Paralleles Sortieren

Stefan Lang

Interdisziplinäres Zentrum für Wissenschaftliches Rechnen
Universität Heidelberg
INF 368, Raum 532
D-69120 Heidelberg
phone: 06221/54-8264
email: Stefan.Lang@iwr.uni-heidelberg.de

WS 13/14

Themen

Parallele Sortierverfahren

- Mergesort
- Bitonisches Sortieren
- Quicksort

Einführung

- Es gibt eine ganze Reihe verschiedener Sortieralgorithmen: Heapsort, Bubblesort, Quicksort, Mergesort, ...
- Bei der Betrachtung paralleler Sortierverfahren beschränken wir uns auf
  - interne Sortieralgorithmen, d.h. solche, die ein Feld von (Ganz-) Zahlen im Speicher (wahlfreier Zugriff möglich!) sortieren, und
  - vergleichsbasierte Sortieralgorithmen, d.h. solche, bei denen die Grundoperationen aus Vergleich zweier Elemente und eventuellem Vertauschen besteht.
- Für eine allgemeine Einführung in Sortieralgorithmen sei auf [Sedgewick] verwiesen.
- Eingabe des parallelen Sortieralgorithmus besteht aus N Zahlen. Diese sind auf P Prozessoren verteilt, d.h. jeder Prozessor besitzt ein Feld der Länge N/P.
- Ergebnis eines parallelen Sortieralgorithmus ist eine sortierte Folge der Eingabezahlen, die wiederum auf die Prozessoren verteilt ist, d.h. Prozessor 0 enthält den ersten Block von N/P Zahlen, Prozessor 1 den zweiten usw.
- Wir behandeln zwei wichtige sequentielle Sortierverfahren: Mergesort und Quicksort.

Mergesort

- Mergesort basiert auf folgender Idee:

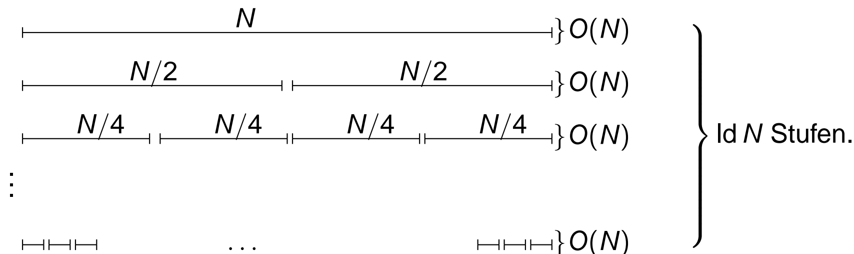
  Es sei eine Folge von N Zahlen zu sortieren. Angenommen wir teilen die Folge in zwei der Länge $\frac{N}{2}$ und sortieren diese jeweils getrennt, so können wir aus den beiden Hälften leicht eine sortierte Folge von N Zahlen erstellen, indem wir jeweils das nächste kleinste Element der Teilfolgen wegnehmen.

- Algorithmus von Mergesort

Input: $a = \langle a_0, a_1, \ldots, a_{N-1} \rangle$;
$l = \langle a_0, \ldots, a_{N/2-1} \rangle$; $r = \langle a_{N/2}, \ldots, a_{N-1} \rangle$;
sortiere l; sortiere r;
$i = j = k = 0$;
while ($i < N/2 \ \wedge \ j < N/2$)
{
  if ($l_i \leq r_j$) { $s_k = l_i$; $i++$; }
  else { $s_k = r_j$; $j++$; }
  $k++$;
}
while ($i < N/2$)
{
  $s_k = l_i$; $i++$; $k++$;
}
while ($j < N/2$)
{
  $s_k = r_j$; $j++$; $k++$;
}

Mergesort

- Ein Problem von Mergesort ist, dass zusätzlicher Speicherplatz (zusätzlich zur Eingabe) erforderlich ist. Obige Variante kann sicher stark verbessert werden, „sortieren am Ort“, d.h. in *a* selbst, ist jedoch nicht möglich.
- *Laufzeit:* Das Mischen der zwei sortierten Folgen (die drei **while**-Schleifen) braucht $O(N)$ Operationen. Die Rekursion endet in Tiefe $d = \text{ld}\, N$, die Komplexität ist also $O(N \,\text{ld}\, N)$.



