Philipps-Universität Marburg
Fachbereich Mathematik und Informatik
Prof. Dr. Rita Loogen, Dipl. Inform. Jost Berthold

Sortieren auf PRAMs

Seminar zur Parallelen Programmierung
Wintersemester 2005/2006

Michael Lemler

Marburg, 10.01.06

Kurzzusammenfassung

PRAMs sind theoretische Maschinenmodelle zur Analyse paralleler Algorithmen und Bestimmung der Komplexitätsklasse der effizient parallel entschiedbaren Probleme. In dieser Arbeit soll das Loch zwischen der Thorie und der Praxis dieser Klassenzuordnung anhand zweier Sortieralgorithmen aufgezeigt und Dank zusätzlicher Analysemöglichkeiten verkleinert werden. Cole's Mergesort ist im Vergleich zu Batcher's Bitonicsort zwar in der Theorie der bessere Algorihtmus, steht in der Praxis diesem jedoch nach.

Inhaltsverzeichnis

1 Einführung

In vielen Arbeiten in dem Forschungsfeld der parallelen Programmierung spielt die Zugehörigkeit einzelner Algorithmen in der Komplexitätsklasse NC (Nick's Class, nach Nick Pippenger) eine große Rolle. Die Zuordnung eines Algorithmus in diese Klasse besagt, dass er auf Parallelrechner in deutlich besserer Zeit berechnet werden kann, als auf einem sequentiell arbeitendem Rechner. Entschieden werden kann dies eben durch eine PRAM (Prallel Random Access Maschine). Ist ein Algorithmus in logarithmischer Zeit bei polynomialem Aufwand (entspricht dem Produkt aus Rechenzeit und der Anzahl der Prozessoren) entscheidbar, ist er Element der Klasse NC. Wie ist allerdings der pratische Nutzen von NC-Algorithmen zu sehen, bzw. gibt es überhaupt einen?

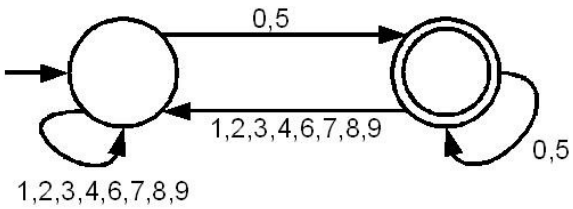
Um die Zeit- oder auch Speicherplatzkomplexität eines Algorithmuses anzugeben wird in der Komplexitättheorie die O-Notation (auch das Landau-Symbol, nach Edmund Landau) benutzt. Hier wird das asymptotische Verhalten der Rechenzeit oder des Speicherplatzbedarfs des Algorithmuses bei verschieden langen Eingaben betrachtet, wobei dann meist eine Art obere Schranke angegeben wird, die Eingabelänge also gegen unendlich geht. So ergeben sich Komplexitätsklassen in denen die Algorithmen sich einordnen lassen, wie zum Beispiel verschiedene Sortieralgorithmen. So gehört Bubblesort mit $O(n^2)$ zu den Sortieralgorithmen mit quadratischem Zeitaufwand und Mergesort mit O(n*log n) zu den logarithmischen. Hierfür ist die O-Notation sehr sinnvoll, Algorithmen können auf einem sehr hohen Abstraktionslevel beschireben und analysiert werden. Werden allerdings Algorithmen für den tatsächlichen gebrauch, mit einer großen aber eben doch endlichen Eingabe, verglichen können bei dieser Methode wichtige Detais wegfallen, die für den Vergleich unendberlich wären. Dies soll nun an einem Bespiel dargestellt werden und damit dazu beitragen das Loch zwischen dem theoretischen und praktischem Nutzen von PRAMs zu erkennen und mittels zusätzlicher praktischer Analyse zu verkleinern.

2 Grundlagen

In diesem Kapitel erarbeiten wir die Grundlagen zum Verständnis einer PRAM (Parallel Random Access Machine) und zur Zeitmessung der beiden Sortieralgorithmen, die verglichen werden.

2.1 PRAMs und and Maschinenmodelle

Maschinenmodelle (Automaten) sind gedachte/abstrakte Modelle einer Maschine, die sich nach bestimmten Regel verhalten. Ein Automat hat verschiedene Zustände und bekommt eine Eingabe, die er Zeichen für Zeichen einliest und nach jedem gelesenen Zeichen seinen Zustand entsprechend den Regeln ändert bis die Eingabe (das Wort) zuende gelesen ist oder sich vorher ein Fehler eintritt. Hierbei wird noch zwischen deterministischen und nichtdeterministischen Automaten unterschieden, wobei bei einem nichtdeterministischen Automaten der neue Zustand nicht eindeutig bestimmt sein muss. Sie werden in der theoretischen Informatik benutzt um bestimmte Eigenschaften von Programmen zu analysieren. Es gibt verschiedene Arten von Maschinenmodellen, die verschieden mächtig sind. So kann ein Endlicher Automat lediglich die Regulären Sprachen erkennen, welche in der Chomsky-Hierarchie als Typ 3-Grammatik bezeichnet werden und somit die meisten Einschränkungen beinhaltet. Das bedeutet, dass nicht alle Probleme in ihr dargestellt werden können.



Hier ein kleines Beispiel. Dieser Automat erkennt alle Zahlen, die durch fünf teilbar sind. Man startet links im ersten Knoten und liest das Eingabewort (in diesem Fall Eingabezahl) Element für Element ein und wechselt den Zustand anhand der Regeln (Pfeile). Liest man eine fünf oder null ein wechselt (bzw. bleibt) man in den rechten Knoten (Zusatnd), der ein akzeptierender Zustand ist, an der doppelten Line zu erkennen. List man eine andere Ziffer wechselt man (bzw. bleibt) in den linken Knoten. Hat man die letzte Ziffer gelesen und befindet sich dann in einem akzeptierenden Zustand, hier also im rechten Knoten, ist das Wort in der Sprache. Wird also als letzte Ziffer eine fünf oder null gelesen ist die gaze Zahl durch fünf teilbar.

