Eine Zusammenstellung aus Prüfungsprotokollen
bei Professor Verbeek mit vielen Fragen zur

Prüfungsvorbereitung
Diplomprüfung
Parallele Algorithmen

Thomas Schwarze
Thomas.Schwarze@FernUni-Hagen.de

2. März 2004

Fragen zu „Parallele Algorithmen“

1 Maschinenmodelle und Komplexitätsmaße

1.1 Welche Maschinenmodelle wurden im Kurs verwendet?

Die **RAM** **r**andom **a**ccess **m**achine und
die **PRAM** **p**arallel **r**andom **a**ccess **m**achine.

1.2 Woraus besteht eine RAM?

Die RAM besteht aus Seite 1:5

- Eingabeband mit Lesekopf
- Ausgabeband mit Schreibkopf
- potenziell unendliche Folge von Registern $r_0, r_1, r_2, \ldots$
- einem Programm
- einem Befehlszähler

sowie aus Lese-, Schreib-, Speicher-, Sprung- und Grundrechenbefehlen mit einer PASCAL ähnlichen Notation der Programme. Die RAM ist ein sequenzielles Modell.

1.3 Wie wird ein solches Programm dargestellt?

Ein Programm wird in PASCAL-ähnlicher Notation angegeben.

1.4 PASCAL bietet ja keine Möglichkeit zu parallelisieren. Wie macht man das hier?

Mittels der Anweisung *pardo* für „parallel do“ wird ein Befehl auf einer beliebigen Anzahl von Registern und Prozessoren innerhalb eines Taktes synchron ausgeführt.

`for` Bedingung `pardo` Anweisung;

1.5 Wie funktioniert eine PRAM?

Die PRAM besteht aus einer Folge von Prozessoren $p_1, \ldots, p_n$ und einem globalen Speicher. Separate Ein- und Ausgabebänder fallen weg. Deren Funktion werden durch bestimmte Zellen im globalen Speicher übernommen. Jeder Prozessor p_i kann eine natürliche Zahl speichern und kennt „seine“ Adresse i und kann daher über den Index bzw. die Adresse i angesprochen werden. Mit *Register* ist immer entweder eine globle Speicherstelle oder ein Prozessor gemeint. Seite 1:9

1.6 Wie läuft das in der Maschine ab?

Die RAM wird zur PRAM, indem die Register zu eigenständigen Prozessoren werden, so dass eine gleichzeitige Ausführung eines Befehls in mehreren Prozessoren erfolgen kann. Dabei ist wichtig, dass alle Prozessoren nach *einem* Takt synchron arbeiten und dabei je Takt nur einen Befehl ausführen aber unterschiedliche Daten (SIMD, **s**ingle **i**nstruction **m**ultiple **d**ata) verarbeiten können.

1.7 Bei Parallelität können gewisse Konflikte auftreten. Wie werden diese hier behandelt? Wie werden gleichzeitige Zugriffe mehrerer Prozessoren auf die gleiche Speicherzelle behandelt?

Parallele Prozessoren können dabei auf gemeinsame Register zugreifen. Konflikthandling wird unterschieden nach Seite 1:13

EREW PRAM Exclusive Read, Exclusive Write

CREW PRAM Concurrent Read, Exclusive Write

CRCW PRAM Concurrent Read, Concurrent Write

Bei der CRCW PRAM wird beim gleichzeitigen Schreiben auf dasselbe Register noch unterschieden nach

common CRCW PRAM: Gleichzeitiges Schreiben nur dann möglich, wenn die betreffenden Prozessoren denselben Wert in das entsprechende Register schreiben wollen.

arbitrary CRCW PRAM: Der Wert aus einem beliebigen Prozessor wird in das Register geschrieben.

priority CREW PRAM: Von den betreffenden Prozessoren schreibt derjenige mit der kleinsten Adresse.

1.8 Wozu benötigt man die verschiedenen PRAM-Typen?

Für eine Problemlösungen können unterschiedliche PRAM-Typen unterschiedliche Laufzeiten aufweisen. So ist die Maximumbildung in O(1) nur auf einer common CRCW PRAM möglich, da für diese Laufzeit gleichzeitige Lese- und Schreibzugriffe aller Prozessoren auf alle Register notwendig ist.

