Das Lesen eines Registers M_j, auf dem simultan auch eine Schreiboperation ausgeführt wird, ist unkritisch – wir vereinbaren nämlich, dass in jedem synchronen Schritt alle *READ*s vor den *WRITE*s ausgeführt werden. Versuchen dagegen mehrere Prozessoren gleichzeitig in ein globales Register zu schreiben, muss eine Vereinbarung getroffen werden, wie sich dessen Inhalt verändert. Wir unterscheiden drei grundlegende PRAM-Modelle:

EREW PRAM (**E**xclusive **R**ead, **E**xclusive **W**rite):
Simultane *READ*s und *WRITE*s sind nicht erlaubt.

CRCW PRAM (**C**oncurrent **R**ead, **C**oncurrent **W**rite):
Simultanes Lesen ist uneingeschränkt erlaubt, simultanes Schreiben ist ebenfalls erlaubt.

Es gibt verschiedene Modelle von *CRCW PRAM*. Sie unterscheiden sich durch die Art, wie der Inhalt eines Registers nach einer simultanen Schreiboperation festgelegt wird:

CRCW$^{\text{com}}$ PRAM (**com**mon):
Ein gleichzeitiges Schreiben in ein Register M_j ist nur zulässig, wenn alle beteiligten Prozessoren versuchen denselben Wert zu schreiben.

CRCW$^{\text{arb}}$ PRAM (**arb**itrary):
Wenn einige Prozessoren versuchen, in ein Register zu schreiben, ist einer erfolgreich. Es ist aber nicht a priori festgelegt, welcher.

CRCW$^{\text{pri}}$ PRAM (**pri**ority):
Wenn einige Prozessoren versuchen, in ein Register zu schreiben, ist der mit dem kleinsten Index erfolgreich (Man sagt, dieser besitzt die höchste Priorität).

CREW PRAM (**C**oncurrent **R**ead, **E**xclusive **W**rite):
Gleichzeitiges Lesen erlaubt, jedoch nur exklusives Schreiben.

10.2.3 Komplexitätsmaße der PRAM

Definition 10.10 *Die uniforme (logarithmische)* Zeitkomplexität $time_M(x)$ *einer PRAM M auf eine Eingabe x ist entsprechend wie für die RAM in Definition 10.4 definiert, wobei die synchronen Schritte bis zur Termination gerechnet werden. Die Zeitkomplexität $T_M(n)$ von M ist wieder das Maximum aller $time_M(x)$, wobei das Maximum über alle Eingaben x der uniformen (logarithmischen) Größe n gerechnet wird.*

Anders als bei den Pseudokode-Programmen am Ende dieses Kapitels ist die Anzahl der benutzten Prozessoren unendlich, wobei meist nur endlich viele davon effektiv genutzt werden. Das sind diejenigen, die einen Schreibbefehl ausführen und dadurch das Ergebnis beeinflussen können. Die anderen Prozessoren lesen nur „still“ mit und werden bei einer Implementation nicht berücksichtigt. Deshalb definieren wir:

Definition 10.11 *Die* Prozessorkomplexität $proc_M(x)$ *einer PRAM M auf eine Eingabe x sei erklärt durch* $proc_M(x) = max\{i \mid i = 1 \text{ oder } P_i \text{ fuehrt auf } x \text{ ein } WRITE \text{ aus}\}$.

Die Prozessorkomplexität $P_M(n)$ *von* M *ist das Maximum aller* $proc_M(x)$*, wobei das Maximum über alle Eingaben („Probleminstanzen“) der uniformen (logarithmischen) Größe* n *gebildet wird,*

Für ein gegebenes algorithmisches Problem $\mathcal{P}$ ist es eine wichtige Aufgabe den besten Algorithmus für PRAM zu finden, der $\mathcal{P}$ lösen kann. Die Lösung hängt oft von der Anzahl der Prozessoren ab, die man benutzen kann. Z. B. benötigen alle sequentiellen Algorithmen um n Zahlen zu sortieren, mindestens $\Omega(n \log n)$ Zeit. Es gibt aber einen PRAM-Algorithmus, der dieses in $O(\log n)$ Zeit leistet. Dieser benötigt allerdings $O(n^2)$ Prozessoren.

Definition 10.12 *Zwei weitere wichtige Komplexitätsmaße für PRAMs sind das Prozessor-Zeit-Produkt für die Eingabe* x*:*

$$pt_M(x) = time_M(x) \cdot proc_M(x)$$

und Work, das die Gesamtzahl der Schritte aller Prozessoren, die während der Berechnung aktiv sind nach oben hin abschätzt. Letztere wird als

$$work_M(x)$$

bezeichnet. Die Operationenkomplexität $W_M(n)$ *ist dann wieder das Maximum aller* $work_M(x)$*, wobei das Maximum über alle Eingaben* x *der uniformen (logarithmischen) Größe* n *gerechnet wird.*

In der Regel ist das Prozessor-Zeit-Produkt viel größer als das exaktere, aber schwerer zu berechnende Work. Das Prozessor-Zeit-Produkt ist also eine (grobe) Abschätzung von Work nach oben.
Wir definieren wieder, dass eine PRAM für ein algorithmisches Problem $\mathcal{P}$ *optimal* ist, wenn $W_M(x) = O(T(n))$ gilt, wobei $T(n)$ die Zeitkomplexität des schnellsten RAM-Algorithmus für $\mathcal{P}$ ist.
Das Ziel beim Entwerfen von PRAM-Algorithmen ist es, für wichtige algorithmische Probleme optimale PRAM-Algorithmen zu konstruieren, deren Zeitkomplexität so klein wie möglich ist – am Besten (fast) konstant ($O(1)$).
Überraschenderweise gibt es für viele wichtige algorithmische Probleme deterministische oder probabilistische PRAM-Algorithmen, die optimal und sehr schnell sind – mit Zeitkomplexität $O(\log n)$, $O(\log \log n)$, $O(\log \log \log n)$, $O(\log^* n)$ und $O(1)$.
Versucht man Zeit durch Erhöhung der Parallelität zu sparen, so steht man vor dem *Problem der Parallelisierung*: Ist eine Verringerung der Laufzeit um ein beliebiges k möglich? Mit diesem Problem werden wir uns im Folgenden beschäftigen.

Satz 10.13 (Brent's scheduling principle) *Wenn man eine Berechnung auf* $m = max\ x_i$ *Prozessoren in Zeit* t *ausführen kann, wobei* x_i *die Anzahl der Operationen im* i*-ten Schritt ist, dann kann man auf*

$$p < m$$

Prozessoren dieselbe Berechnung in der Zeit

$$t + \lfloor \frac{x}{p} \rfloor$$

ausführen, wobei $x = \sum x_i$

Beweis:

$$\text{Mit } \left\lceil \frac{x_i}{p} \right\rceil \leq \frac{x_i}{p} + 1 \text{ gilt } \sum_{i=1}^{t} \left\lceil \frac{x_i}{p} \right\rceil = \left\lfloor \sum_{i=1}^{t} \left\lceil \frac{x_i}{p} \right\rceil \right\rfloor \leq \left\lfloor \sum_{i=1}^{t} (\frac{x_i}{p} + 1) \right\rfloor = t + \left\lfloor \frac{x}{p} \right\rfloor$$

□

Korollar 10.14 *Wenn man eine Berechnung, die* $O(n)$ *sequentielle Zeit braucht, auf* n *Prozessoren in* $O(\log n)$ *Zeit ausführen kann, dann kann man diese Berechnung auf* $O\left(\frac{n}{\log n}\right)$ *Prozessoren in Zeit* $O(\log n)$ *ausführen – d. h. in* optimaler Zeit: $O(n) = O(\log n) \cdot O\left(\frac{n}{\log n}\right)$.

