähnlich. ■

Folgerung 1
Ist Levelarray B bekannt und können wir das Range-Minima-Problem lösen, haben wir die LCA-Bestimmung gelöst!

4.6 Das Range-Minima-Problem

Wir beginnen mit der Bestimmung der Eulertour von T (setzen voraus, dass der Baum in einer wohl-verarbeitbaren Form vorliegt) und erhalten sowohl das Eulerarray A als auch das Levelarray B.

Zusätzlich kennen wir für jeden Knoten v l(v) und r(v) (linkestes und rechtestes Vorkommen von v im Eulerarray).

Zur Bestimmung des Range-Minimas reicht es nicht, die Minima der Blätter der Teilbäume in den Knoten zu speichern!

Definition 13
Die Präfixminima eines Arrays $(c_1, \ldots, c_m)$ sind die Werte $(e_1, \ldots, e_m)$ mit $e_i = min\{c_1, \ldots, c_i\}$

Idee: Speichere in jedem Knoten die Suffix- und Präfixminima
- Sei $v = LCA(b_i, b_j)$
- Allgemein: Teilbaum mit Knoten v enthält Blätter $\{b_r, \ldots, b_i, \ldots, b_j, \ldots, b_s\}$
- Teilung der Folge in linken und rechten Teil: $\{b_r, \ldots, b_i, \ldots, b_p\}, \{b_{p+1}, \ldots, b_j, \ldots, b_s\}$
Suchen: Minimum(Minumum($\{b_i, \ldots, b_p\}$), Minimum($\{b_{p+1}, \ldots, b_j\}$))
Vorteil dieses Mal: Die Minuma liegen am Rand. Min($b_i, \ldots, b_p$) ist ein Suffix-Minimum, Min($b_{p+1}, \ldots, b_j$) ist ein Präfix-Minimum

Input : $B^n, n = 2^l$
Output : vollständiger Binärbaum mit den Arrays P,S

```
1  for j:=1 to n pardo
2  |  P(0,j) := B(j)
3  |  S(0,j) := B(j)
4  endfor
5  for h:=1 to log(n) do
6  |  for j:=1 to n/2^h pardo
   |  |  /* h ist das Level von den Blättern an (h=0)        */
7  |  |  Merge(P(h-1,2j-1) mit P(h-1,2j) zu P(h,j)
8  |  |  Merge S analog
9  |  endfor
10 end
```

Algorithmus 4.6: Rang-Min

Entscheidend: wie funktioniert das mischen ...
Analyse: TIME: $O(\log n)$ WORK: $O(n \log n)$ ($O(n)$ pro Stufe)

Satz 17
Das Preprocessing für Rang-Minima-Problem ist $O(\log n)$ TIME mit $O(n \log n)$ WORK.

Bemerkung 13
Im Seriellen gilt: optimal in O(n) Zeit sind LCA und Range-Minima lösbar. (im Sinne der Einzelanfrage in $O(1)$ TIME)

Folgerung 2
Satz 17 ist nicht optimal.

Im Parallelen gilt: LCA geht in $O(\log n)$ TIME optimal Range-Minima geht in $O(\log \log n)$ TIME optimal
Aufgabe: Range-Minima *optimal* in $O(\log n)$ TIME
Standardtechnik:

1. Zerlege B in Blöcke gleicher Länge $\log n$
2. Preprocessing mit seriellem Algorithmus parallel für alle Blöcke
3. berechne für jeden Block das Minimum $x_i (i = 1, \ldots, \frac{n}{\log n})$ und die Präfix- und Suffix-Minima innerhalb der Blöcke
4. wende den Algorithmus Range-Minima auf Array B'=$(x_1, \ldots, x_{\frac{n}{\log n}})$

Analyse:

1. $O(\log n)$ TIME, $O(n)$ WORK
2. $O(\log n)$ TIME, $O(n)$ WORK

3. $O(\log n)$ TIME, $O(n)$ WORK
4. $O(\log n)$ TIME, $O(\frac{n}{\log n}\log n) = O(n)$ WORK

Algo:
$MIN(b_i,\ldots,b_j) = MIN(\text{Suffixmin}_{B_{s-1}}(b_i), \min(x_s,\ldots,x_t), \text{Präfixmin}_{B_{t+1}}(b_j)$
Ergebnis: Satz * geht auch mit O(n) WORK.

Folgerung 3
Die analogen Schranken gelten für LCA.

5 Suchen, Mischen, Sortieren

5.1 Suchen in einer sortierten Menge nach Kruskal

Lemma 4
Mit k Schritten lässt sich ein Element in einem sortierten Feld mit n Elementen finden, wenn $n = (p+1)^k - 1$ ist und p Prozessoren eingesetzt werden. Abbildung 5.1

BEWEIS:
Der Beweis erfolgt induktiv.

$k = 1$**:** Dann ist $n = (p+1)^1 - 1 = p$. Ein Feld von n Elementen mit $p = n$ Prozessoren zu durchsuchen geht in einem Schritt.

$k-1 \to k$**:** Verteilt man die p Prozessoren gleichmäßig auf die n Elemente, so dass sich $p+1$ Abschnitte der Länge $\frac{n+1}{p+1}$[1] ergeben, und ordnet dem j. Prozessor ($j = 1, \ldots, p$) den j. Abschnitt zu, kann jeder Prozessor das letzte Element $j\frac{n+1}{p+1}$ in seinem Abschnitt untersuchen.

1. Fall: Das Element ist das gesuchte $\to$ fertig.

2. Fall: Wenn das Element nicht gefunden wird, hat man aber einen Abschnitt bestimmt, in dem sich das Element befinden muss. Da in dem Abschnitt bereits ein Element untersucht wurde, verbleiben noch $\frac{n+1}{p+1} - 1$ Elemente.

$$\frac{n+1}{p+1} - 1 = \frac{(p+1)^k - 1 + 1 - p - 1}{p+1} = (p+1)^{k-1} - 1$$

In diesem Abschnitt lässt sich nach Induktionsvorraussetzung in $k-1$ Schritten das Element finden. Also lässt sich das Feld in $1 + (k-1) = k$ Schritten durchsuchen.■

