Kapitel 8

Parallele Algorithmen

Der Literaturtip. Zu parallelen Algorithmen vgl. [GR90].

Beim Entwurf paralleler Algorithmen ist in stärkerem Maß als bei sequentiellen Algorithmen das zugrundeliegende Berechnungsmodell von besonderer Bedeutung. Das Hauptmerkmal paralleler Algorithmen ist, dass anstatt eines Prozessors mehrere Prozessoren gleichzeitig, mehr oder weniger unabhängig voneinander, Operationen ausführen können. Mögliche Berechnungsmodelle, und auch wirkliche Parallelrechner, unterscheiden sich etwa darin, wie unabhängig die Prozessoren voneinander sind (gleichartige bzw. unterschiedliche Operationen in einem parallelen Schritt; Kommunikation kostet mehr oder weniger viel Berechnungszeit, etc.)

Ein sinnvolles Berechnungsmodell für den Entwurf von parallelen Algorithmen auf relativ hohem Abstraktionsniveau und für theoretische Betrachtungen wie etwa Komplexitätsbetrachtungen ist die PRAM (Parallel Random Access Machine), eine parallele Version der RAM (siehe Einführung). Das Konzept einer PRAM, das am meisten zitiert und für Algorithmen zugrundegelegt wird, stammt von Fortune & Wyllie (1978).

8.1 Das PRAM Modell

- unbegrenzte Prozessorenzahl;
- unbegrenzter globaler Speicher, auf den alle Prozessoren Zugriff haben;
- Speicherzellen des globalen Speichers wie bei RAM;
- Operationen, die jeder einzelne Prozessor ausführen kann wie bei RAM;
- Jeder Prozessor hat einen eigenen lokalen Speicher, der ebenfalls unbegrenzt ist.

Jeder Prozessor darf nur auf seinen eigenen lokalen Speicher zugreifen, nicht aber auf den lokalen Speicher eines anderen Prozessors. Alle Prozessoren dürfen auf den globalen Speicher zugreifen.

Problem. Was passiert, wenn mehrere Prozessoren gleichzeitig aus derselben Speicherstelle des globalen Speichers lesen wollen oder dieselbe Speicherzelle des globalen Speichers beschreiben wollen? Alle der folgenden Kombinationen sind denkbar und werden als Modelle untersucht bzw. beim Algorithmus zugrundegelegt:

gleichzeitiges Lesen erlaubt CR (**concurrent read**)	gleichzeitiges Lesen verboten ER (**exclusive read**)
gleichzeitiges Schreiben erlaubt CW (**concurrent write**)	gleichzeitiges Schreiben verboten EW (**exclusive write**)

Wir wollen uns hier auf die CREW-PRAM beschränken.

8.2 Komplexität von parallelen Algorithmen

Wir betrachten wieder die Laufzeit im worst-case. Beim Ablauf eines parallelen Algorithmus werden N Operationen, die gleichzeitig von N Prozessoren ausgeführt werden, als ein (paralleler) Berechnungsschritt gezählt. Dann ist die **Laufzeit** $T_{\mathcal{A}}(n)$ eines parallelen Algorithmus $\mathcal{A}$ definiert als

Definition 8.1 $T_{\mathcal{A}}(n) := \max\{$*Anzahl der Berechnungsschritte von* $\mathcal{A}$ *bei Eingabe des Problembeispiels* I*, wobei* I *ein Problembeispiel der Größe* n *ist*$\}$.

Bemerkung. Die Berechnung beginnt, wenn der erste Prozessor aktiv wird, also irgendeine Operation ausführt, und endet, nachdem der letzte Prozessor inaktiv geworden ist, also keine Operation mehr ausführt. Die Prozessoren arbeiten synchron.

Für die Güte eines parallelen Algorithmus ist neben der Laufzeit (und dem Speicherplatzbedarf, den wir hier nicht betrachten) die Anzahl der benötigten Prozessoren relevant. Die **Prozessorenzahl** $P_{\mathcal{A}}(n)$ eines parallelen Algorithmus $\mathcal{A}$ ist definiert als

Definition 8.2 $P_{\mathcal{A}}(n) := \max\{$*Anzahl an Prozessoren, die während des Ablaufs von* $\mathcal{A}$ *bei Eingabe des Problembeispiels* I *gleichzeitig aktiv ist, wobei* I *ein Problembeispiel der Größe* n *ist*$\}$.

(Dies entspricht also wieder einer worst-case-Abschätzung).

Ein paralleler Algorithmus $\mathcal{A}$ steht natürlich in Konkurrenz zu sequentiellen Algorithmen. Dabei interessiert uns vor allem:

$$\text{speed-up}(\mathcal{A}) := \frac{\begin{array}{c}\text{worst-case Laufzeit des schnellsten}\\ \text{bekannten sequentiellen Algorithmus}\end{array}}{\begin{array}{c}\text{worst-case Laufzeit des}\\ \text{parallelen Algorithmus } \mathcal{A}\end{array}}$$

Je größer der speed-up($\mathcal{A}$) ist, umso besser ist natürlich der Algorithmus $\mathcal{A}$. Im Idealfall hofft man natürlich, einen speed-up von N zu erreichen, wenn etwa $P_{\mathcal{A}}(n) =: N$ (für alle n). Bei PRAM-Algorithmen geht man allerdings meist davon aus, dass die Prozessorenzahl abhängig von der Problemgröße ist (wie aus der Definition 8.2 von $P_{\mathcal{A}}(n)$ ersichtlich). Man betrachtet also bei $T_{\mathcal{A}}$, $P_{\mathcal{A}}$, speed-up($\mathcal{A}$), usw. asymptotisches Verhalten.

Ein Qualitätsmaß für einen parallelen Algorithmus $\mathcal{A}$, in welches also sinnvoll Laufzeit und Prozessorenzahl eingehen, sind die **Kosten** $C_{\mathcal{A}}$ von $\mathcal{A}$:

$$C_{\mathcal{A}}(n) := T_{\mathcal{A}}(n) \cdot P_{\mathcal{A}}(n)$$

Sind asymptotisches Wachstum von $C_{\mathcal{A}}(n)$ und die schärfste asymptotische untere Schranke für die Laufzeit eines sequentiellen Algorithmus gleich, so nennt man $\mathcal{A}$ **kostenoptimal**. Kennt man keine gute untere Schranke, so betrachtet man die **Effizienz** $E_{\mathcal{A}}$ von $\mathcal{A}$:

$$E_{\mathcal{A}}(n) := \frac{\begin{array}{c}\text{worst-case Laufzeit des schnellsten}\\ \text{bekannten sequentiellen Algorithmus}\end{array}}{\text{Kosten des parallelen Algorithmus } \mathcal{A}}$$

Man kann davon ausgehen, dass $E_{\mathcal{A}}(n) \leq 1$, denn sonst könnte man aus $\mathcal{A}$ einen sequentiellen Algorithmus ableiten mit (asymptotisch) besserer Laufzeit, als die Laufzeit des schnellsten bekannten sequentiellen Algorithmus,

indem man einen Prozessor alle Operationen eines parallelen Berechnungsschritts von $\mathcal{A}$ hintereinander ausführen lässt. Dann ist die Laufzeit $\approx C_{\mathcal{A}}(n)$.

8.3 Die Komplexitätsklassen

Die am meisten untersuchte Klasse in Zusammenhang mit parallelen Algorithmen ist die Klasse $\mathcal{NC}$:

Definition 8.3 (Nick's Class nach Nicholas Pippinger) *Die Klasse $\mathcal{NC}$ ist die Klasse der Probleme, die durch einen parallelen Algorithmus $\mathcal{A}$ mit polylogarithmischer Laufzeit und polynomialer Prozessorenzahl gelöst werden kann, d.h. $T_{\mathcal{A}}(n) \in \mathcal{O}((\log_n)^{k_1})$ mit k_1 Konstante, und $P_{\mathcal{A}}(n) \in \mathcal{O}(n^{k_2})$ mit k_2 Konstante.*

Eine wichtige offene Frage ist: Gilt $\mathcal{P} = \mathcal{NC}$? Vermutung ist: „nein". Wie bei der Frage, ob $\mathcal{P} = \mathcal{NP}$ ist, gibt es auch hier ein Vollständigkeitskonzept. Parallele Laufzeitkomplexität steht in engem Zusammenhang zu sequentieller Speicherplatzkomplexität.

Definition 8.4 (Steve's Class nach Stephan Cook) *Die Klasse $\mathcal{SC}$ ist die Klasse der Probleme, die durch einen sequentiellen Algorithmus mit polylogarithmischem Speicherplatzbedarf und polynomialer Laufzeit gelöst werden kann.*

Die offene Frage ist hier: $\mathcal{NC} = \mathcal{SC}$?

