Bitonisches Sortieren

Der folgende parallele Algorithmus zum *bitonischen Sortieren* stammt von K.E. Batcher 1968. Eine Folge $a_1, a_2, \ldots, a_n$ von ganzen Zahlen heißt *bitonisch*, wenn der erste Teil der Folge aufsteigend und der zweite Teil absteigend sortiert ist, oder wenn man die Folgenglieder so verschieben kann (wie beim Rundschieben von Schieberegistern), daß diese Bedingung gilt.

Beispiel: Bitonische Folgen sind

7, 12, 13, 16, 29, 17, 12, 1 oder
16, 29, 17, 12, 1, 7, 12, 13 oder
1, 2, 3, 4, 5, 6, 7, 8.

Der Algorithmus von Batcher arbeitet folgendermaßen. Angenommen es liegt bereits eine bitonische Folge vor. Dann mischt man die beiden sortierten Teilfolgen zu einer sortierten Folge und ist fertig. Anderenfalls geht man rekursiv vor und versucht den gewünschten Zustand herzustellen, indem man die beiden Hälften der Folge sortiert, die erste aufsteigend, die zweite absteigend.

Beispiel: Abb. 1 zeigt die Einzelschritte des Verfahrens für eine Folge mit acht Elementen. Man benötigt drei Sortierphasen.