1.9 Welcher Algorithmus läuft so nur auf CRCW PRAM? Erläutern Sie den Algorithmus.

Die Maximumbildung in O(1) Zeit mit $O(n^2)$ Aufwand. Seite 2:19

Im ersten Schritt wird jedes x_i n-mal parallel gelesen. Im zweiten Schritt werden in O(1) Zeit die b_{ij}-Register über den Index i mittels UND verknüpft. Nur für die „Reihe“ i von Registern, in denen allen eine 1 steht, wenn also diese x_i das Maximum darstellt, wird in das Register m_i eine 1, in alle anderen $m_j, 1 \leq j \leq n, j \neq i$ Registern wird eine 0 geschrieben.

```
for all i, j pardo if x_i ≥ x_j  then  b_ij := 1
                                 else  b_ij := 0;
for all i pardo m_i := b_i1 ∧ b_i2 ∧ ... ∧ b_in;
```

1.10 Welche Komplexitätsmaße gibt es?

Seite 1:19

- Laufzeit $t(n)$ (time), Anzahl der Takte
- Aufwand $w(n)$ (work), Gesamtkosten der ausgeführten Befehle
- Prozessoren $p(n)$ (prozessor), Anzahl der Prozessoren

1.11 Welche Kostenmaße gibt es?

Seite 1:17

uniformes Kostenmaß: Anzahl der Zahlen bzw. Operanden unabhängig von der Länge der dafür notwendigen Darstellung bzw. Speicherung.

logarithmisches Kostenmaß: Die Länge der Operanden, i.d.R. in Binärdarstellung mit logarithmischer Länge, wird in die Komplexität mit einbezogen. Sinnvoll erst bei sehr, sehr großen Zahlen, bei der die Speicherung erheblichen Einfluss auf die Implementierung eines Algorithmus hat.

1.12 Was ist der Unterschied zwischen Laufzeit und Aufwand?

Seite 1:15

Die Laufzeit $t(n)$ bestimmt die Anzahl der Takte, in der das Programm abgearbeitet wurde. Der Aufwand $w(n)$ bezeichnet die Gesamtkosten aller auf allen Prozessoren $p(n)$ ausgeführten Befehle. Bei einer sequenziellen Ausführung auf einem Prozessor ist die Laufzeit identisch mit dem Aufwand $t(n) = w(n)$. Allgemein gilt $w(n) \leq p(n) \cdot t(n)$. Seite 1:20

1.13 Wie würden Sie diese Maschinenmodelle leistungsmäßig ordnen?

Es gilt folgende Rangliste (gem. Prüfungsprotokoll, steht nicht im Kurstext):

(1) priority CRCW

(2) arbitrary CRCW

(3) common CRCW

(4) CREW

1.14 Wie ergeben sich bei einer PRAM Laufzeit und Aufwand?

Laufzeit Maximale Anzahl von Takten, die die beteiligten Prozessoren zur Abarbeitung des Programms benötigen.

Aufwand Anzahl insgesamt ausgeführter Operationen.

Es gilt $w(n) \leq p(n) \cdot t(n)$. Seite 1:20

1.15 Was heißt „parallelisierbar“

Eine Funktion f heißt *parallelisierbar*, wenn es eine PRAM M und Konstanten $k, \ell > 0$ gibt, so dass M die Funktion f in $t(n) = \mathrm{O}(\log^k n)$ Zeit mit $w(n) = \mathrm{O}(n^\ell)$ Aufwand berechnet. Def. 1.2.6, Seite 1:20

„polylogarithmische Zeit und polynomialer Aufwand“.

4 Matrixoperationen

4.1 Kann man die Matrixmultiplikation beschleunigen?

Seite 4:1

Das Produkt zweier $n \times n$ Matrizen erfordert n^2 Multiplikationen von Vektoren mit jeweils n Komponenten, was bei naivem Vorgehen $\mathrm{O}(n^3)$ Operationen kostet. Mathe I, Kap. 5.1

Mit dem Verfahren von *Strassen* kann die Matrixmultiplikation in $\mathrm{O}(n^{\log 7})$ durchgeführt werden. Vermutet wird sogar, dass die Komplexität beliebig nahe an $\mathrm{O}(n^2)$ angenähert werden kann. Mathe III, Kap. 3.5

