Abschlussbericht
Proseminar Algorithmen
Vortrag :Parallele Sortierung

Patrick Winterstein,Björn Rathjen

SS14

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einführung

Der Titel Paralleles Sortieren beschreibt zwei grundsätzliche Bestrebungen bei der Optimierung von Netzwerk- (hauptsächlich Routing) und Datenstrukturen (Listen, Wörterbücher, ...). Somit ist die Motivation dieses Thema zu behandeln groß, da sie für jene Projekte die Basis für deren Geschwindigkeit legen kann. Separat betrachtet sind diese von folgender Bedeutung :

Sortierung ist die Basis für jede Suche auf Daten und Ressourcen. Wenn diese nicht wenigstens einer groben Ordnung unterliegen, können effiziente Algorithmen nicht angewendet werden, da keine Annahmen über den Zustand getroffen werden können.

Parallelität stellt eine hohe Optimierung dar, da durch den Aufwand von mehr Ressourcen ein Sortierprozess beschleunigt werden kann, ohne die Korrektheit des Ergebnisses zu gefährden. Der Datendurchsatz wird dabei maximiert.

2 Grundlagen

2.1 Sortieren

Als Grundlage um eine Menge M sortieren zu können muss eine totale Ordnungsrelation $R \subseteq MxM$ vorhanden sein, diese schreibt eine Ordnung zwischen zwei Elementen vor. Der Anspruch an diese Relation ist, dass diese Reflexivität

$$\forall x \in M : xRx \tag{1}$$

und Transitivität beinhaltet.

$$\forall x, y, z \in M : xRy \wedge yRz \Rightarrow xRz \tag{2}$$

Des weiteren muss diese antisymmetrisch

$$\forall x, y \in M : \; xRy \wedge yRx \; \Rightarrow \; x = y \tag{3}$$

und total

$$\forall x, y \in M : \; xRy \vee yRx \tag{4}$$

sein.

2.2 Komparator

Ein Komparator stellt den kleinsten Baustein dar, der eine zweielementige Menge entsprechend der auf ihr liegenden Ordnungsrelation sortiert ausgibt. Dieser ist wie folgt aufgebaut: Er besitzt zwei Eingangsleitungen auf dem die zu sortierenden 2-elementigen Mengen eingegeben werden, den vergleichenden Teil, der die Ordnungsrelation anwendet und die beiden Ausgangsleitungen, auf denen das sortierte Ergebnis ausgegeben wird. Beide werden in Abbildung 2 (Software, Seite 5) und Abbildung 1 (Hardware, Seite 4) dargestellt. Die Annahme für die folgenden Abbildungen ist, dass das sortierte Ergebnis von oben nach unten größer wird.

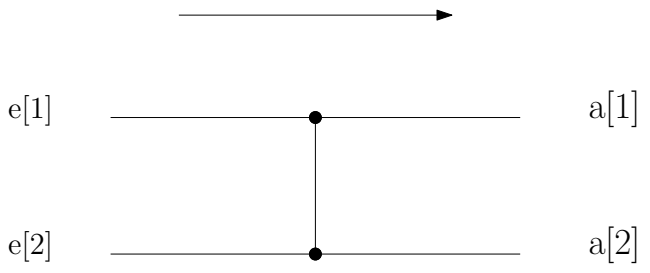

Abbildung 1: Schema eines Komparators. e entspricht den Eingängen und a den Ausgängen. Der Pfeil gibt die Durchlaufrichtung an. Die waagerechten schwarzen Linien entsprechen den Datenleitungen, die senkrechte Linie zeigt die vergleichenden Teil an.

```
1   void comp(chan in1, in2, out1, out2 Comparer{}){
2       a := <- in1
3       b := <- in2
4
5       if (a < b){
6           out1 <- a
7           out2 <- b
8           return void
9       }
10      out1 <- b
11      out2 <- a
12      return void
13  }
```

Abbildung 2: Implementierung eines Komparators in Pseudocode (An Go angelehnt). inX entspricht den Eingangsleitungen, outX entspricht den Ausgangsleitungen, Comparer besagt, dass der Datentyp die beschriebenen Voraussetzungen ((1) bis (4)) erfüllt. „return void“ zeigt, dass die Funktion nur die Ein- und Ausgänge verwendet.