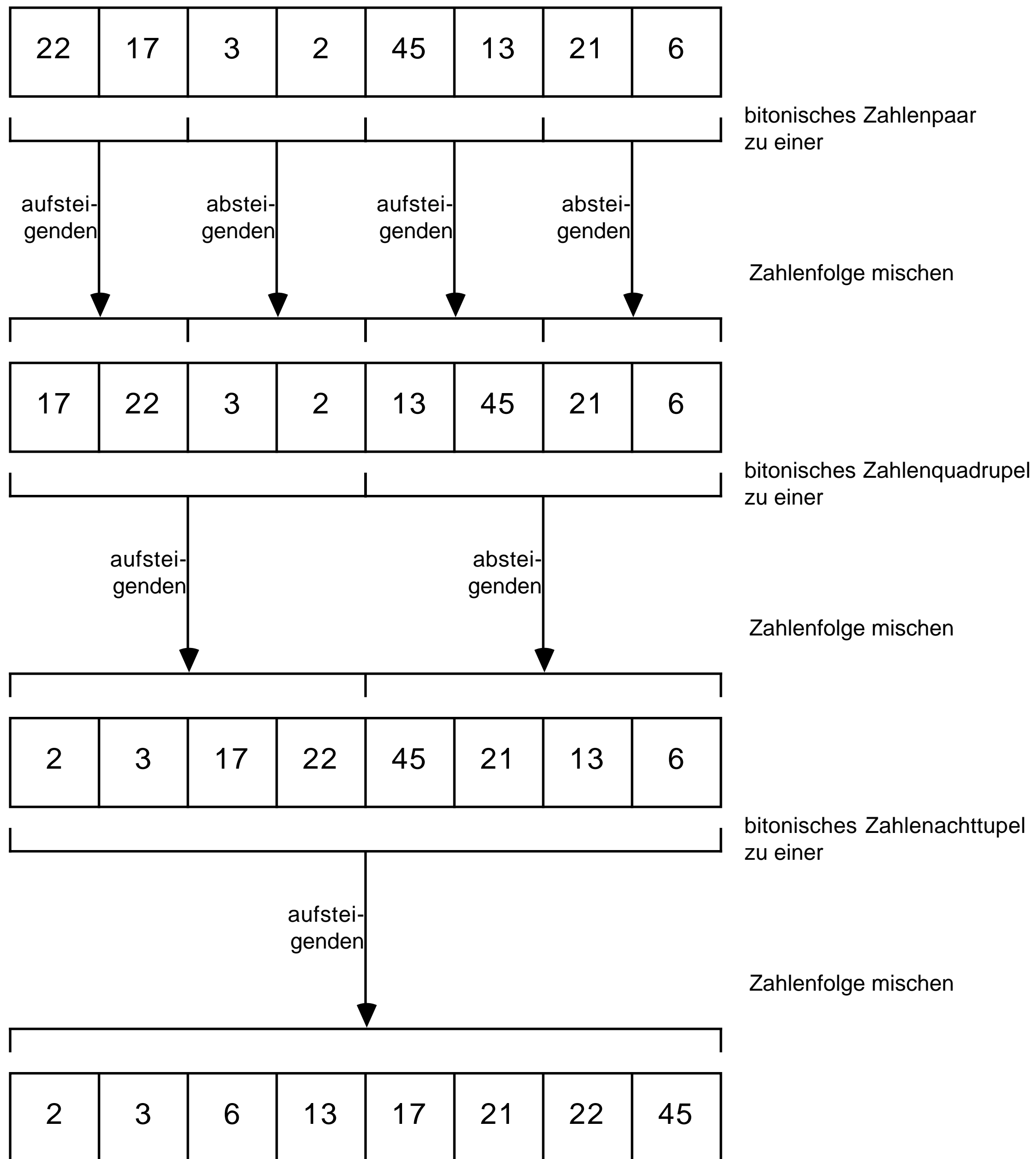

Abb. 1: Prinzip des bitonischen Sortierens

Zum Mischen einer bitonischen Folge $a_1, a_2, ..., a_{2n}$ der Länge 2n geht man wie folgt vor:

1. Schritt: Jedes Element a_j, $1 \leq j \leq n$, der ersten Hälfte wird mit dem korrespondierenden Element a_{j+n} der zweiten Hälfte verglichen; beide werden vertauscht, wenn sie in der falschen Reihenfolge stehen, falls also $a_j > a_{j+n}$ ist.

Nach diesem Schritt sind alle Elemente in der ersten Hälfte der Folge kleiner als alle Elemente in der zweiten Hälfte, und beide Hälften sind wieder bitonisch.

2. Schritt: Man wendet das Verfahren rekursiv getrennt auf die beiden Hälften der Folge an.

Offenbar ist die Folge sortiert, wenn die Rekursion abbricht..

Mit Hilfe eines parallelen Algorithmus realisiert man das Verfahren wie folgt: Man verwendet hierzu einen Prozessor zum Vergleichen und Vertauschen zweier Zahlen (Abb. 2) mit folgendem Funktionsverhalten:

$X' := \min(X,Y)$,

$Y' := \max(X,Y)$, $X,Y,X',Y' \in Z$.

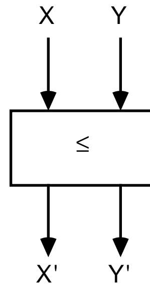

Abb. 2: Vergleicher

Offenbar müssen nur alle Paare a_j und a_{j+n}, $1 \le j \le n$, jeweils in einen der n Prozessoren zusammengeführt, verglichen und anschließend ggf. vertauscht an ihre Plätze j und j+n zurücktransportiert werden. Die beiden Hälften werden anschließend rekursiv bitonisch gemischt. Abb. 3 zeigt das entsprechende Netzwerk.