[1] Ergebnis der Kombinatorik

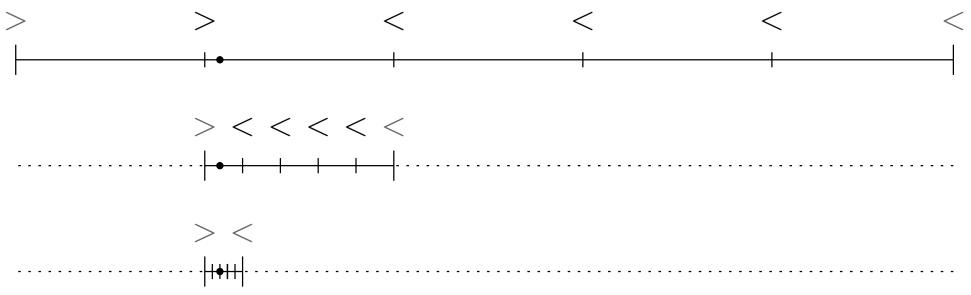


Abbildung 5.1: parallele Suche nach Kruskal in $n = 124$ Elementen mit $p = 4$ Prozessoren

Satz 18 (Satz von Kruskal (1984))
Um eine Folge von n Elementen mit p Prozessoren zu durchsuchen, sind $T_p(n)$ Schritte notwendig und hinreichend.

$$T_p(n) = \Theta\left(\frac{\log(n+1)}{\log(p+1)}\right)$$

BEWEIS:

hinreichend: Nach Lemma 4 kann man bei einer geschickten Platzierung der Prozessoren ein Feld von $(p+1)^k - 1$ Elementen mit k Schritten durchsuchen. Für die Zeit $T_p(n)$ (Anzahl der Schritte k) gilt also $T_p(n) = O\left(\frac{\log(n+1)}{\log(p+1)}\right)$.

$$\begin{aligned} n \leq (p+1)^k - 1 \quad &\Leftrightarrow \quad \log(n+1) \leq k\log(p+1) \\ &\Leftrightarrow \quad \frac{\log(n+1)}{\log(p+1)} \leq k \\ &\Leftrightarrow \quad k = \left\lceil \frac{\log(n+1)}{\log(p+1)} \right\rceil \end{aligned}$$

notwendig: Wenn man die Prozessoren beliebig positioniert, bleibt nach dem ersten Suchschritt mindestens ein Abschnitt mit mehr als $\frac{n+1}{p+1} - 1$ nicht untersuchten Elementen übrig. Positioniert man in diesem die Elemente wieder beliebig, so bleicht mindestens ein Abschnitt, der größer ist als

$$\frac{\overbrace{\frac{n+1}{p+1} - 1}^{\text{Elem. aus 1. Schritt}} \quad +1}{\underbrace{p+1}_{\text{Anz. der Abschnitte}}} - 1 = \frac{n+1}{(p+1)^2} - 1$$

Induktiv kann man also zeigen, dass nach k Schritten mehr als $\frac{n+1}{(p+1)^k} - 1$ Elemente nicht untersucht wurden. Damit keine nicht untersuchten Elemente mehr verbleiben, das Feld also komplett durchsucht ist, sind mindestens $\frac{\log(n+1)}{\log(p+1)}$ Schritte notwendig. Die Laufzeit ist also $T_p(n) = \Omega\left(\frac{\log(n+1)}{\log(p+1)}\right)$

$$\frac{n+1}{(p+1)^k} - 1 \leq 0 \quad \Leftrightarrow \quad n+1 \leq (p+1)^k \quad \Leftrightarrow \quad \frac{\log(n+1)}{\log(p+1)} \leq k \qquad \blacksquare$$

5.2 Mischen

todo: In diesem Abschnitt fehlen sehr viele Bilder, die das Vorgehen wesentlich anschaulicher machen würden. Einarbeiten.

Das Thema Mischen war bereits Gegenstand des Abschnitts über Partitioning (Abschnitt 3.4). Dabei wurden für die Operation $Rang(x : A)$ die serielle, binäre Suche eingesetzt. Mit der neuen Idee von Kruskal für die parallele Suche lässt sie das Mischen optimal in $O(\log\log n)$ TIME (statt $O(\log n)$ TIME mit binärer Suche) lösen.

Input : Ein aufsteigend sortiertes Feld A von n Zahlen und ein gesuchtes Element x
Output : Index i, so dass $A[i] \leq x < A[i+1]$

```
1  beg := 1
2  end := n
3  for k := 1 to log(n) do
4  |  for j := 1 to p pardo
5  |  |  len := ende − beg + 1  pos := beg + j · (len+1)/(p+1)
6  |  |  if A[pos] < x then
7  |  |  |  if j = p then
8  |  |  |  |  beg := pos + 1
9  |  |  |  end
10 |  |  else if x < A[pos] then
11 |  |  |  if j = 1 then
12 |  |  |  |  end := pos − 1
13 |  |  |  else
14 |  |  |  |  A[beg + (j − 1)(len+1)/(p+1)] < x
15 |  |  |  end
16 |  |  |  beg := (j − 1)(len+1)/(p+1) + 1
17 |  |  |  end := pos − 1
18 |  |  else                                        /* A[pos] = x */
19 |  |  |  return j
20 |  |  end
21 |  endfor
22 end
```

Algorithmus 5.1: Parallele Suche in einer sortierten Folge nach **Kruskal**

Satz 19
Für die Bestimmung des Rangs $Rang(X:Y)$ einer Folge X mit m Gliedern bezüglich einer größeren, sortierten Folge Y mit n Gliedern ($n > m = \Theta(n^s)$ für $0 < s < 1$) benötigt man $O(1)$ TIME und $O(n)$ WORK.

BEWEIS:
Für ein Element aus X kann der Kruskel-Algorithmus mit $p = \lfloor \frac{n}{m} \rfloor$ Prozessoren durchgeführt. Dies geht in $O(1)$ TIME.

$$\frac{\log(n+1)}{\log(p+1)} = \frac{\log(n+1)}{\log(\lfloor \frac{n}{m} \rfloor + 1)} = \frac{\log(n+1)}{\log(\lfloor n \cdot n^{-s} \rfloor + 1)} \leq \frac{\log(n+1)}{\log(n^{1-s})} = \frac{1}{1-s} \cdot \frac{\log(n+1)}{\log n}$$

Verwendet man $n = m \cdot \frac{n}{m}$ Prozessoren, kann man für die m Elemente von X in $m \cdot O(1) = O(1)$ TIME mit $W(n) = m \cdot O(\frac{n}{m}) = O(n)$ WORK den Rang bezüglich Y bestimmen. ∎