Allerdings gibt es auch Maschinenmodelle, die so mächtig sind, dass sie alle Probleme, die von Computern gelöst werden können, ebenfalls lösen kann und somit einen Computer simulieren können. Ein wichtiges dieser Modelle der sogenannten Typ 0-Grammatiken, also Automaten ohne Einschränkungen, ist die Turingmaschine, die 1936 von Alan Turing entwickelt wurde um das Entscheidungsproblem zu lösen. Ein ebenso mächtiges Modell wie die Turingmaschine ist die RAM (Random Access Machine, deutsch: Registermaschine) aus dem Jahr 1963. Die RAM und die Turingmaschine können sich in polynomieller Laufzeit gegenseitig simulieren und somit gelten alle Aussagen, die auf der einen bewiesen werden können auch für die andere. Allerdings kann auf der RAM in der Regel die Zeitkomplexität besser bestimmt werden, was für unser Vorhaben von Vorteil ist. Erweitert man nun die RAM um die Möglichkeit Eingaben parallel zu verarbeiten erhalten wir die PRAM (Parallel Random Access Machine). Hier arbeiten mehrere Prozessorelemente (PE) auf einem gemeinsamem Speicher. Eine einzelne PE stratet die Berechnung und kann dann in einem Berechnungsschritt entweder eine andere PE aktivieren, eine RAM-Berechnung ausführen oder in den lokalen oder globalen Speicher schreiben. PRAMs werden in der Komplexitättheorie zur Definition der Klasse NC (Nick's Class, die Klasse der effizent parallel entschiedbaren Probleme) benutzt. Dessweiteren lassen sich damit natürlich auch Analysen paralleler Algorithmen durchführen. Jetzt gibt es allerdings noch unterschiedliche Arten von PRAM, die sich zum einen auf die mögliche Ausführung verschiedener Befehle in einem Zeittakt und zum anderen auf die Regelung des Lese- und Schreibezugriffs beziehen.

2.1.1 Ein Beispielprogramm

Gegeben sei eine unsortierte Liste von n unterschiedlichen Integerelemente x_1 bis x_n. Gesucht ist das Element x_i mit dem höchsten Wert.

```
for i,j=1 to n pardo
  if x_i ≥ x_j
   then b_ij := 1
   else b_ij := 0
  fi

for i=1 to n pardo
  m_i := b_i1 ∧ b_i2 ∧ ... ∧ b_in
```

Pardo steht hier für „do in parallel“, hier also für die prallele Ausführung der Schleifen, d.h. jeder einzelne Schleifendurchlauf wird gleichzeitig von unterschiedlichen Prozessoren berechnet. In der ersten Schleife wird für jedes Element geprüft ob es größergleich (1) oder kleiner (0) als die anderen Elemente ist und dies in eine Speicherzelle mit passendem Index geschrieben. Durch die parallele Ausführung ist diese Berechnung für beliebig große n konstant, allerdings wächst bei großerem n auch die Anzahl der benötigten Prozessoren. Der zweite Teil bleibt ebenso konstant. Hier wird durch eine und-Verknüpfung das Element auf 1 gesetzt, das größergleich alle Elemente ist. Jedes Element das kleiner als ein anderes Element ist hat nämlich mindestens ein b_{ij} das 0 gesetzt ist und damit den ganzen oder-Ausdruck 0 setzt.

2.2 Grundlagen der Messung

2.2.1 Der Simulator

Da jedoch genau auf den Unterschied zwischen dieser theoretischen Zeitbetrachtung und dem Verhalten in einem realen Fall hingeweisen werden soll braucht man eine andere Methode. Die hier betrachtete Möglichkeit ist es den zu untersuchenden Algorithmen in PIL zu implementieren und auf einem CREW PRAM Simulator laufen lassen. PIL ist eine Erweiterung der Hochsprache SIMULA mit der Möglichkeit der Prozessorverteilung, -aktivierung und -sychronisation, Mittel zur Kommunikation/Interaktion mit dem Simulator und noch einigem anderen.

Die Zeit wird nun in CREW PRAM Befehlszeiteinheiten, kurz Zeiteinheiten gemessen. Wie viel Zeiteinheiten ein Befehl verbraucht steht als Parameter im Simulator und kann somit leicht geändert werden. Da von einem kleinen und einfachen Befehlssatz ausgegangen wird verbrauchen die meisten Befehle nur eine Zeiteinheit. Der Konsum dieser Zeiteinheiten muss im PIL Programm festgeschrieben werden. Dessweiteren verfügt der Simulator über folgende Eigenschaften zur Analyse der implementierten Algorithmen.

- *Die Globale Uhr* ist notwendig, da alle Prozessoren synchron laufen sollen. Sie kann gelesen und zurückgesetzt werden. Dessweiteren gibt der Simulator im Default-Modus die exakte Zeit jedes Ereignisses, das eine Ausgabe produziert, aus.
- In der aktuellen Version des Simulators muss der *Konsum/Verbrauch der Zeiteinheiten, die jeder Prozessor für die lokalen Berechnungen braucht ,im PIL Programm vorgegeben* werden. Dadruch kann man die Genauigkeit der Anaylse selbst einstellen und man hat die Freiheit jeden Befehlssatz oder Zeitdarstellung zu benutzen die man möchte.
- Die *Anzahl der nötigen Prozessoren* müssen ebenfalls im PIL Programm feststehen und sind daher leicht abzulesen und auch *die benötigte Größe des globalen Speichers* kann leicht vom Stack abgelesen werden.
- In der *Zugriffsstatistik des globelen Speichers* werden alle Lese- und Schreibezugriffe auf diesen erfasst und gespeichert.
- Auserdem sind die Möglichkeiten der *DEMOS Datenerhebung* verfügbar. Mit `count` können alle möglichen Aufkommen gezählt werden, mit `tally` können verschiedene Statistiken über Zeitunabhängige Variablen gespeichert werden und `histogram` bietet die einfache Möglichkeit zur Ausgabe der gemessenen Daten.