Einschub. Verschiedene PRAM-Modelle wie CREW-PRAM und EREW-PRAM, oder ein Modell, bei dem zu Beginn der Berechnung eine beliebige Anzahl von Prozessoren aktiv sind gegenüber einem Modell, bei dem zu Beginn nur ein Prozessor aktiv ist und alle anderen erst „aufgeweckt" werden müssen, unterscheiden sich in der Laufzeit im wesentlichen nur um einen Faktor $\log N$ (bei N Prozessoren, die gleichzeitig lesen wollen, bzw. aktiviert werden sollen). Dies lässt sich durch den folgenden Übertragungsalgorithmus, bei dem ein Wert an N Prozessoren auf einer EREW-PRAM übertragen wird, veranschaulichen.

Formale Beschreibung des Algorithmus BROADCAST.

Eingabe: Wert m.
Datenstruktur: Array A der Länge N im globalen Speicher.

BROADCAST(N)
1. P_1 liest m,
   kopiert m in den eigenen Speicher
   und schreibt m in $A[1]$.
2. Für $i = 0$ bis $\lceil \log N \rceil - 1$ führe aus: (MERGEWITHHELP)
3. Für alle j $(2^i + 1 \leq j \leq 2^{i+1})$ führe parallel aus:
4. P_j liest m aus $A[j - 2^i]$,
   kopiert m in den eigenen Speicher
   und schreibt m in $A[j]$.

Die Laufzeit ist in $\mathcal{O}(\log N)$.

8.4 Parallele Basisalgorithmen

8.4.1 Berechnung von Summen

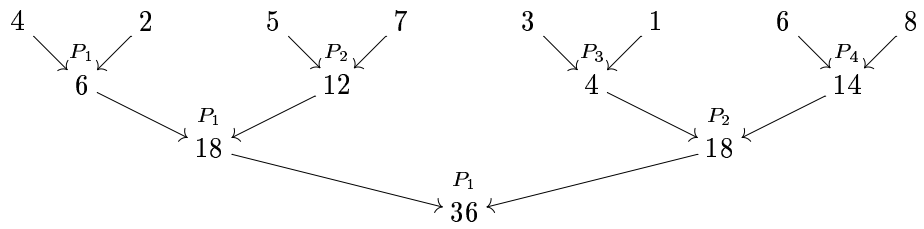
Ziel ist die Berechnung der Summe (bzw. Produkt, Minimum, Maximum, allgemein n-fache binäre Operation) aus n Werten $a_1, \ldots a_n$.

Formale Beschreibung des Algorithmus SUMME.

Eingabe: n Werte $a_1, \ldots, a_n$, o.B.d.A. sei $n = 2^m$.
Ausgabe: $\sum_{i=1}^{n} a_i$

SUMME($a_1, \ldots, a_n$)
1. Für $i = 1$ bis m führe aus:
2. Für alle j, $1 \leq j \leq \frac{n}{2^i}$ führe parallel aus:
3. Prozessor P_j berechnet $a_j := a_{2j-1} + a_{2j}$.
4. Gib a_1 aus.

Beispiel: Die Berechnung einer Summe.



Es ist $P_{\mathcal{A}}(n) = n/2$, $T_{\mathcal{A}}(n) \in \mathcal{O}(\log n)$, $C_{\mathcal{A}} \in \mathcal{O}(n \log n)$. Der Algorithmus $\mathcal{A}$ ist also nicht kostenoptimal.

Bemerkung. Das logische ODER aus n boooleschen Variablen (0 und 1) kann ebenso auf einer CREW-PRAM in $\mathcal{O}(\log n)$ berechnet werden. Auf einer CRCW-PRAM könnte man mit n Prozessoren das logische ODER aus n Werten in $\mathcal{O}(1)$ bestimmen, wenn concurrent-write aufgelöst würde, indem mehrere Prozessoren an dieselbe Speicherzelle schreiben dürften g.d.w. sie denselben Wert schreiben wollen:

1. Für alle i, $1 \leq i \leq n$ führe parallel aus:
2. Prozessor P_i liest i-ten Eingabewert x_i.
3. Falls $x_i = 1$:
4. P_i schreibt Wert 1 in die Speicherzelle „1“ des globalen Speichers.

Geht man davon aus, dass zu Beginn der Berechnung in allen Speicherzellen des globalen Speichers 0 steht, so hat das Ergebnis des logischen ODER den Wert 1 g.d.w. am Ende in Speicherzelle „1“ eine 1 steht.

Zurück zum Algorithmus $\mathcal{A}$ zur Berechnung von SUMME. Durch Rescheduling lässt sich aus $\mathcal{A}$ ein ein paralleler Algorithmus $\mathcal{A}'$ gewinnen, der Kosten $C_{\mathcal{A}'}(n) \in \mathcal{O}(n)$ hat. Dazu werden anstatt n Prozessoren $\lceil \frac{n}{\log n} \rceil$ Prozessoren verwendet. Die $\frac{n}{2}$ Operationen, die im ersten Durchlauf von $\mathcal{A}$ von $\frac{n}{2}$ Prozessoren in einem parallelen Berechnungsschritt ausgeführt werden, werden stattdessen von $\lceil \frac{n}{\log n} \rceil$ in höchstens $\frac{\log n}{2}$ parallelen Berechnungsschritten ausgeführt, bzw. allgemein im i-ten Durchlauf $\frac{n}{2^i}$ Operationen in höchstens $\frac{\log n}{2^i}$ Berechnungsschritten. Damit ergibt sich insgesamt

$$T_{\mathcal{A}}(n) \leq c \cdot \sum_{i=1}^{\log n} \frac{\log n}{2^i} = c \cdot \log n \underbrace{\sum_{i=1}^{\log n} \frac{1}{2^i}}_{\leq 1} \in \mathcal{O}(\log n).$$

Es gilt $P_{\mathcal{A}'}(n) = \lceil n / \log n \rceil$, also $C_{\mathcal{A}'} \in \mathcal{O}(n)$.

8.4.2 Berechnung von Präfixsummen

Wir berechnen die Präfixsummen $\sum_{i=1}^{k} a_i$ $(1 \leq k \leq n)$ aus den n gegebenen Werten $a_1, \ldots, a_n$ (bzw. Produkt, Minimum, Maximum). Aus technischen Gründen nennen wir die Eingabewerte $a_n, a_{n+1}, \ldots, a_{2n-1}$; es sei wieder o.B.d.A. $n = 2^m$. Das Verfahren besteht aus zwei Phasen. In der ersten wird SUMME$(a_n, \ldots, a_{2n+1})$ ausgeführt. In der zweiten Phase werden aus dem Ergebnis von SUMME und aus Zwischenergebnissen des Verfahrens die $\sum_{i=n}^{k} a_i$ $(n \leq k \leq 2n-1)$ als Differenzen geeignet berechnet.

Formale Beschreibung des Algorithmus

Eingabe: n Werte $a_n, \ldots, a_{2n-1}$; $n = 2^m$,
Ausgabe: Die n Präfixsummen $\sum_{i=1}^{k} a_i$ für $1 \leq k \leq n$.