Beweis: Man wähle $x = O(n),\ t = O(\log n),\ p = O\left(\frac{n}{\log n}\right)$. □

Wenn die Zuordung von Daten zu Prozessoren kein Problem ist, dann kann man dieses Korollar auf PRAMs anwenden.

10.2.4 Beispiele für PRAM-Algorithmen

Um die Algorithmen zu beschreiben, benutzen wir den parallelen Ausdruck

$$\underline{\text{par}}\ [a \leq i \leq b]\ S_i$$

um zu beschreiben, dass die Anweisungen S_i für alle i mit $a \leq i \leq b$ parallel ausgeführt werden sollen.

Beispiel 10.15 Maximum finden *Gegeben:* $n = 2^m$ natürliche Zahlen $a_n, a_{n+1}, \ldots, a_{2n-1}$. Gesucht wird ihr Maximum.
Algorithmus (für EREW-PRAM):

```
for l ← m − 1 down to 0 do
    par [2^l ≤ j ≤ 2^(l+1) − 1] a_j ← max{a_2j, a_2j+1}
```

Erklärung:
Der Algorithmus benötigt $O(m) = O(\log n)$ Zeit und $\frac{n}{2}$ Prozessoren. Die Berechnung kann man für $m = 3$ durch den Berechnungsbaum von Abbildung 10.9 darstellen.
Nun genügt es, Brent's principle zu benutzen, um einen optimalen Algorithmus zu erhalten.

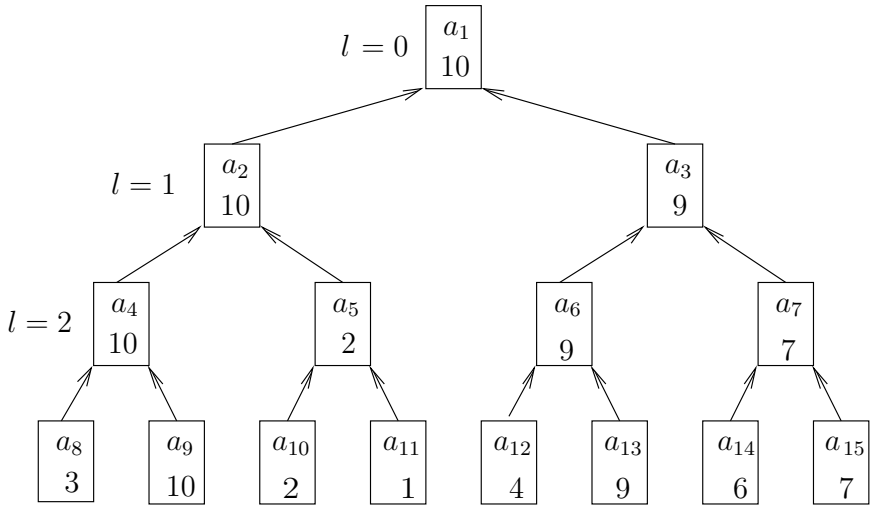


Abbildung 10.9: Maximum finden durch die PRAM

Beispiel 10.16 Maximum finden Der vorhergehende parallele Algorithmus scheint zeitoptimal zu sein. Das hieße, dass das Maximum nicht schneller als in $O(\log n)$ Zeit berechnet werden kann. Das folgende Beispiel zeigt, dass solches „common sense reasoning“ falsch sein kann. Dieses Beispiel zeigt außerdem die Mächtigkeit des parallelen Schreibens.

Folgender Algorithmus (für CRCW$^{\text{arb}}$, CRCW$^{\text{com}}$, oder CRCW$^{\text{pri}}$–PRAM) kann mit n^2 Prozessoren das Maximum (Minimum) von n Zahlen in der Zeit $O(1)$ finden.

Prozessoren: $P_{ij},\ 1 \leq i, j \leq n$;

Input: $M(0) \leftarrow n$, $M(1) \leftarrow x_1, \ldots, M(n) \leftarrow x_n$. Die Register des globalen Speichers erhalten die Zahlen x_i, unter denen das Maximum gefunden werden soll;

Arbeitsspeicher: $Y(1), \ldots, Y(n)$ – Register des globalen Speichers;

Output: Maximum liegt in $M(0)$.

Algorithmus für den Prozessor P_{ij}:

1. $Y(i) \leftarrow 0$;
2. if $M(i) < M(j)$ then $Y(i) \leftarrow 1$;
3. if $Y(i) = 0$ then $M(0) \leftarrow M(i)$

Man beachte, daß in allen parallelen Schreibanweisungen der gleiche Wert geschrieben wird, und daher der *common-write* Modus nicht wirklich ausgenutzt wird!

Beispiel 10.17 Berechnung der ODER-Funktion auf EREW-PRAM Der folgende Algorithmus berechnet die Funktion

$$\text{ODER: } \{0,1\}^n \to \{0,1\},\ OR(x_1, \ldots, x_n) = \text{ if any } x_i = 1 \text{ then } 1 \text{ else } 0.$$

Input: $M(1), \ldots, M(n)$ – Register des globalen Speichers
Output: $M(1)$
Prozessoren: $P_1, \ldots, P_n$
Arbeitsspeicher: Y_i – jeweils lokales Register von P_i
Algorithmus für den Prozessor P_i:

```
begin t ← 0;
   Y_i ← M(i);
   while i + 2^t ≤ n
      do Y_i ← Y_i ∨ M(i + 2^t)
         M(i) ← Y_i;
         t ← t + 1
      endwhile
end
```

Analyse:
Zeitkomplexität: $\lceil \log n \rceil + 1$, Prozessorkomplexität: n

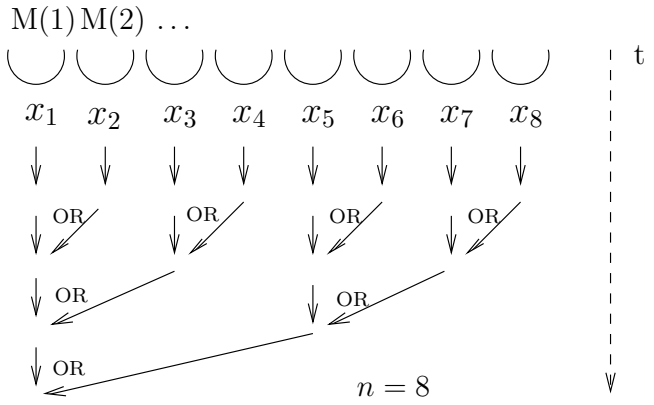


Abbildung 10.10: ODER-Berechnung durch die PRAM

Benutzen wir Brent's principle, bekommen wir einen optimalen Algorithmus mit $O(\log n)$ Zeit und $O\left(\frac{n}{\log n}\right)$ Prozessoren.

