Skript zur Vorlesung
Parallele Algorithmen

Andreas Goerdt
Technische Universität Chemnitz
Fakultät Informatik
Theoretische Informatik

Wintersemester 1994

Das Skript ist eine etwas breiter dargestellte Version ausgewählter Teile des Buches

An indroduction to Parallel Algorithms

von **Joseph JáJá** aus dem **Addison – Wesley Verlag**.
Die Vorlesung war 3–stündig. Zusätzlich war eine Übungsstunde pro Woche.

Inhalt

1 Einführung

Wir befassen uns mit der Darstellung paralleler Algorithmen und der Messung ihrer Leistungsfähigkeit.

1.1 Darstellung paralleler Algorithmen

Wir stellen 3 Möglichkeiten der Darstellung paralleler Algorithmen vor:

- gerichtete azyklische Graphen
  (*dag*'s – directed acyclic graphs).
- (Programme für) parallele random–access–Maschinen.
  (PRAM's)
- (Programme für) Netzwerkarchitekturen.

Zunächst die gerichteten azyklischen Graphen. Das while–Programm

Eingabe: $n \in N$

Ausgabe: x

```
begin
  y := 1;
  x := 1;
  while y ≤ n do
    x := x × y;
    y := y + 1;
  end
end
```

berechnet auf x den Wert $n!$.

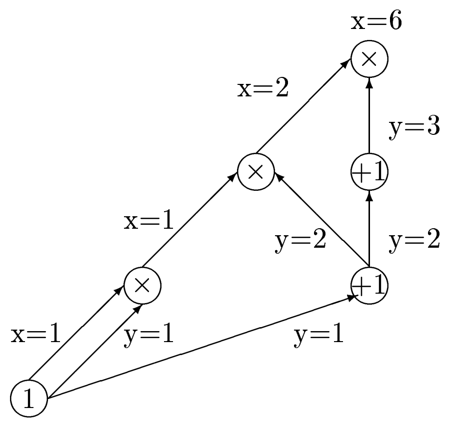
Ist ein Eingabewert gegeben, sagen wir $n = 3$, so läßt sich das Programm abwickeln. Wir bekommen die Sequenz von Zuweisungen

```
begin
y := 1;
x := 1;

x := x × y;
y := y + 1;
```

```
{ Hier ist y = 2. }
x := x × y;
y := y + 1;
{ Hier ist y = 3. }
x := x × y;
y := y + 1;
{ Hier ist y = 4 > n = 3. }
end
```

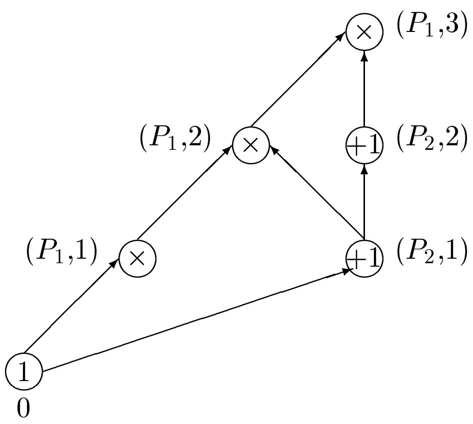
Diese Sequenz läßt sich als *dag* darstellen, der von unten nach oben abgearbeitet wird:



An den Knoten findet sich der jeweils berechnete Wert, der dann über die Kanten weitergeleitet wird. Man beachte: Der dag hängt von dem Eingabewert n ab: Man bekommt für jedes n einen anderen dag. D.h. unser Programm gibt uns eine Familie von dag's $(G_n)_{n \in N}$. Im allgemeinen Fall können wir uns die Graphen so konstruieren, daß Graphen für verschiedene Werte von n vollkommen unabhängig voneinander sind. Man sagt deshalb, die dag's stellen ein nicht–uniformes Berechenbarkeitsmodell dar. Man erkennt an dem dag, welcher Rechenschritt vor welchen anderen ausgeführt werden muß und welche Rechenschritte unabhängig voneinander ausgeführt werden können. Zum Beispiel sind die unterste Multiplikation und die unterste Addition mit 1 unabhängig, können also parallel ausgeführt werden.

Angenommen wir haben p Prozessoren $P_1, P_2, ..., P_p$ zur Verfügung, dann können wir jedem Knoten das dag's ein Paar $(P_i,\ t)$ zuweisen. Das bedeutet: Der Knoten soll zum Zeitpunkt t von Prozessor P_i berechnet werden. Eine solche Zuordnung heißt *Schedule* des dag's. Ein Schedule für den Beispiel–

dag ist:



Wir gehen hier also davon aus, daß P_1 auf die Daten, die P_2 berechnet vollkommen gleichberechtigt zugreifen kann und umgekehrt. Der wichtige Aspekt des parallelen Rechnens, die *Kommunikation*, wird hier gar nicht berücksichtigt.

Definition 1.1 (Schedule)

(a) Ein Schedule eines Berechnungsgraphen auf p Prozessoren $P_1, ..., P_p$ ist eine Abbildung

$$\mathcal{S} : \{\text{Nicht–Eingabeknoten des Graphen}\} \rightarrow \{(P_i,\ t) | 1 \leq i \leq p,\ t \in N\},$$

so daß für alle Knoten k_1, k_2 gilt:

- $\mathcal{S}(k_1) = (P_i,\ t),\ \mathcal{S}(k_2) = (P_j,\ t) \ \Rightarrow\ i \neq j$.
  Jeder Prozessor kann zu einem Zeitpunkt t nur einen Knoten berechnen.
- Ist $(k_1,\ k_2)$ eine Kante (also $k_1 \rightarrow k_2$) und $\mathcal{S}(k_1) = (P_i,\ t_1)$, $\mathcal{S}(k_2) = (P_j,\ t_2)$ so gilt $t_2 \geq t_1 + 1$.

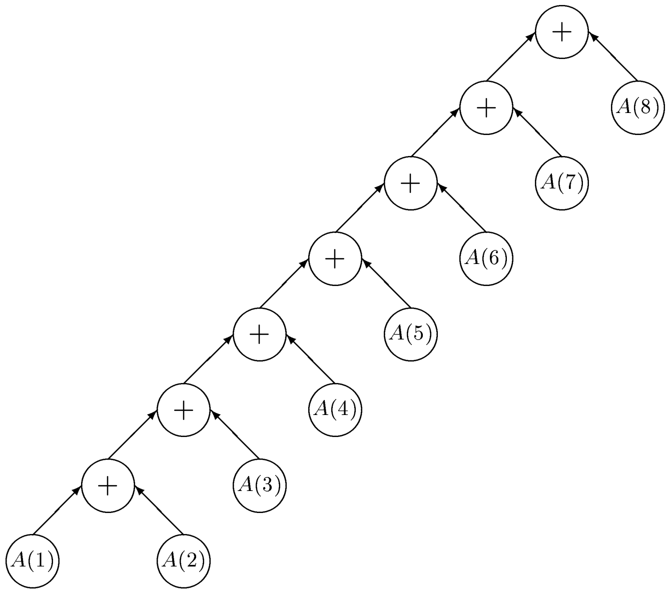
(b) Zeit von $\mathcal{S}$ = Max $\{\ t \mid$ es gibt einen Knoten k mit $\mathcal{S}(k) = (P_i,\ t)\}$.

(c) Die parallele Komplexität eines dag's auf p Prozessoren ist gegeben durch

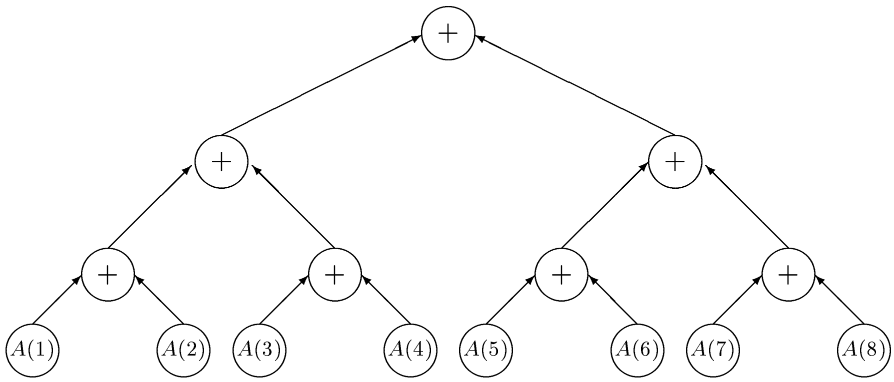
$$T_p = \text{Min } \{ \text{ Zeit von } \mathcal{S} | \mathcal{S} \text{ ist ein Schedule des dag's auf } p \text{ Prozessoren } \}$$

Man beachte: Die Länge (= # Kanten) des längsten Weges des dag's ist eine untere Schranke für T_p (unabhängig von p). □

Beispiel 1.2 Angenommen, wir wollen die 8 Elemente eines 8–dimensionalen Feldes A zusammenaddieren. Dann können 2 mögliche Vorgehensweisen durch folgende dag's dargestellt werden:



Hier gilt $T_p \geq 7$.



Hier ist $T_p \geq 3$.

Verallgemeinern wir diese Berechnungen auf beliebige Zweierpotenz n, dann gilt für den ersten dag:

$$T_p = n - 1 = \Omega(n)$$

Man beachte, daß T_p unabhängig von p ist, also sowohl für p = 1 als auch für p $\to \infty$ gilt. Im zweiten Fall gilt:

$$T_p = \log_2 n \quad \text{für } p \geq \frac{n}{2},$$

wogegen

$$T_1 = n - 1.$$

□

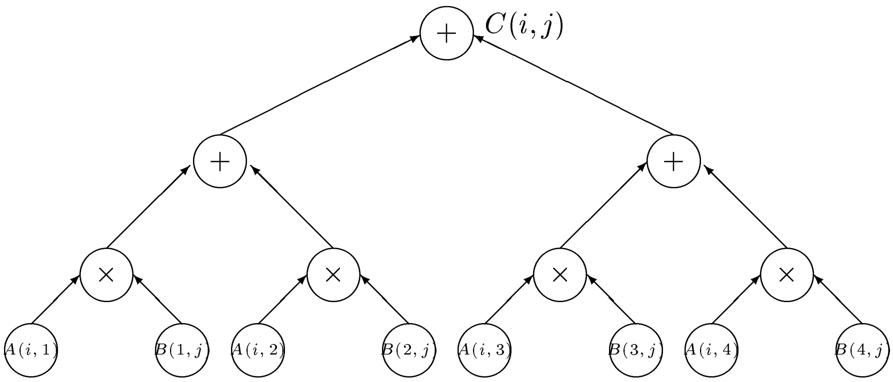
Beispiel 1.3 Seien A und B 2 $(n \times n)$–Matrizen.

$$A = \begin{pmatrix} A(1,1) & A(1,2) & \cdots & A(1,n) \\ A(2,1) & & & A(2,n) \\ \vdots & & & \vdots \\ A(n,1) & & \cdots & A(n,n) \end{pmatrix}, B = \begin{pmatrix} B(1,1) & B(1,2) & \cdots & B(1,n) \\ B(2,1) & & & B(2,n) \\ \vdots & & & \vdots \\ B(n,1) & & \cdots & B(n,n) \end{pmatrix}$$

Der Eintrag $C(i,j)$ des Produkts von A und B ergibt sich zu

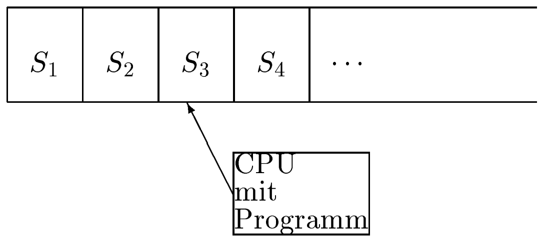
$$C(i,j) = \sum_{l=1}^{n} A(i,l) \times B(l,j).$$

Der folgende dag zeigt eine Berechnung von $C(i,j)$ für n = 4:



Um das Produkt $\mathcal{C} = A \times B$ zu berechnen, brauchen wir einen solchen dag für jedes $\mathcal{C}(i,j)$. Mit n Prozessoren kann jeder einzelne dag in Zeit $O(\log n)$ ausgewertet werden. Mit n^3 Prozessoren können wir alle dag's parallel und damit das Produkt in $O(\log n)$ Zeit berechnen. □

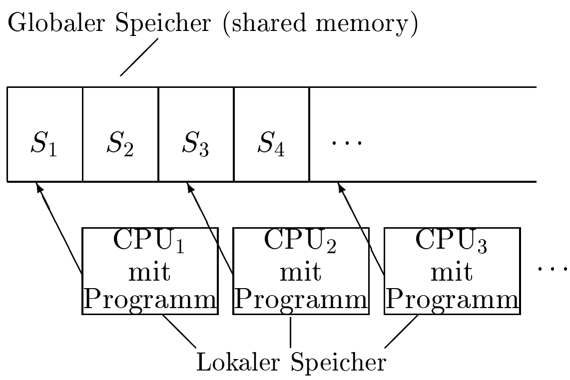
Wir erinnern uns, daß ein grundlegendes Modell zur Analyse sequentieller Algorithmen die random–access–Maschine (RAM) ist. Die RAM ist charakterisiert durch die Existenz einer CPU und eines random–Speichers, der aus unendlich vielen Speicherzellen besteht, deren Inhalt in einem Schritt von der CPU gelesen *und* geändert werden kann. Übliche Rechenoperationen lassen sich in einem Schritt ausführen.



S_i = Speicherplatz i

Im Gegensatz zur Turing–Maschine hat die RAM den Vorteil, daß sie zur Modellierung der Arbeitsweise eines Von–Neumann–Rechners geeignet ist. Das bei der Turing–Maschine nötige Hin– und Herfahren des Kopfes auf den Bändern hat keine Entsprechung beim Von–Neumann–Rechner und kommt bei der RAM nicht mehr vor.

Die parallele random–access–Maschine ist die direkte Erweiterung der RAM um weitere CPU's, die auf den gemeinsamen Arbeitsspeicher zugreifen:



Wir beachten schon jetzt, daß die Anzahl der CPU's einer PRAM unbegrenzt groß werden kann (was für die theoretische Behandlung günstiger ist). Im Gegensatz zur RAM ist die PRAM kein unbestritten geeignetes Modell paralleler Computer. Der simultane Zugriff mehrerer CPU's auf einen Speicher ist problematisch zu realisieren. In dem Modell ist es zulässig, daß *mehrere* CPU's in *einem* Schritt *dieselbe* Speicherzelle lesen oder in sie schreiben können. Technisch spricht man von einer shared–memory Architektur. Man spricht auch vom *globalen Speicher* (*shared–memory*) im Gegensatz zum *lokalen*, der zu einer einzigen CPU gehört. (Neben den shared–memory Architekturen gibt es noch die distributed–memory Architekturen, die uns aber nur am Rande beschäftigen.) Man beachte folgenden wichtigen Punkt: Die Kommunikation der CPU's untereinander führt *immer* über den globalen Speicher. Die CPU's sind nicht direkt miteinander verbunden.

Um mit der PRAM programmieren zu können, brauchen wir Befehle, die es erlauben, globale Daten, also Daten im Arbeitsspeicher, in den lokalen Speicher der CPU's zu holen, denn *nur dort* finden die eigentlichen Rechnungen statt. Dazu dienen die folgenden Befehle:

Syntax: global read (X,Y) und global write (U,V)

Semantik: Der Befehl global read (X,Y) liest den Datenblock mit Namen X aus dem globalen Arbeitsspeicher in die lokale Variable Y des lokalen Speichers ein. Der Befehl global write (U,V) lagert das lokale U auf das globale V aus.

Algorithmus 1.4 Sei A eine $(n \times n)$–Matrix und x ein Vektor der Dimension n. Wir nehmen an, A und x sind im globalen Speicher abgelegt. Wir haben $p \leq n$ Prozessoren $P_1, .., P_p$ in unserer PRAM, so daß $r = \frac{n}{p}$ ganzzahlig ist. Wir können also Matrix A in p Teile, die jeweils aus r Zeilen von A bestehen, aufteilen,

$$A = \begin{pmatrix} A_1 \\ A_2 \\ \vdots \\ A_p \end{pmatrix},$$

wobei A_1 aus Zeile 1 bis Zeile r besteht und A_i aus Zeile $((i-1) \cdot r) + 1$ bis Zeile $i \cdot r$ für $i \leq p$. Jedes A_i ist eine $(r \times n)$–Matrix. Wir setzen die Prozessoren $P_1, .., P_p$ folgendermaßen ein:

P_1 berechnet $A_1 \cdot x =$ die ersten r Elemente des Vektors $A \cdot x$,

P_2 berechnet $A_2 \cdot x =$ die zweiten r Elemente des Vektors $A \cdot x$,

$\vdots$

P_p berechnet $A_p \cdot x =$ die p–ten r Elemente des Vektors $A \cdot x$.

Weiterhin schreibt P_i den berechneten Vektor an die entsprechenden Stellen des Ergebnisvektors y im globalen Speicher. Der Prozessor P_i führt das folgende Programm aus:

Eingabe: Eine $(n \times n)$–Matrix A und ein n-dimensionaler Vektor x im globalen Speicher. Lokale Variablen sind folgendermaßen initialisiert:

- n enthält die Dimension n.
- i enthält die Nummer des Prozessors, auf dem das Programm abläuft. Jeder Prozessor kennt seinen Namen. Das werden wir ab jetzt als selbstverständlich voraussetzen.
- r enthält $r = \frac{n}{p}$.

Ausgabe: Die Elemente in den Zeilen $(i-1) \cdot r + 1, ..., i \cdot r$ des Vektors $A \cdot x$ in den Zeilen $(i-1) \cdot r + 1, ..., i \cdot r$ des globalen Vektors y.

```
begin
  1.global read (x, z);
  2.global read (A_i, B);
      { A_i = Zeile (i − 1) · r + 1 bis Zeile i · r von A. }
  3.w = B · z;
  4.global write (w, Element (i − 1) · r + 1 bis Element i · r des Vektors y );
end
```

□

Bevor wir weiter auf das Programm eingehen, wollen wir die Programmkonstrukte, die wir bei der Formulierung unserer Algorithmen gebrauchen, zusammenstellen.

- Zuweisung

  Syntax: Variable := Ausdruck

  Semantik: Erst wird die rechte Seite berechnet, dann wird der berechnete Wert der Variablen auf der linken Seite zugewiesen.

- begin–end statement

Syntax:

```
begin
    statement 1;
      ⋮
    statement n;
end
```

Semantik: Die statements werden der Reihe nach ausgeführt: Erst statement 1, dann statement 2, bis zu statement n.

- Fallunterscheidung

  Syntax: if Boolesche_Bedingung then statement
  else statement

  Semantik: Zuerst wird die Bedingung ausgewertet, dann das jeweilige statement.
  Man beachte: Bei geschachtelten Fallunterscheidungen wie:
  if b_1 then if b_2 then S_1 else S_2
  ist nicht eindeutig zu erkennen, was gemeint ist:
  if b_1 then (if b_2 then S_1) else S_2
  oder
  if b_1 then (if b_2 then S_1 else S_2).
  In solchen Fällen benötigen wir also eine zusätzliche Klammerung.

- Schleifen

  Syntax: for variable = initial_value to final_value do statement und
  while Boolesche_Bedingung do statement

  Semantik: Bei der for–Schleife wird das statement solange ausgeführt, bis variable > final_value ist. Der Wert von variable wird am Ende von statement automatisch um 1 erhöht. Die Semantik der while–Schleife ist wie üblich.

- **Exit**–Statement

  Syntax: exit

  Semantik: Dieses statement bewirkt das Ende des Programms bzw. das Herausspringen aus dem rekursiven Aufruf, in dem es ausgeführt wird.

- Schließlich haben wir noch die Kontrollstruktur der Rekursion und die schon erwähnten Lese- und Schreibbefehle.

Bemerkung 1.5

(a) Wir gehen i.a. davon aus, daß die PRAM *synchron* arbeitet: Alle Prozessoren arbeiten mit einer gemeinsamen Uhr. In jedem Schritt führt jeder Prozessor eine elementare Instruktion eines Programms aus. Im Programm von Prozessor P_i wissen wir zu jedem Zeitpunkt genau, was das Programm von P_j macht und schon gemacht hat. In Algorithmus 1.4 führen also alle Prozessoren gleichzeitig den ersten Befehl aus, gleichzeitig den zweiten Befehl aus und so weiter.

(b) Problematisch ist immer der Zugriff von mehreren Prozessoren *gleichzeitig* (*concurrent*) auf dieselbe Variable. Man unterscheidet:

- lesenden Zugriff (concurrent read),
- schreibenden Zugriff (concurrent write).

Im ersten Befehl unseres Algorithmus 1.4 haben wir ein concurrent read der globalen Variablen $\mathbf{x}$.

(c) Zum Rechenaufwand des Programms zu Algorithmus 1.4: Jeder Prozessor P_i führt in Schritt 3 $n \cdot r = \frac{n^2}{p}$ Multiplikationen und $(n-1)\frac{n}{p}$ Additionen, also insgesamt $O(\frac{n^2}{p})$ viele arithmetische Operationen durch. Da die Prozessoren synchron arbeiten, brauchen wir eine Zeit von $O(\frac{n^2}{p})$.

Wie messen wir die Zeit unserer Algorithmen auf random–access–Maschinen? Wir sagen: Jede Operation, die mit einer von vornherein endlichen Anzahl von Speicherworten arbeitet braucht *eine* Zeiteinheit. Solche Operationen sind: Lesen vom globalen Speicher, Schreiben in den globalen Speicher, Addieren zweier Worte, logisches **Und** zweier Worte, Multiplizieren, Dividieren zweier Worte usf. Wir gebrauchen das *uniforme Kostenmaß* oder auch Enheitskostenmaß. Das heißt, die tatsächliche Größe der Operanden geht nicht in die Betrachtung ein. Dieses Maß ist realistisch, solange die entstehenden Worte nur so groß sind, daß sie in ein Speicherwort passen. Geht die Größe des Operanden mit ein, spricht man vom *logarithmischen Kostenmaß.*

(d) Zum Kommunikationsaufwand von Algorithmus 1.4: Schritte 1 und 2 übertragen $n + n \cdot r = O(\frac{n^2}{p})$ viele Zahlen vom globalen Speicher in jeden lokalen. Schritt 4 überträgt $\frac{n}{p}$ Zahlen von jedem lokalen in den globalen Speicher.

(e) Man beachte, daß die Korrektheit von Algorithmus 1.4 *nicht* davon abhängt, daß die Prozessoren synchron arbeiten. Das liegt daran, daß kein Prozessor P_i mit von einem Prozessor P_j berechneten Werten weiterrechnet. □

Die Korrektheit des folgenden Algorithmus für die Mutliplikation von A mit x ist dagegen nur dann gewährleistet, wenn die Prozessoren synchron arbeiten.

Algorithmus 1.6 Wir teilen A und x folgendermaßen in Teilstücke $A_1, ..., A_p$ und $x_1, ..., x_p$ ein:

$$A = (A_1, A_2, ..., A_p), \quad x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_p \end{pmatrix}$$

A_i ist eine $(n \times r)$–Matrix, enthält also die Spalten $(i-1) \cdot r + 1$ bis $i \cdot r$ von A. Der Vektor x_i besteht aus den entsprechenden r Werten von x. Dann ist $A_i \cdot x_i$ eine $(n \times 1)$–Matrix und $A \cdot x = A_1 \cdot x_1 + \cdots + A_n \cdot x_n$. Mit einer PRAM können wir analog zu dem Algorithmus 1.4 die $A_i \cdot x_i$ sagen wir auf die Variable z_i parallel ausrechnen. Die Addition der z_i darf aber erst dann stattfinden, wenn $A_i \cdot x_i$ wirklich auf z_i berechnet ist! Man sieht an diesem Beispiel, wie mit einmal berechneten Werten weitergearbeitet wird.

□

Algorithmus 1.7 Wir haben ein Feld **A** mit $n = 2^k$ Zahlen und eine PRAM mit n Prozessoren $P_1, ..., P_n$ gegeben. Wir wollen die Summe $S = A(1) + A(2) + \cdots + A(n)$ berechnen. Prozessor P_i führt das folgende Programm aus:

Eingabe: Ein Feld **A** bestehend aus $n = 2^k$ Elementen im globalen Speicher der PRAM mit n Prozessoren.

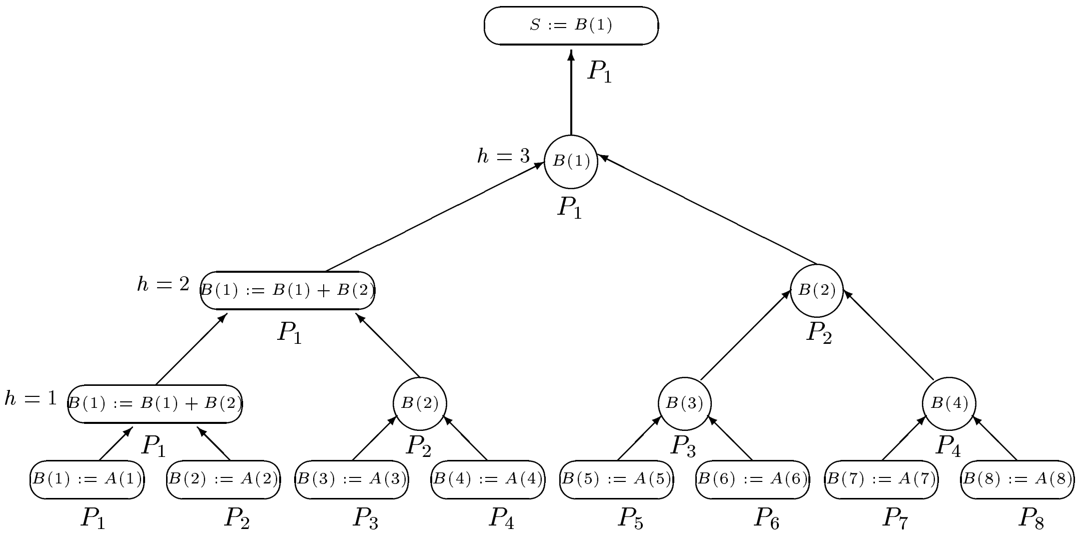
Ausgabe: Der Wert $A(1)+A(2)+\cdots+A(n)$ in dem globalen Speicherplatz **S** .

```
begin
  1.global read (A(i),a)
  2.global write (a,B(i))
  3.for h = 1 to log n do
```

```
      if 1 ≤ i ≤ n/2^h then
        { Beachte: i ist die Nummer des Prozessors,
        auf dem das Programm abläuft. }
        begin
          global read (B(2i - 1), x);
          global read (B(2i), y);
          z := x + y;
          global write (z,B(i));
        end
   4.if i = 1 then global write (z,S )
      { Obwohl jeder Prozessor Instruktion 4 in
      in einem Programm hat, schreibt nur Prozessor 1 }
end
```

□

Die Berechnung der Summe läßt sich durch folgenden binären Baum veranschaulichen (für $n = 2^3 = 8$):



Man sieht, daß $P_5, ..., P_8$ beginnend mit Schritt 3 gar nichts mehr machen.

In diesem Programm haben wir *keinen* concurrent Zugriff auf Variablen, weder lesend noch schreibend.

Wir betonen noch einmal die folgenden Charakteristika von PRAM–Algorithmen:

Shared–memory = globaler Speicher: Im Beispiel sind **A** und **B** allen Prozessoren gleichberechtigt zugänglich. (Man beachte, daß **a** eine lokale Variable ist, die für jeden Prozessor existiert.)

Synchrone Programmausführung In jedem Schritt führt jeder Prozessor genau ein Instruktion aus oder bleibt im Ruhezustand (d.h. ist idle).

In Abhängigkeit von den Möglichkeiten des Zugriffs der Prozessoren auf den globalen Speicher unterscheidet man folgende Arten von PRAM's:

- Die EREW–PRAM = exclusive read, exclusive write.
- Die CREW–PRAM = concurrent read, exclusive write.
- Die CRCW–PRAM = concurrent read, concurrent write.

Bei der CRCW–PRAM können Konflikte beim gleichzeitigen Schreiben in eine Variable auftreten: Was soll gemacht werden, wenn Prozessor i den Wert a in die globale Variable **x** schreiben will, Prozessor j dagegen b? Man unterscheidet 3 Möglichkeiten:

- Die *common* CRCW–PRAM erlaubt ein concurrent write nur dann, wenn die beteiligten Prozessoren alle den gleichen Wert schreiben wollen.
- Die *arbitrary* CRCW–PRAM löst Konflikte dadurch, daß einer (unbestimmt welcher) der am Konflikt beteiligten Prozessoren seinen Wert schreibt.
- Die *priority* CRCW–PRAM nimmt an, daß die Prozessoren total (=linear) geordnet sind. Konflikte werden dadurch gelöst, daß der kleinste Prozessor schreibt.

Man beachte, daß jedes Programm der EREW–PRAM auf der CREW–PRAM emuliert werden kann und jedes CREW–PRAM auf der (egal welcher) CRCW–PRAM.
Frage: Was passierte, wenn unser Algorithmus 1.7 lautete:

    4.write (z,**S**)

anstelle von

    4.if i = 1 then global write (z,**S**)?

$\square$

Bemerkung 1.8 Im weiteren Verlauf werden wir die global–read und global–write Befehle zur Vereinfachung aus den Programmen auslassen. D.h. also dann, ein Befehl im Programm des Prozessors P_i

```
A := B + C
```

wo **A, B, C** globale Variablen sind, steht als Abkürzung für:

```
global read (B,x);
global read (C,y);
z := x + y;
global write (z,A);
```

$\square$

Algorithmus 1.9 Wir wollen das Produkt C zweier $(n \times n)$–Matrizen A und B berechnen. Der Einfachheit halber sei $n = 2^k$ für eine ganze Zahl $k > 0$. Wir nehmen an, unsere PRAM hat n^3 Prozessoren zur Verfügung. Wir gehen davon aus, daß die Prozessoren numeriert sind als $P_{i,j,l}$, wobei $1 \leq i, j, l \leq n$ ist. Unser Programm ist so, daß jeder Prozessor $P_{i,j,l}$ das Produkt $A(i,l) \cdot B(l,j)$ in einem Schritt berechnet. Danach benutzen wir für jedes Paar (i,j) die Prozessoren $P_{i,j,l}$ mit $1 \leq l \leq n$ zur Berechnung der Summe $\sum_{l=1}^{n} A(i,l) \cdot B(l,j)$. Zur Erinnerung:

$$A \cdot B = \begin{pmatrix} \sum_l A(1,l) \cdot B(l,1) & \cdots & \sum_l A(1,l) \cdot B(l,n) \\ \vdots & & \vdots \\ \sum_l A(n,l) \cdot B(l,1) & \cdots & \sum_l A(n,l) \cdot B(l,n) \end{pmatrix}$$

Das Programm für Prozessor $P_{i,j,l}$ ist gegeben durch:

Eingabe: Zwei $(n \times n)$–Matrizen A und B im globalen Speicher, wobei $n = 2^k$. Lokale Konstanten sind a und (i,j,l), die Prozessornummern.

Ausgabe: Das Produkt $C = A \cdot B$ im globalen Speicher.

```
begin
  1.C ′ (i,j,l) := A(i,l) · B(l,j),
      { Die 3–dimensionale Matrix C ′ (i,j,l) ist im globalen Speicher }
  2.for h = 1 to log n do
      if 1 ≤ l ≤ n/2^k then C ′ (i,j,l) := C ′ (i,j,2l - 1) + C ′ (i,j,2l)
  3.if l = 1 then C(i,j) := C ′ (i,j,1)
end
```

Man beachte das concurrent read von **A(i,l)** von Prozessoren $P_{i,j,l}$ für alle j und von **B(l,j)** von $P_{i,j,l}$ für alle i. Der Algorithmus ist damit nicht für die EREW–PRAM geeignet, aber für die CREW–PRAM.

Der Algorithmus braucht $O(\log n)$ Schritte, da die anfänglichen Multiplikationen in einem Schritt ausgeführt werden. □

Bemerkung 1.10 Falls wir die 3. Instruktion ohne die Bedingung schreiben, d.h. einfach:

```
begin
  3.C(i,j) := C ' (i,j,1)
end
```

dann weist *jeder* Prozessor $P_{i,j,l}$ der Variablen C(i,j) den Wert C $'$ (i,j,1) zu. Man bräuchte also dann auf jeden Fall eine common CRCW–PRAM. □

Wir werden uns hauptsächlich mit PRAM–Algorithmen befassen. Zur Orientierung betrachten wir aber noch einen Typ paralleler Rechnermodelle, in denen sich der Kommunikationsaspekt parallelen Rechnens deutlicher widerspiegelt, als bei den PRAM's: das Netzwerkmodell. In diesem Modell haben wir keinen globalen Speicher mehr. Jeder Prozessor hat seinen eigenen lokalen Speicher. Ein Netzwerk ist ein ungerichteter Graph $G = (N, E)$. Dabei stellt jeder Knoten $i \in N$ einen Prozessor, der mit einem lokalen Speicher ausgestattet ist, dar. Jede Kante $(i, j) \in E$ stellt eine Verbindung zwischen den Prozessoren i und j dar.

Wir betrachten 3 typische Netztopologien:

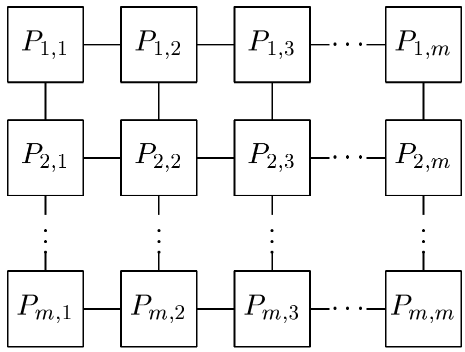
Definition 1.11 (3 Netztopologien)

(a) Ein lineares Feld ist ein Graph der Form

$$\boxed{P_1}\text{———}\boxed{P_2}\text{———}\boxed{P_3}\text{—}\cdots\text{—}\boxed{P_p}$$

Haben wir noch eine Kante von P_p nach P_1, so sprechen wir von einem Ring.

(b) Das zweidimensionale Gitter ist die zweidimensionale Version des linearen Feldes. Wir haben $p = m^2$ viele Prozessoren, die folgendermaßen angeordnet sind:

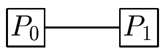


(c) Der Hypercube besteht aus $p = 2^d$ Prozessoren, die in einem d–dimensionalen Würfel miteinander verbunden sind. Wir bezeichnen die Prozessoren als:

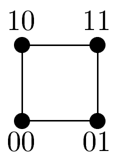
$$P_{0\cdots 0}, P_{0\cdots 1}, P_{0\cdots 10}, \cdots, P_{1\cdots 1}$$

Jeder Prozessor ist mit einer d–stelligen Binärzahl indiziert. Zwei Prozessoren sind miteinander verbunden, wenn sich ihr Index in genau einem Bit unterscheidet.

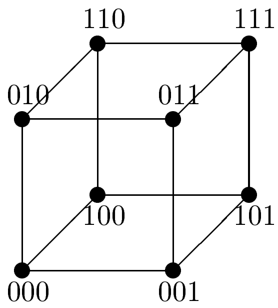
$d = 1$, dann



$d = 2$, dann

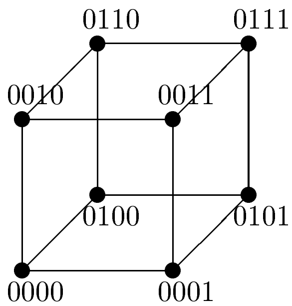


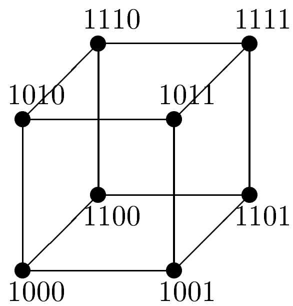
$d = 3$, dann



Das vordere Quadrat von Prozessoren ist eine Kopie des Würfels für den Fall d = 2, wobei das erste Bit, eine Null, hinzugefügt wurde. Das hintere Quadrat ist analog gebildet, wobei als erstes Bit eine 1 hinzugefügt ist. Von den Prozessoren, die sich in dem ersten Bit unterscheiden sind jeweils die mit dem gleichen Rest verbunden, also 010 mit 110 usf.

d = 4. Wir gehen jetzt nach beschriebenem Bildungsgesetz vor:





Durch Verallgemeinerung dieses Bildungsgesetzes kommen wir zu dem Hypercube für beliebiges d.

Man beachte noch einmal, daß 2 Prozessoren genau dann verbunden sind, wenn sich ihre Indizes in genau einem Bit unterscheiden. □

Algorithmus 1.12 Wir wollen eine $(n \times n)$–Matrix A mit einem $(n \times 1)$–Vektor x auf einem Ring, der aus p Prozessoren $P_1 \dots P_p$ besteht, multiplizieren. Nehmen wir nun an, daß die Zahl der Prozessoren p die Dimension des Problems n ganzzahlig teilt. Sei $r = \frac{n}{p}$. Wir nehmen an, daß A in Blocks

$A_1, ..., A_p$ und x in Blocks $x_1, ..., x_p$ eingeteilt ist.

$$A = (A_1, ..., A_p), \quad x = \begin{pmatrix} x_1 \\ \vdots \\ x_p \end{pmatrix}$$

Dann ist A_i eine $(n \times r)$–Matrix, x_i eine $(r \times 1)$–Matrix. Wir nehmen an, daß der lokale Speicher von Prozessor P_i die Matrix A_i und den Vektor x_i enthält.

Wir benötigen die für ein asynchron arbeitendes Netzwerk typischen Kommunikationsbefehle.

receive (X,j) und send (X,j).

Diese Befehle haben folgende Semantik: Die Ausführung von send (X,j) auf Prozessor P bewirkt, daß P den Wert von **X** an den Prozessor mit Namen **j** sendet. Man beachte, daß Prozessor **j** *nicht* unbedingt im Netz benachbart zu P sein muß. Netze arbeiten mit *routing*–Algorithmen, die die Nachrichten an die gewünschten Prozessoren senden. Die Ausführung von receive (X,j) auf Prozessor P bewirkt, daß P solange wartet, bis er eine Nachricht von Prozessor j bekommen hat. Die Nachricht wird in **X** gespeichert. Erst dann setzt P die Programmausführung fort. Bei einem Netzwerk, in dem die Prozessoren unabhängig sind, also insbesondere asynchron arbeiten, abgesehen von der Kommunikation über Befehle receive (X,j) und send (X,j) , spricht man vom *message–passing* Modell.

Jetzt der Algorithmus: Jeder Prozessor P_i berechnet lokal $z_i = A_i \cdot x_i$. Hinterher wird dann summiert, so daß schließlich $y = z_1 + z_1 + \cdots + z_p$.

Das Programm von Prozessor P_i ist:

Eingabe: Prozessornummer i, Gesamtzahl von Prozessoren p, $B = A_i =$ Spalten$(i-1) \cdot r + 1, ..., i \cdot r$ von A, i–ter Teilvektor $w = x_i$ von x, bestehend aus den Zeilen $(i-1) \cdot r + 1, ..., i \cdot r$ von x.

Ausgabe: Prozessor P_i berechnet den Vektor $y = A_1 \cdot x_1 + \cdots + A_i \cdot x_i$ und gibt das Ergebnis zu Prozessor $P_{(i+1}$, falls $i + 1 \leq p$ bzw. zu P_1, falls $i + 1 = p + 1$ ist, d.h. nach rechts.

```
begin
  1.z := B · w
  2.if i = 1 then y := 0 else receive (y,left)
      { left ist der linke Nachbar von P_i }
```

```
  3.y := y + z
  4.send (y, right)
      { right ist der rechte Nachbar von P_i }
  5.if i = 1 then receive (y,left)
end
```

□

Bemerkung 1.13 Um die Laufzeit von Algorithmus 1.12 überhaupt irgendwie ermitteln zu können, nehmen wir an, daß er synchron abläuft. Nehmen wir einmal an, wir haben p = 5 Prozessoren P_1, P_2, P_3, P_4, P_5. Wir stellen den Ablauf des Algorithmus in einer Tabelle dar:

P_1	P_2	P_3	P_4	P_5
z := B · w	z := B · w	z := B · w	z := B · w	z := B · w
y := 0	warten	warten	warten	warten
y := z	"	"	"	"
send (y,2)	"	"	"	"
warten	**receive** (y,1)	"	"	"
"	y := y + z	"	"	"
"	**send** (y,3)	"	"	"
"	fertig	**receive** (y,2)	"	"
"		y := y + z	"	"
"		**send** (y,4)	"	"
"		fertig	**receive** (y,3)	"
"			y := y + z	"
"			**send** (y,5)	"
"			fertig	**receive** (y,4)
"				y := y + z
"				**send** (y,1)
receive (y,5)				fertig
fertig				

Damit bekommen wir folgenden Zeitverbrauch:

Zeile 1: $\frac{n^2}{p} + (\frac{n^2}{p} - n) = n \cdot (\frac{n}{b} + (\frac{n}{b} - 1))$ arithmetische Operationen.
Zeile 2: n arithmetische Operationen
Zeile 3: n arithmetische Operationen.
Zeile 4/5: Kommunikation von n Zahlen.