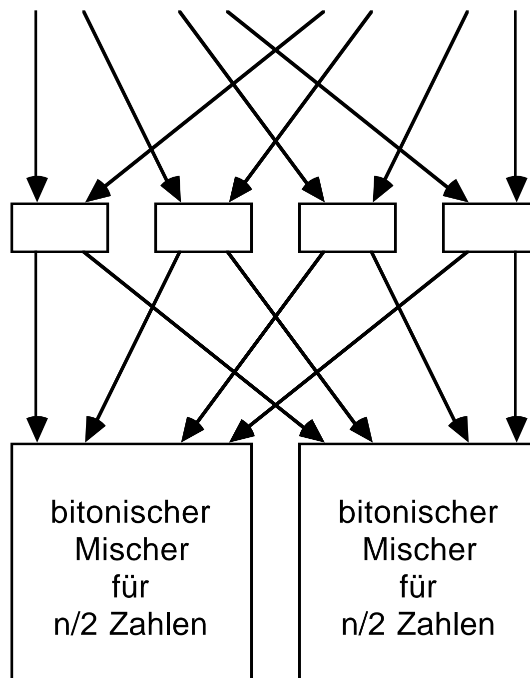

Abb. 3: Prozessornetzwerk für bitonisches Mischen

Die Zusammenführung der Operanden der Vergleichsprozessoren bezeichnet man als *Shuffling* (von engl. shuffle=Karten mischen). Dieser Bezeichnung liegt folgende Anschauung zugrunde: Man stelle sich einen Stapel von acht Spielkarten vor, die von oben nach unten mit 1 bis 8 durchnumeriert sind. Mischt man die Karten nach der amerikanischen Methode, so teilt man das Spiel in zwei gleichgroße Hälften, die man mit beiden Händen zu einem neuen Stapel zusammenschiebt. Führt man dies exakt aus, so befindet sich nachher abwechselnd eine Karte der einen und der anderen Hälfte hintereinander (Abb. 4). Umgekehrt stellt man die ursprüngliche Reihenfolge der Karten wieder her, indem man die Karten des Stapels abwechselnd auf zwei Häufchen verteilt, die man anschließend aufeinanderlegt. Diese Operation heißt analog *Unshuffling*.
Das Netzwerk aus Abb. 5 besteht also aus einer Shuffle-Stufe gefolgt von einer Vergleichs-/Vertauschungsstufe (*Exchange-Stufe*) und einer Unshuffle-Stufe. Man nennt es daher *Shuffle-Exchange-Netzwerk* (der Ordnung 8). Es besitzt eine Vielzahl von wichtigen praktischen Anwendungen, von denen hier aber nur das Sortierproblem behandelt wird.