Input : A^n und B^m, wobei $\sqrt{m}$ ganzzahlig
Output : $Rang(B:A)$

```
1  if m < 4 then                                          /* B zu kurz */
2  |  trivial mit Kruskal, da p = n
3  |  return
4  end
5  J[√m + 1] := 0
6  for i := 1 to √m pardo
7  |  J[i] := Rang(B[i·√m] : A) mit √n Prozessoren
8  |  B_i := B[i·√m + 1, ..., (i+1)·√m − 1]
9  |  A_i := A[J[i] + 1, ..., J[i+1]]
10 |  if J[i] = J[i+1] then
11 |  |  Rang(B_i : A_i) = (0, 0, ...)
12 |  else
13 |  |  berechne mit diesem Algorithmus R = Rang(B_i : A_i) rekursiv
14 |  |  return ∀k: i·√m < k < (i+1)·√m ist Rang(B[k] : A) := J[i] + R[k − i·√m + 1]
15 |  end
16 endfor
```

Algorithmus 5.2: Rangbestimmung einer Folge bezüglich einer wesentlich größere Folge

Satz 20
Für zwei sortierte Folgen A mit n Gliedern und B mit m Gliedern, wobei $m \leq n$ ist, geht $Rang(B:A)$ in $O(\log\log m)$ TIME mit $O((n+m)\log\log m)$ WORK.

BEWEIS:
Die (worst case) Laufzeit für den gesamten Algorithmus sei $T(n,m)$.

Der erste Schritt (Zeile 2 in Algorithmus 5.2) geht in $O(1)$ TIME mit $O(n)$ WORK, also ist $T(n,3) = O(1)$.

Der zweite Schritt (Zeile 7 in Algorithmus 5.2) zur Bestimmung von $Rang(B[i \cdot \sqrt{m}] : A)$ verwendet $\sqrt{n}$ Prozessoren und benötigt nach Satz 18 $O\left(\frac{\log(n+1)}{\log(p+1)}\right) = O(1)$ TIME und $O(\sqrt{m} \cdot \sqrt{n}) = O(n+m)$ WORK.

$$\frac{\log(p \cdot p+1)}{\log(p+1)} \leq \frac{\log(p \cdot (p+1))}{\log(p+1)} \leq \frac{\log(p+1)+\log(p+1)}{\log(p+1)}$$
$$\sqrt{m} \cdot \sqrt{n} \leq \left(\sqrt{\max\{n,m\}}\right)^2 \leq \max\{n,m\} + \min\{n,m\} = n+m$$

Der zweite Schritt zerlegt A in Teile A_i $(i = 1, \ldots, \sqrt{m})$ der Länge $n_i := |A_i|$. In diese Teile können unabhängig voneinander jeweils die entsprechenden Teile von B einsortiert werden, da die Elemente aus einem B_i nicht in einem A_k $(k \neq i)$ liegen können – die Pfeile überschneiden sich nicht.

Die Zeit für den dritten Schritt $Rang(B_i : A_i)$ (Zeile 13 in Algorithmus 5.2) ist $T(n_i, \sqrt{m})$ TIME, da diese alle parallel ausgeführt werden, ist

$$T(n,m) \leq O(1) + \max_i T(n_i, \sqrt{m}) = O(\log\log m)$$

da man nach k rekursiven Rufen die Menge auf $m^{2^{-k}}$ Elemente reduziert hat. Zum Schluss hat man eine konstante Anzahl an Elementen $(c \leq 3)$, die in $T_p(n,3) = O(1)$ eingegliedert werden. Also gilt:

$$m^{(\frac{1}{2})^k} = m^{2^{-k}} \leq c \Leftrightarrow \log\log m - \log\log c \leq k$$

Der Aufwand ergibt sich aus der Arbeit im zweiten Schritt und der Anzahl der rekursiven Aufrufe: $O\big((n+m)\log\log m\big)$ WORK. ∎

Folgerung 4
Zwei sortierte Folgen A und B mit je n Elementen können in $O(\log\log n)$ TIME mit $O(n\log\log n)$ WORK gemischt werden.

Leider ist das Ergebnis noch nicht WT-optimal, da das Mischen im seriellen Fall in $T^*(n,m) = O(n+m)$ TIME geht. Zur Vereinfachung sei im Folgenden immer $n = m$.

todo: Der Rest dieses Abschitts ist wahrscheinlich falsch.

Mit Accellerated Cascading können wir auch hier zu einem optimalen Algorithmus kommen.

1. Zerlege A und B in $\frac{n}{\log\log n}$ Abschnitte der Länge $\log\log n$: Die Abschnitte seien A_i und B_i.
2. Erzeuge aus den Elementen an den Abschnittsgrenzen zwei neue Folgen A' und B'.
3. Bestimme $Rang(A' : B')$ und $Rang(B' : A')$ – damit hat man bestimmt, in welchen Abschnitt der anderen Folge die jeweiligen Grenzelemente müssen. Zum Beispiel ergibt sich, dass $B'[j]$ zwischen $A'[k]$ und $A'[k+1]$ kommt. $A'[k]$ und $A'[k+1]$ ist der Abschnitt A_k
4. Man kennt also zu jedem Grenzelement von A einen Abschnitt B_j der Länge $\log\log n$, in den man das Element liegen muss. Innerhalb dessen kann man die genaue Position seriell mit binärer Suche bestimmen. Ebenso kann man in die umgekehrte Richtung für die Grenzelemente von B ihre genaue Position bestimmen.

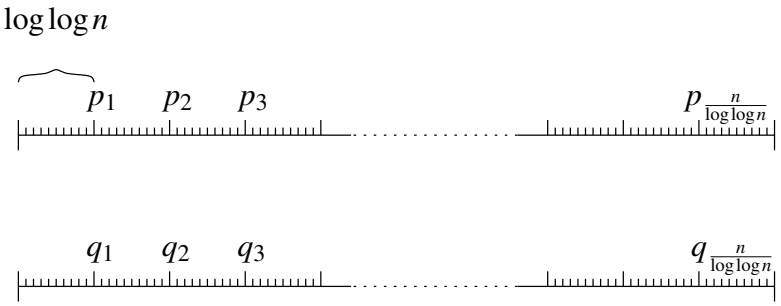


Abbildung 5.2: todo: ausfüllen
todo: noch die Bilder für die anderen Schritte zeichnen

5. Damit hat man A und B in $2 \cdot \frac{n}{\log\log n}$ Abschnitte der Größe $\leq \log\log n$ zerlegt und kennt jeweils den korrespondierenden Abschnitt in der anderen Folge. Diese Abschnitte kann man jetzt paarweise und unabhängig ineinander ordnen.