Beispiel 10.18 Präfixsumme Gegeben seien $n = 2^m$ Zahlen $a_n, a_{n+1}, \ldots a_{2n-1}$. Zu berechnen sind alle partiellen Summen $b_{n+j} = \sum_{k=n}^{n+j} a_k = a_n + \ldots + a_{n+j}$,mit $j = 0, \ldots, n-1$. Wir zeigen, dass es für dieses Problem einen $O(\log n)$-Algorithmus für CREW-PRAM gibt. Dieser Algorithmus wird sehr oft benutzt, um schnelle Algorithmen für PRAM zu erstellen.
Algorithmus für CREW-PRAM:

```
for l ← m − 1 downto 0 do
   par [2^l ≤ j < 2^(l+1)] a_j ← a_2j + a_2j+1;
```

$b_1 \leftarrow a_1$;
for $l \leftarrow 1$ to m do
  par $[2^l \leq j < 2^{l+1}]$
    $b_j \leftarrow$ if j odd then $b_{(j-1)/2}$ else $b_{j/2} - a_{j+1}$

Dieser Algorithmus braucht $O(\log n)$ Zeit und $O(n)$ Prozessoren. Mit Hilfe von Brent's Prinzip läßt sich ein optimaler Algorithmus finden.

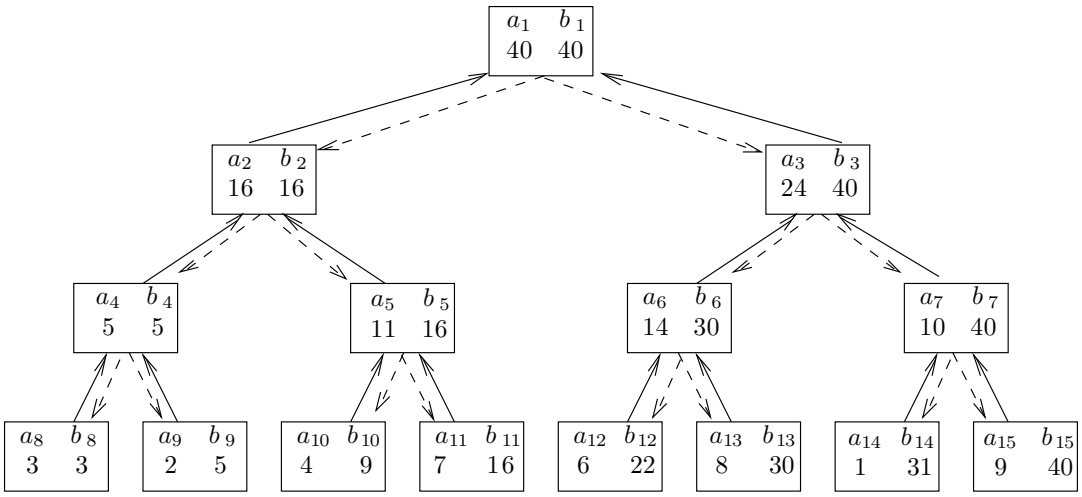


Abbildung 10.11: Präfixsummenberechnung durch die PRAM

Erste Phase (erste for-Schleife, durchgezogene Pfeile): Berechnung von a_i , $1 \leq i \leq n-1$.
Zweite Phase (zweite for-Schleife, unterbrochene Pfeile): Jeder Vater schickt seinen b-Wert seinen beiden Kindern. Das rechte Kind behält diesen als seinen b-Wert. Das linke Kind subtrahiert von diesem b-Wert den a-Wert des rechten Bruders.

10.2.5 PRAM-Hauptkomplexitätsklassen

Sei $\mathcal{M}$ eine beliebige PRAM-Variante. Wir bezeichnen mit

$$\mathcal{M}TimeProc(T, P)$$

die Menge der Funktionen bzw. Sprachen, die von einer T-Zeit– und P-Prozessorbeschränkten PRAM vom Typ $\mathcal{M}$ berechnet werden können.
Zwischen Prozessor- und Zeitkomplexität bei PRAMs besteht die folgende Beziehung:

Satz 10.19 *Seien $T, P : \mathbb{N} \to \mathbb{N}$ Funktionen und $k : \mathbb{N} \to \mathbb{N}$ eine rekursive Funktion, die auf RAM_+ in der Zeit T berechenbar ist. Für eine feste Konstante c gilt:*

$$\mathcal{M}TimeProc(T, P) \subsetneq \mathcal{M}TimeProc\left(c \cdot k \cdot T, \left\lceil \frac{P}{k} \right\rceil\right)$$

Beweis: Um eine P-Prozessorbeschränkte PRAM M mit $\lceil \frac{P}{k} \rceil$ Prozessoren zu simulieren, konstruieren wir eine PRAM M' mit P' Prozessoren, die folgendermaßen arbeitet:
P'_1 berechnet auf Eingabe n zunächst $k(n)$. Anschließend simuliert jeder Prozessor P'_i die Prozessoren von M, die die Prozessorkennung (PID) $(i-1)k(n)+1, \ldots, ik(n)$ haben. Jeder synchrone Schritt von M wird in $k(n)$ Phasen simuliert, in denen nacheinander für jeden korrespondierenden Prozessor sein entsprechender Schritt nachgeahmt wird. Dazu unterteilt P'_i seinen lokalen Speicher, um die lokalen Daten der zu simulierenden Prozessoren speichern zu können. Zur Simulation eines Schrittes eines Prozessors P_j lädt P'_i die zugehörigen Operanden in seine Operationsregister.
Bei CRCW-PRAMs muss dabei sichergestellt werden, dass die alten Inhalte der Register während der gesamten Simulation eines Schrittes zur Verfügung stehen und im Fall der Prioritätsregel beim simultanen Schreiben der Prozessor mit höchster Priorität am Ende den neuen Inhalt bestimmt.
Zu diesem Zweck kann man jedes Register durch ein Tripel von Registen ersetzen, mit denen der alte Inhalt, der momentane neue Inhalt und der Prozessor, der bislang die höchste Priorität besaß, dargestellt werden.
Zum Lesen wird das erste Register eines Tripels mit dem alten Inhalt benutzt. Vor einem Schreiben in CRCW^{pri}-PRAM wird überprüft, ob der momentan simulierte Prozessor eine höhere Priorität besitzt, und nur in diesem Fall geschrieben. □

Als ein Korollar bekommen wir:

Korollar 10.20
$CRCW_+\ TimeProc(\text{POL},\ \text{POL}) = \mathcal{P}$ *(Polynomialzeit)*

Beweis: Für Polynome P, T und $k = P$ ergibt sich aus dem vorigen Satz:

$$\begin{aligned} CRCW_+\ TimeProc(T, P) &\subseteq CRCW_+\ TimeProc(O(P \cdot T), 1) \\ &= RAM_+ - Time(O(T \cdot P)) \subseteq \mathcal{P} \end{aligned}$$