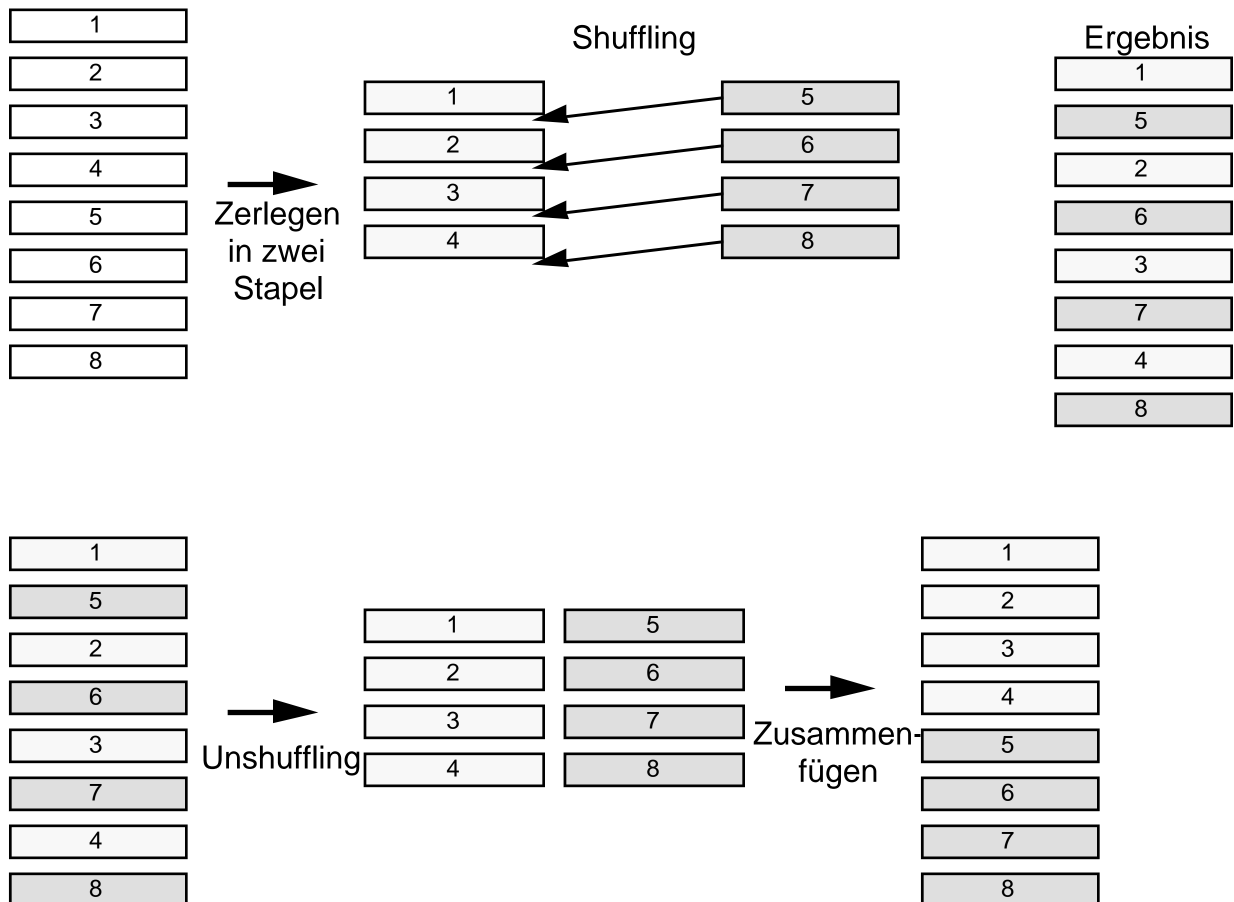

Abb. 4: Shuffling und Unshuffling

Das Netzwerk in Abb. 3 realisiert also den ersten Schritt des bitonischen Mischens. Alle weiteren Schritte führt man rekursiv nach dem gleichen Verfahren durch, da die beiden Hälften selbst wieder bitonische Folgen bilden. Man schaltet also zwei parallele Shuffle-Exchange-Netzwerke der Ordnung 4 hinter das Netzwerk in Abb. 3, und danach noch vier weitere der Ordnung 2. Als Endergebnis erhält man das Netzwerk in Abb. 5, das man zum Netzwerk in Abb. 6 vereinfachen kann, wenn man die Übergabepunkte wegläßt, die in Abb. 5 nur zur Veranschaulichung der Shuffle- und Unshuffle-Stufen eingefügt sind.