Es sei noch angemerkt, dass alle dieser Zeitmessungshilfen den zu testenden Algorithmus an sich nicht beeinflussen und somit das Ergebnis nicht verfälschen. Weitere Informationen zu PIL und dem Simulator finden sich in Lasse Natvig's CREW PRAM Simulator-Users's Guide.

2.2.2 Skelette

Danke dem Hilfsmittel der Skelette kann das mergen zweier sortierter Folgen in konstanter Zeit erfolgen. Eine Folge X heißt Skelett von Y ($X \propto Y$), falls für alle $k \geq 2$ gilt: zwischen je k Elementen von X liegen höchstens 2k-1 Elemente von Y, wobei X und Y in sich sortiert sein müssen und in X üblicher Weise an erster und letzter Stelle noch -∞ und +∞ eingefügt werden. Haben nun zwei Folgen ein gemeinsammes Skelett können sie durch, anhand des Skeletts, zerlegte Teilfolgen in kostanter Zeit, also O(1), zusammengemerged werden. Hier ein kleines Beispiel in dem X das gemeinsamme Skelett von Y und Z ist:

Y = (1 4 6 9 11 12)

X = (5 10)

Z = (2 3 7 8 10 14)

Für k = 2 gilt jetzt 2k-1 = 3, also dürfen maximal drei Elemente in Y und Z zwischen je zwei auf einander folgenden Zahlen aus X liegen, wobei man die hinzugefügten +∞ und -∞ natürlich nicht vergessen darf.

Y = (1 4 | 6 9 | 11 12)

Z = (2 3 | 7 8 10 | 14)

Das selbe muss für k = 3 gelten, d.h. zwischen -∞ und 10 dürfen genauso wie zwischen 5 und +∞ in den Folgen Y und Z maximal 2k-1 = 5 Zahlen liegen.

Y = ($|_1$ 1 4 $|_2$ 6 9 $|_1$ 11 12 $|_2$)

Z = ($|_1$ 2 3 $|_2$ 7 8 $|_1$ 10 14 $|_2$)

Zu guter letzt dürfen nun bei k = 4 maximal 2k-1 = 7 Elemente in den Folgen Y und Z zwischen plus und minus unendlich liegen. Da beide Folgen aus 6 (< 7) Elemente bestehen ist auch dies gegeben und somit ist X ein gemeinsammes Skelett von Y und Z.

3 Die zwei Sortieralgorithmen

3.1 Cole's Mergesort-Algorithmus

Cole's Algorithmus geht von einem Array mit $n = 2^k$ zu sortierenden Elementen aus die einzeln in die Blätter eines Binärbaums aufgeteilt werden. Jeder Knoten verschmelzt in mehreren Schritten und unter Hilfe der Skletttheorie von Folgen die bereits sortierten Teilarrays seiner Kindknoten bis zuletzt in der Wurzel des Binärbaums das sortierte Array liegt.

3.1.1 Der Algorithmus

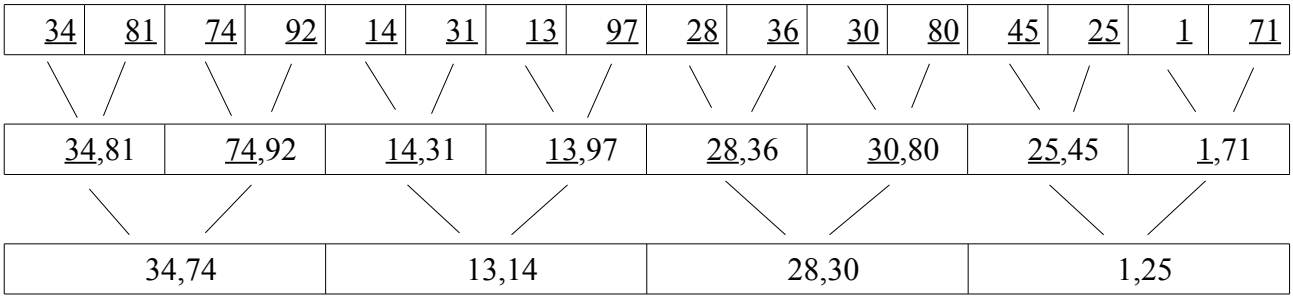
Wie schon gesagt werden die zu sortierenden Elemente auf die Blätter eines Binärbaums verteilt, was genau passt, da man von einem $n = 2^k$ ausgeht. Sollte es nicht passen könnte man die fehlenden Elemente bis zur nächsten Zweierpotenz zum Beispiel unendlich aufgefüllt werden. Wir betrachten nun die Arbeitsweise eines beliebigen inneren Knotens u.

T(u) sei der Unterbaum mit Wurzel u, val(u) sei die momentane (sortierte) Folge der Werte des Knotens u und list(u) sei fertige sortierte Folge aller Werte der Blätter in T(u), mit val(u) ist eine geordnete Teilfolge von list(u). Zeil eines jeden Knotens ist das erreichen von list(u) über die Teil- bzw. Hilfslisten val(u). Ein Knoten heißt vollständig wenn val(u) = list(u), ansonsten unvollständig. Zu Beginn ist val(u) leer. Der Knoten u wird aktiv wenn er von seinen direkten Kindknoten jeweils ein Element erhält, die er zur ersten Teilfolge val(u), der Länge 2, verschmilzt (merged). Im jeweils nächsten Schritt erhält u von seinen Kindknoten je eine doppelt solange sortierte Liste wie im letzten Schritt und verschmilzt diese unter zuhilfenahme des alten val(u), welches ein Skelett der beiden Listen der Kindknoten ist, zur neuen val(u) bis die Länge von list(u) erreicht ist und u somit vollständig wird.