Bilder zum Vorgehen: Abbildung 5.2
Zeit- und Aufwandsanalyse:

Schritt	TIME	WORK	Bemerk.
1.+2.	$O(1)$	$O(1)$	
3.	$O\left(\log\log \frac{n}{\log\log n}\right) = O(\log\log n)$	$O\left(\frac{n}{\log\log n} \cdot \log\log \frac{n}{\log\log n}\right) = O(n)$	Satz 20
4.	$O(\log\log\log n) = O(\log\log n)$	$\frac{n}{\log\log n} \cdot O(\log\log\log n) = O(n)$	
5.	$O(1)$	$\frac{n}{\log\log n} \cdot O(\log\log n) = O(n)$	Satz 20

1. Zerlege die Folgen A, B in $\sqrt{n}$ Abschnitte der Länge $\sqrt{n}$.

2. Erstelle aus den Abschnittsgrenzen zwei neue Folgen A' und B'.

3. Für jedes Element aus A' mit $\sqrt{n}$ Prozessoren bestimme $Rang(A' : B')$. Ebenso für $Rang(B' : A')$.

4. Bestimme $Rang(A' : B)$

Bemerkung 14
Die Klasse P der in polynomeller Zeit lösbaren Algorithmen zerfällt für parallele Ansätze in drei Klassen:

- $O(1)$, z. B. Eulertour-Technik
- $O(\log n)$, vielzahl der Probleme
- $O(\log\log n)$, z. B. Merge

5.3 Sortieren

Baumparadigma Ansatz
- Umwandlung in Binärbaum
oBdA

1. Binärbaum
2. Innere Knoten haben genau 2 Söhne
3. In Blättern stehen Einerlisten

Richard Cole:

Vereinbarung 1
v: Vaterknoten u: linke Sohn w: rechter Sohn

Trivial gilt: Sortieren geht mit der Idee des Baumparadigmas („merge-sort“) in $O(\log n \log\log n)$ TIME (optimal).

BEWEIS:
Merge in $O(\log\log n)$ TIME optimal ■

5.3.1 Merge with the help of a cover

c-Decke $c \in \mathbb{Z}$

Definition 14
sortierte Folge X heiße c-Decke einer sortierten Folge Y, wenn Y höchstens c Elemente zwischen zwei aufeinanderfolgenden Elementen der Folge $X_\infty := (-\infty, X, \infty)$ enthält

Satz 21
Seien A^n, B^m sortiert, sei X c-Decke von A^n und B^m.
Wenn der Rang(X:A) und Rang(X:B) bekannt sind, so kann Merge(A, B) in O(1) TIME und $O(|X|)$ WORK erhalten werden.

BEWEIS:
X = $(x_1, \ldots, x_s)$, Rang(X:A)=$(r_1, \ldots, r_s)$, Rang(X:B)=$(t_1, \ldots, t_s)$
Da c-Decke: $|B_i|, |A_i| \leq c$ ■

Sei T unser Baum, v ein Knoten von T, $Level(v)$, Höhe $h(T)$, Altitude $alt(v) := h(T) - Level(v)$
Algorithmus arbeitet in Schritten s.

Definition 15
$L[v]$: Lister der sortierten Elemente der Söhne
$L_s[v]$: Liste im Knoten v nach Schritt s, Ziel: $L_s[v] = L[v]$ für $s \geq 3 \cdot alt(v)$
v *voll* im Schritt s, wenn $L_s[v] = L[v]$

Definition 16
v heißt *aktiv* im Schritt $s :\Leftrightarrow alt(v) \leq s \leq 3 \cdot alt(v)$

Folgerung 5
Wir bekommen für root:

$$alt(root) = h(T) \leq s \leq 3 \cdot alt(root) = 3 \cdot h(T) \tag{5.1}$$

Root ist damit voll nach $3 \cdot h(T)$ Schritten.

5.3.2 Optimales Sortieren - Pipeline Merge Sort

Hier widmen wir uns der Herleitung eines optimalen Sortieralgorithmus, der mit $O(logn)$ TIME auskommt. Allerdings betrachten wir hier ein allgemeineres Problem, dessen Lösung uns den optimalen Algorithmus liefert.

Formulierung des allgemeineren Problems: sei T ein Binärbaum, für den jedes Blatt u eine Liste $A(u)$ enthält, die aus einer geordneten Menge stammt. Problem sei, für jeden inneren Knoten v die sortierte Liste $L(v)$ der in im Teilbaum mit Wurzel v gespeicherten Elemente (die Liste $A(u)$ eines Blattes u kann auch leer sein).

Wir beginnen mit einer Menge von Transformationen: jedes Blatt u wird durch einen balancierten Binärbaum mit $|A(u)|$ Blättern ersetzt, so dass jedes Element von $A(u)$ in den Blättern des Binärbaums abgelegt ist. Die Höhe des Baumes T ist demnach nun um $O(log(max_u|A(u)|))$ gewachsen, nur beinhaltet nun jedes Blatt von T wenigstens ein Element.

Die zweite Transformation erzwingt, dass für jeden inneren Knoten zwei Kinder existieren. Ist dies nicht der Fall, so wird ein Blatt, welches kein Element beinhaltet, angefügt.

Definition 17
Sei L sortierte Liste. Das c-Raster $Raster_c(L)$ (c-sample) ist die sortierte Teilliste aus jedem c-ten Element.

Sei $L = (l_1, l_2, \ldots)$ eine sortierte Liste, so ist $Raster_c(L) = (l_c, l_{2c}, \ldots)$.

Beispiel 9

$$L = (1,3,5,6,7,8,11,13,14,15,16) \tag{5.2}$$

$$Raster_4(L) = (6,13) \tag{5.3}$$

$$Raster_2(L) = (3,6,8,13,15) \tag{5.4}$$

$$Raster_1(L) = L \tag{5.5}$$

Die früher vorgestellte Strategie für paralleles Merge-Sort basiert auf einer Vorwärtstraversierung des Binärbaums, so dass für alle Knoten v in einer speziellen Höhe h die Liste $L(v)$ komplett bestimmt ist, bevor die Berechnung der Knoten bei Höhe $h+1$ beginnt.

Die Pipeline-Strategie besteht aus der Bestimmung der $L(v)$ über eine Anzahl von Schritten, so dass bei Schritt s $L_s(v)$ eine Approximation von $L(v)$ ist, die im Schritt $s+1$ verbessert wird. Gleichzeitig wird ein Raster von $L_s(v)$ im Richtung Wurzel propagiert, um dort für Annäherungen der Listen auf eben anderen Leveln genutzt zu werden.