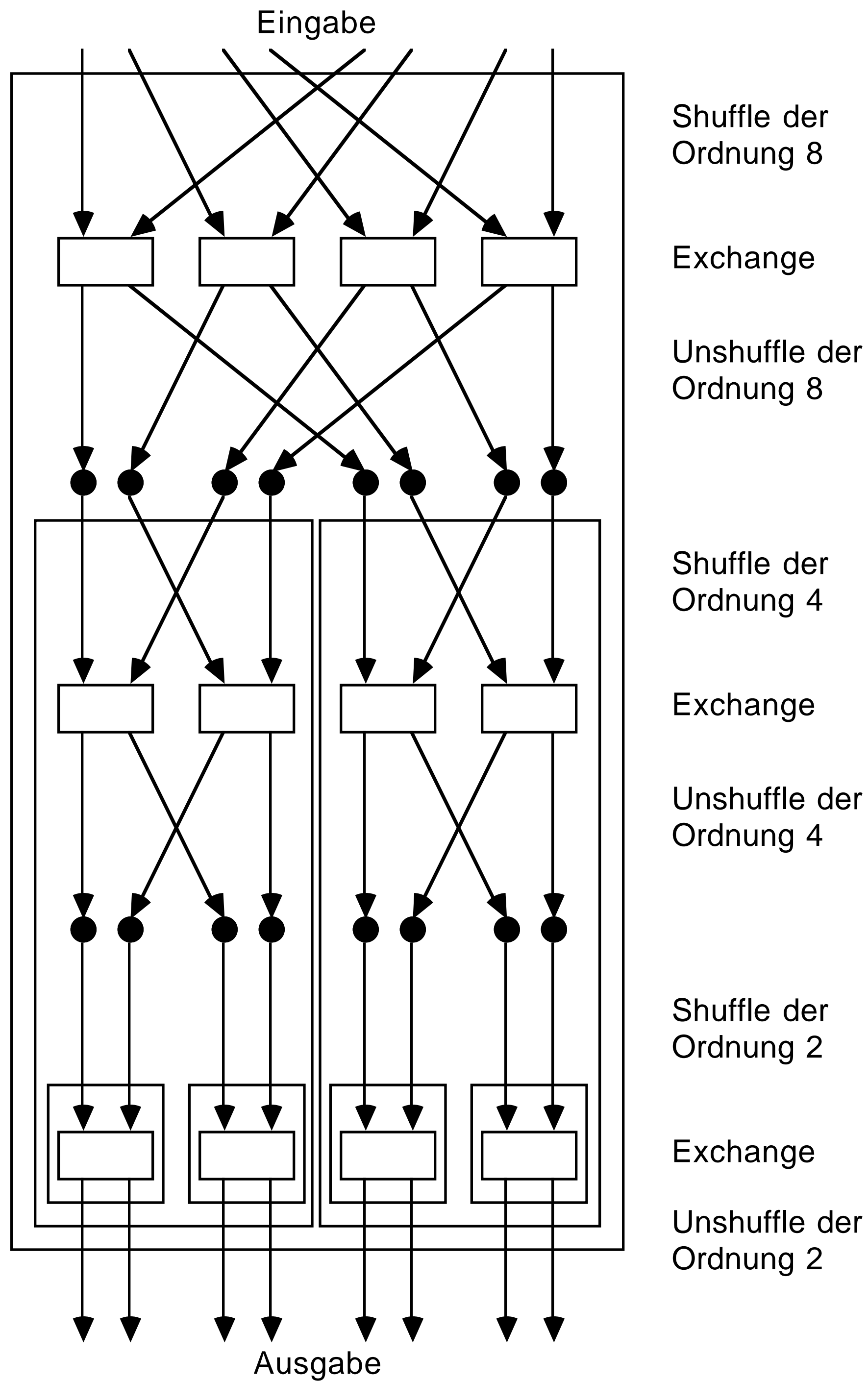

Abb. 5: Netzwerk für bitonisches Mischen von acht Zahlen

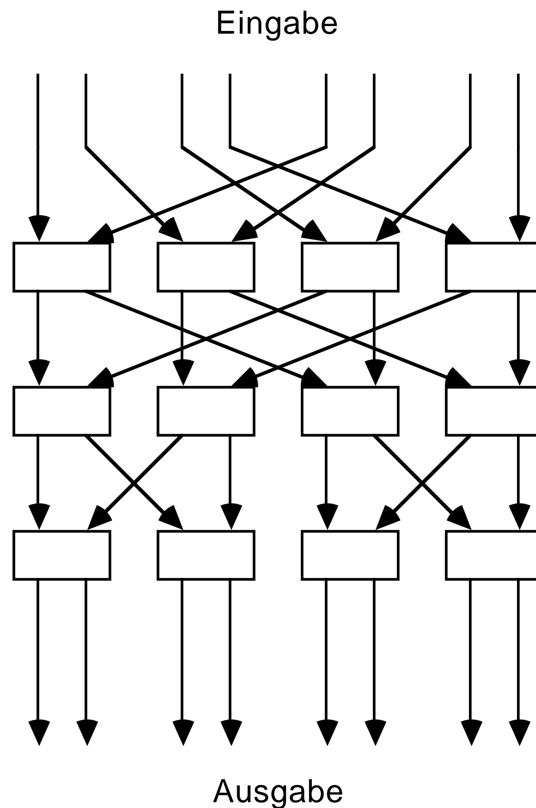

Abb. 6: Vereinfachtes Netzwerk für bitonisches Mischen von acht Zahlen

Offenbar benötigt man für ein vollständiges Netzwerk zum bitonischen Mischen von n Zahlen genau $[\log_2 n]$ Shuffle-Exchange-Stufen und damit auch $[\log_2 n]$ Takte, wobei $[\log_2 n]$ die kleinste ganze Zahl bezeichnet, die größer oder gleich $\log_2 n$ ist. Soweit das bitonische Mischen als grundlegende Operation. Nun zum Sortieren. Wie stellt man aus der anfangs unsortierten Zahlenfolge eine bitonische Folge her, die man anschließend zur sortierten Folge mischt?

Der Einfachheit halber sei die Länge der Folge $a_1,a_2,...,a_n$ eine Potenz von 2 ist, also $n=2^r$ für ein geeignetes $r\in$ IN. Anderenfalls ergänzt man die Eingabefolge zur nächsthöheren Zweierpotenz um Zahlen der Größe ∞.