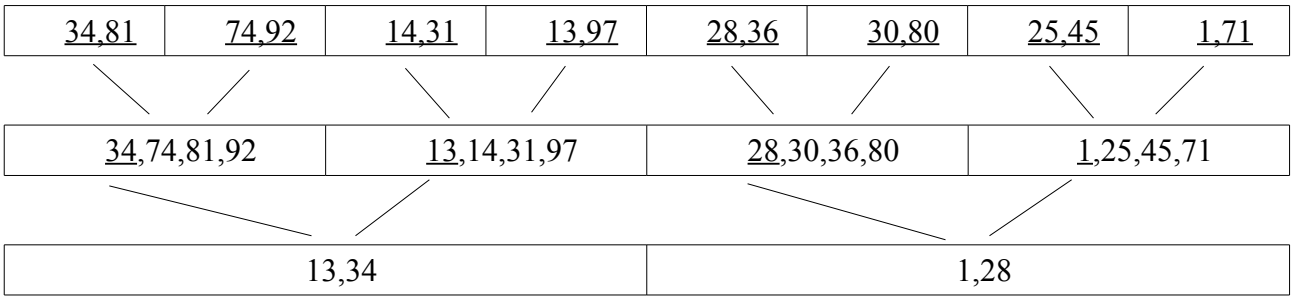
Natürlich muss auch der Knoten u seine Teilfolgen an seinen Elternknoten v übermitteln. Diese Ausgabe erfolgt nach folgenden Regel die gewährleisten, dass die im letzten Schritt im Elternknoten v erzeugte Teilfolge val(v) ein Skelett der aktuell gesendeten Teilfolge ist. Solange der Knoten u unvollständig ist, aber val(u) eine Länge von 4 oder mehr erreicht hat, sendet u in jedem Schritt jedes vierte Element seiner aktuellen Teilfolge val(u) als eigene Folge an seinen Elternknoten. Sobald u vollständig ist, bleibt er noch zwei weitere Schritte aktiv. Im vorletzten Schritt sendet er jedes zweite Element seiner inzwischen feritgen sortierten Folge list(u) als eigene Folge an seine Elternknoten, im letzen Schritt sendet er die ganze Folge list(u) und wird dannach inaktiv. Folgendes Beispiel mit 16 Zahlen soll das Verfahren verdeutlichen.

3.1.2 Ein Beispiel mit 16 Zahlen

Im ersten Schritt des Algorithmus sind alle Blätter aktiv[1]. Jeder befindet sich theoretisch im zweiten und damit letzten Schritt seiner Vollständigkeit und sendet nun seine list(u) an den Elternknoten, der diese beiden einelementigen Listen verschmilzt. Im zweiten Schritt befinden sich dann die Knoten in der zweiten Stufe im ersten Schritt ihrer Vollständigkeit und senden somit jedes zweite Element (in dem Fall also jeweils nur das erste) ihrer list(u) an den Elternknoten, der diese dann ebenfalls zu einer einelementigen Liste verschmilzt. Die Knoten der dritten Stufen senden noch nicht an ihre Elternknoten, da sie noch keine val(v) der Länge größergleich vier haben. Wir sehen, dass erst auf dieser Stufe diese Regel greift. Die ersten Schritte laufen außerhalb von diesem Schema. Die Knoten der ersten Stufe sind jetzt inaktiv.

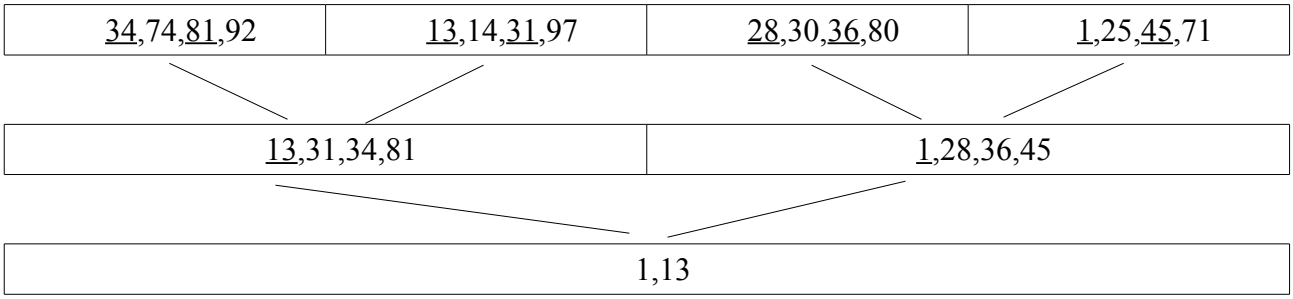


Im dritten Schritt senden die Knoten der zweiten Stufe ihre kompletten Folgen list(u) an die Elternknoten v und werden mit Hilfe ihrer Skelettfolgen val(v) zu einer vierelementigen Folge verschmolzen. Da jetzt die Knoten der dritten Stufe vier Elemente haben senden sie jedes vierte Element, also nur das erste, an den Elternknoten, der diese wieder verschmilzt. Jetzt sind diese Knoten vollständig. Die Knoten der zweiten Stufe werden inaktiv.

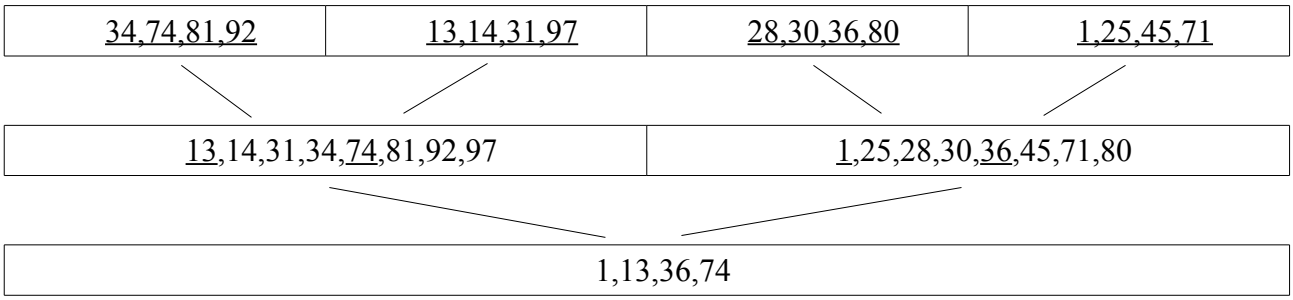


1 In einer PRAM startet immer nur ein einzelner Prozessor, der dann die Anzahl der benötigten Prozessoren berechnet und diese dann nach und nach aktivieren kann, z.B. per Broadcast. Wir gehen hier also davon aus, dass jetzt alle nötigen Prozessoren aktiviert wurden und daraufhin der erste Schritt des Algorithmuses starten kann.