□

Das bedeutet, dass die PRAMs, die nur polynomiell viele Prozessoren benutzen und nur in Polynomialzeit arbeiten, nicht mächtiger sind als die sequentiellen RAMs, die auch in Polynomialzeit arbeiten!
Wenn wir die Prozessoranzahl aber exponentiell beschränken, dann erhalten wir eine größere Klasse. Wir definieren dazu:

$$PRAM - Time(t(n)) := CRCW_+\ TimeProc(t(n), 2^{t(n)})$$

Satz 10.21
$PRAM - Time(POL) = \mathcal{PSPACE}$

Dies liegt zum einen daran, dass man jede TM, die mit der konstruierbaren Platzkomplexität $s(n) \geq \log(n)$ arbeitet, durch eine PRAM mit $O(s)$ Prozessoren in der Zeit $O(s)$ simulieren kann:

$$PSpace(s(n)) \subseteq PRAM - Time(O(s(n)))$$

Zum anderen kann man jede PRAM, die mit der konstruierbaren Zeitkomplexität $t(n) \geq \log(n)$ und mit $2^{t(n)}$ Prozessoren arbeitet, auf einer TM simulieren, die $O(t^2(n))$ Platz benötigt:

$$PRAM - Time(t(n)) \subseteq PSpace(t^2(n))$$

Der vorhergehende Satz besagt, dass Polynomialzeit für PRAMs genauso mächtig ist, wie Polynomialplatz für RAMs!
Dasselbe gilt auch für andere Modelle paralleler Computer, z.B. auch für APM. Diese Resultate sprechen für folgende These auf der die moderne parallelkomplexitätstheorie begründet ist.

Parallel Computation Thesis *There exists a standard class of (parallel) machine models, which includes among others all variants of PRAM machines, and also APM, for which polynomial time is as powerful as polynomial space for sequencial machines (from the first machine class).*

Die Maschinen-Modelle, die dieser These genügen, bilden die sogenannte *zweite Maschinenklasse*.
Es scheint, dass $PSPACE$ viel mächtiger ist als $\mathcal{P}$, aber niemand hat das bisher bewiesen. Deshalb ist die Frage

$$\mathcal{P} \stackrel{?}{=} \mathcal{PSPACE}$$

eines der wichtigsten und bedeutendsten Probleme der Informatik.

10.2.6 Grenzen der Parallelität

Natürliche Frage: Gibt es ein natürliches und einfaches Problem, für das es keinen parallelen Algorithmus gibt, der schneller ist als der schnellste sequentielle Algorithmus?

Satz 10.22 (Kung) *Kein paralleler Algorithmus, der die Operationen $+, -, *, \div$ benutzt, kann ein Polynom n-ten Grades schneller als in $\log n$ Schritten berechnen.*

Korollar 10.23 *Kein paralleler Algorithmus kann x^n schneller als ein sequentieller Algorithmus berechnen.*

10.3 Paralleles Suchen und optimales Mischen

Während die bisher behandelten parallelen Algorithmen relativ einfach waren und mehr der Erläuterung des Maschinenmodells dienten, wird in den beiden letzten Abschnitten dieses Kapitels gezeigt, wie ein prominentes Problem, nämlich das Sortieren, nach nicht elementaren Überlegungen mit parallelen Algorithmen exponentiell beschleunigt werden kann. Dabei ist die Anzahl der Prozessoren immer von der Ordnung $O(n)$. Zunächst werden Algorithmen zum „Mischen“ entsprechend Tabelle 10.4 schrittweise verbessert, um dann als Prozedur für das parallele Sortieren in Algorithmus 10.4 eingesetzt zu werden.
Anders als bisher benutzen wir einen Pseudokode, der aber bei Bedarf leicht auf eine PRAM übertragen werden kann. So gehen auch oft Lehrbücher zu parallelen Algorithmen vor [Jáj92], [Rei93], [Cha92] und [GS93]. Die folgenden Algorithmen sind in [Jáj92] ausführlicher beschrieben. (Wie üblich bezeichnet $|M|$ die Mächtigkeit einer Menge M.) **pardo** ist eine Anweisung zur (synchron-)parallelen Ausführung.

	Zeit $T_A(n)$	Operationen $W_A(n)$	optimal ?
Mischen 1: Satz 10.27	$O(log\ n)$	$O(n)$	ja
Mischen 2: Korollar 10.31	$O(log\ log\ n)$	$O(n\ log\ log\ n)$	nein
Mischen 3: Satz 10.32	$O(log\ log\ n)$	$O(n)$	ja
Sortieren: Satz 10.33	$O(log\ n\ log\ log\ n)$	$O(n\ log\ n)$	ja

Tabelle 10.4: Mischen und Sortieren

Definition 10.24 *Gegeben sei eine lineare Ordnung $(S, \leq)$ (Def. 5.2) und zwei Teilmengen $\tilde{A}, \tilde{B} \subseteq S$ mit $|\tilde{A}| = |\tilde{B}| = n > 0$, zu denen wir die Folgen: $A = (a_1, a_2, \ldots, a_n)$ und $B = (b_1, b_2, \ldots, b_n)$, mit $i < j \Rightarrow a_i \leq a_j \quad \wedge \quad b_i \leq b_j \quad (1 \leq j, j \leq n)$ benutzen. Die* Mischung *(merge) von A und B ist die Folge $C = (c_1, c_2, \ldots, c_{2n})$ mit $i < j \rightarrow c_i \leq c_j$, die A und B als disjunkte Teilfolgen enthält.*

Beispiel: $(2,4,4) \quad (3,4,7)$
$\searrow \quad \swarrow$
$(2,3,4,4,4,7)$

Definition 10.25 *Sei $X = (x_1, x_2, \ldots, x_t)$ eine Folge mit Elementen aus der linear geordneten Menge S. Für $x \in S$ ist der* Rang *von x in X definiert als:*
$rank(x : X) := |\{i | i \in \{1, \ldots, t\} \wedge x_i < x\}|$. Für eine weitere solche Folge
$Y = (y_1, \ldots, y_s)$ sei $rank(Y : X) := (r_1, r_2, \ldots, r_s)$ mit $r_i = rank(y_i : X)$

Beispiel: Für $X = (25, -13, 26, 31, 54, 7)$ und $Y = (13, 27, -27)$ ist $rank(Y : X) = (2, 4, 0)$.
Zur Vereinfachung nehmen wir jetzt $A \cap B = \emptyset$ an. Das Problem, die Folgen A und B zu mischen, wird dann zu:

$$\text{Bestimme für jedes } x \in A \cup B : i := rank(x : A \cup B)$$
$$\text{(dann ist } x \text{ das } (i+1)\text{-te Element } c_i \text{ in C)}$$