4.2 Wie funktioniert die Matrixmultiplikation nach Strassen?

Seite 4:1

Der Algorithmus nach Strassen bezieht sich auf quadratische Matrizen. Das ist aber keine Einschränkung, da die Zeilen und Spalten durch Auffüllen von Nullen entsprechnd vergrößert werden können. Beide $n \times n$ Matrizen A und B, für die $C = AB$ berechnet werden soll, werden in je vier $\frac{n}{2} \times \frac{n}{2}$ Matrizen aufgeteilt. Dann gilt

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Setzt man

$$\begin{aligned}
M_1 &:= (A_{11} + A_{22})(B_{11} + B_{22}), \\
M_2 &:= (A_{21} + A_{22})B_{11}, \\
M_3 &:= A_{11}(B_{12} - B_{22}), \\
M_4 &:= A_{22}(B_{21} - B_{11}), \\
M_5 &:= (A_{11} + A_{12})B_{22}, \\
M_6 &:= (A_{21} - A_{11})(B_{11} + B_{12}), \\
M_7 &:= (A_{12} - A_{22})(B_{21} + B_{22}),
\end{aligned}$$

so gilt für die Unterblöcke C_{ij}

$$\begin{aligned}
C_{11} &= M_1 + M_4 - M_5 + M_7, \\
C_{21} &= M_2 + M_4, \\
C_{12} &= M_3 + M_5, \\
C_{22} &= M_1 - M_2 + M_3 + M_6.
\end{aligned}$$

4.3 Wie kommt man auf die Komplexität beim Strassen-Verfahren und geht es noch besser?

Seite 4:2

Beim Strassen Algorithmus benötigt man pro Teilschritt 7 Multiplikationen und 18 Additionen bzw. Subtraktionen. Die Multiplikationen werden rekursiv auf dieselbe Weise ausgeführt, bis nur noch 4×4 Matrizen direkt berechnet werden. Das ergibt eine Komplexität von $\mathrm{O}(n^{\log 7})$ im sequenziellen Fall.

Auf einer EREW PRAM können die erforderlichen A_{ij}, B_{ij}, teilweise doppelten Matrizen in einem Schritt erzeugt und dann parallel multipliziert und in jeweils konstant vielen parallelen Schritten addiert werden, so dass im parallelen Fall eine Komplexität von $O(\log n)$ für $\log n$ Rekursionsschritte erreicht wird.

Das Strassen-Verfahren hat seit 1969 zur Entwicklung vieler weiterer schnellen Matrixmultiplikationsverfahren geführt. Dazu gehört der Schönhage-Strassen-Algorithmus. Seite 5:10

4.4 Worauf basiert der Schönhage-Strasse-Algorithmus (ganz grob)?

Der Schönhage-Strassen-Algorithmus basiert auf der schnellen Fourier Transformation FFT. Seite 5:10

4.5 Wie schnell kann man Matrizen invertieren?

Eine obere Dreiecksmatrix kann sequenziell in der gleichen Komplexität wie die Multiplikation ausgeführt werden, da die Inversion auf die Matrixmultiplikation zurückgeführt wird. Seite 4:5

Im parallelen Fall kann auf einer EREW PRAM die Inversion in $O(\log^2 n)$ Zeit durchgeführt werden, da $\log n$ Rekursionsstufen mit jeweils im Wesentlichen nur Matrixmultiplikationen durchgeführt werden müssen. Seite 4:19

Inverse einer beliebigen Matrix (falls sie existiert), kann in $O(\log^2 n)$ Zeit auf einer EREW PRAM berechnet werden.

5 Fourier Transformation und Polynomoperationen

5.1 Können sie sich vorstellen, welcher der im Kurs behandelten parallelen Algorithmen in der Praxis besonders häufig zur Anwendung kommt?

Die schnelle Fourier Transformation (FFT).

5.2 Was versteht man unter FFT?

Die schnelle Fourier Transformation (FFT) ist die Zurückführung der Diskreten Fourier Transformation (DFT) der Größe $n = 2m$ auf zwei DFT-Probleme der Größe m mit dazugehörigen Datumumordnungen. Sie hat im sequenziellen Fall eine Zeitkomplexität von $O(n \log n)$. Dadurch ist die *Polynommultiplikation* sogar einschließlich Transformation in Stützstellendarstellung und Interpolation bzw. Rücktransformation noch schneller als die direkte Multiplikation in Koeffizientendarstellung mit $O(n^2)$. Für die FFT gilt: Transformation + einfache Multiplikation + Interpolation $= O(n \log n) + O(n) + O(n \log n) < O(n^2)$.