3 Sortiernetzwerk

Die im vorherigen Abschnitt beschriebenen Voraussetzungen dienen nun als Grundlage dafür ein größeres Netzwerk aufzubauen, das folgende Eigenschaften besitzt:

- mehr als 2 Eingabeleitungen
- Ausgabe soll sortiert sein

Das Netzwerk soll zusätzlich nur aus Komperatoren bestehen. Es wird zunächst noch kein Anspruch an Komplexität und Effizienz erhoben.

3.1 Aufbau

Ein naiver Ansatz für den Aufbau des Netzwerks wird in Abbildung 3 (Seite 6) gezeigt. Die Anzahl der Leitungen wurde verdoppelt und die Anzahl der Komparatoren angepasst. In Abbildung 4 (Seite 6) wird der Vollständigkeit halber ein größeres Zahlenbeispiel gezeigt. Die Sortierung der vorher unsortierten Eingabe kann schrittweise verfolgt und überprüft werden.

3.2 Korrektheit

3.2.1 Betrachtung der Analyse

Problematisch bei der Überprüfung durch Beispiele ist die große Anzahl an zu überprüfenden Fälle. Im Beispiel in Abbildung 4 (Seite 6) bei einer Eingabemenge mit den Elementen 1 bis 8 wären dies $8! = 40320$ unterschiedliche Fälle (diese Anzahl ist durch den Ausschluss, dass