Zeile 6: n arithmetische Operationen.
Zeile 7/8: Kommunikation von n Zahlen.

Zeile 9: n arithmetische Operationen.
Zeile 10/11: Kommunikation von n Zahlen.

Zeile 12: n arithmetische Operationen.
Zeile 13/14: Kommunikation von n Zahlen.

Zeile 15: n arithmetische Operationen.
Zeile 16/17: Kommunikation von n Zahlen.

Bei p Prozessoren haben wir genau p–mal das Zeilentripel bestehend aus Berechnung und Kommunikation. Wir bekommen also

$$T_p(n) = \alpha \cdot (\frac{n^2}{p} + (\frac{n^2}{p} - n)) + \alpha \cdot n + p \cdot (\alpha \cdot n + (\sigma + \tau \cdot n)).$$

Dabei

$\alpha =$ Zeit für eine Operation,

$\sigma =$ Zeit, um eine Kommunikation aufzubauen,

$\tau =$ die eigentliche Übertragungszeit für eine Zahl.

Sei n fest. Wir fragen: Mit wievielen Prozessoren p wird $T_p(n)$ minimal? Man sieht, daß mit steigendem p der Anteil der reinen Rechenzeit an $T_p(n)$, das ist $\alpha \cdot (\frac{n^2}{p} + \frac{n^2}{p} - n) + \alpha \cdot n$, kleiner wird. Dafür steigt die Kommunikations- und Wartezeit: $p \cdot (\alpha n + (\sigma + \tau \cdot n))$.

Berechnen wir noch, für welche Prozessorenzahl p die Zeit $T_p(n)$ minimal wird, bei gegebenen n:

$$\text{Ableitung nach } p \text{ von } T_p(n) = \alpha \cdot (-\frac{2n^2}{p^2}) + \alpha \cdot n + \sigma + \tau \cdot n.$$

$$\text{Ableitung} = 0 \quad \Longleftrightarrow \alpha \cdot \frac{2n^2}{p^2} = \alpha \cdot n + \sigma + \tau \cdot n$$

$$\Longleftrightarrow p = n \cdot \sqrt{\frac{2\alpha}{\alpha \cdot n + \sigma + \tau \cdot n}} = n \cdot \frac{\sqrt{2\alpha}}{\sqrt{\alpha \cdot n + \sigma + \tau \cdot n}}$$

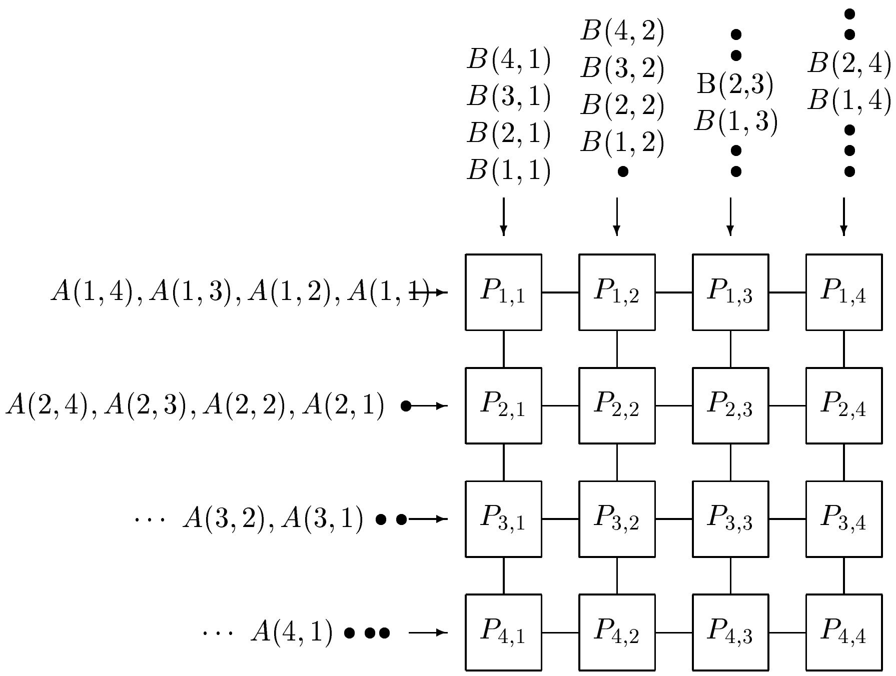
Die Berechnung bisher wurde für jedes feste n durchgeführt. Betrachten wir jetzt $p = p(n)$ als Funktion von n, dann gilt:

$$p = p(n) = \Theta(\sqrt{n})$$

,

wobei die Konstante über dem Θ von unserem α, σ und τ abhängt. $\square$

Algorithmus 1.14 Zwei $(n \times n)$–Matrizen A und B können in einem synchron arbeitenden Gitter aus n^2 Prozessoren in Zeit $O(n)$ multipliziert werden. Für $n = 4$ bekommen wir folgende Situation:



Prozessor $P_{i,j}$ hat die Variable $C(i,j)$ in seinem Speicher, die am Anfang auf 0 gesetzt ist und führt folgende Operationen aus: Er empfängt A(i,l) von links und B(l,j) von oben, berechnet $C(i,j) := C(i,j) + A(i,l) \cdot B(l,j)$ und schickt A(i,l) nach $P_{i,j+1}$ (also nach rechts), falls j + 1 $\leq$ 4, B(l,j) nach $P_{i+1,j}$ (also nach unten), falls i + 1 $\leq$ 4. $\square$

Man nennt solche Algorithmen auf dem Gitter systolische Algorithmen. Charakteristisch ist, daß die Prozessoren vollkommen synchron arbeiten: In jedem Schritt macht jeder Prozessor 3 einzelne Schritte

- Er empfängt Daten von einigen Nachbarn.

- Er führt lokal einige Rechnungen aus, die in diesem Fall sogar vollkommen gleich sind.
- Er sendet Daten an Nachbarn.

Zum Abschluß betrachten wir noch 3 Algorithmen für den synchron arbeitenden Hypercube.

Algorithmus 1.15 Wir wollen die Summe der Elemente des Feldes A bilden. Genauer: Sei $n = 2^d$. Wir arbeiten auf dem d–dimensionalen Hypercube. Das Feld A hat die Elemente $A(0), ..., A(n-1)$. Wir gehen davon aus, daß $A(i)$ in Prozessor P_i gespeichert ist, wobei jetzt i mit seiner Binärdarstellung identifiziert wird. Wir wollen die Summe $A(0) + \cdots + A(n-1)$ in Prozessor P_0 bekommen. Der Algorithmus arbeitet in d Iterationen. In der ersten Iteration wird $A(0w) + A(1w)$ berechnet für alle $w \in \{0,1\}^{d-1}$ und in P_{0w} gespeichert. Danach summieren wir die Werte in P_{00v} und P_{01v} für $v \in \{0,1\}^{d-2}$ und speichern sie in P_{00v}, usw. Der Algorithmus für P_i sieht so aus:

Eingabe: Wie oben beschrieben.

Ausgabe: $\sum_{i=0}^{n-1} A(i)$ in P_0 gespeichert.

```
begin
  for l = d - 1 to 0 do
    if 0 ≤ i ≤ 2^l − 1 do
      A(i) := A(i) + A(i^(l))
end
```

Dabei bezeichnet $i^{(l)}$ den Wert, den wir aus der Binärdarstellung von i durch Komplementieren des l–ten Bits bekommen. Dabei sind die Bits wie folgt numeriert: (Bit$(d-1)$) ... (Bit 1) (Bit 0).

Der Algorithmus braucht $d = \log\, n$ Rechenschritte. □

Algorithmus 1.16 Das broadcasting–Problem: Wir wollen den Speicherinhalt X des Registers D(0) des Prozessors P_0 allen Prozessoren des d–dimensionalen Hypercube zusenden.

Das Prinzip unseres Algorithmus ist rekursiv gemäß der rekursiven Struktur des Hypercube.

1. Schritt	$00\cdots0 \to 100\cdots0$,
	d.h. X wird von P_{000} nach $P_{100\cdots0}$ gesendet.
2. Schritt	$00\cdots0 \to 010\cdots0$
	$100\cdots0 \to 110\cdots0$
3. Schritt	$00\cdots0 \to 0010\cdots0$
	$100\cdots0 \to 1010\cdots0$
	$010\cdots0 \to 0110\cdots0$
	$110\cdots0 \to 1110\cdots0$
	$\vdots$

Der Algorithmus für Prozessor P_i:

Eingabe: Prozessor P_0 eines d–dimensionalen Hypercube hat den Wert X im Register $D(0)$.

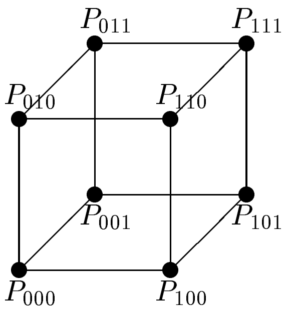
Ausgabe: Für jeden Prozessor P_i ist $D(i) = X$, dabei ist $D(i)$ ein lokales Register von P_i.

```
begin
  for l = d - 1 to 0 do
    if i ≤ 2^(d-1) and (i mod 2^(l+1)) = 0 then
      D(i^(l)) := D(i)
end
```

Der Algorithmus ist nach d Schritten beendet, dabei ist d der Logarithmus der Anzahl der Prozessoren. □

Ein Charakteristikum der beiden angegebenen Algorithmen für den Hypercube ist, daß in jedem Schritt die Verbindungen derselben Dimension benutzt werden. Solche Algorithmen heißen *normal*. Insbesondere werden alle Dimensionen einmal der Reihe nach benötigt. Solche Algorithmen heißen *vollständig normal (fully normal)*.

Algorithmus 1.17 Ermittlung des Produkts von 2 $(n \times n)$–Matrizen A und B auf dem Hypercube mit n^3 Prozessoren. Dabei sei n eine Zweierpotenz $n = 2^q$. Zunächst betrachten wir als Beispiel den Fall $n = 2$. Wir haben folgenden Hypercube:



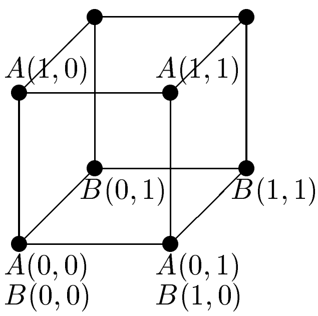
Wir haben folgende Matrizen:

$$A = \begin{pmatrix} A(0,0) & A(0,1) \\ A(1,0) & A(1,1) \end{pmatrix} \quad B = \begin{pmatrix} B(0,0) & B(0,1) \\ B(1,0) & B(1,1) \end{pmatrix}$$

Wie sind A und B gespeichert? A ist im Teilwürfel, dessen letztes Bit 0 ist, gespeichert, B in dem, dessen vorletztes Bit 0 ist und zwar nach folgender Vorschrift:

$$A(i,l) \text{ in } P_{li0} \text{ und } B(l,j) \text{ in } P_{l0j}.$$

Wir bekommen folgende Situation:



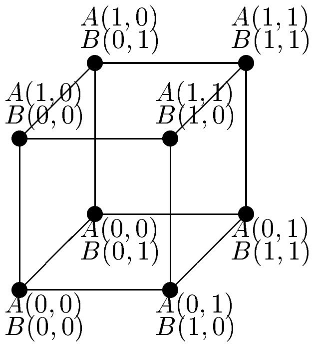
Wir wollen für i,j mit $0 \leq i,j \leq 1$

$$C(i,j) = \sum_{l=0}^{1} A(i,l) \cdot B(l,j)$$

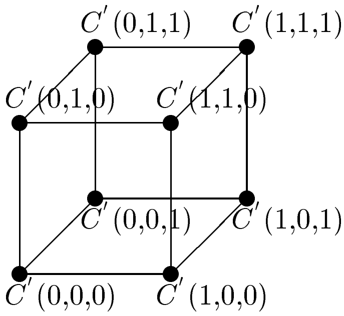
berechnen. Der Algorithmus läuft in 3 Schritten ab:

1. Die Eingabedaten werden so verteilt, daß P_{lij} die Werte $A(i,l)$ und $B(l,j)$ enthält. Wir müssen also $A(i,l)$ auf P_{li1} senden und $B(l,j)$

auf P_{lij}. Das geht hier in einem Schritt. Nach dem Verteilen ist unser Hypercube im folgenden Zustand:



2. Prozessor P_{lij} berechnet C $'$(l,i,j) = A(i,l) $\cdot$ B(l,j). Dann bekommen wir die Situation:



3. Wir bilden die Summe $C(i,j) = \sum_{l=0}^{1} A(i,l) \cdot B(l,j)$ im Prozessor P_{0ij}.

Wir verallgemeinern die skizzierte Vorgehensweise auf beliebiges $n = 2^q$. Wir haben also einen Hypercube mit $n^3 = 2^{3 \cdot q}$ Prozessoren. Im Falle n = 2 ist jeder Prozessor durch ein Tripel von Bits gekennzeichnet. Analog kennzeichnen wir hier im allgemeinen Fall jeden Prozessor durch ein Tripel von Zahlen l, j, i mit $0 \leq l, i, j \leq n-1$: Jeder Prozessor ist durch eine Dualzahl von $3 \cdot q$ Bits gekennzeichnet. Seien w_1, w_2, w_3 jeweils Worte von q Bits, so daß w_1 die Zahl l, w_2 die Zahl i und w_3 die Zahl j darstellt, so ist

$$P_{l,i,j} = P_{w_1,w_2,w_3}.$$

Man beachte, daß so tatsächlich $0 \leq l, i, j \leq n-1$ ist. Mit dieser Notation gehen wir vollkommen analog zu Fall n = 2 vor. Wir haben die Matrizen

$$A = \begin{pmatrix} A(0,0) & A(0,1) & A(0,2) & \cdots & A(0,n-1) \\ \vdots & & & & \vdots \\ A(n-1,0) & & & \cdots & A(n-1,n-1) \end{pmatrix}$$

und B von derselben Form. Die Matrix A ist in dem Teilwürfel $P_{l,i,0}$ gespeichert:

$$A(i,l) \text{ in } P_{l,i,0}.$$

Matrix B ist in $P_{l,0,j}$ gespeichert:

$$B(l,j) \text{ in } P_{l,0,j}.$$

Der Algorithmus besteht aus 3 Schritten:

1. Versenden von $A(i,l)$ auf alle Prozessoren $P_{l,i,j}$ für $0 \leq j \leq n-1$. Verteilen von $B(j,l)$ auf alle Prozessoren $P_{l,i,j}$ für $0 \leq i \leq n-1$. Nach diesem Schritt haben wir $A(i,l)$ und $B(l,j)$ auf allen Prozessoren $P_{l,i,j}$. Unter Anwendung von Algorithmus 1.16 brauchen wir hier $O(\log n)$ Schritte. Man beachte, daß wir die broadcasts für alle $A(i,l)$ parallel ausführen können. Danach die für alle $B(l,j)$.

2. Auf $P_{l,i,j}$ wird $C'(l,i,j) = A(i,l) \cdot B(l,j)$ berechnet. Das geht in einem Schritt.

3. Wir berechnen $C(i,j) = \sum_{l=0}^{n-1} C'(l,i,j)$ auf $P_{0,i,j}$. Da wir alle Elemente auf einem Teilhypercube der Dimension n aufaddieren, brauchen wir mit Algorithmus 1.15 $O(\log n)$ Schritte.

Insgesamt können also zwei $(n \times n)$–Matrizen in $O(\log n)$ vielen Schritten auf einem Hypercube mit n^3 Prozessoren multipliziert werden (falls die Initialisierung richtig ist). □

Dieser Abschnitt sollte einen einführenden Überblick über die verschiedenen Ansätze, parallele Algorithmen zu beschreiben, geben. Im Rest der Vorlesung werden wir uns auf Algorithmen für die synchrone PRAM beschränken. Zugunsten dieser Entscheidung sprechen folgende Gründe:

- Es gibt eine zufriedenstellende Theorie für PRAM–Algorithmen.

- Das PRAM–Modell abstrahiert zunächst einmal von den Details der Kommunikation und erlaubt es uns so, die wesentlichen Probleme für die Parallelisierung besser zu erkennen.

- Das PRAM–Modell spiegelt einige wichtige Aspekte der Realität wider: die Operationen, die zu einem Zeitpunkt ausgeführt werden können, und wie die Operationen auf die Prozessoren verteilt werden.
- Man kann PRAMs auf den realitätsnäheren Netzen relativ effizient simulieren. (Diese Simulationen haben in den letzten Jahren eine umfangreiche Literatur hervorgebracht.)
- Man kann Fragen der Synchronisation und Kommunikation in das PRAM–Modell aufnehmen und damit weitere Aspekte der Realität paralleler Algorithmen mit PRAMs untersuchen.

1.2 Effizienz paralleler Algorithmen

(Die folgenden Bemerkungen sind entscheidend zum Verständnis der weiteren Vorlesung.)

Bevor wir uns der Erstellung komplizierter Algorithmen zuwenden, müssen wir noch folgende Fragen klären: Wann ist ein paralleler Algorithmus effizient? Wie schnell kann ein paralleler Algorithmus für ein gegebenes Problem überhaupt werden? Man kann selbst durch den Einsatz beliebig vieler parallel arbeitender Prozessoren nicht beliebig schnell werden, da man sonst jedes Problem in einem Schritt lösen könnte (vorausgesetzt wir haben genügend Prozessoren). Das geht aber bei solchen Problemen nicht, bei deren Lösung es erforderlich ist, mehrere Schritte hintereinander auszuführen, deren Lösung also einen inhärent sequentiellen Anteil hat.

Dazu zunächst folgende Betrachtung:

Lemma 1.18 Sei A ein Algorithmus für eine PRAM mit p Prozessoren, dann können wir einen Algorithmus B für eine PRAM mit $q \leq p$ Prozessoren konstruieren, der semantisch äquivalent zu A ist. Weiterhin gilt für alle Eingaben E von A oder B: Braucht A bei Eingabe von E T Schritte, so ist B nach

$$\leq c \cdot \left\lceil \frac{p}{q} \right\rceil \cdot T$$

Schritten fertig. Dabei ist c eine kleine Konstante unabhängig von p, q, T oder E und auch von A.

Beweis
Wir müssen die $\leq p$ Operationen pro Schritt in A durch eine PRAM simulieren, die nur $q \leq p$ Prozessoren hat. Dazu werden diese p Operationen

möglichst gleichmäßig auf die q Prozessoren verteilt. Pro Schritt der Maschine für A muß die neue Maschine $\leq \left\lceil \frac{p}{q} \right\rceil$ Schritte machen. Das stimmt noch nicht so ganz (aber fast): Sei $p = 4$ und $q = 2$ und nehmen wir an, die Prozessoren P_i für $1 \leq i \leq 4$ führen die folgenden Instruktionen in einem Schritt aus:

$$\mathbf{x_i} := \mathbf{x_i} + \mathbf{x_1}$$

Dann können wir auf einer q–Prozessor PRAM nicht einfach sagen: Prozessor 1 rechnet

$$\mathbf{x_1} := \mathbf{x_1} + \mathbf{x_1};\ \ \mathbf{x_2} := \mathbf{x_2} + \mathbf{x_1}$$

und Prozessor 2 rechnet

$$\mathbf{x_3} := \mathbf{x_3} + \mathbf{x_1};\ \ \mathbf{x_4} := \mathbf{x_4} + \mathbf{x_1}.$$

Statt dessen müssen wir zunächst auf *neue* Variablen zuweisen, die danach den x_i zugewiesen werden:

Prozessor 1:

$$\begin{aligned}
\mathbf{x_1'} &:= \mathbf{x_1} + \mathbf{x_1}; \\
\mathbf{x_2'} &:= \mathbf{x_2} + \mathbf{x_1}; \\
\mathbf{x_1} &:= \mathbf{x_1'}; \\
\mathbf{x_2} &:= \mathbf{x_2'};
\end{aligned}$$

Prozessor 2

$$\begin{aligned}
\mathbf{x_3'} &:= \mathbf{x_3} + \mathbf{x_1}; \\
\mathbf{x_4'} &:= \mathbf{x_4} + \mathbf{x_1}; \\
\mathbf{x_3} &:= \mathbf{x_3'}; \\
\mathbf{x_4} &:= \mathbf{x_4'};
\end{aligned}$$

Gegenüber dem ursprünglichen Programm verdoppelt sich also die Anzahl der einzelnen Instruktionen. $\square$

Im allgemeinen betrachtet man die Schrittanzahl eines Algorithmus nicht auf einer festen Eingabe E, sondern als Funktion, die von einem Wert n abhängt, der in irgendeinem Sinne die Kompliziertheit der Eingabe mißt.

Definition 1.19

(a) Sei A ein Algorithmus (für eine PRAM), die *Zeit* (oder *Schrittanzahl*) von A ist die Funktion $T\ : N \to N$ mit

$$T(n) \quad = \quad \text{Max } \{ \quad \#\text{ Schritte von A auf E } | \\ \text{E ist eine Eingabe der „Größe" } n \quad \}.$$

Wir betrachten also die *worst–case* Komplexität (im Gegensatz zur *average–case* Komplexität).

(b) Die Arbeit von A ist die Funktion $W\ : N \to N$ mit

$$W(n) \quad = \quad \text{Max } \{ \quad \#\text{ Ausführungen von Instruktionen von } A \text{ auf } E| \\ E \text{ ist eine Eingabe der „Größe" } n\}.$$

□

Man beachte auch noch einmal Bemerkung (1.5(c), nach der wir uns auf das uniforme Kostenmaß beschränken wollen.

Lemma 1.18 können wir nun schreiben als:

Folgerung 1.20
Ist A ein Algorithmus für die PRAM mit $p = p(n)$ Prozessoren und Laufzeit $T^p(n)$, so bekommen wir einen Algorithmus für die PRAM mit $q = q(n) \leq p(n)$ Prozessoren mit Laufzeit

$$T^q(n) = O(\frac{p(n)}{q(n)} \cdot T^p(n)),$$

wobei die Konstante möglicherweise vom Programm abhängen kann. (Die Konstante kann evtl. vom Programm abhängen, da $p = p(n)$ und $q = q(n)$ nicht konstant sind. Damit erfordert eine Simulation wie im Beweis von Lemma 1.18 den Durchlauf von geeigneten Schleifen.) □

Wir können so einen Algorithmus auf einer kleineren Anzahl von Prozessoren als die, für die er ursprünglich geschrieben war unter begrenzbarem Zeitverlust simulieren. Das motiviert die folgende Vorgehensweise:

Zunächst formulieren wir unser Programm für eine PRAM mit unendlich vielen Prozessoren. In jeder terminierenden Berechnung werden aber natürlich nur endlich viele dieser Prozssoren eingesetzt. Wir bestimmen die Zeit $T^{\infty}(n)$ von diesem Programm. Danach implementieren wir das Programm auf einer PRAM mit $p = p(n)$ Prozessoren. Es wird sich herausstellen, daß dieser zweite Schritt kaum interessant ist, bzw. sich durch ein allgemeines scheduling–Prinzip erledigen läßt, und deshalb meist weggelassen wird.

Wir verändern deshalb unsere PRAM–Programmiersprache so, daß wir die Möglichkeit haben, direkt beliebig viele Prozessoren anzusprechen. Wir erweitern unsere Programmiersprache um das for–pardo statement.

Syntax: for $l \leq i \leq u$ pardo statement

Semantik: Der Effekt des angegebenen statements ist, daß statement für alle i mit $l \leq i \leq u$ gleichzeitig ausgeführt wird. Dabei hängt das statement im allgemeinen von i ab. (l steht für lower bound, u für upper bound .)

Man beachte, daß man mit dem for–pardo statement nur endlich, aber von der Eingabe unabhängig viele Prozessoren ansprechen kann.

Algorithmus 1.20 Der Algorithmus 1.7 läßt sich jetzt schreiben als:

Eingabe: $n = 2^k$ Zahlen in einem Feld A gespeichert

Ausgabe: $S = \sum_{i=1}^{n} A(i)$

```
begin
  1.for 1 ≤ i ≤ n pardo
      B(i) := A(i);
  2.for h = 1 to log n do
      for 1 ≤ i ≤ n/2^h pardo
        B(i) := B(2i - 1) + B(2i);
  3.S := B(1)
end
```

Man beachte, wie das par–do statement sogar in einer Schleife aufgerufen wird. Einzelne Prozessoren werden nicht mehr erwähnt. □

Definition 1.21 Bei der Zeitmessung von Algorithmen mit dem for–pardo statement gehen wir davon aus, daß *alle* unter for–pardo zusammengefaßten statements parallel ausgeführt werden. Eine PRAM mit unendlich vielen Prozessoren könnte die so gemessene Zeit realisieren. Wir bezeichnen diese Zeit als $T^\infty(n)$ und nennen sie *parallele Zeit.* Die Arbeit dagegen ist die Anzahl der insgesamt ausgeführten Instruktionen. Wir nennen PRAM–Algorithmen mit dem for–pardo statement *WT–Algorithmen* (für work–time Algorithmen). □

Bemerkung 1.22

(a) Sei $T^{\infty(n)}$ die Zeit parallele von Algorithmus 1.20, dann gilt:

$$T^\infty(n) = 1 + \log n + 1 = O(\log n).$$

(b) Sei $W(n)$ die Arbeit von Algorithmus 1.20, dann gilt:

$$\begin{aligned} W(n) &= n + \left(\textstyle\sum_{k=1}^{\log n} \frac{n}{2^k}\right) + 1 \\ &= n + 1 + n\left(\textstyle\sum_{k=1}^{\log n} \frac{1}{2^k}\right) \\ &= n + 1 + n\left(\frac{1-\left(\frac{1}{2}\right)^{\log n + 1}}{\frac{1}{2}} - 1\right) \\ &= n + 1 + n\left(\frac{1-\frac{1}{2n}}{\frac{1}{2}} - 1\right) \\ &= n + 1 + n - 1 = 2 \cdot n \end{aligned}$$

Man beachte, daß *jeder* binäre Baum (d.h. jeder innere Knoten hat genau zwei Söhne) mit n Blättern genau innere $n-1$ Knoten hat. □

Der folgende Satz ist entscheident für die Analyse von WT–Algorithmen. Sein Grundgedanke geht auf Ideen von *Brent* aus den 70er Jahren zurück. Der Satz ist als *WT–scheduling Prinzip* oder als *Brent's–scheduling–Prinzip* bekannt.

Satz 1.23 Gegeben ist ein paralleler Algorithmus A in der WT–Darstellung. Sei $T^\infty(n)$ der parallele Zeitaufwand dieses WT–Algorithmus und $W(n)$ die Arbeit. Dann gilt:

(a) Wir können A für eine PRAM mit Prozessorzahl $p(n)$ implementieren, so daß gilt:

$$Max\{\frac{W(n)}{p(n)}, T^{\infty}(n)\} \leq T^{p(n)}(n) = O(\frac{W(n)}{p(n)} + T^{\infty}(n)),$$

wobei $T^{p(n)}(n)$ der Zeitaufwand der Implementierung von **A** ist.

Die Konstante in dem O hängt dabei höchstens von dem A ab, nicht aber von der Prozessorzahl $p(n)$.

(b) Mit den Bezeichnungen wie in (a) gilt: Falls

$$p(n) = \Omega(\frac{W(n)}{T^{\infty}(n)}),$$

dann

$$T^{\infty}(n) \leq T^{p(n)}(n) = O(T^{\infty}).$$

(c) Falls dagegen $p(n) = O(\frac{W(n)}{T^{\infty}(n)})$ ist, dann gilt

$$\frac{W(n)}{p(n)} \leq T^{p(n)}(n) = O(\frac{W(n)}{p(n)}).$$

Beweis:

(a) Sei für $1 \leq i \leq T^{\infty}(n)$

$W_i(n) =$ die Menge der Instruktionen, die im i–ten Schritt ausgeführt werden,

Im weiteren schreiben wir $W_i(n)$ und $W_i'(n)$ auch für die Anzahl der Instruktionen im i–ten Schritt.

Die Implementierung folgt dem Muster des Beweises von Lemma 1.18: $p(n)$-1 Prozessoren führen $\lceil \frac{W_i(n)}{p(n)} \rceil$ viele Instruktionen aus $W_i(n)$ aus, ein Prozessor $\lceil \frac{W_i(n)}{p(n)} \rceil$ viele Instruktionen. Damit ist ganz $W_i(n)$ simuliert. Die programmiertechnische Realisierung dieser Idee erfordert zunächst die Ermittlung von $W_i(n)$ und dann das Verteilen (d.h. das scheduling) der Instruktionen aus $W_i(n)$ auf die Prozessoren. Die Prozessoren durchlaufen dann jeweils Schleifen der oben angegebenen Länge. Unsere Implementierung implementiert also ein $W_i(n)$ in $O(\lceil\frac{W_i(n)}{p(n)}\rceil)$ vielen Schritten. Daraus folgt:

$$T^p(n) = O(\sum_{i=1}^{T^\infty(n)} \lceil\frac{W_i(n)}{p(n)}\rceil) = O(\frac{W(n)}{p(n)} + T^\infty(n)),$$

da gilt:

$$\sum_{i=1}^{T^\infty(n)} \lceil\frac{W_i(n)}{p(n)}\rceil \le \sum(\frac{W_i(n)}{p(n)}+1) = (\sum \frac{W_i(n)}{p(n)})+T^\infty(n) = \frac{W(n)}{p(n)}+T^\infty(n)$$

Man sieht also hier, was der $\lceil\ \rceil$-Operator doch für einen großen Einfluß haben kann: Er hat das $T^\infty(n)$ zur Folge.

Für die angegebene Implementierung gilt:

$$T^\infty(n) \le \sum \lceil\frac{W_i(n)}{p(n)}\rceil = T^p(n),$$

sofern jedes $W_i(n)$ zumindest eine Instruktion beinhaltet und

$$\frac{W(n)}{p(n)} = \frac{\sum W_i(n)}{p(n)} \le \sum \lceil\frac{W_i(n)}{p(n)}\rceil = T^p(n).$$

(b) Folgt mit (a), da die Voraussetzung impliziert, daß

$$\frac{W(n)}{p(n)} \le C \cdot T^\infty(n)$$

gilt. (Man beachte, daß hier die Konstante C von der in der Voraussetzung verborgenen, im dem Ω abhängt.)

(c) Folgt mit (a), da die Voraussetzung impliziert, daß

$$T^\infty(n) \le C \cdot \frac{W(n)}{p(n)}$$

für eine geeignete Konstante C gilt. $\square$

Was sagt uns (b) dieses Satzes? Mit einer Prozessorzahl einer geeigneten Größe, nämlich $\Omega(\frac{W(n)}{T^\infty(n)})$ sind wir an der schnellstmöglichen Geschwindigkeit, nämlich $T^\infty(n)$, angelangt.

Was ist $\frac{W(n)}{T^\infty(n)}$?

$$W(n) = |W_1(n)| + |W_2(n)| + \ldots + |W_{T^\infty(n)}(n)|.$$

Also ist $\frac{W(n)}{T^\infty(n)}$ das arithmetische Mittel der $|W_i(n)|$. Da die Instruktionen in $W_i(n)$ alle parallel ausgeführt werden können, ist $\frac{W(n)}{T^\infty(n)}$ der *mittlere Parallelitätsgrad* des Programms.

Also sagt uns (b), daß mehr Prozessoren als der mittlere Parallelitätsgrad zu nicht viel von Nutzen sind.

Was sagt uns (c)?

Ist $p(n)$, $q(n) = O(\frac{W(n)}{T^\infty(n)})$ und $p(n) \leq q(n)$, dann wissen wir, daß für den Speedup von $T_{q(n)}(n)$ über $T_{p(n)}(n)$ gilt:

$$Speedup = \frac{T^{p(n)}(n)}{T^{q(n)}(n)} \geq \frac{\frac{W(n)}{p(n)}}{C \cdot \frac{W(n)}{q(n)}} = \frac{1}{C} \cdot \frac{q(n)}{p(n)}$$

Er ist also linear im Verhältnis der Anzahl der Prozessoren. Das heißt z.B.: nimmt die Anzahl der Prozessoren um den Faktor 3 zu, so verringert sich die Laufzeit um mindestens $\frac{1}{3}$, d.h. wir führen wirklich pro Schritt 3–mal soviele Instruktionen aus. Die neu hinzugekommenen Prozessoren werden also vernünftig genutzt.

Algorithmus 1.24 Wir wenden das WT–scheduling Prinzip jetzt einmal auf den Algorithmus 1.20 an. Der Algorithmus von Prozessor P_s lautet dann:

Eingabe: Feld der Größe $n = 2^k$. Die folgenden lokalen Variablen sind initialisiert: (1) Die Dimension des Feldes n. (2) Die Anzahl der Prozessoren $p = 2^q \leq n$. (3) Die Prozessornummer s. (4) $l = \frac{n}{p}$.

Ausgabe: Die Summe wie bekannt.

```
begin
  1.for j = 1 to l = n/p do
       B(l(s-1)+j) := A(l(s-1)+j)
  2.for k = 1 to log n do
       2.1 if k - h - q ≥ 0 then
               for j = 2^(k-h-q) · (s - 1) + 1 to 2^(k-h-q) do
                 B(j) := B(2j-1) + B(2j)
       2.2 else if s ≤ 2^(k-h) then
                 B(s) := B(2s-1) + B(2s)
  3.if s = 1 then S := B(1)
end
```

□

Man beachte, daß der sich nach dem WT–scheduling Prinzip ergebende Algorithmus keinesfalls eindeutig ist. Wichtig ist nur, daß sich die Instruktionen in $W_i(n)$ möglichst gleichmäßig auf die vorhandenen Prozessoren verteilen.

Bemerkung 1.25 Die Laufzeit von Algorithmus 1.24 ist:

Schritt 1:	$O\left(\frac{n}{p}\right)$
Die h–te Iteration von Schritt 2:	$O\left(\left\lceil\frac{n}{2^h \cdot p}\right\rceil\right)$
Schritt 3:	$O(1)$

Also gilt für die Laufzeit

$$T_p(n) = O\left(\frac{n}{p} + \sum_{h=1}^{\log n}\left\lceil\frac{n}{2^h \cdot p}\right\rceil\right) = O\left(\frac{n}{p} + \log n\right),$$

wie es ja auch von dem WT–scheduling Prinzip vorausgesagt wurde. □

Man beachte, daß sich die Arbeit bei Anwendung des WT–scheduling Prinzips größenordnungsmäßig nicht erhöht. Das heißt: braucht die WT–Darstellung des Algorithmus die Arbeit $W(n)$, so braucht der Algorithmus, der sich durch Anwendung des WT–scheduling Prinzips für die PRAM ergibt, die Arbeit $\Theta(W(n))$. Die im Algorithmus 1.24 auftretenden Schleifen erhöhen den Rechenaufwand nur um einen konstanten Faktor.

Zum Abschluß dieses Abschnittes müssen wir noch etwas detaillierter auf den Speedup paralleler Algorithmen eingehen.

Definition 1.26 Sind A und B 2 Algorithmen für dasselbe Problem, so definieren wir den *Speedup* von A über B als

$$Speedup_{A,B}(n) = \frac{T_B(n)}{T_A(n)}.$$

Braucht etwa A die Hälfte der Zeit von B, so ist der Speedup $= 2$.
Man beachte, daß hier *worst–case*–Laufzeiten verglichen werden.

□

Um die Qualität des parallelen Algorithmus vernünftig einschätzen zu können, betrachten wir einen Speedup über einen sequentiellen Algorithmus. Nun ist es nicht besonders sinnvoll, einen beliebig schlechten sequentiellen Algorithmus als Vergleichsmaßstab zu betrachten. Wir beziehen uns immer auf den besten bekannten sequentiellen Algorithmus.

Definition 1.27 Sei B ein sequentieller Algorithmus zur Lösung eines Problems P. Sei $T_B(n)$ die Laufzeit von P. Wir sagen B ist *optimal*, gdw. für jeden anderen bekannten sequentiellen Algorithmus A zur Lösung von P mit Laufzeit $T_A(n)$ gilt: $T_A(n) = \Omega(T_B(n))$.

□

Das folgende Lemma faßt die Eigenschaften des Speedup eines parallelen Algorithmus über einen im Sinne der vorangegangenen Definition optimalen sequentiellen Algorithmus zusammen.

Lemma 1.28 Sei B ein optimaler sequentieller Algorithmus für ein Problem P und A ein paralleler Algorithmus für die PRAM mit $p(n)$ Prozessoren. Dann gilt:

(a) $Speedup_{A,B}(n) = O(p(n))$.

(b) Falls $Speedup(n) = \Omega(p(n))$, dann $W(n) = O(T_B(n))$, wobei $W(n)$ die Arbeit von A ist.

□

Beweis:

(a) Folgt, da A nach Folgerung **1.20** einen sequentiellen Algorithmus mit Laufzeit $O(p(n) \cdot T_A(n))$ nach sich zieht, also

$$p(n) \cdot T_A(n) = \Omega(T_B(n))$$

sein muß, wegen der Optimalität von B.

(b) Es gilt:

$$\frac{T_B(n)}{T_A(n)} = Speedup_{A,B}(n) = \Omega(p(n)).$$

Damit folgt, da A pro Zeiteinheit nur $p(n)$ viele Schritte machen kann, daß

$$W(n) = O(T_A(n) \cdot p(n)) = O(T_B(n))$$

gilt. □

Wir können also zu einem optimalen sequentiellen Algorithmus nur einen Speedup proportional zur Prozessorenzahl bekommen.

Aussage (b) des Satzes sagt: Ein paralleler Algorithmus mit maximalen Speedup braucht nicht mehr Arbeit als ein optimaler sequentieller. Man kann sich also vorstellen, daß die Arbeit des optimalen sequentiellen Algorithmus nur gleichmäßig auf die Prozessoren verteilt wird.

Wann können wir einen WT–Algorithmus mit dem WT-scheduling Prinzip so implementieren, daß die Implementation einen Speedup von $\Omega(p(n))$ über einen optimalen sequentiellen Algorithmus hat? Nach Folgerung 1.28 (b) auf jeden Fall darf die Arbeit des WT–Algorithmus nicht größer sein, als die eines optimalen sequentiellen Algorithmus sein. Wie definieren also:

Definition 1.29 Ein WT–Algorithmus heißt optimal, gdw. er nicht mehr Arbeit als ein optimaler sequentieller Algorithmus für dasselbe Problem braucht.

□

Ließe sich ein optimaler Algorithmus für alle Prozessorzahlen $p(n)$ in Laufzeit $O(\frac{W(n)}{p(n)})$ implementieren, so hätten wir immer einen Speedup von $\Omega(p(n))$ gegenüber einem optimalen sequentiellen Algorithmus, also bestmöglichen Speedup. Nun ist aber nach Satz 1.23 (c) $T^\infty(n)$ eine untere Schranke an jede Implementierung eines WT–Algorithmus. Der maximale Speedup wird nur für die Prozessorzahlen erreicht, wo die Laufzeit $O(\frac{W(n)}{p(n))}$ ist, also wo nach Satz $p(n) = O(\frac{W(n)}{T^\infty(n)})$ gilt.

Die Umkehrung dieser Aussage gilt auch:

Satz 1.30 Sei B ein optimaler sequentieller Algorithmus für ein Problem mit Zeit $T^*(n)$. Sei A' ein (im Sinne der letzten Definition) optimaler WT–Algorithmus für dasselbe Problem mit paralleler Zeit $T\infty(n)$. Dann gilt: Das WT–scheduling Prinzip gibt uns einen Algorithmus A mit $Speedup_{A,B}(n) = \Omega(p(n))$ gdw.

$$p(n) = O(\frac{T_B(n)}{T^\infty(n)}) = O(\frac{W(n)}{T^\infty(n)}),$$

wobei $W(n)$ die Arbeit von A' ist (und damit $W(n) = O(T_B(n))$, da A' optimal ist).