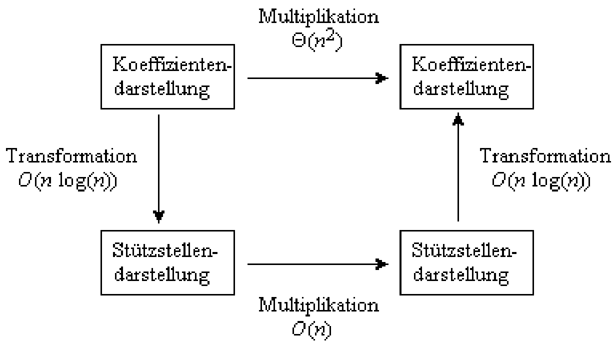


Abbildung 5.1: Polynommultiplikation mit der FFT

Die DFT ist eine Matrix-Vektor Multiplikation $b = F \cdot a$, mit a, b als Spaltenvektoren und F als Fouriermatrix in einem Ring. Die $n \times n$ Fouriermatrix F besteht aus $F_{i,j} = \omega^{i \cdot j}$, wobei ω die primitive n-te Einheitswurzel darstellt.

Sei $\mathbb{C}$ der Ring und $n = 4$, dann ist i die n-te primitive Einheitswurzel. Dann ist die Fouriermatrix:

$$F = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{pmatrix}$$

Die Fouriertransformation des Vektors $a = (1,1,1,0)^T$ ergibt $b = F \cdot a = (3, i, 1, -i)^T$.

Die Matrix-Vektor Multiplikation ist eine *Polynomauswertung* an n Stellen gleichzeitig, d.h. eine Auflösung des Polynoms $p(x)$ an den Stellen $x_0, \ldots, x_{n-1}$. Wenn alle Stellen $x_i (i = 0, \ldots, n-1)$ verschieden sind, ist eine Matrix X invertierbar, so dass die Koeffizienten $a_0, \ldots, a_{n-1}$ des Polynoms zurückgewonnen werden können. Für bekannte $x_0, \ldots, x_{n-1}$ und $y_0, \ldots, y_{n-1}$ errechnet sich damit $a = X^{-1} \cdot y$, mit a, y als Spaltenvektoren. Diese Invertierbarkeit ist bei der Fouriermatrix F gegeben.

Die FFT führt die einzelnen Berechnungen der Matrix-Vektor Multiplikation $b = F \cdot a$ in einer speziellen Reihenfolge aus, so dass jeweils auf schon berechnete Zwischenergebnisse zurückgegriffen wird. Sei ω_n primitive n-te Einheitswurzel. Dann wird ausgenutzt, dass $\omega^{\frac{n}{2}} = -1$ und $\omega_n^2 = \omega_{\frac{n}{2}}$ gilt. Dazu muss n eine Zweierpotenz sein.

Als erstes werden die Komponenten von b mit *geradem* Index berechnet, indem die entsprechenden Zeilen der Fouriermatrix mit dem Vektor a multipliziert werden:

$$b_{2m} = \sum_{k=0}^{n-1} \omega^{2mk} a_k = \sum_{k=0}^{\frac{n}{2}-1} \omega^{2mk} (a_k + a_{\frac{n}{2}+k}) \quad \forall m < \frac{n}{2}$$

weil $\omega^{2m(\frac{n}{2}+k)} = \omega^{mn} \cdot \omega^{2mk} = \omega^{2mk}$.

Analog berechnet man die Komponenten mit *ungeradem* Index:

$$b_{2m+1} = \sum_{k=0}^{n-1} \omega^{(2m+1)k} a_k = \sum_{k=0}^{\frac{n}{2}-1} \omega^{2mk} \omega^k (a_k - a_{\frac{n}{2}+k}) \quad \forall m < \frac{n}{2}$$

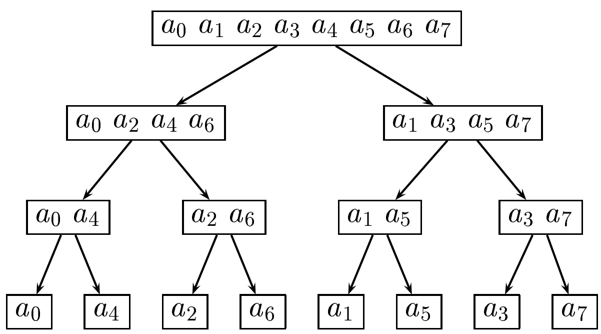
weil $\omega^{(2m+1)(\frac{n}{2}+k)} = \omega^{mn} \cdot \omega^{\frac{n}{2}} \cdot \omega^{(2m+1)k} = -\omega^{(2m+1)k}$.