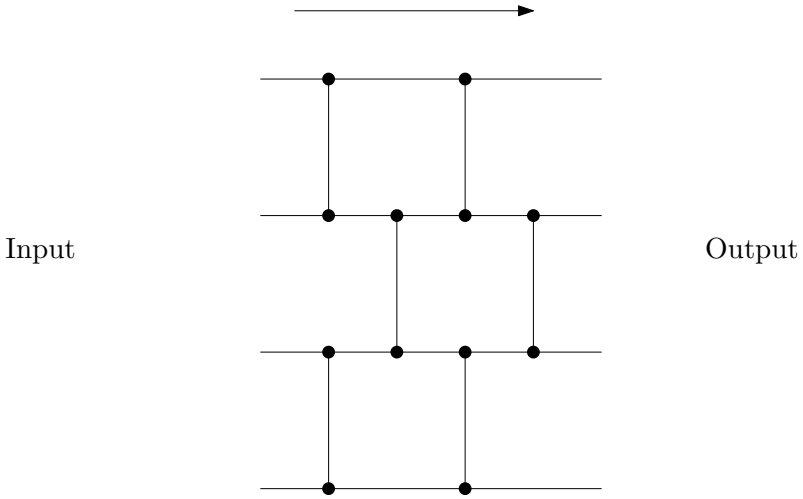

Abbildung 3: Komparatornetzwerk mit mehreren Datenleitungen

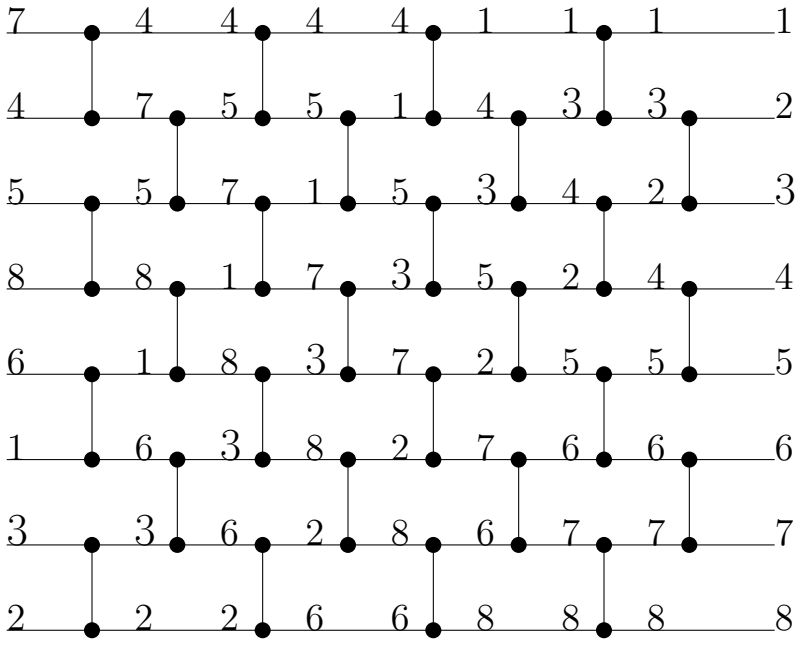

Abbildung 4: Beispiel mit acht Datenleitungen, das einen schrittweisen Verlauf des Sortierens zeigt, und eine vollständig sortierte Ausgabe.

eine Zahl mehrfach vorkommt, bereits reduziert worden). Allgemein ist die Anzahl $^{n!}/_{(n-l)!}$, wobei n die möglichen Elemente und l die Anzahl der Datenleitungen ist. Nachfolgend wird nun eine einfachere Möglichkeit gezeigt, wie die Korrektheit überprüft werden kann.

3.2.2 0,1-Prinzip

Das 0,1-Prinzip ist ein Werkzeug zur Überprüfung, ob ein Sortiernetzwerk alle Eingaben richtig sortiert. Dabei wird die Menge von Testfällen reduziert, indem man die Eingabemenge auf die Zeichen 0 und 1 beschränkt.

Lemma Wenn es eine Folge A gibt, die ein Sortiernetzwerk nicht sortiert, so existiert auch eine 0,1-Folge, die von diesem Netzwerk nicht sortiert wird.

Beweis Geführt wird ein Beweis durch Widerspruch.

i) Annahme : Wenn ein Sortiernetzwerk / Algorithmus eine Eingabe nicht sortiert, so sortiert er trotzdem alle 0,1-Folgen.

ii) Voraussetzungen :

- Eingabefolge $E = e_0 \dots e_l$
- sortiere Eingabefolge $S = s_0 \dots s_l$
- falsch sortierte Ausgabefolge von E $U = u_0 \dots u_l$
- kleinster Index an dem $u_k \neq s_k$

iii) Folgerungen aus ii) :

$$u_i = s_i \qquad \forall\ 0 \leq i < k \tag{5}$$

$$u_r = s_k \qquad \text{für ein } r > k \tag{6}$$

iv) Funktion, die aus E eine 0,1-Folge erstellt. Der Schwellwert, der dazu verwendet wird, ist in diesem Fall s_k.

$$f(i) = \begin{cases} 0, & if\ \ e_i \leq s_k \\ 1, & if\ \ e_i > s_k \end{cases} \tag{7}$$

v) Wendet man das Sortiernetzwerk nun auf die konstruierte Folge an, so entsteht eine 0,1-Folge der Form

$$00 \dots 01_k \dots 0_r \dots \tag{8}$$

vi) Folgerung :
⇒ die Folge ist unsortiert.
⇒ Widerspruch zur Annahme

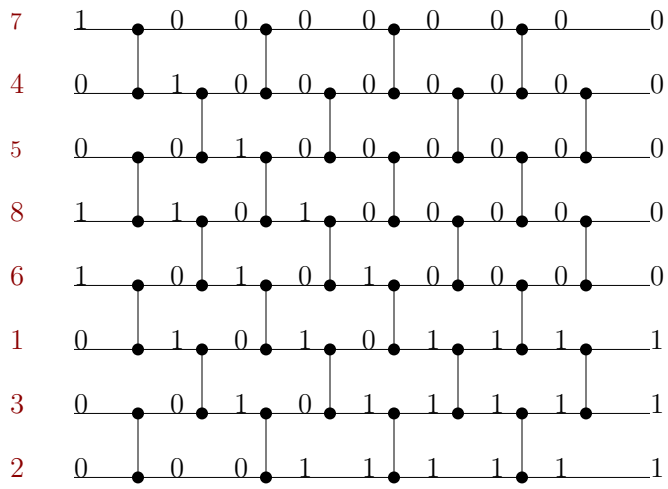

Abbildung 5: Beispiel für das 0,1-Prinzip. Folge wird auf das Beispiel aus Abbildung 4 (Seite 6) angewendet. Es zeigt, dass das Resultat für die gewählte Konstante sortiert ist. Genauso zeigt es auch, dass bis Schritt 5 die Folge nicht sortiert ist.

