An Optimal Parallel Algorithm for Merging using Multiselection[†]

Narsingh Deo Amit Jain Muralidhar Medidi

Department of Computer Science, University of Central Florida, Orlando, FL 32816

Keywords: selection, median, multiselection, merging, parallel algorithms, EREW PRAM.

1 Introduction

We consider the problem of merging two sorted arrays A and B on an exclusive read, exclusive write parallel random access machine (EREW PRAM, see [8] for a definition). Our approach consists of identifying elements in A and B which would have appropriate rank in the merged array. These elements partition the arrays A and B into equal-size subproblems which then can be assigned to each processor for sequential merging. Here, we present a novel parallel algorithm for selecting the required elements, which leads to a simple and optimal algorithm for merging in parallel. Thus, our technique differs from those of other optimal parallel algorithms for merging where the subarrays are defined by elements at fixed positions in A and B.

Formally, the problem of *selection* can be stated as follows. Given two ordered multisets A and B of sizes m and n, where $m \leq n$, the problem is to select the jth smallest element in A and B combined. The problem can be solved sequentially in $O(\log(min\{j, m\}))$[‡] time without explicitly merging A and B [5, 6]. *Multiselection*, a generalization of selection, is the problem where given a sequence of r integers $1 \leq K_1 < K_2 < \ldots < K_r \leq (m + n)$, all the K_ith, $1 \leq i \leq r$, smallest elements in A and B combined are to be found.

[†] Supported in part by NSF Grant CDA-9115281.

[‡] For clarity in presentation, we use $\log x$ to mean $\max\{1, \log_2 x\}$.

Parallel merging algorithms proposed in [1] and [5] employ either a sequential median or a sequential selection algorithm. Even though these parallel algorithms are cost-optimal, their time-complexity is $O(\log^2(m+n))$ on an EREW PRAM. Parallel algorithms for merging described in [2, 3, 7, 10, 11] use different techniques essentially to overcome the difficulty of multiselection.

Without loss of generality, we assume that A and B are disjoint and contain no repeated elements. First, we present a new algorithm for the selection problem and then use it to develop a parallel algorithm for multiselection. The algorithm uses r processors, of an EREW PRAM, to perform r selections in $O(\log m + \log r)$ time. We further show that the number of comparisons in our merging algorithm matches that of Hagerup and Rüb's algorithm [7] and is within lower-order terms of the minimum possible, even by a sequential merging algorithm. Moreover, our merging algorithm uses fewer comparisons when the two given arrays differ in size significantly.

2 Selection in Two Sorted Arrays

The median of $2r$ elements is defined to be the rth smallest element, while that of $2r+1$ elements is defined to be the $(r+1)$th element. Finding the jth smallest element can be reduced to selecting the median of the appropriate subarrays of A and B as follows: When $1 \le j \le m$ and the arrays are in nondecreasing order, the required element can only lie in the subarrays $A[1..j]$ and $B[1..j]$. Thus, the median of the $2j$ elements in these subarrays is the jth smallest element. This reduction is depicted as Case III in Figure 1. On the other hand, when $m < j \le \lceil (m+n)/2 \rceil$, the jth selection can be reduced to finding the median of the subarrays $A[1..m]$ and $B[(j-m)..j]$, which is shown as Case I in Figure 1. When $j > \lceil (m+n)/2 \rceil$, we can view the problem as that of finding the kth largest element, where $k = m+n-j+1$. This gives rise to Cases II and IV which are symmetric to Cases I and III, respectively, in Figure 1. From now on, these subarrays will be referred to as *windows*.

The median can be found by comparing the individual median elements of the current

