Abschlussbericht
Proseminar Algorithmen
Vortrag :Parallele Sortierung

Patrick Winterstein,Björn Rathjen

SS14

Contents

List of Figures

List of Tables

1 Einführung

Der Titel Paralleles Sortieren beschreibt zwei grundsätzliche Bestrebungen bei der Optimierung von Netzwerk- (hauptsächlich Routing)und Datenstruckturen (Listen, Wörterbücher, ...). Somit ist die Motivation diesen Thema zu behandeln groß, da sie für die Projekt die Basis für dessen Geschwindigkeit legen kann. Für sich alleine sind diese von folgender Bedeutung :

Sortierung 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 Durch den Aufwand von mehr Ressourcen kann ein Sortierprozess beschleunigt werden ohne die Korrektheit den Ergebnisses zu gefährden. Dabei ist Parallelität der Extremfall, der eine Maximierung des Datendurchsatzes erlaubt und somit in einer höheren Geschwindigkeit gegenüber sequentiellen Ansätzen resultiert.

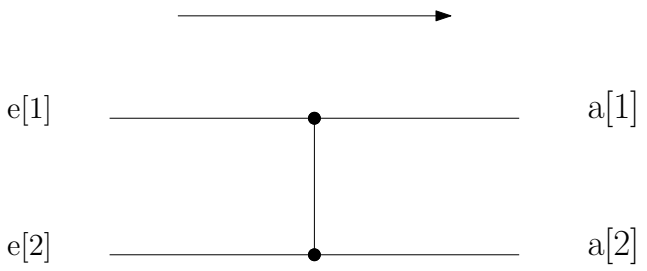

Figure 1: Schema eines Komparators. e entspricht den Eingängen und a den Ausgängen. Der Pfeil gibt die Durchlaufrichtung an. Die waagerechten schwarzen Linien entsprechen den Datenleitungen, die senkrechte Linie zeigt die vergleichenden Teil an.

2 Grundlagen

2.1 Sortieren

Als Grundlage um eine Menge M sortieren zu können muss eine Ordnungsrelation R vorhanden sein, diese schreibt eine Ordnung zwischen zwei Elementen vor.

$$R \subseteq M \times M \tag{1}$$

Der Anspruch an diese Relation ist, dass diese Transitivität beinhaltet.

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

Würde eine dieser Voraussetzungen nicht erfüllt sein, so würde bei (1) kein sortieren möglich sein, da keine Relation vorhanden ist, und bei (2) die Relation zweier Elemente untereinander keine Aussage über die Relation zu anderen Elementen aussagt. Daraus resultiert das das "sortierte" Ergebnis nur zu einem Element als sortiert betrachtet werden kann.

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 Mengen eingegeben werden, den Vergleichen Teil der die Ordnungsrelation anwendet und die beiden Ausgangsleitungen auf den 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.

```
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  }
```

Figure 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) und (2)) 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 :

- mehrere Eingabeleitungen
- mekrere Vergleicher
- Ausgabe soll sortiert sein

Es wird 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.

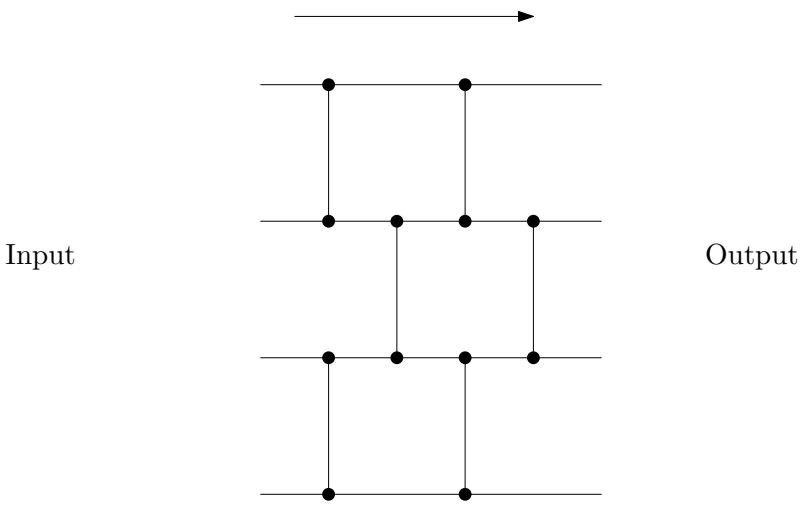

Figure 3: Komparatornetzwerk mit mehreren Datenleitungen

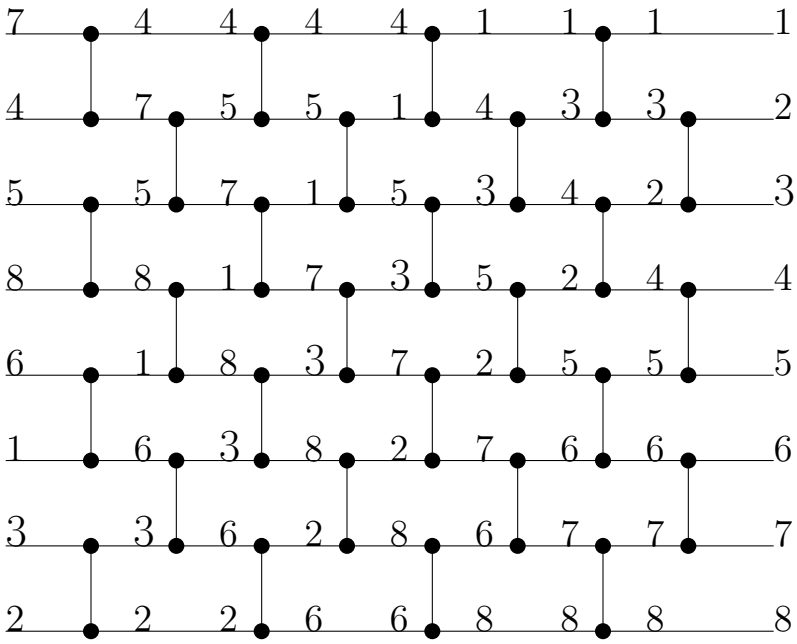

Figure 4: Beispiel mit acht Datenleitungen dass einen schrittweisen Verlauf des sortierens zeigt und eine vollständig sortierte Ausgabe.