Wegen $rank(x : A \cup B) = rank(x : A) + rank(x : B)$ genügt uns ein Algorithmus für:

$$\text{Berechne } rank(B : A)$$

Für $b_i \in B$ kann durch binäres Suchen j_i mit $a_{j_i} < b_i < a_{j_i+1}$ in $O(\log n)$ (sequentieller) Zeit berechnet werden, d.h. $j_i = rank(b_i : A)$. Dieses Verfahren kann man parallel auf alle Elemente von B anwenden. D.h. wir erhalten einen parallelen Algorithmus mit $O(\log n)$ (paralleler) Zeit und $O(n \log n)$ Operationen.
Da es sequentielle Algorithmen mit linearer Zeitkomplexität gibt (d.h. $O(n)$), ist der parallele Algorithmus *nicht optimal*. Es soll daher jetzt ein optimaler, paralleler Algorithmus entwickelt werden. Als Hilfsalgorithmus bildet der folgende parallele Algorithmus Blöcke A_i und B_i von A und B:

A:	A_0	A_1	...	A_{k_m-1}
B:	B_0	B_1	...	B_{k_m-1}

wobei $A = (a_1, a_2, \ldots, a_n)$, $B = (b_1, b_2, \ldots, b_m)$ und $k_m = \frac{m}{\log m} \in \mathbb{N}$. Außerdem sollen alle $x \in A_i \cup B_i$ größer als die Elemente in $A_{i-1} \cup B_{i-1}$ sein. (Im Algorithmus 10.1 wird k_m als $k(m)$, a_{j_i+1} als $a_{j(i)+1}$ usw. geschrieben.)

Algorithmus 10.1 (Partitionieren)

Eingabe: Zwei Felder $A = (a_1, \ldots, a_n)$ und $B = (b_1, \ldots, b_m)$ in aufsteigender Reihenfolge sortiert, wobei $\log m$ und $k(m) = \frac{m}{\log m}$ ganze Zahlen sein müssen.
Ausgabe: $k(m)$ Paare (A_i, B_i) von Teilfolgen von A und B, so dass
(1) $|B_i| = \log m$
(2) $\sum_i |A_i| = n$ und
(3) jedes Element von A_i und B_i ist größer als jedes Element von A_{i-1} oder B_{i-1} für alle $1 \le i \le k(m) - 1$.

begin
1. Set $j(0) := 0, j(k(m)) := n$
2. **for** $1 \le i \le k(m) - 1$ **pardo**
   2.1 Berechne $rank(b_{i \log m} : A)$ durch binäre Suche und setze $j(i) = rank(b_{i \log m} : A)$
3. **for** $0 \le i \le k(m) - 1$ **pardo**
   3.1 $B_i := (b_{i \log m + 1}, \ldots, b_{(i+1) \log m}$;
   3.2 $A_i := (a_{j(i)+1}, \ldots, a_{j(i+1)})$
   (A_i ist leer, wenn $j(i) = j(i+1)$ gilt.)

end

Beispiel: (zum Algorithmus 10.1) Gegeben seien die folgenden zwei Felder A und B mit $m = 4$ und $k_m := k(m) = \frac{m}{\log m} = 2$

$$\begin{array}{rcccccccccccc} & a_1 & & \ldots & & & a_8 & & b_1 & \ldots & b_4 \\ \text{A}=(4, & 6, & 7, & 10, & 12, & 15, & 18, & 20) & \text{B}=(3, & 9, & 16, & 21) \end{array}$$

Dann erhält man folgende Teilmengen: $A_0 = (4, 6, 7)$, $A_1 = (10, 12, 15, 18, 20)$, $B_0 = (3, 9)$ und $B_1 = (16, 21)$.

Das folgende Lemma gibt die Zeitkomplexität und Anzahl der Operationen an.

Lemma 10.26 *Sei C die sortierte Folge, welche man durch Mischen der sortierten Folgen A und B mit den Längen n bzw. m erhält. Dann teilt der Algorithmus 10.1 A und B in Paare von Teilfolgen (A_i, B_i) auf, so dass $|B_i| = O(\log m), \sum_i |A_i| = n$ und $C = (C_0, C_1, \dots)$, wobei C_i die sortierte Folge ist, welche aus A_i und B_i durch Mischen entsteht. Dieser Algorithmus benötigt in $O(\log n)$ Zeit mit insgesamt $O(n+m)$ Operationen.*

Beweis: Wir zeigen zuerst, dass jedes Element in den Teilfolgen A_i und B_i größer ist, als jedes Element von A_{i-1} oder B_{i-1}. Die zwei kleinsten Elemente von A_i und B_i sind $a_{j(i)+1}$ und $b_{i \log m+1}$, während die größten Elemente von A_{i-1} und B_{i-1}, $a_{j(i)}$ und $b_{i \log m}$ sind. Da $rank(b_{i \log m} : A) = j(i)$, erhalten wir $a_{j(i)} < b_{i \log m} < a_{j(i)+1}$. Dies impliziert, dass $b_{i \log m+1} > b_{i \log m} > a_{j(i)}$ und $a_{j(i)+1} > b_{i \log m}$ ist. Daher ist jedes Element von A_i und B_i größer als jedes Element von A_{i-1} oder B_{i-1}. Die Korrektheit des Algorithmus folgt hieraus unmittelbar.
Zeitanalyse: Der 1. Schritt braucht $O(1)$ sequentielle Zeit. Schritt 2 benötigt $O(\log n)$ Zeit, da die binäre Suche auf alle Elemente parallel angewendet wird. Die Gesamtzahl der für diesen Schritt auszuführenden Operationen ist $O\left((\log n) \times (\frac{m}{\log m})\right) = O(m + n)$, da $\left(\frac{m \log n}{\log m}\right) < \left(\frac{m \log(n+m)}{\log m}\right) \leq n + m$ für $n, m \geq 4$. Der 3. Schritt benötigt $O(1)$ paralleler Zeit, bei Benutzung einer linearen Anzahl von Operationen. Daher läuft dieser Algorithmus in $O(\log n)$ Zeit mit insgesamt $O(n + m)$ Operationen. □

Mit diesem Algorithmus haben wir das Mischproblem der Größe n auf Unterprobleme kleinerer Größe reduziert.
Optimaler Algorithmus für das Mischen:

- wende Algorithmus 10.1 an
- behandle die Paare (A_i, B_i) getrennt und parallel (es gilt $|B_i| = \log n$)
- falls $|A_i| \leq c \log n$ mische (A_i, B_i) in $O(\log n)$ Zeit mit einem sequentiellen Mischalgorithmus
- falls $|A_i| \not\leq c \log n$, dann wende Algorithmus 10.1 an, um A_i in Blöcke der Länge $\log n$ zu zerlegen. Dazu werden $O(\log \log n)$ Zeit und $O(|A_i|)$ Operationen benötigt.

Insgesamt ergibt sich:

Satz 10.27 *Das Mischen zweier Folgen A und B der Länge n ist von einem parallelen Algorithmus in $O(\log n)$ Zeit mit $O(n)$ Operationen durchführbar.*

Methode des Algorithmus: aufteilen (partitioning)
zu unterscheiden von: teile und herrsche (divide-and-conquer)
Um zu einem schnelleren Algorithmus zu kommen, betrachten wir paralleles Suchen.