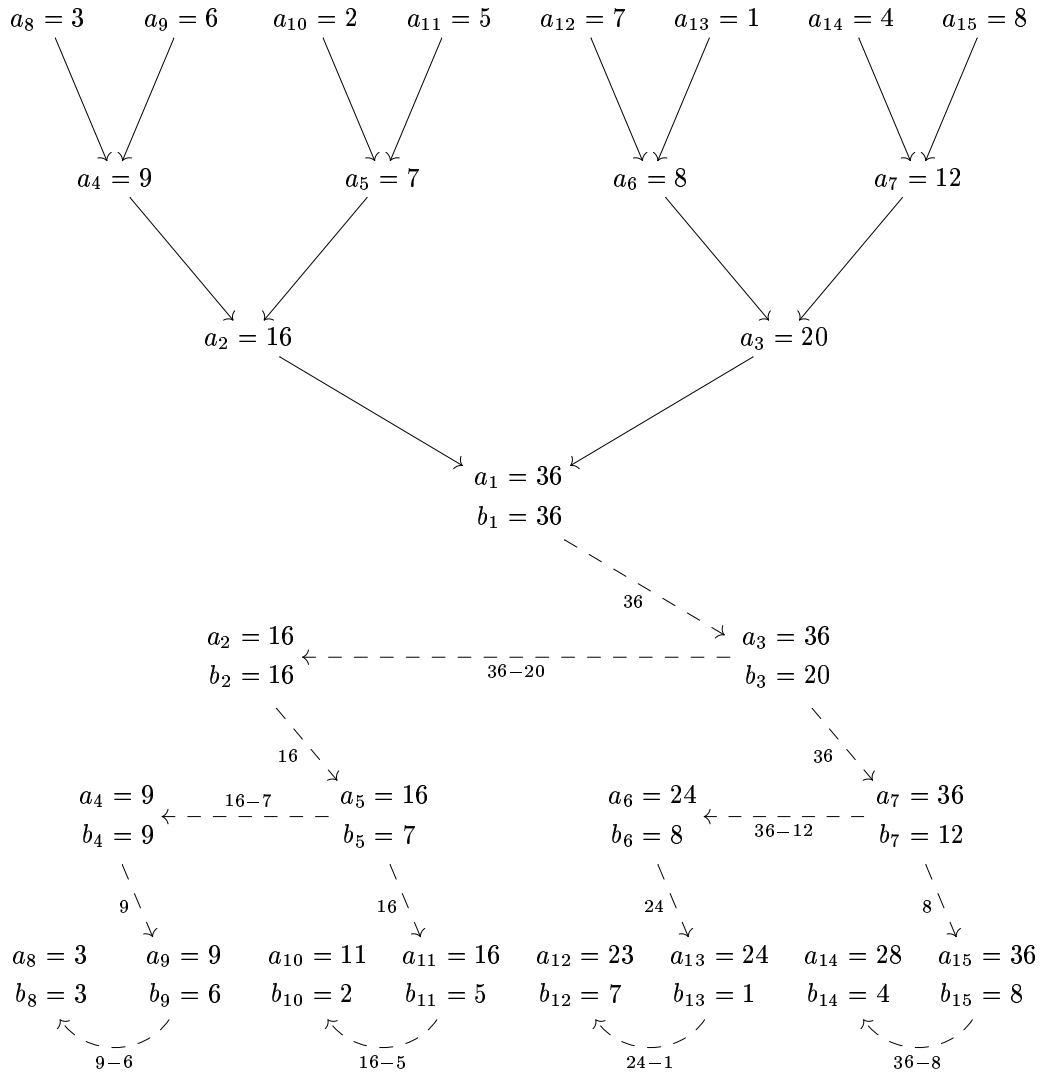
PRÄFIXSUMME$(a_n, \ldots, a_{2n-1})$

1. Für $j = m - 1$ bis 0 führe aus:
2. Für alle i $(2^j \leq i \leq 2^{j+1} - 1)$ führe parallel aus:
3. $a_i := a_{2i} + a_{2i+1}$
4. Setze $b_1 := a_1$.
5. Für $j = 1$ bis m führe aus:
6. Für alle i $(2^j \leq i \leq 2^{j+1} - 1)$ führe parallel aus:
7. Falls i ungerade ist, setze $b_i := b_{\frac{i-1}{2}}$.
8. Falls i gerade ist, setze $b_i := b_{\frac{i}{2}} - a_{i+1}$.
9. Das Ergebnis steht in $b_n, \ldots, b_{2n-1}$.

Durch Induktion lässt sich leicht zeigen, dass $b_{2^j+\ell} = a_{2^j} + \ldots + a_{2^j+\ell}$ ist für $\ell + 0, \ldots, 2^j - 1$.

Komplexität. Offensichtlich ist die Laufzeit in $\mathcal{O}(\log n)$. Die Prozessorenzahl ist in $\mathcal{O}(n)$, es sind maximal $n/2$ Prozessoren gleichzeitig aktiv. Durch Rescheduling kann die Prozessorenzahl wieder auf $\mathcal{O}(n/\log n)$ bei Laufzeit $\mathcal{O}(\log n)$ reduziert werden, womit die Kosten wieder optimal sind.

Beispiel.



8.4.3 Die Prozedur List Ranking oder Short Cutting

In einer linearen Liste (oder allgemeiner einem Wurzelbaum, in dem jedes Element i nur seinen direkten Vorgänger kennt) sollen alle Elemente das erste Element in der Liste bzw. die Wurzel kennenlernen. Dies kann in $\mathcal{O}(\log n)$, wobei n die Länge der Liste ist, bzw. in $\mathcal{O}(\log h)$, wobei h die Höhe des Wurzelbaumes ist, mit n Prozessen (n Anzahl der Elemente im Baum) realisiert werden. Es sei Vor[root]=root.

Formale Beschreibung des Algorithmus.

Eingabe: Lineare Liste bzw. Wurzelbaum der Höhe h mit n Elementen.
Ausgabe: Vorgänger VOR$[i]$ für jedes Element $1 \leq i \leq n$.

LIST RANKING(n, h)
1. Für $j = 1$ bis $\lceil \log h \rceil$ führe aus:
2. Für alle i $(1 \leq i \leq n)$ führe parallel aus:
3. Setze VOR$[i]$:= VOR[VOR$[i]$].

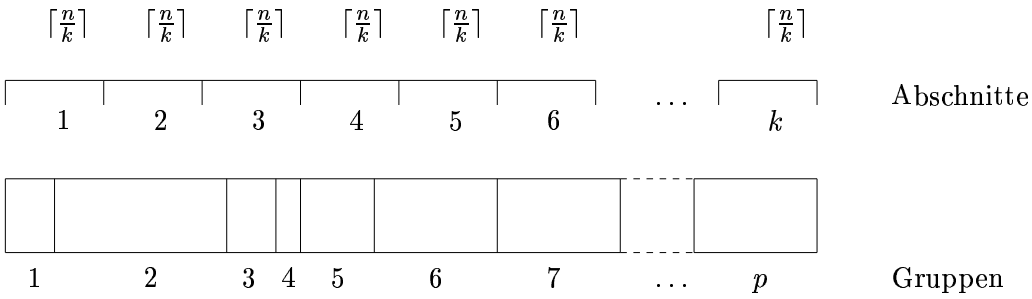
8.4.4 Binäroperationen einer partitionierten Menge mit K Prozessoren

Gegeben. n Werte, die in p Gruppen aufgeteilt sind.

Problem. Mit K Prozessoren sollen die p Werte bestimmt werden, die sich als Binärverbindungen (also Summe, Minimum etc.) der Werte der p Gruppen ergeben. Die Laufzeit soll $t(n)$ sein, mit

$$t(n) \in \begin{cases} \lceil \frac{n}{K} \rceil - 1 + \log K, & \text{falls } n > K \\ \log n, & \text{sonst.} \end{cases}$$

Mit $K \geq n$ Prozessoren können die p Binärverbindungen wie üblich in $\mathcal{O}(\log n)$ berechnet werden. Falls $K < n$ ist, teile die n Werte in K Abschnitte auf. Jeder Abschnitt erhält einen Prozessor.



Die Werte eines Abschnitts gehören entweder alle zu derselben Gruppe oder zu verschiedenen Gruppen. Gehören alle Werte zu derselben Gruppe, so wird die Binärverbindung aus diesen berechnet, ansonsten, falls sie zu m Gruppen gehören, werden m solche Binärverbindungen berechnet. Alle diese werden mit den K Prozessoren jeweils sequentiell in höchstens $\lceil \frac{n}{K} \rceil - 1$ Schritten berechnet. Gehören alle Werte einer Gruppe zu demselben Abschnitt, so sind nach diesen $\lceil \frac{n}{K} \rceil - 1$ Schritten ihre Binärverbindungen endgültig berechnet. Für jeden Abschnitt sind jedoch höchstens 2 (bzw. 1)

Ergebnisse noch nicht endgültig, sondern müssen mit anderen zusammengefasst werden. Falls also für jede Gruppe G_i $(1 \leq i \leq p)$ die Anzahl der noch zusammenzufassenden Teilergebnisse n_i ist, so gilt

$$\sum_{i=1}^{p} n_i \leq 2K - 2.$$

Ordne den Gruppen G_i $(1 \leq i \leq p)$ jeweils $\lfloor \frac{n_i}{2} \rfloor$ Prozessoren zu, dann können diese in $\log n_i$ Zeit die endgültigen Ergebnisse bestimmen. Da $n_i \leq K$ gilt, ist dies in $\log K$. Die Anzahl an Prozessoren ist ausreichend, da

$$\sum_{i=1}^{p} \lfloor \frac{n_i}{2} \rfloor \leq \frac{1}{2} \cdot (2K - 2) < K.$$

8.5 Ein paralleler Algorithmus für die Berechnung der Zusammenhangskomponenten

Gegeben sei ein ungerichteter Graph $G = (V, E)$ mit $V = \{1, \ldots, n\}$. Bestimme durch einen CREW-PRAM-Algorithmus die Zusammenhangskomponenten von G.

Idee. Zunächst wird jeder Knoten als eine aktuelle Zusammenhangskomponente aufgefasst. Der Algorithmus besteht aus maximal $\lceil \log n \rceil$ Phasen, wobei in jeder Phase gewisse aktuelle Zusammenhangskomponenten zu neuen Zusammenhangskomponenten vereinigt werden. In einer Phase wird zunächst für alle Knoten parallel die aktuelle Zusammenhangskomponente (ungleich der eigenen) kleinster Nummer gewählt, die einen Nachbarknoten enthält. Dann wird für alle Knoten i parallel die aktuelle Zusammenhangskomponente kleinster Nummer gewählt unter allen zuvor gewählten Zusammenhangskomponenten, die benachbart sind zu einem Knoten, der in derselben Komponente wie i liegt. Jede Komponente kann nun mit der so bestimmten Zusammenhangskomponente kleinster Nummer zusammengefasst werden. Da bei diesem Schritt Ketten von Zusammenhangskomponenten entstehen können, muss in einer anschließenden Berechnung für alle Zusammenhangskomponenten, die zuvor zusammengefasst worden sind, eine gemeinsame neue Nummer bestimmt werden. Dies wird mit LIST RANKING realisiert. In jeder Phase wird die Zahl der aktuellen Zusammenhangskomponenten, die echte Subgraphen von Zusammenhangskomponenten von G sind, halbiert. Daher endet der ALgorithmus tatsächlich nach maximal $\lceil \log n \rceil$ Phasen.

