Information Processing Letters 33 (1989/90) 157–162
North-Holland

30 November 1989

CONSTRUCTING SORTING NETWORKS FROM k-SORTERS

Bruce PARKER

Department of Computer and Information Science, New Jersey Institute of Technology, Newark, NJ 07102, USA

Ian PARBERRY

Department of Computer Science, The Pennsylvania State University, University Park, PA 16802, USA

Communicated by David Gries
Received 14 November 1988
Revised 15 March 1989

We study the problem of using sorters of k values to form large sorting and merging networks. For n an integral power of k, we show how to merge k sorted vectors of length n/k each using $4\log_k n - 3$ layers of k-sorters and $4\log_k n - 5$ layers of k-input binary mergers. As a result, we show how to sort n values using $2\log_k^2 n - \log_k n$ layers of k-sorters and $2\log_k^2 n - 3\log_k n + 1$ layers of k-input binary mergers.

Keywords: Parallel processing, analysis of algorithms

1. Introduction

It is reasonable to assume that sorting chips will be available in the near future. It would be interesting to know how to wire these chips together to form a fast sorting network. In particular, if each chip has k inputs and k outputs, where constant $k \geq 2$, we show how to sort using at most $4\log_k^2 n$ levels of k-sorters. We do so by means of a modified ColumnSort [8].

This question was posed in a different setting by Knuth [7, Problem 5.3.4.44, page 243]:

> Study the properties of sorting networks which are made from k-sorter modules instead of 2-sorters. ... Are there efficient ways to sort k^2 elements with k-sorter modules, for all k?

Our technique results in a sorting network for k^2 elements using 6 layers of k-sorters plus 3 layers of k-input binary mergers.

In various modifications, this problem has been studied before, most intensively for the case $k = 2$ [1,2]. Green [6] examined a 4-way merge using 2-sorters, which Drysdale and Young [4] and Van Voorhis [11] generalized to a k-way merge, but still using 2-sorters. Leighton [8] gave a generalization of Batcher's Odd-Even MergeSort [2] by showing how to sort an $r \times c$ matrix. Bilardi and Preparata [3] used a tree of mergers of various sizes to sort using cube-connected cycles. Our method differs from the later two in that we sort using no primitive object with more than k inputs or outputs. Finally, Tseng and Lee [9] presented a generalization of Batcher's Merge Sort, but used $O(k)$ layers of k-sorters to finish the sort after diagonalization, compared to our use of only 4 layers (plus mergers).

We will approach the problem by first showing how to construct $(k, n/k)$-mergers from k-sorters, where an (x, y)-merger is a sorting network which takes as input x sorted sequences of y values each and produces as output a single sorted sequence of xy values. As a special case, we refer to a $(2, k/2)$-merger as a *k-input binary merger*. We start by presenting a modified version of Leighton's ColumnSort [8] in Section 2. As corollaries,

we obtain a slightly improved processor saving theorem (Section 3) and a construction of a $(k, n/k)$-merger using $4 \log_k n - 3$ layers of k-sorters and $4 \log_k n - 5$ layers of k-input binary mergers (Section 4). Finally, in Section 4, we show how to sort n values using $2 \log_k^2 n - \log_k n$ layers of k-sorters, and $2 \log_k^2 n - 3 \log_k n + 1$ layers of k-input binary mergers.

2. Modified ColumnSort

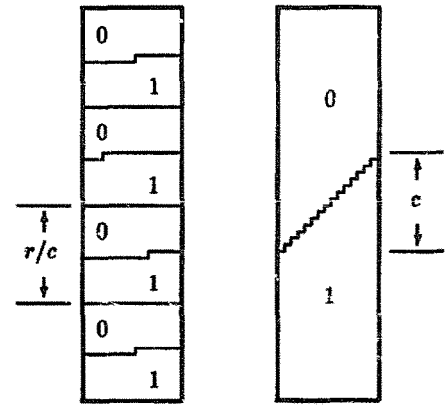
We want to sort n values arranged in a matrix with r rows and c columns. The columns are numbered 0 through $c-1$ and the rows 0 through $r-1$. We require that $n = rc$, $c \mid r$, and $r \geq c$.

Before we present the algorithm, we should be precise about the notion of diagonalizing a matrix. Here is the intuition: we cut diagonals through the matrix, just as Leighton describes for an alternative version of ColumnSort. However, instead of laying out the values in column order, we lay them out in row order.