Beweis:

„$\Leftarrow$“ Ist $p(n) = O(\frac{W(n)}{T^\infty(n)})$, so ist nach Satz **1.24 c)**

$$T_A(n) = O(\frac{W(n)}{p(n)}).$$

Dann ist $Speedup_{A,B}(n) = \Omega(p(n))$, da $W(n) = \Theta(T_B(n))$ ist.

„$\Rightarrow$“ Ist $\frac{T_B(n)}{T_A(n)} = Speedup_{A,B}(n) = \Omega(p(n))$, so folgt $p(n) = O(\frac{T_B(n)}{T_A(n)}) = O(\frac{W(n)}{T\infty(n)})$, da $T_A(n) \geq T^\infty(n)$.

□

Der Satz zeigt uns, daß optimale WT–Algorithmen umso mehr Prozessoren mit maximalen Speedup nutzen können, je kleiner ihre parallele Zeit $T^\infty(n)$ ist.

1.3 Kommunikationskomplexität

Neben Laufzeit und Speicherplatz beeinflußt noch ein weiterer Parameter die Effizienz eines PRAM–Algorithmus: die Menge der Kommunikation.

Definition 1.31 Sei A ein Algorithmus für die PRAM. Die Kommunikationskomplexität von A, $C_A(n)$, ist definiert durch

$$C_A(n) \quad = \quad \text{Max } \{ \quad \begin{array}{l} \text{\# Bits, die bei Eingabe } E \text{ zwischen dem globalen} \\ \text{und irgendeinem lokalen Speicher hin und her trans} \\ \text{portiert werden} \mid \text{E ist ein Eingabe der „Größe" } n \quad \}. \end{array}$$

□

Wir wollen am Beispiel der Matrixmultiplikation zeigen, daß unser WT–scheduling Prinzip nicht unbedingt Algorithmen mit minimaler Kommunikationskomplexität liefert. Zunächst einmal ein WT–Algorithmus für die Matrixmultiplikation.

Algorithmus 1.32

Eingabe: Zwei $(n \times n)$–Matrizen A und B im globalen Speicher, wobei $n = 2^k$ für eine Zahl $k > 0$.

Ausgabe: Das Produkt $C = A \cdot B$ im globalen Speicher.

begin
  1.for $1 \leq i, j \leq n$ pardo
    $C^{'}(i,j,l) \ := A(i,l) \cdot B(l,j)$
  2.for $h = 1$ to $\log n$ do
    for $1 \leq i, j \leq n$ and $1 \leq l \leq \frac{n}{2^k}$ pardo
      $C^{'}(i,j,l) \ := C^{'}(i,j,2l-1) + C^{'}(i,j,2l)$
  3.for $1 \leq i, j \leq n$ pardo
    $C(i,j) \ := C^{'}(i,j,1)$
end

Die Laufzeit ist $O(\log n)$ und die Arbeit $O(n^3)$. Für die PRAM mit p Prozessoren gibt uns das WT–scheduling Prinzip einen Algorithmus mit Laufzeit $O(\frac{n^3}{p} + \log n)$. Ist nun $p = p(n) = n$, so haben wir eine Laufzeit von $O(n^2)$.

Wir ermitteln die Kommunikationskomplexität des Algorithmus, den uns das WT–scheduling Prinzip bei n verfügbaren Prozessoren gibt. Dazu schauen wir uns an, wie die Instruktionen auf die Prozessoren verteilt werden.

Statement 1: Jeder Prozessor bekommt n^2 viele Instruktionen zur Ausführung. Prozessor P_i berechnet

$$C^{'}(i,j,l) \ := A(i,l) \cdot B(l,j)$$

für alle $1 \leq j,l \leq n$. Dazu muß P_i die i–te Zeile von A und ganz B in seinen lokalen Speicher lesen. D.h. $\Theta(n^2)$ Zahlen werden zwischen jedem P_i und dem globalen Speicher transportiert.

Statement 2: Die h–te Iteration verlangt die Ausführung von $n^2 \cdot \frac{n}{2^k}$ vielen Instruktionen. Diese können folgendermaßen verteilt werden: P_i berechnet alle $C^{'}(i,j,l)$ für $1 \leq i,j \leq n$. Das erfordert wieder eine Kommunikation von $\Theta(n^2)$ vielen Zahlen.

Statement 3: Kommunikation von $\Theta(n^2)$ Zahlen, da P_i alle Zahlen $C^{'}(i,j,1)$ für alle j in seinem lokalen Speicher hat.

Insgesamt ist also die Kommunikationskomplexität für die die angegebene Implementation $\Omega(n^2)$.

Als nächstes geben wir eine andere Implementierung auf der PRAM mit n Prozessoren an, so daß die Kommunikationskomplexität nur noch $O(n^{\frac{4}{3}} \cdot \log n)$ ist. Dabei erhöht sich nicht die Laufzeit! Nehmen wir an, daß $\alpha = \sqrt[3]{n}$ eine ganze Zahl ist. (Also $n \in \{8, 27, 64, 125, 216, ...\}$). Wir teilen A in $(n^{\frac{2}{3}} \times n^{\frac{2}{3}})$–Matrizen ein, und zwar so:

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,\alpha} \\ A_{2,1} & & & \vdots \\ \vdots & & & \vdots \\ A_{\alpha,1} & & \cdots & A_{\alpha,\alpha} \end{pmatrix}.$$

Ebenso B und C. Beachte, daß es genau n Paare $(A_{i,l}, B_{l,j})$ gibt, da
$1 \leq i,j,l \leq \sqrt[3]{n}$.

Jeder Prozessor liest genau ein Paar $(A_{i,l}, B_{l,j})$ in einen lokalen Speicher und berechnet $D_{i,j,l} = A_{i,l} \cdot B_{l,j}$, das dann in dem globalen Speicher abgelegt wird. Damit haben wir $O(n^{\frac{4}{3}})$ Zahlen hin- und hergeschickt. Die Laufzeit dieser Schritte ergibt sich zu $O((n^{\frac{2}{3}})^3) = O(n^2)$.

Jeder Block $C_{i,j}$ der Produktmatrix ergibt sich als $C_{i,j} = \sum_{l=1}^{\alpha} D_{i,j,l}$ und wir haben $n^{\frac{2}{3}}$ solcher Blöcke. Wir berechnen jedes $C_{i,j}$ durch $\sqrt[3]{n} = \alpha$ viele Prozessoren nach dem Muster des binären Baums. Für jede Stufe des Baums haben wir pro Prozessor höchstens die Kommunikation eines Blocks, also von $O(n^{\frac{4}{3}})$ Zahlen. Insgesamt gibt das eine Kommunikation von $O(n^{\frac{4}{3}} \cdot \log n)$. Die Laufzeit dieses Schrittes ergibt sich zu $O(n^{\frac{2}{3}} \cdot \log n)$. $\square$

2 Grundlegende Techniken

Die Erstellung effizienter paralleler Algorithmen ist komplizierter als die sequentieller. Wichtig ist die Kenntnis grundlegender, einfacher Techniken, die hinterher bei komplizierten Problemen verwendet werden.

2.1 Balancierte Bäume

In Algorithmus 1.20 haben wir gesehen, wie die Strukturierung der Rechnung als binärer Baum die WT–Darstellung eines Algorithmus zur Berechnung der Summe von $n = 2^k$ Elementen gibt, wobei $W(n) = O(n)$ ist und $T^\infty(n) = O(\log n)$ ist. Da jeder sequentielle Algorithmus mindestens n Additionen braucht, haben wir einen *optimalen* WT–Algorithmus von logarithmischer Laufzeit. Frage: Für welche Prozessorenzahlen $p(n)$ gibt uns also das WT–scheduling Prinzip einen speedup von $\Omega(p(n))$?

In diesem Abschnitt wenden wir das Prinzip der Strukturierung der Rechnung als binären Baum auf das Präfix–Summen Problem an.

Definition 2.1 (Präfix–Summen Problem) Die Operation $*$ sei assoziativ. D.h.

$$x * (y * z) = (x * y) * z$$

(z.B. $*$ = Addition, daher der Name Präfix–Summen Problem).

Eingabe: Eine Folge von n Elementen $(x_1, ..., x_n)$.

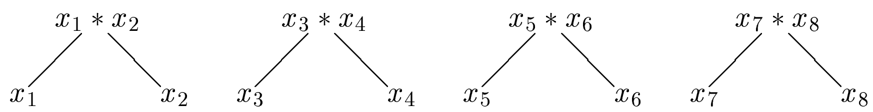
Ausgabe: Die Folge der Präfix–Summen $(s_1, ..., s_n)$, wobei $s_i = x_1 * x_2 * \cdots * x_i$ für alle i zwischen 1 und n.

□

Ein sequentieller Algorithmus für das Präfix–Summen Problem ist leicht zu bekommen: Berechne erst s_1, dann $s_2 = s_1 * x_2$ dann $s_n = s_{n-1} * x_n$. Für den Algorithmus gilt $T(n) = \Theta(n)$. Diesen Algorithmus können wir auch mit mehreren Prozessoren nicht ohne weiteres beschleunigen, da das Ergebnis von Schritt i in Schritt $i + 1$ benötigt wird. Der Algorithmus ist *inhärent sequentiell* . Wir können davon ausgehen, daß der angegebene Algorithmus ein optimaler sequentieller Algorithmus ist.

Das folgende Beispiel zeigt, wie wir durch Strukturierung der Rechnung in einen binären Baum das Problem mit mehreren Prozessoren schneller berechnen können.

Beispiel 2.2 Sei $n = 2^k$ für k$\geq$0, hier k = 3. Eingabe ist die Folge $(x_1, ..., x_8)$. In der untersten Stufe des Baums werden jeweils 2 benachbarte Blätter zusammengefaßt:

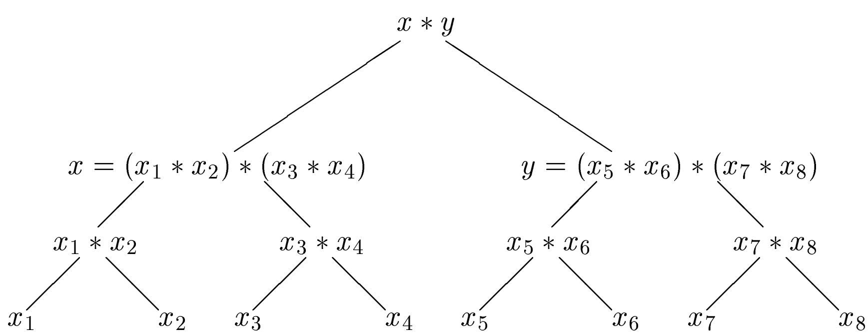


Angenommen wir haben die Präfix–Summen für $(x_1 * x_2, ..., x_7 * x_8)$ als Folge $S^{'} = (s_1^{'}, ..., s_4^{'})$ berechnet. Dann bekommen wir die Präfix–Summen $S = (s_1, ..., s_8)$ von $(x_1, ..., x_8)$ mit Hilfe von $S^{'}$ in paralleler Zeit von $O(1)$:

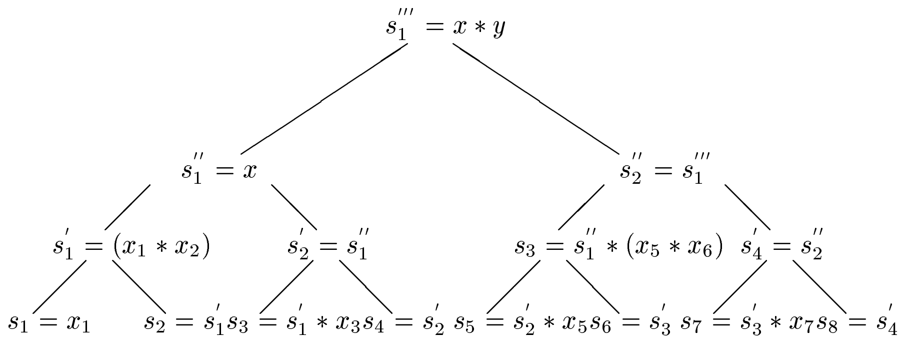
$$\begin{array}{llll} s_1 = x_1, & s_2 = s_1^{'}, & s_3 = s_1^{'} * x_3, & s_4 = s_2, \\ s_5 = s_2^{'} * x_5, & s_6 = s_3^{'}, & s_7 = s_3^{'} * x_7, & s_8 = s_4^{'}. \end{array}$$

Diese Vorgehensweise gibt einen rekursiven Algorithmus, der durch 2–maliges Durchlaufen eines Baumes veranschaulicht werden kann:

1. Durchlauf: Aufbau des Baums von unten nach oben.



2. Durchlauf: Abbau des Baums von oben nach unten unter Berechnung der Präfix–Summen der Knoten in einer Höhe.



Die Verallgemeinerung dieser Vorgehensweise gibt einen WT–Algorithmus mit folgenden Eigenschaften:

- Zeit für den 1. Durchlauf = $O(\log n)$.
- Zeit für den 2. Durchlauf = $O(\log n)$.
- Arbeit im 1. Durchlauf = n-1.
- Arbeit im 2. Durchlauf $\leq$ 2n.

Also ist

$$T^{\infty}(n) = O(\log n) \quad \text{und} \quad W(n) = O(n).$$

Der Algorithmus ist damit optimal und von logarithmischer Zeit. □

Damit ergibt sich ein WT–Algortihmus zur Berechnung der Präfix–Summen:

Algorithmus 2.3

Eingabe: Ein Feld aus $n = 2^k$ Elementen der Form $(x_1, ..., x_n)$, wobei k $\geq$ 0.

Ausgabe: Auf der Folge von Speicherplätzen $S = (s_1, ..., s_n)$ die Präfix–Summen von $(x_1, ..., x_n)$.

```
begin
  1.if n = 1 then
      begin s_1 := x_1; exit end
```

2. for $1 \leq i \leq \frac{n}{2}$ pardo
   $y_i := x_{(2i-1)} * x_{2i}$
   od
3. Berechne rekursiv die Präfix–Summen von $(y_1, ..., y_{\frac{n}{2}})$.
   { Die Präfix–Summen stehen jetzt in den Speicherplätzen $(s_1, ..., s_{\frac{n}{2}})$.}
4. $(z_1, ..., z_{\frac{n}{2}}) := (s_1, ..., s_{\frac{n}{2}})$
5. for $1 \leq i \leq n$ pardo
   if i gerade then $s_i := z_{(i/2)}$;
   if i = 1 then $s_1 := x_1$;
   if i > 1 und i ungerade then $s_i := z_{(i-1)/2} * x_i$
   od
end

Die globalen Variablen des Programms sind $s_1, ..., s_n$ und $x_1, ..., x_n$, die Variablen $y_1, ..., y_n$, $y_1, ..., y_{\frac{n}{2}}$ sind lokal. □

Satz 2.4 Der Präfix–Summen Algorithmus 2.3 hat die folgenden Eigenschaften:

(a) Für i mit $1 \leq i \leq n$ gilt: $s_i = x_1 * \cdots * x_i$.

(b) $T^\infty(n) = O(logn)$.

(c) $W(n) = O(n)$.

Beweis

(a) Beweis durch Induktion über k, wobei wir annehmen, daß $n = 2^k$.

Induktionsanfang k = 0, wegen Schritt 1.

Induktionsschluß Die Induktionsvoraussetzung lautet: Der Algorithmus ist korrekt für alle Folgen der Länge $n = 2^{k-1}$. Nach Induktionsvoraussetzung enthalten $(z_1, ..., z_{\frac{n}{2}})$ nach Schritt 4 die Präfix–Summen von $(x_1 * x_2, ..., x_{n-1} * x_n)$. Dann folgt die Behauptung durch Inspektion von statement 5.

(b) Induktion über k. Für $k = 1$ gilt die Behauptung. Gelte die Behauptung für k und sei $n = 2^{k+1}$. Schritte 2, 4 und 5 brauchen konstante parallele Zeit, sagen wir a. Zu Schritt 3: Die Berechnung der Präfix–Summen von $(y_1, ..., y_{\frac{n}{2}})$ braucht $\leq T^\infty(\frac{n}{2})$ an Zeit. Die Implementation des rekursiven Aufrufs auf einer PRAM mit unendlich vielen

Prozessoren ist auch in *konstanter* paralleler Zeit möglich. Auf einer PRAM mit unendlich vielen Prozessoren gilt also für die Zeitfunktion $T^{\infty}(n)$:

$$T^{\infty}(n) \leq T^{\infty}(\frac{n}{2}) + a$$

und

$$T^{\infty}(1) = 1.$$

Dann gilt also mit $T = T^{\infty}$, daß

$$\begin{array}{rcl} T(n) & \leq & T(\frac{n}{2}) + a \leq T(\frac{n}{4}) + 2a \\ & \leq & T(\frac{n}{8}) + 3a \leq \cdots \leq T(1) + (\log n) \cdot a = O(\log n). \end{array}$$

(c) Schritt 1 verbraucht konstante Arbeit $O(1)$, Schritt 2 Arbeit von $O(\frac{n}{2})$, Schritt 4 von $O(\frac{n}{2})$, Schritt 5 von $O(n)$. Die Implementation des rekursiven Aufrufs kann mit $O(\frac{n}{2})$ Arbeit geschehen. Damit gilt für die Arbeit $W(n)$ und geeignete Konstante b:

$$W(n) \leq W(\frac{n}{2}) + b \cdot n$$

und

$$W(1) = b.$$

Damit folgt

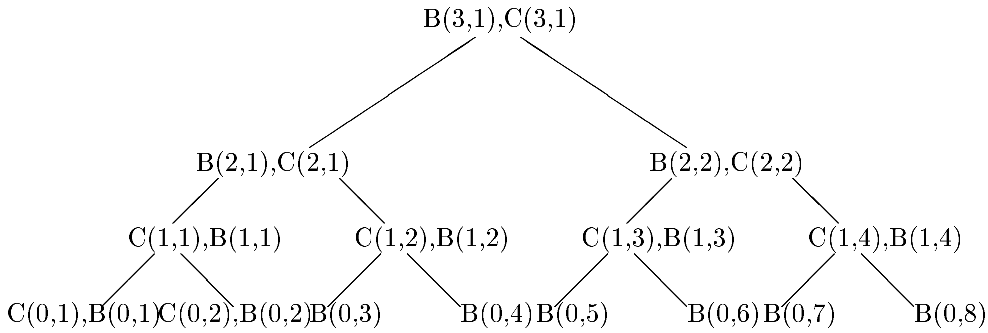
$$\begin{array}{ll} W(n) & \leq W(\frac{n}{2}) + b \cdot n \leq W(\frac{n}{4}) + b \cdot n + b \cdot \frac{n}{2} \\ & \leq \cdots \leq \qquad b \cdot \sum_{i=0}^{\log n} \frac{n}{2^i} = O(n). \end{array}$$

□

Man beachte, daß Algorithmus 2.3 auf einer EREW–PRAM implementierbar ist.

Das nächste Ziel, ein nicht–rekursiver Algorithmus für das Präfix–Summen Problem, ist auch nicht weiter kompliziert.

Beispiel 2.5 Der nicht–rekursive Algorithmus geht nach demselben Prinzip vor wie Algorithmus 2.3. Die Werte an den Knoten des Baums werden in Variablen gespeichert:



Die Knoten $B(i,j)$ enthalten die Werte des 1. Durchlaufs, die Knoten $C(i,j)$ die des 2. Durchlaufs. □

Algorithmus 2.6

Eingabe: Ein Feld A der Größe $n = 2^k$ mit $k \geq 0$.

Ausgabe: Ein Feld C, so daß $C(0,j)$ die j–te Präfix–Summe von A für $1 \leq j \leq n$ enthält.

```
begin
  1.for 1 ≤ j ≤ n pardo
      B(0,j) := A(j)
  2.for h = 1 to log n do
      for 1 ≤ j ≤ n/2^h pardo
        B(h,j) := B(h-1,2j-1) × B(h-1,2j)
  3.for h = log n to 0 do
      for 1 ≤ j ≤ n/2^h pardo
        if j gerade then C(h,j) := C(h+1, 0.5j)
          { Hier ist h < log n. }
        if j = 1 then C(h,1) := B(h,1)
          { Hier wird der Fall h = log n mit erledigt. }
        if j ungerade und j > 1 then C(h,j) := C(h+1,0.5(j-1)) × B(h,j)
          { Hier ist h < log n. }
end
```

□

Hinweis: An diesem Punkt sollten Sie Aufgabe 1 des 3. Übungsblattes angehen.

2.2 Pointer jumping

Die pointer jumping Technik erlaubt die schnelle parallele Verarbeitung von Daten, die als Menge von gerichteten Wurzelbäumen gespeichert sind.

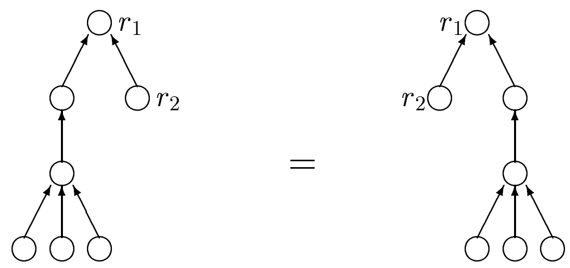
Definition 2.7 (Gerichteter Wurzelbaum) Ein gerichteter Wurzelbaum T ist ein gerichteter Graph, mit folgenden Eigenschaften:

(a) Genau ein Knoten r ist als Wurzel ausgezeichnet.

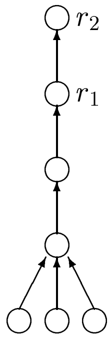
(b) Für jeden Knoten $v \neq r$ gibt es genau eine Kante mit v als Anfangspunkt.

(c) Für jeden Knoten v gibt es einen gerichteten Pfad von v nach r.

Beispiel:
Ein gerichteter Wurzelbaum mit Wurzel r_1 ist:



Man beachte, daß wir die Bäume nicht nach der Position der Söhne unterscheiden, wohl aber danach, welcher Knoten Wurzel ist: Mit Wurzel r_2 hat der Baum die Struktur



Man beachte, daß sich die Richtung der Kante zwischen r_1 und r_2 umgekehrt hat. □

Definition 2.8 (Problem: Wurzeln im Wald) Das Problem, Wurzeln im Wald zu finden ist definiert durch:

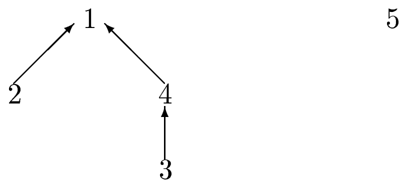
Eingabe: Ein Wald F, bestehend aus einer Menge paarweise disjunkter, gerichteter Wurzelbäume. Der Wald F ist spezifiziert durch ein Feld P (für *predecessors*) über $\{1, ..., n\}$ mit:

$$\begin{array}{lcl} P(i) = j \text{für } i \neq j & \Leftrightarrow & \text{Es gibt eine Kante} \\ & & \text{von Knoten } i \text{ nach } j. \\ P(i) = i & \Leftrightarrow & \text{Knoten } i \text{ ist eine Wurzel.} \end{array}$$

Ausgabe: Ein Feld S über $\{1, ..., n\}$, so daß für alle i gilt:

$$\begin{array}{lcl} S(i) = j & \Leftrightarrow & \text{Der Knoten } j \text{ ist die Wurzel des Baums, der} \\ & & \text{den Knoten } i \text{ enthält.} \end{array}$$

Beispiel
Ein Wald über der Knotenmenge {1,2,3,4,5} ist:



Das Feld ist gegeben durch $P = (1, 1, 4, 1, 5)$.

Das Feld S ist $S = (1, 1, 1, 1, 5)$.

Ein einfacher sequentieller Algorithmus für das Problem besteht aus 3 Schritten:

1. Schritt	Erzeugung der Wurzeln.
2. Schritt	Umkehren der Kanten
3. Schritt	Depth first search in den Bäumen.

Mit dem Wald aus dem letzten Beispiel wird also so verfahren:
Eingabe: $P = (1, 1, 4, 5, 1)$

1. Umkehren der Richtung der Kanten von P im Feld Q:

$$\begin{array}{rl} \mathrm{Q} = & (0,0,0,0,0) \\ \mathrm{Q} = & (1,0,0,0,0) \\ \mathrm{Q} = & ((1,2),0,0,0,0) \\ \mathrm{Q} = & ((1,2),0,3,0,0) \\ \mathrm{Q} = & ((1,2,4),0,3,0,0) \\ \mathrm{Q} = & ((1,2,4),0,3,0,5) \end{array}$$

2. Ermitteln der Wurzeln aus P und eintragen in S ergibt:

$$S = (1, 0, 0, 0, 0, 5)$$

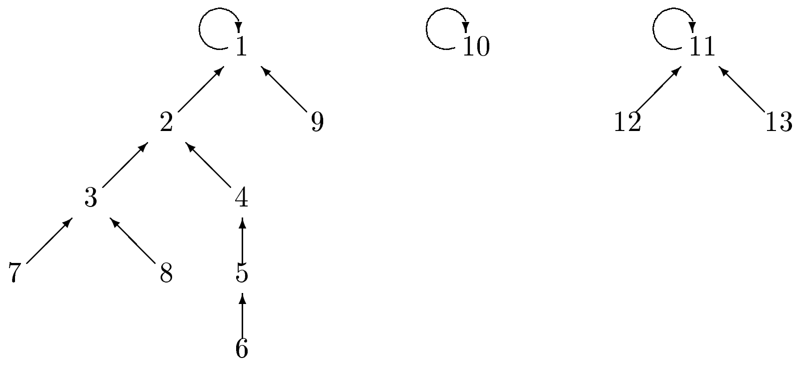
3. Depth first search durch den in Q gegebenen Wald. Eintragen der gefundenen Söhne in S ergibt schließlich

$$\mathrm{S} = (1,1,1,1,5)$$

Die Laufzeit dieses Verfahrens ist $O(n)$. Man beachte, daß die depth–first–search linear in der Knotenanzahl der Bäume ist. □

Wir konstruieren einen parallelen Algorithmus der in Zeit $O(\log h)$ läuft, wobei h die maximale Höhe der einzelnen Bäume ist. Das ist bemerkenswert, denn einen solchen Algorithmus können wir nicht durch ein einfaches scheduling des sequentiellen Algorithmus bekommen, da dieser “sequentiell entlang der Höhe” ist. Die Idee des parallelen Algorithmus zeigt das folgende Beispiel.

Beispiel 2.9 Der Wald ist gegeben durch 3 Bäume.



Eingabe des Problems ist das 13–dimensionale Feld P mit

$$P = (1, 1, 2, 2, 4, 5, 2, 2, 1, 10, 11, 11, 11).$$

Der Algorithmus arbeitet mit dem 13–dimensionalen Feld S wie folgt:

1.Schritt Initialisierung Von jedem Knoten gehen wir 1 Schritt zurück, sofern möglich, d.h. sofern wir keine Wurzel vorliegen haben.
Also $S = P$.

2. Schritt: Von jedem Knoten 2 Schritte zurück, sofern möglich. Das wird erreicht durch die Zuweisung S(i) := S(S(i)) für alle i.

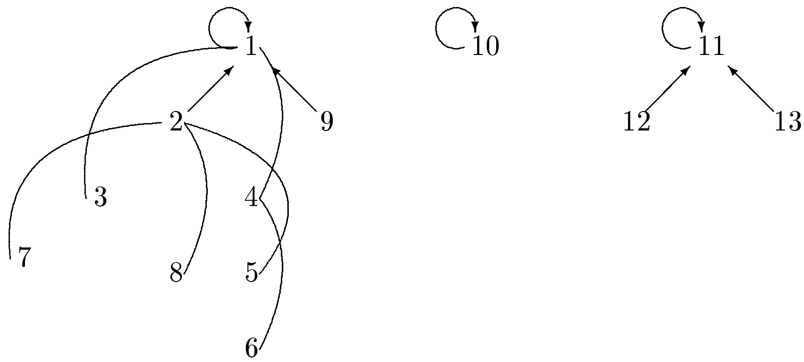
3. Schritt: Von jedem Knoten 4 Schritte zurück, sofern möglich. Zuweisung S(i) := S(S(i)) für alle i.

$\vdots$

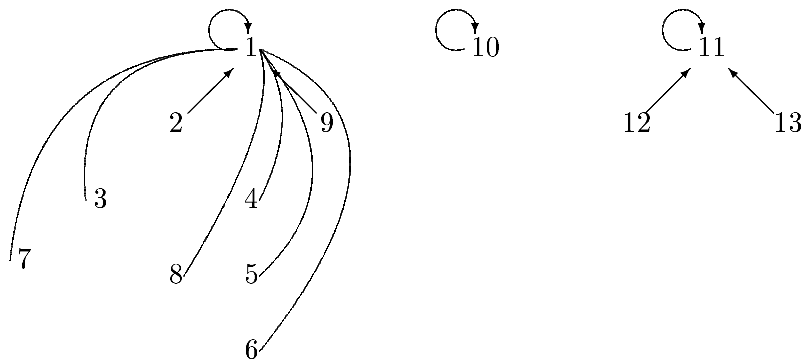
Und so weiter.

□

Wir stellen den Zustand des Feldes S durch einen Wald dar: Nach dem 1. Schritt ist $S = P$ wie oben.
2. Schritt



Hier ist $S(7) = 2$ und $S(2) = 1$.
3. Schritt



□

Die beschriebene Technik nennt man pointer jumping oder auch path doubling. Als Algorithmus in der WT–Darstellung sieht das folgendermaßen aus:

Algorithmus 2.10

Eingabe: Ein Wald von disjunkten, gerichteten Wurzelbäumen über einer Knotenmenge $\{1, ..., n\}$ spezifiziert durch das Feld P.

Ausgabe: Ein Feld S, so daß $S(i)$ die Wurzel des Baums, der den Knoten i enthält, ist.

begin

```
1.for 1 ≤ i ≤ n pardo
    S(i) := P(i)
    while S(i) ≠ S(S(i)) do
      S(i) := S(S(i))
end
```

□

Satz 2.11 Für Algorithmus 2.10 gilt:

(a) $S(j)$ = Wurzel des Baums, der den Knoten j enthält.

(b) $T(n, h) = O(\log h)$, wobei h die maximale Höhe der beteiligten Bäume ist, n die Gesamtknotenzahl.

(c) $W(n, h) = O(n \cdot \log h)$

Beweis:

(a) Sei i ein Knoten. Induktiv über $j \geq 0$ definieren wir den j–ten Vorgänger des Knotens i. Der 0–te Vorgänger von i ist i.

$$(j+1)\text{–ter Vorgänger von } i = \begin{cases} i & \text{falls } i \text{ Wurzel ist} \\ j\text{–ter Vorgänger} & \text{falls } k \text{ der Vater} \\ \text{von } k & \text{von } i \text{ ist.} \end{cases}$$

Es gilt die folgende Eigenschaft für die while–Schleife:
Sei $j \geq 0$ und gilt vor der while–Schleife für alle Knoten i

$$S(i) = j\text{–ter Vorgänger von } i$$

dann gilt danach für alle Knoten i

$$S(i) = (2 \cdot j)\text{–ter Vorgänger von } i.$$

Das impliziert die Korrektheit durch Induktion über die Anzahl der Schleifendurchläufe. Beachte, daß vor Beginn der Schleife gilt

$$S(i) = 1\text{–ter Vorgänger von } i.$$

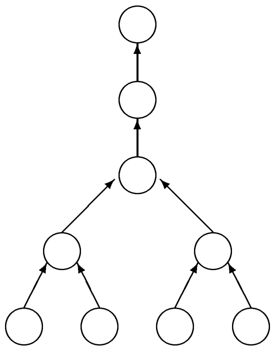
(b) Die Maximalzahl der Schleifendurchläufe für ein festes i ist $\log h$. Die Anzahl der Instruktionen pro Durchlauf und festes i ist konstant. Also folgt die Behauptung.

(c) Die Schleife wir für alle i gleichzeitig durchlaufen. Also ist $W(n) = O(n \cdot \log h)$.

□

Man sieht, traurigerweise: Der parallele Algorithmus ist *nicht* optimal, da der vor Beispiel 2.9 angegebene sequentielle Algorithmus in linearer Zeit läuft.

Frage: Wird der Algorithmus 2.10 durch das WT–scheduling Prinzip auf eine PRAM mit $p(n)$ Prozessoren gebracht, so daß der speedup $\Omega(p(n))$ ist? Man braucht zur eigentlichen Implementierung eine concurrent read PRAM, da $S(S(i))$ von mehreren Prozessoren gelesen werden kann. Zum Beispiel in der Situation



Definition 2.12 (Problem paralleler Präfix auf Bäumen)

Eingabe: Ein Wald F wie in Definition 2.8. Wir nehmen zusätzlich an, daß jeder Knoten ein Gewicht hat, das durch das Feld W gegeben ist:

$$W(i) \quad = \quad \text{Gewicht von Knoten } i$$

Ist r eine Wurzel, so ist $W(r) = 0$.

Ausgabe: Das Feld W, wobei jetzt

$$W(i) \quad = \quad \text{Summe der Gewichte auf dem Pfad von } i \text{ zur Wurzel}$$

□

Aus Algorithmus 2.10 bekommen wir direkt:

Algorithmus 2.13

Eingabe: Wie oben, d.h. in Definition 2.12

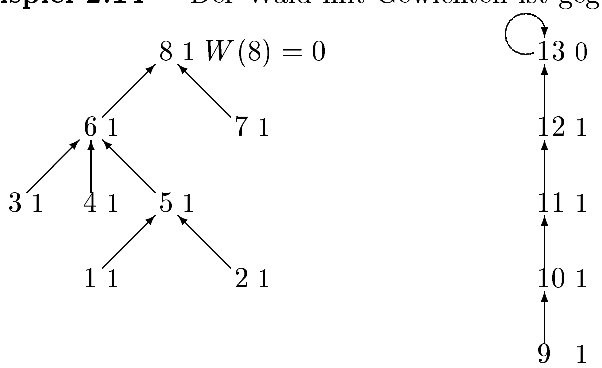
Ausgabe: Auch wie oben

```
begin
  1.for 1 ≤ i ≤ n pardo
        S(i) := P(i)
        while S(i) ≠ S(S(i)) do
          W(i) := W(i) + W(S(i))
          S(i) := S(S(i))
end
```

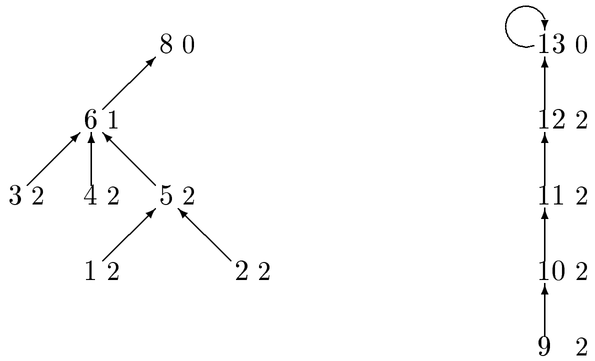
□

Wie vorher brauchen wir hier eine CREW–PRAM zur Implementierung.

Beispiel 2.14 Der Wald mit Gewichten ist gegeben durch:



Alle Knoten außer den Wurzeln haben das Gewicht 1.
1. Schritt



Der 2. Schritt gibt uns dann das Ergebnis. □

Bemerkung:
Man beachte, daß die pointer jumping Technik insbesondere auf Wälder der Art:

anwendbar ist, also auf *linked lists*. Dort haben wir eine Zeitkomplexität von $O(\log n)$ und eine Arbeit von $O(n \cdot \log n)$. Das Problem auf Listen heißt *paralleles Präfix*.

2.3 Divide and Conquer (teile und herrsche)

Die divide and conquer Technik besteht aus 3 Schritten:

1. Das gegebene Problem wird in eine Anzahl von Teilproblemen, die möglichst gleich groß, aber kleiner als das ursprüngliche Problem sind, geteilt.

2. Die Teilprobleme werden unabhängig voneinander rekursiv gelöst.

3. Die Lösungen der Teilprobleme werden wieder zusammengesetzt, um eine Lösung des ursprünglichen Problems zu erhalten.

Divide and conquer ist ein bekanntes Prinzip aus der Theorie der sequentiellen Algorithmen. Quicksort ist ein Beispiel eines divide and conquer Algorithmus. Divide and conquer Algorithmen bieten sich auf natürliche Weise zur Implementierung auf PRAM's an: Die in Schritt 2 zu lösenden Teilprobleme sind volkommen unabhängig und können deshalb ohne weiteres parallel behandelt werden. Wir führen diese Technik an einem Beispiel aus der algorithmischen Geometrie ein.

Definition 2.15 (Problem: konvexe Hülle)

Eingabe: Eine Menge $S = \{p_1, ..., p_n\}$ von Punkten in der Ebene gegeben durch x- und y–Koordinaten: $p_i = (x_i, y_i)$.

Ausgabe: Eine geordnete Liste von Punkten $KH(S) \subseteq S$, so daß $KH(S)$ die Eckpunkte des kleinsten konvexen Polygons darstellt, das S enthält.

Dabei ist definiert:

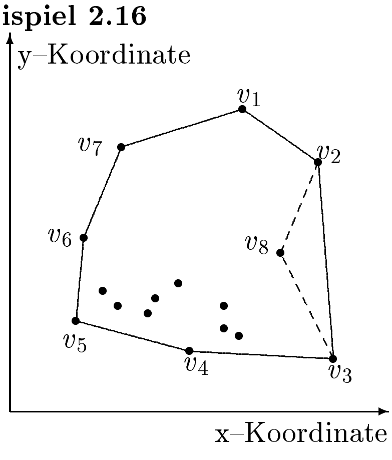
Ein Polygon Q ist konvex

$\Longleftrightarrow$

Sind p, q zwei Punkte in Q, so gehört die Strecke von p nach q ganz zu Q.

□

Beispiel 2.16



Die konvexe Hülle der Punktmenge, nennen wir sie S, ist:

$$KH(S) = (v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8)$$

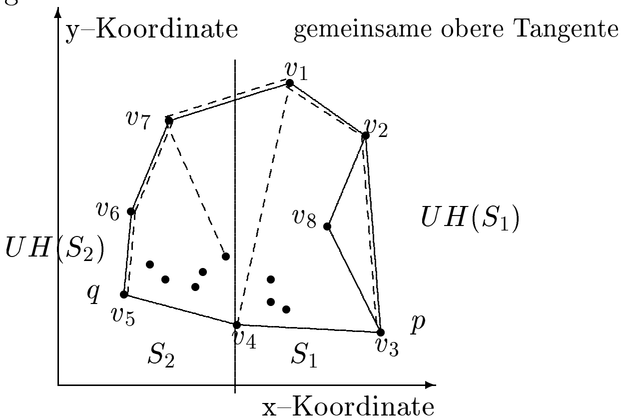
Man beachte, daß das durch den Vektor von Eckpunkten

$$(v_1, v_2, v_8, v_3, v_4, v_5, v_6, v_7)$$

gegebene Polygon nicht konvex ist. $\square$

Das Problem der konvexen Hülle ist eines der grundlegenden Probleme der algorithmischen Geometrie. Das Problem ist in sequentieller Zeit von $O(n \cdot \log n)$ lösbar, und $\Omega(n \cdot \log n)$ ist auch eine untere Schranke für die Laufzeit der sequentiellen Algorithmen für das Problem. Im folgenden Beispiel führen wir einen sequentiellen Algorithmus ein.

Beispiel 2.17 (Fortsetzung von Beispiel 2.16) Wir betrachten die Menge von Punkten S aus 2.16



Sei

$$\begin{array}{rcl} p & = & \text{der Punkt mit der größten x–Koordinate} \\ q & = & \text{der Punkt mit der größten y–Koordinate} \end{array}$$

Also hier ist $p = v_3$ und $q = V_5$. Dann muß gelten: $\quad p, q \in KH(S)$ Die Punkte p und q teilen $KH(S)$ in eine obere und eine untere Hülle:

$$\begin{array}{rcl} OH(S) & = & (v_5, v_6, v_7, v_1, v_2, v_3) \\ UH(S) & = & (v_3, v_4, v_5) \end{array}$$

(Wir ordnen die Hüllen immer im Uhrzeigersinn.)