Vorteile Durch das Anwenden des 0,1-Prinzips wird die Anzahl der Testfälle von

$$n! \rightarrow 2^l \tag{9}$$

reduziert. Daraus resultiert, dass mit größerer Sicherheit und effektiver getestet werden kann. In einem Beispiel in Abbildung 5 (Seite 8) wird die Zuweisung mit $s_k = 6$ und die anschließende Sortierung demonstriert.

3.3 Übertragung von Sortieralgorithmen auf ein Sortiernetzwerk

3.3.1 Bubblesort

Netzwerkimplementierung (Abbildung 6 Seite 9) entspricht der Softwareimplementierung. Die Datenleitungen entsprechen den Indizes einer Liste von oben nach unten. Die Vergleiche folgen dem Algorithmus, Optimierungen in Form von Parallelen Vergleichen wurden bereits vollzogen.

3.3.2 Quicksort , Mergesort

Beim Versuch schnellere Algorithmen zu implementieren entstehen durch direkte Umwandlung Probleme.

Quicksort Bei Quicksort wird in jedem Schritt ein Pivotelement gewählt nach dem die übrigen Werte auf zwei Listen aufgeteilt werden. Dies stellt in einem starren Netzwerk ein Problem dar, da die Größen dieser Listen von der Eingabe abhängen, und somit nach dem ersten Vergleich die Position des Pivotelemnts nicht mehr festgelegt ist.

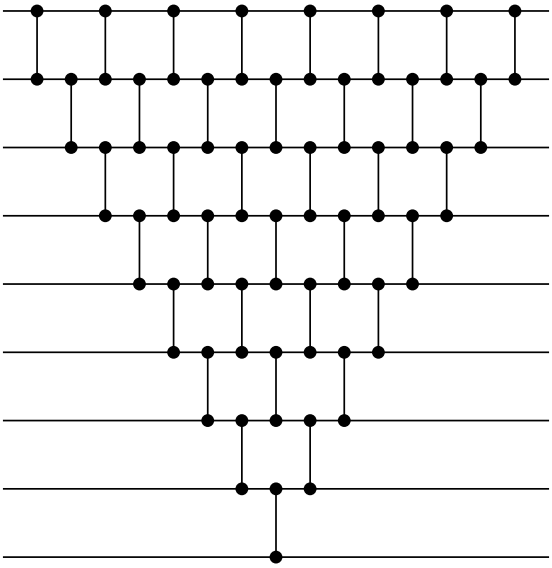

Abbildung 6: Bubblesort: Bubblesort mit parallelen Vergleichen.

Mergesort Bei Mergesort wird die Eingabe in maximal 2-elementige Listen unterteilt, der danach dynamische merge-Prozess lässt sich nicht 1:1 auf ein starres Netzwerk übertragen

Dies wird in Abbildung 7 (Seite 9) verdeutlicht.

3.4 Biton-Sortierer

Der Biton-Sortierer ist ein Sortierverfahren, welches mit dem Teile-und-Hersche-Prinzip eine Menge sortiert.

Abbildung 7: Auf der linken Seite ist der Versuch Quicksort, auf der rechten Seite Mergesort zu implementieren zu sehen