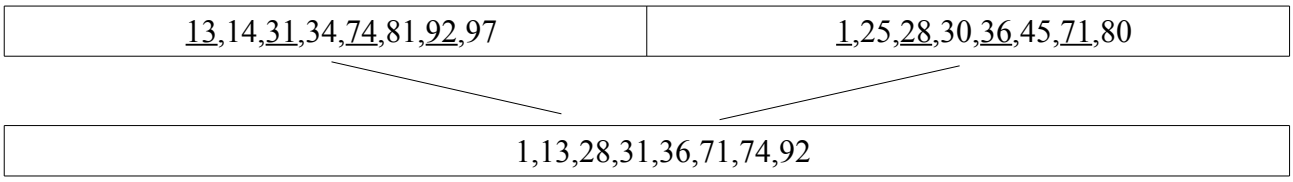
Nun sind wir im vierten Schritt und damit ganz im Algorithmus. Die inzwischen vollständigen Knoten der Stufe drei senden nun jedes zweite Element ihrer list(u) an ihre Elternknoten, wo sie mittel der im letzten Schritt entstandenen Sklettfolge zu einer neuen val(v) verschmolzen wird. Da diese Folgen jetzt auch vier Elemente lang ist senden sie jedes vierte Element an ihre Elternknoten.



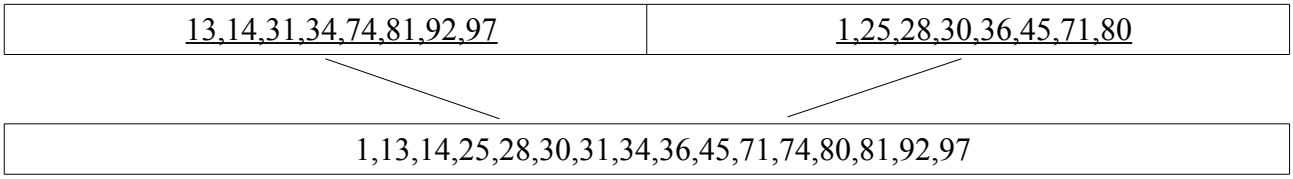
Im fünften Schritt sendet nun Stufe drei ihre kompletten list(u), die von ihren Elternknoten wieder zu neuen val(v) verschmolzen wird und wird inaktiv. Von den Knoten der vierten Stufe wiederum wird jedes vierte Element an deren Elternknoten gesendet, wo jetzt zwar auch vier Elemente sind, jedoch handelt es sich um die Wurzel, also existiert kein weiterer Elternknoten und demnach sendet dieser Knoten auch nichts. Die Knoten der Stufe vier sind jetzt auch vollständig.



Nun folgen noch die letzten beiden Schritte. Die Knoten der Stufe vier senden erst jedes zweite Element ihrer list(u) und im letzten die ganze Folge list(u), die jeweils von Wurzelknoten verschmolzen werden, der am Ende die sortierte Liste enthält.



Und hier der letzte Schritt:



Natvig hat diesen Algorithmus nun in PIL implementiert und auf dem PRAM-Simulator laufen lassen. Die Anzahl der benötigten Prozessoren entspricht bei ihm der maximalen größe der val(u), sample(u) und list(u) Arrays, wobei sample(u) die Teilfolge von val(u) (bzw. list(u), falls u vollständig ist) enthält, die der Knoten an seinen Elternknoten schickt, über alle Knoten u. Also folgender Summe:

$$AnzahlProzessoren = \sum_u |val(u)| + \sum_u |list(u)| + \sum_u |sample(u)|$$

Wobei die Teilsummen wie folg abgeschätzt werden können:

$$\sum_n |val(u)| \leqslant n + n/2 + n/16 + n/128 + ... = 11n/7$$

$$\sum_n |list(u)| \leqslant n + n/8 + n/64 + n/512 + ... = 8n/7$$

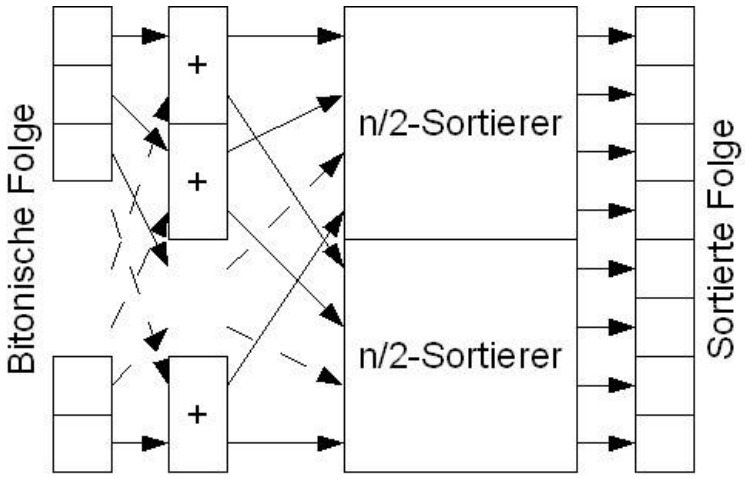
$$\sum_n |sample(u)| \leqslant n + n/8 + n/64 + n/512 + ... = 8n/7$$

Dies ergibt in der Summe ungefähr 4n, also eine kostant (O(n)) wachsende Anzahl an nötigen Prozessoren. Im Algorithmus wandert am Ende jedes drittes Schrittes die niedrigste aktive Stufe eine Stufe hoch richtung Wurzel und er braucht exakt 3*log(n) Schritte, hat also logarithmischen Zeitaufwand (O(log(n))).Dies entspricht genau dem Wunschergebnis der Kostenoptimalität. In Kapitel 4 wird gezeigt, zu welchem Ergbnis die Auswertung der Simulation gekommen sind. Hierbei wird das Hauptaugenmerk dem Vergleich zu Batcher's Bitonicsort gelten, der nicht die Bedingungen der Kostenoptimalität erfüllen kann.