Somit gilt:

$$\begin{aligned} (b_0, b_2, b_4, \ldots, b_{n-2})^T &= F_{\frac{n}{2}}(s_0, s_1, s_2, \ldots, s_{\frac{n}{2}-1})^T \\ (b_1, b_3, b_5, \ldots, b_{n-1})^T &= F_{\frac{n}{2}}(d_0, d_1, d_2, \ldots, d_{\frac{n}{2}-1})^T \end{aligned}$$

wobei $s_k := a_k + a_{\frac{n}{2}+k}$ und $d_k := \omega^k (a_k - a_{\frac{n}{2}+k})$ gesetzt ist.

Das Schema der input-Vektoren für die rekursive Aufrufe bei Berechnung von FFT_8 sieht als Beispiel wie folgt aus:



Man beachte, dass die Bläter dieses Baumes mit den Elementen des ursprünglichen input-Vektors

$$a = (a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7)$$

in permutierter Reihenfolge

$$a^{\text{rev}} = (a_0, a_4, a_2, a_6, a_1, a_5, a_3, a_7)$$

beschriftet sind. Dabei fällt auf, dass die Indizes der Komponenten in Basis-2 Darstellung bei a^{rev} rückwärts gegenüber a erfolgt: An Stelle von a_3 in a steht also a_6 in a^{rev}, denn $3 = (011)_2$ und $6 = (110)_2$. Diese Regel gilt allgemein und wird als *bit reversal* bezeichnet.

Da durch diesen *Divide and Conquer* Algorithmus eine parallele Verarbeitung ermöglicht wird, reduziert sich die Zeitkomplexität von $\mathrm{O}(n \log n)$ auf $\mathrm{O}(\log n)$. Der Aufwand beträgt $\mathrm{O}(n \log n)$. Die parallele FFT ist auf einer EREW PRAM ausführbar.

Die Interpolation, die bei de *Polynommultiplikation* notwendig ist, um die FFT und anschließende Multiplikation wieder in die gewünschte Koeffizientendarstellung zu bringen, ist wieder eine Matrix-Vektor-Multiplikation. Unter erneuter Ausnutzung der besonderen Eigenschaften der n-ten Einheitswurzeln kann auch diese Berechnung mittels der FFT in $\mathrm{O}(n \log n)$ Zeit ausgeführt werden.

5.3 Wie ist bei der FFT die Zeitkomplexität im sequenziellen Fall?

Matrix-Vektor-Multiplikation: $\mathrm{O}(n)$

Polynommultiplikation: $\mathrm{O}(n^2)$

5.4 Wie ist bei der FFT die Zeitkomplexität im parallelen Fall?

Auf einer EREW PRAM jeweils

Matrix-Vektor-Multiplikation: $\mathrm{O}(\log n)$ Zeit und $\mathrm{O}(n \log n)$ Aufwand

Polynommultiplikation: $\mathrm{O}(\log n)$ Zeit und $\mathrm{O}(n \log n)$ Aufwand

Polynomdivision: $\mathrm{O}(\log^2 n)$ Zeit und $\mathrm{O}(n \log n)$ Aufwand

6 Sortieren und Suchen

6.1 Welche schnellen Sortierverfahren gibt es?

Quicksort, Mergesort und Heapsort z.B.

6.2 Kann Sortieren parallelisiert werden?

Durch einen Divide and Conquer Algorithmus ist das Teilen problemlos parallelisierbar. Für das parallele Zusammenfügen, das „Mergen“, müssen jedoch Besonderheiten beachtet werden.