- Es zeigt sich, dass dies die optimale asymptotische Komplexität für vergleichsbasierte Sortieralgorithmen ist.

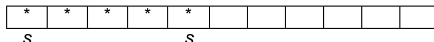
Quicksort

- Quicksort und Mergesort sind in gewissem Sinn komplementär zueinander.
- Mergesort sortiert rekursiv zwei (gleichgroße) Teilfolgen und mischt diese zusammen.
- Bei Quicksort zerteilt man die Eingabefolge in zwei Teilfolgen, so dass *alle* Elemente der ersten Folge kleiner sind als *alle* Elemente der zweiten Folge. Diese beiden Teilfolgen können dann getrennt (rekursiv) sortiert werden. Das Problem dabei ist, wie man dafür sorgt, dass die beiden Teilfolgen (möglichst) gleich groß sind, d.h. je etwa $N/2$ Elemente enthalten.
- Üblicherweise geht man so vor: Man wählt ein Element der Folge aus, z.B. das erste oder ein zufälliges und nennt es „Pivotelement“ Alle Zahlen kleiner gleich dem Pivotelement kommen in die erste Folge, alle anderen in die zweite Folge. Die Komplexität hängt nun von der Wahl des Pivotelementes ab:

$O(N \log N)$	falls immer in zwei gleichgroße Mengen zerlegt wird,
$O(N^2)$	falls immer in eine einelementige Menge und den Rest zerlegt wird.

- Bei einer zufälligen Auswahl des Pivots ist der zweite Fall extrem unwahrscheinlich. In der Regel ist Quicksort schneller als Mergesort.

Quicksort

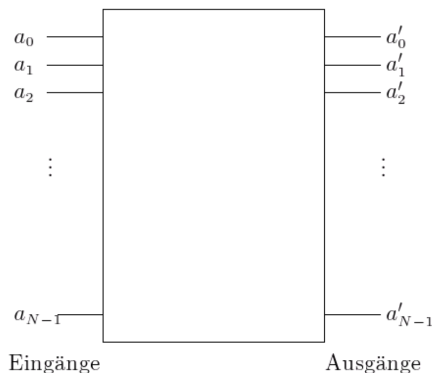



```
void Quicksort(int a[]; int first; int last)
{
    if (first ≥ last) return;                   // Ende der Rekursion
    // partitioniere
    wähle ein q ∈ [first, last];                // Pivotwahl
    x = a[q];
    swap(a, q, first);                          // bringe x an den Anfang
    s = first;                                  // Marke Folge 1: [first . . . s]
    for (i = first + 1; i ≤ last; i++)
        if (a[i] ≤ x)
        {
            s++;
            swap(a, s, i);                      // a[i] in erste Folge
        }
    swap(a, first, s);

    //  Nun gilt
    //    1  alle a[i] mit first ≤ i ≤ s sind ≤ x
    //    2  a[s] = x ist bereits in der richtigen Position!

    Quicksort(a, first, s − 1);                 // a[s] wird nicht mitsortiert
    Quicksort(a, s + 1, last);                  // beide Folgen zusammen eins weniger
}
```

Stack könnte zum Problem werden, falls N groß und worst case erreicht wird (N rekursive Aufrufe).

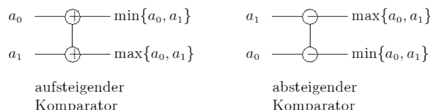
Sortiernetzwerke I

- Ein Sortiernetzwerk übersetzt eine Folge von N unsortierten Zahlen in eine Folge von N aufsteigend oder absteigend sortierten Zahlen.
- An den N Eingängen werden die unsortierten Zahlen angelegt, am Ausgang erscheint die sortierte Folge.



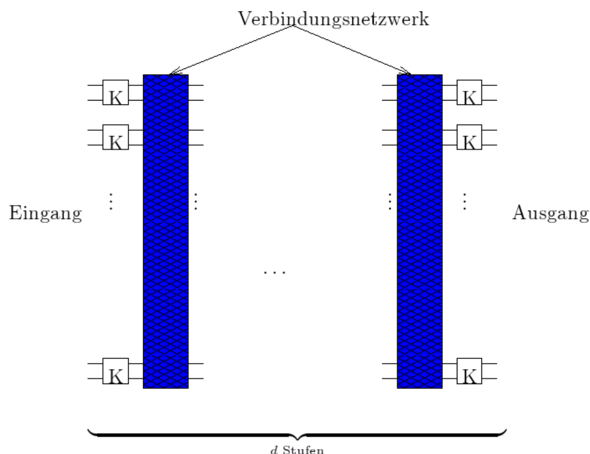
Sortiernetzwerke II

- Intern ist das Sortiernetzwerk aus elementaren Schaltzellen, sogenannten Komparatoren, aufgebaut, die genau zwei Zahlen aufsteigend oder absteigend sortieren.