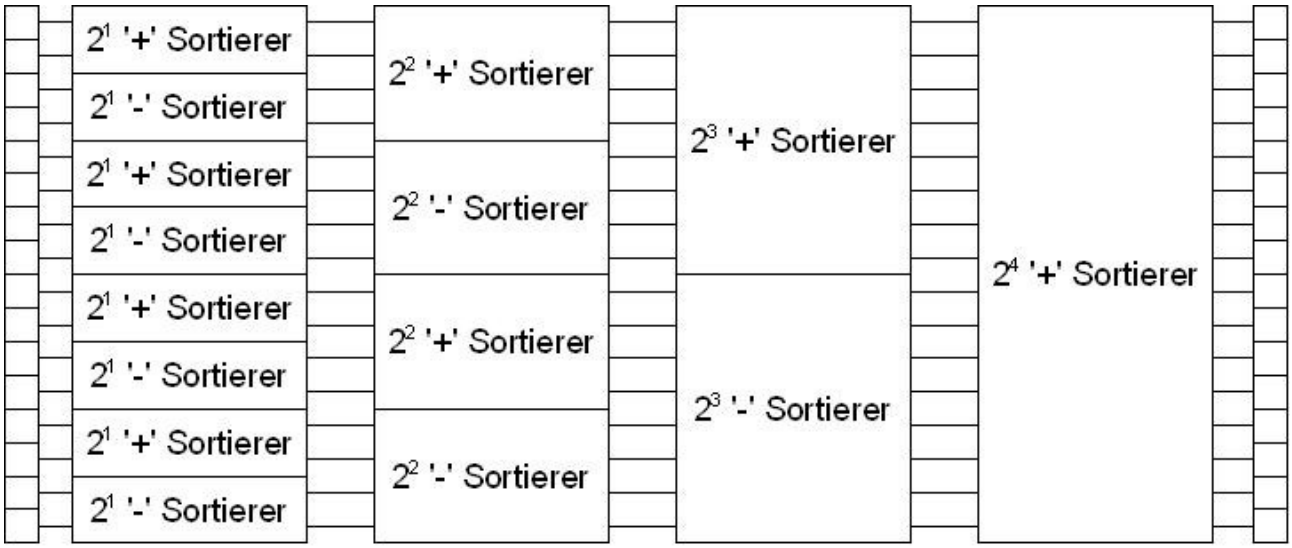
3.2 Batcher's Bitonicsort-Algorithmus

Batcher's bitonisches Sortiernetzwerk geht ebenfalls von einer Zweierpotenz als Eingabe aus, also einem $n = 2^m$. Es besteht aus (m/2)*(m+1) Spalten, die nacheinander abgearbeitet werden und jeweils n/2 komparatoren enthalten. Ein Pluskomparator (Minuskomparator) bekommt zwei Elemente als Eingabe, eines oben, eines unten (siehe Grafik unten), vergleicht diese und liefert als Ergebnis oben das kleinere (größere) und unten das größere (kleinere) Element. Jeder komparatorvergleich wird von einem Prozessor ausgeführt, somit brauchen wir n/2 Prozessoren, die dann in (m/2)*(m+1) Schritten die zu sortierende Eingabe erst in eine bitone Sequenz überführen und dann sortieren. Eine bitone Sequenz ist ein Folge $a_1,...,a_n$ von Elemente in der ein erster Teil $a_1,...,a_i$ aufsteigend und ein zweiter Teil $a_{i+1},...,a_n$ absteigend sortiert ist oder sich durch zyklische Verschiebung in eine solche Folge überführen lässt. Daraus folgt, dass jede sortierte (auf- oder absteigende) Folge ein bitone Sequenz ist, da man sie durch zyklisches Verschieben in eine solche überführen kann.

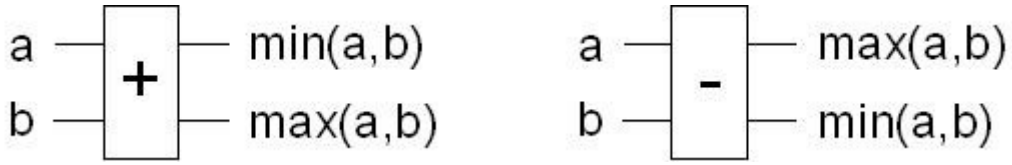
Per Definition ist jede Folge bestehend aus 2 Elementen eine bitone Folge. Im ersten Schritt des Netzwerks werden somit jeweils zwei zweielementige Folgen zu einer vierelementigen bitonen Folge verschmolzen. Dies erreicht man, indem man die erste Folge auf- und die zweite Folge absteigend sortiert, was mittels der komparatoren ganz leicht ist. Man schickt die erste zweielementige Folge durch den Pluskomparator, wodurch diese dannach aufsteigendsortiert ist, und die zweite Folge durch einen Minuskomparator, wodurch diese dann absteigend sortiert ist. Im nächsten Schritt vergleicht man bei den so entstandenen vierelementigen bitonischen Folgen das jeweils i-te element der aufsteigenden mit dem i-ten Element der absteigenden Hälfte und vertauscht diese gegebenenfalls. So schafft man es, dass die erste, aufsteigende Hälfte aus den kleineren Elementen besteht, die jetzt mittels des Sortierers der ersten Stufe, ebenso wie die zweite Hälfte mit den größeren Elementen, sortiert werden kann. Es bleibt darauf zu achten, dass die Folgen abwechselnd auf- und absteigend sortiert werden. In der nächsten Stufe werden dann wieder jedes i-te Element der ersten Hälfte, aus der so entstandenen doppelt so langen bitonen Folge, mit dem i-ten Element der zweiten Hälfte (also dem (i+2/n)-tem Element) mit dem Pluskomparator verglichen. Diese folgende Grafik soll dieses rekursive Verhalten des Algorithmuses veranschaulichen.