6.3 Wie funktioniert Mergesort?

Mergesort arbeitet rekursiv, indem die zu sortierende Menge jeweils halbiert, sortiert und wieder zusammengefügt wird. Dieses Zusammenfügen erfolgt über zwei Zeiger, die sequenziell durch die beiden sortierten Teillisten laufen und die entsprechenden Elemente paarweise vergleichen. Die Ergebnisliste wird in ein neues Array geschrieben. Das Mergen hat eine Komplexität von $O(n)$. Da das Mergen in jeder Rekursionsstufe erfolgt, hat der sequenzielle Sortieralgorithmus eine Komplexität von $O(n \log n)$

Im einfachsten parallelen Fall wird der Merge-Schritt in $\frac{n}{\log n}$ viele Teile aufgeteilt, die dann sequenziell abgearbeitet werden. Eine der beiden Listen mit Länge n wird dabei in $\log n$ große Teile geteilt. Das kleinste Element eines jeden Blocks wird dann in der ersten Liste binär gesucht, was $O(\log n)$ Zeit kostet, so dass die erste Liste „passend“ zur zweiten unterteilt wird. Das Zusammenfügen der so zusammengehörenden Blöcke von jeweils $\log n$ bzw. $\log m$ Länge erfolgt dann sequentiell, so dass für das Mergen insgesamt $O(\log n)$ Zeit erforderlich ist. Zusammen mit der Rekursion für die Aufteilung in Blöcke ergibt sich für einfaches paralleles Mergesort eine Komplexität von $O(\log^2 n)$.

1. Verbesserung:
Im Folgenden sei $n = 2^k,\ k \in \mathbb{N}$ stets eine Zweierpotenz. Um ein paralleles Mergen zu ermöglichen, muss von jedem Element x der Rang bekannt sein.

Rang $\text{Rang}(x : A)$ eines beliebigen Elements x und einer Menge A bezeichnet die Anzahl der Elemente von A, die kleiner als x sind: $|\{y\ :\ y < x, y \in A\}|$. Ferner sei $\text{Rang}(B : A) = (r_1, \ldots, r_m)$ mit $r_i = \text{Rang}(b_i : A)$. Vorausgesetzt ist eine Ordnungsrelation auf x, A und B.

Anmerkung: Sei C die zu sortierende Menge, die in die Mengen A und B aufgeteilt wird, so dass gilt: $A \subseteq C, B \subseteq C, A \cap B = \emptyset, A \cup B = C$. Dann ist $\text{Rang}(x : C) = \text{Rang}(x : A) + \text{Rang}(x : B)$. Parallel kann somit für alle x der Rang in $O(1)$ berechnet werden, wenn $\text{Rang}(B : A)$ und $\text{Rang}(A : B)$ bekannt sind.

Sei A mit $n = |A|$ eine sortierte und B mit $m = |B|$ eine beliebige Folge mit $m \leq n$. Dann können $p \approx \frac{m}{n}$ Prozessoren für jedes $b_j \in B, j = 1, \ldots, m$ in $O(1)$ dieses b_j mit allen Elementen

$a_i \in A, i = 1, \ldots, n$ auf $a_y < b_j < a_{y-1}$ vergleichen. Der Aufwand beträg $\mathrm{O}(m \cdot p) = \mathrm{O}(m \cdot \frac{m}{n}) = \mathrm{O}(n)$.

Das weitere Vorgehen für beide sortierte Folgen A und B ist ähnlich dem des einfachen Algorithmus. Die Partitionierung von B erfolgt jedoch in rund $\sqrt{m}$ Blöcken von $\approx \sqrt{m}$ Länge. Bei Einsatz von $p = \sqrt{n}$ Prozessoren kann Rang$(B : A)$ rekursiv für jedes b_j in $\mathrm{O}(1)$, somit für B in $\mathrm{O}(\log m)$ und über alle Rekursionsstufen in $\mathrm{O}(\log\log m)$ errechnet werden.

Somit ist das Mergen von zwei sortierten Folgen der Länge n in $\mathrm{O}(\log\log n)$ Zeit und $\mathrm{O}(n \log\log n)$ Aufwand möglich.