- Eine Anzahl von Komparatoren wird zu einer sogenannten „Stufe“ zusammengefasst.
- Das ganze Netzwerk besteht aus mehreren Stufen, die ihrerseits durch ein Verbindungsnetzwerk verbunden sind.
- Die Anzahl der Stufen wird auch als Tiefe des Sortiernetzwerkes bezeichnet.

Sortiernetzwerke III



- Alle Komparatoren einer Stufe arbeiten parallel. Sortiernetzwerke können gut in Hardware realisiert werden oder lassen sich in entsprechende Algorithmen für Parallelrechner übertragen (weshalb wir sie hier studieren wollen).

Bitonisches Sortieren

Wir betrachten zunächst einen Algorithmus, der sowohl Eigenschaften von Quicksort als auch von Mergesort hat:

- Wie bei Mergesort wird die Eingabefolge der Länge N in zwei Folgen der Länge $N/2$ zerlegt. Dadurch ist die Rekursionstiefe immer ld N (allerdings wird die Gesamtkomplexität schlechter sein!)
- Wie bei Quicksort sind alle Elemente der einen Folge kleiner als alle Elemente der anderen Folge und können somit unabhängig voneinander sortiert werden.
- Das Zerlegen kann voll parallel mit $N/2$ Komparatoren geschehen, d.h. der Algorithmus kann mit einem Sortiernetzwerk realisiert werden.

Bitonische Folge

Definition (Bitonische Folge)

Eine Folge von N Zahlen heißt *bitonisch*, genau dann, wenn eine der beiden folgenden Bedingungen gilt

1. Es gibt einen Index $0 \leq i < N$, so dass

$$\underbrace{a_0 \leq a_1 \leq \cdots \leq a_i}_{\text{aufsteigend}} \text{ und } \underbrace{a_{i+1} \geq a_{i+2} \geq \cdots \geq a_{N-1}}_{\text{absteigend}}$$

2. Man kann die Indizes zyklisch schieben, d.h. $a'_i = a_{(i+m)\,\%\,N}$, so dass die Folge a' die Bedingung 1 erfüllt.

Normalform bitonischer Folgen

- Jede bitonische Folge lässt sich auf folgende Form bringen:

$$a'_0 \leq a'_1 \leq \cdots \leq \mathbf{a'_k} > \mathbf{a'_{k+1}} \geq a'_{k+2} \geq \cdots \geq \mathbf{a'_{N-1}} < \mathbf{a'_0}$$

- Wesentlich ist, dass das letzte Element des aufsteigenden Teils größer ist als der Anfang des absteigenden Teils.
- Ebenso ist das Ende des absteigenden Teils kleiner als der Anfang des aufsteigenden Teils.

Beweis.

Für die Eingabefolge a gelte 1 aus der Definition der bitonischen Folgen. Es sei $a_0 \leq a_1 \leq \cdots \leq a_i$ der aufsteigende Teil. Entweder es gilt nun $a_i > a_{i+1}$ oder es gibt ein $j \geq 0$ mit

$$a_i \leq a_{i+1} = a_{i+2} = \cdots = a_{i+j} > a_{i+j+1}$$

(oder die Folge ist trivial und besteht aus lauter gleichen Elementen). Somit kann man die $a_{i+1}, \ldots, a_{i+j}$ zum aufsteigenden Teil hinzunehmen und es gilt $k = i + j$.

...

Normalform bitonischer Folgen

Beweis.

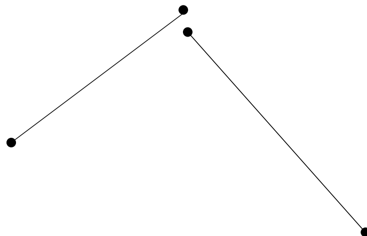
...