More formally, we move each value at position (i, j) to position $(i+j, j)$. The matrix expands by $c-1$ rows downward. We fill the space vacated in the upper right-hand corner with $\frac{1}{2}c(c-1)$ "small" values (negative infinity for sorting integers, zeroes for sorting zeroes and ones) and fill the lower left-hand corner with $\frac{1}{2}c(c+1)$ "large" values (positive infinity for sorting integers, ones for sorting zeroes and ones). These dummy values will later be pushed to the extreme rows of the matrix and thus be discarded. The extra row of ones is just to keep the total number of rows a multiple of c.

Algorithm 2.1. *Modified ColumnSort*

1. Partition the matrix into r/c by c rectangles and sort each into row major order.
2. Sort the columns downwards.
3. Diagonalize the matrix as described above.
4. Partition the matrix into $c \times c$ squares and sort each into row major order.
5. Merge each half square with its mate in the adjacent square, that is, merge the last $\frac{1}{2}(c^2 - 3c + 2)$ values of square i with the first $\frac{1}{2}(c^2 - 3c + 2)$ values of square $i+1$, $0 \leq i < r/c$.



After Step 1 After Step 2

Fig. 1. Matrix after first two steps.

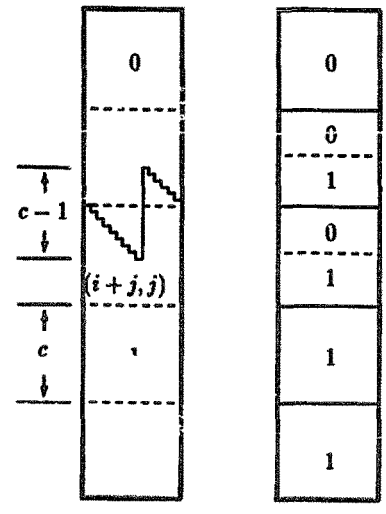
See Figs. 1 and 2 for an example.

Remark. We note the following in comparison with Leighton's ColumnSort:

(1) The resulting matrix is in row-major, instead of column-major order as in Leighton's algorithm.

(2) Our Step 1 corresponds to ColumnSort's Steps 1 and 2.

(3) By Step 3, we have reduced the problem of sorting a rectangular array to the problem of sorting the columns (twice) and sorting squares



After Step 3 After Step 4

Fig. 2. Matrix after second two steps

(once). For r not much bigger than c, we have not improved the sortedness very much, as we rely in Step 4 on an ability to sort a region almost as large as the original problem. However, for r much larger than c, a smaller sorter will suffice for Step 4 and thus the time for Steps 1 and 2 dominate the running time.

We now prove the correctness of the algorithm. Our proof uses the 0/1 Principle (a corollary of a theorem due to Bouricius [7, p. 224]) instead of Leighton's approach of bounding the displacement of any given value from its final position. We will use the terminology "clean" and "dirty" to describe regions of zeroes and ones which are homogeneous (all ones or all zeroes) and heterogeneous (a mixture of zeroes and ones), respectively. The idea is to show that the matrix can be partitioned so that there exists a boundary between the zeroes and ones. More formally we need to show that after applying some alleged sorting method, there exists an index p, $0 \leqslant p \leqslant n$, such that all positions $\text{rank}^{-1}(i)$ for $i < p$ are zeroes and all positions $\text{rank}^{-1}(i)$ for $i \geqslant p$ are ones. For a rank, we will be satisfied with any bijection rank: $\{0 \ldots r-1\} \times \{0 \ldots c-1\} \to \{0 \ldots n-1\}$ which informs us that position (i, j) holds a value which is the (rank(i, j))th smallest.

Lemma 2.1. *After Step 2, the zeroes and ones are separated by a monotone non-decreasing boundary, that is, for all i, j, $0 \leqslant i < j < r$, there are at least as many zeroes in the ith row as in the jth.*

Proof. The rows remain sorted after sorting the columns [5]. Thus, there are at least as many ones in row $i+1$ as row i, $0 \leqslant i < r-1$. Suppose otherwise. Consider a row i' which has more ones than row $i'+1$. Since the rows are sorted, there must exist a column j such that (i', j) has a one and $(i'+1, j)$ has a zero, meaning that the columns are not sorted as was assumed. □

Lemma 2.2. *After Step 2, the dirty region has height at most c.*

Proof. After Step 1, each $r/c \times c$ rectangle has at most one dirty row. After sorting the columns is Step 2, these c dirty rows will be adjacent. □

Remark. For $c = 2$, as in Batcher's Odd-Even Merge, the dirty region has height at most 1 after diagonalization, that is, there is at most one dirty row. Thus 2-sorters applied to each row, in place of Steps 4 and 5, will suffice to finish the sort.