Wir zeigen, wie $OH(S)$ bestimmt werden kann. Die Bestimmung von $UH(S)$ geht analog.

Wir machen der Einfachheit halber folgende Annahmen:

- $n =$ Anzahl der Punkte ist eine 2–er Potenz. (Hier im Beispiel $n = 16$).
- Die x- und y–Koordinaten der Punkte sind paarweise verschieden.

Wir sortieren die Punkte nach ihren x–Koordinaten. D.h. wir haben die Punkte als $p_1, p_2, ...p_n$ vorliegen mit

$$\text{x–Koordinate } p_1 \leq \text{x–Koordinate } p_2 \leq \cdots \leq \text{x–Koordinate } p_n.$$

Wir teilen die Punktmenge S in 2 gleich große Hälften ein (beachte, daß n eine 2–er Potenz ist):

$$S_1 = (p_1, p_2, ..., p_{\frac{n}{2}}) \quad \text{und} \quad S_2 = (p_{\frac{n}{2}+1}, ..., p_n).$$

Angenommen, wir haben (rekursiv, gemäß dem divide and conquer Prinzip) $OH(S_1)$ und $OH(S_2)$ bestimmt.

Die gemeinsame obere Tangente von $OH(S_1)$ und $OH(S_2)$ ist die gemeinsame Tangente von $OH(S_1)$ und $OH(S_2)$, so daß $OH(S_1)$ und $OH(S_2)$ unter ihr sind. Die gemeinsame obere Tangente ergibt sich durch Bestimmung der Punkte mit größter y–Koordinate. Im Beispiel ist die Strecke zwischen v_1 und v_7 die gemeinsame obere Tangente.

$OH(S)$ ergibt sich nun durch Verbindung über die gemeinsame obere Tangente:

$$OH(S) = (\underbrace{v_5, v_6, v_7,}_{\text{Von } OH(S_1)} \underbrace{v_1, v_2, v_3}_{\text{Von } OH(S_2)}).$$

Zur sequentiellen Laufzeit $T(n)$ dieser Methode:

- Sortieren der n Punkte nach dem Wert der x–Koordinate $O(n \cdot \log n)$.
- Rekursion (mit Aufwand für den Aufruf) $T(\frac{n}{2}) + n$.
- Bestimmung der gemeinsamen oberen Tangente $O(\log n)$. Wie binäres Suchen. Mittleres Element und nachsehen, ob es ansteigt.
- Zusammensetzen $O(n)$. Man braucht es in aufeinander folgenden Speicherplätzen.

Also

$$\begin{aligned} T(n) &\leq c \cdot n \cdot \log n + 2 \cdot T(\tfrac{n}{2}) + c \cdot \log n + c \cdot n \\ &\leq d \cdot n \cdot \log n + 2 \cdot T(\tfrac{n}{2}) \\ &= O(n \cdot \log n) \end{aligned}$$

für geeignete Konstanten c und d. □

Man beachte, daß der in Beispiel 2.17 beschriebene Algorithmus ein optimaler sequentieller Algorithmus ist, vgl. die untere Schranke vor Beispiel 2.17. Der im letzten Beispiel beschreibene Algorithmus läßt sich leicht parallelisieren.

Algorithmus 2.18

Eingabe: Ein Vektor $S = (p_1, ..., p_n)$ von n Punkten mit paarweise verschiedenen x- und y–Koordinaten, so daß

$$\text{x–Koordinate von } p_1 < \text{x–Koordinate } p_2 < \cdots < \text{x–Koordinate } p_n.$$

und n eine 2–er Potenz ist.

Ausgabe: Die obere Hülle von S.

begin
1. if n $\leq$ 4 bestimmen der oberen Hülle durch einfaches Durchsuchen aller Tupel; exit
2. Sei $S_1 = (p_1, p_2, ..., p_{\frac{n}{2}})$ und $S_2 = (p_{\frac{n+1}{2}}, ..., p_n)$.
Berechne $OH(S_1)$ und $OH(S_2)$ rekursiv und parallel.
3. Finde die gemeinsame obere Tangente zwischen $OH(S_1)$ und $OH(S_2)$
und konstruiere die obere Hülle.

end

□

Satz 2.19 Der Algorithmus 2.18 hat die folgenden Eigenschaften:

(a) Er berechnet die obere Hülle.

(b) $T^\infty(n) = O(log^2 n)$ (dabei ist $\log^2 n = (\log n)^2$).

(c) $W(n) = O(n \cdot \log n)$.

Beweis

(a) Induktion über n unter der Annahme, daß der Teilalgorithmus zur Bestimmung der oberen Tangente korrekt ist.

(b) Schritt 1 braucht Zeit $O(1)$, Schritt 2 braucht parallele Zeit $T(\frac{n}{2}) + O(1)$, Schritt 3 braucht $O(\log n)$ sequentielle Zeit. Damit gilt für eine geeignete Konstante a: Da das Zusammensatzen in paralleler Zeit erfolgen soll sei $T(n)T^{\infty}(n)$.

$$\begin{aligned}
T(n) &\leq T(\tfrac{n}{2}) + a \cdot \log n \leq T(\tfrac{n}{4}) + a \cdot \log \tfrac{n}{2} \cdot \log n \\
&\leq T(\tfrac{n}{8}) + a \cdot \log \tfrac{n}{8} + a \cdot \log \tfrac{n}{4} + a \cdot \log \tfrac{n}{2} + a \cdot \log n \\
&\leq a \cdot \left(\textstyle\sum_{i=0}^{\log n} \log \tfrac{n}{2^i}\right) = a \cdot \left(\textstyle\sum_{i=0}^{\log n} (\log n - \log 2^i)\right) \\
&= a \cdot (\log n + 1) \cdot (\log n) - \left(\textstyle\sum_{i=0}^{\log n} \log 2^i\right) \\
&= a \cdot \log^2 n + a \cdot \log n - \tfrac{\log n \cdot (\log n + 1)}{2} \\
&= O(\log^2 n).
\end{aligned}$$

(c) Schritt 1 braucht Arbeit von $O(1)$, Schritt 2 von $2 \cdot W(\frac{n}{2}) + c \cdot n$, Schritt 3 braucht Arbeit von $O(n)$. Insgesamt gilt für eine geeignete Konstante b:

$$\begin{aligned}
W(n) &\leq 2 \cdot W(\tfrac{n}{2}) + b \cdot n \leq \\
4 \cdot W(\tfrac{n}{4}) + b \cdot n + b \cdot n & \\
&\leq 8 \cdot W(\tfrac{n}{8}) + 4 \cdot b \cdot \tfrac{n}{4} + b \cdot n + b \cdot n \\
&\leq \cdots \leq \textstyle\sum_{i=0}^{\log n} b \cdot n = O((\log n) \cdot n).
\end{aligned}$$

□

Nach Satz 2.19 (c) und der Bemerkung vor Beispiel 2.16 ist Algorithmus 2.18 *optimal.* Zur Implementierung brauchen wir eine Möglickeit des concurrent read, zum Zusammenbau von $OH(S_1)$ und $OH(S_2)$ zu $OH(S)$. Wir brauchen kein concurrent write, also kann Algorithmus 2.18 auf einer CREW–PRAM implementiert werden.

2.4 Partitionierung

Die Technik der Partitionierung besteht, ähnlich wie das divide and conquer, aus 3 Schritten.

1. Das gegebene Problem wird in eine Anzahl von Teilproblemen zerlegt. Diese Anzahl ist üblicherweise nicht konstant sondern hängt von der Eingabegröße ab.

2. Die Teilprobleme werden parallel gelöst. Im Gegensatz zum divide and conquer sind die Teilprobleme so klein, daß keine Rekursion erforderlich ist. Die Teilprobleme werden durch einen eigenen Algorithmus direkt gelöst.

3. Das Zusammensetzen der Lösungen der Teilprobleme. Im Gegensatz zum divide and conquer ist das Zusammensetzen hier im allgemeinen sehr einfach, d.h. in $O(1)$ paralleler Zeit ausführbar.

Wir führen die Technik der Partitionierung am Beispiel des Problems zwei geordnete Felder zu mischen, ein.

Definition 2.20 (Mischen zweier geordneter Felder, merging)

Eingabe: Zwei gerordnete Felder A, B von Elementen einer totalen Ordnung $X: \; A = (a_1, ..., a_m)$ und $B = (b_1, ..., b_n)$.

Ausgabe: Die Folge $C = (c_1, c_2, c_3, ..., c_{m+n})$, die geordnet ist und genau die Elemente aus A und B enthält. Man beachte, Vorkommen möglich sind.

□

Es ist leicht, das Problem mit einem sequentiellen Algorithmus von Laufzeit $O(m + n)$ zu lösen. Wir geben einen parallelen Algorithmus an, der in der WT–Darstellung Zeit von $O(\log n)$ und Arbeit von $O(n)$ hat, also einen optimalen Algorithmus mit logarithmischer Zeit (hier nehmen wir an $m = n$).

Definition 2.21 Sei $X = (x_1, ..., x_t)$ eine (nicht unbedingt geordnete) Folge über der total geordneten Menge S, weiterhin sei $x \in S$. (x kann aus X sein, muß es aber nicht.) Der Rang von x in X ist gegeben durch

$$\text{Rang}(\; x \; : \; X) = |\{i | 1 \leq i \leq t, x_i \leq x\}|$$

Sei $Y = (y_1, ..., y_s)$ eine Folge über S. Dann ist

$$\text{Rang}(\; Y \; : \; X) = (r_1, ..., r_s),$$

wobei $r_i = \text{Rang}(y_i, X)$. □

Beispiel

$$X = (25, -13, 26, 31, 54, 7) \quad \text{und} \quad Y = (13, 27, -27)$$

dann

$$\text{Rang}(Y \ : \ X) = (2, 4, 0).$$

□

Für den Rest dieses Abschnitts treffen wir die folgende Annahme

Annahme 2.22 Seien A und B zwei Folgen die wir mischen wollen, etwa $A = (a_1, ..., a_n)$ und $B = (b_1, ..., bm)$, dann nehmen wir an, daß

$$|\{a_1, ..., a_n, b_1, ..., b_m\}| = n + m.$$

D.h. die Elemente in $A \cup B$ sind paarweise verschieden. □

Wir zeigen zunächst, wie wir mit $O(\log n)$ Zeit und $O(n \cdot \log n)$ Arbeit parallel mischen können. Seien A und B gegeben. Sei C durch mischen aus A und B entstanden. Etwa $C = (c_1, ..., c_l)$. Dann gilt:

$$c_i = x \quad \Leftrightarrow \quad \text{Rang}(x : A \cup B) = i.$$

(Man beachte, daß unsere Annahme 2.22 gilt.)

Mischen läuft also auf die Bestimmung des Rangs hinaus. Wie kann man $\text{Rang}(x : A \cup B)$ bestimmen? Es gilt ganz einfach:

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

Da für $x \in A$ $\text{Rang}(x : A)$ bekannt ist (bzw. parallel in $O(1)$ Schritten bestimmt werden kann), reicht es, die Felder $\text{Rang}(A : B)$ und $\text{Rang}(B : A)$ zu bestimmen.

Wir geben einen Algorithmus zur Berechnung von $\text{Rang}(B : A)$ an.

Sei $b \in B$. Binaäre Suche von b in a gibt uns den Index j mit $a_j < b < a_{j+1}$ oder $a_j < b$ und a_j ist das letzte Element von A. (Man beachte wieder Annahme 2.22.) Dann ist $\text{Rang}(b : A) = j$.

Wir bekommen so einen parallelen Algorithmus der in Zeit $O(\log n)$ mit Arbeit von $O(n \cdot \log n)$ läuft. Dieser Algorithmus ist nicht optimal, da wir einen sequentiellen Algorithmus haben, der in Linearzeit läuft. Einen optimalen parallelen Algorithmus mit Laufzeit von $O(\log n)$ bekommen wir nach folgender Idee (der Algorithmus ist dann noch komplizierter):

1. Einteilung (Partitionierung) von A und B in $\frac{n}{\log n}$ Blocks der Länge $\log n$. { Wir nehmen jetzt der Einfachheit halber an, daß sich die Mischung von A und B durch Aneinanderhängen der Mischungen der entsprechenden Blocks ergibt. Im allgemeinen ist es wie gesagt etwas komplizierter. }
2. Mischen der Blocks durch einen sequentiellen Algorithmus, der in Linearzeit läuft. Das Mischen der Blocks geschieht parallel.

Die Arbeit des Verfahrens ist $O(\frac{n}{\log n} \cdot \log n) = O(n)$ also optimal. Die Zeit ist $O(\log n)$ und wir haben einen optimalen Algorithmus von logarithmischer Laufzeit. Wir geben zunächst einen etwas allgemeinen Algorithmus zur Partitionierung an:

Algorithmus 2.23

Eingabe: Zwei Felder $A = (a_1, ..., a_n)$ und $B = (b_1, ..., b_m)$ beide ansteigend geordnet, wobei $\log m$ und $k(m) = \frac{m}{\log m}$ ganze Zahlen sind. (Zum Beispiel sind $m = 4,\ m = 16$ solche Zahlen.)

Ausgabe: Eine Menge von $k(m)$ Paaren (A_i, B_i), von Teilfolgen von A und B, so daß gilt:

- $|B_i| = \log m$ für alle i.
- $\sum |A_i| = n$.
- $A_i > A_{i-1}, A_i > B_{i-1}, B_i > A_{i-1}, B_i > B_{i-1}$ für alle i mit $2 \leq i \leq k(m)$.

```
begin
  1.j(0) = 0, j(k(m)) = n
     { Wir haben ein Feld j der Dimension k(m) + 1 }
  2.for 1 ≤ i ≤ k(m) − 1 pardo
     2.1Berechne Rang(b_{i·log m} : A) mit
        der Methode des binären Suchens
        j(i) := Rang(b_{i·log m} : A)
  3.for 1 ≤ i ≤ k(m) − 1 pardo
     3.1B_i := (b_{(i·log m)+1}, ..., b_{(i+1)·log m})
     3.2A_i := (a_{j(i)+1}, ..., a_{j(i+1)})   { A_i ist leer, wenn j(i) = j(i + 1) }
end
```

□

Beispiel 2.24

$$\text{A} = (4{,}6{,}7{,}10{,}12{,}15{,}18{,}20) \quad \textit{und} \quad \text{B} = (3{,}9{,}16{,}21)$$

Sei $m = |B| = 4$, dann $\log m = 2$, $\frac{m}{\log m} = 2$.

1. Einteilen von B in $\frac{m}{\log m}$ Blöcke der Länge $\log m$:
   $B_1 = (3, 9)$, $B_2 = (16, 21)$.

2. Einteilen von A: Rang$(9 : A) = 3$. also
   $A_1 = (4, 6, 7)$, $A_2 = (10, 12, 15, 18, 20)$.
   (Beachte, daß $A_0, B_0 < A_1$ und $A_0, B_0 < A_1$)

Die folgenden Schritte werden erst vom vollständigen Programm ausgeführt.

3. Mischen von A_0 und B_0 gibt: (3,3,6,7,9)
   Mischen von A_1 und B_1 gibt: (10,12,15,16,18,20,21)

4. Zusammenhängen der Listen ergibt Mischung von A und B.

$\square$

Die Wirkung von Algorithmus 2.23 noch einmal graphisch:

$$\text{B} = \underbrace{\boxed{b_1 \cdots b_{1\cdot\log m}}}_{B_1}\underbrace{\boxed{b_{1\cdot(\log m)+1} \cdots b_{2\cdot\log m}}}_{B_2}\boxed{\cdots}\underbrace{\boxed{b_{(i\cdot\log m)+1} \cdots b_{(i+1)\cdot\log m}}}_{B_i}$$

$$\text{A} = \underbrace{\boxed{a_1 \cdots a_{j(1)}}}_{A_1}\underbrace{\boxed{a_{j(1)+1} \cdots a_{j(2)}}}_{A_2}\boxed{\cdots}\underbrace{\boxed{a_{j(i)+1} \cdots a_{j(i+1)}}}_{A_i}$$

Dabei ist $a_{j(i)} < b_{i\cdot\log m}$ für $1 \leq i \leq k(m) - 1$ und $a_{j(i)+1} > b_{i\cdot\log m}$ für $1 \leq i \leq k(m) - 1$. Man beachte, daß jedes B_i $\log m$ viele Elemente hat, die Anzahl in den A_i dagegen schwankt.

Satz 2.25 Algorithmus 2.23 hat folgende Eigenschaften:

(a) Entstehe C_i durch Mischen von A_i und B_i dann gilt: $C = (C_1, ..., C_{k(m)})$ ist die Folge, die man durch Mischen von A und B bekommt.

(b) $T(n) = O(\log n)$.

(c) $W(n) = O(n + m)$.

Beweis

(a) Es gilt $C_1 < C_2 < \cdots < C_{k(m)}$ und alle Elemente aus A und B kommen in C vor, das impliziert die Behauptung.

(b),(c) 1.Schritt: $O(1)$ Arbeit und Zeit.

2.Schritt: $O(\log n)$ Zeit und
$O(\log n) \cdot \frac{m}{\log m} = O(m+n)$ Arbeit.

Man beachte, daß gilt

$$\log n \cdot \frac{m}{\log m} \leq m + n,$$

denn falls $m \leq n$ ist

$$\frac{\log n}{\log m} \cdot m \leq m$$

und falls $m < n$ ist

$$\log n \cdot \frac{m}{\log m} < \log n \frac{n}{\log n} = n,$$

da

$$\frac{m}{\log m} < \frac{m+1}{\log(m+1)},$$

denn es gilt:

$$\begin{array}{lll} & \frac{m}{\log m} & < \frac{(m+1)}{\log(m+1)} \\ \Leftrightarrow & m \cdot (\log(m+1)) & < (m+1)(\log m) \\ \Leftrightarrow & (m+1) \cdot 2^m & < 2^{m+1} \cdot m \\ \Leftrightarrow & m+1 & < 2 \cdot m \end{array}$$

3.Schritt: $O(1)$ Zeit und $O(n+m)$ Arbeit.

□

Schließlich der eigentliche Mischalgorithmus.

Algorithmus 2.26

Eingabe: 2 Folgen A und B, so daß alle Elemente paarweise verschieden sind daß beide Folgen genau n Elemente haben.

Ausgabe: Die Mischung C von A und B.

begin
1.Anwendung von Algorithmus 2.23 auf A und B
{ Wir bekommen Paare von Blocks (A_i, B_i) für $1 \leq i \leq \log m$,
wobei jedes B_i genau $\log n$ viele Elemente enthält. }
2.Für die Paare von Blocks $A_i, B_i)$ bei denen $|A_i| > \log n$ ist, wenden wir Algorithmus 2.23 an, wobei jetzt A_i die Rolle von B_i spielt.
{ Jetzt haben wir auf jeden Fall eine Menge von Paaren von Blocks (A_i, B_i)
wo sowohl $|A_i| \leq \log n$ als auch $|B_i| \leq \log n$ für alle i }
3.for $1 \leq i \leq l$ pardo
{ l sei die Anzahl der Paare von Blocks }
Mische A_i und B_i sequentiell zu C_i;
$C := (C_1, C_2, ..., C_l)$
end

□

Satz 2.27 Für Algorithmus 2.26 gilt:

(a) Er mischt korrekt.

(b) $T(n) = O(\log n)$.

(c) $W(n) = O(n)$.

Beweis
(b),(c) Schritt 1 braucht parallele Zeit von $O(\log n)$ und Arbeit von $O(n)$ nach Satz 2.25. Schritt 2 braucht Zeit $O(\log(\log n))$ (beachte $|B_i| = \log n$ für alle i) und Arbeit von $O(N)$. Schritt 3 braucht Zeit $O(\log n)$ und Arbeit $O(n)$. Die Behauptungen folgen. □

Da die Elemente $b_{i \cdot \log n}$ parallel mit binärer Suche in A eingeordnet werden, brauchen wir bei der Implemntierung von Algorithmus 2.26 auf der PRAM die Möglichkeit des concurrent–read. Eine CREW–PRAM erlaubt die Implementierung des Algorithmus.

2.5 Pipelining

Unter pipelining versteht man folgendes: Eine Aufgabe wird in eine Folge von Teilaufgaben $t_1, ..., t_m$ aufgeteilt, daß folgendes möglich ist: Zur Erledigung zweier solcher Aufgaben A_1 und A_2 erledigt man zuerst $t_{1_{A_1}}$ und dann $t_{1_{A_1}}$ und $t_{2_{A_2}}$ parallel dann $t_{2_{A_1}}$ und $t_{3_{A_1}}$ parallel und so fort. Wir werden diese Vorgehensweise für folgendes Beispiel besprechen: Die Aufgabe, die in Teilaufgaben zerlegt wird, ist das EIgfügen eines Elements in einen 2–3 Baum. Wir wollen eine Folge von Elementen einfügen.

Definition 2.28

- (a) Ein 2–3 Baum ist ein geordneter Wurzelbaum mit folgenden Eiganschaften:

  - Jeder innere Knoten hat entweder 2 oder 3 Söhne.
  - Alle Blätter sind in gleicher Höhe.

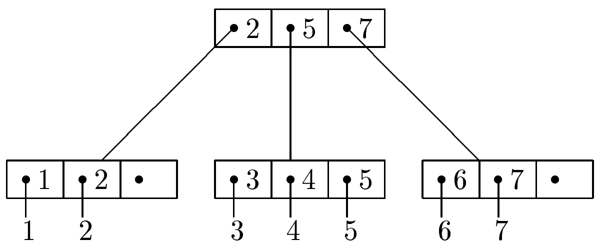
  Der Baum ist geordnet heißt: Die Söhne eines Knotens sind geordnet. Wir können vom 1. Sohn, dem am weitesten links stehenden, vom 2. Sohn usf. sprechen. Das geht in ungeordneten Bäumen nicht.

- (b) Eine sortierte Liste von Elementen einer totalen Ordnung $(a_1, a_2, ..., a_n)$ kann durch einen 2–3 Baum dargestellt werden, wobei die n Elemente der Liste in den Blättern des Baums stehen.

□

Wir geben die Darstellung durch einige Beispiele an:

Die Folge (1,2,3,4,5,6,7) wird zum Beispiel dargestellt durch:



In 2–3 Bäumen gelten immer folgende Eigenschaften:

Kleinstes Element eines inneren Knotens v (Bezeichnung $L(v)$) = *Größtes* Datenelement im ersten (am weitesten links stehenden) Teilbaum dieses Knotens.

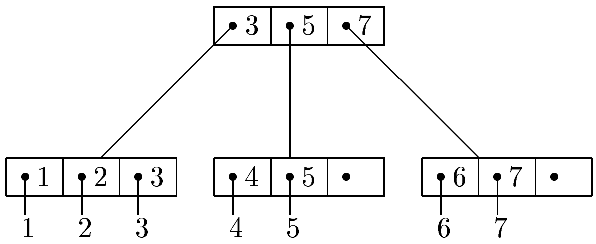
Zweitkleinstes Element eines inneren Knotens v (Bezeichnung $M(v)$) = *Größtes* Datenelement im zweiten (von links) Teilbaum des Knotens.

Und falls vorhanden:

> *Drittkleinstes* Element eines inneren Knotens v (Bezeichnung $R(v)$)
> $=$ *Größtes* Datenelement im dritten (von links) Teilbaum des Knotens.

Man beachte, daß die inneren Knoten immer genau 2 oder 3 Elemente enthalten, die Blätter immer nur genau 1 Element enthalten.

Eine andere Möglichkeit die Folge (1,2,3,4,5,6,7) darzustellen, ist:



Man sieht also, daß die Darstellung einer gegebenen Folge als 2–3–Baum nicht eindeutig ist.

Bemerkung 2.29 Für einen 2–3–Baum B der Höhe h gilt:

$$2^h \leq \#\text{ Blätter von } B \leq 3^h.$$

In dem obigen Beispielbaum ist $h = 2$ und $\#$ Blätter $= 7$. Da gilt $3 = 2^{\log_2 3}$, gilt $3^h = 2^{\log_2 3 \cdot h}$. Weiter gilt:

$$2^h \leq \#\text{ Blätter von } B \leq 3^h$$

Also

$$h \leq \log_2(\#\text{ Blätter von } B) \leq (\log_2 3) \cdot h.$$

D.h.

$$h \geq \frac{\log_2(\#\text{ Blätter von } B)}{\log_2 3} \quad \text{und} \quad h \leq \log_2(\#\text{ Blätter von } B).$$

Also

$$h \in \left[\frac{\log_2(\#\text{ Blätter von } B)}{\log_2 3},\ \log_2(\#\text{ Blätter von } B)\right],$$

also

$$h = \Theta(\log_2(\#\text{ Blätter von } B)).$$

$\square$

Der 2–3–Baum ist eine Datenstruktur zur Darstellung einer Menge von Elementen. Unterstützt werden die folgenden Operationen:

Suchen, Einfügen und Löschen,

die alle in Zeit $O(\log n)$, wobei n die Anzahl der gespeicherten Datenelemente ist, möglich ist. Eine Darstellung von Mengen, die diese Operationen effizient unterstützt, nennt man: *Wörterbuch (dictionary).*

Der Vollständigkeit halber skizzieren wir die Algorithmen für das Suchen, Einfügen und Löschen eines Elements in einem 2–3–Baum.

Algorithmus 2.30 Eingabe: Ein 2–3–Baum B, der die geordnete Folge $(a_1, a_2, ..., a_n)$ darstellt. Die a_i seien paarweise verschieden. Weiterhin ein Element b aus zu Grunde liegenden Menge.

Ausgabe: Das Blatt (einen Zeiger auf das Blatt), das das Element a_i enthält, so daß gilt:

$$a_i \leq b \text{ und falls } i < n \text{ ist } b < a_{i+1}.$$

Der Algorithmus verfolgt genau einen Pfad von der Wurzel zu dem gesuchten Blatt: Am Knoten v wird folgende Fallunterscheidung gemacht:

if $b \leq L(v)$ then "gehe in den ersten Baum" fi

if $L(v) \leq b \leq M(v)$ then "gehe in den zweiten Baum" fi

if $M(v) < b$ then "gehe in den dritten Baum" fi

Hat der Knoten nur 2 Söhne, gehen wir analog vor.

Der Algorithmus braucht Zeit von $O(\log n)$, da wir mit dem uniformen Kostenmaß arbeiten. $\square$

Algorithmus 2.31 Eingabe: Genau wie in Algorithmus 2.30

Ausgabe: Ein 2–3–Baum, der die Liste $(a_1, a_2, ..., a_i, b, a_{i+1}, ..., a_n)$ darstellt, wobei $a_i < ba_{i+1}$ falls $b \notin (a_1, ..., a_n)$. Falls $b \in (a_1, ..., a_n)$ soll an dem Eingabebaum nichts geändert werden.

Zunächst wird Algorithmus 2.30 aufgerufen, um das Blatt n mit dem Element a_i zu finden, für das gilt:

$$a_i \leq b \text{ und } b < a_{i+1} \text{ falls } i < n.$$

Ist $a_i = b$, so sind wir fertig.
{ Ein 2–3–Baum enthält immer nur paarweise verschiedene Elemente. }

Ist $a_i < b$, so generieren wir ein neues Blatt v, das b enthält. Wir fügen v rechts von n ein. Falls der Vorgänger von n jetzt 4 Kinder hat, spalten wir ihn auf in 2 Knoten, die jeder 2 Kinder haben. Falls man nun der Vorgänger 4 Kinder hat, spalten wir ihn auf. Dieses Aufspalten kann bis zur Wurzel nötig werden. Zusätzlich zu dem Aufspalten müssen auch die Werte $L(v)$, $M(v)$, $R(v)$ bei dem Vorgänger des neu eingefügten Blattes angepaßt werden. □

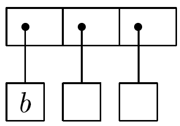
Algorithmus 2.32 Eingabe: Genau wie in Algorithmus 2.30

Ausgabe: Ein 2–3–Baum, der die Liste $(a_1, ..., a_{i-1}, a_{i+1}, ..., a_n)$ darstellt, falls $b = a_i$. Falls b nicht in $(a_1, ..., a_n)$ vorkommt, bleibe der Baum unverändert.

Zunächst wenden wir Algorithmus 2.30 an, um zu sehen, ob das Element b überhaupt im Baum vorkommt. Falls nicht, sind wir fertig.

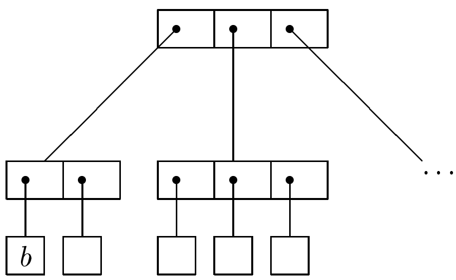
Kommt das Element b in einem Blatt vor, löschen wir das Blatt. Folgende Situationen können dabei auftreten

(1)



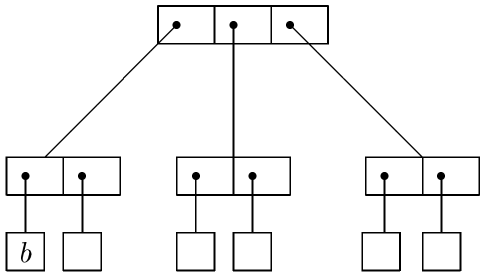
In diesem Fall sind wir nach Löschen von b fertig. Eventuell anpassen der Elemente in den Vorgängerknoten.

(2)



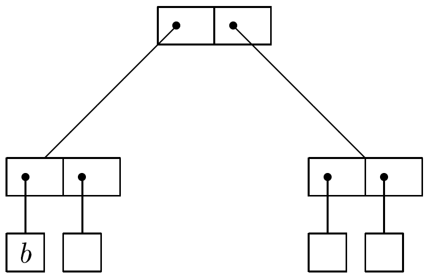
In diesem Fall löschen wir b und holen uns das kleinste Element von rechts. Die Elemente müssen angepaßt werden.

(3)



In diesem Fall fassen wir die 3 Vorgängerknoten der Blätter zu zweien zusammen. Die Elemente in den Vorgängerknoten müssen noch angepaßt werden.

(4)



Wir fassen die Vorgängerknoten der Blätter zusammen. Das entspricht dem Löschen eines Vorgängerknotens der Blätter. Wir machen dann rekursiv nach oben hin weiter.

□

3 Algorithmen für das parallele Einfügen. **Seiten 2–56 - 2–58 fehlen**

2.6 Accelerated cascading

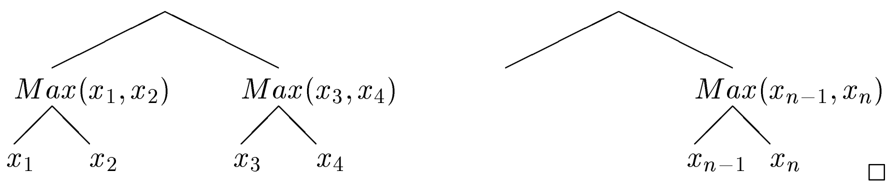
Das Problem, das Maximum einer Menge $X = \{x_1, ..., x_n\}$ zu finden, deren Elemente aus einer total geordneten Menge S kommen, ist durch einen einfachen sequentiellen Algorithmus in Linearzeit zu lösen. Beachte, daß X

nicht geordnet ist. Wir nehmen an, daß X in einem Feld dargestellt ist und daß die x_i paarweise verschieden sind. (Andernfalls ersetzen wir x_i durch das Paar (x_i, i) und sagen: $(x_i, i) > (x_j, j) \Leftrightarrow x_i > x_j$ oder $x_i = x_j$ und $i > j$, d.h. wir haben die lexikographische Ordnung auf den Paaren.)

Algorithmus 2.33 **Eingabe:** X wie gesagt.

Ausgabe: Das Maximum von X.

Wir arbeiten nach der Methode des binären Baums.



□

Algorithmus 2.33 braucht $O(n)$ Arbeit und $O(\log n)$ parallele Zeit. Er ist also *optimal*, da jeder sequentielle Algorithmus jedes Element von X einmal untersuchen muß.

Wir werden einen optimalen Algorithmus mit dem Zeitverhalten $O(\log(\log n))$ konstruieren.

Dazu zunächst ein Algorithmus, der das Maximum in konstanter paralleler Zeit ermittelt, aber nicht optimal ist.

Algorithmus 2.34 **Eingabe:** Ein Feld A, das p verschiedene Elemente enthält.

Ausgabe: Das boolesche Feld M, so daß $M(i) = 1$ gdw. $A(i)$ ist das Maximum der Elemente in A.

```
begin
  1.for 1 ≤ i,j ≤ p pardo
      if A(i) ≥ A(j) then B(i,j) := 1
                      else B(i,j) := 0
  2.for 1 ≤ i ≤ p pardo
      M(i) := B(i,1) ∧ B(i,2) ∧ ··· ∧ B(i,p)
end
```

Der Algorithmus kann auf der common CRCW–PRAM in $O(1)$ Zeit und $O(p^2)$ Arbeit implementiert werden. Er ist also nicht optimal. □

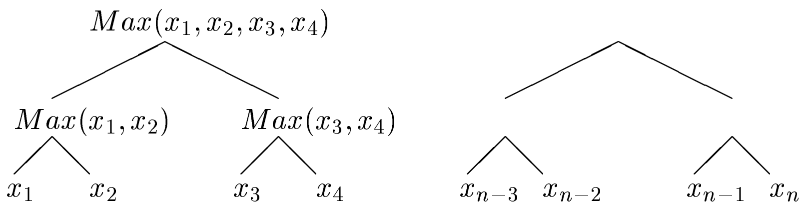
Definition 2.35

(a) Die Tiefe eines Knotens v in einem Wurzelbaum ist die Anzahl der Kanten zwischen der Wurzel und v. Damit ist die Wurzel in Tiefe 0.

(b) Sei $n = 2^{(2^k)}$ (also $\log n = 2^k$ und $\log(\log n) = k$). Der Baum doppelt–logarithmischer Tiefe mit n Blättern ist wie folgt definiert: Die Wurzel hat $2^{(2^{k-1})} = \sqrt{2^{(2^k)}}$ Kinder. Jedes Kind der Wurzel hat $2^{(2^{k-2})}$ Kinder. Ein Knoten in Tiefe i hat $2^{(2^{k-i-1})}$ Kinder für $0 \leq i \leq k-1$. Ein Knoten in Tiefe k hat genau 2 Blätter als Kinder.

$\square$

Beispiel
Für $n = 16$ ist $n = 2^{(2^2)}$ also $k = 2$. Der doppelt–logarithmische Baum mit n Blättern hat die Form



Lemma 2.36 Die Anzahl der Knoten in Tiefe i des doppelt–logarithmischen Baums mit $n = 2^{(2^k)}$ Blättern ist

$$2^{(2^k - 2^{k-i})} \text{ für } 0 \leq i < b$$

und

$$2^{(2^{k-1})} = \frac{n}{2} \text{ für } i = k$$

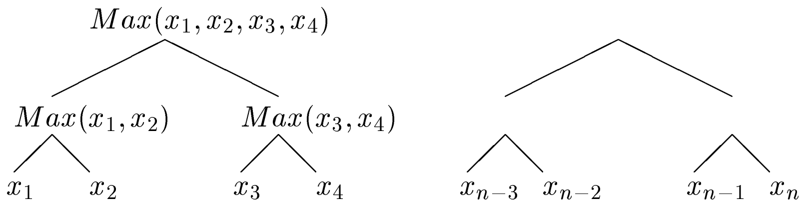
und

$$2^{2^k} = n \text{ für } i = k+1$$

Die Tiefe des Baums, d.h. die Tiefe der Blätter des Baums, ist also $\log(\log n) + 1$. $\square$

Mit dem doppelt–logarithmischen Baum bekommen wir folgenden Algorithmus für das Maximum.

Algorithmus 2.37 Statt mit dem binären Baum wie in Algorithmus 2.33 arbeiten wir mit dem doppelt–logarithmischen Baum: Sei $n = 2^{(2^k)}$.



□

Mit Algorithmus 2.34 können wir das Maximum der Söhne eines Knotens (beachte, daß diese Zahl im doppelt–logarithmischen Baum nicht endlich ist) in Zeit $O(1)$ berechnen. Damit kann das Maximum von X in Zeit $O(\log(\log n))$ paralleler Zeit ermittelt werden. Wieviele Instruktionen müssen wir dazu ausführen? Die Anzahl von Instruktionen zur Berechnung eines Knotens der Tiefe i ist $O((2^{(2^{k-i-1})})^2)$. Also haben wir pro Stufe

$$O((2^{(2^{k-i-1})})^2 \cdot 2^{(2^k - 2^{k-1})}) = O(2^{(2^k)}) = O(n)$$

Instruktionen und damit ist insgesamt $W(n) = O(n \cdot \log(\log n))$. Der Algorithmus ist zwar nicht optimal, aber sehr schnell.

Wir haben also 2 Algorithmen für das Problem: Einer ist optimal und läuft in logarithmischer Zeit, der andere ist nicht optimal, ist dafür aber schneller: doppelt–logarithmische Zeit. In einer solchen Situation ist die Strategie des *accelerated cascading* angebracht:

(1.) Beginne mit dem optimalen, aber langsamen Algorithmus, bis das Problem auf eine bestimmte Größe verkleinert ist.

(2.) Löse das Problem mit dem schnellen, aber nicht optimalen Algorithmus.

Algorithmus 2.38

Eingabe: Eine Menge $X = \{x_1, ..., x_n\}$, dargestellt durch ein Feld.

Ausgabe: Das Maximum von x_i.

1. Anwendung von Algorithmus 2.33, dem binären Baum, bis wir $\lceil \log(\log(\log n)) \rceil$ Stufen nach oben gegangen sind. Wir haben jetzt $n' = O\left(\frac{n}{\log(\log n)}\right)$ viele Elemente generiert, unter denen wir das Maximum von X finden.

Analyse von Schritt 1: Arbeit $O(n)$ und parallele Zeit $O(\log(\log(\log n)))$.

2. Anwendung von Algorithmus 2.37, dem doppelt–logarithmischen Baum, auf die n' Elemente, die sich aus dem 1. Schritt ergeben haben.

Analyse von Schritt 2: Parallele Zeit $O(\log(\log n')) = O(\log(\log n))$
und Arbeit $O(n' \cdot \log(\log n')) = O(n)$

Damit braucht der Algorithmus isgesamt Zeit $O(\log(\log n))$ und Arbeit $O(n)$.
□

Man beachte, daß Algorithmus 2.38 eine CRCW–PRAM zur Implementierung benötigt.

2.7 Zerstören von Symmetrie

Am Beispiel des Problems, einen gerichteten Kreis zu färben illustrieren wir die Problematik der Symmetrie für effiziente parallele Algorithmen und wie die Symmetrie zerstört werden kann.

Definition 2.39

(a) Sei $G = (V, E)$ ein gerichteter Graph. G ist ein gerichteter Kreis $\Leftrightarrow$ Für jeden Knoten $v \in V$ gibt es genau 2 Kanten (v, u) und (w, v) in E, wobei $u \neq v$ und $w \neq v$, und für je 2 Knoten $u, v \in V$ gibt es einen gerichteten Weg von u nach v.

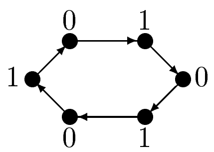
(b) Eine k–Färbung von G ist eine Abbildung

$$c \ : \ V \rightarrow \{0, 2, ..., k-1\},$$

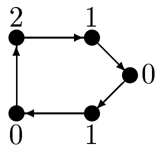
so daß gilt: $c(v) \neq c(w)$ falls $(v, w) \in E$. D.h. benachbarte Knoten haben verschiedene Farben.

□

Jeder Kreis kann mit 3 Farben gefärbt werden, wie z.B. in:



Oder in:



Man beachte, daß bei einer ungeraden Anzahl von Knoten 3 Farben erforderlich sind.

Es ist leicht, einen sequentiellen Algorithmus, der eine 3–Färbung eines Kreises ausgibt, zu finden: Die Knoten des Kreises werden einfach der Reihe nach durchgegangen und gefärbt. Diese Vorgehensweise sieht inhärent sequentiell aus: Wir wissen nicht wie wir einen Knoten färben müssen, solang wir nicht wissen, welche Farbe der Vorgänger hat usf. Der Grund ist, daß alle Knoten zunächst einmal gleich aussehen. Diese Tatsache macht die Symmetrie des Problems aus. Die folgenden Algorithmen zeigen uns, wie diese Symmetrie gebrochen werden kann. Ähnliche symmetrische Situationen treten auch noch bei vielen anderen Problemen auf.

