

Parallel Algorithms for Merging and Sorting

NARSINGH DEO

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

and

DILIP SARKAR

*Department of Mathematics and Computer Science, University of Miami,
Coral Gables, Florida 33124***ABSTRACT**

We present an $O(\log(\min(m, n, j)))$ -time sequential algorithm to select the j th-smallest element of an array resulting from the merging of two sorted arrays A and B of sizes m and n . This algorithm is then used to design two parallel algorithms to partition A and B into p subarrays of size $O((m + n)/p)$ in $O(N/p + \log N)$ and $O(N/p + \log p \log N)$ times on CREW PRAM and EREW PRAM models with p processors, where $N = m + n$. A third parallel algorithm has been presented that partitions A and B in $O(N/p + \log p \log N)$ time into p pairs of subarrays of size $O(N/p)$ on EREW PRAM with p processors without using any selection algorithm. Finally, these partitioning algorithms are used to obtain optimal parallel merging and sorting algorithms on CREW PRAMS and EREW PRAMS.

1. INTRODUCTION

Given two sorted arrays $A = \langle a_1, a_2, \dots, a_m \rangle$ and $B = \langle b_1, b_2, \dots, b_n \rangle$, and a parallel machine with p processors, our goal is to merge A and B into a single array C of size $N = m + n$, efficiently. In order to perform merging in parallel, we first devise three different parallel algorithms for partitioning the given arrays A and B into p pairs of subarrays of approximately equal size. Each partitioning algorithm gives rise to a distinct merging algorithm. However, before presenting these six parallel algorithms (three for partitioning and corresponding three for merging), we first present a sequential selection algorithm, which finds the j th element of the merged array C without explicitly merging A and B . This selection algorithm constitutes the core of two of the partitioning algorithms, using an approach similar to the one taken by Akl and Santoro [2]. Finally, we discuss how these merging algorithms can be used for sorting in parallel, efficiently.

Our computation model is the commonly used synchronous parallel random access memory (PRAM) machine consisting of a shared global memory and p processors $P_1, P_2, \dots, P_p$. We are considering two distinct members of PRAM

family here: a) concurrent read and exclusive write (CREW) model and b) exclusive-read and exclusive-write (EREW) model. A detailed discussion of the PRAM family can be found in a survey by Vishkin [15].

Let $T(p)$ be the time required by a parallel algorithm to solve a given problem using p processors. The parallel algorithm is said to be *cost-optimal* or simply *optimal* if the product of parallel-time $T(p)$ and the number of processors p is within a multiplicative constant of the lower bound on sequential computation time for the problem. At present only a few problems are known to have polylogarithmic-time optimal parallel algorithms. Also, an optimal parallel-algorithm on CRCW (respectively CREW) PRAM may not always be transformed into an optimal parallel-algorithm in CREW (respectively EREW) PRAM [8]. Only three optimal parallel sorting algorithms are known on EREW PRAM. The parallel bitonic-sorting algorithm in [4] is optimal and the multiplicative constant is also small. The merge sort algorithm in [7] on EREW PRAM is optimal but the multiplicative constant is "somewhat less small" as observed by the author himself. Parallel merge-sort algorithm in [2] on EREW PRAM would be optimal if the selection algorithm were properly adapted (see Section 2). Details about sequential and parallel sorting can be found in [1, 5, 6, 9, 10, 13, 14].

2. SELECTION ALGORITHM

In this section a sequential algorithm for a selection problem is presented. The selection problem considered here is different from the usual selection problem. We are given two monotonically increasing sorted arrays $A = \langle a_1, a_2, \dots, a_m \rangle$ and $B = \langle b_1, b_2, \dots, b_n \rangle$ of distinct elements. (The last constraint is imposed just to make the presentation and analysis of the algorithms simple.) Let $C = \langle c_1, c_2, \dots, c_N \rangle$ be the sorted array obtained by merging two arrays A and B , where $N = m + n$. The problem is of finding k th-smallest element in C without explicitly constructing C .