Lemma 2.3. *After Step 3, there are no two adjacent squares such that the lower square has more than $\frac{1}{2}(c^2 - 3c + 2)$ zeroes and the upper square has more than $\frac{1}{2}(c^2 - 3c + 2)$ ones.*

Proof. The only interesting case occurs when the dirty region spans two adjacent squares. Suppose there is at least one 1 in the upper square. Consider the situation before Step 3. A row after Step 3 was a back diagonal before diagonalization. Thus a square after Step 3 was a rhombus before diagonalization. Let the highest 1 in the upper rhombus be at position (i, j), choosing the leftmost 1 in case of a tie. Let $i_0 = i + c$, $i' = \lceil (i+1)/c \rceil c$, and $i'' = i' - j$ (cf. Fig. 3). By Lemma 2.1, the region to the right and below the position (i, j) must consist of all 1's. This region includes at least $c(c-j) - \frac{1}{2}(i''-i)(i''-i+1) - \frac{1}{2}(i_0 - i')(i_0 - i' + 1)$ positions inside the lower rhombus, all of which must therefore be 1's. The region of the i_0th row and below must also be 1's by Lemma 2.2. This region includes at least $\frac{1}{2}(i' + c - i_0)(i'$

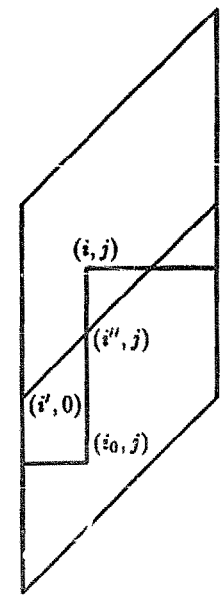


Fig. 3. Situation before diagonalization.

$+ c - i_0 + 1$) positions. We observe that the number of 0's in the bottom rhombus is maximized when $i + j = i' - 1$, that is, $i = i' - j - 1$. Thus $i'' - i = 1$ and $i_0 - i' = c - j - 1$ and the number of 1's in the lower rhombus is at least $\frac{1}{2}(j+1)(j+2) + c(c-j) - 1 - \frac{1}{2}(c-j-1)(c-j) = \frac{1}{2}c^2 + \frac{1}{2}c + j$. Since $j \leq c - 1$, we can conclude that there are at least $\frac{1}{2}(c^2 + 3c - 2)$ 1's in the bottom rhombus and at most $\frac{1}{2}c^2 - (c^2 + 3c - 2) = \frac{1}{2}(c^2 - 3c + 2)$ 0's. Symmetrically, we observe that if there is a zero in the lower square, then there are at most $\frac{1}{2}(c^2 - 3c + 2)$ ones in the upper square. □

Theorem 2.4. *Our sorting algorithm finishes with the matrix in row order.*

Proof. By Lemma 2.2, the algorithm reduces the dirty region to a $c \times c$ area after Step 3. Thus the dirty region either fits entirely within one of the $c \times c$ squares of Step 4 or it straddles two adjacent squares. In the former case, Step 4 completes the sort and Step 5 does nothing. In the latter case, we use Lemma 2.3 to assert that Step 5 will patch up the remaining dirty region. □

Remark. Unlike Leighton's ColumnSort but like Batcher's Odd-Even MergeSort, we diagonalize the matrix. Doing so after Step 2 reduces the height of the dirty region to $c - 1$ and requires only that we merge half squares with their neighbors in the adjacent squares in Step 5. Without diagonalization, we would need to merge each whole square with each of its neighbors.

3. A processor-saving theorem

To be complete, we ought to re-establish Leighton's processor saving theorem. This result relies on a family of sorting circuits and does not show how to construct them. However, within this context, we show that when our modified ColumnSort (Algorithm 2.1) is used in place of Leighton's, we effectively halve the time of the resulting sorting circuits.

Theorem 3.1. *Given a monotone function f such that $f(N) = o(N^{1/3})$ for all N and a family of $f(N)$-level circuits for sorting N numbers, one can construct a family of bounded-degree $O(N)$-node networks that can sort N numbers in $4f(N) + o(f(N))$ word steps.*

Proof. We set $r = N/f(N)$ and $c = f(N)$ and use Algorithm 2.1. We implement Steps 1 and 2 by pipelining an $N/f(N)$-input $f(N/f(N))(= O(f(N)))$ time sorting circuit. The number of vectors to be pushed through the sorter is $f(N)$. Since the total running time of a pipelined circuit is the sum of the circuit depth plus the number of vectors minus one, the running time is at most $2f(N) = o(N^{1/3})$. The number of nodes is bounded from above by the circuit width $N/f(N)$ times the circuit depth at most $2(f(N))$ or $O(N)$.