3.2 Korrektheit

3.2.1 Betrachtung der Analyse

Problematisch bei der Überprüfung durch Beispiele ist die große Anzahl an der zu überprüfenden Fälle sehr groß ist. Im Beispiel in Abbildung 4 (Seite 6) bei einer Eingabemenge mit den Elementen 1 bis 8 wären dies 40320 unterschiedliche Fälle (diese Anzahl ist durch den Ausschluss, dass 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 werden die verschiedenen möglichen Eingaben durch 0,1-Folgen äquivalente (Zuweisung wird um Beweis gezeigt) und somit festgestellt, ob die simulierten Eingaben auch sortiert werden würden.

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 $S = s_1 \dots s_l$
- unsortierte Ausgabefolge von E $U = u_1 \dots u_l$
- kleinster Index an dem $u_k \neq s_k$

iii) Folgerungen aus ii) :

$$u_i = s_i \qquad \forall 0 \neq i < k \tag{3}$$

$$u_r = s_k \qquad \text{mit } r > k \tag{4}$$

iv) Funktion die ein 0,1 Folge aus beliebiger anderer Folge erstellt. Die Konstante, die dazu verwendet wird ist in diesem Fall s_k.

$$f(e) = \begin{cases} 0, & if \;\; e_i \leq s_k \\ 1, & if \;\; e_i > s_k \end{cases} \tag{5}$$

v) Wendet man die Funktion nun auf die Eingabe / Ausgabe an so entsteht eine 0,1-Folge der Form

$$00 \dots 01_k \dots 0_r \dots \tag{6}$$

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

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

$$n! \to 2^l \tag{7}$$

reduziert. Daraus resultiert auch das mit größerer Sicherheit getestet werden kann und somit auch sauberer Testcode sauberer geschrieben werden kann.

In einem Beispiel in Abbildung 5 (Seite 9) wird die Zuweisung mit $s_k = 6$ demonstriert und die anschließende Sortierung.

3.3 Übertragung von Sortieralgorithmen auf ein Sortiernetzwerk

3.3.1 Bubblesort

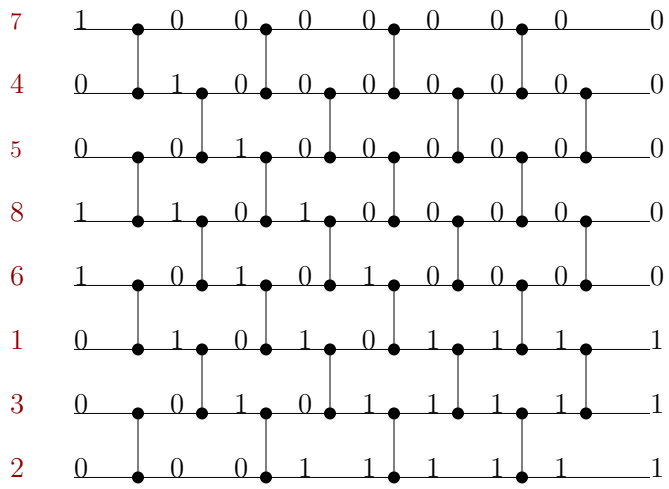

Figure 5: Beispiel für das 0,1-Prinzip. Folge wird auf das Beispiel aus Abbildung 4 (Seite 6) angewendet. Es zeigt, dass das Resultat für die gewählte Konstante sortiert ist. Genauso zeigt es auch das bis Schritt 5 die Folge nicht sortiert ist.

Netzwerkimplementierung (6 Seite 10) 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.

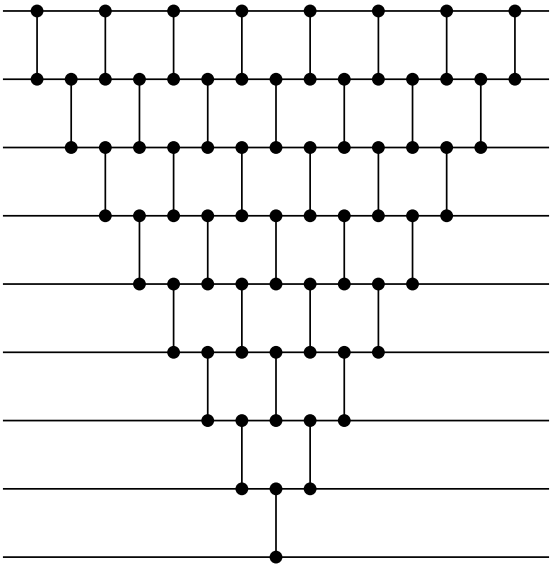

Figure 6: Bubblesort: Bubblesort mit parallelen Vergleichen.

3.3.2 Quicksort , Mergesort

Bei der Implementierung schnellere Algorithmen zu implementieren entstehen durch direkte Umwandlung Probleme.

Quicksort Bei Quicksort wird vorweg 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 den Pivot nicht mehr festgelegt ist.

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 11) verdeutlicht.

3.4 Biton-Sortierer

3.5 Odd-Even-Mergesort

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

4 Laufzeit

4.1 Herleitung

5 Resultat

5.1 Vergleich zu Softwaresortierung

6 Anhang

6.1 Text-Quellen

6.2 Bild-Quellen