Der Algorithmus ZUSAMMENHANG(G)
(Chandra, Hirschberg & Sarwarte 1979)

Eingabe: $G = (V, E)$, $V = \{1, \ldots, n\}$
Ausgabe: Zu $i \in V$ steht in $K[i]$ kleinster Knoten, der in derselben Zusammenhangskomponente wie i in G liegt.
Datenstruktur:

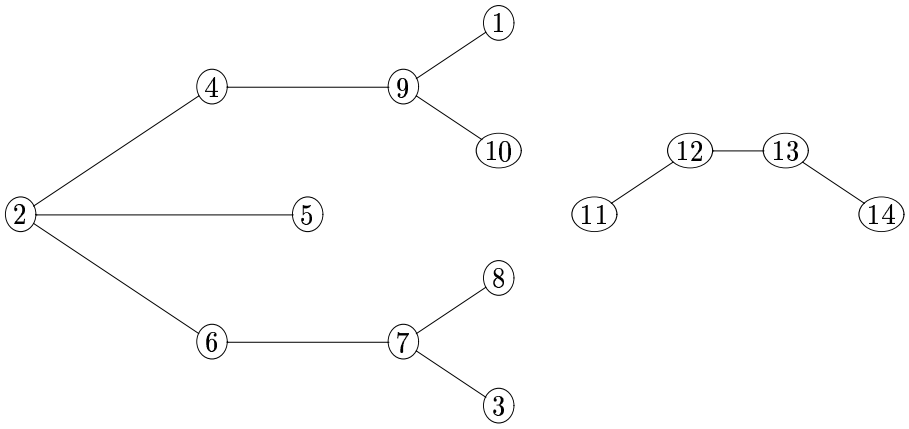
- Array K der Länge n ($K[i]$ enthält Nummer der aktuellen Zusammenhangskomponente, in der i liegt.)
- Array N der Länge n ($N[i]$ enthält die Nummer der aktuellen Zusammenhangskomponente kleinster Nummer, in der ein Nachbar von i bzw. eines Knoten aus $K[i]$ liegt).

ZUSAMMENHANG(G)

1. Für alle i, $1 \leq i \leq n$ führe parallel aus:
2. $\quad K[i] := i$.
3. Für $\ell = 1$ bis $\lceil \log n \rceil$ führe aus:
4. $\quad$ Für alle i $(1 \leq i \leq n)$ führe parallel aus:
5. $\quad\quad N[i] := \min\{K[j] : \{i, j\} \in E \text{ und } K[i] \neq K[j]\}$.
6. $\quad\quad$ Falls kein solches j existiert, setze $N[i] := K[i]$.
7. $\quad$ Für alle i $(1 \leq i \leq n)$ führe parallel aus:
8. $\quad\quad N[i] := \min\{N[j] : K[i] = K[j], N[j] \neq K[j]\}$.
9. $\quad$ Für alle i $(1 \leq i \leq n)$ mit $N[N[i]] = K[i]$ führe parallel aus:
10. $\quad\quad N[i] := \min\{N[i], K[i]\}$.
11. $\quad$ Für alle i $(1 \leq i \leq n)$ führe parallel aus:
12. $\quad\quad K[i] := N[i]$.
13. $\quad$ Für $m = 1$ bis $\lceil \log n \rceil$ führe aus:
14. $\quad\quad$ Für alle i $(1 \leq i \leq n)$ führe parallel aus:
15. $\quad\quad\quad K[i] := K[K[i]]$.

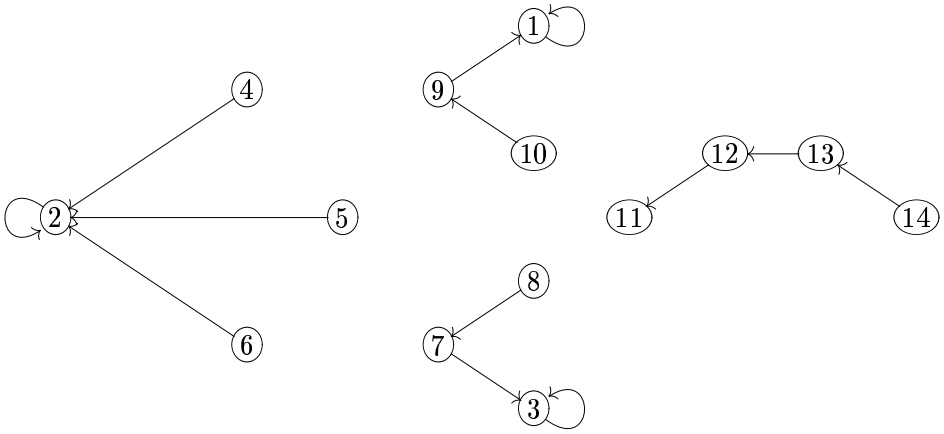
Komplexität: Die Laufzeit ist in $\mathcal{O}(\log^2 n)$: Schleife 3. hat Tiefe $\log n$ und innerhalb von 3. werden mehrere Minimum-Berechnungen vom Aufwand $\mathcal{O}(\log n)$ vorgenommen. 13. ist eine Ausführung von LIST RANKING und benötigt $\mathcal{O}(\log n)$. Die benötigte Prozessorenzahl ist zunächst n^2. Durch Rescheduling kann diese auf $\lceil \frac{n^2}{\log n} \rceil$ gesenkt werden, womit die Kosten in $\mathcal{O}(n^2 \log n)$ sind. Dies ist nicht kostenoptimal.

Beispiel. Wir betrachten folgenden Graphen:

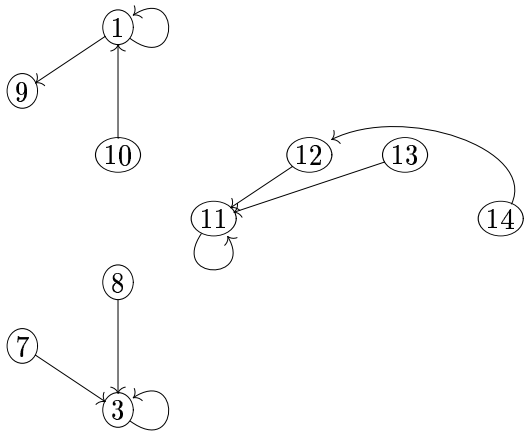


Dann ist $n = 14$, $\lceil \log n \rceil = 4$. Der Algorithmus ZUSAMMENHANG geht wie folgt vor:

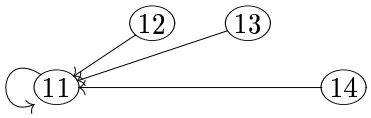
1. $K = [1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13\ 14]$
3. $\ell = 1$
4. $N = [9\ 4\ 7\ 2\ 2\ 2\ 3\ 7\ 1\ 9\ 12\ 11\ 12\ 13]$
7. $N = [9\ 4\ 7\ 2\ 2\ 2\ 3\ 7\ 1\ 9\ 12\ 11\ 12\ 13]$
9. $N = [1\ 2\ 3\ 2\ 2\ 2\ 3\ 7\ 1\ 9\ 11\ 11\ 12\ 13]$
11. $K = [1\ 2\ 3\ 2\ 2\ 2\ 3\ 7\ 1\ 9\ 11\ 11\ 12\ 13]$



13. $m = 1$
14. $K = [1\ 2\ 3\ 2\ 2\ 2\ 3\ 3\ 1\ 1\ 11\ 11\ 11\ 12]$



13. $m = 2$
14. $K = [1\ 2\ 3\ 2\ 2\ 2\ 3\ 3\ 1\ 1\ 11\ 11\ 11\ 11]$



13. Für $m = 3, 4$ ergibt sich keine Änderung mehr.
3. $\ell = 2$
4. $N = [1\ 2\ 3\ 1\ 2\ 3\ 2\ 3\ 2\ 1\ 11\ 11\ 11\ 11]$
7. $N = [2\ 1\ 2\ 1\ 1\ 1\ 2\ 2\ 2\ 2\ 11\ 11\ 11\ 11]$
9. $N = [1\ 1\ 2\ 1\ 1\ 1\ 2\ 2\ 1\ 1\ 11\ 11\ 11\ 11]$
11. $K = [1\ 1\ 2\ 1\ 1\ 1\ 2\ 2\ 1\ 1\ 11\ 11\ 11\ 11]$
13. $m = 1$
14. $K = [1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 11\ 11\ 11\ 11]$
13. Für $m = 2, 3, 4$ ergibt sich keine Änderung mehr.
3. Für $\ell = 3, 4$ ergibt sich keine Änderung mehr.

8.6 Modifikation zur Bestimmung eines MST

Der Algorithmus zur Bestimmung der Zusammenhangskomponenten kann geeignet modifiziert werden, so dass er in $\mathcal{O}(\log^2 n)$ Zeit mit n^2 bzw. $\lceil \frac{n^2}{\log n} \rceil$ Prozessoren einen MST in einem gewichteten Graphen bestimmt.