Without loss of generality it can be assumed that $m \leq n$, as otherwise we can rename the arrays. We consider two different cases: (i) $k \leq (m + n)/2$ and (ii) $k > (m + n)/2$. First, case (i) is considered and then it is shown how case (ii) can be converted into a case similar to case (i).

If $k \geq m$, each of the first $(k - m)$ elements of the larger array B is always smaller than k th-smallest elements in the merged array C . Also, each of the last $(n - k)$ elements in the larger array B is greater than the k th-smallest element in C . Thus, the search for the k th-smallest element in the array B could be confined to the elements b_{k-m+1} to b_k . But in array A we need to search among all the elements. The procedure presented here for searching k th-smallest element always keeps two elements a_u from A and b_v from B under consideration such that total number of elements in $A_1 = \langle a_1, a_2, \dots, a_u \rangle$

and $B_1 = \langle b_1, b_2, \dots, b_r \rangle$ is k . Whenever the value of u is decremented (incremented) by d , the value of v is incremented (decremented) by d , keeping $u + v = k$. We use a binary search technique to select the k th-smallest element. An informal description of the selection procedure is as follows. Initially we set $u = \lfloor m/2 \rfloor$ and $v = (k - u)$. At i th iteration, $1 \leq i \leq \lfloor \log m \rfloor - 1$, if $a_u \geq b_v$, then u is decremented by $\lfloor m/2^{i+1} \rfloor$ and v is incremented by an equal amount. At termination, one of the elements from $\{a_u, b_v, b_{v+1}\}$ is the k th-smallest element, which can then be determined in constant time. On the other hand, if $k \leq m \leq n$, we initially set $u = \lfloor k/2 \rfloor$ and $v = k - u$, and only $\lfloor \log k \rfloor - 1$ iterations are required. A Pascal-like description of the selection procedure is as follows.

```

PROCEDURE SELECT- $k$  th-SMALLEST ( $A, B, k, x, w$ );
(* Comments: Input:  $A = \langle a_1, a_2, \dots, a_m \rangle$  and
 $B = \langle b_1, b_2, \dots, b_n \rangle$  are sorted arrays,  $k \leq \lfloor (m + n)/2 \rfloor$  and  $m \leq n$ .
Output:  $x$  is the array name ( $A$  or  $B$ ) and  $w$  is the index of the element
selected. *)
if  $k \geq m$  then
   $u := \lfloor m/2 \rfloor$ ;  $v := \max(1, k - u)$ ;
  for  $i := 1$  to  $\lfloor \log m \rfloor - 1$  do
    if  $a_u \geq b_v$  then
       $u := \max(1, u - \lfloor m/2^{i+1} \rfloor)$ ;  $v := k - u$ ;
    else (*  $b_v > a_u$  *)
       $u := \min(m, u + \lfloor m/2^{i+1} \rfloor)$ ;  $v := \max(1, k - u)$ ;
    endif
  else (*  $k < m$  *)
     $u := \lfloor k/2 \rfloor$ ;  $v := k - u$ ;
    for  $i := 1$  to  $\lfloor \log k \rfloor - 1$  do
      if  $a_u \geq b_v$  then
         $u := \max(1, u - \lfloor k/2^{i+1} \rfloor)$ ;  $v := k - u$ ;
      else (*  $b_v > a_u$  *)
         $u := u + \lfloor k/2^{i+1} \rfloor$ ;  $v := \max(1, k - u)$ ;
      endif
    endif
  endif
  if ( $u = k$ ) then
    if ( $a_u > b_v$ ) then
       $x := 'B'$ ;  $w := v$ 
    else (*  $a_u \leq b_v$  *)
       $x := 'A'$ ;  $w := u$ 
    endif
  else
    if ( $u = 1$ ) then