Konvention 2.40 Wir nehmen an, daß der gerichtete Kreis $G = (V, E)$ bei der Eingabe in unsere Algorithmen folgendermaßen dargestellt ist: Die Menge von Knoten ist dargestellt durch $V = \{1, 2, ..., n\}$, die Kanten sind dargestellt durch ein Feld S (für *successor*) der Dimension n, so daß $S(i) = j$ gdw. $(i, j) \in V$.

Man beachte, daß wir aus S in einem parallelen Schritt das Feld P der Vorgänger finden können und auch die Menge E der Kanten. □

Bezeichnung 2.41 Sei i eine natürliche Zahl mit der Binärdarstellung $i = i_{t-1}\ i_{t-2}\ \cdots\ i_1\ i_0$, wobei eben $i_j \in \{o, 1\}$. Dann nennen wir i_k die k–te Dualstelle vo hinten. □

Der folgende Algorithmus zeigt, wie wir die Anzahl von Farben einer gegebenen Färbung reduzieren können. Wir können immer eine Färbung c mit vielen Farben finden, indem wir setzen: $c(i) = i$.

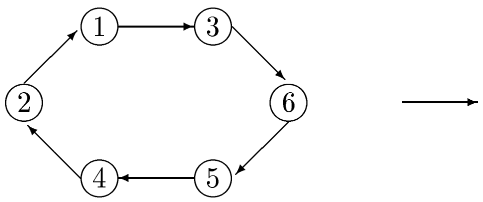
Algorithmus 2.42 (Einfache Färbung) Eingabe: Ein gerichteter Kreis, dessen Kanten durch ein Feld S der Dimension n gegeben sind und eine Färbung c der Knoten.

Ausgabe: Eine andere Färbung c' der Knoten des Kreises.

begin
  for $1 \leq i \leq n$ pardo
  1. k := Min { l | c(i) und c(S(i)) unterscheiden sich auf
     der l'–ten Stelle von hinten }
     { Hier ist k als lokale Variable zu sehen. }
  2. $c' := 2 \cdot k + (c(i))_k$
     { Mit $(c(i))_k$ meinen wir das k–te Bit von hinten
     von $c(i)$. }
end

□

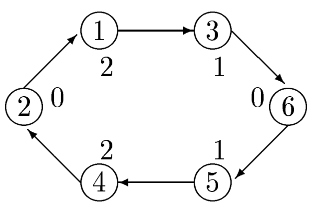
Beispiel 2.43 Wir betrachten einmal den folgenden Kreis:



Die Zahlen an den Knoten sind die Namen der Knoten. Wir nehmen an, die Färbung c ist gegeben durch $c(i) = i$ für alle Knoten $i \in \{1, 2, ..., 6\}$. Die folgende Tabelle zeigt, wie der Algorithmus arbeitet: Die Farben liegen zwischen 001 und 110.

Knoten	Alte Farbe	Farbe des Nachfolgers	k	Neue Farbe
1	001	011	1	$2 \cdot 1 = 010$
2	010	001	0	000
3	011	110	0	001
4	100	010	1	010
5	101	100	0	001
6	110	101	0	000

Unsere Färbung ist damit:



Man sieht, daß benachbarte Knozten verschiedene Farben haben. $\square$

Lemma 2.44 Der Algorithmus der einfachen Färbung hat folgende Eigenschaften:

(a) Die Ausgabefunktion $c^{'}$ ist eine Färbung der Knoten im Sinne von Definition 2.39 (b).

(b) $T(n) = O(1)$ und $W(n) = O(n)$.

Damit ist der Algorithmus *optimal.*

Beweis:

(a) Da c eine zulässige Färbung ist, existiert k immer. Nehmen wir an, daß $c^{'}$ keine zulässige Färbung ist. Sei also $(i, j) \in E$ mit $c^{'}(i) = c^{'}(j)$. Es ist ja

$$c^{'}(i) = 2 \cdot k + (c(i))_k \text{ und } c^{'}(j) = 2 \cdot l + (c(j))_l,$$

wobei k und l sich wie in dem Algorithmus ergeben. Da die Binärdarstellung von $2 \cdot k$ und $s \cdot l$ am Ende eine Null hat, ist $k = l$ und damit auch $(c(i))_k = (c(j))_l$. Das steht aber im Widerspruch zur Definition von k (beachte $j = S(i)$). Also muß $c^{'}(i) \neq c^{'}(j)$ sein für $(i, j) \in E$.

(b) Wir nehmen an, daß k mit einer konstanten Anzahl von Operationen bestimmt werden kann, dann folgen die Schranken für $T(n)$ und $W(n)$. Diese Annahme hängt von den Basisoperationen, die unserer PRAM zur Verfügung stehen ab, ist aber realistisch, solange unsere Farben $\leq n$ sind, also mit $O(\log n)$ vielen Bits dargestellt werden können.

$\square$

Man beachte, daß der Algorithmus der einfachen Färbung nur die Anzahl der Farben einer gegebenen Färbung reduziert, aber nicht eine 3–Färbung produziert. Durch wiederholtes Ausführen der einfachen Färbung bekommen wir einen 3–Färbungsalgorithmus, der in "fast konstanter" Zeit arbeitet, aber nicht optimal ist.

Definition 2.45 Sei $x > 0$. Dann ist die Funktion $\log^*$ definiert durch

$$\log^*(x) = Min\{i | \log^{(i)}(x) \leq 1\}$$

mit $\log^{(0)}(x) = x$ und $\log^{(i+1)}(x) = \log(\log^{(i)}(x))$. $\square$

Für $0 < x \leq 1$ ist $\log^*(x) = 0$. Für $1 < x \leq 2$ ist $\log^*(x) = 1$. Für $2 < x \leq 4$ ist $\log^*(x) = 2$, für $4 < x \leq 16$ ist $\log^*(x) = 3$, denn $\log 16 = 4$, $\log 4 = 2$, $\log 2 = 1$. Für x mit $\log x = 16$ ist $\log^*(x) = 4$. Die Funktion $\log^*(x)$ ist also eine äußerst langsam wachsende Funktion. Zum Beispiel ist $\log^*(x) \leq 5$ für $x \leq 2^{65536}$.

Sei nun $t > 3$ die Anzahl Bits, die wir benötigen, um die Farben der anfänglichen Färbung darzustellen. Die neuen Farben sind jetzt Zahlen zwischen 0 und $2 \cdot (t-1) + 1 = 2 \cdot t - 1$. Sie lassen sich also durch

$$\begin{array}{rl} & \lfloor \log(2 \cdot t - 1) \rfloor + 1 \\ = & \lceil \log((2 \cdot t - 1) + 1 \rceil \\ = & \lceil \log 2 \cdot t - 2 + 1 \rceil \\ = & \lceil \log(2 \cdot t - 1) \rceil \\ = & \lceil \log 2 \cdot t \rceil \\ = & \lceil \log t + 1 \rceil \\ = & \lceil \log t \rceil + 1 \end{array}$$

viele Bits darstellen. Das heißt, die neue Färbung gebraucht höchstens

$$2^{\lceil \log t \rceil + 1} \leq 2^{\log t + 2} = O(t)$$

viele Farben.

Die Anwendung der einfachen Färbung reduziert die Anzahl der Farben, solange wie für die Anzahl Bits, die benötigt werden, um die Farben der Färbung c darzustellen, gilt: $t > \lceil \log t \rceil + 1$. Nun gilt

$$\begin{array}{rcl} t & > & \lceil \log t \rceil + 1 \\ & \Leftrightarrow & \\ t & < & 3 \end{array}$$

Beachte, für $t = 3$ ist $\lceil \log t \rceil = 2$.

Die Anwendung der einfachen Färbung im Falle $t = 3$ gibt uns die Färbung mit ≤ 16 Farben, da die kleinste auftretende Farbe 0 und die größte $2 \cdot 2 + 1 = 5$ ist.

3 Listen und Bäume

Listen und Bäume sind grundlegende Datenstrukturen, die in vielen Algorithmen vorkommen. Wir behandeln folgendes grundlegende Problem für Listen: list ranking und das Problem des Eulerschen Kreis für Bäume. Die Lösungen dieser Probleme hinterher bei der Auswertung arithmetischer Ausdrücke und beim Problem der Bestimmung des kleinsten gemeinsamen Vorgängers in einem Baum gebraucht.

3.1 List ranking

Das list ranking Problem besteht darin, die Position eines Knotens in einer Liste zu bestimmen.

Definition 3.1 (Problem list ranking)

(a) Eine lineare Liste ist ein gerichteter Graph $G = (V, E)$ so daß gilt: Es gibt genau 2 Knoten $a, b \in V$ mit:

- Für alle $v \in V \setminus \{b\}$ gibt es genau ein $u \in V$ mit $(v, u) \in E$.
- Für alle $v \in V \setminus \{a\}$ gibt es genau ein $w \in V$ mit $(w, v) \in E$.
- Für je zwei Knoten $u, v \in V$ gilt: Es gibt einen Weg von u nach v oder von v nach u.

Wir nehmen an: $V = \{1, ..., n\}$ und stellen die Liste $G = (V, E)$ dann durch ein n–dimensionales Feld S *successor* dar mit $S(i) = j \Leftrightarrow (i, j) \in E$ und $S(b) = 0$.

(b) Das Problem des list ranking ist gegeben durch:

Eingabe: Die Distanz (= # von Knoten) von jedem Knoten i bis zum Ende der Liste, angegeben durch ein Feld R mit $R(i)$ = Distanz von i bis zum Ende.

□

Die sequentielle Komplexität des Problems ist linear. Wir ermitteln aus S das Vorgängerfeld P (*predecessor*) und gehen dort einmal durch. Dabei ist $P(i) = j \Leftrightarrow S(j) = i \Leftrightarrow (j, i) \in E$ und $P(a) = 0$. Unser Algorithmus ist analog zu Algorithmus 2.10 und Algorithmus 2.13, die mit der Technik des pointer jumping arbeiten.

Algorithmus 3.2

Eingabe: Wie in Definition 3.1(b)

Ausgabe: Wie in Definition 3.1(b)

```
begin
  1.for 1 ≤ i ≤ n pardo
      if (S(i) ≠ 0) then R(i) := 1
                    else R(i) := 0
  2.for 1 ≤ i ≤ n pardo
      Q(i) := S(i)
      while Q(i) ≠ 0 and Q(Q(i)) ≠ 0 do
                    R(i) := R(i) + R(Q(i))
                    Q(i) := Q(R(i))
end
```

□

Der Algorithmus bestimmt den Rang $R(i)$ jedes Knotens in Zeit $O(\log n)$ mit $O(n \cdot \log n)$ Instruktionen, siehe Satz 2.11. Im Gegensatz zum pointer jumping in Bäumen, kommen wir hier ohne concurrent read aus und können den Algorithmus auf einer EREW–PRAM implemenmtieren.

Der Algorithmus ist nicht optimal, da die Arbeit $O(n \cdot \log n)$ ist, wir aber einen sequentiellen Algorithmus mit Zeit von $O(n)$ haben. Um einen optimalen und schnellen Algorithmus zu bekommen, wenden wir die folgende Strategie an:

1. "Zusammenschrumpfen" der Liste bis wir nur noch $O\left(\frac{n}{\log n}\right)$ viele Knoten haben.

2. Algorithmus 3.2 auf der geschrumpften Liste (das gibt Zeit $O(\log n)$ und Arbeit $O(n)$).

3. Rekonstruktion der ursprünglichen Liste und Ermittlung der noch fehlenden Ränge.

Die Frage ist: Wie sollen wir die Liste schrumpfen? Dazu zunächst folgende Bezeichnung:

Bezeichnung 3.3 Eine Menge von Knoten I einer Liste ist unabhängig $\Leftrightarrow$ Für $i \in I$ gilt $S(i) \notin I$. □

Der folgende Algorithmus entfernt eine Menge unabhängiger Knoten I aus einer Liste. Er benutzt das Vorgängerfeld P gegeben durch: $P(i) = j$ gdw. Knoten j ist der Vorgänger von i und $P(i) = 0$ falls i der Anfangsknoten ist. Aus dem Feld S der Nachfolger kann P in Zeit $O(1)$ ermittelt werden.

Algorithmus 3.4

Eingabe: (1) Felder S und P der Dimension n, die die Nachfolger- und Vorgängerbeziehungen einer Liste darstellen. (2) Eine unabhängige Menge von Knoten. (3) Ein Wert $R(i)$ für jeden Knoten i.

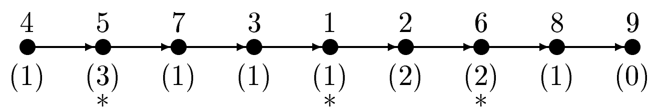
Ausgabe: Die Liste, die durch Herausnehmen der Knoten von I entsteht.

```
begin
  1. Bestimme ein Feld N(i) i ∈ I, das den Elementen
     von I aufeinanderfolgende Zahlen zuweist.
  2. for alle i ∈ I pardo
       U(N(i)) := (i,S(i),R(i))
         { Wir merken uns die Information über
         die Knoten i ∈ I im Feld U }
       if P(i) ≠ 0 then
         R(P(i)) := R(P(i) + R(i))
         { Die Bedingung P(i) ≠ 0 stellt sicher, daß
         i nicht am Anfang der Liste ist. }
       if P(i) ≠ 0 then
         S(P(i)) := S(i)
       if S(i) ≠ 0 then
         P(S(i)) := P(i)
end
```

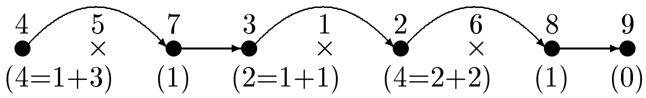
□

Zunächst ein Beispiel zu Algorithmus 3.4.

Beispiel 3.5 Die Eingabeliste mit dem Eingaberang $R(i)$ in Klammern unter den Knoten. Die Knoten in I sind durch einen Stern $(*)$ gekennzeichnet.



Die Liste nach dem Schrumpfen:



□

Das nächste Lemma faßt die Eigenschaften von Algorithmus 3.4 einmal zusammen.

Lemma 3.6 Gegeben ist eine unabhängige Menge I in einer Liste L mit R–Werten, wobei I weder Anfangs- noch Endpunkt der Liste enthält.

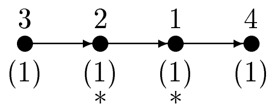
(a) Algorithmus 3.4 entfernt I und paßt die Werte $R(i)$, $S(i)$ und $P(i)$ an.

(b) $T(n) = O(\log n)$.

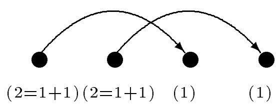
(c) $W(n) = O(n)$.

Beweis

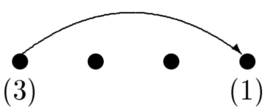
(a) Wichtig für die Korrektheit ist, daß I eine unabhängige Menge von Knoten ist. Bei dem folgenden Beispiel



macht unser Algorithmus folgendes



statt dem erwünschten



(b) Schritt 1 können wir in $O(\log n)$ Zeit realisieren, indem wir eine Präfix–Summen Berechnung auf L ausführen, wobei die Knoten aus I das Gewicht 1 und die $L \setminus I$ das Gewicht 0 haben. Schritt 2 kann in Zeit $O(1)$ ausgeführt werden.

(c) Die Arbeit der in (b) gegebenen Implementierung ist $O(n)$.

□

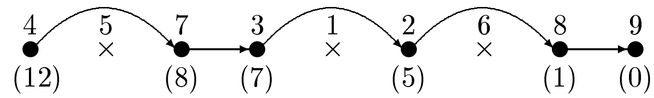
Beispiel 3.7 Schauen wir uns Beispiel 3.5 noch einmal etwas näher an: In Schritt 1 wird jedem Knoten aus I eine Nummer zugeteilt:

$$N(1) = 1,\ N(5) = 2,\ N(6) = 3.$$

Das Feld U hat dann folgende Struktur:

$$U(1) = (1, 2, 1),\ U(2) = (5, 7, 3),\ U(3) = (6, 8, 2).$$

Nehmen wir einmal an, wir haben die endgültigen Ränge für die verkürzte Liste ermittelt:



Dann bekommen wir die Ränge der Knoten aus I folgendermaßen

$$R(6) = 2 + R(8) = 3,\ R(1) = 1 + R(2) = 7,\ R(5) = 3 + R(7) = 11.$$

Außerdem können wir Knoten 6 wieder in die Liste aufnehmen, indem wir setzen

$$P(6) = P(8) = 2,\ S(P(6)) = S(2),\ P(8) = 6,$$

ebenso für die anderen Knoten. □

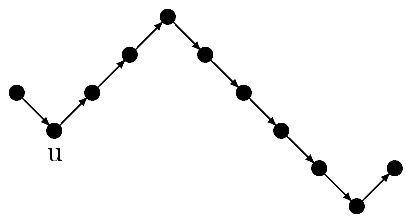
Bemerkung 3.8 Die Erstellung des Feldes U hat folgenden Sinn: Man braucht es, um die Liste hinterher wiederherstellen zu können. Die Nummerierung $N(i)$ wird benötigt, um hinterher wieder auf die Werte zugreifen zu können. □

Die nächste Frage ist: Wie finde ich eine unabhängige Menge? Mit einer k–Färbung! Dazu zunächst eine Bezeichnung.

Bezeichnung 3.9 Ein Knoten i einer gefärbten Liste L ist ein lokales Minimum (lokales Maximum) bzgl. der betrachteten Färbung gdw. die Farbe i kleiner (größer) als die Farben der beiden Nachbarn von i ist. □

Lemma 3.10 Gegeben ist eine k–Färbung der Knoten einer Liste, die aus n Knoten besteht. Die Menge der lokalen Minima (oder Maxima) ist eine unabhängige Menge der Größe $\Omega(\frac{n}{k})$. Wir können die Menge der lokalen Minima (Maxima) in Zeit $O(1)$ mit $O(n)$ Operationen finden.

Beweis
Seien u, v 2 Knoten der Liste, die lokale Minima sind, so daß kein weiteres Minimum zwischen u und v ist: Dann muß folgende Situation vorliegen:



Dann gilt: # Knoten zwischen u und $v \leq 2 \cdot k - 3$, da wir nur k Farben haben. Damit folgt, daß für alle m, k gilt: # lokaler Minima $\leq \frac{n}{2 \cdot k - 1} = \Omega\left(nk\right)$. Ebenso kann man für lokale Maxima argumentieren.

Man kann durch Vergleich mit den Nachbarn von jedem Knoten bestimmen, ob er ein lokales Minimum ist. □

Wir bekommen nun eine große unabhängige Menge, indem wir den optimalen Algorithmus zur 3–Färbung auf die Liste anwenden. Die entstehende unabhängige Menge ist dann sicherlich $\leq \frac{1}{5} \cdot n = \Omega(n)$. Durch Schrumpfung mit dieser unabhängigen Menge wird die Liste um einen konstanten Faktor ≤ 1 kleiner: Die ursprüngliche Liste hat die Länge n, die geschrumpfte eine Länge $\leq \left(1 - \frac{1}{5}\right) \cdot n$.

Algorithmus 3.11

Eingabe: Eine Liste mit n Knoten, so daß der Nachfolger des Knotens i durch $S(i)$ gegeben ist.

Ausgabe: Für jeden Knoten i die Entfernung $R(i)$ vom Ende der Liste.

begin
  1. $\mathsf{n}_0 := 0$; K := 0;
  2. while $\mathsf{n}_k \leq \frac{\mathsf{n}}{\log \mathsf{n}}$ do
    2.1 k := k + 1;
    2.2 Färbe die Liste mit 3 Farben und bestimme die Menge I der lokalen Minima.
    2.3 Entferne die Knoten aus I und speichere die entsprechende Information bzgl. der entfernten Knoten.
    { Die Entfernung der Knoten macht Algorithmus 3.4. }

2.4 Sei n_k die Größe der übriggebliebenen Liste. Kompaktifiziere die Liste in benachbarte Speicherplätze.

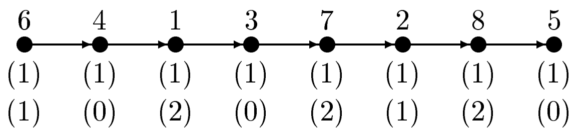
3. Wende die pointer jumping Technik auf die in 2. entstandene Liste an.

4. Kehre den Prozeß aus 2. um und rekonstruiere die ursprüngliche Liste schrittweise.

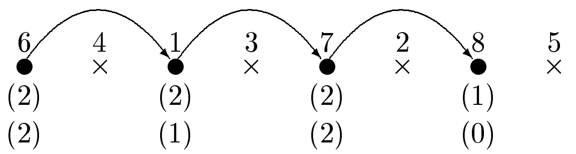
end

□

Beispiel 3.12 Gegeben ist eine Liste mit Rang und 3–Färbung in Klammern unter den Knoten:



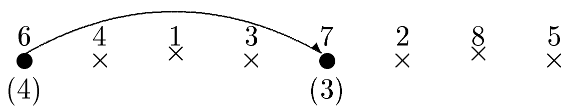
Die erste Iteration der Schleife nimmt $I = \{4, 3, 2, 5\}$ als unabhängige Menge. Wir bekommen jetzt die Liste (schon mit 3–Färbung versehen):



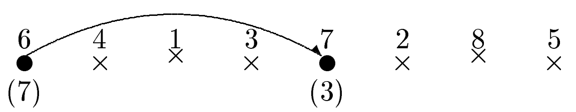
Die Information über die entfernten Knoten wird in dem Feld U aus Algorithmus 3.4 gespeichert, wobei U um eine Dimension, die die Iteration anzeigt, erweitert ist. Bezüglich Knoten 4 haben wir gespeichert

$$U(1, N(4)) = (4, S(4), R(4)) = (4, 1, 1).$$

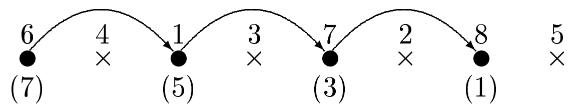
Die zweite Iteration gibt uns die Liste:

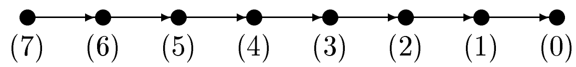


Hier nun ist $n_k = 2$. Damit gilt: $n_k \leq \frac{n}{\log n} = \frac{8}{3}$ und die while–Schleife ist beendet. Die pointer jumping Technik ermittelt die Ränge der kurzen Liste:



Die Rekonstruktion der ursprünglichen Liste erfolgt nun wieder in 2 Iterationen, die die Iterationen der while–Schleife umkehren:





Die ursprüngliche Liste hat ihren korrekten Rang. □

Satz 3.13 Für Algorithmus 3.11 gilt:

(a) Er hat das gewünschte Ein-, Ausgabeverhalten.

(b) $T(n) = O((\log n) \cdot (\log(\log n)))$.

(c) $W(n) = n$.

Beweis

(a) Die Korrektheit folgt aus der Korrektheit der verwendeten Algorithmen.

(b),(c) Zunächst beschränken wir die # Iterationen in Schritt 2. Da die aktuelle Liste jeweils um einen konstanten Faktor verkleinert wird, ist diese $O(\log(\log n))$. Nun schätzen wir Zeit und Arbeit für jeden Schleifendurchlauf: Die 3–Färbung braucht $O(\log n)$ Zeit und $O(n)$ Operationen.

Beweis:

(a) Die Korrektheit folgt aus der Korrektheit der verwendeten Algorithmen.

(b),(c) Zunächst beschränken wir die Anzahl der Iterationen in Schritt 2; Sei

$$n_k := \#\text{Elemente der Liste zu Beginn der } (k+1)\text{–ten Iteration.}$$

(Also $n_0 = n = \#$ Elemente der Eingabeliste.)
Es gilt immer: Die Größe von I in der $(k+1)$–ten Iteration ist $\geq \frac{n_k}{5}$.

Also folgt für alle k

$$n_{k+1} \leq \left(1 - \frac{1}{5}\right) \cdot n_k = \frac{4}{5} \cdot n_k.$$

Dann induktiv für alle k

$$n_k \leq \left(\frac{4}{5}\right) \cdot n.$$

Also $n_k \leq \frac{n}{\log n}$ in $O(\log(\log n))$ Iterationen. Jetzt ermitteln wir den Aufwand jeder Iteration:

– Optimale 3–Färbung: $O(\log n)$ Zeit, $O(n)$ Arbeit. Also braucht Schritt 2.2 diesen Aufwand.

– Schritt 2.3 braucht $O(\log n)$ Zeit, $O(n)$ Arbeit.

– Schritt 2.4 wird mit einem Präfix–Summen Algorithmus implementiert: $O(\log n)$ Zeit, $O(n)$ Arbeit.

Damit folgt: Die $(k+1)$–te Iteration des while–loops geht in $O(\log n)$ Zeit mit $O(n_k)$ Arbeit.

Insgesamt: Zeit in Schritt 2 $O((\log n) \cdot (\log(\log n)))$ Arbeit von Schritt 2: $O(n)$, da:

$$\sum_{k=0} n_k = \sum_k \left(\frac{4}{5}\right)^k \cdot n = n \cdot \sum_k \left(\frac{4}{5}\right)^k = O(n).$$

Schritt 3: $O(\log n)$ Zeit, $O(n)$ Arbeit.
Schritt 4: $O(\log(\log n))$ Zeit, $O(n)$ Arbeit.

□

Man beachte, daß Algorithmus 3.11 auf einer EREW–PRAM implementiert werden kann.

Bemerkung 3.14

(a) Algorithmus 3.11 kann den Rang vom Listenanfang oder Listenende ermitteln, je nachdem, ob wir mit der Vorgänger- oder Nachfolgerfunktion arbeiten.

(b) Das Problem des parallelen Präfix auf Listen ist wie das list ranking Problem, nur mit Gewichten und vom Listenanfang. Es gibt zwei Lösungsmöglichkeiten:

- $R(i)$ = Gewicht von i, dann list ranking von vorne.
- List ranking von vorne, Elemente geordnet nach ihrem Rang in einem Feld, Präfix–Summen mit den Gewichten.

Die Komplexität beider Verfahren ist wie die des list ranking. □

Bemerkung 3.15 Man kann das list ranking mit

$$W(n) = O(n) \quad \text{und} \quad T^{\infty}(n) = O(\log n)$$

lösen. Dazu ist es nötig, die anfängliche Kontraktion in Zeit $O(\log n)$ durchzuführen. □

3.2 Die Technik der Euler–Tour

Wir definieren zunächst den Begriff des Eulerschen Kreises eines Graphen. Man beachte den Unterschied des Eulerschen Kreises zum Hamiltonschen Kreis.

Definition 3.16 Sei $G = (E, V)$ ein zusammengehängender, gerichteter Graph.

(a) Ein Eulerscher Kreis von G ist Weg (dargestellt als Folge von Kanten) durch G, so daß

- Der Weg ist geschlossen, d.h. beginnt und endet am gleichen Knoten.
- Der Weg benutzt jede Kante von G *genau* einmal.

(b) Der Graph G ist Eulersch gdw. G einen Eulerschen Kreis hat.

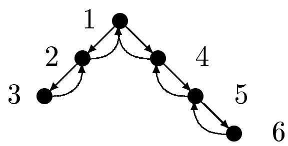
□

Beispiel 3.17

(a) Der folgende Graph

ist *nicht* Eulersch. Wir haben zwar einen Weg, der jede Kante genau einmal durchläuft, aber der Weg ist nicht geschlossen.

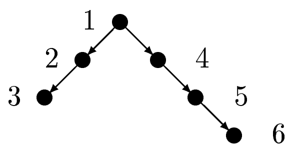
(b) Dagegen ist



Eulersch. Ein Eulerscher Kreis ist

$$(1,2)(2,3)(3,2)(2,1)(1,4)(4,5)(5,6)\ (6,5)(5,4)(4,1).$$

Man beachte, wie sich der obige Graph zum ungerichteten Baum



verhält.

□

Lemma 3.18 Ein zusammenhängender, gerichteter Graph ist Eulersch gdw. gilt, daß für jeden Knoten des Graphen die Anzahl der hereingehenden Kanten (= *indegree*) gleich der Anzahl der hinausgehenden Kanten (= *outdegree*) ist.

Beweis:
Der Beweis ist eine Aufgabe auf dem 5. Übungsblatt.
Ein ungerichteter Graph G ist Eulersch gdw. der Grad jedes Knotens von G gerade ist. □

Das Lemma 3.18 hat die interessante Konsequenz: Im Gegensatz zum Hamiltonschen Kreisen sind Eulersche Kreise schnell (d.h. zunächst einmal in Polynomialzeit) zu finden. Wir interessieren uns hier für Eulersche Kreise auf gerichteten Graphen, die durch Verdopplung der Kanten von Bäumen entstehen (vgl. Beispiel 3.17(b)). Der folgende Satz sagt uns wie wir, wenn wir für jeden Knoten eine Ordnung auf einem Nachbarn (im ursprünglichen Baum) gegeben haben, einen Eulerschen Kreis (im gerichteten Baum mit doppelten Kanten) angeben können.

Satz 3.19 Sei $G = (V, E)$ ein ungerichteter Baum (d.h. einfach ein zusammenhängender Graph ohne Kreise, wir haben keine Wurzel ausgezeichnet (!)). Für $v \in V$ sei

$$\begin{aligned} N(v) &= \text{Die Menge der Nachbarn} \\ & \text{des Knotens } v \text{ in } G. \end{aligned}$$

Wir nehmen an, daß wir für jedes v eine feste Numerierung der Knoten in $N(v)$ vorliegen haben, so daß

$$\begin{aligned} N(v) = \{ &\text{ Nachbar 0 von } v, \text{ Nachbar 1 von } v, \\ &\ldots, \text{ Nachbar } d-1 \text{ von } v\}, \end{aligned}$$

wobei d = Grad von v ist.

Sei nun $G^{'} = (V, E^{'})$ der gerichtete Graph der aus G durch Ersetzen der ungerichteten Kanten $\{u, v\}$ durch das Paar (u, v) und (v, u) entsteht.

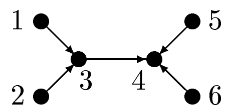
Dann gilt: Die Nachfolgerfunktion

$$S : E^{'} \to E^{'}$$

mit $S((\text{Nachbar } i \text{ von } v, v)) = (v, \text{ Nachbar}(i + 1 \bmod d) \text{ von } v)$ gibt uns einen Eulerschen Kreis auf $G^{'}$. □

Vor dem Beweis von Satz 3.19 ein Beispiel.

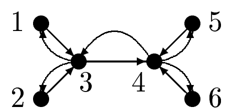
Beispiel 3.20 Zunächst der Baum G:



Wir wählen die folgende Ordnung auf den Nachbarn der Knoten:

Knoten	Ordnung auf den Nachbarn
1	3
2	3
3	2,1,4
4	5,6,3
5	4
6	4

Wir bekommen folgenden gerichteten Graph:



Und folgenden Eulerschen Kreis (beginnend mit der Kante (1,3)):

$$(1, 3)(3, 4)(4, 5)(5, 4)(4, 6)(6, 4)(4, 3)(3, 2)(2, 3)(3, 1)$$

□

Im Beispiel sieht alles vernünftig aus, trotzdem erfordert Satz 3.19 einen Beweis: Nach Definition der Nachfolgefunktion S hat zwar jede Kante genau einen Vorgänger und genau einen Nachfolger, d.h. S beschreibt einen Weg ja sogar Kreise, aber es ist nicht gesagt, daß S *einen* Kreis über *alle* Kanten beschreibt. Beachte, daß der Graph $G = (V, E^{'})$ nach Lemma 3.18 Eulersch ist.

Beweis von Satz 3.19
Induktion über die Anzahl der Knoten n von G.

Induktionsanfang, $n = 2$.
In diesem Falle gilt die Behauptung.

Induktionsschluß
Sei also $n > 2$. Sei $G = (V, E)$ ein Baum und sei v ein Blatt, d.h. ein Knoten vom Grad 1. (Einen solchen gibt es, da G azyklisch ist.) Sei $N(v) = \{u\}$, also ist

$$S((u, v)) = (v, u),$$

denn von einem Blatt geht unser Eulerscher Kreis direkt wieder zurück. Sei

$$\begin{array}{lcl} v & = & i\text{–ter Nachbar von } u, \\ v^{'} & = & \text{Vorgänger von } v \text{ (in der Ordnung der Nachbarn)} \\ v^{''} & = & \text{Nachfolger von } v \text{ (in der Ordnung der Nachbarn)} \end{array}$$

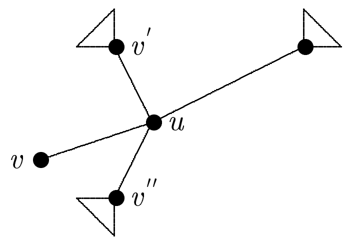
(Falls u vom Grad 2 ist, ist $v^{'} = v^{''}$.) Für unser S gilt nun:

$$S((v^{'}, u)) = (u, v), \quad S((v, u)) = (u, v^{''}).$$

Für unser S gilt nun:

$$S((v^{'}, u)) = (u, v) \quad , \quad S((v, u)) = (u, v^{''}).$$

Das heißt wir haben folgende Situation vorliegen:



Wir nehmen v aus V heraus und die zugehörigen Kanten aus $E^{'}$. Dann wird

$$v^{''} = i\text{–ter Nachbar von } u.$$

Nach Induktionsvoraussetzung beschreibt S mit

$$S((v^{'}, u)) = (u, v^{''})$$

einen Eulerschen Kreis auf dem Graphen ohne den Knoten v. Ersetzen wir $S((v^{'}, u)) = (u, v^{''})$ durch

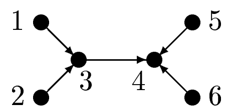
$$S((v^{'}, u)) = (u, v) \quad S(v, u) = (u, v^{''})$$

gibt uns dann wirklich einen Eulerschen Kreis auf $G^{'}$. $\square$

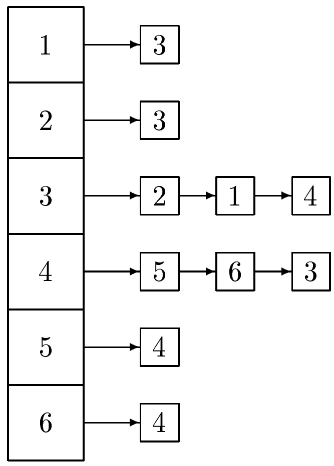
Bezeichnung 3.21 Sei $G = (V, E)$ ein Baum, wobei zu jedem $v \in V$ die Nachbarn von $v, N(v)$ geordnet sind. Sei $G^{'} = (V, E^{'})$ der zu G gehörende gerichtete Graph. Den durch die oben beschriebene Funktion S gegebene Eulersche Kreis in $G^{'}$ nennen wir Euler–Tour von G bzgl. der gegebenen Ordnung auf den Nachbarn. $\square$

Wie finden wir schnell die Euler–Tour zu einem Baum G mit Ordnung auf den Nachbarn? Das Ziel ist es, die Euler–Tour in Zeit $O(1)$ zu ermitteln. Um das sicherzustellen, muß der Baum in einer bestimmten Datenstruktur vorliegen. Dazu wieder ein Beispiel:

Beispiel 3.22 Wir nehmen an, daß der Baum als Feld von pointern auf die Listen der Nachbarn dargestellt ist. D.h. der Baum



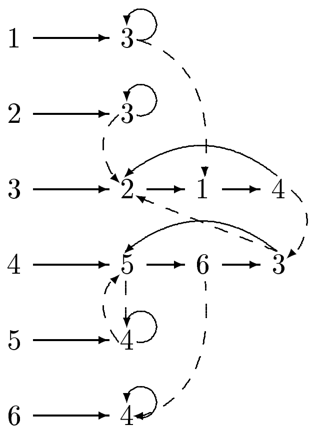
wird dargestellt durch:



Damit haben wir auch schon die Ordnung auf den Nachbarn aus Beispiel 3.20 dargestellt. Wir nehmen an, daß die Struktur in aufeinanderfolgenden Speicherplätzen gespeichert ist. Das hat zur Folge, daß wir parallel alle Einträge in Zeit $O(1)$ lesen können. In der Struktur ist jede Kante mit den doppelten, gerichteten Kanten genau einmal dargestellt:

- Knoten 3 in der 1. Zeile stellt die Kante $1 \to 3$ dar.
- Knoten 3 in der 4. Zeile die zugehörige Kante $3 \to 1$.

Man versuche einmal durch paralleles Lesen der dargestellten Struktur die Nachfolgerfunktion unserer Euler–Tour zu ermitteln, so sieht man, daß das nicht in Zeit $O(1)$ geht. Wir brauchen einige zusätzliche pointer, die in folgender Struktur enthalten sind:



Aus dieser Darstellung kann man leicht, d.h. parallel in $O(1)$ die Nachfolgefunktion S ermitteln: Wir lesen zum Beispiel den Knoten 4 in der 3. Zeile. Der gestrichelte pointer sagt uns, daß wir hier an dem Punkt sind, der die Kante (3,4) darstellt. Durch Verfolgen des Nachfolgepointers von Knoten 3 in der Adjazenzliste von 4 sehen wir, daß Knoten 5 der nächste ist. Wir bekommen also die Information für unsere Nachfolgerfunktion S, daß $S(3,4) = (4,5)$ ist. Da wir pro Eintrag in die Struktur nur konstant viel Zeit brauchen, geht das Ganze in paralleler Zeit $O(1)$ mit Arbeit $O(n)$, wobei n die Knotenanzahl des Baums ist. Sequentiell brauchen wir natürlich eine Zeit von $O(n)$. $\square$

Zur Sicherheit fassen wir unsere Erkenntnisse aus Beispiel 3.22 in folgendem Satz zusammen:

Satz 3.23 Sei ein Baum $T = (V, E)$ wie in Beispiel 3.22 dargestellt, gegeben. Wir können wir die Euler–Tour von T in Zeit $T^\infty(n) = O(1)$ und Arbeit $W(n) = O(n)$ konstruieren, wobei $n = |V|$ ist. Damit ist der Algorithmus optimal. $\square$

Der Algorithmus, der Satz 3.23 zur Folge hat, läuft auf einer EREW–PRAM.

Wenn wir es mit der Euler–Tour zu tun haben, nehmen wir ab jetzt immer an, daß der Baum in der Darstellung aus Beispiel 3.22 gegeben ist.

Aufgabe: Wir wollen die Euler–Tour als Liste, die durch die Nachfolgerfunktion S gegeben ist, sehen. Insbesondere wollen wir auf der Liste das Problem des parallelen Präfix lösen. Sie sollten sich einige Gedanken über die Darstellung von S machen, damit dieses Problem effizient parallel lösbar ist. Es soll Algorithmus 3.11 anwendbar sein. $\square$

Wir wenden die Euler–Tour jetzt an, um bestimmte Funktionen auf Bäumen zu ermitteln.

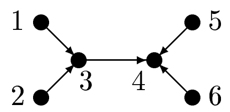
Definition 3.24 (Problem Wurzel geben) Das Problem einem Baum eine Wurzel zu geben ist definiert durch:

Eingabe: Ein Baum $G = (V, E)$ und ein Knoten (die beabsichtigte Wurzel) $r \in V$.

Ausgabe: Eine Funktion $p : V \to V$, so daß $p(v) =$ der Vater von v, wenn r die Wurzel ist.

$\square$

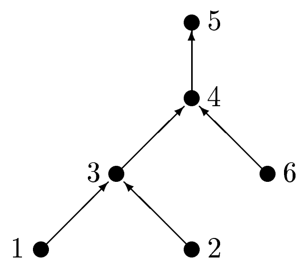
Beispiel 3.25 Wir betrachten wieder unseren bekannten Baum:



Wir wollen Knoten 5 zur Wurzel machen. Bezüglich der vorher gegebenen Ordnung der Nachbarn der Knoten bekommen wir folgende Euler–Tour, die mit Kante (5,4) beginnt (beachte, daß 4 = Nachbar 0 von Knoten 5 ist), und mit Kante (4,5) endet, d.h. wir setzen $S(4,5) = 0$:

$$(5,4)(4,6)(6,4)(4,3)(3,2)(2,3)(3,1)(1,3)(3,4)(4,5).$$

Der Baum mit Wurzel 5 hat die Form



Man sieht, daß die Euler–Tour eine Tiefensuche durch den Baum mit Wurzel r beschreibt.