8.6.1 Der Algorithmus

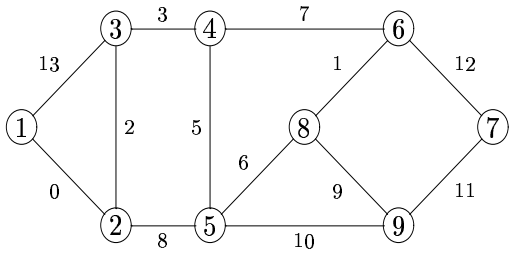
Gegeben. Graph $G = (V, E)$, $V = \{1, \ldots, n\}$ und Kantengewichte c_{ij} für $(i, j) \in V \times V$, wobei $c_{ij} = \infty$, falls $\{i, j\} \notin E$. Zusätzlich werden im Algorithmus Datenstrukturen T und S benutzt. T ist ein Array, das die zum MST gehörenden Kanten enthält. S ist ein Array der Länge n, in dem an der Stelle $S[i]$ die Nummer desjenigen Knoten steht, der in derselben Komponente wie i ist und eine Kante minimalen Gewichts zu einem Knoten einer anderen Komponente hat. In dem Array N steht nun an der Stelle $N[i]$ die Nummer desjenigen Knotens, zu dem es von i aus eine Kante minimalen Gewichts gibt und der in einer anderen Komponente als i liegt.

Im Algorithmus ZUSAMMENHANG werden die Schritte 4. - 12. ersetzt. Der neue Algorithmus sieht damit aus wie folgt:

MST(G)

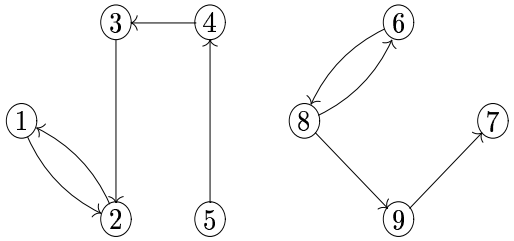
1. Für alle i, $1 \leq i \leq n$ führe parallel aus:
2. $\quad K[i] := i$.
3. Für $\ell = 1$ bis $\lceil \log n \rceil$ führe aus:
4. $\quad$ Für alle i $(1 \leq i \leq n)$ führe parallel aus:
5. $\quad\quad$ Finde $k \in V$ mit $K[i] \neq K[k]$ und
   $\quad\quad$ $c_{ik} = \min\{c_{ij} : 1 \leq j \leq n, K[i] \neq K[j]\}$,
6. $\quad\quad$ setze $N[i] := k$.
7. $\quad$ Für alle i $(1 \leq i \leq n)$ führe parallel aus:
8. $\quad\quad$ Finde $t \in V$ mit $K[i] = K[t]$ und
   $\quad\quad$ $c_{tN[t]} = \min\{c_{jN[j]} : 1 \leq j \leq n, K[i] = K[j]\}$,
9. $\quad\quad$ setze $N[i] := N[t]$ und $S[i] := t$.
10. $\quad$ Für alle i $(1 \leq i \leq n)$ führe parallel aus:
11. $\quad\quad$ Falls $N[N[i]] = S[i]$ und $K[i] < K[N[i]]$:
12. $\quad\quad\quad$ Setze $S[i] := 0$.
13. $\quad$ Für alle i $(1 \leq i \leq n)$ führe parallel aus:
14. $\quad\quad$ Falls $S[i] \neq 0$ und $K[i] = i$ und $c_{N[i],S[i]} \neq \infty$:
15. $\quad\quad\quad$ Setze $T := T \cup \{N[i], S[i]\}$.
16. $\quad\quad$ Falls $S[i] \neq 0$:
17. $\quad\quad\quad$ Setze $K[i] = K[N[i]]$.
18. $\quad$ Für $m = 1$ bis $\lceil \log n \rceil$ führe aus:
19. $\quad\quad$ Für alle i $(1 \leq i \leq n)$ führe parallel aus:
20. $\quad\quad\quad$ $K[i] := K[K[i]]$.

Beispiel. Wir betrachten folgenden Graphen:

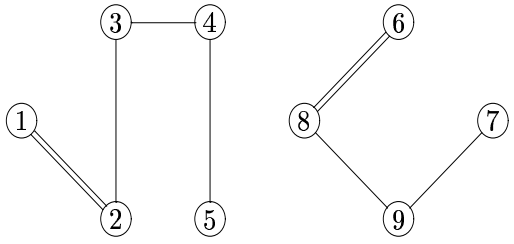


Der Algorithmus geht nun wie folgt vor:

1. $K := [1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9]$
3. $\ell = 1$
4. $N := [2\ 1\ 2\ 3\ 4\ 8\ 9\ 6\ 8]$
7. $N := [2\ 1\ 2\ 3\ 4\ 8\ 9\ 6\ 8]$
   $S := [1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9]$



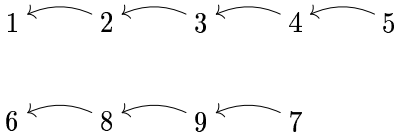
10. $S := [0\ 2\ 3\ 4\ 5\ 0\ 7\ 8\ 9]$
14. $T := \{\{1,2\},\{2,3\},\{3,4\},\{4,5\},\{6,8\},\{7,9\},\{8,9\}\}$
16. $K := [1\ 1\ 2\ 3\ 4\ 6\ 9\ 6\ 8]$



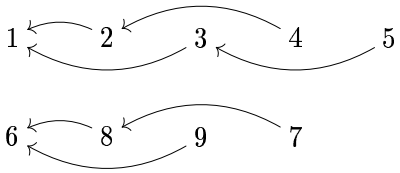
18. $m = 1$
19. $K := [1\ 1\ 1\ 2\ 3\ 6\ 8\ 6\ 6]$
18. $m = 2$
19. $K := [1\ 1\ 1\ 1\ 1\ 6\ 6\ 6\ 6]$
18. Für $m = 3$ und $m = 4$ ergibt sich keine Änderung mehr.

Die Schritte 18–20 lassen sich wie folgt darstellen:

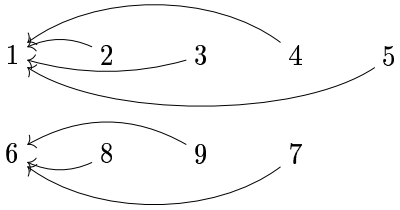
- $m = 0$.



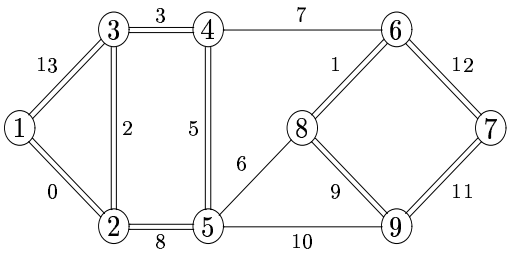
- $m = 1$.



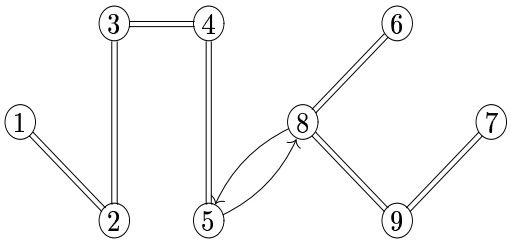
- $m = 2$.



- Ergebnis:

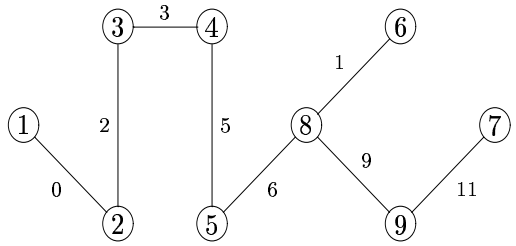


3. $\ell = 2$
4. $N := [6\ 6\ 6\ 6\ 8\ 4\ 1\ 5\ 5]$
7. $N := [8\ 8\ 8\ 8\ 8\ 5\ 5\ 5\ 5]$
   $S := [5\ 5\ 5\ 5\ 5\ 8\ 8\ 8\ 8]$



10. $S := [0\ 0\ 0\ 0\ 0\ 8\ 8\ 8\ 8]$
14. $T := [\{1,2\}\{2,3\}\{3,4\}\{4,5\}\{5,8\}\{6,8\}\{7,9\}\{8,9\}]$
16. $K := [1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1]$
18. Für $m = 1$ ergibt sich keine Veränderung mehr.

Endergebnis:



8.6.2 Reduzierung der Prozessorenzahl auf $\frac{n^2}{\log^2 n}$

Idee. Schritt 4. bis 15. müssen jeweils nur für die Repräsentanten der Komponenten ausgeführt werden, nicht für alle Knoten. Die Anzahl der Komponenten halbiert sich jedoch mindestens in jeder Phase. Bei Vorgabe von $\lceil \frac{n^2}{\log^2 n} \rceil$ Prozessoren wird der Algorithmus an zwei Stellen modifiziert:

1. Minimum in Schritt 4. und 7. wird mit nur $\lceil \frac{n}{\log^2 n} \rceil$ Prozessoren ausgeführt;
2. nach jeder Phase wird die Adjazenzmatrix des Graphen neu aufgestellt, d.h. für alle Knoten i und j werden die Kanten $\{i,j\}$ auf die Repräsentanten der Komponenten von i bzw. j übertragen.