Ebenso verfährt man mit dem absteigenden Teil. Entweder ist schon $a_{N-1} < a_0$ oder es gibt ein l, so dass

$$a_{N-1} \geq a_0 = a_1 = \cdots = a_l < a_{l+1}$$

(oder die Folge ist trivial). In diesem Fall nimmt man $a_0, \ldots, a_l$ zum absteigenden Teil hinzu. □

- Man hat also die folgende Situation:



Min-Max Charakterisierung bitonischer Folgen

- In der folgenden Definition brauchen wir den Begriff „zyklischer Nachbar“, d.h. wir setzen

$$
\begin{aligned}
a_{i \oplus l} &:= a_{(i+l)\,\%\,N} \\
a_{i \ominus l} &:= a_{(i+N-l)\,\%\,N}
\end{aligned}
$$

für $l \leq N$

- Damit kommen wir zur
  Min-Max Charakterisierung bitonischer Folgen

Definition

Ein a_k heisst (lokales) Maximum, falls es ein $l \geq 0$ gibt mit

$$a_{k \ominus (l+1)} < a_{k \ominus l} = \cdots = a_{k \ominus 1} = a_k > a_{k \oplus 1}$$

Entsprechend heisst a_k (lokales) Minimum, falls es ein $l \geq 0$ gibt mit

$$a_{k \ominus (l+1)} > a_{k \ominus l} = \cdots = a_{k \ominus 1} = a_k < a_{k \oplus 1}$$

Min-Max Charakterisierung bitonischer Folgen

- Für nicht triviale Folgen (d.h. es sind nicht alle Elemente identisch) gilt damit:
  Eine Folge $a = \langle a_0, \dots, a_{N-1} \rangle$ ist bitonisch genau dann, wenn sie genau ein Minimum und genau ein Maximum hat (oder trivial ist).

Beweis.

$\Rightarrow$ gilt wegen der Normalform. a'_k ist das Maximum, a'_{N-1} das Minimum.

$\Leftarrow$ Wenn a genau ein Minimum bzw. Maximum hat, zerfällt sie in einen aufsteigenden und einen absteigenden Teil, ist also bitonisch.

Bitonische Zerlegung

Nach obigen Vorbereitungen können wir die Bitonischen Zerlegung charakterisieren

- Es sei $s = \langle a_0, \ldots, a_{N-1} \rangle$ eine gegebene bitonische Folge der Länge N.
- Wir konstruieren zwei neue Folgen der Länge $N/2$ (N gerade) auf folgende Weise:

$$\begin{aligned} s_1 &= \left\langle \min\{a_0, a_{\frac{N}{2}}\}, \min\{a_1, a_{\frac{N}{2}+1}\}, \ldots, \min\{a_{\frac{N}{2}-1}, a_{N-1}\} \right\rangle \\ s_2 &= \left\langle \max\{a_0, a_{\frac{N}{2}}\}, \max\{a_1, a_{\frac{N}{2}+1}\}, \ldots, \max\{a_{\frac{N}{2}-1}, a_{N-1}\} \right\rangle \end{aligned} \tag{1}$$

  Für s_1, s_2 gilt:

  1. Alle Elemente von s_1 sind kleiner oder gleich allen Elementen in s_2.
  2. s_1 und s_2 sind bitonische Folgen.

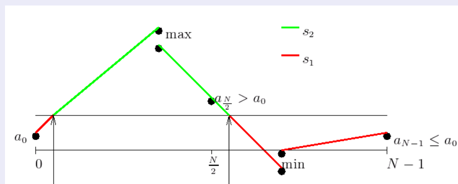
- Offensichtlich können s_1 und s_2 aus s mit Hilfe von $N/2$ Komparatoren konstruiert werden.

Bitonische Zerlegung: Beweis I

Beweis.