Kommen wir nun dazu, wie man die $L_s(v)$ präzise bestimmt ...
Sei $L_0(v) = \emptyset$, wenn v ein innerer Knoten ist bzw. $L_0(v)$ enthält das Element, was im Blatt v gespeichert ist.

Definition 18
Sei $alt(v) = height(T) - level(v)$, wobei $level(v)$ als Abstand von v zur Wurzel definiert ist.

Die Liste, die in den internen Knoten v gespeichert wird, wird in den Schritten s aktualisiert, wenn die Bedingung $alt(v) \leq s \leq 3alt(v)$ erfüllt ist. Ist dies der Fall, dann sprechen wir davon, dass v in diesem Schritt s aktiv ist. Der Algorithmus aktualisiert die Liste $L_s(v)$, so dass v gefüllt wird, d.h. $L_s(v) = L(v)$ - wenn also $s \geq 3alt(v)$. Somit ist klar, dass nach $3height(T)$ Schritten der Wurzelknoten voll ist und die inneren Knoten die sortierten Listen enthalten.
Bevor wir zum Algorithmus kommen, führen wir eine weitere Notation ein.

Definition 19
Für einen beliebigen Knoten x:

$$Sample(L_s[x]) := \begin{cases} Raster_4(L_s[x]) & s \leq 3 \cdot alt(x) \\ Raster_2(L_s[x]) & s = 3 \cdot alt(x) + 1 \\ Raster_1(L_s[x]) & s = 3 \cdot alt(x) + 2 \end{cases} \tag{5.6}$$

Also ist $Sample(L_s[x])$ die Teilliste bestehend aus jedem vierten Element von $L_s[x]$, bis es voll ist. Anschließend ist $Sample(L_s[x])$ die Teilliste aus jedem zweiten Element im Schritt $3alt(x)+1$ und schließlich die vollständige Liste im Schritt $3alt(x)+2$.

Input : $\forall v : L_s[v]$ v voll, wenn s$\geq$ 3 alt(v)
Output : $\forall$ v: $L_{s+1}[v]$, v voll, wenn s+1$\geq$ 3alt(v)
1 **for** $\forall$ *aktiven Knoten v* **pardo**
2 $\quad L'_{s+1}[w] = Sample(L_s[w])$
3 $\quad L'_{s+1}[u] = Sample(L_s[u])$
4 $\quad Merge(L'_{s+1}[u], L'_{s+1}[w])$ zu $L_{s+1}[v]$
5 **endfor**

Algorithmus 5.3: Algorithmus von Richard Cole

Hinweis: es fehlen an dieser Stelle zwei Vorlesungen. Diese beinhalteten 180 Minuten Beweise und Lemmas zum Richard-Cole-Algo und sind nicht Prüfungsrelevant. Es reicht ein Verständnis dessen, was passiert. Natürlich absolut inoffiziell. Es sei hiermit auf Ja'Ja (Seiten 164 ff) verwiesen.

5.4 Selektion

Input: $A = (a_1, \ldots, a_n)$, $k := 1 \leq k \leq n$
Output: Element a_i mit $Rang(a_i : A) = k$

Lösung: SORT A $\Rightarrow O(logn)$ TIME mit $O(nlogn)$ WORK nicht optimal, da man seriell mit $O(n)$ WORK auskommt.

Ziel: CREW (EREW, optional): $O(lognloglogn)$ TIME, $O(n)$ WORK; nutzen R.Cole und Accelerated Cascading

Median der Mediane-Technik; R. Cole; Prefixsummenalgo.; Litheratur liefert u. a. 2 Ansätze: Akl (s. Info 3), R.Cole

Akl: 0<x<1: O(n^x) TIME optimal

Fälle k=1,n. $O(\log\log n)$ CRCW, optimal WORK $O(n)$ Median: nicht erreichbar. Bem. dazu: $par_n(\alpha_1,\ldots,\alpha_n) := \alpha_1 \oplus \alpha_2 \oplus \ldots \oplus \alpha_n$ (α_i boolsche Werte)

Die Berechnung von par_n erfordert auf der PRIORITY-CRCW-PRAM $\omega(\frac{\log n}{\log\log n})$ TIME. (Beweis schwer!) Die gleiche Schranke gilt für den Median.

Input : $A = (a_1,\ldots,a_n), k : 1 \leq k \leq n$
Output : $ldots$

1 $n_0 := n$
2 $s := 0$
3 **while** $n > \frac{n}{logn}$ **do**
4 $s := s+1$
5 zerlege A in Blöcke B_i mit je log n Elementen; Berechne den Median m_i für Block B_i parallel $\Rightarrow O(logn)/O(n_s)$
6 Berechne den Median der Mediane: $m \Rightarrow O(logn)/O(n)$
7 Bestimme die Anzahlen s_1, s_2, s_3 der Elemente von A, die kleiner, gleich oder größer als m sind (mit Präfixsumme in $O(logn)$ TIME optimal)
8 **if** $s_1 < k \leq s_1 + s_2$ **then**
9 OUTPUT m
10 **end**
11 **if** $k \leq s_1$ **then**
12 kontrahiere die Elemente von A, die <m sind, in aufeinander folgende Positionen
13 $s := s_1$
14 **end**
15 **if** $k > s_1 + s_2$ **then**
16 kontrahiere ...
17 $n_s = s_3$
18 $k := k - (s_1 + s_2)$
19 **end**
20 **end**
21 SORT die restlichen $n_s \leq \frac{n}{logn}$ Elemente, gib das k-te Element aus

Algorithmus 5.4: Algorithmus: Parallele Selektion

n_s... Zahl der Elemente nach Schritt s

Lemma 5
$n_{s+1} \leq \frac{3n_s}{4}$; Beweis mittels Median-der-Mediane-Argument

Folgerung 6
fertig mit Schritt 3 nach $O(loglogn)$ Iterationen
WORK: $\sum_{s=1}^{\infty} O(n_s) = O(n)$
TIME: $O(lognloglogn)$

6 Parallelisierbarkeit

Laut dem Satz von Brent (Satz 5) kann jeder parallele Algorithmus in $O\left(\frac{W(n)}{p} + T(n)\right)$ TIME realisiert werden. Jedoch ist es in vielen Fällen nicht sinnvoll, mehr als $\alpha(n)$ Prozessoren zu verwenden, da dies keine Verbesserung bringt. Um z. B. ein Element in 100 Zahlen zu suchen, ist der Einsatz von mehr als 100 Prozessoren nicht sinnvoll.