Step 3 is just a matter of rewiring and so requires zero time.

We implement Step 4 by pipelining $N/(f(N)^2 \cdot f(f(N)^2))$ copies of a $f(N)^2$ input $f(f(N)^2)$ depth sorting circuit. We use these circuits in parallel by dividing the N values into $f(f(N)^2)$ vectors. Each vector is further partitioned into $N/(f(N)^2 f(f(N)^2))$ subvectors of $f(N)^2$ values each. Each subvector is associated with a separate sorter. At each stage of the pipeline we input a vector by inputting each subgroup into the first stage of its sorter. Since there are $f(f(N)^2)$ vectors and the depth of each sorter is $f(f(N)^2)$, the running time for this step is $2f(f(N)^2) = o(N^{2/9})$. The number of nodes is then at most the number of circuits $N/(f(N)^2 f(f(N)^2))$ times the circuit width $f(N)^2$ times its depth $f(f(N)^2)$ or $O(N)$.

We complete the construction by using an $f(N)^2$ input Odd-Even Merge circuit of $2f(N)^2 \cdot \log f(N)$ $(= o(N^{1/9} \log N))$ nodes and $2 \log f(N) (= O(\log N))$ depth to implement Step 5.

Thus the total number of nodes required is $O(N)$ and the running time is $O(f(N))$. □

In particular, for $f(N) = O(\log^c N)$ for constant c, Steps 4 and 5 each require $O(\log \log N)$ time. The depth of any such ColumnSort circuit is dominated by the circuits used in Steps 1 and 2, effectively having the running time of the network, as compared with the $8f(N) + o(f(N))$ word steps of Leighton's version.

4. How to construct n-sorters from k-sorters

In this section, we show how to construct large mergers and sorters as a direct consequence of our modified ColumnSort.

We begin by constructing an $(k, n/k)$-merger. We assume that n is an integral power of k and that k is a perfect square. We can save mergers by using a $(\sqrt{k}, n/\sqrt{k})$-merger. We do so by directly implementing our modified ColumnSort.

Construction of a $(\sqrt{k}, n/\sqrt{k})$-merger

We use Algorithm 2.1. Let $r = n/\sqrt{k}$ and $c = \sqrt{k}$.

1. Lay out the $\sqrt{k}$ sequences of $n/\sqrt{k}$ sorted values in row major order.
2. Recursively merge each column using $(\sqrt{k}, n/k)$-mergers.
3. Diagonalize the matrix.
4. Use one layer of k-sorters to sort the $\sqrt{k} \times \sqrt{k}$ squares into row major order.
5. Use one layer of $(2, \frac{1}{2}k)$-mergers to merge each half square with its mate in the adjacent square.

The recurrence for the number of layers of k-sorters is $d_s(n) = d_s(n/\sqrt{k}) + 1$ for $n > k$, and $d_s(k) = 1$, while that for the $(2, \frac{1}{2}k)$-mergers is $d_m(n) = d_m(n/\sqrt{k}) + 1$ for $n > k$, and $d_m(k) = 0$. Thus we use $2 \log_k n - 1$ layers of k-sorters and $2 \log_k n - 2$ layers of $(2, \frac{1}{2}k)$-mergers to construct our $(\sqrt{k}, n/\sqrt{k})$-merger.

We form a $(k, n/k)$-merger as follows: partition the k vectors into $\sqrt{k}$ groups of $\sqrt{k}$, merge each group, and then merge the resulting $\sqrt{k}$ vectors. This can be accomplished by a $(\sqrt{k}, n/k)$-merger and a $(\sqrt{k}, n/\sqrt{k})$-merger. A $(\sqrt{k}, n/k)$-merger can be constructed in a recursive manner similar to that of our $(\sqrt{k}, n/\sqrt{k})$-merger, but in $2 \log_n n - 2$ layers of k-sorters and $2 \log_k n - 3$ layers of $(2, \frac{1}{2}k)$-mergers. Thus a $(k, n/k)$-merger can be done in $4 \log_k n - 3$ layers of k-sorters and $4 \log_k n - 5$ layers of $(2, \frac{1}{2}k)$-mergers. If we restrict ourselves to use only k-sorters, we use $8 \log_k n - 8$ layers of k-sorters to form our $(k/n/k)$-merger.