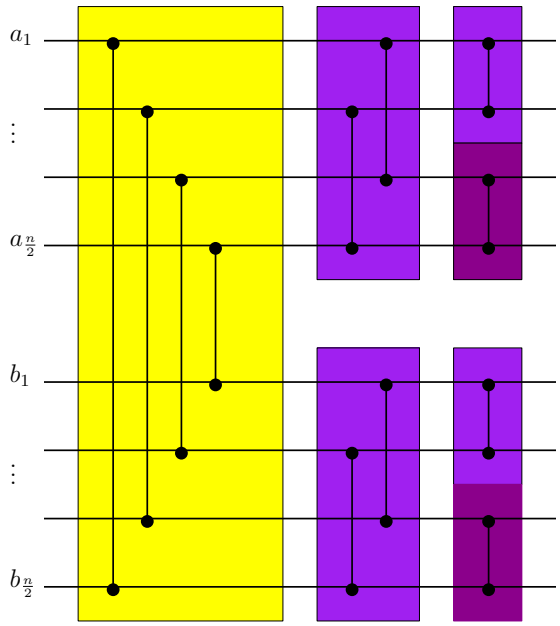

Abbildung 8: Schematische Darstellung des „Herschens“ eines Biton-Sortierers

3.4.1 Das Verfahren

Zur Vereinfachung betrachten wir Eingabemengen M der Länge 2^x.

Schritt 1 Teilen
Zuerst wird Eingabemenge in die 2 Hälften $m_0, m_1, ..., m_{n/2-1}$ und $m_{n/2}, m_{n/2+1}, ..., m_{n-1}$ aufgeteilt. Diese werden rekursiv mit einem Biton-Sortierer sortiert.

Schritt 2 Herschen
Vergleiche $[m_i{:}m_{2^x-i}]$ i = $\{0, 1, ..., 2^{x-1}\}$
Solange n < 1:
Vergleiche $[m_i{:}m_{n/4+i}]$ und $[m_{n/2+i}{:}m_{3n/4+i}]$ i = $\{0, 1, ..., 2^{x-2}\}$
Wiederhole für die Teilmengen m_0,m_1,...,$m_{n/2}$ und $m_{n/2}$,$m_{n/2+1}$,...,m_n
In einem Sortiertnetzwerk sieht das Herschen nun wie folgt aus:

3.4.2 Korrektheit

Um die Korrektheit des Bitonen-Sortierers zu zeigen, benutzen wir das 0-1 Prinzip und beschränken die Eingabefolge auf 0’en und 1’en.
Seien (a_1, a_2, ..., $a_{n/2}$) und (b_1, b_2, ..., b_{n_2}) die 2 Hälften der Eingabefolge, die beide sortiert sind.

Schritt 1 Vorsortierung
Im 1. Schritt werden die kleinereren Elemente nach „oben“ und die größeren Elemen-

te nach „unten“ sortiert. Bei einer Eingabemenge von {0, 1} besteht entweder die „obere“ Hälfte nur aus 0'en oder die “untere“ Hälfte nur aus 1'en.
Die unsortierte Hälfte hat nun die Form 0...01...10...0 („oben“) oder 1...10...01...1 („unten“).

Schritt 2 „Rekursion“

Im 2. Schritt wird das Prinzip aus dem 1. Schritt für die immer kleiner werden Hälften angewendet, d.h. wir sortieren die kleineren Elemente nach “oben“ und die größeren nach “unten“.
Damit ist die Folge sortiert, wenn wir bei 1-elementigen Teilmengen angekommen sind.

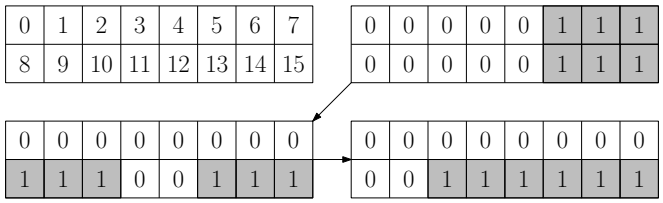

Abbildung 9: Beispiel eines Sortierschrittes des Biton-Sortierers und die sortierte Folge

3.5 Odd-Even-Mergesort

Das Odd-Even-Mergesort ist ein Sortierverfahren, welches dem Biton-Sortierer ähnelt. Da es, wie der Biton-Sortierer, eingabeunabhängig ist, eignet es sich perfekt für die Umsetzung in ein Sortiernetzwerk.

3.5.1 Merge-Verfahren

Das Merge-Verfahren setzt vorraus, dass die beiden Hälften $m_0, m_1, ..., m_{n/2-1}$ und $m_{n/2}, m_{n/2+1}, ..., m_{n-1}$ der Eingabefolge $m_0, m_1, ..., m_n$ soriert sind.

Wenn n > 2 .