Wir überlegen uns das graphisch anhand von verschiedenen Fällen:

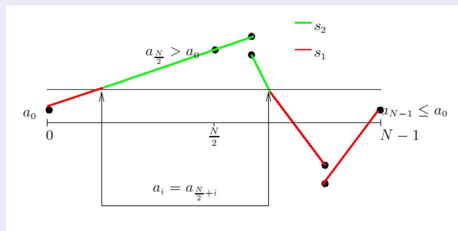
1. Es sei $a_0 < a_{\frac{N}{2}}$: Wir wissen, dass jede bitonische Folge genau ein Maximum und Minimum hat. Da $a_0 < a_{\frac{N}{2}}$ kann das Maximum nur zwischen a_0 und $a_{\frac{N}{2}}$ oder zwischen $a_{\frac{N}{2}}$ und a_{N-1} liegen.
   In diesem Fall gilt also $\min\{a_0, a_{\frac{N}{2}}\} = a_0$ und solange $a_i \leq a_{\frac{N}{2}+i}$ enthält s_1 die Elemente aus dem aufsteigenden Teil der Folge. Irgendwann gilt $a_i > a_{\frac{N}{2}+i}$, und dann enthält s_1 Elemente aus dem absteigenden Teil und anschließend wieder aus dem aufsteigenden Teil. Aus der Graphik ist sofort ersichtlich, dass s_1 und s_2 die Bedingungen 1 und 2 von oben erfüllen.
2. Maximum zwischen a_0 und $a_{\frac{N}{2}}$:



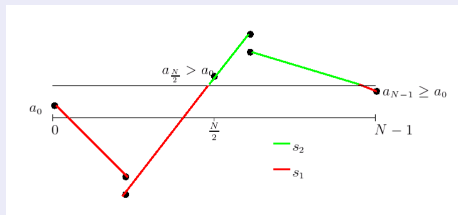
Bitonische Zerlegung: Beweis II

Beweis.

1. Maximum zwischen $a_{\frac{N}{2}}$ und a_{N-1}:



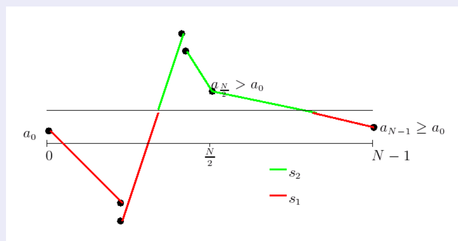
2. Minimum zwischen a_0 und $a_{\frac{N}{2}}$:



Bitonische Zerlegung: Beweis III

Beweis.

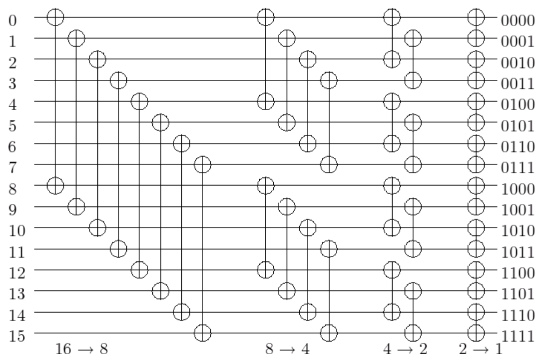
1. Minimum und Maximum zwischen a_0 und $a_{\frac{N}{2}}$:

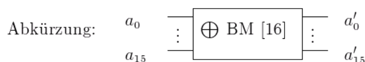


2. Die anderen Fälle: $a_0 = a_{\frac{N}{2}}$ bzw. $a_0 > a_{\frac{N}{2}}$ überlegt man sich analog.

Bitonische Folgen

- Um eine *bitonische* Folge der länge $N > 2$ zu sortieren, wendet man die bitonische Zerlegung rekursiv auf beide Hälften an. Die Rekursionstiefe ist natürlich d.
- Ein bitonisches Sortiernetzwerk, um eine bitonische Folge von 16 Zahlen zu sortieren, hat folgende Gestalt:





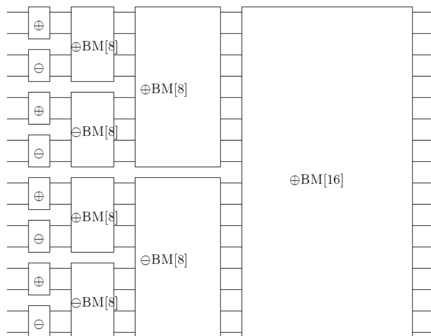
Bitonische Folgen

- Um nun N *unsortierte* Zahlen zu sortieren, muss man diese in eine bitonische Folge verwandeln.
- Wir bemerken zunächst, dass man durch bitonisches Zerlegen eine bitonische Folge auch leicht in absteigender Reihenfolge sortieren kann. Dazu ist in der bitonischen Zerlegung (1) max mit min zu vertauschen. Entsprechend sind im Netzwerk die $\oplus$-Komparatoren durch $\ominus$-Komparatoren zu ersetzen. Das entsprechende Netzwerk bezeichnet man mit $\ominus$BM[N].