Man kann die Laufzeitverbesserung auf zwei Wegen angehen. Wenn mehr Prozessoren eine bessere Zeit liefern, so ist natürlich die Erhöhung der Prozessoranzahl das Mittel der Wahl. Der andere Weg ist einen anderen Algorithmus zu finden, der das Problem schneller löst.

Definition 20
Die Menge NC[1] ist die Klasse der Sprachen, die sich in polylogarithmischer[2] Zeit mit polynomiell[3] vielen Prozessoren auf einer PRAM entscheiden[4] lassen.

$$\mathrm{NC} = \{L \subset \Sigma^* : \forall w \in \Sigma^n \text{ ist } \chi_L(w) \text{ in } O\left((\log n)^k\right) \text{ TIME} \\ \text{mit } O(n^k) \text{ Prozessoren (für festes } k\text{) auf einer PRAM berechenbar}\}$$

Eine Sprache L heißt genau dann **hochgradig parallelisierbar** oder NC-**berechenbar**, wenn $L \in \mathrm{NC}$.

Definition 21
Seien L_1 und L_2 zwei Sprachen. L_1 heißt NC-**reduzierbar** auf L_2 (Schreibweise: $L_1 \preceq_{\mathrm{NC}} L_2$), wenn es eine hochgradig parallelisierbare Funktion $f\colon \Sigma^* \to \Sigma^*$ gibt, so dass für alle $w \in \Sigma^*$ gilt: $w \in L_1$ dann und nur dann, wenn $f(w) \in L_2$

Definition 22
Eine Sprache L heißt genau dann P-**vollständig**, wenn L in polynomieller Zeit auf einer deterministischen Turingmaschine berechenbar ist ($L \in \mathrm{P}$) und alle Sprachen $L' \in \mathrm{P}$ NC-reduzierbar auf L sind: $L' \preceq_{\mathrm{NC}} L$.

Satz 22
Alle Probleme, die sich in polylogarithmischer Zeit mit polynomiell vielen Prozessoren auf einer PRAM berechnen lassen, können in polynomieller Zeit auf einer deterministischen Turingmaschine berechnet werden: $\mathrm{NC} \subset \mathrm{P}$.

[1] NC: Nick's class, nach seinem Erfinder Nick Pipenger benannt
[2] **polylogarithmisch**: $O\left((\log n)^k\right)$ für ein festes k
[3] **polynomiell**: $O(n^k)$ für ein festen k
[4] Eine Sprach $L \subset \Sigma^*$ heißt genau dann **entscheidbar**, wenn sich ihre **charakteristische Funktion** χ_L[5] berechnen lässt
[5] $\chi_L(w) = 1$, wenn $w \in L$, sonst $\chi_L(w) = 0$

BEWEIS:
über serielle Modellierung der Rechnung todo: machen ■

Satz 23
Wenn $L_1 \preceq_{\mathrm{NC}} L_2$ und $L_2 \preceq_{\mathrm{NC}} L_3$, dann ist auch $L_1 \preceq_{\mathrm{NC}} L_3$ (Transitivität).

BEWEIS:
Hintereinanderausführung zweier NC-berechenbarer Funktionen; trivial ■

6.1 P-vollständige Probleme

6.1.1 Maximaler Fluss in einem Netzwerk

Ein **Netzwerk** $N = (V, E, s, t, c)$ ein gerichteter Graph, dessen Kanten $e \in E$ eine Kapazität $c(e) \in \mathbb{N}_0$ zugeordnet ist. Der Knoten $s \in V$ wird als Startknoten und der Knoten $t \in V$ als Zielknoten bezeichnet.

Der **Fluss** $f\colon E \to \mathbb{N}$ über eine Kante e ist durch ihre Kapazität beschränkt: $0 \leq f(e) \leq c(e)$. Als **Zufluss** $f^+(v)$ eines Knotens $v \in V$ bezeichnet man die Summe über den Fluss aller eingehenden Kanten und als **Abfluss** $f^-(v)$ die Summe über den Fluss aller ausgehenden Kanten. Für alle Knoten außer dem Start- und dem Zielknoten soll der Zufluss gleich dem Abfluss sein.

$$f^+(v) := \sum_{e \in E\colon e=(*,v)} f(e) \qquad\qquad f^-(v) := \sum_{e \in E\colon e=(v,*)} f(e)$$

Als den Wert $\mathrm{val}(f_N)$ eines Flusses f durch ein Netzwerk bezeichnet man die Summe $\mathrm{val}(f) = f^+(s) - f^-(s) = f^-(t) - f^+(t)$.

Definition 23
Das Problem „**maximaler Fluss**“ ist die Frage nach dem größtmöglichen Fluss $\widetilde{f}$ durch ein Netzwerk, so dass für alle anderen möglichen Flüsse f gilt: $\mathrm{val}(\widetilde{f}) \geq \mathrm{val}(f)$.

Im Seriellen existieren Algorithmen mit $O(n^3)$ TIME, die dieses Problem lösen, jedoch ist kein paralleler Algorithmus mit $O(log^k n)$ TIME bekannt, der mit polynomiell vielen Prozessoren auskommt. Es wird zudem angenommen, dass es einen solchen Algorithmus generell nicht gibt; einen Beweis dafür, dass Max-Flow nicht in NC liegt, hat bisher noch niemand erbracht.

6.1.2 Lineare Optimierungsprobleme

Definition 24
Als ein **lineares Optimierungsproblem** bezeichnet man die Aufgabe, zu einer gegebenen Matrix $A \in \mathbb{R}^{n \times m}$ und zwei Vektoren $b \in \mathbb{R}^m, c \in \mathbb{R}^n$ einen Vektor $x \in \mathbb{R}^n$ zu finden, so dass $c^T x$ maximal ist, wobei $x_i \geq 0$ und $Ax \leq b$ gilt.

$$\max\{c^T x\colon Ax \leq b, 0 \leq x\}$$

help: Warum ist das jetzt in P \ NC?

6.1.3 Das circuit value problem

Definition 25
Als einen **Schaltkreis** $c = \langle g_1, \ldots, g_n \rangle$ (engl. **circuit**) bezeichnet man eine Menge von **Gattern** $g_1, \ldots, g_n$, die entweder ein Eingabegatter mit den möglichen Werten $\{0,1\}$ sind oder eine boolsche Verknüpfung von einem oder zwei anderen Gattern realisieren, d. h. für $1 \leq j,k < i \leq n$ ist $g_i = \neg g_j$ oder $g_i = g_j \vee g_k$ oder $g_i = g_j \wedge g_k$.