```

```

    if ( $a_u > b_{v+1}$ ) then
       $x := 'B'; w := v + 1;$ 
    else ( $* a_u < b_{v+1} *$ )
       $x := 'A'; w := u$ 
    endif
  endif
endif
if ( $1 < u$ ) and ( $u < k$ ) then
  if  $a_u > b_v$  then
     $x := 'A'; w := u;$ 
  else
     $x := 'B'; w := v;$ 
  endif
endif
endif

```

Procedure **SELECT- k th-SMALLEST** (A, B, k, x, w) describes how to select k th-smallest element from A and B when $k \leq [(m+n)/2]$. If $k > [(m+n)/2]$, k th-smallest is l th largest ($l = m + n - k + 1$) and $l \leq [(m+n)/2]$. It is an easy task to modify procedure **SELECT- k th-SMALLEST** (A, B, k, x, w) to find the l th largest for $l \leq [(m+n)/2]$. We leave the necessary modifications of **SELECT- k th-SMALLEST** (A, B, k, x, w) to the reader. Let the modified procedure be **SELECT- l th-LARGEST** (A, B, l, x, w). These two special case selection procedures are used to develop a j th-smallest selection algorithm for $1 \leq j \leq N$. A Pascal-like description of the algorithm is presented next.

```

PROCEDURE SELECT- $j$  th-SMALLEST ( $A, B, j, x, w$ );
(* Comments: Input:  $A = \langle a_1, a_2, \dots, a_m \rangle$  and
 $B = \langle b_1, b_2, \dots, b_n \rangle$  are sorted arrays,
 $m \leq n$ , and  $1 \leq j \leq (m+n)$ .
Output:  $x$  is the array name ( $A$  or  $B$ ) and  $w$  is the
index of the element in the array  $x$ . *)
if  $j \leq [(m+n)/2]$  then
  SELECT- $k$  th-SMALLEST ( $A, B, j, x, w$ );
else
   $l := m + n - j + 1;$ 
  SELECT- $l$  th-LARGEST ( $A, B, l, x, w$ );
endif

```

Two remarks are in order here. First, Akl and Santoro have also presented a sequential selection algorithm for two sorted arrays A and B . Unfortunately, their algorithm may fail if $m \neq n$. For example, when $A = \langle 1, 2 \rangle$ and $B = \langle 3, 4, \dots, 34 \rangle$ the Akl-Santoro algorithm will terminate with $a_u = 2$ and $b_v = 10$. Thus, $X = \langle 1, 2 \rangle$ and $Y = \langle 9, 10, 11 \rangle$ do not include the median which

is 17. We believe that their selection algorithm fails in general because it is a special case of the algorithm in [11] for computing median of two equal size sets. However, the algorithm may be modified by adapting the method presented in [12]. But then the modified algorithm would not utilize the special structure of the arrays. The selection algorithm presented here is designed to utilize the special property (sortedness) of the arrays.

Second, our algorithm is different from the distributed selection algorithms described in [11] and [12] in three ways: Input arrays here are sorted. The complexity measure of the distributed algorithms is the number of communication steps and not the total number of computational steps. In the distributed algorithms, the median of each set is computed, and then these medians are used to reduce the size of each set.

3. PARTITIONING ALGORITHMS

We now consider the problem of partitioning of a pair of sorted arrays of sizes m and n into p pairs of subarrays $\langle A_i, B_i \rangle$, $1 \leq i \leq p$, such that total number of elements in A_i and B_i is $O(N/p)$, where $N = m + n$. Moreover, the largest element in $\langle A_i, B_i \rangle$ is smaller than every element in $\langle A_{i+1}, B_{i+1} \rangle$ for $1 \leq i \leq (p-1)$. Using the j th-smallest selection algorithm presented in previous section, two algorithms are presented. The first algorithm partitions A and B into p pairs of subarrays in $O(\log N)$ time using a p -processor CREW PRAM. The second algorithm uses the less powerful model, EREW PRAM, also with p processors, but runs for a longer time, namely $O(\log p \log N)$ time. Let $P = \{P_1, P_2, \dots, P_p\}$ be the set of processors available for each model.