Aufgabe: Man mache sich klar, daß in analogen Situationen immer gilt, daß die Euler–Tour eine Tiefensuche beschreibt. □

Woran erkennt man jetzt den Vater eines Knotens u? In der Euler–Tour wie oben kommen natürlich alle Kanten, die den Knoten u berühren vor. Aber: Nur für eine Kante gilt: Kante (v,u) steht vor (u,v) in der Euler–Tour. Für die übrigen Kanten (u,v'), die u berühren, gilt: (u,v') steht vor (v',u)!. Dann ist $v = p(u)$. Im Beispiel ist (4,3) vor (3,4), aber (3,1) vor (1,3) und (3,2) vor (2,3). Und es ist ja auch $p(3) = 4$. □

Algorithmus 3.26 (Wurzel geben)

Eingabe: (1) Ein Baum wie in Beispiel 3.22 dargestellt. (2) Eine Euler–Tour, die durch die Nachfolgerfunktion S gegeben ist. (3) Ein ausgezeichneter Knoten r.

Ausgabe: Für jeden Knoten $v \in V,\ v \neq r$ den Vorgänger $p(v)$, wenn r die Wurzel ist.

```
begin
  1.Finde den letzten Knoten u auf der Adjazenzliste von r
    Setze S((u,r)) = 0.
    {Mit der Kante (u,r) endet die Tiefensuche, die S beschreibt}
  2.Gewicht von 1 zu jeder Kante (x,y). Paralleles Präfix
    auf der Liste von Kanten, die durch S gegeben ist.
    {Man muß sich S geeignet dargestellt vorstellen. }
  3.Für (x,y) setze x = p(y) wenn Präfix–Summe von (x,y) kleiner
    als Präfix–Summe von (x,y) ist.
    {Das stellt sicher, daß (x,y) die Kante ist, mit der unsere
    Euler–Tour den Knoten y das erste Mal betritt. }
end
```

□

Satz 3.27 Für Algorithmus 3.26 gilt:

(a) Er ist korrekt.

(b) $T^{\infty}(n) = O(\log n)$.

(c) $W(n) = O(n)$.

Beweis:

(b),(c) Man muß hier den nach Bemerkung 3.15 existierenden Algorithmus mit $T^{\infty}(n) = O(\log n)$ für das Parallele Präfix benutzen!

□

Mit derselben Technik des Parallelen Präfix auf der Euler–Tour mit geeigneten Gewichten lassen sich weitere elementare Operationen auf Bäumen ausführen.

Definition 3.28 (Problem postorder Numerierung) Sei ein geordneter Baum $T = (V, E)$ mit Wurzel $r \in V$ gegeben. Ein postorder Durchlauf von T ist die Folge der Knoten von T, die folgendermaßen spezifiziert werden kann:

$$\text{Postorder von } T = (\quad \begin{array}{l} (\text{ Postorder 1. Teilbaum von } r), \\ (\text{ Postorder 2. Teilbaum von } r), \\ \dots \\ (\text{ Postorder } d\text{–ter Teilbaum von } r), r), \end{array}$$

wobei d der Grad von r ist. Für jeden Knoten v ist post(v), die postorder Nummer von v, die eindeutig bestimmte Stelle in dem postorder Durchlauf, an der v steht. Das Problem einen postorder Durchlauf eines Baums T zu bestimmen, ist folgendermaßen spezifiziert:

Eingabe: (1) Ein Baum $T = (V, E)$ mit Wurzel $r \in V$. (2) Die Euler–Tour ET von T, die mit der Kante (r, u) beginnt, wobei u der erste Nachbar von r ist. Es ist dann $S(v, r) = 0$, wo v der letzte Nachbar von r ist.

Ausgabe: Das Feld post(v), wobei sich die Ordnung der Teilbäume eines Knotens durch die Reihenfolge, in der ET die Teilbäume besucht, ergibt.

□

Aufgabe: Man überlege sich, daß die Reihenfolge in der die Euler–Tour ET die Teilbäume eines Knotens u besucht, nur im Falle, daß u die Wurzel ist, gleich der Reihenfolge der Nachbarn von u sein muß. □

Definition 3.29 (Postorder Numerierung)

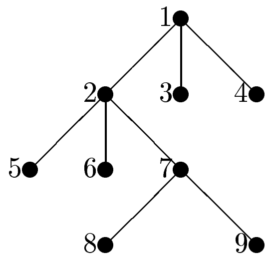
Eingabe: Wie in Definition 3.28.

Ausgabe: Wie in Definition 3.28.

```
begin
  1.Gewichte für die Kanten der Euler–Tour:
    Gewicht von (v,p(v)) = 1, Gewicht von (p(v),v) = 0
    {Beachte, daß das Vorgängerfeld p(v) zur Eingabe gehört.}
  2.Paralleles Präfix auf der Liste S mit den angegebenen Gewichten.
  3.Für v = r setze post(v) = Präfix Summe von (v,p(v)). Für v ist
    post(r) = n, die Anzahl der Knoten des gegebenen Baums.
end
```

□

Beispiel 3.30 Wir betrachten folgenden Baum mit Wurzel 1:



Die Euler–Tour S gehe immer zuerst in den linken Teilbaum eines Knotens.

Euler–Tour	Gewicht	Präfix–Summe
(1,2)	0	0
(2,5)	0	0
(5,2)	1	1
(2,6)	0	1
(6,2)	1	2
(2,7)	0	2
(7,8)	0	2
(8,7)	1	3
(7,9)	0	3
(9,7)	1	4
(7,2)	1	5
(2,1)	1	6
(1,3)	0	6
(3,1)	1	7
(1,4)	0	7
(4,1)	1	8

Der zu der Euler–Tour gehörende postorder Durchlauf ist:

$$(5, 6, 8, 9, 7, 2, 3, 4, 1).$$

So ist post(4) = 8, und Präfix Summe von (4,1) = 8. Man sieht, daß die Kante $(v, p(v))$ die letzte Kante der Euler–Tour EP ist, die den Knoten v berührt. □

Definition 3.31 (Problem Anzahl der Knoten in den Teilbäumen) Gegeben ist wieder ein Baum mit Wurzel. Für jeden Knoten v ist die Anzahl der Knoten in dem Teilbaum mit Wurzel v eindeutig definiert. Wir bezeichnen sie als size(v).

Eingabe: $T = (V, E)$ mit Wurzel r. Euler–Tour beginnend mit r.

Ausgabe: size(v)

Methode: Paralleles Präfix mit Gewichten wie folgt:

Gewicht von $(v, p(v)) = 1$.
Gewicht von $(p(v), v) = 0$.

Dann gilt

size(v) = Präfix–Summe von $(v, p(v))$ = Präfix–Summe von $(p(v), v)$.

□

Beispiel 3.32 (Fortsetzung von Beispiel 3.30)

Euler–Tour	Gewicht	Präfix–Summe
(1,2)	0	0
(2,5)	0	0
(5,2)	1	1
(2,6)	0	1
(6,2)	1	2
(2,7)	0	2
(7,8)	0	2
(8,7)	1	3
(7,9)	0	3
(9,7)	1	4
(7,2)	1	5
(2,1)	1	6
(1,3)	0	6
(3,1)	1	7
(1,4)	0	7
(4,1)	1	8

□

Definition 3.33 (Problem Tiefe der Knoten) Die Tiefe eines Knotens v in einem Wurzelbaum ist die Anzahl der Kanten zwischen v und der Wurzel. Folgender Algorithmus bestimmt die Tiefe:

Eingabe: (1) Ein Baum $T = (V, E)$ mit Wurzel $r \in V$, gegeben durch die Funktion $p(v)$. (2) Die Euler–Tour EP von T, beginnend mit (r, u), wobei u der erste Nachbar von r ist.

Ausgabe: Das Feld level(v), wobei eben level(v) die Tiefe von v in T ist.

Methode: Paralleles Präfix auf S, wobei die Gewichte so gesetzt sind:

Gewicht von $(p(v), v) = +1$.

Gewicht von $(v, p(v)) = -1$.

Der Eintrag level(v) ergibt sich als die Präfix–Summe der Kante $(p(v), v)$.

□

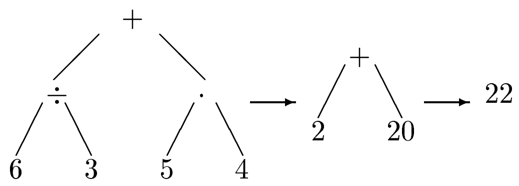
Beispiel 3.34 (Fortsetzung von Beispiel 3.30)

Euler–Tour	Gewicht	Präfix–Summe
(1,2)	1	1
(2,5)	1	2
(5,2)	-1	1
(2,6)	1	2
(6,2)	-1	1
(2,7)	1	2
(7,8)	1	3
(8,7)	-1	2
(7,9)	1	3
(9,7)	-1	2
(7,2)	-1	1
(2,1)	-1	0
(1,3)	1	1
(3,1)	-1	0
(1,4)	1	1
(4,1)	-1	0

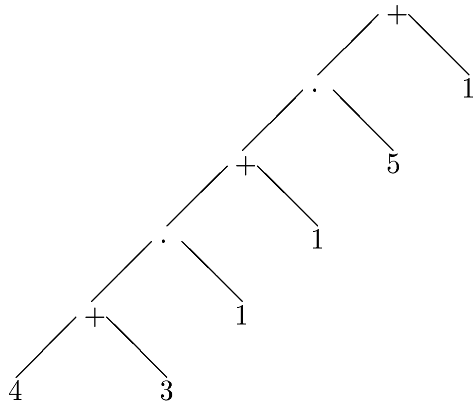
□

3.3 Baumkontraktion

Die Technik der Euler–Tour erlaubt zwar die schnelle Lösung einer Reihe von Baumproblemen, das folgende Problem ist aber nicht nur mit dieser Technik lösbar: die schnelle, parallele Auswertung arithmetischer Ausdrücke, wobei das Problem als Baumproblem aufgefaßt wird:

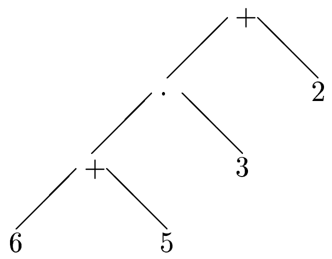


Der Beispielausdruck bietet sich für eine parallele Auswertung direkt an. Anders liegt der Fall bei Ausdrücken der Art:

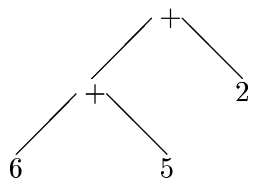


Erfreulicherweise kann man auch solche Ausdrücke in Zeit $O(\log n)$, wobei n die Blätterzahl ist, auswerten. Die Grundidee dieser Vorgehensweise soll zunächst an einigen Beispielen behandelt werden: Was ist das Problem? Um die logarithmische Laufzeit zu erzielen, müssen wir an Knoten "irgendetwas berechnen", obwohl die Argumente noch nicht bekannt sind.

Nehmen wir einmal an, daß zumindest ein Sohn des Knotens, an dem wir etwas machen wollen, ein Blatt ist. Betrachten wir den einfachen Fall

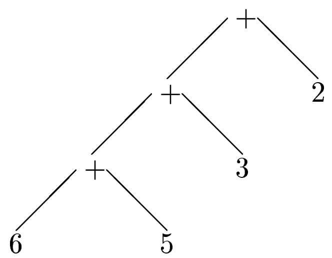


Wir wollen das Blatt mit Inhalt 3 loswerden, d.h. der Baum

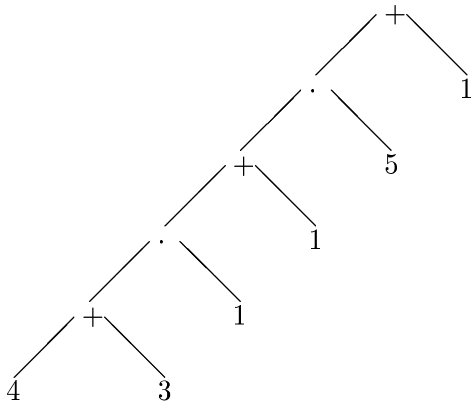


soll herauskommen. Dazu müssen wir uns merken, daß der Wert, den der linke Sohn der Wurzel liefert jetzt nicht der Wert 11 ist, sondern daß noch mit 3 multipliziert werden muß.

Hätten wir dagegen den Fall

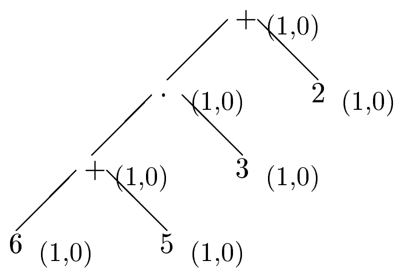


vorliegen, kann nur noch Elimination des Blattes mit der 3 wieder zu dem Baum

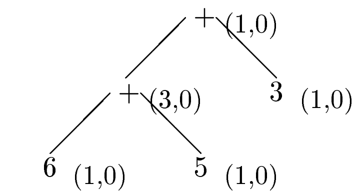


Wir müssen uns jetzt aber merken, daß der linke Sohn der Wurzel nicht 11 als Wert hat, sondern noch die 3 addiert werden muß. Wir brauchen also auf jeden Fall an *jedem* Knoten des Baums eine zusätzliche Information, die uns angibt wie der Wert an dem Knoten beim Hinaufreichen (!) geändert werden muß, damit beim Vater der richtige Wert ankommt.

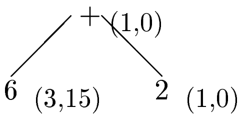
Jeder Knoten bekommt ein zusätzliches label der Form (a, b) mit der Bedeutung: Ist X der Wert, den wir an dem Knoten berechnen, so müssen wir nicht X, sondern den Wert $a \cdot X + b$ nach oben reichen. Dadurch werden die Blätter, die wir zwischen dem Knoten und seinem Vorgänger vorher schon eliminiert haben, berücksichtigt. Am Anfang hat der Baum mit den neuen labels die Form:



Eliminieren wir den Knoten mit der 3, dann bekommen wir



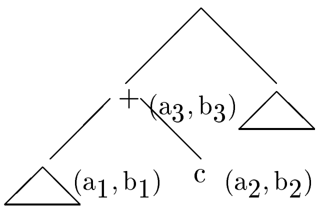
Eliminieren wir nach dem selben Muster die 5, bekommen wir



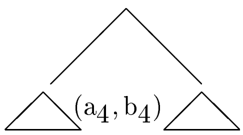
Daraus bekommen wir dann direkt $18 + 15 + 2 = 35$. Und tatsächlich gilt für den Wert des ursprünglichen Baums:

$$2 + (3 \cdot (6 + 5)) = 35.$$

Im allgemeinen haben wir folgende Regeln: Der Baum



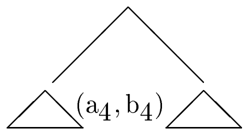
kann zu



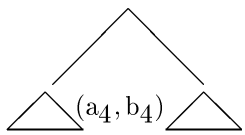
transformiert wurden, wobei sich (a_4, b_4) bestimmen aus der Gleichung:

$$\begin{aligned} a_4 \cdot X + b_4 &= ((a_1 \cdot X + b_1) + (a_2 c + b_2)) \cdot a_3 + b_3 \\ &= a_1 a_3 X + b_1 a_3 + a_2 c a_3 + b_2 a_3 + b_3 \end{aligned}$$

In der Situation



bekommen wir

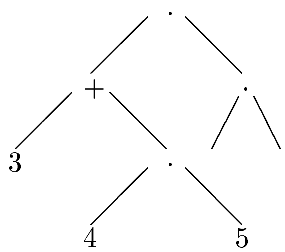


wobei gilt:

$$a_4 X + b_4 = ((a_1 \cdot X + b_1) \cdot a_2 c + b_2) \cdot a_3 + b_3,$$

woraus sich wieder a_4 und b_4 eindeutig ermitteln lassen.

Wir wollen die beschriebene Transformation an möglichst vielen Blättern gleichzeitig ausführen: Dazu ist zu beachten, daß sie nur für solche Blätter parallel ausgeführt werden kann, die selbst und deren Vorgänger *nicht* benachbart sind: Wir können in dem folgenden Baum nicht die Blätter mit 3 und 4 gleichzeitig reduzieren:



Auch nicht 3 und 5. Denn wir merken uns: Die Reduktionsoperation eliminiert ein Blatt zusammen mit seinem Vater und hängt den Bruder an den Vorvorgänger. Eliminieren wir 3 ist der Vorgänger von 4 oder 5 nicht mehr existent!

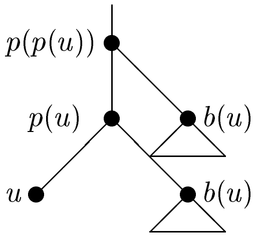
Einige weitere Beispiele überzeugen uns aber, daß man immer einen Wert in etwa in der Nähe der Hälfte aller Blätter parallel reduzieren kann. Wir rechnen also optimistisch mit einem Algorithmus mit

$$T^{\infty}(n) = O(\log n) \quad \text{und} \quad W(n) = O(n)$$

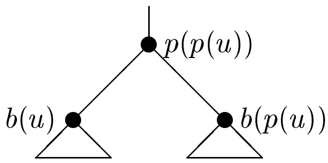
zur Auswertung arithmetischer Ausdrücke über den Operationen $+$ und $\cdot$ mit n Blättern.

Wir studieren den obigen Beispielen zu Grunde liegenden Kontraktionsprozeß zunächst einfach für Bäume und wenden ihn dann auf arithmetische Ausdrücke an. Die Grundoperation der Elimination eines Blatts nennen wir *rake*.

Definition 3.35 Die Operation *rake* angewendet auf ein Blatt n eines Wurzelbaums, so daß $p(n) \neq r$ ist (r ist die Wurzel), arbeitet wie folgt: Der Baum

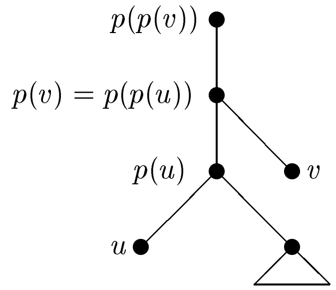


wird zu



Die Knoten u und $p(u)$ werden also eliminiert. □

Die parallele Anwendung dieser Operation auf alle Blätter eines Baums ist nicht ohne weiteres möglich: Die Operation kann nicht auf benachbarte Blätter angewendet werden und auch nicht auf Blätter deren Eltern benachbart sind: In folgender Situation:



können wir nicht u und v gleichzeitig *raken*! Diese Schwierigkeit ist aber leicht zu umgehen.

Algorithmus 3.36 (Baumkontraktion mit rake)

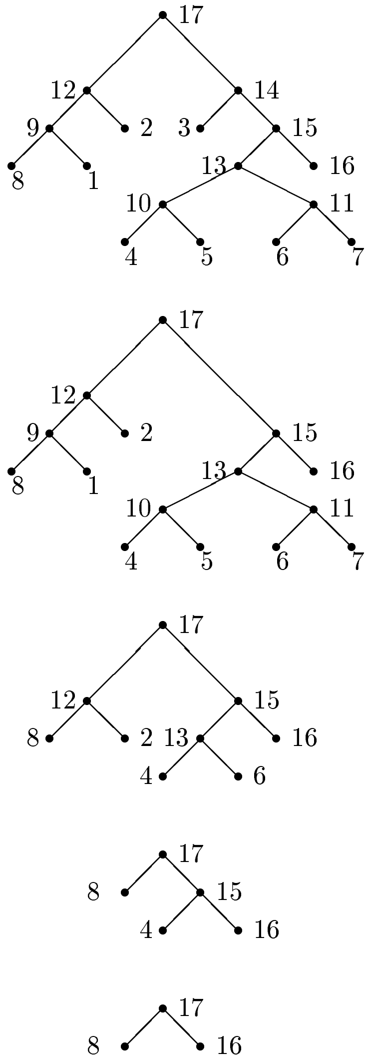
Eingabe: (1) Ein vollständiger binärer Wurzelbaum. (2) Für jeden Knoten v, der nicht die Wurzel ist, seinen Vorgänger $p(v)$ und den Bruder $b(v)$.

Ausgabe: Die Kontraktion des Eingabebaums zu einem Baum bestehend aus genau 3 Knoten.

```
begin
  1.Numerieren der Blätter von links und rechts, ohne das erste und
    letzte Blatt, mit aufeinanderfolgenden Zahlen; Speicherung der
    Blätter in einem Feld A. Sei n die Anzahl der Blätter. Bestimmung
    jedes Blatts, ob es linker oder rechter Sohn sein soll.
  2.for ⌈ log (n+1) ⌉ Iterationen do
    2.1Wende die rake Operation gleichzeitig auf alle Elemente
       von A_odd an, die linker Sohn sind.
       A_odd ist das Feld, das aus den ungeraden
       Einträgen von A besteht.
    2.2Anwendung der rake Operation auf den Rest von A_odd.
    2.3A := A_even
       {A_even ist das Feld, das aus den geradzahligen
       Einträgen von A besteht. }
end
```

□

Beispiel 3.37 Wir geben die Schritte des Algorithmus bei folgenden Baum an:



□

Satz 3.38 Algorithmus 3.36 hat die folgenden Eigenschaften:

(a) Der eingegebene Baum wird korrekt kontraktiert.

(b) $T^{\infty}(n) = O(\log n)$, n ist die Anzahl der Knoten.

(c) $W(n) = O(n)$.

(d) Die Implementierung ist auf einer EREW–PRAM möglich.

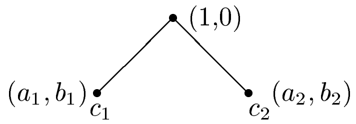
Beweis:

(a) Die Korrektheit folgt, da sichergestellt ist, daß die rake Operation weder auf Blätter mit dem gleichen Vater, noch auf Blätter deren Eltern benachbart sind, angewendet wird.

(b) Haben wir am Anfang einer Iteration m Blätter, so haben wir hinterher $\lfloor m/2 \rfloor$; also ist der Baum nach $\lceil \log(n+1) \rceil$ Iterationen reduziert.

Schritt 1 ist mit der Euler–Tour Technik in Zeit $O(\log n)$ zu machen. Schritt 2.1 und 2.2 gehen in Zeit $O(1)$, Schritt 2.3 ebenfalls. Die Arbeit halbiert sich bei jedem Schleifendurchlauf. □

Definition 3.39 (Auswertung arithmetischer Ausdrücke) Der Algorithmus ist eine Baumkontraktion, wobei die zusätzlichen labels (a, b) mitgeführt werden müssen. Am Ende bekommen wir dann den Baum



Aus diesem können wir den Wert leicht berechnen. □

Wir können also arithmetische Ausdrücke in Zeit $T^{\infty}(n) = O(\log n)$ mit linearer Arbeit berechnen (falls sie als Baum dargestellt sind, so daß die Euler–Tour auf bekannte Weise berechnet werden kann).

4 Suchen, Mischen und Sortieren

Zu diesem Kapitel geht es um die Verarbeitung von Datensätzen (=*records*), deren Schlüssel aus einer linear geordneten Menge kommen. Typische Operationen in diesem Zusammenhang sind natürlich Suchen und Sortieren. Für das ganze Kapitel treffen wir folgende (fast selbstverständliche) Annahme.

Konvention 4.1 Die Schlüssel sind atomare Einheiten, die nur miteinander verglichen werden können. Sie werden nicht als Bitstrings oder Zahlen gesehen, die als solche manipuliert (z.B. addiert) werden dürfen. □

4.1 Suchen

Das Problem des Suchens ist definiert, wie man es erwartet:

Definition 4.2 (Problem Suchen)

Eingabe: Ein Feld $X = (x_1, ..., x_n)$ von n *verschiedenen* Elementen aus einer linear geordneten Menge S, so daß X sortiert ist, also

$$x_1 < x_2 < x_3 < \cdots < x_{n-1} < x_n.$$

Weiterhin ein $y \in S$.

Ausgabe: Der eindeutig bestimmte Index $i \in \{0, ..., n+1\}$, so daß gilt

$$x_i \leq y < x_{i+1},$$

wobei wir setzen: $x_0 := -\infty$ und $x_{n+1} := +\infty$ und natürlich $-\infty < S$ und $S < +\infty$.

□

Mit der Methode des binären Suchens ist das Problem sequentiell in Zeit $T(n) = O(\log n)$ lösbar. Die natürliche Verallgemeinerung dieser Methode ist das parallele, p–adische Suchen. Mit p Prozessoren wird das zu suchende y mit p Elementen von X in einem Schritt verglichen. Diese p Elemente werden so gewählt, daß X in $p+1$ etwa gleichgroße Stücke einteilen. Man sucht dann in dem Stück weiter, in dem y liegen muß, wenn es überhaupt vorkommt.

Algorithmus 4.3 (Paralleles Suchen) Wir geben den Algorithmus hier nicht mit dem pardo–statement an, sondern wir nehmen an, wir haben p Prozessoren $P_1, ..., P_n$ zur Verfügung. Das Programm für den Prozessor P_j lautet:

Eingabe: (1) Ein Feld $X = (x_1, ..., x_n)$, so daß $x_1 < \cdots < x_n$. (2) Ein Element y. (3) Die Gesamtzahl der Prozesoren, wobei $p \leq n$. { Im Falle $p > n$ sind wir direkt fertig. } (4) Die Nummer des Prozessors, hier j.

Ausgabe: Der eindeutig bestimmte Index i, so daß $x_i \leq y < x_{i+1}$, (wobei wie gesagt $x_0 = -\infty$ und $x_{n+1} = \infty$ ergänzt sind).

begin
1. if j = 1 then do
   1.1 l:=0; r:=n+1; x_0:=$-\infty$; x_{n+1}:=$+\infty$
   1.2 c_0:=0; c_{p+1} = 1;
   { Wenn j = 1 ist, haben wir es mit Prozessor 1 zu tun.
   Der macht hier gewisse Initialisierungen:
   l = linker Rand des gesamten Suchintervalls
   r = rechter Rand des gesamten Suchintervalls
   r - l - 1 = # Elemente, unter denen wir y erwarten müssen.}
2. while (r - l) > p do
   { Die Bedingung fragt ab, ob # Elemente, unter denen wir
   y erwarten $\geq$ p ist. }
   2.1 if j = 1 then q_0:=l; q_{p+1} := r;
   2.2 q_j:=l+j$\cdot \left(\left\lfloor \frac{r-l}{p+1} \right\rfloor \right)$
   { Die Zahlen q_0,...,q_{p+1} stellen die Intervallgrenzen der p+1 Teile
   des Suchintervalls dar. Beachte hier: concurrent read! }
   2.3 if y = $x_{(q_j)}$ then return (q_j); exit
   { Da 1 $\leq$ j $\leq$ p werden hier die Vergleiche nur für
   $x_{(q_1)}$,..., $x_{(q_p)}$ vorgenommen. }
   else (if y > $x_{(q_j)}$ then $x_{(c_j)}$:= 0);
   (if y < $x_{(q_j)}$ then $x_{(c_j)}$:= 1);
   2.4 if c_j < c_{j+1} then l:=q_j;r:=q_{j+1}
   { Die Bedingung c_j < c_{j+1} zeigt an, daß
   y > $x_{(q_j)}$ und y < $x_{(q_{(j+1)})}$. }

```
    2.5 if j = 1 and c0 < c1 then l:=q0;r:=q1;
        { Der Fall j=0 wird in 2.4 nicht behandelt.
        Beachte: p Prozessoren, p+1 Teile des Suchintervalls. }
    od
  3. if j ≤ r-l then do
    3.1 case
                y = x_{l+j}; return (l+j); exit
                y > x_{l+j}; c_j:=0
                y < x_{l+j}; c_j:=1
        end case
    3.2 if c_{j-1} < c_j then return (l+j-1)
end
```

□

Beispiel 4.4 Das Feld X ist gegeben durch

$$X = (2, 4, 6, 8, 10, 12, ..., 30),$$

und sei $y = 19$. Die Prozessorzahl ist $p = 2$. Nach der Ausführung der Initialisierung haben wir folgenden Zustand der Variablen:

$$l = 0,\ r = 16,\ c_0 = 0,\ c_3 = 1,\ x_0 = -\infty,\ x_{16} = +\infty.$$

Die Schleife läuft 3 Iterationen. In der Spalte mit Nummer i stehen die Werte nach der i–ten Iteration:

Iter. / Var.	1	2	3
q_0	0	5	7
q_1	5	6	8
q_2	10	7	9
q_3	16	10	10
c_0	0	0	0
c_1	0	0	0
c_2	1	0	0
c_3	1	1	1
l	5	7	9
r	10	10	10

□

Satz 4.5 Für Algorithmus 4.3 gilt:

(a) Er ist richtig.

(b) Die Zeit mit p Prozessoren ist $T^p(n) = O\left(\frac{\log(n+1)}{\log(p+1)}\right)$.

(c) Die Arbeit bei p Prozessoren ist $W^p(n) = O\left(p \cdot \frac{\log(n+1)}{\log(p+1)}\right)$.

Beweis:

(b) Der Algorithmus ist doch relativ subtil. Deshalb müssen wir genau definieren:

Suchintervall bei gegebenen Werten von l und r
$= x_l,\ x_{l+1},\ x_{l+2}, ..., x_r$.

Wir zählen also x_l und x_r noch dazu, obwohl $y \in \{x_l, x_r\}$ nie auftreten kann. Der Grund dafür ist, daß l oder $r-1$ immer noch als Ergebnis auftreten können: Wenn

$$x_l < y < x_{l+1} \quad \text{und} \quad x_{r-1} \leq y < x_r.$$

Zunächst interssieren wir uns für die Anzahl der Schleifendurchläufe. Dazu müssen wir den Wert

$$r - l = (\text{Größe des Suchintervalls}) - 1$$

verfolgen. Haben wir vor der i–ten Iteration der while–Schleife für $i \geq 0$ die Intervallgrenzen r und l gegeben und ist $r - l > p$, so wird das Intervall $x_l, ..., x_r$ durch die Werte $q_0 = l, q_1, ..., q_{p+1} = r$ in $p+1$ Teilintervalle eingeteilt:

$$x_l = x_{q_0}, ..., x_{q_1}, \quad x_{q_1}, ..., x_{q_2} \quad , ..., \quad x_{q_p}, ..., x_{q_{p+1}} = x.$$

Für die Grenzen der ersten p dieser Teilintervalle gilt:

$$\begin{array}{rcl} q_{j+1} - q_j & = & l + (j+1) \cdot \left\lfloor \frac{r-l}{p+1} \right\rfloor - \left(l + j \cdot \left\lfloor \frac{r-l}{p+1} \right\rfloor\right) \\ & = & \left\lfloor \frac{r-l}{p+1} \right\rfloor \leq \frac{r-l}{p+1}, \end{array}$$

wobei $0 \leq j < p$ ist. Ist $j = p$, gilt:

$$q_{j+1} - q_j = r - p \cdot \left\lfloor \frac{r-l}{p+1} \right\rfloor \leq r - p \cdot \left(\frac{r-l}{p+1} - 1\right) = \frac{r-l}{p+1} + p.$$

Wir sehen:

$$
\begin{array}{lcl}
r-l \text{ nach dem 0–ten Durchlauf} & = & n+1 \\
r-l \text{ nach dem 1–ten Durchlauf} & \leq & \frac{n+1}{p+1}+p \\
r-l \text{ nach dem 2–ten Durchlauf} & \leq & \frac{n+1}{(p+1)^2}+p+1,
\end{array}
$$

denn,

$$\frac{\frac{n+1}{p+1}+p}{p+1}+p \leq \frac{n+1}{(p+1)^2}+\frac{p}{p+1}+p,$$

$$r-l \text{ nach dem } i\text{–ten Durchlauf} \quad \leq \quad \frac{n+1}{(p+1)^i}+p+1,$$

denn,

$$\frac{\frac{n+1}{(p+1)^i}+p+1}{p+1}+p = \frac{n+1}{(p+1)^{i+1}}+p+1.$$

Damit folgt für jedes $i > 0$:

$$
\begin{array}{llll}
 & i & < & \frac{\log_2(n+1)}{\log_2(p+1)} \\
\Leftrightarrow & \log_{p+1}(n+1) & < & i \\
\Leftrightarrow & \frac{(n+1)}{(p+1)^i} & < & 1 \\
\Leftrightarrow & \frac{(n+1)}{(p+1)^i}+p+1 & < & p+1 \\
\Leftrightarrow & r-l \text{ nach dem} & & \\
 & i\text{–ten Durchlauf} & \leq & p
\end{array}
$$

Beachte, daß $m^{\log_m(n)} = 2^{\log_2(m)\cdot\log_2(n)} = 2^{\log_2(n)}$ gilt $\log_m(n) = \frac{\log_2(n)}{\log_m(n)}$.

Da jeder einzelne Durchlauf der Schleife die Zeit $O(1)$ braucht und Schritt 1 und 3 ebenfalls folgt die Behauptung

(c) Die Behauptung folgt, da alle p Prozessoren während der Schleife arbeiten.

□

Verfolgen wir noch einmal die Änderung der Werte r und l: Angenommen, der Algorithmus ist irgendwann in der Situation $l = 100$ und $r = 124$ angekommen. Sei die Prozessorzahl $p = 4$. Dann ergeben sich die Intervallgrenzen zu

$$q_0 = l = 100,\ q_1 = 104,\ q_2 = 108,\ q_3 = 112,\ q_4 = 116,\ q_5 = 124,$$

da $\left\lfloor \frac{r-l}{p+1} \right\rfloor = \left\lfloor \frac{24}{5} \right\rfloor = 4$ ist. Man sieht, daß das letzte Intervall auch um 4 größer ist als die anderen Intervalle.

Da alle Prozessoren die Grenzen r und l gleichzeitig lesen müssen, brauchen wir zur Implementierung eine CREW–PRAM.

Es läßt sich zeigen, daß die EREW–PRAM $\Omega((\log n) - (\log p))$ parallele Schritte braucht. Da $\Omega((\log n) - (\log p)) = \Omega(\log n)$, sofern $p \leq n^c$ für eine Konstante $c < 1$, ist kein vernünftiger speedup auf der EREW–PRAM zu erzielen.

Punkt (c) von Satz 4.5 zeigt uns, daß der Algorithmus optimal ist, solange p konstant ist und damit $T(n) = \Theta(\log n)$, d.h. wie beim binären Suchen ist. Es läßt sich jedoch zeigen, daß es keine besseren Algorithmen zum Suchen gibt.

4.2 Mischen

In Unterkapitel 2.4 haben wir die Technik der Partitionierung angewendet, um zwei geordnete Folgen in Zeit $T^\infty(n) = O(\log n)$ mit Arbeit $W(n) = O(n)$ zu mischen. Dazu brauchten wir eine CREW–PRAM. Hier zeigen wir, wie wir mit Anwendung des parallelen Suchalgorithmus sogar optimal in doppelt–logarithmischer Zeit mischen können.

Erinnern wir uns einiger Definitionen aus Unterkapitel 2.4. Wir haben es hier immer mit Elementen einer total geordneten Menge zu tun. Sei X eine Folge von Elementen, x ein Element, dann ist der *Rang* von x in X gegeben durch

$$\text{Rang}(x : X) = \# \text{ Elemente von } X, \text{ die} \leq x \text{ sind.}$$

Sind X und Y zwei Folgen von Elementen, so ist

$$\text{Rang}(Y : X) \;= (r_1, ..., r_n),$$

wobei eben $r_i = \text{Rang}(y_i : X)$ und $Y = (y_1, ..., y_n)$. Man beachte, daß es hier auf die Ordnung in Y ankommt. Sind X und Y zwei geordnete Folgen, so reicht es um X und Y zu mischen, die Folgen $\text{Rang}(Y : X)$ und $\text{Rang}(X : Y)$ zu berechnen.

Lemma 4.6 Sei Y eine beliebige Folge mit $m = m(n)$ Elementen und X eine sortierte Folge mit n Elementen. Sei $m = O(n^s)$ für ein festes $s < 1$. Dann gilt:

Wir können für alle Elemente y aus Y das Feld Rang$(Y : X)$ mit $T^\infty(n) = O(1)$ und $W(n) = n$ ermitteln.

Beweis:
Da X geordnet ist (!), können wir unseren Suchalgorithmus anwenden. Wir wenden ihn mit
$$p = \lfloor n/m \rfloor = \Omega(n^{1-s})$$
an. Dann gilt:
$$\text{Zeit für 1 Element } = O\left(\frac{\log(n+1)}{\log(p+1)}\right) = O(1),$$
da
$$\frac{\log(n+1)}{\log(p+1)} \leq c \cdot \frac{\log(n+1)}{\log(n^{1-s}+1)} \leq d \cdot \frac{\log n}{(1-s)\cdot \log n} = \frac{d}{1-s}$$
und s eine Konstante ist. Die Arbeit ist dann $W(n) = O(\frac{n}{m})$, da die Laufzeit konstant ist.

Lassen wir nun den angegebenen Algorithmus für alle Elemente von Y parallel laufen, so folgt das Gewünschte. $\square$

Frage: Wieviele Iterationen macht unser Suchalgorithmus bei n Elementen mit $\sqrt{n}$ Prozessoren, wieviele mit $\sqrt[3]{n}$ Prozessoren, wieviele bei n^s, $s < 1$ konstant vielen Prozessoren?

Der Algorithmus der Lemma 4.6 zur Folge hat kann auf einer CREW–PRAM implementiert werden.

Jetzt sollen 2 geordnete Folgen A und B gemischt werden. Wir nehmen an, daß alle vorkommenden Elemente paarweise verschieden sind. Wir wenden das Prinzip der Partitionierung an

Algorithmus 4.7 (Rang$(A : B)$, wobei A und B geordnet sind)

Eingabe: Zwei Felder $A = (a_1, ..., a_n)$ und $B = (b_1, ..., b_m)$ aufsteigend geordnet. Sei $\sqrt{m}$ ganzzahlig.

Ausgabe: Das Feld Rang$(B : A)$.

```
begin
  1.Falls die Länge von B, m < 4, berechne Rang(B:A) unter
    Anwendung von Algorithmus 4.3 mit p = n und exit.
```

2. Berechne den Rang der Elemente $b_{\sqrt{m}}, b_{2\cdot\sqrt{m}}, ..., b_{i\cdot\sqrt{m}}, ..., b = b_{\sqrt{m}\cdot\sqrt{m}}$ in A. Wende dazu Algorithmus 4.3 mit $p = \sqrt{n}$ gleichzeitig auf alle b's an. Setze das Feld j durch j(i)=Rang($b_{i\cdot\sqrt{m}} : A$) für alle i mit $1 \leq i \leq \sqrt{m}$. Setze weiter $j(0) = 0$.
3. Für i mit $0 \leq i \leq \sqrt{m} - 1$ setze
   $B_i = (b_{(i\cdot\sqrt{m})+1}, ..., b_{((i+1)\cdot\sqrt{m})-1})$
   $A_i = (a_{j(i)+1}, ..., a_{j(i+1)})$ falls $j(i) < j(i+1)$.

Jetzt werden die Felder Rang(B_i, A_i) berechnet. Ist $a_{j(i)} = a_{j(i+1)}$ und damit das Feld A_i leer, so setze Rang$(B_i : A_i)$=(0,...,0). Ist dagegen $a_{j(i)} < a_{j(i+1)}$, berechne Rang$(B_i : A_i)$ rekursiv.

4. Ist k mit $1 \leq k \leq m$ ein beliebiger Index, der nicht ein Vielfaches von $\sqrt{m}$ ist, setze $i = \left\lfloor \frac{k}{\sqrt{m}} \right\rfloor$.
   Setze weiter Rang($b_k : A$) = j(i) + Rang($b_k : A_i$).

end

$\square$

Erläuterung und Beispiel

Die von Algorithmus 4.7 vorgenommene Aufteilungen haben folgende Form:

Die B_i haben alle $\sqrt{m}-1$ viele Elemente, wogegen die Größe der A_i variiert. Beachte auch noch, daß $j(i) = \text{Rang}(b_{i\cdot\sqrt{m}} : A)$ ist.

Wir betrachten ein Beispiel:

$$\begin{array}{lcl} \text{A} & = & (-5,0,3,4,17,18,24,28) \\ \text{B} & = & (1,2,15,21) \end{array}$$

Dann sind die b's aus Schritt 2 gegeben durch 2,21. Das Feld j ergibt sich zu: $j(0) = 0,\ j(1) = 2,\ j(3) = 6$.