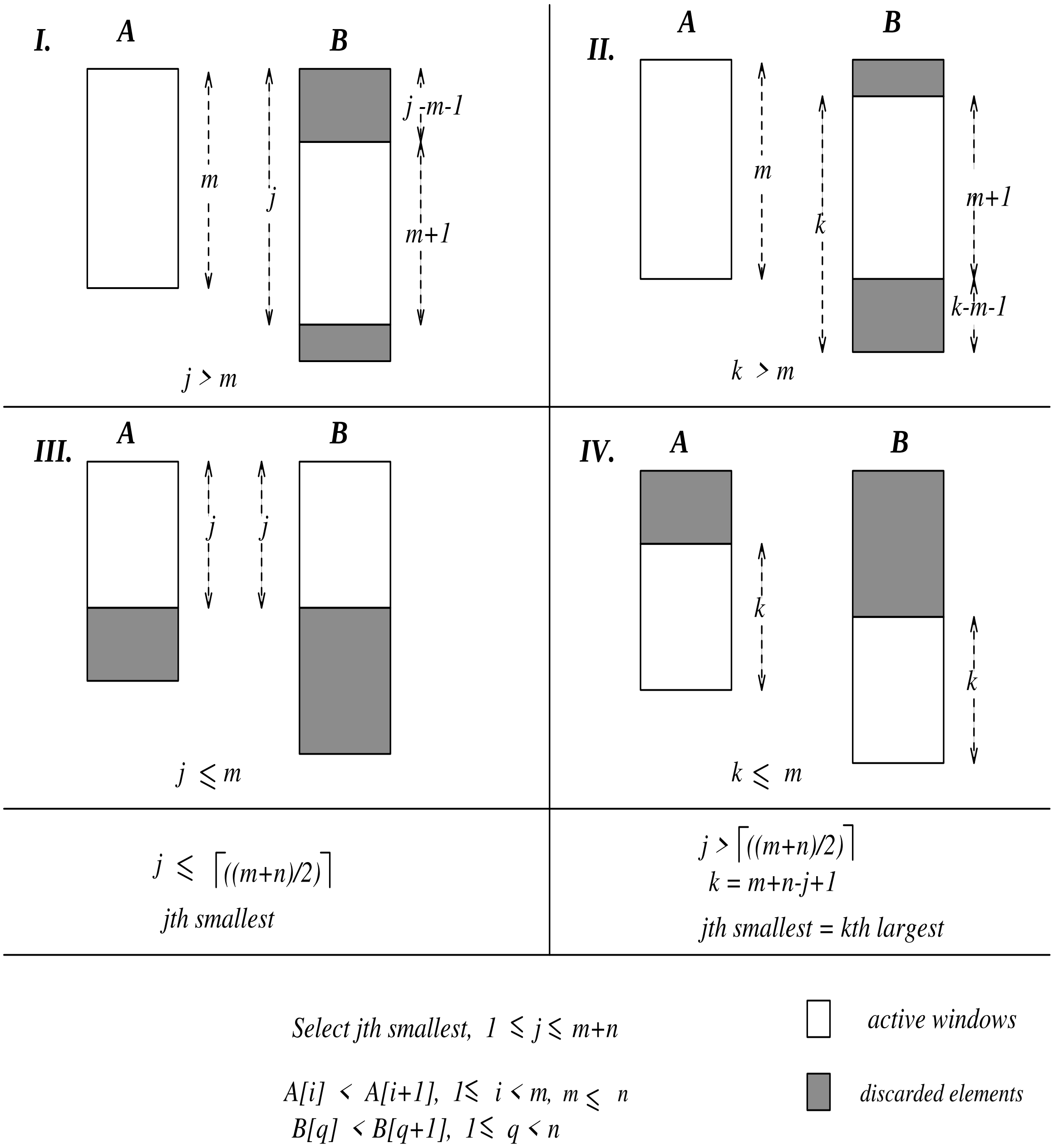

Figure 1: Reduction of selection to median finding

windows and suitably truncating the windows to half, until the window in A has no more than one element. The middle elements of the windows will be referred to as *probes*. A formal description of this median-finding algorithm follows.

```
procedure select_median(A[lowA, ..., highA], B[lowB, ..., highB])
{       A[p] < A[p+1], lowA ≤ p ≤ highA, A[m+1] = ∞
        B[q] < B[q+1], lowB ≤ q ≤ highB, B[n+1] = ∞
        highA − lowA ≤ highB − lowB ≤ highA − lowA + 1
        [lowA, highA], [lowB, highB]: current windows in A and B
        probeA, probeB : next position to be examined in A and B }
1.      while (highA > lowA)
2.              probeA ← ⌊ (lowA+ highA)/2 ⌋; sizeA ← (highA-lowA+1)
3.              probeB ← ⌊ (lowB + highB)/2 ⌋; sizeB ← (highB-lowB+1)
4.              case    (A[probeA] < B[probeB]) :
5.                              lowA ← probeA + 1; highB ← probeB
6.                              if (sizeA = sizeB) and (sizeA is odd)   lowA ← probeA
7.                      (A[probeA] > B[probeB]) :
8.                              highA ← probeA; lowB ← probeB
9.                              if (sizeA = sizeB) and (sizeA is even)   lowB ← probeB+1
10.             endcase
11.     endwhile
12.     merge the remaining (at most 3) elements from A + B and return their median
endprocedure
```