2. Verbesserung:
Um den Aufwand weiter zu reduzieren, werden die beiden Folgen A und B jetzt in Blöcke der Größe $\log\log n$ zerlegt. Das Mergen erfolgt nicht mehr über Folgen der Länge n sondern der Länge $\frac{n}{\log\log n}$. Dadurch verringert sich der Aufwand auf $\mathrm{O}(n)$. Man erhält $\mathrm{O}\left(\frac{n}{\log\log n}\right)$ unabhängige Merging-Probleme der Größe $\mathrm{O}(\log\log n)$, die parallel in $\mathrm{O}(\log\log n)$ Zeit und mit $\mathrm{O}\left(\frac{n}{\log\log n}\log\log n\right) = \mathrm{O}(n)$ Aufwand gelöst werden können.

Da diese Algorithmen nur eine Merging-Stufe durchführen, kommt für den vollständigen Sortieralgorithmus bei allen Merging-Verfahren noch die rekursive Teilung der Folgen hinzu, was jeweils in $\mathrm{O}(\log n)$ Zeit und Aufwand erfolgt.

Zusammenfassung
Die Mergesort-Algorithmen für n Elemente haben folgende Komplexitäten:

sequenziell: $\mathrm{O}(n \log n)$

Sort mit einfachem parallelen Merge: $\mathrm{O}(\log^2 n)$ Zeit, $\mathrm{O}(n \log n)$ Aufwand auf einer CREW PRAM

Sort mit schnellem parallelen Merge: $\mathrm{O}(\log n \log\log n)$ Zeit, $\mathrm{O}(n \log n \log\log n)$ Aufwand auf einer CREW PRAM

Sort mit schnellem speicheroptimierten parallelen Merge: $\mathrm{O}(\log n \log\log n)$ Zeit, $\mathrm{O}(n \log n)$ Aufwand auf einer CREW PRAM

6.4 Wie kann man beim Sortieren eine gegebene Folge in zwei Teilfogen aufteilen?

Bei Mergesort wird die Folge in zwei gleichgroße Teilfolgen aufgeteilt. Beim Quicksort ist die Größe der beiden Teilfolgen von dem Schlüsselelement abhängig.

6.5 Wie kann man beim Merge-Schritt zwei Folgen verbinden?

Sequenziell in $\mathrm{O}(n)$ Zeit. Parallel in $\mathrm{O}(\log\log n)$ Zeit durch Aufteilung der beiden Teilfolgen in $\log\log n$ große Blöcke und Mergen dieser Blöcke in $\frac{n}{\log\log n}$ parallelen Prozessen, indem von je zwei Folgen der Rang der einen Folge in der anderen bestimmt wird.

Die kürzere Folge B wird in $\approx \log\log n$ große Teile aufgeteilt. Mittels Rangvergleich wird die zweite (längere) Folge A synchron dazu partitioniert. Die so entstandenen kurzen, synchronen Folgen aus A und B werden sequenziell zusammengefügt.

6.6 Auf welche Werte für Zeit und Aufwand kommt man beim Teilen für die Sortierung bzw. beim Mergen schließlich?

Die Mergesort-Algorithmen für n Elemente haben folgende Komplexitäten:

sequenziell: $\mathrm{O}(n \log n)$

Sort mit einfachem parallelen Merge: $\mathrm{O}(\log^2 n)$ Zeit, $\mathrm{O}(n \log n)$ Aufwand auf einer CREW PRAM

Sort mit schnellem parallelen Merge: $\mathrm{O}(\log n \log\log n)$ Zeit, $\mathrm{O}(n \log n \log\log n)$ Aufwand auf einer CREW PRAM

Sort mit schnellem speicheroptimierten parallelen Merge: $\mathrm{O}(\log n \log\log n)$ Zeit, $\mathrm{O}(n \log n)$ Aufwand auf einer CREW PRAM

6.7 Wie kommt die untere Grenze beim Aufwand für Merge-Sort zustande?

Da es bei n Elementen $n!$ Permutationen gibt, die in einem Entscheidungsbaum den Blättern zugeordnet werden können, existieren in diesem Baum somit $n!$ Blätter. Eine Abschätzung zeigt: $n! \geq \left(\frac{n}{2}\right)^{\frac{n}{2}}$ und damit $\log n! \geq \frac{n}{2} \log \frac{n}{2} = \Omega(n \log n)$.

6.8 Kann man Quicksort parallelisieren?

Ja, ähnlich wie Mergesort, wird aber nicht im Kurs beschrieben!