- *Gegeben:* Eine linear geordnete Folge $X = (x_1, \dots, x_n)$ von verschiedenen Elementen einer linear geordneten Menge $(S, \leq)$
- $y \in S$
- *Suchproblem:* Finde Index $i \in \{0, 1, \dots, n\}$ mit $x_i \leq y < x_{i+1}$

Dabei seien $x_0 = -\infty, x_{n+1} = +\infty$ neue Elemente mit $-\infty < x < +\infty$ für alle $x \in S$.
Binäres Suchen: Zeitkomplexität $O(\log n)$
paralleles Suchen mit $p \leq n$ Prozessoren: Prozessor P_1: zuständig für Initialisierung und Randsingularitäten. Aufteilen von X in $p+1$ Blöcke von etwa gleicher Länge. Jede parallele Runde findet $x_i = y$ oder einen y enthaltenden Block.

Algorithmus 10.2 (Paralleles Suchen für Prozessor P_j)

Eingabe: (1) Ein Feld $X = (x_1, x_2, \ldots, x_n)$ mit $x_1 < x_2 < \ldots < x_n$
(2) ein Element y
(3) die Anzahl p der Prozessoren mit $p \leq n$
(4) die Prozessornummer j mit $1 \leq j \leq p$
Ausgabe: ein Index i mit $x_i \leq y < x_{i+1}$
begin
1. **if**(j=1) **then do**
   1.1 Set $l := 0; r := n+1; x_0 := -\infty; x_{n+1} := +\infty$
   1.2.Set $c_0 := 0; c_{p+1} := 1$
2. **while**$(r - l > p)$ **do**
   2.1.**if**(j=1)**then**{set $q_0 := l; q_{p+1} := r$}
   2.2. Set $q_j := l + j\lfloor \frac{r-l}{p+1} \rfloor$
   2.3. **if**$(y = x_{q_i})$**then** {**return**(q_j); **exit**}
   else {set $c_j := 0$ **if** $(y > x_{q_j})$ and $c_j := 1$ **if** $(y < x_{q_j})$}
   2.4. **if**$(c_j < c_{j+1})$**then**{set $l := q_j; r := q_{j+1}$}
   2.5. **if**$(j = 1$ and $c_0 < c_1)$**then**{set $l := q_0; r := q_1$}
3. **if**$(j \leq r - l)$**then do**
   3.1. Case statement:
   $y = x_{l+j}$:{**return**$(l+j)$;**exit**}
   $y > x_{l+j}$:set $c_j := 0$
   $y < x_{l+j}$:set $c_j := 1$
   3.2. **if**$(c_{j-1} < c_j)$**then return**$(l + j - 1)$
end

Beispiel: (zum Algorithmus 10.2) Für die Eingabe

$$X = (2, 4, 6, \ldots, 30), y = 19, p = 2$$

hat P_1 nach Schritt 1 berechnet:

$$l = 0, r = 16, c_0 = 0, c_3 = 1, x_0 = -\infty, x_{16} = +\infty.$$

Die while-Schleife durchläuft drei Iterationen.

Iteration:	1	2	3
q_0	1	1	1
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

Ergebnis durch P_1: **return**(q_1) mit $q_1 = 9$

Satz 10.28 *Zu der Folge $X = (x_1, x_2, \ldots, x_n)$ mit $x_1 < x_2 < \ldots < x_n$ und $y \in S$ berechnet der Algorithmus 10.2 einen Index i mit $x_i \leq y < x_{i+1}$ in der Zeitkomplexität $O\left(\frac{\log(n+1)}{\log(p+1)}\right)$, wobei $p \leq n$ die Anzahl der Prozessoren ist.*

Beweis: Zur Komplexität: In der i+1-ten Iteration der while-Schleife wird die Länge der zu durchsuchenden Unterfolge von $s_i = r - l$ auf $s_{i+1} \leq \frac{r-l}{p+1} + p = \frac{s_i}{p+1} + p$ gesetzt, was die Länge des (p+1)-ten Blockes beschränkt. Mit $s_0 = n+1$ löst man die rekurrente Ungleichung zu $s_i \leq \frac{n+1}{(p+1)^i} + p + 1$. Also werden $O\left(\frac{\log(n+1)}{\log(p+1)}\right)$ Iterationen der Länge $O(1)$benötigt. Schritt 3 benötigt $O(1)$ Zeit. □

Anmerkung: Der Algorithmus ist optimal, wenn p konstant ist.

Als Nächstes wird ein Algorithmus zum parallelen Mischen entwickelt, der in $O(\log \log n)$ Zeit arbeitet. Das ist erstaunlich, da schon die Maximumbestimmung $\Omega(\log n)$ parallele Schritte erfordert (bezogen auf eine CREW PRAM, d.h. eine PRAM die parallel lesen (concurrent read), aber nur exklusiv schreiben (exclusive write) darf. Diese PRAM-Modelle werden in anderen Veranstaltungen, wie z.B. BUK genauer erklärt.)

Lemma 10.29 *Sei Y eine Folge von m Elementen einer linear geordneten Menge $(S, \leq)$ und X eine Folge von n verschiedenen Elementen mit $m \in O(n^s)$ für eine Konstante $0 < s < 1$. Dann kann $rank(Y : X)$ in $O(1)$ Zeit mit $O(n)$ Operationen berechnet werden.*

Beweis:

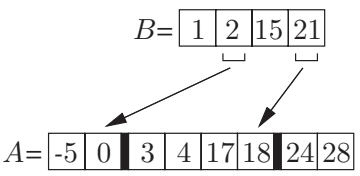
- Bestimme $rank(y : X)$ für jedes $y \in X$ mit Algorithmus 10.2 mit $p = \lfloor \frac{n}{m} \rfloor \in \Omega(n^{1-s})$.
- Also kann $(y : X)$ für jedes $y \in Y$ in der Zeit $O\left(\frac{\log(n+1)}{\log(p+1)}\right) = O\left(\frac{\log(n+1)}{\log(n^{1-s})}\right) = O(1)$ berechnet werden.
- Die Anzahl der Operationen für ein Element ist $O(p) = O(\frac{n}{m})$, da $p = \lfloor \frac{n}{m} \rfloor$ und die Zeitkomplexität für jeden Prozessor $O(1)$ ist. Bei m Elementen erhält man also $O(n)$ Operationen.