Schritt 3 gibt uns

$$B_0 = (1)\ ,\ B_1 = (15)$$

und

$$A_0 = (-5, 0)\ ,\ A_1 = (3, 4, 17, 18).$$

Wir bekommen

$$\text{Rang}(1 : A_0) = 2\ ,\ \text{Rang}(15 : A_1) = 2.$$

In Schritt 4 passen wir die Ränge an und bekommen

$$\text{Rang}(B : A) = (2, 2, 4, 6).$$

□

Satz 4.8 Sind A und B geordnete Folgen der Längen n und m, dann gilt für den Algorithmus 4.7:

(a) Er berechnet tatsächlich das Feld Rang$(B : A)$.

(b) Die parallele Zeit ist $O(\log(\log m))$.

(c) Die Arbeit beträgt $O((n+m) \cdot \log(\log m))$ Operationen.

Beweis

(a) Der Beweis geht induktiv über m. Am Induktionsanfang ist $m < 4$, dort gilt die Behauptung mit Schritt 1. Angenommen, die Behauptung gilt für alle $m' < m$ und $m > 4$. Offensichtlich gilt, daß

$$b_{i\sqrt{m}} < B_i < b_{(i+1)\sqrt{m}}$$

für $1 \leq i \leq (\sqrt{m}-1)$ und $B_0 < b_{1 \cdot \sqrt{m}}$. Damit folgt $a_{j(i)} < B_i < a_{j(i+1)}$ für $1 \leq i \leq (\sqrt{m}-1)$ und $A_0 < b_{1 \cdot \sqrt{m}}$. Also gilt für $b \in B_i$ und $0 \leq i \leq \sqrt{m}-1$, daß

$$\text{Rang}(b : A) = j(i) + \text{ Rang}(b : A_i),$$

da $j(i) = \#$ Elemente von A_i ist. Die Korrektheit des Algorithmus folgt aus der Induktionsvoraussetzung, da die Länge von B_i kleiner als die Länge von B ist.

(b),(c) Sei $T^{\infty}(n) =$ die Zeit, die wir für Folgen A und B brauchen, wobei A aus n und B aus m Elementen besteht:

Schritt 1 braucht konstante Arbeit und Zeit.

Schritt 2 führt $\sqrt{m}$–mal parallel den parallelen Suchalgorithmus 4.3 mit $p = \sqrt{n}$ aus.

$$\begin{array}{llll} \text{Parallele Zeit} & = & O\left(\frac{\log(n+1)}{\log(p+1)}\right) & = & O(1) \\ \text{Arbeit} & = & O(\sqrt{m} \cdot \sqrt{n}) & = & O(n+m), \end{array}$$

denn $O(\sqrt{m} \cdot \sqrt{n}) \leq \; Max \; \{m, n\} \leq m + n.$

Schritt 3 ohne die rekursiven Aufrufe braucht

$$\begin{array}{lcl} \text{Parallele Zeit} & = & O(1) \\ \text{Arbeit} & = & O(n+m). \end{array}$$

Schritt 4 erhöht Arbeit und Zeit nicht.

Zu den rekursiven Aufrufen: Sei

$$m_i = \text{Größe von } A_i$$

für $0 \leq i \leq \sqrt{m} - 1$. Der Aufruf für das Paar von Listen (B_i, A_i) braucht parallele Zeit $T(n_i, \sqrt{m})$. Also

$$T(n,m) \leq Max\{T(n_i, \sqrt{m}) \;\; |0 \leq i \leq \sqrt{m} - 1\} + O(1)$$

und

$$T(n,3) = O(1) \;\; \text{für alle } n.$$

Daraus folgt

$$T(n,m) = O(\log(\log m)).$$

Beachte, daß gilt

$$m^{\left(\frac{1}{2}\right)^i} \leq C \rightarrow i = O(\log(\log m)).$$

Die Arbeit in jedem Satz rekursiver Aufrufe ist $O(n+m)$ also ist die Gesamtarbeit $O((n+m) \cdot \log(\log m))$.

□

Natürlich brauchen wir eine CREW–PRAM zur Implementierung des Algorithmus.

Folgerung 4.9 Sind A und B geordnete Folgen der Länge n, so können wir A und B in paralleler Zeit

$$T^{\infty}(n) = O(\log(\log n))$$

mit Arbeit

$$W(n) = O(n \cdot \log(\log n))$$

mischen. □

Als nächstes machen wir Algorithmus 4.3 optimal. Wenn wir einen sehr schnellen, aber nicht optimalen Algorithmus haben, ist folgendes eine brauchbare Technik:

(1) Verkleinere das Problem mit einem optimalen, aber langsamen Algorithmus soweit, bis wir den nicht optimalen, aber schnellen Algorithmus ohne Arbeitserhöhung annehmen können.

(2) Wende den schnellen, aber nicht optimalen Algorithmus auf das in (1) passend verkleinerte Problem an.

Wir bekommen auf die Art einen Algorithmus, der zwei geordnete Folgen der Länge n in paralleler Zeit $O(\log(\log n))$ und Arbeit $O(n)$ mischt. Da der parallele Suchalgorithmus angewendet wird, brauchen wir zur Implementierung eine CREW–PRAM. Den eben beschriebenen Mischalgorithmus wenden wir jetzt an, um einen optimalen Sortieralgorithmus, d.h. mit Arbeit $W(n) = O(n \cdot \log n)$ mit paralleler Laufzeit $T^{\infty}(n) = O(\log n \cdot (\log(\log n)))$ zu bekommen. Der Algorithmus arbeitet nach der Strategie des merge–sort, d.h. Sortieren durch Mischen. Diese Strategie läßt sich als Anwendung des divide–and–conquer Prinzips verstehen:

(1) Teile die Eingabefolge in 2 disjunkte Teilfolgen.

(2) Sortiere getrennt, meistens rekursiv.

(3) Mische zusammen.

Bei dem hier betrachteten Algorithmus merge–sort schenken wir uns die Schritte 1 und 2, die divide Phase. Wir fangen direkt mit der conquer Phase an. D.h. der folgende Baum wird von den Blättern ausgehend abgearbeitet: Die Eingabefolge ist durch $(x_1, ..., x_8)$ gegeben.

Der dazugehörige Algorithmus sieht so aus:

Algorithmus 4.10 (Einfaches merge–sort)

Eingabe: Ein Feld X der Größe n, wobei $n = 2^l$.

Ausgabe: Ein balancierter binärer Baum mit n Blättern. Für $0 \leq k \leq \log n$ enthält $L(k, j)$ die Sortierung von $X(2^k \cdot (j-1)+1), ..., X(2^k \cdot j)$.

begin
  1. $\mathbf{1 \leq j \leq n}$
    $\mathbf{L(0,j) := X(j)}$
  2. $\mathbf{h = 1}$ **to** $\log \mathbf{n}$ **do**
    $\mathbf{1 \leq j \leq \frac{n}{2^k}}$
      Mische $\mathbf{(L(k-1, 2 \cdot j - 1)}$ **und** $\mathbf{L(k-1, 2 \cdot j)}$
      in die geordnete Liste $\mathbf{L(k,j)}$**.**
end

□

Satz 4.11 Für Algorithmus 4.10 gilt:

(a) Er sortiert das Eingabefeld korrekt.

(b) $T^{\infty}(n) = O(\log n \cdot (\log(\log n)))$.

(c) $W(n) = O(n \cdot (\log n))$.

Beweis:

(b),(c) Die Anzahl der Iterationen von Schritt 2 ist $O(\log n)$. Da wir auf jeder Stufe *insgesamt* n Elemente mischen, brauchen wir für jede Iteration die Zeit $O(\log(\log n))$ mit $O(n)$ Operationen. Daraus folgt die Behauptung. (Wir nehmen hier den oben nur vage beschriebenen Mischalgorithmus.)

□

Zur Implementierung des Algorithmus brauchen wir eine CREW–PRAM.

Bemerkung 4.12 Der berühmte, aber komplizierte Algorithmus namens Cole's merge sort sortiert in

$$T^{\infty}(n) = O(\log n) \text{ mit } W(n) = On \cdot \log n.$$

□

5 Randomisierte Algorithmen

Bisher haben wir nur deterministische Algorithmen zur Lösung unserer Probleme kennengelernt. Hier betrachten wir Algorithmen, die mit einem Zufallszahlengenerator arbeiten, sogenannte randomisierte Algorithmen. Es ist eine der interessanteren Erkenntnisse der Algorithmenforschung, daß randomisierte Algorithmen für manche Probleme bemerkenswert einfach und effizient sein können.

Zur Analyse der Laufzeit randomisierter Algorithmen brauchen wir einige Grundtatsachen aus der Wahrscheinlichkeitstheorie, die wir zunächst wiederholen wollen.

5.1 Wahrscheinlichkeitstheoretische Grundlagen

Fangen wir einfach mit 2 Beispielen an:

Beispiel 5.1 (Münzen werfen - coin tossing) Angenommen wir werfen n–mal eine Münze. Das Ergebnis dieser Würfe ist entweder Kopf, abgekürzt durch K, oder Zahl, abgekürzt durch Z. Damit können wir unsere n Münzwürfe als Folge $L_1L_2...L_n$, wobei $L_i \in \{K, Z\}$ für alle i ist, darstellen. Jede Folge der Länge n ist eine Elementarereignis. Die Menge aller Folgen ist der Wahrscheinlichkeitsraum. Teilmengen des W–Raums nennen wir Ereignisse. Die Menge

$$E = \{L_1...L_n | \text{Die Folge } L_1...L_n \text{ enthält genau 2–mal Kopf.}\}$$

ist ein Ereignis. Es gilt, daß

$$|E| = \binom{n}{2}$$

ist. □

Beispiel 5.2 (Kugeln und Eimer) Wir werfen irgendwie zufällig m verschiedene Kugeln, die von 1 bis m durchnumeriert sind, in n Eimer, die von 1 bis n durchnumeriert sind.

$$|\text{Wahrscheinlichkeitsraum}| = |\{f : \{1, ...m\} \to \{1, ..., n\}\}| = n^m.$$

Ein Ereignis in diesem Raum ist, daß die ersten beiden Eimer, also 1 und 2 leer sind. Sei E dieses Ereignis, dann:

$$E = \{f : \{1, ...m\} \to \{3, ..., n\}\} \text{ und } |E| = (n-2)^m.$$

□

Definition 5.3

(a) Wahrscheinlichkeitsraum = Menge von Elementarereignissen.

(b) Ereignis = Teilmenge eines Wahrscheinlichkeitsraums.

(c) Raum diskret $\Leftrightarrow$ Raum endlich oder abzählbar unendlich.

(d) Ein Wahrscheinlichkeitsmaß auf einem diskreten W–Raum S ist eine Funktion

$$Pr : P(S) \to R$$

so daß gilt:

(i) Für alle Elemente A ist $0 \leq Pr(A) \leq 1$.

(ii) $Pr(S) = 1$.

(iii) $Pr(A \cup B) = Pr(A) + Pr(B)$, falls $A \cap B = \emptyset$.

Wir sagen

$$Pr(A) = \text{Wahrscheinlichkeit von } A.$$

□

Beispiel 5.4 (Fortsetzung von Beispiel 5.1) Sei S der W–Raum des Experiments, dann

$$S = \{K, Z\}^n.$$

Wir definieren ein W–Maß Pr durch

$$Pr(A) = \frac{|s|}{2^n}.$$

Dann gilt, daß Pr die in 5.3 (d) geforderten Eigenschaften hat. Durch die Funktion Pr nehmen wir an, daß die Münze vollkommen korrekt ist (unbiased coin). Sprechweise: Ist jedes Elementarereignis gleichwahrscheinlich, dann hat das zugehörige Wahrscheinlichkeitsmaß die uniforme Wahrscheinlichkeitsverteilung. Hier ist die Wahrscheinlichkeit eines Elementarereignisses durch $\frac{1}{2^n}$ gegeben. □

Definition 5.5 Seien $A, B \in S$, S ein W–Raum mit W–Maß Pr. Sei $Pr(B) > 0$. Die bedingte Wahrscheinlichkeit von A vorausgesetzt B ist definiert durch

$$Pr(A|B) = \frac{Pr(A \cap B)}{Pr(B)}.$$

A und B heißen unabhängig $\Leftrightarrow$ $Pr(A \cap B) = Pr(A) \cdot Pr(B)$. □

Bemerkung 5.6

(a) Ist $Pr(A), Pr(B) > 0$ und sind A, B unabhängig, so ist

$$Pr(A|B) = Pr(A) \text{ und } Pr(B|A) = Pr(B).$$

(b) Es gilt Boole's Ungleichung

$$Pr\left(\bigcup_i A_i\right) \leq \sum_i Pr(A_i).$$

$\square$

Beispiel 5.7 (Fortsetzung von Beispiel 5.1 und 5.4) 3 Münzwürfe.

$$W\text{–Raum } = \{KKK, ..., ZZZ\}$$

Zwei Ereignisse A und B

$$\begin{aligned} A &= \text{Menge der Elementarereignisse mit} \geq 2 \text{ mal Zahl.} \\ B &= \text{Menge der Elementarereignisse mit Zahl im ersten Wurf.} \end{aligned}$$

Was ist $Pr(A|B)$? Da $A \cap B = \{ZKZ, ZZK, ZZZ\}$ ist

$$Pr(A|B) = \frac{3}{4}.$$

Man beachte, daß $Pr(A) = Pr(B) = \frac{1}{2}$. Weiter ist

$$Pr(A \cup B) = \frac{5}{8}$$

aber $Pr(A) + Pr(B) = 1$. $\square$

Beispiel 5.8 (Fortsetzung von Beispiel 5.2) Sei jetzt m viel größer als n,d.h.$m \geq 2 \cdot n \cdot (\ln n)$. Es gibt n^m viele Möglichkeiten und jede ist gleichwahrscheinlich. Wir wollen die Wahrscheinlichkeit von ≥ 2 leeren Eimern von oben beschränken. Sei $i \neq j$ mit $1 \leq i, j \leq n$. Sei

$$E_{ij} = \text{Ereignis, daß Eimer } i \text{ und } j \text{ leer bleiben.}$$

Dann gilt

$$Pr(E_{ij}) = \frac{(n-2)^m}{n^m} = \left(\frac{n-2}{n}\right)^m = \left(1 - \frac{2}{n}\right)^m.$$

Mit Boole's Ungleichung folgt:

$$Pr\left(\bigcup_{i\neq j} E_{ij}\right) \leq \left(\begin{array}{c} n \\ 2 \end{array}\right) \cdot \left(1-\frac{2}{n}\right)^m.$$

Da $1+x \leq e^x$ für alle x, folgt

$$Pr\left(\bigcup_{i\neq j} E_{ij}\right) \leq \left(\begin{array}{c} n \\ 2 \end{array}\right) \cdot e^{-\frac{2m}{n}} \leq \frac{1}{n^2}.$$

□

Definition 5.9

(a) Eine Zufallsvariable X ist eine Funktion

$$X : S \to R,$$

wobei S ein W–Raum mit W–Maß Pr ist.

(b) Sei X eine Zufallsvariable. Für $x \in R$ ist das Ereignis "$X = x$" gegeben durch:

$$(X = x) = \{w \in S | X(w) = x\} = X^{-1}(x).$$

Die Wahrscheinlichkeitsverteilung von X ist die Funktion $p : R \to [0,1]$ mit

$$p(x) = Pr(X = x).$$

Die Verteilungsfunktion von X ist die Funktion $F : R \to [0,1]$ mit

$$F(x) = Pr(X \leq x).$$

□

Beispiel 5.10 Betrachten wir W–Raum $S = \{K, Z\}^n$. Dann ist

$$X : S \to R$$

$$X(L_1...L_n) = \#\text{Kopf in } L_1...L_n$$

eine ZV. Verteilung von X:

$$p(x) = \left(\begin{array}{c} n \\ x \end{array}\right) \cdot \left(\frac{1}{2}\right)^x \cdot \left(\frac{1}{2}\right)^{n-x} = \left(\begin{array}{c} n \\ x \end{array}\right) \cdot \left(\frac{1}{2}\right)^n.$$

Man beachte, daß wir hier brauchen $Pr(A \cup B) = Pr(A) + Pr(B)$, wenn A und B disjunkt sind.
Verteilungsfunktion von X:

$$F(x) = R(X \leq x) = \sum_{j=0}^{x} \binom{n}{j} \cdot \left(\frac{1}{2}\right)^n .$$

□

Definition 5.11 Sei X eine ZV auf einem dikreten W–Raum S.

(a) Der Erwartungswert oder Mittelwert von X ist gegeben durch

$$E(X) = \sum_x x \cdot p(x).$$

Bezeichnungsweise $\mu_X = EX$.

(b) Die Varianz von X ist gegeben durch

$$V(X) = E(X - \mu_X)^2 = E(X^2) - (\mu_X)^2 \geq 0.$$

Die Standardabweichung σ_x von X ist gegeben durch

$$(\sigma_X)^2 = VX \text{ und } \sigma_X \geq 0.$$

□

Es gilt der folgende Satz:

Satz 5.12

(a) Sei X eine ZV so daß $X \geq 0$, dann gilt:

$$Pr(X \geq a) \leq \frac{EX}{a} \text{ für } a > 0.$$

(Markov–Ungleichung)

(b) Sei X eine ZV mit Standardabweichung σ und Mittelwert μ, so gilt:

$$Pr\{|X - \mu| > t \cdot \sigma\} \leq \frac{1}{t^2}.$$

(Tschebyscheff Ungleichung)

(c) Sei $X_1, ..., X_n$ eine Familie beliebiger Zufallszahlen, so gilt:

$$E(\sum X_i) = \sum(EX_i).$$

Für jede Zufallsvariable X und jede Zahl a gilt:

$$E(a \cdot X) = a \cdot (EX).$$

D.h. der Erwartungswert ist in jedem Falle linear.

□

Definition 5.13

(a) Ein Bernoulli Experiment ist eines mit genau 2 möglichen Reusltaten: Erfolg oder Verlust.

$$\begin{array}{lcl} p & = & \text{Wahrscheinlichkeit des Erfolgs.} \\ q = 1 - p & = & \text{Wahrscheinlichkeit des Verlusts.} \end{array}$$

Ein Münzwurf ist ein Bernoulli Experiment.

(b) Sei $n > 0$

$$X \quad = \quad \text{\# von Erfolgen in } n \text{ unabhängig aufeinanderfolgenden Bernoulli Experimenten.}$$

Dann gilt für $0 \leq k \leq n$, daß

$$p(k) = Pr(X = k) = \left(\begin{array}{c} n \\ k \end{array} \right) \cdot p^k \cdot q^{n-k}.$$

Man beachte, daß wir auch hier wieder gebrauchen

$$Pr(A \cup B) = Pr(A) + Pr(B) \text{ für } A \cap B = \emptyset.$$

Wir nennen p die Binomialverteilung mit Parametern n und p. Für p schreiben wir $b(k; n, p)$.

□

Beispiel 5.14 Wir betrachten wieder das Beispiel mit den n Eimern und den m Bällen. Wir suchen die Wahrscheinlichkeit dafür, daß ein fester Eimer genau k Bälle enthält, wobei natürlich $0 \leq k \leq m$ ist. Wir haben $\begin{pmatrix} m \\ k \end{pmatrix}$ viele Möglichkeiten, die k Bälle auszuwählen. Die verbleibenden $m-k$ Bälle müssen in irgendeinem anderen Eimer kommen. Also ist die Wahrscheinlichkeit unseres Ereignisses gegeben durch

$$\begin{pmatrix} m \\ k \end{pmatrix} \cdot \frac{(n-1)^{m-k}}{n^m} = \begin{pmatrix} n \\ k \end{pmatrix} \cdot \frac{1}{n^k} \cdot \left(1 - \frac{1}{n}\right)^{m-k} .$$

Das ist die Binomialverteilung $b(k; m, 1/n)$.

Jetzt berechnen wir den Erwartungswert einer ZV X mit Binomialverteilung $b(k; n, p)$. Sei $1 \leq i \leq n$ und

$$X_i = \begin{cases} 1 & \text{falls } i\text{–tes Bernoulli Experiment Erfolg} \\ 0 & \text{sonst} \end{cases}$$

Dann ist $E(X_i) = p$, wobei $X = \sum X_i$ ist. Damit ist $EX = p \cdot n$, da Erwartungswerte linear sind. □

Die folgenden kombinatorischen Tatsachen sind noch wichtig:

Satz 5.15

(a)

$$(x+y)^n = \sum_{i=0}^{n} \begin{pmatrix} n \\ i \end{pmatrix} \cdot x^i \cdot y^{n-i}$$

für alle x, y
(Binomialsatz)

(b)

$$n! = \sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n (1 + o(1))$$

(Stirlings Formel)

(c)

$$\begin{pmatrix} n \\ k \end{pmatrix}^k \leq \begin{pmatrix} n \\ k \end{pmatrix} \leq \begin{pmatrix} ne \\ k \end{pmatrix}^k$$

(Schranken für Binomialkoeffizienten)

Beachte, daß

$$\left(1+\frac{1}{n}\right)^n \to e$$

für $n \to \infty$. $\square$

5.2 Pattern matching

Das pattern matching Problem tritt beispielsweise beim Texteditieren, bei der Bild- und Spracherkennung oder auch in der Molekularbiologie auf.

Definition 5.16 (Problem pattern matching) Das Problem des pattern matching ist folgendermaßen definiert:

Eingabe: Ein string T über dem Alphabet {0,1}, dargestellt als Feld der Länge n : $T = (T(1), ..., T(n)) = T[1 : n]$ und ein string $P = (P(1), ..., P(m)) = P[1 : m]$. ($T$ steht für Text, P für pattern.)

Ausgabe: Das boolesche Feld Match$[1 : n-m+1]$ mit Match$(i) = 1 \Leftrightarrow (X(i), ..., X(i+m-1)) = (T(1), ..., T(m))$.

Wir gehen wie immer davon aus, daß die Feldindizes nur so groß werden, daß sie in einem Schritt bearbeitet werden können. $\square$

Das pattern matching Problem ist sequentiell in linearer Zeit lösbar. Ein erster paralleler Algorithmus, der in paralleler Zeit $O(1)$ mit Arbeit $O(n \cdot m)$ arbeitet ist:

Eingabe: T und P wie oben.

Ausgabe: Match wie oben.

```
begin
  1.for 1 ≤ i ≤ n − m + 1
      for 1 ≤ j ≤ m pardo
        B(i,j) := 1;
        {Initialisierung von B(i,j).}
  2.for 1 ≤ i ≤ n − m + 1
      for 1 ≤ j ≤ m pardo
        if T(i+j-1) ≠ P(j) then B(i,j) = 0
  3.for 1 ≤ i ≤ n − m + 1
      Match(i) := B(i,1) ∧...∧ B(i,m).
end
```

Frage: Welche PRAM brauchen wir für angegebenen Algorithmus? Man beachte die Konjunktion der $B(i,j)$. Wir lernen einen randomisierten Algorithmus kennen mit $T^{\infty}(n) = O(\log n)$ und $W(n) = O(n)$, der mit einer gewissen (kleinen) Wahrscheinlichkeit Match(i)=1 ausgibt, obwohl pattern P nicht matcht. Diese Fehlerwahrscheinlichkeit kann beliebig klein gemacht werden, bleibt aber > 0! Es handelt sich um einen sogenannten Monte–Carlo–Algorithmus.

Wichtig ist es, strings, dargestellt als Feld über {0,1}, in sequentieller Zeit $O(1)$ zu vergleichen, denn es müssen ja $n-m+1$ strings miteinander verglichen werden:

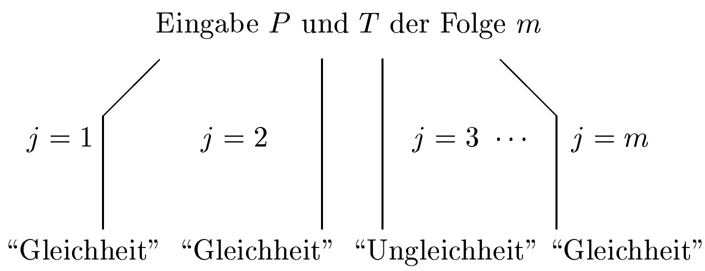
$$\begin{array}{lcl} T[1:m] & \text{mit} & P[1:m] \\ T[2:m+1] & \text{mit} & P[1:m] \\ & \vdots & \\ T[n-m+1:n] & \text{mit} & P[1:m] \end{array}$$

Damit hätten wir dann lineare Arbeit.

Die Möglichkeit der Randomisierung und die Möglichkeit Fehler zu machen, erlaubt uns solche schnellen Vergleiche. Die Idee ist, einfach nur einen Teil der Information von $T[i:i+m-1]$ und $P[1:m]$ zu berücksichtigen. Ein erster Gleichheitstest (mit Fehlerwahrscheinlichkeit > 0) könnte so aussehen (für $P[1:m]$ und $T[1:m]$):

1. Wähle zufällig eine Zahl aus $j \in \{1, ..., m\}$.
2. Falls T(j) = P(j) melde Gleichheit, sonst Ungleichheit.

Dann gilt auf jeden Fall, daß dieser Text Ungleichheit von P und T richtig erkennt. Dagegen kann er "Gleichheit" ausgeben, obwohl P und T ungleich sind. Mit welcher Wahrscheinlichkeit geschieht das? Was ist überhaupt der betrachtete Wahrscheinlichkeitsraum? Für *jedes* Paar von Eingaben P und T bekommen wir einen Wahrscheinlichkeitsraum von Rechnungen, dargestellt durch folgenden Berechnungsbaum:



Jeder Ast des Baums wird mit gleicher Wahrscheinlichkeit, nämlich mit Wahrscheinlichkeit $\frac{1}{m}$ durchlaufen. Die Elementarereignisse des Wahrscheinlichkeitsraums sind also die Elemente:

$$\text{Rechnung für } j = 1,\ \text{Rechnung für } j = 2,\ ...,\ \text{Rechnung für } j = m.$$

Jedes Elementarereignis wird mit gleicher Wahrscheinlichkeit angenommen. Ist nun $P = T$, so gilt: Das Ereignis “richtiges Ergebnis” hat die Wahrscheinlichkeit 1! Ist aber $P \neq T$, so kann das Ereignis “richtiges Ergebnis” mit Wahrscheinlichkeit 1 auftreten. (wenn z.B. $P = (1, 0, ..., 1, 0)$ und $P = (0, 1, ..., 0, 1)$, also wenn sich P und T überall unterscheiden). Es kann aber auch (sozusagen im worst–case) nur mit Wahrscheinlichkeit $\frac{1}{m}$ auftreten (wenn sich T und P an genau einer Stelle unterscheiden.)

Zusammenfassend können wir sagen:

$$\begin{aligned}
\text{Prob (Ausgabe “Ungleichheit” und Eingaben gleich)} \quad &= \quad 0 \\
\text{Prob (Ausgabe “Gleichheit” und Eingaben ungleich)} \quad &\leq \quad 1 - \tfrac{1}{m}
\end{aligned}$$

Ein Trick ist es nun, den Algorithmus mehrmals, sagen wir k–mal, laufen zu lassen. (Was ist dann der zu betrachtende Wahrscheinlichkeitsraum?) Wir bekommen folgende Wahrscheinlichkeiten :

$$\begin{aligned}
\text{Prob (}\quad &\text{In } k \text{ Läufen kommt} \geq 1\text{–mal “Ungleichheit”} \\
&\text{als Ausgabe, aber die Eingaben sind gleich)} \quad = \quad 0 \\[1em]
\text{Prob (}\quad &\text{In } k \text{ Läufen kommt } k\text{–mal “Gleichheit”} \\
&\text{als Ausgabe, aber die Eingaben sind ungleich)} \quad \leq \quad (1 - \tfrac{1}{m})^k
\end{aligned}$$

D.h. selbst für $m = k$ haben wir im Falle von k–mal “Gleichheit” im schlimmsten Falle eine Fehlerwahrscheinlichkeit von $\frac{1}{e} > \frac{1}{3}$! Das ist schlecht.

Man wäre besser dran, die Felder direkt zu vergleichen.

Ein Problem bei dem beschriebenen Test ist, daß es auf verschiedenen Paaren von strings ganz verschiedene Fehlerwahrscheinlichkeiten hat: Auf strings, die sich in genau i Positionen unterscheiden ist die Fehlerwahrscheinlichkeit bei der Ausgabe "Gleichheit" gleich $\frac{m-i}{m} = 1 - \frac{i}{m}$. Wir werden für den nächsten Gleichheitstest die strings so transformieren, daß die Anzahl der verschiedenen Bits nicht mehr einen solchen Einfluß hat.

Nehmen wir einmal an, wir können T und P als Zahlen auffassen, die wir aber aus irgendwelchen Gründen nicht direkt in $O(1)$ sequentieller Zeit vergleichen. Unser Gleichheitstest geht dann folgendermaßen:

1. Wähle zufällig eine Primzahl $p \leq m^2$.
2. Berechne RestT = T mod p.
3. Berechne RestP = P mod p.
4. Wenn RestT = RestP melde Gleichheit, sonst Ungleichheit.

Die Anzahl und die Wahrscheinlichkeit der möglichen Rechnungen hängt jetzt von der Anzahl der Primzahlen $\leq m^2$ ab. Man weiß aus der Zahlentheorie:

Lemma 5.17 Sei $m > 17$ und ist $\pi(m) = (\#\text{ Primzahlen} \leq m)$. Dann gilt:

$$\frac{m}{\ln m} \leq \pi(m) \leq 1,2551 \cdot \frac{m}{\ln m}$$

(Dabei ist $\ln m = \log_e m$.) □

Wie verhält es sich hier mit den Fehlerwahrscheinlichkeiten?

$$\begin{aligned} \text{Prob (Ausgabe "Ungleichheit" und } T \text{ und } P \text{ gleich)} &= 0 \\ \text{Prob (Ausgabe "Gleichheit" und } T \text{ und } P \text{ ungleich)} &\leq ? \end{aligned}$$

Zur Ermittlung der Fehlerwahrscheinlichkeit einige Bezeichnungen:

$$\begin{aligned} T' &= T \bmod p = \text{Rest von } T \text{ dividiert durch } p. \\ P' &= P \bmod p = \text{Rest von } P \text{ dividiert durch } p. \end{aligned}$$

Es ist $T' = P'$ das Äquivalent zu $|T - P| \bmod p = 0$. Da $T, P \leq 2^m$ ist, ist $|T - P| \leq 2^m$ und es gilt sicherlich:

$$(\#\text{ Verschiedene Primteiler von } |T - P|) < m.$$

Wir veranschaulichen die Situation jetzt in folgender Tabelle:

$$
\begin{array}{ccccc}
 & p_1 & p_2 & \cdots & p_k \\
u_1 & u_1 \bmod p_1 & u_1 \bmod p_2 & \cdots & u_1 \bmod p_k \\
u_2 & u_2 \bmod p_1 & u_2 \bmod p_2 & \cdots & u_2 \bmod p_k \\
u_3 & & & & \\
\vdots & & & & \\
u_{2^m} & & & &
\end{array}
$$

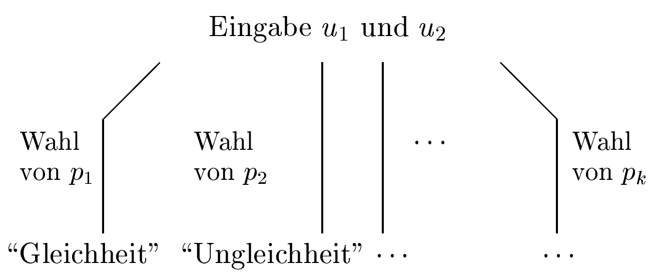
Dabei sind $u_1, ..., u_{2^m}$ die möglichen strings und $p_1, ..., p_k$ alle Primzahlen $\leq m^2$. Also ist $k \geq \frac{m^2}{2(\ln m)} = m \cdot \frac{m}{2(\ln m)}$. Wir betrachten einmal u_1 und u_2, die verschieden sind. Die $< m$ vielen verschiedenen Primteiler von $|u_1 - u_2|$ können unter Umständen alle unter den $p_1, ..., p_k$ zu finden sein (müssen es aber nicht). Genau für diese $< m$ vielen verschiedenen Primteiler p ist nun

$$u_1 \bmod p = u_2 \bmod p.$$

Für alle anderen Primzahlen q ist dagegen

$$u_1 \bmod q \neq u_2 \bmod q.$$

Unser Berechnungsbaum bei Eingabe von u_1 und u_2 sieht folgendermaßen aus



Da $u_1 \neq u_2$, steht an $< m$ vielen Positionen "Gleichheit". Die Wahrscheinlichkeit eine feste Position zu erreichen ist

$$\text{Prob(feste Position erreicht)} \;\leq \frac{1}{\frac{m^2}{2\ln m}}.$$

Die Wahrscheinlichkeit eine von $< m$ vielen festen Positionen zu erreichen ist

$$\text{Prob(eine von} < m \text{ festen Positionen wird erreicht)} \;\leq \frac{m}{\frac{m^2}{2\ln m}} = \frac{2\ln m}{m}.$$

Das geht in jedem Falle gegen 0. Durch konstant häufige Iteration wird die Fehlerwahrscheinlichkeit kleiner als jede Konstante. Und das für alle (!) m (nicht erst für m groß).

Im Falle unseres string matching können wir nun nicht so einfach mit T und P rechnen. Auch müssen wir die Wahrscheinlichkeit in dem ganzen Feld Match berücksichtigen. Die Sache ist also etwas komplizierter.
Die Funktion der Ausdrücke $u_i \bmod p_i$ ist die eines Fingerabdrucks. Hier brauchen wir etwas andere Fingerabdrücke.

Definition 5.18

(a) Ein Ring ist eine algebraische Struktur

$$R = (R, +, \cdot, 0, 1)$$

wobei R eine Menge ist, $+, \cdot$ 2–stellige Operationen auf R sind und $0, 1 \in R$ sind mit folgenden Eigenschaften:

1. $+, \cdot$ sind assoziativ.
2. $+$ ist kommutativ ($\cdot$ braucht nicht kommutativ zu sein).
3. $\cdot$ ist distributiv von beiden Seiten über $+$, d.h.

   $$a \cdot (b + c) = a \cdot b + a \cdot c \text{ und } (b + c) \cdot a = b \cdot a + c \cdot a$$

   für alle $a, b, c \in R$.
4. 0 ist Einselement bezüglich $+$, 1 ist Einselement bezüglich $\cdot$.
5. Jedes $a \in R$ hat ein Inverses bezüglich $+$, genannt $(-a)$, d.h. $a + (-a) = 0$.

Also ist $(R, +, 0)$ eine abelsche Gruppe und $(R, \cdot, 1)$ eine Halbgruppe mit 1 (auch Monoid genannt).

(b) Z ist der Ring der ganzen Zahlen. Ist $n \in N$, so ist

$$Z_n = \{0 + n \cdot Z, 1 + n \cdot Z, ..., (n-1) + n \cdot Z\}$$

die Menge der Restklassen modulo n. Man beachte ist $a = (i + n \cdot Z)$ mit $0 \leq i \leq n - 1$, so ist $a \bmod n = i$. Die Umkehrung gilt auch. Für $n \geq 1$ ist Z_n wieder ein Ring mit: Für $i, j < n$ ist

$$\begin{aligned} (i + n \cdot Z) + (j + n \cdot Z) &= ((i + j) \bmod n) + n \cdot Z \\ (i + n \cdot Z) \cdot (j + n \cdot Z) &= ((i \cdot j) \bmod n) + n \cdot Z \end{aligned}$$

(c) Ist R ein Ring so ist $R^{n \times n}$ der Ring der $n \times n$ Matrizen über R.

□

Um zu unserer Fingerabdruckfunktion zu kommen, beginnen wir mit der Abbildung

$$f : \{0,1\}^* \backslash \{e\} \to Z^{2\times 2}.$$

Wir setzen

$$f(0) = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \text{ und } f(1) = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

und für $u, v \neq e$ ist

$$f(u,v) = f(u) \cdot f(v).$$

Damit ist f eindeutig definiert. Setzen wir noch

$$f(e) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix},$$

so ist ein Monoidhomomorphismus zwischen $\{0,1\}^*$ und $Z^{2\times 2}$. (Diese Homomorphismuseigenschaft ist wichtig, um unserer Fingerabdruckfunktion effizient berechnen zu können.)

Beispiel 5.19

$$f(1011) = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 5 \\ 1 & 3 \end{pmatrix}$$

□

Den Beweis des folgenden Lemmas lassen wir zur Übung:

Lemma 5.20

(a) f ist injektiv.

(b) $\det f(X) = 1$ für jedes X.
Die Determinante ist gegeben durch:

$$\det \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = a_{11} \cdot a_{22} - a_{21} \cdot a_{12}.$$

(c) Jeder Eintrag in $S(X)$, wobei X die Länge m hat ist $\leq F_{m+1}$, der $(m+1)$ten Fibonacci Zahl.
(Es ist $F_1 = F_2 = 1$ und $F_{m+1} = F_m + F_{m-1}$.)

$\square$

In unserer Problemstellung haben wir angenommen, daß Zahlen der Länge $O(\log n)$ in einem Schritt manipuliert werden dürfen. Die Einträge in $f(X)$ sind aber auf jeden Fall größer, da die Folge der Fibonacci Zahlen von exponentiellem Wachstum ist. Wir rechnen stattdessen mit Resten!

Definition 5.21

(a) Ist p eine Primzahl und sei

$$f(X) = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}.$$

Dann ist

$$f_p(X) = \begin{pmatrix} a_{11} \bmod p & a_{12} \bmod p \\ a_{21} \bmod p & a_{22} \bmod p \end{pmatrix}.$$

Man beachte, daß $0 \leq a_{ij} \bmod p \leq p-1$.

(b) Die Menge F ist gegeben durch :

$$F = \{f_p | p \text{ Primzahl in } \{1, 2, ..., M\}\}.$$

Ist M polynomial in m, so ist die Länge der Einträge in $f_p(X) = O(\log m)$, wobei X die Länge m hat. Wir werden das M weiter unten noch genau festlegen.

$\square$

Beispiel 5.22 Ist $X = (1011)^4$, so ist

$$f(X) = \begin{pmatrix} 206 & 575 \\ 115 & 321 \end{pmatrix} \text{ und } f_7(X) = \begin{pmatrix} 3 & 1 \\ 3 & 6 \end{pmatrix}$$

$\square$

Damit haben wir eigentlich alles zusammen, um unseren string matching Algorithmus hinzuschreiben:

Definition 5.23 (Monte Carlo string matching)

Eingabe: $T[1:n]$ und $P[1:m]$, die Text und pattern darstellen. Eine ganze Zahl M. (Wir legen M weiter unten noch fest. Es ist dasselbe M wie aus Definition 5.21(b).)

Ausgabe: Match$(1:n-m+1)$ vom Typ bool.

```
begin
  1.for 1 ≤ i ≤ n − m + 1 pardo
      Match(i) := 0
  2.Wähle zufällig eine Primzahl in {1,...,M}.
    Berechne f_p(P).
    { Man beachte, daß P ein Feld ist. Die schnelle Berechnung
    von f_p(P) erfordert noch etwas Aufwand. }
  3.for 1 ≤ i ≤ n − m + 1 pardo
      L_i := f_p(T[i : i + m − 1])
      { Derselbe Kommentar wie oben gilt. }
  4.for 1 ≤ i ≤ n − m + 1 pardo
      if (L_i = f_p(P)) then Match(i) := 1
end
```

□

Zur Analyse dieses Algorithmus brauchen wir noch ein vorbereitendes Lemma aus der Zahlentheorie:

Lemma 5.24 Sei $n \leq 2^m$, dann gilt:

$$\#\text{ verschiedene Primteiler von } n \leq \pi(m),$$

sofern $m \geq 29$. □

Wir betrachten zunächst wieder den unserem pattern matching Algorithmus zu Grunde liegenden Gleichheitstest:

Eingabe: T und P Felder, beide der Länge m. Eine natürliche Zahl M.