When the procedure *select_median* is invoked, there are two possibilities: (i) the size of the window in A is one less than that of the window in B (ii) or the sizes of the windows are equal. Furthermore, considering whether the size of the window in A is odd or even, the reader can verify (examining Steps 4 through 9) that an equal number of elements are being discarded from above and below the median. Hence, the scope of the search is narrowed to at most three elements (1 in A and at most 2 in B) in the two arrays; the median can then be determined easily in Step 12, which will be denoted as the postprocessing phase. The total time required for selecting the jth smallest element is $O(\log(\min\{j, m\}))$. With this approach, r different selections, $\{K_1, \ldots, K_r\}$, in A and B can be performed in $O(r \log m)$ time. Note that the information-theoretic lower bound for the problem of multiselection is $\binom{m+r}{r}$ which turns out to be $O(r \log m/r)$ when $r \leq m$ and $O(m \log r/m)$ when $r > m$.

A parallel algorithm for r different selections based on the above sequential algorithm is presented next.

3 Parallel Multiselection

Let the selection positions be $(K_1, K_2, \ldots, K_r)$, where $1 \leq K_1 < K_2 < \ldots < K_r \leq (m+n)$. Our parallel algorithm employs r processors with the ith processor assigned to finding the K_ith element, $1 \leq i \leq r$. The distinctness and the ordered nature of the K_is are not significant restrictions on the general problem. If there are duplicate K_is or if the selection positions are unsorted, both can be remedied in $O(\log r)$ time using r processors [4]. (On a CREW PRAM the problem admits a trivial solution, as each processor can carry out the selection independently. On an EREW PRAM, however, the problem becomes interesting because the read conflicts have to be avoided.)

In the following, we will outline how multiselections can be viewed as multiple searches in a search tree. Hence, we can exploit the well-known technique of *chaining* introduced by Paul, Vishkin and Wagener [9]. For details on EREW PRAM implementation of chaining, the reader is referred to [9] or [8, Exercise 2.28].

Let us first consider only those K_is that fall in the range $[m+1 \ldots \lceil (m+n)/2 \rceil]$, that is, those for which Case I, in Figure 1, holds. All of these selections initially share the same probe in array A. Let $m < K_l < K_{l+1} < \ldots < K_h \leq \lceil (m+n)/2 \rceil$ be a sequence of K_is that share the same probe in A. Following the terminology of Paul, Vishkin and Wagener [9], we refer to such a sequence of selections as a *chain*. Note that these selections will have different probes in array B. Let the common probe in array A be x for this chain, and the corresponding probes in array B be $y_l < y_{l+1} < \ldots < y_h$. The processor associated with K_lth selection will be active for the chain. This processor compares x with y_l and y_h, and based on these comparisons the following actions take place:

- $x < y_l$: The chain stays intact.
- $x > y_l$

- $\star$ $x < y_h$: The chain is split into two subchains.
- $\star$ $x > y_h$: The chain stays intact.

Note that at most two comparisons are required to determine if the chain stays intact or has to be split. When the chain stays intact, the window in array A remains common for the whole chain. Processor P_l computes the size of the new common window in array A. The new windows in the array B can be different for the selections in the chain, but they all shrink by the same amount, and hence the size of the new window in B and the offset from the initial window in B are the same for all the selections. The two comparisons made by the active processor determine the windows for all the selections in the chain (when the chain stays intact). The chain becomes inactive when it is within 3 elements to compute the required median for all the selections in the chain. The chain does not participate in the algorithm any more, except for the postprocessing phase.

When a chain splits, processor P_l remains in charge of the chain $K_l, \ldots, K_{\lceil (l+h)/2 \rceil - 1}$ and activates processor $p_{\lceil (l+h)/2 \rceil}$ to handle the chain $K_{\lceil (l+h)/2 \rceil}, \ldots, K_h$. It also passes the position and value of the current probe in array A, the offsets for the array B, and the parameter h. During the same stage, both these processors again check to find whether their respective chains remain intact. If the chains remain intact they move on to a new probe position. Thus, only those chains that do not remain intact stay at their current probe positions to be processed in the next stage. It can be shown that at most two chains remain at a probe position after any stage. Moreover, there can be at most two new chains arriving at a probe position from the previous stages. The argument is the same as the one used in the proof of Claim 1 in Paul, Vishkin and Wagener [9]. All of this processing within a stage can be performed in $O(1)$ time on an EREW PRAM, as at most four processors may have to read a probe. When a chain splits into two, their windows in A will overlap only at the probe that splits them. Any possible read conflicts at this common element can happen only during the postprocessing phase (which can be handled as described in the next paragraph). Hence, all of the processing can be performed without any read conflicts.