□

Nun soll $rank(B : A)$ für sortierte Folgen A mit $|A| = n$ und B mit $|B| = m$ berechnet werden. A und B sollen keine gemeinsamen Elemente enthalten.
Wieder: Berechnen durch Aufteilen von B in Blöcke der Länge von etwa $\sqrt{m}$:
Daraus: Algorithmus 10.3

Beispiel: (zum Algorithmus 10.3) $(m = 4, \sqrt{m} = 2)$

- *1. Schritt*



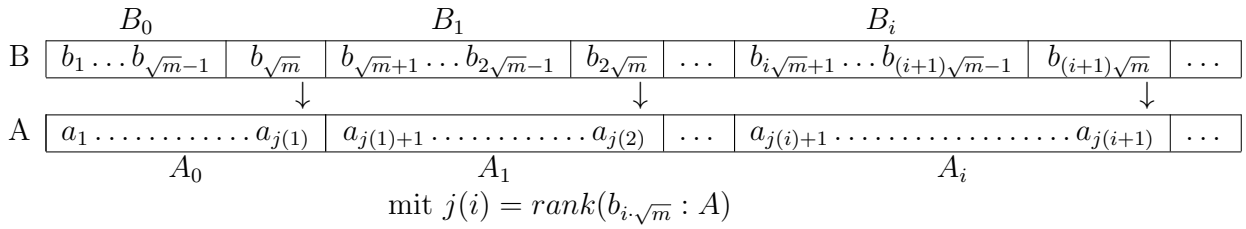


Abbildung 10.12: Aufteilung von B in Blöcke

Algorithmus 10.3 (Ordne eine sortierte Folge in eine andere sortierte Folge ein)

Eingabe: Zwei Felder $A = (a_1, \ldots, a_n)$ und $B = (b_1, \ldots, b_m)$ in aufsteigender Reihenfolge.
Ausgabe: Das Feld $rank(B : A)$.
begin

1. Wenn $m < 4$ dann ordne die Elemente von B durch Anwendung des Algorithmus 10.2 mit $p = n$. Fertig.
2. Ordne die Elemente $b_{\lfloor\sqrt{m}\rfloor}, b_{2\lfloor\sqrt{m}\rfloor}, \ldots, b_{i\lfloor\sqrt{m}\rfloor}, \ldots, b_m$ in A mit Hilfe des Algorithmus 10.2 ein. Dabei sei $p = \sqrt{n}$ und $rank(b_{i\lfloor\sqrt{m}\rfloor} : A) = j(i)$, für $1 \leq i \leq \sqrt{m}$ und $j(0) = 0$
3. Für $0 \leq i \leq \sqrt{m} - 1$ sei $B_i = (b_{i\lfloor\sqrt{m}\rfloor+1}, \ldots, b_{(i+1)\lfloor\sqrt{m}\rfloor-1})$ und $A_i = (a_{j(i)+1}, \ldots, a_{j(i+1)})$; Wenn $j(i) = j(i+1)$ dann setze $rank(B_i : A_i) = (0, \ldots, 0)$, ansonsten berechne $rank(B_i : A_i)$ rekursiv.
4. Sei $1 \leq k \leq m$ ein willkürlicher Index, der kein Vielfaches von $\lfloor\sqrt{m}\rfloor$ ist und sei $i = \lfloor\frac{k}{\lfloor\sqrt{m}\rfloor}\rfloor$. Dann ist $rank(b_k : A) = j(i) + rank(b_k : A_i)$

end

- *2.Schritt* $j(0) = 0 \quad j(1) = 2 \quad j(2) = 6$
- *3.Schritt* $B_0 = (1) \quad A_0 = (-5, 0)$
  $B_1 = (15) \quad A_1 = (3, 4, 17, 18)$
- *4.Schritt* $rank(b_1, A) = rank(1, A) = j(0) + rank(1 : A_0) = 2$
  $rank(b_3, A) = rank(15, A) = j(1) + rank(15 : A_1) = 2 + 2 = 4$
  Also: $rank(B : A) = (\underline{2}, 2, \underline{4}, 6)$

Lemma 10.30 *Der Algorithmus 10.3 berechnet* $rank(B : A)$ *in der Zeit* $O(\log\log n)$ *mit* $O\left((n+m) \cdot \log\log m\right)$ *Operationen.*

Beweis: Die Korrektheit wird durch Induktion über m bewiesen.
Der Induktionsanfang $m = 3$ bedeutet die Folge (b_1, b_2, b_3) in A einzuordnen. Dies erfolgt mit Zeile 1. Wir nehmen jetzt an, dass die Induktionsbehauptung für alle $m' < m$ mit $m \geq 4$ gilt und beweisen, dass alle Elemente in B_i zwischen $a_{j(i)}$ und $a_{j(i+1)+1}$ liegen (für jedes i mit $0 \leq i \leq \sqrt{m} - 1$).
Jedes Element p in B_i erfüllt $b_{i\sqrt{m}} < p < b_{(i+1)\sqrt{m}}$. Da $j(i) = rank(b_{i\sqrt{m}} : A)$ und $j(i+1) = rank(b_{(i+1)\sqrt{m}} : A)$ gilt $a_{j(i)} < b_{i\sqrt{m}}$ und $b_{(i+1)\sqrt{m}} < a_{j(i+1)+1}$, und somit auch $a_{j(i)} < p < a_{j(i+1)+1}$. Diese Tatsache zeigt, das jedes Element p des Blocks B_i in den Block A_i eingefügt wird. Damit gilt $rank(p : A) = j(i) + rank(p : A_i)$, weil $j(i)$ die Anzahl der Elemente in A ist, die vor A_i liegen. Damit folgt die Korrektheit durch Induktion.

Nun zu den Komplexitätsschranken. Sei $T(n, m)$ die parallele Zeit, die benötigt wird, um B in A einzufügen, wobei $|B| = m$ und $|A| = n$ sei.
Schritt 2 bewirkt $\sqrt{m}$ Aufrufe des Algorithmus 10.2, wobei $p = \sqrt{n}$ gilt. Dessen Zeitkomplexität ist $O(\frac{\log(n+1)}{\log(\sqrt{n}+1)}) = O(1)$ und die Anzahl der Operationen wird durch $O(\sqrt{m}.\sqrt{n}) = O(n + m)$ beschränkt, da $2\sqrt{m}.\sqrt{n} \leq n + m$. Außerhalb der rekursiven Aufrufe benötigen Schritt 3 und 4 $O(1)$ Zeit mit $O(n + m)$ Operationen.
Sei $|A_i| = n_i$ für $0 \leq i \leq \sqrt{m} - 1$. Der zum Paar (B_i, A_i) gehörende rekursive Aufruf benötigt $T(n_i, \sqrt{m})$ Schritte. Also gilt $T(n, m) \leq max_i T(n_i, \sqrt{m}) + O(1)$ und $T(n, 3) = O(1)$. Eine Lösung dieser Rekurrenzungleichung ergibt $T(n, m) = O(\log \log m)$. Da die Anzahl der Schritte jedes rekursiven Aufrufs $O(n + m)$ ist, ergibt sich die Gesamtzahl der Operationen des Algorithmus 10.3 mit $O((n + m) \log \log m)$. □

Korollar 10.31 *Zwei sortierte Folgen der Länge n können in $O(\log \log n)$ Zeit mit $O(n \cdot \log \log n)$ Operationen gemischt werden.*

Anmerkung: Dieser Algorithmus ist nicht optimal.