Bitonische Folgen

Bitonische Folgen erzeugt man folgendermaßen:

- Eine Folge der Länge zwei ist immer bitonisch, da $a_0 \leq a_1$ oder $a_0 > a_1$. Hat man zwei bitonische Folgen der Länge N, so sortiert man die eine mit $\oplus$BM[N] aufsteigend und die andere mit $\ominus$BM[N] absteigend und erhält so eine bitonische Folge der Länge $2N$. Das vollständige Netzwerk zum Sortieren von 16 Zahlen sieht so aus:



Bitonische Zerlegung

- Betrachten wir die Komplexität des bitonischen Sortierens.
- Für die Gesamttiefe $d(N)$ des Netzwerkes bei $N = 2^k$ erhalten wir

$$
\begin{aligned}
d(N) &= \underbrace{\operatorname{ld} N}_{\oplus\mathrm{BM}[N]} + \underbrace{\operatorname{ld} \tfrac{N}{2}}_{\mathrm{BM}[N/2]} + \operatorname{ld} \tfrac{N}{4} + \ldots + \operatorname{ld} 2 = \\
&= k + k-1 + k-2 + \ldots + 1 = \\
&= O(k^2) = O(\operatorname{ld}^2 N).
\end{aligned}
$$

- Damit ist die Gesamtkomplexität für bitonisches Mischen also $O(N \operatorname{ld}^2 N)$.

Bitonisches Sortieren auf dem Hypercube

- Wir überlegen nun, wie bitonisches Sortieren auf dem Hypercube realisiert werden kann.
- Dazu unterscheiden wir die Fälle $N = P$ (jeder hat eine Zahl) und $N \gg P$ (jeder hat einen Block von Zahlen).
- Der Fall $N = P$:
  Bitonisches Sortieren läst sich optimal auf den Hypercube abbilden! Am 4-stufigen Sortiernetzwerk erkennt man, daß nur nächste Nachbar Kommunikation erforderlich ist! Im Schritt $i = 0, 1, \ldots, d - 1$ kommuniziert Prozessor p mit Prozessor $q = p \oplus 2^{d-i-1}$ (das $\oplus$ bedeutet hier wieder `XOR`). Offensichtlich erfordert aber jeder Vergleich eine Kommunikation, die Laufzeit wird also von den Aufsetzzeiten t_s der Nachrichten dominiert.
- Der Fall $N \gg P$:
  Nun erhält jeder Prozessor einen Block von $K = \frac{N}{P}$ (N durch P teilbar) Zahlen.
  Nun kann man die Aufsetzzeiten amortisieren, da jeder Prozessor K Vergleiche durchführt, er „simuliert“ sozusagen K Komparatoren des Netzwerkes.

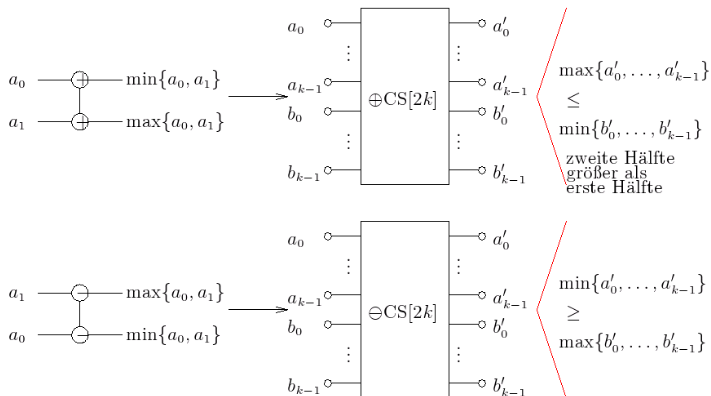
Bitonisches Sortieren auf dem Hypercube

- Es bleibt noch das Problem, dass bitonisches Sortieren keine optimale sequentielle Komplexität besitzt, und somit keine isoeffiziente Skalierung erreicht werden kann.
- Dies kann man verbessern durch folgenden Ansatz:

  > *Es sei $P = 2^k$. Wir stellen uns bitonisches Sortieren auf P Elementen vor, nur dass jedes Element nun seinerseits eine Folge von K Zahlen ist.*