Definition 26
Als **circuit value problem** (*CVP*) bezeichnet man das Entscheidungsproblem, ob für einen gegebenen Schaltkreis $c = \langle g_1, \ldots, g_n \rangle$ der Wert des letzten Gatters $g_n = 1$ ist.

$$CPV(c) = \begin{cases} 0 & g_n = 0 \\ 1 & g_n = 1 \end{cases}$$

6.2 P-Vollständigkeit von CVP

Um zu zeigen, dass das circuit value problem P-vollständig ist, muss man für jede beliebige Sprache $L \in \mathrm{P}$ zeigen, dass sie sich auf *CVP* reduzieren lässt ($L \preceq_{NC} CVP$) und dass $CVP \in \mathrm{P}$ liegt.

Beweisskizze $CVP \in \mathrm{P}$? Klar, linear über die g_i laufen und mitrechnen; Entscheidung für $g_n = 1$ in $O(n)$ TIME mit $O(n)$ WORK.

Behauptung Sei $L \in \mathrm{P}$ und $L \preceq_{\mathrm{NC}} CVP$

Da $L \in \mathrm{P}$ liegt, gibt es eine Turingmaschine M, die nach polynomiell vielen Schritten eine Eingabe w genau dann akzeptiert, wenn $w \in L$, indem sie eine $1 \in \Gamma$ in die erste Zelle schreibt. Die **Turingmaschine** M ist ein Sextupel $(Q, \Sigma, \Gamma, \delta, q_1, q_s)$ mit dem Arbeitsalphabet Γ, dem Eingabealphabet $\Sigma \subset \Gamma$, der Zustandsmenge Q, wobei $q_1 \in Q$ der Startzustand ist, und der Zustandsüberführungsfunktion $\delta \colon Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$.

Zu zeigen: es existiert eine NC-berechenbare Funktion f: wenn I_L ein Input-Beispiel für L ist, so ist $f(I_L) = I_{CVP}$ ein Input-Beispiel für CVP:
$I_L \in L \Leftrightarrow Wert(f(I_L)) = Wert(I_{CVP}) = 1$

Damit muss man einen NC-berechenbaren Algorithmus angeben, der für eine Turingmaschine M einen Schaltkreis c konstruiert, der genau dann durchschaltet ($g_n = 1$), wenn M die Eingabe akzeptiert.

Einige Hilfsfunktionen für die Konstruktion der Gatter:

$$H(i,t) = \begin{cases} 1 & \text{der Lese-Schreib-Kopf steht zum Zeitpunkt } t \text{ auf Zelle } i \\ 0 & \text{sonst} \end{cases}$$

$$C(i,j,t) = \begin{cases} 1 & \text{das Zeichen } a_j \in \Gamma \text{ steht zum Zeitpunkt } t \text{ in Zelle } i \\ 0 & \text{sonst} \end{cases}$$

$$S(k,t) = \begin{cases} 1 & M \text{ befindet sich zum Zeitpunkt } t \text{ im Zustand } q_k \\ 0 & \text{sonst} \end{cases}$$

Modellierung der Startkonfiguration ($t = 0$) ist

$$H(i,0) = \begin{cases} 1 & i = 1 \\ 0 & i > 1 \end{cases}$$

$$C(i,j,0) = \begin{cases} 1 & \text{Zelle } i \text{ enthält Zeichen } a_j \\ 0 & \text{sonst} \end{cases}$$

$$S(k,0) = \begin{cases} 1 & k = 1 \\ 0 & k > 1 \end{cases}$$

Das letzte Gatter sei $g_n = C(1,1,T(n))$, d. h. g_n ist genau dann eins, wenn die Turingmaschine nach $T(n)$ Schritten eine Eins in die erste Zelle geschrieben hat.

todo: Das ganze muss nochmal überarbeitet werden. Professor Hecker hat dazu auch eine Kopie des Beweises aus dem Ja'Ja ausgeteilt. Es sollte noch erklärt werden, dass die Gatter für H in $O(1)$ TIME mit $O(T^2(n))$ WORK, die Gatter für C in $O(1)$ TIME mit $O(T^2(n))$ WORK und die Gatter für S in $O(\log n)$ TIME mit $O(T^2(n))$ WORK erzeugt werden können. Die Erzeugung des Schaltkreises geht also in polylogarithmischer Zeit mit polynomiell vielen Prozessen → der Algorithmus ist NC-berechenbar. http://www.informatixx.de/privat/files/informatik/pdf/Parallele_Algorithmen_Komplett.pdf Seite 58

In $T(n)$ Schritten kann die Turingmaschine auf höchstens $T(n)$ Zellen des Arbeitsbands zugreifen: $i \in \{1,\ldots,T(n)\}$. Für die Zeit t gilt: $t \in \{0,\ldots,T(n)\}$.

Mittels δ beschreiben wir $t \to t+1$

induktiv ...

6.3 DFS-Theorem (geordnete Tiefensuche)

Theorem: DFS ist P-vollständig

INPUT Digraph, Kanten markiert mit $1,2,3,\ldots$, Start s
OUTPUT Knotenfolge beginnend bei s entsprechend der DFS- Reihenfolge

todo: Der Abschnitt muss auch nochmal überarbeitet werden.

INPUT: beginnend bei a OUTPUT:

Ecken	DFS-Liste
a	1
b	2
c	5
d	3
e	4
f	6
g	7

Formulieren wir ein Problem „Kommt e vor d in DFS-Liste“ so haben wir damit ein Entscheidungsproblem – ein Sprachproblem.

relevante Entscheidungsproblem: – wenn das Ursprungsproblem lösbar ist, dann auch das relevante Entscheidungsproblem. Wenn das relevante Entscheidungsproblem unlösbar ist, so erstrecht das Ursprungsproblem.

Mit dem letzten Satz haben wir gezeigt, dass CVP schwer entscheibar ist. Diese Aussagen wollen wir jetzt nutzen, um zu zeigen, dass auch DFS schwer ist.

Um zu zeigen, dass DFS P-vollständig ist, verwendet man den folgenden Satz 24 und zeigt, dass $CVP \preceq_{NC} DFS$ und $DFS \in$ P. Da $CVP \in$ P folgt aus Satz 24, dass $DFS \notin$ NC ist.

Satz 24 (Hauptsatz über die Reduktion)
Seien $L_1, L_2 \in \Sigma^*$ zwei Sprachen und L_1 sei NC-reduzierbar auf L_2. Dann ist $L_1 \in$ NC, wenn $L_2 \in$ NC ist: $L_1 \preceq_{NC} L_2 \wedge L_2 \in \text{NC} \Rightarrow L_1 \in \text{NC}$.