Der Algorithmus arbeitet wiederum rekursiv. Im ersten Schritt ordnet man die Elemente der Folge so an, daß je vier aufeinanderfolgende Elemente eine bitonische Folge bilden. Dies ist sehr einfach: Von zwei benachbarten Paaren von Elementen sortiert man das erste aufsteigend und das zweite absteigend. Nun besteht die Folge aus $^n/_4$ bitonischen Folgen der Länge 4.

Anschließend verwendet man den bereits bekannten Algorithmus zum bitonischen Mischen und erzeugt aus den $^n/_4$ bitonischen Folgen gleichviele Folgen, die im Wechsel aufsteigend und absteigend sortiert sind. Ergebnis: $^n/_8$ bitonische Folgen der Länge 8.

Mit diesen Folgen arbeitet man nach dem gleichen Verfahren weiter, bis eine sortierte Folge entsteht.

Das Prozessornetzwerk für diesen Algorithmus ergibt sich unmittelbar. Man benötigt jedoch, um aufsteigend *und* absteigend sortierte Folgen herstellen zu können, neben dem Prozessor für "≤" gemäß Abb. 2 noch einen Prozessor für "≥" mit folgendem analogen Funktionsverhalten:

$X':=\max(X,Y)$,

$Y':=\min(X,Y)$, $X,Y,X',Y' \in Z$.

Abb. 7 zeigt das vollständige Netzwerk für die Sortierung von n=16 Zahlen.