Realisierung von 1. Binäroperation aus n Werten mit K Prozessoren.

Gegeben. n Werte $a_1, \ldots, a_n$.

Problem. Die Summe (bzw. Produkt, Minimum, etc) aus $a_1, \ldots, a_n$ soll mit K Prozessoren in Zeit $\mathcal{O}(t(n))$ bestimmt werden, wobei

$$t(n) \leq \begin{cases} \lceil n/K \rceil - 1 + \log K, & \text{falls } \lceil n/2 \rceil > K \\ \log n, & \text{sonst.} \end{cases}$$

Für $K \geq \lceil n/2 \rceil$ ist das, wie bereits bewiesen, in $\mathcal{O}(\log n)$ Zeit möglich.

Sei $K < \lceil n/2 \rceil$: Teile $a_1, \ldots, a_n$ in K Gruppen auf, wobei jede Gruppe $\lceil n/K \rceil$ bzw. die letzte Gruppe $n - (K-1) \cdot \lceil n/K \rceil$ Werte enthält. Jeder Gruppe wird ein Prozessor zugeteilt, der die Binärportion der Werte seiner Gruppe sequentiell in maximal $\lceil n/K \rceil - 1$ Schritten bestimmt. Die Binäroperation aus diesen K Ergebnissen wird dann in $\log K$ Zeit mit K Prozessoren parallel bestimmt.

Im Algorithmus ergibt sich dann eine Gesamtlaufzeit von

$$T(n) \in \mathcal{O}(\frac{n}{k} + \log n \cdot \log k)$$

für die Schritte 4. und 7. mit $K \leq \lceil n/2 \rceil$ Prozessoren: Da sich die Anzahl der zu behandelnden Repräsentanten in jeder Phase halbiert und nach $\ell = \log n - \lceil \log K \rceil$ höchstens $2K$ Repräsentanten übrig sind, gilt:

$$\begin{aligned} T(n) &\leq \sum_{k=0}^{\ell-1} (\lceil \tfrac{n}{2^k \cdot K} \rceil - 1 + \log K) + \sum_{k=\ell}^{\log n - 1} (\log(\frac{n}{2^k}) \\ &\leq \frac{2n}{K} + \log n \cdot \log K - \log^2 K + \sum_{k=\ell}^{\log n - 1} \log n - k \\ &\in \mathcal{O}(\frac{n}{K} + \log n \cdot \log K). \end{aligned}$$

Die letzte Ungleichung gilt, da $\sum_{k=0}^{\infty} \frac{1}{2^k} \leq 2$.

Realisierung von 2. Um die Adjazenzmatrix des Graphen nach jeder Phase geeignet neu aufzustellen, werden die Knoten der Komponenten entsprechend der Nummer ihrer Repräsentanten angeordnet und jeweils an den Anfang einer solchen Knotengruppe ihr Repräsentant geschrieben. Zum Ermitteln der Einträge für alle Repräsentantenpaare muss jeweils für jede Knotengruppe die ODER-Verbindung der entsprechenden Einträge berechnet werden.

Dafür werden jeder Komponente $K = \lceil \frac{n}{\log^2 n} \rceil$ Prozessoren zugeordnet, um die ODER-Verbindungen für alle Adjazenzen von Knoten der Komponenten zu bilden. Dies ist eine Binärverbindung einer partitionierten Menge und kann mit k Prozessoren in $\mathcal{O}(\lceil \frac{n}{k} \rceil - 1 + \log k)$ berechnet werden (siehe 8.4.4)

Analog wie bei 1. ist die Gesamtlaufzeit über alle Phasen für die Neuberechnungen der Adjazenzmatrix mit K Prozessoren in $\mathcal{O}(\frac{n}{K} + \log n \cdot \log K)$, also bei $K = \lceil \frac{n}{\log^2 n} \rceil$ in $\mathcal{O}(\log^2 n)$.

8.7 ParallelSelect: Parallelisierung von Select zur Bestimmung des k-ten Elements aus n Elementen

Das Verfahren ParallelSelect benutzt p Prozessoren (wobei $p \leq n$) auf einer CREW-PRAM. Sei S die Menge der gegebenen n Elemente. Falls $|S| = n < 5$ ist, wird das k-te Element durch Sortieren bestimmt. Ansonsten wird S in p Teilfolgen der Länge höchstens $\lceil \frac{n}{p} \rceil$ aufgeteilt, und jeder Prozessor bestimmt mittels Select in $\mathcal{O}(\lceil \frac{n}{p} \rceil)$ das mittlere Element seiner Teilfolge. Beachte, dass Select wieder „ganz normal“ in Teilfolgen der Länge 5 unterteilt. Die mittleren Elemente bilden die Menge M, deren mittleres Element m (d.h. das $\lceil \frac{|M|}{2} \rceil$-kleinste) durch einen rekursiven Aufruf von ParallelSelect bestimmt wird. S wird dann wieder in Teilfolgen

$$S_< = \{x \in S : x \leq m\} \text{ und}$$
$$S_> = \{x \in S : x > m\}$$

geteilt. (Voraussetzung sei wieder, dass alle Elemente verschieden sind.) Falls $|S_<| \geq k$ ist, wird dann ParallelSelect rekursiv auf $S_<$ mit k aufgerufen, ansonsten auf $S_>$ mit $k - |S_<|$. Da ParallelSelect kostenoptimal arbeiten soll, können $S_<$ und $S_>$ nicht durch sequentielles Durchlaufen der Menge S und Vergleichen mit m bestimmt werden. Stattdessen unterteilen die p Prozessoren jeweils parallel ihre Teilfolgen S^i $(1 \leq i \leq p)$ in

$$S^i_< = \{x \in S^i : x \leq m\} \text{ und}$$
$$S^i_> = \{x \in S^i : x > m\}.$$

$S_<$ und $S_>$ werden dann aus diesen $S^i_<$ und $S^i_>$ zusammengesetzt, indem die Prozessoren gleichzeitig in verschiedene Positionen eines Arrays $S_<$ im gleichen Speicher ihre $S^i_<$ schreiben. Die Positionen werden zuvor mittels Prefix-Summe berechnet.

Formale Beschreibung von ParallelSelect$(S, k; p)$

Eingabe: S mit $|S| = n$, k, p.

Datenstruktur: Arrays $S_<$ und $S_>$ und M im globalen Speicher. Arrays S^i, $S^i_<$ und $S^i_>$ in den lokalen Speichern für $1 \leq i \leq p$.

1. Falls $n < 5$:
2. Prozessor p_1 liest Folge S, sortiert S und gibt das k-kleinste Element aus.
3. Falls $n \geq 5$ führe aus:
4. Prozessor p_1 berechnet aus n und p die Zahl ℓ mit $p = n^{1-\ell}$.
5. Für alle i $(1 \leq i \leq p)$ führe parallel aus:
6. Prozessor p_i berechnet $(i-1) \cdot \lfloor n^\ell \rfloor$ und $i \cdot \lfloor n^\ell \rfloor - 1$ und kopiert die Elemente aus S, die zwischen Position $(i-1) \cdot \lfloor n^\ell \rfloor$ und Position $i \cdot \lfloor n^\ell \rfloor - 1$ stehen, nach S^i.
7. Prozessor p_i berechnet $n' := n^{1-\ell} \cdot \lfloor n^\ell \rfloor$ und kopiert das Element aus S, das an Position $n' + i$ steht, nach S^i, falls vorhanden.
8. Prozessor p_i führt Select$(S^i, \lceil \frac{1}{2} n^\ell \rceil)$ aus und schreibt das Ergebnis m_i an die i-te Position von M.
9. Prozessor p_1 berechnet $p' := \lceil |M|^{1-\ell} \rceil$ (beachte $|M| = p = n^{1-\ell}$).
10. ParallelSelect$(M, \lceil \frac{|M|}{2} \rceil; p')$ Ergebnis sei m.
11. Für alle i $(1 \leq i \leq p)$ führe parallel aus:
12. Prozessor p_i berechnet
    $S^i_< = \{x \in S^i : x \leq m\}, |S^i_<| =: a_i$
    $S^i_> = \{x \in S^i : x > m\}, |S^i_>| =: b_i$.
13. Präfix-Summe$(a_1, \ldots, a_p)$
    Präfix-Summe$(b_1, \ldots, b_p)$
    Die Ergebnisse seien $A_1, \ldots, A_p$ und $B_1, \ldots, B_p$.
14. Setze $A_0 = B_0 = 0$.
15. Für alle i $(1 \leq i \leq p)$ führe parallel aus:
16. Prozessor p_i schreibt $S^i_<$ an die Positionen $(A_{i-1} + 1)$ bis A_i von $S_<$ und $S^i_>$ an die Positionen $(B_{i-1} + 1)$ bis B_i von $S_>$.
17. Prozessor p_1 berechnet aus $A_p = |S_<|$ die Zahl $s_1 := \lceil A_p^{1-\ell} \rceil$ und aus $B_p = |S_>|$ die Zahl $s_2 := \lceil B_p^{1-\ell} \rceil$.
18. Falls $|S_<| \geq k$ führe aus:
19. ParallelSelect$(S_<; k; s_1)$.
20. Ansonsten:
21. Prozessor p_1 berechnet $k' := k - |S_<|$.
22. ParallelSelect$(S_>; k'; s_2)$.