PROCEDURE PARTITION-1(A, B, P, I, J);

(* Comments: Input: A and B are sorted
arrays and P is the set of processors.

Output: $I = \langle i_1, i_2, \dots, i_p \rangle$ and

$J = \langle j_1, j_2, \dots, j_p \rangle$ are the indices of the
beginning elements of the partitions A_k, B_k . *)

do in parallel for all k , $1 \leq k \leq (p-1)$

 SELECT- j th-SMALLEST($A, B, \lfloor N/p \rfloor * k, x, w$);

 compute i_k, j_k ;

PROCEDURE PARTITION-2(A, B, P, I, J);

(* Parameters are as in the previous procedure *)

if $|P| > 1$ **then**

P_1 finds k such that $2^{k-1} < |P| < 2^k$;

 SELECT- j th-SMALLEST($A, B, \lfloor N/p \rfloor * 2^{k-1}, x, w$);

```

compute  $i_{2^{k-1}}$  and  $j_{2^{k-1}}$ ;
 $\underline{P} := P_1, P_2, \dots, P_{2^{k-1}}$ ;
 $\bar{P} := P_{2^{k-1}+1}, P_{2^{k-1}+2}, \dots, P_p$ ;
do in parallel
  PARTITION-2( $A_1, B_1, \underline{P}, I_1, J_1$ );
  PARTITION-2( $A_2, B_2, \bar{P}, I_2, J_2$ );
endif

```

Now we present another partitioning algorithm that does not use a selection algorithm. The following two lemmas are consequence of our assumption about arrays A and B .

LEMMA 1. *For each a_i , $1 \leq i \leq m$, one and only one of the following holds: (i) $b_n < a_i$, that is, every element in array B is smaller than a_i , (ii) $b_1 > a_i$, that is, every element in array B is greater than a_i , (iii) $b_j < a_i < b_{j+1}$, $1 \leq j < n$.*

LEMMA 2. *For each b_j , $1 \leq j \leq n$, one and only one of the following holds: (i) $a_m < b_j$, that is, every element in array A is smaller than b_j , (ii) $a_1 > b_j$, that is, every element in array A is greater than b_j , (iii) $a_i < b_j < a_{i+1}$, $1 \leq i < m$.*

The following procedure, PROCEDURE DIVIDE, partitions arrays A and B into two pairs of subarrays (possibly empty) $\langle A_1, A_2 \rangle$ and $\langle B_1, B_2 \rangle$ such that

$$\frac{1}{4}N \leq (|A_i| + |B_i|) \leq \frac{3}{4}N, \quad \text{for } i = 1 \text{ and } 2.$$

```

PROCEDURE DIVIDE( $A, B, P_1, u, v$ );
(* Comments: Input: Sorted arrays  $A$  and  $B$ 
Output: Indices  $u$  and  $v$  such that
        either (i)  $b_v < a_u < b_{v+1}$  or (ii)  $a_u < b_v < a_{u+1}$ .
 $P_1$  is the processor that does the computation. *)
if  $m \geq n$  then
   $u := \lceil m/2 \rceil$ ;
  using binary search on array  $B$  find  $v$  such
  that  $b_v < a_u < b_{v+1}$ 
else (*  $n > m$  *)
   $v := \lceil n/2 \rceil$ ;
  using binary search on array  $A$  find  $u$  such
  that  $a_u < b_v < a_{u+1}$ 
endif

```

Define

$$A_1 = \langle a_1, a_2, \dots, a_u \rangle, \quad A_2 = \langle a_{u+1}, a_{u+2}, \dots, a_m \rangle, \\ B_1 = \langle b_1, b_2, \dots, b_v \rangle \quad \text{and} \quad B_2 = \langle b_{v+1}, b_{v+2}, \dots, b_n \rangle.$$