8 Schwer parallelisierbare Probleme

8.1 Stichwort NP, P: Sind alle berechenbaren Funktionen parallelisierbar?

Nein, vermutlich noch nicht einmal alle entscheidbaren Funktionen.

Eine Funktion f heißt parallelisierbar, wenn es eine PRAM M und Konstanten $k, \ell > 0$ gibt, so dass M die Funktion f in $t(n) = \mathrm{O}(\log^k n)$ Zeit mit $w(n) = \mathrm{O}(n^\ell)$ Aufwand berechnet. Def. 1.2.6, Seite 1:20

Die Funktion $f(x) = 2^x$ ist berechenbar, liegt aber nicht mehr in P und ist daher nicht parallelisierbar.

Bezüglich der Klassen NC (Nicks Class, die Klasse aller parallelisierbaren Funktionen) und P gilt NC $\subseteq$ P. Es wird vermutet, dass sogar NC $\subset$ P gilt. Das würde bedeuten, dass P-vollständige Probleme nicht parallelisierbar sind

8.2 Warum wird zur P-Vollständigkeit logarithmische Bandreduzierbarkeit verlangt?

Weil jede Funktion f, die mit logarithmischem Platz berechnet werden kann, parallelisierbar ist.

In der Regel wird die P-Vollständigkeit über polynomielle Zeitreduzierbarkeit definiert. Da für zwei beliebige Sprachen L, M gilt: $L \leq_{\log} M \Rightarrow L \leq_{pol} M$, kann zum Nachweis der P-Vollständigkeit von L auch logarithmische Bandreduzierbarkeit genutzt werden. Die Umkehrung gilt im Allgemeinen jedoch nicht.

Bei der Diskussion paralleler Probleme ist zu beachten, dass jede Funktion f, die mit logarithmischem Platz berechnet werden kann, parallelisierbar ist. Für polynomiell zeitbeschränkte Funktionen fehlt der Nachweis, so dass P $\neq$ NC angenommen wird. Es könnte sein, dass eine Sprache A polynomzeitreduzierbar auf eine bekannte parallelisierbare Sprache B wäre, ohne tatsächlich parallelisierbar zu sein.

8.3 Was hat log-Bandreduzierbarkeit mit Parallelisierbarkeit zu tun?

Sind A,B Sprachen und ist die Sprache A $\in$ NC. Wenn B logarithmisch bandreduzierbar auf A ist (B $\leq_{\log}$ A), dann ist auch B $\in$ NC.

9 Sonstige Bereiche des Skripts, die aber noch nie abgefragt wurden

2.3 Präfixsumme

2.4 Beschleunigungstechnik

3.1 List-Ranking

3.2 Eulerkreis Technik

3.4 Baumkontraktion (Rake)

3.5 Auswertung arithmetischer Ausdrücke

5.4 Toeplitz-Matritzen

6.5 Suche des k-ten Elements

7.1 Konvexe Hülle

7.2 Durchschnitt von Hüllkurven

10 Auflistung der Komplexitätsklassen

	Sequenziell	Parallel		Bemerkung
Angaben in O-Notation	: O()	Zeit O()	Aufwand O()	
Maximumbildung	:	1	n^2	common CRCW
Matrixmultiplikation	: n^3			naiv
$\sim$ nach Strassen	: $n^{log 7} = \log^{2,82}$	$\log n$		EREW
Invertierung einer Matrix	:	$\log^2 n$		EREW
Fast-Fourier-Transformation	: $n \log n$	$\log n$	$n \log n$	EREW
Matrix-Vektor-Multiplikation	: n	$\log n$	$n \log n$	EREW
Polynommultiplikation	: n^2	$\log n$	$n \log n$	EREW mit FFT
Polynomdivision	: n^2	$\log^2 n$	$n \log n$	EREW
Mergesort	: $n \log n$			
$\sim$ mit $\log n$ großen Teilen	:	$\log^2 n$	$n \log n$	CREW
$\sim$ mit $\sqrt{n}$ großen Teilen	:	$\log n \log \log n$	$n \log n \log \log n$	CREW
$\sim$ mit $\log \log n$ großen Teilen	:	$\log n \log \log n$	$n \log n$	CREW
Mergeschritt	: n	$\log \log n$		CREW
minimaler Sortieraufwand	: $\Omega(n \log n)$ entspricht sequenzieller Baumhöhe bei $n!$ Permutationen			