Laufzeit. Die Laufzeit von ParallelSelect ist in $\mathcal{O}(n^\ell)$, wobei $p = n^{1-\ell}$ die Anzahl der Prozessoren ist.

Schritt 1:	$\mathcal{O}(1)$	
Schritt 4:	$\mathcal{O}(1)$	
Schritt 6:	$\mathcal{O}(n^\ell)$	
Schritt 7:	$\mathcal{O}(1)$	
Schritt 8:	$\mathcal{O}(n^\ell)$	
Schritt 9:	$\mathcal{O}(1)$	
Schritt 10:	$t(n^{1-\ell})$,	wobei $t(n)$ Laufzeit von ParallelSelect(n) ist.
Schritt 12:	$\mathcal{O}(n^\ell)$	
Schritt 13:	$\mathcal{O}(\log n^{1-\ell})$	
Schritt 16:	$\mathcal{O}(n^\ell)$	
Schritt 17:	$\mathcal{O}(1)$	
Schritt 19:	$\leq t(\frac{3}{4}n + \frac{n^{1-\ell}}{4})$	
Schritt 21–22:	$\leq t(\frac{3}{4}n + \frac{n^{1-\ell}}{4})$	

Die letzten beiden gelten wegen

$$S_< \leq n - \lceil \tfrac{n^{1-\ell}}{2} \rceil \cdot \lceil \tfrac{\lceil n^\ell \rceil + 1}{2} \rceil \leq \frac{3}{4}n + \frac{n^{1-\ell}}{4}.$$

Denn wenn m das $\lceil \frac{|M|}{2} \rceil$-kleinste Element von M ist, so gibt es $\lceil \frac{|M|}{2} \rceil$ Elemente in M, die größer als m sind, und für jedes dieser Elemente existieren mindestens $\lceil \frac{\lceil n^\ell \rceil - 1}{2} \rceil$ Elemente in S, die größer sind. Damit ergibt sich für geeignete Konstanten c_1, c_2:

$$t(n) \leq c_1 \cdot n^\ell + t(n^{1-\ell}) + t(\frac{3}{4}n + \frac{n^{1-\ell}}{4}) + c_2 \cdot \log n^{1-\ell}.$$

Wähle Konstante c_3 so, dass $c_2 \cdot \log n^{1-l} \leq c_3 \cdot n^\ell$ ist, und setze $c_4 := c_3 + c_1$; dann ergibt sich per Induktion über n:

$$\begin{aligned}
t(n) &\leq c_4 \cdot n^\ell + c \cdot (n^{1-\ell})^\ell + c \cdot (\frac{3}{4}n + \frac{n^{1-\ell}}{4})^\ell \\
&\leq c_4 \cdot n^\ell + c \cdot \frac{1}{n^{\ell^2}} \cdot n^\ell + c \cdot (\frac{3}{4}n)^\ell + c \cdot (\frac{n^{1-\ell}}{4})^\ell, \quad \text{da} \ell < 1 \\
&\leq n^\ell (\frac{c_4}{c} + \frac{1}{n^{\ell^2}} + (\frac{3}{4})^\ell + \frac{1}{4^\ell \cdot n^{\ell^2}}) \\
&\leq n^\ell (\frac{c_4}{c} + \frac{2}{n^{\ell^2}} + (\frac{3}{4})^\ell).
\end{aligned}$$

Für festes ℓ $(0 \leq \ell \leq 1)$ ist $(\frac{3}{4})^\ell = 1 - \varepsilon$, wobei $\varepsilon > 0$ ist. Das bedeutet, für geeignetes c und genügend großes n ist

$$\frac{c_4}{c} + \frac{2}{n^{\ell^2}} + (\frac{3}{4})^\ell \leq 1$$

und damit $t(n) \leq c \cdot n^\ell$.

Beispiel. Sei $n = 17$, $p = 4$, $k = 7$. **Gesucht:** das 7.-kleinste Element.

Vorgehen des Algorithmus.

$S := [24\ \ 3\ \ 12\ \ 21\ \ 7\ \ 6\ \ 13\ \ 20\ \ 23\ \ 14\ \ 1\ \ 4\ \ 15\ \ 16\ \ 2\ \ 22\ \ 19]$

7.

$$\begin{aligned} S^1 &= [24\ \ 3\ \ 12\ \ 21\ \ 19] \\ S^2 &= [7\ \ 6\ \ 13\ \ 20] \\ S^3 &= [23\ \ 14\ \ 1\ \ 4] \\ S^4 &= [15\ \ 16\ \ 2\ \ 22] \end{aligned}$$

8. $M = [19\ \ 13\ \ 14\ \ 16]$

9. $p' = 2$

10. $m = 14$

11.

$$\begin{aligned} S^1_< &= [3\ \ 12] & S^1_> &= [24\ \ 21\ \ 19] \\ S^2_< &= [7\ \ 6\ \ 13] & S^2_> &= [20] \\ S^3_< &= [14\ \ 1\ \ 4] & S^3_> &= [23] \\ S^4_< &= [2] & S^4_> &= [15\ \ 16\ \ 22] \end{aligned}$$

16.

$$\begin{aligned} S_< &= [3\ \ 12\ \ 7\ \ 6\ \ 13\ \ 14\ \ 1\ \ 4\ \ 2] \\ S_> &= [24\ \ 21\ \ 19\ \ 20\ \ 23\ \ 15\ \ 16\ \ 22] \end{aligned}$$

17. $|S_<| = 9$, $|S_>| = 8$, $s_1 = 3$, $(s_2 = 3)$

Aufruf von ParallelSelect$(S_<; 7; 3)$

7.

$$\begin{aligned} S^{1,1} &= [3\ \ 12\ \ 7] \\ S^{1,2} &= [6\ \ 13\ \ 14] \\ S^{1,3} &= [1\ \ 4\ \ 2] \end{aligned}$$

8. M^1=[7 13 2]

10. $m = 7$

11.

$$S_<^{11} = [3\ \ 7] \qquad S_>^{11} = [12]$$
$$S_<^{12} = [6] \qquad S_>^{12} = [13\ \ 14]$$
$$S_<^{13} = [1\ \ 4\ \ 2]$$

16.

$$S_<^1 = [3\ \ 7\ \ 6\ \ 1\ \ 4\ \ 2]$$
$$S_>^1 = [12\ \ 13\ \ 14]$$

20. $|S_<^1| = 6 < 7$, $k' = 7 - 6 = 1$

ParallelSelect mit $S_<^1 = [12\ \ 13\ \ 14]$ und $k = 1$ liefert sofort das Ergebnis:

Das gesuchte 7.-kleinste Element ist die 12.

8.8 Ein paralleler Algorithmus für das Scheduling-Problem für Jobs mit festen Start- und Endzeiten

Gegeben. n Jobs J_i mit Startzeiten s_i und Endzeiten t_i $(1 \leq i \leq n)$, o.B.d.A. $s_i \neq s_j$, $t_i \neq t_j$ für $i \neq j$.

Gesucht. Ein optimaler Schedule, d.h. eine Zuordnung der J_i auf einer minimalen Anzahl an Maschinen, so dass sich keine zwei Jobs auf derselben Maschine überlappen.

Sequentiell kann das Problem in $\mathcal{O}(n \log n)$ Zeit wie folgt gelöst werden:

1. Sortiere die s_1, t_1, s_2, $t_2, \ldots, s_n$, t_n in nichtabsteigender Reihenfolge, wobei t_j vor s_k kommt, falls $t_j = s_k$.
2. Schreibe sortierte Folge in Array U.
3. Setze $\ell := 1$ und $S[i] := i$ für $1 \leq i \leq n$.
4. Für $k = 1$ bis $2n$ führe aus:
5. Falls $U[k] = s_j$,
6. setze $M[j] := S[\ell]$, $S[\ell] := 0$ und $\ell := \ell + 1$;
7. ansonsten, falls $U[k] = t_j$
8. setze $S[\ell - 1] := M[j]$ und $\ell := \ell - 1$.

Dabei realisiert S einen Stack, auf dem die freien Maschinen liegen; $M[j]$ gibt an, welcher Maschine Job J_j zugeordnet wird.

Die parallele Version des Algorithmus besteht aus drei Phasen.