- Es zeigt sich, dass man $P \cdot K$ Zahlen sortieren kann, indem man die Komparatoren für zwei Zahlen durch Bausteine für $2K$ Zahlen ersetzt, wie sie in folgendem Bild dargestellt sind:

Bitonisches Sortieren auf dem Hypercube



Bitonisches Sortieren auf dem Hypercube

- Dabei sind die Folgen $\langle a_0, \ldots, a_{k-1} \rangle$ bzw. $\langle b_0, \ldots, b_{k-1} \rangle$ bereits aufsteigend sortiert, und die Ausgabefolgen $\left\langle a'_0, \ldots, a'_{k-1} \right\rangle$ bzw. $\left\langle b'_0, \ldots, b'_{k-1} \right\rangle$ *bleiben* aufsteigend sortiert.
- CS[2k] kann auf zwei Prozessoren in $O(K)$ Zeit durchgeführt werden, indem jeder dem anderen seinen Block schickt, jeder die beiden Blöcke mischt und die richtige Hälfte behält.

Bitonisches Sortieren auf dem Hypercube

- Für die Gesamtkomplexität erhalten wir:

$$T_P(N,P) = c_1 \underbrace{\log^2 P}_{\text{Stufen}} \underbrace{\frac{N}{P}}_{\substack{\text{Aufwand}\\\text{pro Stufe}}} + \underbrace{c_2 \frac{N}{P} \log \frac{N}{P}}_{\substack{\text{einmaliges}\\\text{Sortieren der}\\\text{Eingabeblöcke}\\\text{(und Ausgabe bei}\\\ominus)}} + \underbrace{c_3 \log^2 P \frac{N}{P}}_{\text{Kommunikation}}$$

- Somit gilt für die Effizienz

$$\begin{aligned}
E(N,P) &= \frac{T_S}{T_P P} = \frac{c_2 N \log N}{\left(c_2 \frac{N}{P} \log \frac{N}{P} + (c_1 + c_3) \frac{N}{P} \log^2 P\right) P} = \\
&\underset{\substack{\log \frac{N}{P} = \\ \log N - \log P}}{=} \frac{1}{\frac{\log N - \log P}{\log N} + \frac{c_1 + c_3}{c_2} \cdot \frac{\log^2 P}{\log N}} = \\
&= \frac{1}{1 - \frac{\log P}{\log N} + \frac{c_1 + c_3}{c_2} \cdot \frac{\log^2 P}{\log N}}
\end{aligned}$$

Bitonisches Sortieren auf dem Hypercube

- Eine isoeffiziente Skalierung erfordert somit

$$\frac{\log^2 P}{\log N} = K$$

  und somit $N(P) = O(P^{\log P})$,
  da $\log N(P) = \log^2 P \iff N(P) = 2^{\log^2 P} = P^{\log P}$).
  Wegen $W = O(N \log N)$ folgt aus $N \log N = P^{\log P} \cdot \log^2 P$ für die Isoeffizienzfunktion

$$W(P) = O(P^{\log P} \log^2 P).$$

- Der Algorithmus ist also *sehr* schlecht skalierbar!!

Paralleles Quicksort

- Quicksort besteht aus der Partitionierungsphase und dem rekursiven Sortieren der beiden Teilmengen.
- Naiv könnte man versuchen, immer mehr Prozessoren zu verwenden, je mehr unabhängige Teilprobleme man hat.
- Dies ist nicht kostenoptimal, da die Teilprobleme immer kleiner werden.
- Formal erhält man:

$$
\begin{aligned}
T_S(N) &= \underbrace{N + 2\frac{N}{2} + 4\frac{N}{4} + \ldots}_{\substack{\log N \text{ Schritte} \\ \text{(average)}}} = \\
&= N \log N \\
T_P(N) &\underset{\substack{N = P \text{ naive} \\ \text{Variante}}}{=} N + \frac{N}{2} + \frac{N}{4} + \cdots = \\
&= 2N
\end{aligned}
$$

- Für die Kosten erhält man ($P = N$)

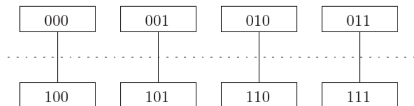
$$
P \cdot T_P(N) = 2N^2 > N \log N
$$

also nicht optimal.