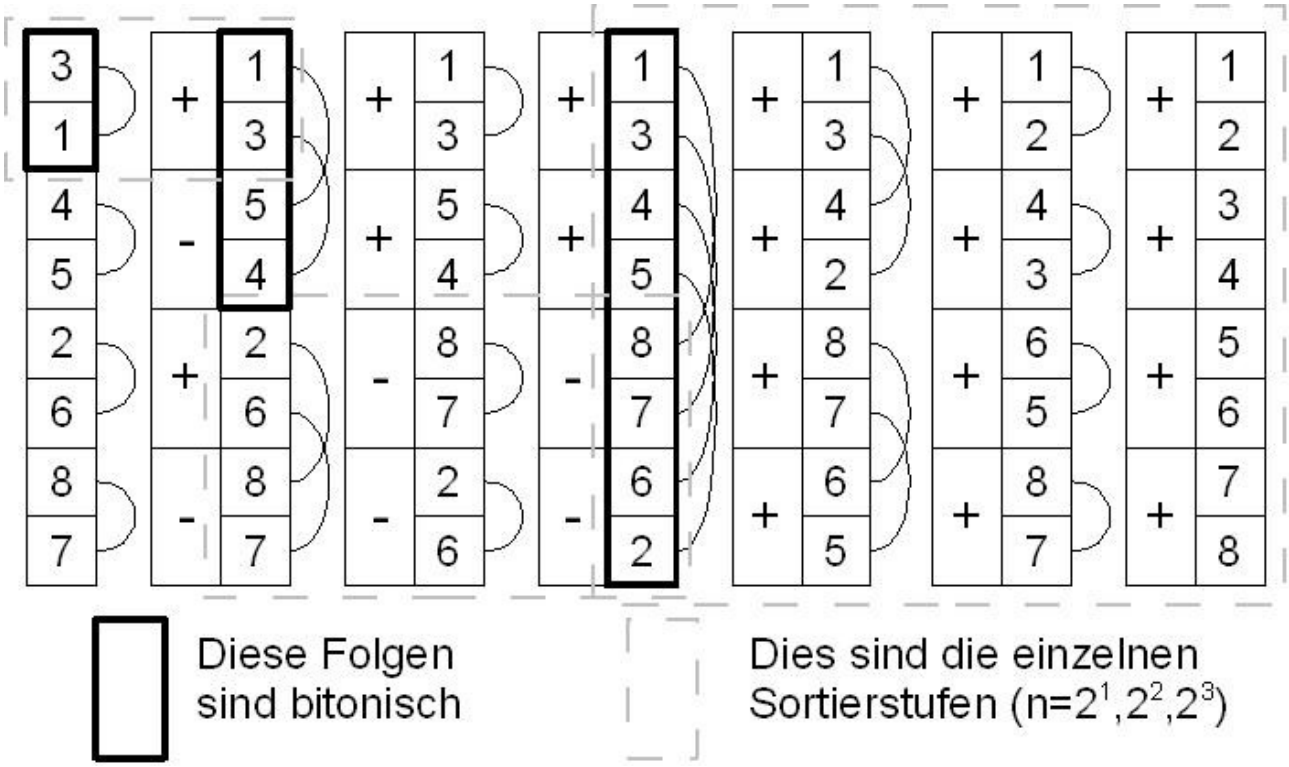
So geht es Stufe für Stufe weiter, wobei jede weitere Stufen einen Schritt mehr benötigt, als die vorige, womit sich auch die Anzahl der nötigen Schritte (m/2)*(m+1) begründen läßt, dem geschlossene Ausdruck der Gauß'schen Folge. Ebenfalls hier raus folgt das logarithmische Verhalten des Algorithmus. Nach ((m-1)/2)*m Schritten hat man die komplette Eingabefolge in eine bitone Folge umgewandelt, die erste Hälfte aufsteigend, die zweite Hälfte absteigend. Jetzt können wir mittels der Pluskomparatoren (Minuskomparatoren) diese Folge in eine aufsteigende (absteigende) und somit komplett sortierte Folge verschmelzen. Die folgdende Grafik zeigt wie das Sortiernetzwerk aus den einzelnen Sortierern aufgebaut wird. Die Eingabe jedes Sortierers ist jeweils bitonisch und die Ausgabe auf- bzw. absteigend sortiert. Den Aufbau eines solchen Sortierers veranschaulicht die obere Grafik, wobei in den Minussortierern die Plus- durch Minuskomparatoren zu ersetzen sind.



3.2.1 Ein Beispiel mit 8 Zahlen



Diese Bausteine stellen die Komparatoren dar. Bei dem Pluskomparator kommt oben der kleinere und unten der größere, bei dem Minuskomparator oben der größere und unten der kleinere Wert raus.



Die Verbindungslinien zweigen an welche Elemente verglichen und gegebenenfalls vertauscht werden, der entsprechende Komparator steht dahinter. Nach dem ersten Komparatorschritt haben wir die zwei vierelementigen bitonischen Folgen (1 3 5 4) und (2 6 8 7). Diese werden dann in zwei weiteren Schritten zu einer achtelementigen bitonen Folge verschmolzen. In den letzten drei Schritten wird diese in eine aufsteigend sortierte Folge verschmolzen. Würde man hier an Stelle der Pluskomparatoren die Minuskomparatoren einsetzten würde man eine absteigend sortierte Folge erhalten.