1. Wende Odd-Even-Mergesort auf die beiden Teilfolgen
$m_0, m_2, m_4, ..., m_{n-2}$ und
$m_1, m_3, m_5, ..., m_{n-1}$ an.
2. Vergleiche m_i und m_{i+1} $\forall\, i \in \{1, 3, 5, n-3\}$

sonst .

Vergleiche m_0 und m_1

3.5.2 Sortier-Verfahren

Das Merge-Verfahren setzt sortierte Teilmengen voraus. Dies ist in der Praxis normalerweise nicht gegeben. Um diese Voraussetzung zu umgehen, muss man das Odd-Even-Mergesort rekursiv auf die Teilmengen anwenden:

Wenn n > 1 .

1. Wende Odd-Even-Mergesort rekursiv auf beiden Hälften $m_0, m_1, ..., m_{n/2-1}$ und $m_{n/2}, m_{n/2+1}, ..., m_{n-1}$ der Eingabemenge anwenden.
2. Wende Odd-Even-Merge auf die Eingabemenge M an.

sonst .Eingabemenge bereits sortiert

3.5.3 Korrektheit

Um die Korrektheit zu zeigen verwenden wir das 0,1-Prinzip und beweisen, dass das Merge-Verfahren für eine 0,1-Eingabefolge M mit n Elementen korrekt arbeitet.

Beweis durch Induktion

Induktionsanker n = 2^1

Es wird der Vergleich der „sonst"-Klausul ausgeführt. Damit ist die Folge offensichtlich sortiert.

Induktionsvoraussetzung (IV) Das Sortierverfahren sei für n = 2^x korrekt

Induktionsschritt $2^x \to 2^{x+1}$ In Schritt 1 des Sortierverfahrens wird die Eingabemenge halbiert und die beiden Hälften mit Odd-Even-Mergesort sortiert. Nach Induktionsvoraussetzung sind die beiden Hälften danach korrekt sortiert. Nun wenden wir das Merge-Verfahren auf die komplette Menge an.
Um die Korrektheit des Merge-Verfahrens zu zeigen, betrachten wir die Anzahl von 1'en und 0'en in den Mengen.
Zuerst betrachten wir die beiden sortierten Hälften aus Schritt 1.

Menge 1: |1'en| sei n_0 und |0'en| sei p_0.

Menge 2: |1'en| sei n_1 und |0'en| sei p_1.

Wenn wir diese beiden Mengen wieder zusammenfügen und dann in folgende Mengen aufteilen:

$$M_0 = m_0, m_2, m_4, ..., m_{n-2}$$
$$M_1 = m_1, m_3, m_5, ..., m_{n-1}$$

Werden die 0'en und 1'en „gleichmäßig" aufgeteilt.

$$M_0 : |1\text{'en}| = \left\lfloor \frac{n_0}{2} \right\rfloor + \left\lfloor \frac{n_1}{2} \right\rfloor \text{ und } |0\text{'en}| = \left\lceil \frac{p_0}{2} \right\rceil + \left\lceil \frac{p_1}{2} \right\rceil$$
$$M_1 : |1\text{'en}| = \left\lceil \frac{n_0}{2} \right\rceil + \left\lceil \frac{n_1}{2} \right\rceil \text{ und } |0\text{'en}| = \left\lfloor \frac{p_0}{2} \right\rfloor + \left\lfloor \frac{p_1}{2} \right\rfloor$$

Nun betrachten wir die Differenz von $|1\text{'en}|_{M_0}$ und $|1\text{'en}|_{M_1}$:

Fall 1: Seien n_0 und n_1 gerade: $|1\text{'en}|_{M_0} = |1\text{'en}|_{M_1}$.

Fall 2: Sei n_0 oder n_1 ungerade: $|1\text{'en}|_{M_0} = |1\text{'en}|_{M_1}$ - 1.

Fall 3: Seien n_0 und n_1 ungerade: $|1\text{'en}|_{M_0} = |1\text{'en}|_{M_1}$ - 2.