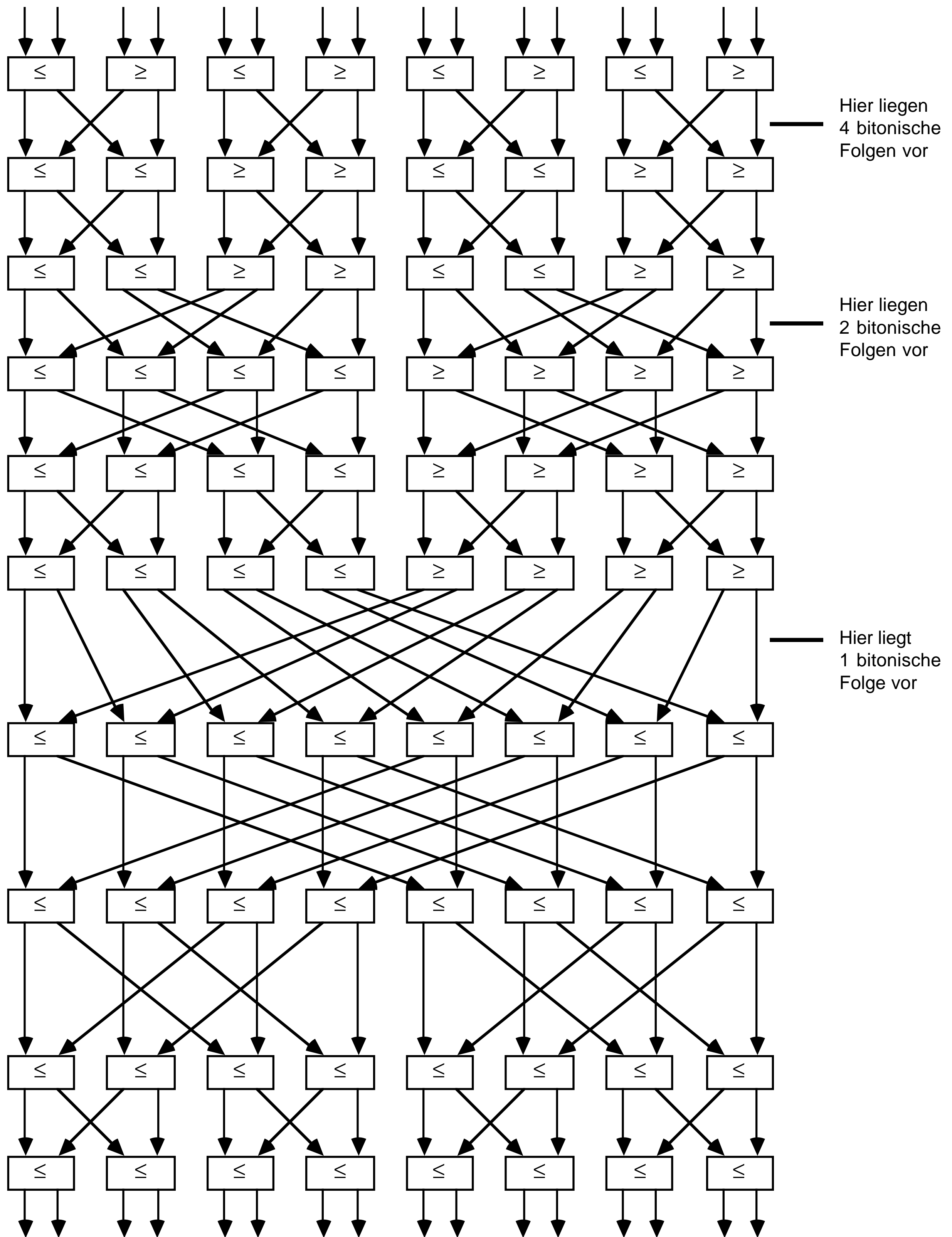

Abb. 7: Prozessornetzwerk zum Sortieren von 16 Zahlen

Zur Laufzeit des Verfahrens: Der letzte Schritt des Algorithmus, das Mischen einer bitonischen Folge, benötigt wie oben gesagt $O(\log_2 n)$ Takte. Das Herstellen einer bitonischen Folge erfordert folgende Takte:

- einen Takt, um im ersten Schritt $^n/_4$ bitonische Folgen der Länge 4 zu erzeugen,
- dann zwei Takte, um im zweiten Schritt $^n/_8$ bitonische Folgen der Länge 8 zu erzeugen,
- dann drei Takte, um im dritten Schritt $^n/_{16}$ bitonische Folgen der Länge 16 zu erzeugen,

...

- dann k Takte, um im k-ten Schritt $^n/_{2^{k+1}}$ bitonische Folgen der Länge 2^{k+1} zu erzeugen,

usw.