Ausgabe: "Gleichheit" oder "Ungleichheit".

```
1.Wähle zufällig Primzahl aus {1,...,M}
2.Falls f_p(T) = f_p(P), dann Ausgabe "Gleichheit"
                         sonst Ausgabe "Ungleichheit"
```

Für den angegebenen Gleichheitstest gilt der folgende Satz:

Satz 5.25

$$\begin{aligned} \text{Prob(Ausgabe “Ungleichheit” wobei } T = P) &= 0 \\ \text{Prob(Ausgabe “Gleichheit” wobei } T \neq P) &\leq \tfrac{\pi(\lfloor 2.776 \cdot m \rfloor}{\pi(M)} \end{aligned}$$

(Wobei m die Länge von P und T ist.)
Beweis
Wir betrachten hier P und T direkt als strings (nicht als Felder). Wir nehmen an $T \neq P$. Es sei

$$f(T) = \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix} \text{ und } f(P) = \begin{pmatrix} b_1 & b_2 \\ b_3 & b_4 \end{pmatrix}.$$

Da f injektiv ist (Lemma 5.24) und $T \neq P$ gibt es einen Index $i \in \{1, 2, 3, 4\}$, so daß $a_i \neq b_i$. Wir beschränken die Wahrscheinlichkeit von $f_p(T) = f_p(P)$. Ist $f_p(T) = f_p(P)$, so gilt, daß

$$p|(a_i - b_i)|\text{für alle } i \in \{1, 2, 3, 4\}.$$

Also folgt: $p|(\Pi_{i=1}^4 |a_i - b_i|)$. Sei $n = \Pi_{i=1}^4 |a_i - b_i|$. Nach Lemma 5.24 ist $n \leq F_{m+1}^4 = 2^{4(\log F_{m+1})}$. Mit Lemma 5.24 ist:

$$(\#\text{ verschiedene Primteiler von } F_{m+1}^4) \leq \pi(\lfloor 4 \cdot \log F_{m+1} \rfloor),$$

sofern $\lfloor 4 \cdot \log F_{m+1} \rfloor \geq 29$. Also folgt für 2 strings T und P:

$$\text{Prob(Ausgabe “Gleichheit” und } T \neq P) \leq \frac{\pi(\lfloor 4 \cdot \log F_{m+1} \rfloor)}{\pi(M)}.$$

Nun ist $F_{m+1} \approx \frac{\Phi^{m+1}}{\sqrt{5}}$, wobei $\Phi = \frac{1+\sqrt{5}}{2}$, der goldene Schnitt ist. Da $\log_2 \Phi \approx 0.694$, folgt, daß obige Wahrscheinlichkeit durch

$$\frac{\pi(\lfloor 3.776 \cdot (m+1) \rfloor)}{\pi(M)}$$

abgeschätzt werden kann. $\square$

Folgerung 5.26 Ist $M = m^k$, dann gilt

$$\text{Prob(Ausgabe "Gleichheit" und } P \neq T) \leq \frac{3.48 \cdot k}{m^{k-1}}$$

Beweis
Nach Lemma 5.24 ist

$$\pi(\lfloor 2.776 \cdot m \rfloor) \leq 1.2551 \cdot \frac{2.776 \cdot m}{\ln(2.776 \cdot m)} \approx \frac{3.48 \cdot m}{\ln(2.776 \cdot m)}$$

und

$$\pi(m^k) \geq \frac{m^k}{k \cdot \ln m}.$$

Die Behauptung folgt mit dem letzten Satz. □

Wir erweitern unseren Gleichheitstest zum Test mehrere Paare von strings:

Eingabe: Eine Folge von t Paaren von strings (X_i, Y_i) mit $1 \leq i \leq t$, wobei X_i, Y_i alle die Länge m haben. Weiter eine ganze Zahl M.

Ausgabe: Ein Feld Matrix$[1 : t]$ vom Typ bool

```
begin
  1.Wähle Primzahl p ∈ {1, ..., M}
  2.for 1 ≤ i ≤ t pardo
      if f_p(X_i) = f_p(Y_i) then Match(i) = 1
                              else Match(i) = 0
end
```

Wir interessieren uns jetzt für eine Schranke für die Wahrscheinlichkeit, daß Match$(i) = 1$ ist aber $X_i \neq Y_i$ ist für irgendein i. Diese Schranke gibt uns der folgende Satz:

Satz 5.27

$$\text{Prob(Match}(i) = 1 \text{ aber } X_i \neq Y_i \text{ für irgendein } i)$$

$$\leq \frac{\pi(\lfloor 2.776 \cdot m \cdot t \rfloor)}{\pi(M)}$$

Der Beweis ergibt sich aus dem Beweis von Satz 5.25. □

Folgerung 5.28 Sei $k > 0$ eine konstante natürliche Zahl, sei $M = m \cdot t^k$. Mit diesem M gilt mit den Voraussetzungen des letzten Satzes:

$$\text{Prob}(\text{Match}(i) = 1 \text{ aber } X_i \neq Y_i \text{ für irgendein } i)$$
$$= O\left(\frac{1}{t^{k-1}}\right).$$

Beweis
Es gilt

$$\pi(\lfloor 2.776 \cdot m \cdot t \rfloor) \leq 1.2551 \cdot \frac{2.776 \cdot m \cdot t}{\ln(2.776 \cdot m \cdot t)} \approx \frac{3.48 \cdot m \cdot t}{\ln(2.776 \cdot m \cdot t)}$$

$$\pi(m \cdot t^k) \geq \frac{m \cdot t^k}{\ln(m \cdot t^k} = \frac{m \cdot t^k}{\ln m + k \cdot \ln t}.$$

Also

$$\frac{\pi(\lfloor 2.776 \cdot m \cdot t \rfloor)}{\pi(m \cdot t^k} \leq \frac{3.48 \cdot \cancel{m} \cdot t \cdot \ln(m \cdot t^k)}{\ln(2.776 \cdot \cancel{m} \cdot t) \cdot m \cdot t^k}$$
$$\leq \frac{3.48 \cdot k \cdot \ln(m \cdot t)}{t^{k-1} \cdot \ln(2.776 \cdot m \cdot t)} = O\left(\frac{1}{t^{k-1}}\right).$$

□

Wir müssen uns noch mit der schnellen parallelen Berechnung der Werte der Funktion f_p befassen.

Zunächst gilt, daß nicht nur f sondern auch f_p ein Homomorphismus ist. Also gilt:

$$f_p(X_1 \cdot X_2) = f_p(X_1) \cdot f_p(X_2)$$

für beliebige strings X_1 und X_2.

Wir geben einen effizienten parallelen Algorithmus an, um die $f_p(T[i : i + m - 1])$ zu berechnen. Dabei ist m die Länge des pattern.

Für i mit $1 \leq i \leq n$ sei

$$N_i = f_p(T[1 : i]).$$

Andererseits ist wegen der Homomorphieeigenschaft von f_p

$$N_i = f_p(T(1)) \cdot \ldots \cdot f_p(T(i)).$$

Wegen der Assoziativität der Matrixmultiplikation ist diese Berechnung eine Präfixsummenberechnung. Jede Matrix ist eine (2×2)–Matrix, deshalb kann man das Produkt zweier solcher Matrizen in sequentieller Zeit $O(1)$ berechnen. Alle N_i können in Zeit $T^{\infty}(n) = O(\log n)$ mit Arbeit $W(n) = O(n)$ berechnet werden.

Sei

$$G_p(0) = (f_p(0))^{-1} = \begin{pmatrix} 1 & 0 \\ p-1 & 1 \end{pmatrix}$$

und

$$G_p(1) = (f_p(1))^{-1} = \begin{pmatrix} 1 & p-1 \\ 0 & 1 \end{pmatrix}.$$

Dann ist

$$R_i = G_p(T(i)) \cdot G_p(T(i-1)) \cdot \ldots \cdot G_p(T(1))$$

wieder eine Präfixsumme mit $T^{\infty}(n) = O(\log n)$ und $W(n) = O(n)$. Es gilt:

$$f_p(T[i : i+m-1]) = R_{i-1} \cdot N_{i+m-1},$$

was sich in sequentieller Zeit $O(1)$ ermitteln läßt. Insgesamt lassen sich die $f_p(T[i : i+m-1])$ in $T^{\infty}(n) = O(\log n)$ und $W(n) = O(n)$ ermitteln.

Satz 5.29 Für unseren Algorithmus gilt:

$$\mathrm{Prob}(\mathrm{Match}(i) = 1 \text{ und } P[1:m] \neq T[i, i+m-1]) = O\left(\frac{1}{n^k}\right),$$

falls $M = m \cdot n^{k+1}$ für k eine Konstante ≥ 1. Der Algorithmus läuft in Zeit $T^{\infty}(n) = O(\log n)$ mit Arbeit $W(n) = O(n)$. (Dabei nehmen wir an, daß eine zufällige Primzahl in einem Schritt werden kann — eine etwas heikle Annahme.) □

Beispiel 5.30 Sei $m = 2^8$, $n = 2^{12}$, wähle $M = m \cdot n^2 = 2^{32}$. Dann wird jeder string der Länge 256 in eine 32–Bit Zahl abgebildet. Die Fehlerwahrscheinlichkeit ist $< \frac{1}{2^9}$. □

5.3 Randomisiertes Quicksort

Es gibt Schätzungen, nach denen in der Praxis bis zu 25 % aller Rechenzeit beim Sortieren verbraucht wird.

Definition 5.31 (Problem Sortieren)

Eingabe: Ein Feld von Elementen einer linearen Ordnung $A[1 : n]$. Die Elemente in A seien paarweise verschieden.

Ausgabe: Das Feld A in sortierter Reihenfolge, d.h. $A(1) < \cdots < A(n)$.

□

Wir lernen hier eine parallele Version des bekannten Quicksort kennen. Zunächst noch einmal das sequentielle Quicksort. Quicksort arbeitet auch wieder nach dem divide–and–conquer Prinzip.

Algorithmus 5.32 (Sequentielles randomisiertes Quicksort)

Eingabe: Ein Feld von Elementen einer linearen Ordnung $A[1 : n]$. Die Elemente in A seien paarweise verschieden.

Ausgabe: Das Feld A in sortierter Reihenfolge, d.h. $A(1) < \cdots < A(n)$.

```
begin
  1. if A enthält nur 1 Element
       then return A exit.
  2. Wähle zufällig ein Element S(A) von A.
       { S(A) steht für splitter von A. }
  3. S_1 := Folge der Elemente von A, die < S(A) sind.
  4. S_2 := Folge der Elemente von A, die > S(A) sind.
     { Beachte, wir haben angenommen, daß die Elemente aus A
     paarweise verschieden sind. }
  5. Besetze A mit S_1,S(A),S_2;
     k := die Position von S(A).
  6. Wende Quicksort rekursiv auf A[1:k] an;
     Wende Quicksort rekursiv auf A[k+1:m] an.
     {Beachte, daß A[1:k-1] und A[k+1:n] echt kleiner als A sind.}
end
```

□

Es gilt, daß Quicksort im worst–case die Laufzeit $O(n^2)$ hat, wogegen die mittlere Laufzeit $O(n \cdot \log n)$ ist. Wir schätzen dabei die Zeit, die in Schritt 2 bis Schritt 6 gebraucht wird (ohne die Zeit innerhalb der Rekursionen) als $O(\text{Länge von } A)$. (Man mache sich klar, was hier worst–case und mittlere Laufzeit bedeuten.) Beim parallelen Quicksort führen wir Schritt 2 und 3 in $O(1)$ paralleler Zeit aus, indem wir alle Elemente aus A mit $S(A)$ vergleichen. Außerdem lassen sich die rekursiven Aufrufe parallel abarbeiten.

Algorithmus 5.33 (Paralleles randomisiertes Quicksort)

Eingabe: Ein Feld A[1:n] paarweise verschiedener Elemente einer linearen Ordnung.

Ausgabe: Das Feld A sortiert.

```
begin
  1.if n ≤ 30 then sortiere A mit einem beliebigen
    Sortieralgorithmus (z.B. selection sort)
  2.Wähle zufällig ein Element S(A) aus A.
  3.for 1 ≤ i ≤ n pardo
      if A(i) < S(A) then Mark(i) := 1
                     else Mark(i) := 0
  4.Setze die Elemente A(i), für die Mark(i) = 1 ist
    an den Anfang von A. Setze dann S(A). Setze dahinter
    die Elemente von A mit Mark(i) = 0. Weiterhin setzen wir
    k := die Position von S(A).
  5.Sortiere rekursiv (und parallel!) die Teilfelder
    A[1:k-1] und A[k+1:n].
end
```

□

Folgendes Beispiel illustriert den Algorithmus:

Beispiel 5.34 Wir nehmen der Einfachheit halber an, daß die Rekursion erst bei $n = 1$ endet.

$$A = (4, 16, -5, 7, 25, -8, 1, 3)$$

Wählen $S(A) = 7$.
Wir transformieren A zu:

$$A = (4, -5, -8, 1, 3, \boxed{7}, 16, 25)$$

Ein Element ist eingerahmt, wenn es an seiner richtigen Stelle steht. Der splitter wird immer direkt an seine richtige Position getan! Wir wählen den splitter des linken Teils als -5, den des rechten Teils als 25.

$$A = (-8, \boxed{\text{-5}}, 4, 1, 3, \boxed{7}, 16, \boxed{25})$$

Wir wählen einen splitter nur noch in dem Teilfeld, das > 1 Element hat. Wählen wir jetzt also 3, so bekommen wir:

$$A = (\boxed{\text{-8}}\boxed{\text{-5}}, 1, \boxed{3}, 4, \boxed{7}\boxed{16}\boxed{25})$$

Damit sind wir fertig, da alle noch zu behandelnden Teilfelder die Länge 1 haben. □

Der wichtigste Punkt, der einer Analyse randomisierter Algorithmen vorausgehen muß ist der folgende: Man muß sich genau darüber klar werden, wie der Wahrscheinlichkeitsraum der Berechnungen für jede Eingabe aussieht. Dazu zunächst einige Beispiele.

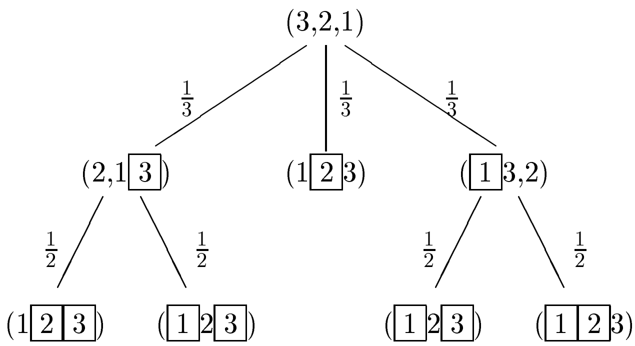
Ein ganz einfacher Fall: $A = (2, 1)$ (und wir nehmen an, die Rekursion hört bei $n = 1$ auf). Mit Wahrscheinlichkeit $\frac{1}{2}$ ist $S(A) = 2$ mit Wahrscheinlichkeit $\frac{1}{2}$ ist $S(A) = 1$. Im ersten Fall bekommen wir das Feld $A = (1\ \boxed{2})$, im zweiten $A = (\boxed{1}\ 2)$. Damit sind wir fertig. Für unser oben gegebenes A (und wir betonen noch einmal, daß wir für *jede* Eingabe einen Wahrscheinlichkeitsraum von Rechnungen bekommen) sind die Elementarereignisse dieses Wahrscheinlichkeitsraum Folgen von Feldinhalten, wobei die splitter gekennzeichnet sind.

$$\begin{array}{cc}
\text{1. Elementarereignis} & \text{2. Elementarereignis} \\
 & \\
A = (2,1) & A = (2,1) \\
| & | \\
A = (1\ \boxed{2}) & A = (\boxed{2}\ 1) \\
\text{Wahrscheinlichkeit } \frac{1}{2} & \text{Wahrscheinlichkeit } \frac{1}{2}
\end{array}$$

Das war zu leicht. Sei einmal $A = (3, 2, 1)$: Wir bekommen eine Menge von 6 Elementarereignissen:

$$\begin{array}{ccccc}
(3,2,1) & (3,2,1) & (3,2,1) & (3,2,1) & (3,2,1) \\
| & | & | & | & | \\
(2,1,\boxed{3}) & (2,1,\boxed{3}) & (1,\boxed{2},3) & (\boxed{1},3,2) & (\boxed{1}\,3,2) \\
| & | & & | & | \\
(1,\boxed{2}\boxed{3}) & (\boxed{1},2,\boxed{3}) & & (\boxed{1},2,\boxed{3}) & (\boxed{1}\boxed{2},3) \\
\frac{1}{3}\cdot\frac{1}{2}=\frac{1}{6} & \frac{1}{3}\cdot\frac{1}{2}=\frac{1}{6} & \frac{1}{3} & \frac{1}{3}\cdot\frac{1}{2}=\frac{1}{6} & \frac{1}{3}\cdot\frac{1}{2}=\frac{1}{6}
\end{array}$$

Die angegebenen Wahrscheinlichkeiten der Elementarereignisse addieren sich zu 1: Wir haben also tatsächlich einen Wahrscheinlichkeitsraum definiert. Wir können diesen ganzen Raum auch als 1 Baum darstellen:

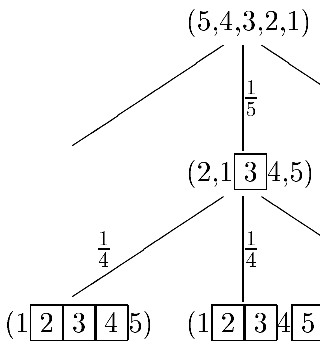


Folgende Punkte sind zu beachten:

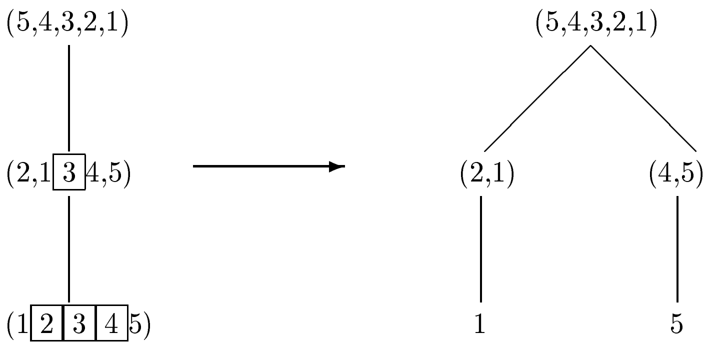
- Jeder Weg von der Wurzel zu einem Blatt stellt eine Berechnung des Algorithmus dar.
- Die Wahrscheinlichkeiten ergeben sich durch Multiplikation der Wahrscheinlichkeit an den Kanten.

Der oben angegebene Baum ist Beispiel einer sogenannten endlichen Markov–Kette, d.h. einfach ein endlicher Automat mit Übergangswahrscheinlichkeiten (aber ohne Eingabewert).

Noch ein Beispiel



Man beachte, daß man die einzelnen Rechnungen gemäß der Struktur der rekursiven Aufrufe auch wieder als Baum darstellen kann:


(5,4,3,2,1)
(2,1 3 4,5)
(1 2 3 4 5)
(5,4,3,2,1)
(2,1)
(4,5)
1
5

Sind unsere Wahrscheinlichkeitsräume von Berechnungen wirklich vernünftige Wahrscheinlichkeitsräume? Dazu:

Definition 5.35

(a) Sei A ein Feld. Die zu A gehörenden Zustände sind "Felder mit splittern" wie oben. Teilfelder eines Zustands sind die Felder zwischen den 2 splittern oder zwischen splitter und Rand von A.

(b) Für jeden Zustand Z bekommen wir den Wahrscheinlichkeitsraum der Zustandsübergänge von Z aus. Bei einem Zustandsübergang wählen wir in jedem Teilfeld einen splitter und passen die Teilfelder an. Die Verteilung ist die uniforme Verteilung. Wir nennen diesen Raum Ω_Z. Ist z.B. $Z = (2, 1, \boxed{3} 5, 4)$, so ist

$$\begin{array}{lll} \Omega_Z & = \{ & (Z, (1 \boxed{2}\boxed{3}\ 4\ \boxed{5})), \quad (Z, (\boxed{1}\ 2\ \boxed{3}\ 4\ \boxed{5})), \\ & & (Z, (\boxed{1}\ 2\ \boxed{3}\boxed{4} 5)), \quad (Z, (1 \boxed{2}\boxed{3}\boxed{4} 5)) \quad \}. \end{array}$$

Für $Z = (1, \boxed{2}\boxed{3}\boxed{4}, 5)$ ist

$$\Omega_Z = \{((1, \boxed{2}\boxed{3}\boxed{4}, 5), (1, \boxed{2}\boxed{3}\boxed{4}, 5)\}.$$

Wegen dem frühen Ende der Rekursion in unserem Algorithmus, bei Größe 30, brauchen wir unsere Ω_Z eigentlich nur für Zustände Z zu definieren, bei denen mindestens ein Teilfeld eine Größe > 30 hat.

(c) Wir definieren für alle Zustände Z die Familie von Wahrscheinlichkeitsräumen $(\Omega_Z^n)_{n \in N}$ durch:

$$\begin{array}{lll} \Omega_Z^1 & = & \Omega \\ \Omega_Z^{m+1} & = & \{(Z, Z_1, ..., Z_{m+1}) | (Z, ..., Z_m) \in \Omega_Z^m,\ (Z_n, Z_{m+1}) \in \Omega_{Z_m}^1\} \end{array}$$

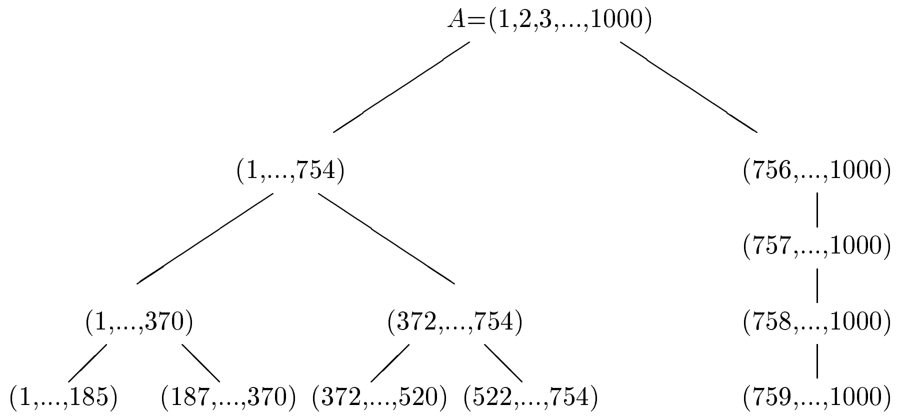
mit

$$\text{Prob}(Z, Z_1, ..., z_{n+1}) = \text{Prob}(Z, Z_1, ...Z_n) \cdot \text{Prob}(Z_n, Z_{n+1}).$$

Man überlegt sich induktiv über n, daß alle Ω_Z^n tatsächlich Wahrscheinlichkeitsräume sind. Man beachte weiterhin, daß Ω_Z^{n+1} nicht etwa ein kartesisches Produkt von Wahrscheinlichkeitsräumen ist.

□

Ein weiterer Punkt verdient Beachtung: Erzwingt der Algorithmus so wie er jetzt ist, daß alle rekursiven Aufrufe derselben Tiefe gleichzeitig ausgeführt werden? Man beachte, daß wir in unseren Berechnungen so getan haben. Ein Beispiel (wir zeichnen hier die Berechnung als Baum):



Die unterschiedliche Länge der Felder hat zur Folge, daß wir nicht alle rekursiven Aufrufe in gleicher Tiefe zur gleichen Zeit abarbeiten. In der folgenden Analyse nehmen wir an, daß die Aufrufe in jeder Tiefe des Baums synchron ausgeführt werden. Man beachte, daß sich die Laufzeit durch diese Annahme höchstens erhöht.

Satz 5.36 Für unseren Algorithmus 5.33 gilt:

(a) Er sortiert korrekt.

(b) $W(n) = O(n \cdot \log n)$ mit hoher Wahrscheinlichkeit.

(c) $T^{\infty}(n) = O((\log n)^2)$ mit hoher Wahrscheinlichkeit.

Beweis:

(a) Induktion über die Größe des Feldes ergibt die Behauptung.

(b),(c) Sei A ein Feld der Länge n. Dieses A bleibt für den ganzen folgenden Beweis fest. Sei die ZV X

$$X : \Omega_A^n \to N$$

mit

$$X((A, Z_1, ..., Z_n)) = \mathrm{Min}\left\{ i \;\middle|\; \begin{array}{l} \text{alle Teilfelder von } Z_i \\ \text{haben eine Größe} \leq 30 \end{array} \right\}$$

gegeben. Es gilt:

$$X(A, Z_1, ..., Z_n) + 1 = \begin{array}{l}\text{Maximum der ineinandergeschachtelten}\\ \text{rekursiven Aufrufe}\end{array}$$

Unser erstes Ziel ist es, zu zeigen, daß $X = O(\log n)$ mit hoher Wahrscheinlichkeit, d.h. es gibt eine Konstante C für die gilt,

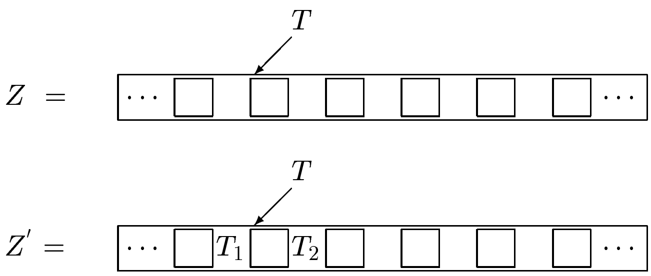
$$\text{Prob}(\{B | B \text{ Berechnung mit } A,\ X(B) \leq C \cdot \log n\}) \geq 1 - \left(\frac{1}{n}\right)^D.$$

Würden wir z.B. in jedem Teilfeld genau das mittlere Element erwischen, gälte diese Aussage in jedem Falle. Das ist natürlich nicht der Fall, aber es gilt: Ist T ein Teilfeld der Größe m von Z, dann ist die Wahrscheinlichkeit, daß T in 2 Teilfelder der Größe $\leq \frac{7}{8}m$ aufgespalten wird $\geq \frac{3}{4}$. D.h. etwas formaler:i Sind T und Z so, daß T ein Teilfeld von Z ist, dann gilt:

$$\text{Prob}\{(Z, Z') \mid T \text{ ist in 2 Teilfelder} \leq \tfrac{7}{8}m \text{ augespalten}\} \geq \frac{3}{4},$$

wobei wir die Wahrscheinlichkeit im Raum Ω_Z berechnen.

Das macht man sich folgendermaßen klar:



Für unseren Algorithmus 5.33 gilt:

$$T_1 \leq \tfrac{7}{8} \cdot m \quad \Leftrightarrow \quad |\{i \mid T(i) > S(T)\}| \leq \tfrac{1}{8} \cdot m$$

$$T_2 \leq \tfrac{7}{8} \cdot m \quad \Leftrightarrow \quad |\{i \mid T(i) < S(T)\}| \leq \tfrac{1}{8} \cdot m$$

Dann gilt:

$$\text{Prob}\left\{(Z, Z') \mid T_1 \geq \frac{7}{8} \cdot m \text{ oder } T_2 \geq \frac{7}{8} \cdot m\right\} \leq \frac{1}{4}$$

denn

$$\left(\left\lfloor\frac{1}{8}\cdot m\right\rfloor+\left\lfloor\frac{1}{8}\cdot m\right\rfloor\right)\cdot\frac{1}{m}\leq\frac{1}{4}.$$

Hinter dem nächsten verbirgt sich die folgende Intuition: Mit Wahrscheinlichkeit $\geq \frac{3}{4}$ verkleinert sich ein Teilfeld um mindestens den Faktor $\frac{7}{8}$. Ein Feld kann

$$\log_{\frac{8}{7}} n - \log_{\frac{8}{7}} 30 = \log_{\frac{8}{7}}\left(\frac{n}{30}\right) = O(\log n)$$

mal um einen Faktor $\leq \frac{7}{8}$ verkleinert werden, bis das verbleibende Feld eine Größe ≤ 30 hat. In dieses Argument müssen wir jetzt nur noch die Wahrscheinlichkeiten einarbeiten und auch, daß wir es in jedem Zustand mit mehreren Teilfeldern zu tun haben.

Wir treffen folgende Annahmen:

$$\log n = \log_{\frac{8}{7}} n$$

Sei a ein Element von A und Z ein Zustand. Wir sagen Das Teilfeld von Z, das a enthält, wird in dem Zustandsübergang $(Z, Z^{'})$ *gut geteilt.* $\Leftrightarrow$ Es gilt genau eine der 3 folgenden Aussagen:

(1) a ist schon in Z als splitter gewählt.

(2) Das Teilfeld, das a enthält hat eine Größe ≤ 30.

(3) Das Teilfeld T, das a enthält, ist in $Z^{'}$ in Teilfelder der Größe $\leq \frac{7}{8}\cdot$ Größe von T aufgeteilt.

Mit dieser Sprechweise gilt für jedes Element a von A:

$$\text{Prob}\left\{(Z,Z^{'})\;\middle|\;\begin{array}{l}\text{Das Teilfeld, das } a \text{ enthält, wird in}\\ \text{Übergang } (Z,Z^{'}) \text{ nicht gut geteilt}\end{array}\right\}\leq\frac{1}{4}$$

In Ω_Z^2 gilt:

$$\text{Prob}\left\{(Z,Z_1,Z_2)\;\middle|\;\begin{array}{l}\text{Das Teilfeld, das } a \text{ enthält, wird}\\ \text{mindestens einmal nicht gut geteilt}\end{array}\right\}\leq\frac{1}{4}+\frac{1}{4}-\left(\frac{1}{4}\right)^2$$

$$\text{Prob}\left\{(Z,Z_1,Z_2)\;\middle|\;\begin{array}{l}\text{Das Teilfeld, das } a \text{ enthält, wird}\\ \text{zweimal nicht gut geteilt}\end{array}\right\}\leq\frac{1}{16}.$$

In Ω_Z^3 gilt:

$$\mathrm{Prob}\left\{(Z,Z_1,Z_2,Z_3)\;\middle|\;\begin{array}{l}\text{Das Teilfeld, das } a \text{ enthält, wird}\\ \geq 2\text{–mal nicht gut geteilt}\end{array}\right\} \leq \begin{pmatrix}3\\2\end{pmatrix}\cdot\frac{1}{16}.$$

Hier wird die Boolesche Ungleichung aus Bemerkung 5.6(b) angewandt. Wir bekommen also im allgemeinen im Wahrscheinlichkeitsraum der Berechnungen von A: Sei a ein Element von A, dann gilt:

$$\mathrm{Prob}\left\{(Z,Z_1,...,Z_n)\;\middle|\;\begin{array}{l}\text{Das Teilfeld, das } a \text{ enthält, wird bis } Z_{20\log n}\\ \geq 19\log n\text{–mal nicht gut geteilt}\end{array}\right\}$$

$$\leq \begin{pmatrix}20\cdot\log n\\19\cdot\log n\end{pmatrix}\cdot\left(\frac{1}{4}\right)^{19\cdot\log n}.$$

Wobei hier $\log n = \lceil \log_{8/7} n\rceil$. Nun gilt:

$$\begin{aligned}
&\begin{pmatrix}20\cdot\log n\\19\cdot\log n\end{pmatrix}\cdot\left(\tfrac{1}{4}\right)^{19\cdot\log n}\\
\leq\;& \left(\tfrac{(20\cdot\log n)\cdot e}{19\cdot\log n}\cdot\left(\tfrac{1}{4}\right)\right)^{19\cdot\log n} && \text{,da } \begin{pmatrix}n\\k\end{pmatrix}\leq\left(\tfrac{n\cdot e}{k}\right)^k.\\
\leq\;& \left(\tfrac{5\cdot l}{19}\right)^{19\cdot\log n}\leq\left(\tfrac{15}{19}\right)^{19\cdot\log n}\\
\leq\;& \left(\tfrac{7}{8}\right)^{19\cdot\log n} && \text{,da } \tfrac{15}{19}\leq\tfrac{7}{8}\\
\leq\;& \left(\tfrac{1}{n}\right)^{19}.
\end{aligned}$$

D.h. also

$$\mathrm{Prob}\left\{(A,Z_1,...,Z_n)\;\middle|\;\begin{array}{l}\text{Das Teilfeld, das } a \text{ enthält, wird}\\ \text{bis } Z_{20\log n}\leq\log n\text{–mal gut geteilt}\end{array}\right\}\leq\left(\frac{1}{n}\right)^{19}$$

Damit folgt:

$$\mathrm{Prob}\left\{(A,Z_1,...,Z_n)\mid X(A,Z_1,...,Z_n)>20\cdot\log n\right\}$$

$$=\mathrm{Prob}\left\{(A,Z_1,...,Z_n)\;\middle|\;\begin{array}{l}\text{Es gibt ein Element } a \text{ aus } A\text{, so daß}\\ \text{das Teilfeld, das } a \text{ enthält bis } Z_{20\cdot\log n}\\ <\log n\text{–mal gut geteilt wird.}\end{array}\right\}$$

$$\leq n \cdot \left(\frac{1}{n}\right)^{19} = \left(\frac{1}{n}\right)^{18}$$

Auch hier haben wir wieder die Boolesche Ungleichung angewendet. Nun folgt die Behauptung leicht, denn: für jeden Zustandsübergang braucht Quicksort folgende Resourcen:

Schritt 1: $O(1)$ sequentielle Zeit.

Schritt 2: $O(1)$ sequentielle Zeit, $O(n)$ Arbeit.
Die Größe der Arbeit folgt, da wir ja in jedem Teilfeld gleichzeitig arbeiten.

Schritt 3: $O(1)$ parallele Zeit, $O(n)$ Arbeit.

Schritt 4: $O(\log n)$ Zeit, $O(n)$ Arbeit.
Man erinnere sich, daß die Präfixsummenbildung lineare Arbeit braucht.

Schritt 5: Wir zählen nur den Verwaltungsaufwand von $O(1)$ sequentieller Zeit.

Mit der angegebenen Wahrscheinlichkeit für die Anzahl der ineinandergeschachtelten Aufrufe folgt die Behauptung.

Daß die Rekursion schon bei einer Feldgröße von 30 endet, wird in dieser Analyse jedenfalls nicht gebraucht. Es kann sein, daß diese Beschränkung aus praktischen Gründen gemacht wird, da die Erzeugung von Zufallszahlen einen gewissen Aufwand benötigt. □

Welche PRAM brauchen wir für unseren Algorithmus? Wir haben simultanes Lesen des splitters. Dieses Lesen läßt sich in $O(\log n)$ Zeit mit $O(n)$ Arbeit vermeiden, indem man Kopien des splitters erzeugt. Eine EREW–PRAM reicht also.

Da das randomisierte Quicksort keine Fehler macht, nennt man es Las Vegas Algorithmus (im Unterschied zu den Monte Carlo Algorithmen). Bei dem randomisierten Quicksort kann höchstens die Laufzeit lang werden.

6 Bemerkungen zum Sinn der Sache

Zwei Kritikpunkte der vorgestellten Theorie sind:

(1) Welchen Sinn haben Algortihmen für parallele Rechner mit beliebig vielen Prozessoren?

(2) Die Modellierung von Parallelrechnern durch PRAMs ist grob unrealistisch, da die nötige Kommunikation zwischen den Prozessoren durch Zugriff der Prozessoren auf gleiche Variablen realisiert wird und somit keine zusätzliche Zeit erfordert. Dagegen braucht man für die Kommunikation zwischen den Prozessoren in realen Parallelrechnern (zB. Hypercube oder Gitter) sehr wohl eine gewisse Zeit.

Der zweite Punkt ist zunächst einmal kaum zu entkräften. Was reale Maschinen angeht, geben PRAM–Algorithmen nur eine gewisse Orientierung. (Aber es gibt Projekte, PRAMs zu bauen.)

Der erste Punkt ist dagegen zu entkräften: Zunächst sagt uns das WT–scheduling Prinzip (nach seinem Schöpfer auch Brents scheduling Prinzip genannt), daß ein PRAM–Algorithmus mit paralleler Zeit $T^\infty(n)$ und Arbeit $W(n)$ in Zeit

$$T^p(n) = O\left(\frac{W(n)}{p(n)} + T^\infty(n)\right)$$

auf natürliche Weise auf der PRAM mit $p = p(n)$ Prozessoren implementierbar ist. Weiter folgt: Ist $p(n) \geq c \cdot \frac{W(n)}{T^\infty(n)}$, so ist $T^{p(n)}(n) \leq d \cdot T^\infty(n)$ für eine Konstante d, die von c abhängt (und n groß genug). Ist dagegen $p(n) \leq c \cdot \frac{W(n)}{p(n)}$ für eine Konstante d, die von c abhängt (und n groß genug). Nehmen wir uns nun einmal $c - 1$ her und sei n genügend groß. D.h. die angegebenen Ungleichungen gelten für das n. Sei $p(n)$ klein genug, daß $p(n) \leq \frac{W(n)}{T^\infty(n)}$ und $2 \cdot p(n) \leq \frac{W(n)}{T^\infty(n)}$. Dann gilt, daß

$$T^p(n) \leq d \cdot \frac{W(n)}{p(n)} \quad \text{und} \quad T^{2 \cdot p}(n) \leq d \cdot \frac{W(n)}{2 \cdot p(n)},$$

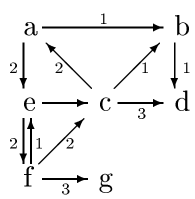
also: Bei Verdopplung der Prozessoren halbiert sich die Schranke an die Laufzeit. Mehr kann man eigentlich kaum erwarten.

Schließlich die letzte Frage: Man ist geneigt zu sagen: Gut, mit vielen Prozessoren sollten wir doch in der Lage sein, schwierige (insbesondere NP–vollständige Probleme) schneller zu lösen. Da man für NP–vollständige Probleme aber mit deterministischen Algorithmen exponentiell lang braucht, d.h. die Arbeit ist exponentiell und die Division $\frac{\text{exponentiell}}{\text{polynomial}}$ exponentiell bleibt, läßt sich hier mit einer polynomialen (also praktisch vertretbaren) Prozessorenzahl nichts Substantielles erreichen. Dagegen lassen sich, wie die ganzen Beispiele der Vorlesung zeigen, Probleme, die sowieso in Polynomialzeit lösbar sind unter Umständen mit polynomial vielen Prozessoren auf logarithmische oder zumindest polylogarithmische Zeit beschleunigen. Man beachte, daß das Verhältnis von exponentiell zu ploynomial wie das von polynomial zu logarithmisch ist.

Nicht für alle Probleme, die in Polynomialzeit lösbar sind, gilt, daß sie sich mit polynomial vielen Prozessoren auf polylogarithmische Zeit bringen lassen. Es gibt die P–vollständigen Probleme, die allem Anschein nach (aber nicht erwiesenermaßen) nicht in obigem Sinne parallelisierbar sind. Die Lösung solcher Probleme ist inhärent sequentiell. Ein Beispeil eines P–vollständigen Problems ist das folgende Problem der geordneten Tiefensuche:

Eingabe: Ein gerichteter Graph $G = (V, E)$ durch seine Adjazenzliste repräsentiert. D.h. wir haben für jeden Knoten $v \in V$ eine geordnete Liste Adj(v) der Nachbarn von v. Weiterhin ist ein Knoten s gegeben, der Startknoten.

Ausgabe: Einen Liste der Knoten von G in der Reihenfolge, die sich aus folgender Vorgehensweise ergibt: Zunächst besuchen wir s (d.h. schreiben s in unsere Liste und markieren s als "besucht" auf der Adjazenzliste). Wir machen rekursiv weiter: Haben wir einen Knoten v besucht, so besuchen wir den nächsten noch nicht besuchten Knoten auf der Adjazenzliste Adj(v). Falls ein solcher Knoten nicht existiert, so schauen wir nach dem letzten vor v besuchten Knoten. Wir gehen dann in der Adjazenzliste dieses Knotens zum Nachfolger von v und machen dort weiter. Ein Beispiel dazu ist:



Die Nummer an den gerichteten Kanten geben die Postion in der Adjazenzliste an, also etwa:

$$\mathrm{Adj}(a) \ = (b, e) \quad , \quad \mathrm{Adj}(c) \ = (b, a, d).$$

Die Ausgabe des Problems, bei Beginn mit Knoten a, ist:

$$(a,\ d,\ e,\ c,\ f,\ g).$$

Letzte Frage: Wie ist s bei solchen Graphen:

$$a \to b \to c \to \cdots \to g?$$