1. Phase. Die s_i, t_i werden mittels eines parallelen Sortieralgorithmus sortiert und in ein Array U der Länge $2n$ geschrieben. Für alle Jobs J_i wird parallel die Anzahl a_i der Maschinen berechnet, die unmittelbar nach der Startzeit von J_i belegt sind sowie die Anzahl b_i der Maschinen, die unmittelbar vor Beendigung von J_i belegt sind. Dazu wird ein Array L der Länge $2n$ angelegt mit

$$L[k] = \begin{cases} 1, & \text{falls } U[k] \text{ Startzeit} \\ -1, & \text{falls } U[k] \text{ Endzeit.} \end{cases}$$

Wenn nun $\sum_{j=1}^{k} L[j] = p_k$ für $1 \leq k \leq 2n$, dann ist

$$\begin{array}{lll} a_i & = & p_k, \qquad \text{falls } U[k] \text{ Startzeit } s_i \text{ und} \\ b_i & = & p_k + 1, \qquad \text{falls } U[k] \text{ Endzeit } t_i. \end{array}$$

2. Phase. Für alle parallelen Jobs J_i wird der unmittelbare Vorgänger auf derselben Maschine berechnet (falls er existiert). Dies ist gerade der Job J_ℓ, der als letzter Job vor bzw. zum Zeitpunkt s_i geendet hat und für den $a_i = b_\ell$ ist.

3. Phase. Durch List Ranking wird der erste Vorgänger eines jeden Jobs J_i auf derselben Maschine berechnet und den Jobs die entsprechende Maschinennummer zugeordnet.

Zum Schluss enthält $M[i]$ die Nummer der Maschine, der der Job J_i zugeordnet wird. Die Schritte 3. bis 20. entsprechen der ersten Phase, die Schritte 21. bis 26. der zweiten Phase und 27. bis 31. der dritten Phase. Der Algorithmus benötigt $\mathcal{O}(\log n)$ Zeit bei $\frac{n^2}{\log n}$ Prozessoren, ist also nicht kostenoptimal.

Formale Beschreibung des Algorithmus PARALLEL SCHEDULING **(Dekel & Sahni 1983)**

Eingabe: n Jobs J_i, Startzeiten s_i, Endzeiten t_i $(1 \leq i \leq n)$.

Ausgabe: $M[i]$ enthält Nummer der Maschine, der Job J_i zugeordnet wird.

Datenstruktur: Arrays U und L der Länge $2n$.

PARALLEL SCHEDULING

1. Für alle i $(1 \leq i \leq n)$ führe parallel aus:
2. Setze $U[i] := s_i$ und $U[n+i] := t_i$.
3. Für alle i, j $(1 \leq i, j \leq 2n)$ führe parallel aus:
4. Falls $U[i] < U[j]$, oder $U[i] = U[j]$ und $U[i]$ Endzeit, $U[j]$ Startzeit:
5. Setze $r_{ij} := 1$,
6. ansonsten
7. setze $r_{ij} := 0$.
8. Für alle j $(1 \leq j \leq 2n)$ führe parallel aus:
9. Berechne $\Pi(j) := \sum_{i=1}^{2n} r_{ij}$,
10. setze $U[\Pi(j)+1] := U[j]$.
11. Für alle k $(1 \leq k \leq 2n)$ führe parallel aus:
12. Falls $U[k]$ Startzeit:
13. Setze $L[k] := 1$,
14. ansonsten
15. setze $L[k] := -1$.
16. Berechne $p_k = \sum_{\ell=1}^{k} L[\ell]$.
17. Falls $U[k] = s_j$:
18. Setze $a_j := p_k$,
19. ansonsten, falls $U[k] = t_j$
20. setze $b_j := p_k + 1$.
21. Für alle i $(1 \leq i \leq n)$ führe parallel aus:
22. Finde k mit $t_k = \max\{t_\ell : t_\ell \leq s_i, b_\ell = a_i\}$.
23. Falls k existiert:
24. Setze VOR$[i] := k$,
25. ansonsten
26. setze VOR$[i] := i$.
27. Für alle $\ell = 1$ bis $\lceil \log n \rceil$ führe aus:
28. Für alle i $(1 \leq i \leq n)$ führe parallel aus:
29. Setze VOR$[i]$:= VOR[VOR$[i]$].
30. Für alle i $(1 \leq i \leq n)$ führe parallel aus:
31. Setze $M[i] := a_{\text{VOR}[i]}$.

Beispiel.

i	1	2	3	4	5	6	7	8
s_i	5	0	1	2	4	3	7	6
t_i	8	4	5	7	9	6	10	12

Vorgehen des Algorithmus.

2. $U = [5\ \ 0\ \ 1\ \ 2\ \ 4\ \ 3\ \ 7\ \ 6\ \ 8\ \ 4\ \ 5\ \ 7\ \ 9\ \ 6\ \ 10\ \ 12]$

5.

r_{ij} :

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	0	0	0	0	0	0	1	1	1	0	0	1	1	1	1	1
2	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
3	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1
4	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
5	1	0	0	0	0	0	1	1	1	0	1	1	1	1	1	1
6	1	0	0	0	1	0	1	1	1	1	1	1	1	1	1	1
7	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	1
8	0	0	0	0	0	0	1	0	1	0	0	1	1	0	1	1
9	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1
10	1	0	0	0	1	0	1	1	1	0	1	1	1	1	1	1
11	1	0	0	0	0	0	1	1	1	0	0	1	1	1	1	1
12	0	0	0	0	0	0	1	0	1	0	0	0	1	0	1	1
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
14	0	0	0	0	0	0	1	1	1	0	0	1	1	0	1	1
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

9. $\Pi = [7\ \ 0\ \ 1\ \ 2\ \ 5\ \ 3\ \ 11\ \ 9\ \ 12\ \ 4\ \ 6\ \ 10\ \ 13\ \ 8\ \ 14\ \ 15]$

7. $U = [0\ \ 1\ \ 2\ \ 3\ \ 4\ \ 4\ \ 5\ \ 5\ \ 6\ \ 6\ \ 7\ \ 7\ \ 8\ \ 9\ \ 10\ \ 12]$

9. $L = [1\ \ 1\ \ 1\ \ 1\ \ -1\ \ 1\ \ -1\ \ 1\ \ -1\ \ 1\ \ -1\ \ 1\ \ -1\ \ -1\ \ -1\ \ -1]$

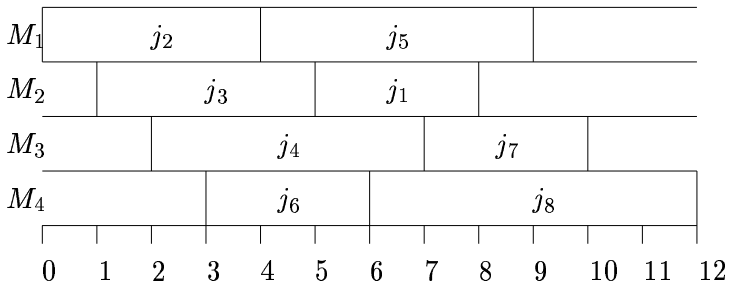
11.

a_j:	4	1	2	3	4	4	4	4
b_j:	4	4	4	4	3	4	2	1

14. $\textsc{Vor} = [3\ \ 2\ \ 3\ \ 4\ \ 2\ \ 6\ \ 4\ \ 6]$

15. $\ell = 1$.

19.



Literaturverzeichnis

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press and McGraw–Hill, 1990. ISBN 0-262-03141-8, 0-07-013143-0. Unibib KN: kid 112/c67, lbs 830/c67.

[Dij59] E. W. Dijkstra. A note on two problems in connection with graphs. *BIT*, 1:269–271, 1959.

[FIN93] András Frank, Toshihide Ibaraki, and Hiroshi Nagamochi. On sparse subgraphs preserving connectivity properties. *J. of Graph Theory*, 17:275–281, 1993.

[GR90] Alan Gibbons and Wojciech Rytter. *Efficient parallel algorithms.* Cambridge University Press, 1990. Unibib KN: kid 112/g41a, kid 112/g41.

[Jun94] Dieter Jungnickel. *Graphen, Netzwerke und Algorithmen.* BI-Wissenschaftsverlag, 1994. Unibib KN: kid 114/j96a(3), kid 604/j96, lbs 840/j96.

[Kru56] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7:48–50, 1956.

[NI92] Hiroshi Nagamochi and Toshihide Ibaraki. A Linear-Time Algorithm for Finding a Sparse k-Connected Spanning Subgraph of a k-Connected Graph. *Algorithmica*, 7:583–596, 1992.

[OW93] Thomas Ottmann and Peter Widmayer. *Algorithmen und Datenstrukturen.* B.I.–Wissenschaftsverlag, 1993. Unibib KN: kid 112/o98(2), lbs 830/o99.

[Pri57] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Tech. J.*, 36:1389–1401, 1957.

[SW94] Mechtild Stoer and Frank Wagner. A Simple Min Cut Algorithm. In Jan v. Leeuwen, editor, *Second European Symposium on Algorithms, ESA'94*, pages 141–147. Springer-Verlag, Lecture Notes in Computer Science, vol. 855, 1994. Unibib KN: kid 100:i/l22-855.

[Tar83] Robert E. Tarjan. *Data Structures and Network Algorithms.* SIAM, 1983. ISBN 0-89871-187-8. Unibib KN: kid 112/t17.