Die beiden Teilmengen M_0 und M_1 werden nun wieder sortiert (da die Länge wiederrum 2^x ist, gilt IV), danach ergibt sich eines der folgenden Schemata:

Wenden wir nun Schritt 2 des Mergeverfahrens an, ist die Liste sortiert.

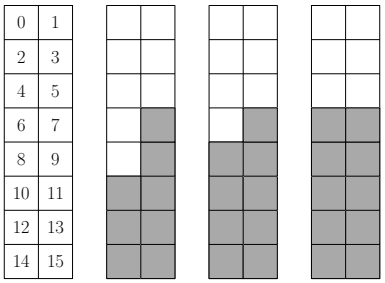

Abbildung 10: Schematische Darstellung der Fälle 1 (links) bis 3 (rechts)

4 Laufzeit

Die Laufzeit von parallelen Algorithmen wird durch die Anzahl nacheinander ausgeführten Schritten bestimmt.

4.1 Herleitung

Die Laufzeit der beiden vorgestellten Sortieralgorithmen unterscheidet sich in O-Notation nicht, deshalb beschränken wir uns auf die Herleitung an einem Beispiel:

Länge der Eingabe	Anzahl der Schritte
2^1	1
2^2	1+2
2^k	1+2+3+...+k-1+k
	$= \sum_{i=1}^{k} i = \frac{1}{2} * log_2 n(log_2 n + 1)$

Tabelle 1: Entwicklung der Laufzeit

Damit ergibt sich eine Laufzeit von $\Theta((log(n))^2)$.

5 Vergleich der Laufzeit

Um die Laufzeit von parallelen Algorithmen mit der von sequentiellen zu Vergleichen, wir die Anzahl der Schritte in parallelen Algorithmen mit der maximalen Anzahl von Komponenten multipliziert.
Damit ergibt sich eine normalisierte Laufzeit von $O(n * log^2)$ für die vorgestellten parallelen Algorithmen.

Algorithmus	Laufzeit		
	best	worst	avarage/normiert
Bubblesort	O(n)	$O(n^2)$	
Mergesort	$O(n * log(n))$		$O(n * log(n))$
Quicksort	$O(n * log(n)$	$O(n^2)$	$O(n * log(n))$
Netzwerk	$O(n * log(n)^2)$		$O(n * log(n)^2)$

Tabelle 2: Vergleich zu Softwaresortierung

6 Abschlussbetrachtung

In diesem Bericht wurde unser Vortrag "Paralleles Sortieren" zusammengefasst. Wir zeigten aufbauend auf einem einfachen Komparator, wie ein Sortiernetzwerk erst naiv und dann an einen Algorithmus angelehnt aufgebaut werden kann. Dies war eine wichtige Erfahrung, da es abweichend vom Lehrplan der Grundvorlesung die reine Softwareanwendung eines bekannten Algorithmus um die Übertragung auf Systeme und Netzwerke erweitert und gleichzeitig den Verlauf der Entwicklung näher gebracht hat. Ein wichtiger Bestandteil ist das "0,1„-Prinzip, dass eine einfache und verlässliche Grundlage für die Überprüfung von Sortiernetzwerken darstellt. Wir haben außerdem gezeigt, dass das entstandene Netzwerk zwar schneller arbeitet, aber die Laufzeitverbesserung nur über einen Mehraufwand an Komperatoren erreicht wird.
Zur weiteren Vertiefung wären die Betrachtung von Hypercubes und AKS-Netzwerke, die entweder schneller oder flexibler als das gezeigte Netzwerk sind, möglich.

7 Anhang

7.1 Text-Quellen

1 Taschenbuch der Algorithmen - Springer Verlag - 2008.

2 Einführung in Parallele Algorithmen und Architekturen - Tom Leighton - Thomsom Publisching - 1997 - 3-8266-0248-X

3 www.wikipedia.de - Laufzeiten der Sortieralgorithmen

4 http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/networks/nulleins.htm - 0,1-Prinzip - Beweis des 0,1-Prinzip

5 http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/networks/oem.htm - Odd-Even-Mergesort

6 http://www.imn.htwk-leipzig.de/j̃ahn/Cprog/Alg_Inf_Jahr_pdf/parallel_sort.pdf - Beweis des Biton-Sortierers