Addiert man die Takte auf, so erhält man für die Anzahl der Takte T des gesamten Netzwerkes in Abhängigkeit von n folgende Formel:

$$T(n)=1+2+3+...+(\log_2 n-1)+\log_2 n=$$
$$^1/_2(\log_2 n)(\log_2 n+1)\approx {}^1/_2(\log_2 n)^2=O((\log_2 n)^2).$$

Die Zahl der benötigten Prozessoren ergibt sich zu $O(n\cdot(\log_2 n)^2)$. Fazit: Zur Sortierung von n Zahlen benötigt der vorgestellte parallele Algorithmus $O((\log_2 n)^2)$ Takte. Speed-up bzw. Effizienz des Batcher-Algorithmus errechnen sich zu $O(n/\log_2 n)$ bzw. $O(1/\log_2 n)$. Der Algorithmus ist also nicht optimal.

Es war lange Zeit offen, ob diese Zeitschranke $O((\log_2 n)^2)$ verbessert werden kann. Erst 1983 haben die drei Wissenschaftler M. Ajtai, J. Komlós, E. Szemerédi aus Ungarn und den USA das Problem gelöst und einen parallelen Algorithmus entwickelt, der in $O(\log_2 n)$ Takten sortiert. Dieses Ergebnis ist jedoch bisher nur von theoretischem Interesse, da sich hinter der "groß-O"-Notation eine gewaltige Konstante (größer als 6000) verbirgt, die das Verfahren für alle praktischen Anwendungsfälle ineffizienter macht als den Batcher-Algorithmus.

Im Bereich der parallelen Algorithmen wird in den letzten Jahren eine breite Forschung betrieben. Dabei interessiert man sich hauptsächlich für Fragestellungen im Bereich der Rechnerarchitekturen (Registermaschine), der Programmiersprachen (z.B. Occam) und der Komplexitätstheorie (s.u.).

Kern innerhalb der Komplexitätstheorie ist die Frage: Wie schnell kann man Probleme, für die es bereits effiziente sequentielle Algorithmen gibt, im parallelen Fall bestenfalls lösen? Als Konzept, welches diese Zusammenhänge akzeptabel beschreibt, hat sich mittlerweile die Klasse *NC* (Abk. für "Nick's class" zu Ehren des amerikanischen Wissenschaftlers Nick Pippenger, auf den dieser Ansatz zurückgeht) erwiesen. NC ist die Klasse aller Probleme, die als effizient parallelisierbar gelten, vergleichbar zur

Klasse P aller Probleme, für die es effiziente (d.h. polynomiell zeitbeschränkte) sequentielle Algorithmen gibt (NP). Für NC-Probleme gibt es parallele Algorithmen, die in poly-logarithmischer Zeit (also polynomiell in $\log_2 n$) auf Architekturen mit polynomiell vielen Prozessoren abgearbeitet werden können. Die Probleme in obigen Beispielen sind offenbar in NC, denn kommt jeweils mit polynomiell vielen Prozessoren und konstanter Zeit (in den ersten beiden Beispielen) bzw. $(\log_2 n)^2$ Zeit (im dritten Beispiel) aus. Offenbar gilt $NC \subseteq P$, denn jeden parallelen NC-Algorithmus kann man in Polynomialzeit mit einem Prozessor simulieren. Bis heute weiß man nicht, ob $NC \neq P$ ist, ob also alle effizienten sequentiellen Algorithmen auch effizient parallelisierbar sind. Man vermutet, daß dies nicht der Fall ist. Ein Kandidat, für den es möglicherweise keinen effizienten parallelen Algorithmus gibt, ist das Problem, den größten gemeinsamen Teiler zu finden.

Literatur

K. E. Batcher: Sorting networks and their applications, Proceedings of the AFIPS Spring Joint Computing Conmference, volume 32, 307-314 (1968)

M. Ajtai, J. Komlós, E. Szemerédi: Sorting in clogn parallel steps, Combinatorica 3 (1983) 1-19