3.3 Vergleich

Da wir nun mit dem Ablauf der Algorithmen vertraut sind und wissen, dass Cole's Mergesort in der Theorie der schnellere Algorithmus ist, schauen wir uns die Ergebnisse der Messungen von Lasse Natvigs Testläufen an. Zuerst zwei Tabellen, die jeweils die Zeit, Anzahl der Prozessoren (P), die daraus errechneten Kosten sowie die Anzahl der Lese (#R) und Schreibeoperationen (#W). Die Daten dafür lieferte ihm der Simulator. Der ersten Tabelle liegt die Problemgröße n=128 und der zweiten n=256 zugrunde. Dessweiteren sind in diesen Tabellen die Messwerte zweier weiterer Algorithmen abgebildet, zum einen der parallele Odd-Even-Sort, der Batcher's Algorithmus recht ähnlich ist und zum anderen der sequentielle Insertsort in den drei Abstufungen schlimmster (worst), Durchschnitt (Average) und bester (best) Fall.

n=128						***n=256***					
Algorithmus	***Zeit***	***P***	***Kosten***	***#R***	***#W***	***Algorithmus***	***Zeit***	***P***	***Kosten***	***#R***	***#W***
Cole	18,2	493	8959,3	44,7	28,2	Cole	21,2	986	20863,8	104,6	65,2
Batcher	2,6	64	169,0	3,9	3,7	Batcher	3,4	128	428,2	9,9	9,4
Odd-Even	2,8	64	176,5	16,6	8,0	Odd-Even	5,3	128	683,7	65,9	34,8
Insert worst	148,7	1	148,7	8,3	8,3	Insert worst	592,4	1	592,4	32,9	32,9
Insert Average	76,2	1	76,2	4,3	4,2	Insert Average	303,3	1	303,3	17,0	16,8
Insert best	2,8	1	2,8	0,3	0,1	Insert best	5,6	1	5,6	5,1	0,3

Wir sehen, dass Batcher's Algorithmus in beiden Fällen wesentlich schneller ist als Cole's Algorithmus. Ebenso werden wesentlich weniger Prozessoren und Lese- und Schreibeoperationen gebraucht, also ist der Ressourceausnutzung um einiges geringer. Wir sehen ebenfalls, dass der ebenfalls relativ einfache Odd-Even-Sort auch schneller und wesentlich kostengünstiger ist, wobei zu beachten ist, dass es sich hier um zwei feste Problemgrößen hadelt. Dessweiteren bleibt in der Kostenfrage der sequentielle Insertsort auch nicht uninteresant, soll hier aber nicht weiter diskutiert werden.

Da die theoretische Betrachtung der Laufzeiten unserer beiden Algorithmen natürlich nicht falsch ist und Cole's Algorithmus bei steigender Problemgröße langsammer wächst als Batcher's Bitonicsort muss es eine Problemgröße geben ab der Cole's Algorithmus auch in der Praxis schneller ist. Diese Grenze lässt sich Dank der Messung von Natvig errechnen und sie wird in folgender Tabelle gezeigt.

Algorithmus	***n***	***Zeit***	***P***
ColeMergeSort	65.536	45.000	250.000
BitonicSort	65.536	12.000	33.000
ColeMergeSort	262.144	52.000	1.000.000
BitonicSort	262.144	15.000	130.000
ColeMergeSort	$2^{69}(\approx 6*10^{20})$	205.972	$2{,}3*10^{21}$
BitonicSort	$2^{69}(\approx 6*10^{20})$	205.194	$3{,}0*10^{20}$
ColeMergeSort	$2^{70}(\approx 1{,}2*10^{21})$	208.958	$4{,}6*10^{21}$
BitonicSort	$2^{70}(\approx 1{,}2*10^{21})$	211.107	$5{,}9*10^{20}$

Hier sehen wir, dass die Grenze, die die Problemgröße überschreiten muss mit 2^{69} (was ausgeschrieben ungefähr so aussieht: 600.000.000.000.000.000.000) sich in, für die Praxis, unrealistischen Dimensionen bewegt. Weiter bleibt nicht zu vergessen, dass Cole's Algorithmus nachwievor ungefähr achtmal soviele Prozessoren benötigt, erheblich mehr Speicher verbraucht und auch der Aufwand der Implementierung war bei Natvig mit 2000 Zeilen PIL-Code und 40 Arbeitstagen erheblich mehr als bei Batcher's Bitonicsort, der in 2 Tagen mit 200 PIL-Code Zeilen implementiert wurde.

4 Schlussbemerkung

Man könnte davonausgehen, dass die sehr komplexen parallelen Algorithmen nur unter Gebrauch von sehr hohen Beispielproblemgrößen und nahezu zu hohem Ressourcenaufwand vernünftig ausgewertet werden könnten. Bei Natvig's Testläufen haben jedoch einige hundert Prozessoren und kleine Speicher ausgereicht die Hauptmerkmale der Algorithmen herauszustellen. Für die Untersuchung sehr großer Problemgrößen beachte man folgende Punkte.

- Der zu untersuchende Algorithmus enthält in allen möglichen Größen die Problemgröße als Parameter.
- Alle in den Grenzen des Simulators möglichen Problemgrößen werden ausführlich getestet.
- Um den Aufwand der Analyse in sinvollen Grenzen zu halten, ist darauf zu achten, dass der zu untersuchende Algorithmus sowohl deterministisch als auch synchron arbeitet.

So ist es möglich dank Hochrechnungen die Analyse auch für Problemgrößen, die über die Möglichkeiten des Simulators hinausgehen, durchzuführen. Die, in dieser Arbeit angedeutete, Methode PRAM Algorithmen zu untersuchen, könnte dazu beitragen, das Loch zwischen Theorie und Praxis ein wenig zu verringern.

Literaturverzeichnis

- L. Natvig. 1990. Logarithmic Time Cost Optimal Parallel Sorting is Not Yet Fast in Practice! Division of Computer Systems and Telematics, The Norwegian Institute of Technology, The University of Trondheim, pages 486 – 493
- Mittschrift der Vorlesung „Parallele Programmierung“ Prof. Dr. Rita Loogen 2005, Universität Marburg
- Folgende Artikel aus http://de.wikipedia.org
  - PRAM
  - Paralleler Algorithmus
  - Maschinenmodelle
- Skript zur Praktische/Angewandte Informatik II von Silke Schomann 2004, TU Clausthal