10.4 Paralleles Sortieren

Schnelles Sortieren ist in Anwendungen von großer Bedeutung. Daher soll hier ein paralleler, auf Mischen beruhender Algorithmus vorgestellt werden. Im vorigen Abschnitt wurde ein paralleler Algorithmus zum Mischen zweier Folgen behandelt, der aber nicht optimal ist. Er kann aber dazu benutzt werden, um einen optimalen Algorithmus zu konstruieren, indem die Folgen A und B in Blöcke der Länge $\lceil log(log(n)) \rceil$ zerlegt werden. Das ergibt folgendes Ergebnis (siehe [Jáj92], Abschnitt 4.2.3):

Satz 10.32 *Die Aufgabe, zwei sortierte Folgen der Länge n zu mischen, kann mit einem parallelen Algorithmus in der Zeit* $O(\log\log\ n)$ *mit einer Gesamtzahl von* $O(n)$ *Operationen erledigt werden.*

Der folgende parallele Algorithmus arbeitet wie *merge-sort*. Die zu sortierende Folge X wird in Teilfolgen X_1 und X_2 ungefähr gleicher Länge zerlegt, die getrennt sortiert und das Ergebnis durch Mischen zusammengefügt wird. Dies kann implementiert werden, indem ein (balancierter) Binärbaum nach Abbildung 10.4 von den Blättern her erzeugt wird. Die Wurzel enthält dann die sortierte Folge. Dazu wird für jeden Knoten v die entsprechende sortierte Teilliste mit $L[v]$ bezeichnet. Der j-te Knoten der Höhe h sei $v = (h, j)$.

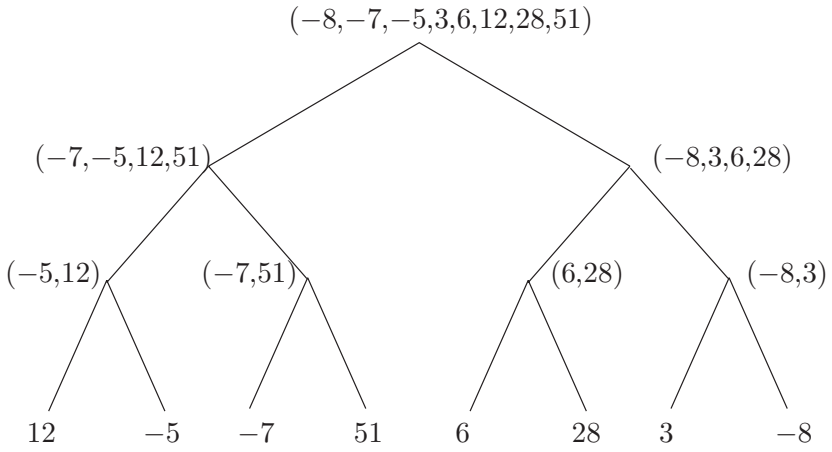


Abbildung 10.13: Binärbaum

Für obiges Beispiel wird das Ergebnis an der Wurzel $L(3, 1)$ erreicht.

Satz 10.33 *Der Algorithmus 10.4 ist mit einer Zeitkomplexität von* $O(\log n \cdot \log\log n)$ *und* $O(n \cdot \log n)$ *Operationen optimal.*

Beweis: In Höhe h sind $\frac{n}{2^h}$ Folgen der Länge 2^hzu mischen. Also gilt nach Satz 10.32 für Operationen in Höhe h:

$$T(n) = O(log\, log\, 2^h) = O(log\, h) \leq O(log\, log\, n)$$

Algorithmus 10.4 (Parallel Merge Sort)

Eingabe: Ein Feld X der Ordnung n, wobei $n = 2^l$ für eine ganze Zahl l ist.
Ausgabe: Ein balancierter binärer Baum mit n Blättern derart, dass $L(h, j)$ für jedes $0 \leq h \leq \log n$ die sortierte Teilfolge der Elemente im Teilbaum mit Wurzel (h, j) enthält (mit $1 \leq j \leq \frac{n}{2^h}$). Damit enthält der Knoten (h, j) die sortierte Liste der Elemente $X(2^h(j-1)+1), X(2^h(j-1)+2), \ldots, X(2^h j)$.
begin
1. **for** $1 \leq j \leq n$ **pardo**
   $L(0, j) := X(j)$
2. **for** $h = 1$ **to** $\log n$ **do**
   for $1 \leq j \leq \frac{n}{2^h}$ **pardo**
   Mische $L(h-1, 2j-1)$ und $L(h-1, 2j)$ zur sortierten Liste $L(h, j)$

end

und

$$W(n) = O(2^h \cdot \frac{n}{2^h}) = O(n).$$

Auf alle $log\, n$ Ebenen bezogen ergibt dies $T(n) = O(log\, n \cdot log\, log\, n)$ und $W(n) = O(n \cdot log\, n)$.

□

Anmerkung: Es existiert ein solcher optimaler Algorithmus mit Zeitkomplexität $O(\log n)$. Dieses Ergebnis kann nicht verbessert werden, wenn $p \leq n$ Prozessoren benutzt werden. Gilt $2n \leq p \leq n^2$, dann kann $O\left(\frac{\log n}{\log\log \frac{2p}{n}}\right)$ erreicht werden.

Aufgabe 10.34 (parallele Algorithmen)
Gegeben sei ein Feld mit $n > 0$ Elementen von ganzen Zahlen. Es sollen verschiedene parallele Algorithmen zur Bestimmung eines maximalen Elementes entwickelt werden. Dabei sollen jeweils die *Zeitkomplexität* $T(n)$, die *Prozessorkomplexität* $P(n)$ und die *Operationenkomplexität* $W(n)$ betrachtet werden.

a) Hier sei $n = 2^k$. Entwickeln Sie mit der Vorstellung eines binären Baumes einen Algorithmus mit $P(n) = \mathcal{O}(n)$, $W(n) = \mathcal{O}(n)$ und $T(n) = \mathcal{O}(log\ n)$. Ist der Algorithmus optimal?
b) Hier sei $n = 2^{2^k}$. Entwickeln Sie mit der Vorstellung eines doppel-logarithmischen Baumes[4] einen Algorithmus mit $P(n) = \mathcal{O}(n)$ und $T(n) = \mathcal{O}(log\ log\ n)$. Ist der Algorithmus optimal? Kann er optimal gemacht werden, falls er es noch nicht ist?
c) Entwickeln Sie mit der Vorstellung einer $(n \times n)$-Matrix einen Algorithmus mit $P(n) = \mathcal{O}(n^2)$ und $T(n) = \mathcal{O}(1)$. Hierbei ist jedoch von Prozessoren auszugehen, die gleichzeitig eine Variable lesen und beschreiben können (*CRCW-PRAM*). Letzteres soll aber nur dann erfolgen, wenn die beteiligten Prozessoren den gleichen Wert schreiben wollen.

[4]Ein doppel-logarithmischer Baum mit $n = 2^{2^k}$ Blättern ist folgendermaßen definiert. Die Wurzel (Höhe 0) hat $2^{2^{k-1}} = \sqrt{n}$ Nachkommen, diese wiederum $2^{2^{k-2}}$ Nachkommen usw. Allgemein haben Knoten mit Höhe $i \in \{0, \ldots, k-1\}$ genau $2^{2^{k-i-1}}$ Nachkommen. Die Knoten mit Höhe k haben zwei Blätter als Nachkommen.