At each stage a chain is either split into two halves or its window size is halved. Hence after at most $O(\log m + \log r)$ stages each selection process must be within three elements of the required position. At this point, each processor has a window of size at most 1 in A and 2 in B. If the windows of different selections have any elements in common, the values can be broadcasted in $O(\log r)$ time, such that each processor can then carry out the required postprocessing in $O(1)$ time. However, we may need to sort the indices of the elements in the final windows in order to schedule the processors for broadcasting. But this requires only integer sorting as we have r integers in the range $1 \ldots n$ which can surely be done in $O(\log r)$ time [4]. Thus, the total amount of data copied is only $O(r)$.

Of the remaining selections, those K_is falling in Case II can be handled in exactly the same way as the ones in Case I. The chaining concept can be used only if $O(1)$ comparisons can determine the processing for the whole chain. In Cases III and IV, different selections have windows of different sizes in both the arrays. Hence, chaining cannot be directly used as in Cases I and II. However, we can reduce Case III (IV) to I (II). To accomplish this reduction, imagine array B to be padded with m elements of value $-\infty$ in locations $-m+1$ to 0 and with m elements of value ∞ in locations $n+1$ to $n+m$. Let this array be denoted as C (which need not be explicitly constructed). Selecting the jth smallest element, $1 \leq j \leq m+n$, in $A[1..m]$ and $B[1..n]$ is equivalent to selecting the $(j+m)$th element, $m+1 \leq (j+m) \leq 2m+n$, in the arrays $A[1..m]$ and $C[1..2m+n]$. Thus, selections in Case III (IV) in the arrays A and B become selections in Case I (II) in the arrays A and C.

Any selection in the interval $[m \ldots n]$ dominates the time-complexity. In such a case, we note that all of the selections in different cases can be handled by one chain with the appropriate reductions. Hence we have the following result.

Theorem 3.1 *Given r selection positions $\{K_1, \ldots, K_r\}$, all of the selections can be made in $O(\log m + \log r)$ time using r processors on the EREW PRAM.*

4 Parallel Merging

Now, consider the problem of merging two sorted sequences A and B of length m and n, respectively. Hagerup and Rüb[7] have presented an optimal algorithm which runs in $O(\log(m+n))$ time using $((m+n)/\log(m+n))$ processors on an EREW PRAM. The algorithm recursively calls itself once and then uses Batcher's bitonic merging. Also in order to avoid read conflicts, parts of the sequences are copied by some processors.

Akl and Santoro [1], and Deo and Sarkar [5] have used selection as a building block in parallel merging algorithms. Even though these algorithms are cost-optimal, their time complexity is $O(\log^2(m+n))$ on the EREW PRAM. By solving the parallel multiselection problem, we obtain a simpler cost-optimal merging algorithm of time-complexity $O(\log(m+n))$ with $((m+n)/\log(m+n))$ processors on the EREW PRAM. The algorithm can be expressed as follows:

1. Find the $i\lfloor\log(m+n)\rfloor$, $i = 1, 2, \ldots, j-1$(where $j = \lceil (m+n)/\lfloor\log(m+n)\rfloor\rceil$), ranked element using multiselection. Let the output be two arrays $R_A[1..j]$ and $R_B[1..j]$, where $R_A[i] \neq 0$ implies that $A[R_A[i]]$ is the $(i\lfloor\log(m+n)\rfloor)$th element and $R_B[i] \neq 0$ implies that $B[R_B[i]]$ is the $(i\lfloor\log(m+n)\rfloor)$th element.

2. Let $R_A[0] = R_B[0] = 0$, $R_A[j] = m, R_B[j] = n$
for $i = 1, \ldots, j-1$ do
if $R_A[i] = 0$ then $R_A[i] = i * \lfloor\log(m+n)\rfloor - R_B[i]$
else $R_B[i] = i * \lfloor\log(m+n)\rfloor - R_A[i]$

3. for $i = 1, \ldots, j$ do
merge $(A[R_A[i-1]+1]..A[R_A[i]])$ with $(B[R_B[i-1]+1]..B[R_B[i]])$.