We now use our $(k, n/k)$-merger to form a recursive sorting network. We partition the inputs into k groups of n/k values, recursively sort each group, and then merge the resulting k vectors using a $(k, n/k)$-merger. This gives us the recurrence $d_s(n) = d_s(n/k) + 4 \log_k n - 3$ when $n > k$ and $d_s(k) = 1$ for the number of layers of k-sorters used and the recurrence $d_m(n) = d_m(n/\sqrt{k}) + 4 \log_k n - 5$ $(n > k)$ and $d_m(k) = 0$ for the number of layers of $(2, \frac{1}{2}k)$-mergers used. Thus we use $2 \log_k^2 n - \log_k n$ layers of k-sorters and $2 \log_k^2 n - 3 \log_k n + 1$ layers of $(2, \frac{1}{2}k)$-mergers. If we restrict ourselves to use only k-sorters, this method uses $4 \log_k^2 n - 4 \log_k - 4 \log_k n + 1$ layers of k-sorters.

5. Summary

We have exhibited a generalization of Batcher's Odd-Even Merge to a k-way merge and from this a construction of a k-way merger and an O(1) depth sorting network for n polynomial in k.

Other improvements are possible in the sorting network construction. For instance, we could delete Step 3 from the modified ColumnSort. Without diagonalization, the rows and columns of the $c \times c$ squares remain sorted. Thus sorting of the $c \times c$ squares does not require the full capability of a sorting network. Drysdale and Young [4] and Van Voorhis [11] both present a $\frac{1}{4} \log^2 n$ depth network which finishes the sorting of a $\sqrt{n} \times \sqrt{n}$ square of values with the rows and columns sorted. Since this is more than twice as fast as Batcher's sorting networks, we can use these networks for Step 4.

We have not yet answered Knuth's implicit question. Here we assume that $k^2 = n$ and ask whether a k^2 sorter can be constructed from three levels of k-sorters (plus O(1) levels of k-input binary mergers). Such a construction could be used recursively to build a $o(\log^2 n)$ depth sorting network with a small constant multiple (depending on how many mergers the construction requires). Suppose that $r = c = k$ and that we preprocess the square by using 2 k-sorters to sort the rows and columns. Given this situation, Drysdale, Young, and Van Voorhis have noted that it is possible to recursively sort 4 $\frac{1}{2}k \times \frac{1}{2}k$ subsquares in one parallel recursive call and then merge these

4 squares together in $O(\log k)$ depth. We have found several ways of partitioning the square into k $\sqrt{k} \times \sqrt{k}$ subsquares but unfortunately with 4 recursive calls, thus using $\Omega(\log^2 n)$ time. Van Voorhis [10] has shown that $\Omega(\log^2 n)$ levels of 2-sorters are required for any sorting network based on a k-way merge where κ is a constant, but the proof yields only a lower bound of $\Omega(\log n)$ for a $\sqrt{n}$-way merge.

Acknowledgment

The first author would like to thank Torben Hagerup for a careful reading of an early draft of this paper.

References

[1] M. Ajtai, J. Komlós, and E. Szemerédi, An $O(n \log n)$ sorting network, in: *Proc. 15th Annual ACM Symposium on Theory of Computing* (1983) 1–9.

[2] K.E. Batcher, Sorting networks and their applications, in: *Proc. AFIPS 1968 SJCC* (1968) 307–314.

[3] G. Bilardi and F.P. Preparata, A minimum area VLSI network for $O(\log n)$ time sorting, in: *Proc. 16th Annual ACM Symposium on the Theory of Computing* (1984) 64–70.

[4] R.L. Drysdale III and F.H. Young, Improved divide/sort/merge sorting network, *SIAM J. Comput.* 4 (3) (1975) 264–270.

[5] D. Gale and R.M. Karp, A phenomenon in the theory of sorting, in: *Proc. 11th IEEE Annual Symposium on Switching and Automata Theory* (1970) 51–59.

[6] M.G. Green, Some improvements in nonadaptive sorting algorithms, in: *Proc. 6th Annual Princeton Conference on Information Sciences and Systems* (1972) 387–391.

[7] D.E. Knuth, *The Art of Computer Programming, Vol. 3* (Addison-Wesley, Reading, MA, 1973).

[8] F.T. Leighton, Tight bounds on the complexity of parallel sorting, *IEEE Trans. Comput.* 34 (4) (1985) 344–354.

[9] S.S. Tseng and R.C.T. Lee, A parallel sorting scheme whose basic operation sorts n elements, *Internat. J. Comput. Inform. Sci.* 14 (6) (1985) 455–467.

[10] D.C. Van Voorhis, A lower bound for sorting networks that use the divide-sort-merge strategy, Technical Report 17, Department of Computer Science, Stanford University, Stanford, CA, 1971.

[11] D.C. Van Voorhis, An economical construction for sorting networks, in: *Proc. AFIPS NCC* 43 (1974) 921–927.