BEWEIS:
Da L_1 NC-reduzierbar auf L_2 ist, existiert eine Funktion $f \in$ NC, die jede Eingabe $w \in \Sigma^*$ für χ_{L_1} in eine Eingabe für χ_{L_2} transformiert, wobei $w \in L_1 \Leftrightarrow f(w) \in L_2$. Da L_2 NC-berechenbar ist, lässt sich die charakteristische Funktion χ_{L_2} in polylogarithmischer Zeit mit polynomiell vielen Prozessoren auf einer PRAM berechnen.

Die charakteristische Funktion von L_1 lässt sich also mit Hilfe der Funktion f und der charakteristischen Funktion χ_{L_2} in polylogarithmischer Zeit mit polynomiell vielen Prozessoren auf einer PRAM berechnen. L_1 ist also $\in$ NP.

$$\chi_{L_1}(u) = \chi_{L_2}\big(f(u)\big)$$ ∎

Wir zeigen, dass CVP leichter ist als DFS. Jedoch ist DFS kein Entscheidungspoblem, was eine direkte Abbildung der INPUT-Mengen aufeinander nicht möglich macht. Daher konstruieren wir ein Entscheidungsproblem, das leichter ist als DFS, an dem wir aber zeigen können, dass es schwerer ist als CVP. Damit wäre dann auch gezeigt, dass DFS schwerer ist, als CVP und damit auch schwer entscheidbar ist.

Dafür brauchen wir noch folgenden Satz:

Satz 25 (Satz über Transitivität)
Die Relation $\preceq_{\text{NC}}$ ist transitiv, d. h. wenn $L_1 \preceq_{NC} L_2$ und $L_2 \preceq_{NC} L_3$, so ist auch $L_1 \preceq_{NC} L_3$.

BEWEIS:
todo: machen Sollte einfach sein. $L_1 \preceq_{NC} L_2 \Rightarrow \exists f \in$ NC, $L_2 \preceq_{NC} L_3 \Rightarrow \exists g \in$ NC, dann ist $f \circ g \in$ NC. ∎

DFS: INPUT: gerichteter Graph, Kantenmarkierung (genormt), Startknoten, Traversierung im Graph soll möglich sein OUTPUT: DFS-Liste

zu konstruieren mit NC-Zuordnung.

Jedem INPUT von CVP: $I_{CVP} \rightarrow f(I_{CVP}) = I_{DFS}(v,w)$: Wert(I_{CVP}) $= 1 \Leftrightarrow$ das relevante Entscheidungsproblem von I_{DVS} wahr ist (=1 ist) (d. h. u wird vor v besucht.

$I_{CVP} :< g_1, g_2, \ldots, g_n >$ entweder $g_i = 1 \vee g_i = \neg(g_i \wedge g_k), k, j < i \rightarrow$ Graph, s,u,v

1. Fall $g_i = 1 \Rightarrow$ Abb.1 g_i sei INPUT für $g_{j_1}, \ldots, g_{j_k}$, bilde <i,j_1>,...,<i,j_k>

   Ind.beweise: Wenn wert=1 $\Leftrightarrow$ so wird s(i) vor t(i) besucht

2. Fall $g_i = \neg(g_j \vee g_k) \Rightarrow$ Abb.2

   Ind.beweis: Sei Wert(g_i)=1 $\Rightarrow (g_j = 0 = g_k)$ Ind.vor. $\Rightarrow$ t(j) vor s(j) besucht (t(k) vor s(k)) genauer: Abb.4-Verlauf: speziell: <j,i>, <k,i> sind nicht besucht $\Rightarrow$ für g_i wird Abb.5 verfolgt $\Rightarrow$ s(i) vor t(i)

   sei Wert(g_i)=0 $\Rightarrow$ (g_j=1 oder g_k=1)

   a) $g_j = 1 \rightarrow G_j$ (Graph, der g_i zugeordnet ist) wird nach Abb.5 durchlaufen $\Rightarrow$ „Zackenweg“

      $\Rightarrow$ in Abb.3 ist <j,i> bereits besucht $\Rightarrow$ für i $\Rightarrow$ Weg wie Abb.4

   b) $g_j = 0, g_k = 1 \Rightarrow$ im wesentlichen Weg wie in Abb.4 (Ausnahme: erst <j,i> dann zurück)

   $\Rightarrow$ t(i) vor s(i)

Definition 27

u:=s(n), v=t(n) $\Rightarrow$ u vor v besucht $\Leftrightarrow$ wert(g_n)=1

Index

CVP, 62
NC, 60
NC-berechenbar, 60
NC-reduzierbar, 60
P-vollständig, 60
TIME, 17
WORK, 17

Abfluss, 61
Ablaufplan, 9
Accelarated Cascading, 34
Allokation, 14
Arbeitsaufwand, 17

charakteristische Funktion, 60
circuit, 62
circuit value problem, 62
CRCW
    arbitrary, 12
    common, 12
    priority, 12
CREW, 12

DAG, 8
Divide and Conquer, 26

Effizienz, 18
entscheidbar, 60
EREW, 12

Fluss, 61
for-pardo, 14

Gatter, 62
globaler Speicher, 12
globalRead, 12
globalWrite, 12

hochgradig parallelisierbar, 60

INPUT, 63
Input, 57

konvex, 26
konvexe Hülle, 26
Kosten, 18
Kruskal, 51

lineares Optimierungsproblem, 61

maximaler Fluss, 61
Merge, 30
Mischen, 30

Netzwerk, 61
Netzwerkmodell, 10

optimal, 19
OUTPUT, 63
Output, 57

parallel Prefix, 25
Partitioning, 30, 31
path doubling, 24
Pipeline-Verfahren, 33
Pointer jumping, 24
polylogarithmisch, 60
polynomiell, 60
PRAM, 10
Präfixsumme, 21, 25

RAM, 8
Rang, 30
receive, 10
Rechenzeit, 17

Schaltkreis, 62
Schedule, 9
send, 10

Sequential Subset, 34
shared memory, 12
sortiert, 30
Speed-up, 17
streng optimal, 19
synchron, 12

Takt, 14
Teile und Herrsche, 26
Time-unit, 14
Turingmaschine, 62

Wald, 24
WT-optimal, 19

Zeiteinheit, 14
Zerlegunsstrategie, 30
Zufluss, 61