Steps 1 and 3 both take $O(\log(m+n))$ time using $(m+n)/\log(m+n)$ processors. Step 2 takes $O(1)$ time using $(m+n)/\log(m+n)$ processors. Thus the entire algorithm takes $O(\log(m+n))$ time using $(m+n)/\log(m+n)$ processors, which is optimal. The total amount of data copied in Step 1 is $O((m+n)/\log(m+n))$, since $r = (m+n)/\log(m+n)$, and compares favorably with $O(m+n)$ data copying required by Hagerup and Rüb's [7] merging algorithm. If fewer processors, say p, are available, the proposed parallel merging algorithm can be adapted to perform $p-1$ multiselections, which will require $O((m+n)/p + \log m + \log p))$-time.

Let us now count the number of comparisons required. Step 2 does not require any comparisons and Step 3 requires less than $m+n$ comparisons. The estimation of the number of comparisons in Step 1 is somewhat more involved. First we need to prove the following lemma.

Lemma 4.1 *Suppose we have a chain of size r, $r \geq 2$. The worst-case number of comparisons required to completely process the chain is greater if the chain splits at the current probe than if it stays intact.*

Proof: We can envisage the multiselection algorithm as a specialized search in a binary tree with height $O(\log m)$. Let $T(r, l)$ be the total number of comparisons needed, in the worst case, to process a chain of size r, which is at a probe corresponding to a node at height l in the search tree. We proceed by induction on the height of the node. The base case, when the chain is at a node of height 1, can be easily verified. Suppose the lemma holds for all nodes at height $\leq l-1$. Consider a chain of size r at a node of height l. If the chain stays intact and moves down to a node of height $l-1$ then

$$T(r, l) = T(r, l-1) + 2, \tag{1}$$

since at most two comparisons are required to process a chain at a node. If the chain splits, one chain of size $\lceil r/2 \rceil$ stays at height l (in the worst-case) and the other chain of size $\lfloor r/2 \rfloor$ moves down to height $l-1$. The worst-case number of comparisons is, then,

$$T(r, l) = T(\lceil r/2 \rceil, l) + T(\lfloor r/2 \rfloor, l-1) + 4. \tag{2}$$

By the hypothesis and Eq. (2)

$$T(r, l-1) \leq T(\lceil r/2 \rceil, l-1) + T(\lfloor r/2 \rfloor, l-2) + 4. \tag{3}$$

Thus, when the chain stays intact, we can combine Eq.s (1) and (3) to obtain

$$T(r, l) \leq T(\lceil r/2 \rceil, l-1) + T(\lfloor r/2 \rfloor, l-2) + 6.$$

Again, by hypothesis $T(\lfloor r/2 \rfloor, l-1) \geq T(\lfloor r/2 \rfloor, l-2) + 2$ and we require at least one comparison for a chain to move down a level. Hence $T(\lceil r/2 \rceil, l) \geq T(\lceil r/2 \rceil, l-1) + 1$ and the lemma holds. $\square$

To determine the number of comparisons required in the multiselection algorithm, we consider two cases. In the first case, when $r \leq m$, we have the following.

Lemma 4.2 *In the worst case, the total number of comparisons required by the parallel multiselection algorithm for r selections is $O(r(1 + \log(m/r)))$, if $r \leq m$.*

Proof: The size of the initial chain is r. Lemma 4.1 implies that the chain must split at every opportunity for the worst-case number of comparisons. Thus, at height i the maximum size of a chain is $r/2^i, 0 \leq i \leq \lfloor \log r \rfloor$. Recall that at most two chains can remain at any node after any stage. The maximum number of chains possible is r (with each containing only one element); which, in the worst case, could be spread over the first $\lceil \log r \rceil$ levels. From a node at height i, the maximum number of comparisons a search for an element can take is $(\lceil \log m \rceil - i)$ (for this chain at this node). Hence the number of comparisons after the chains have split is bounded by

$$\sum_{i=0}^{j} 2(2^i)(\lceil \log m \rceil - i), \quad j = \lfloor \log r \rfloor$$

which is $O(r \log m/r)$. We also need to count the number of comparisons required for the initial chain to split up into r chains, and fill up $\lceil \log r \rceil$ levels. Since the size of a chain at a node of height i is at most $r/2^i$, the maximum number of splits possible is $(\lceil \log r \rceil - i)$. Also, recall that a chain requires four comparisons for each split in the worst-case. Thus, the number of comparisons is bounded by:

$$4 \sum_{i=0}^{j} 2(2^i)(\lceil \log r \rceil - i), \quad j = \lfloor \log r \rfloor$$

since there can be at most $2(2^i)$ chains at level i. Thus the number of comparisons for splitting is $O(r)$. □