Paralleles Quicksort

Fazit: Man muss den Partitionierungsschritt parallelisieren, und das geht so:

- Es sei $P = 2^d$, jeder hat N/P Zahlen.
- Wir benutzen die Tatsache, dass ein Hypercube der Dimension d in zwei der Dimension $(d-1)$ zerfällt, und jeder im ersten Hypercube hat genau einen Nachbarn im zweiten:



- Nun wird das Pivot x gewählt und an alle verteilt, dann tauschen die Partnerprozessoren ihre Blöcke aus.
- Die Prozessoren im ersten Hypercube behalten alle Elemente $\leq x$, die im zweiten Hypercube alle Elemente $> x$.
- Dies macht man rekursiv d mal und sortiert dann lokal in jedem Prozessor mit Quicksort.
- Komplexität (average case: jeder behält immer $\frac{N}{P}$ Zahlen):

$$T_P(N,P) = \underbrace{c_1 \operatorname{ld} P \cdot \frac{N}{P}}_{\text{split}} + \underbrace{c_2 \operatorname{ld} P \cdot \frac{N}{P}}_{\text{Komm.}} + \underbrace{c_3 \frac{N}{P} \operatorname{ld} \frac{N}{P}}_{\text{loka. Quicksort}} + \underbrace{c_4 d^2}_{\substack{\text{Pivot} \\ \text{Broadcast}}} \;;$$

Paralleles Quicksort

- Der Aufwand für den Broadcast des Pivot ist $\operatorname{ld} P + \operatorname{ld} \frac{P}{2} + \cdots = d + d - 1 + d - 2 + \cdots = O(d^2)$.
- Für die Effizienz erhält man

$$
\begin{aligned}
E(N,P) &= \frac{c_3 N \operatorname{ld} N}{\left(c_3 \frac{N}{P} \operatorname{ld} \frac{N}{P} + (c_1 + c_2) \frac{N}{P} \operatorname{ld} P + c_4 \operatorname{ld}^2 P\right) P} = \\
&= \frac{1}{\frac{\operatorname{ld} N - \operatorname{ld} P}{\operatorname{ld} N} + \frac{c_1 + c_2}{c_3} \cdot \frac{\operatorname{ld} P}{\operatorname{ld} N} + \frac{c_4}{c_3} \cdot \frac{P \operatorname{ld}^2 P}{N \operatorname{ld} N}} = \\
&= \frac{1}{1 + \left(\frac{c_1 + c_2}{c_4} - 1\right) \frac{\operatorname{ld} P}{\operatorname{ld} N} + \frac{c_4}{c_3} \cdot \frac{P \operatorname{ld}^2 P}{N \operatorname{ld} N}} .
\end{aligned}
$$

Paralleles Quicksort

- Für eine isoeffiziente Skalierung ist der Term aus dem Broadcast entscheidend:

$$\frac{P \operatorname{ld}^2 P}{N \operatorname{ld} N} = O(1),$$

für $N = P \operatorname{ld} P$ erhalten wir

$$\begin{aligned} N \operatorname{ld} N &= (P \operatorname{ld} P)\ \operatorname{ld}(P \operatorname{ld} P) = (P \operatorname{ld} P)\ (\operatorname{ld} P + \underbrace{\operatorname{ld} \operatorname{ld} P}_{\substack{\text{sehr}\\\text{klein!}}}) \approx \\ &\approx P \operatorname{ld}^2 P. \end{aligned}$$

- Mit einem Wachstum $N = O(P \operatorname{ld} P)$ sind wir also auf der sicheren Seite (es genügt sogar etwas weniger).
- Für die Isoeffizienzfunktion gilt wegen $W = N \log N$

$$W(P) = O(P \operatorname{ld}^2 P),$$

also deutlich besser als bei bitonischem Sortieren.

Empfindlichkeit gegenüber Pivotwahl

- Das Problem dieses Algorithmus ist, dass die Wahl eines schlechten Pivotelements zu einer schlechten Lastverteilung in *allen* nachfolgenden Schritten führt und somit zu schlechtem Speedup.
- Wird das allererste Pivot maximal schlecht gewählt, so landen alle Zahlen in *einer* Hälfte und der Speedup beträgt höchstens noch $P/2$.
- Bei gleichmäßiger Verteilung der zu sortierenden Elemente kann man den Mittelwert der Elemente eines Prozessors als Pivot wählen.