In particular, Lemma 4.2 implies that, when $m = \Theta(n)$, the total number of comparisons for the merging algorithm is $(m + n) + O(n \log \log n / \log n)$. This matches the number of comparisons in the parallel algorithm of Hagerup and Rüb [7]. When one of the list is smaller than the other, however, our algorithm uses fewer comparisons. Consider the second case, when $r > m$.

Lemma 4.3 *If $r > m$, the parallel multiselection algorithm performs r selections in $O(m \log(r/m))$ comparisons.*

Proof: Using Lemma 4.1 and arguments similar to the ones in the proof of previous lemma, we know that the maximum size of a chain is $r/2^i$ at height i. This chain can split at most $(\lceil\log r\rceil - i)$ times. Hence the number of comparisons needed for splitting is bounded by

$$4\sum_{i=0}^{\lfloor\log m\rfloor} 2(2^i)(\lceil\log r\rceil - i)$$

which is $O(m\log(r/m))$. After the chains have split, there may be at most $O(m)$ chains remaining. The number of comparisons required is then bounded by

$$\sum_{i=0}^{\lfloor\log m\rfloor} 2(2^i)(\lceil\log m\rceil - i) \quad = O(m).$$

□

Hence, if $(m+n)/\log(m+n) > m$, or $m < n/\log n$ approximately, then our merging algorithm requires only

$$(m+n) + O\left(m\log\frac{m+n}{m\log(m+n)}\right)$$

comparisons, which is better than that of Hagerup and Rüb's parallel algorithm [7]. Note that in the preceding analysis, we need not consider the integer sorting used in the post-processing phase of the multiselection algorithm as it does not involve any key comparisons.

The sequential complexity of our multiselection algorithm matches the information-theoretic lower bound for the multiselection problem.The number of operations performed by our parallel multiselection algorithm also matches the lower bound if we have an optimal integer sorting algorithm for the EREW PRAM.

References

[1] S. G. Akl and N. Santoro. Optimal parallel merging and sorting without memory conflicts. *IEEE Transactions on Computers*, C-36(11):1367–1369, November 1987.

[2] R. J. Anderson, E. W. Mayr, and M. K. Warmuth. Parallel approximation algorithms for bin packing. *Information and Computation*, 82:262–277, September 1989.

[3] G. Bilardi and A. Nicolau. Adaptive bitonic sorting: An optimal parallel algorithm for shared memory machines. *SIAM Journal on Computing*, 18(2):216–228, April 1989.

[4] R. J. Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, August 1988.

[5] N. Deo and D. Sarkar. Parallel algorithms for merging and sorting. *Information Sciences*, 51:121–131, 1990. Preliminary version in Proc. Third Intl. Conf. Supercomputing, May 1988, pages 513–521.

[6] G. N. Frederickson and D. B. Johnson. The complexity of selection and ranking in $X + Y$ and matrices with sorted columns. *Journal of Computer and System Sciences*, 24:197–208, 1982.

[7] T. Hagerup and C. Rub. Optimal merging and sorting on the EREW PRAM. *Information Processing Letters*, 33:181–185, December 1989.

[8] J. JaJa. *An Introduction to Parallel Algorithms*. Addison-Wesley, Reading, MA, 1992.

[9] W. Paul, U. Vishkin, and H. Wagener. Parallel dictionaries on 2-3 trees. In *Proceedings of ICALP, 154*, pages 597–609, July 1983. Also R.A.I.R.O. Informatique Theorique/Theoretical Informatics, 17:397–404, 1983.

[10] Y. Shiloach and U. Vishkin. Finding the maximum, merging, and sorting in a parallel computation model. *Journal of Algorithms*, 2:88–102, 1981.

[11] P. J. Varman, B. R. Iyer, B. J. Haderle, and S. M. Dunn. Parallel merging: Algorithm and implementation results. *Parallel Computing*, 15:165–177, 1990.