LEMMA 3. *PROCEDURE DIVIDE partitions A and B into $\langle A_1, A_2 \rangle$ and $\langle B_1, B_2 \rangle$ such that, $\frac{1}{4}N \leq (|A_i| + |B_i|) \leq \frac{3}{4}N$, for $i = 1$ and 2 .*

Proof. Two cases to be considered:

Case 1: $m \geq n$. In this case $u = \lceil m/2 \rceil$ and $0 \leq v \leq n$. Thus, $m/2 + 0 \leq u + v \leq m/2 + n$ which implies that $\frac{1}{4}N \leq u + v \leq \frac{3}{4}N$ or $\frac{1}{4}N \leq (|A_1| + |B_1|) \leq \frac{3}{4}N$ (since $|A_1| = u$ and $|B_1| = v$).

Similarly, it can be shown that $\frac{1}{4}N \leq (|A_2| + |B_2|) \leq \frac{3}{4}N$.

Case 2: $m < n$. Proof for this case is similar to the one for Case 1. ■

The following procedure, PROCEDURE PARTITION-3, partitions two sorted arrays A and B into p subarrays calling PROCEDURE DIVIDE. We show that PROCEDURE PARTITION-3 partitions A and B into p pairs $\langle A_i, B_i \rangle$ in $O(\log p \log N)$ parallel time steps using p processors such that $(|A_i| + |B_i|) = O(N/p)$.

PROCEDURE PARTITION-3(A, B, P, I, J);

if $|P| > 1$ then

DIVIDE(A, B, P_1, u, v);

$k := (u + v) * p / N$;

$k := \text{round}(k)$;

$\underline{P} := P_1, P_2, \dots, P_k$;

$\bar{P} := P_{k+1}, P_{k+2}, \dots, P_p$;

do in parallel

PARTITION-3($A_1, B_1, \underline{P}, I_1, J_1$);

PARTITION-3($A_2, B_2, \bar{P}, I_2, J_2$);

endif

The preceding procedure partitions the arrays A and B into p pairs of subarrays. The following lemmas give running time of procedure PARTITION-3 and the size of each pair of arrays at termination of procedure PARTITION-3.

LEMMA 4. *When PROCEDURE PARTITION-3 terminates, the largest element in A_i and B_i is smaller than the smallest element in A_{i+1} and B_{i+1} , $1 \leq i \leq p - 1$.*

Proof. The boundary of A_i and A_{i+1} (respectively B_i and B_{i+1}) are nonoverlapping. Also, at every stage of partitioning procedure DIVIDE is called which ensures that each element to the left of the boundary is smaller than those to the right of the boundary. Thus, the lemma holds. ■

LEMMA 5. *Each pair $\langle A_i, B_i \rangle$, $1 \leq i \leq p$, has $O(N/p)$ elements, when PARTITION-3 terminates.*

Proof. We prove this lemma by induction on the number of partitions.

Basis: If number of partitions is two, $(|A_1| + |B_1|) = O(N/2)$ and $(|A_2| + |B_2|) = O(N/2)$ by Lemma 3. Thus, the lemma holds for $p = 2$.

Induction Hypothesis: Let $(|A_i| + |B_i|) = O(N/h)$, for $2 \leq h < p$, and $1 \leq i \leq h$.

Induction Step: To show that $(|A_i| + |B_i|) = O(N/(h+1))$ for $1 \leq i \leq (h+1)$.

At the first call of PROCEDURE DIVIDE, arrays A and B are divided into subarrays $A = \langle A', A'' \rangle$ and $B = \langle B', B'' \rangle$. Now we have two problems: Partitioning A' and B' using $k = \text{round}((|A'| + |B'|)/N * p)$ processors and partitioning A'' and B'' using $((h+1) - k)$ processors. The number of elements in each partition can be bounded by the inequalities

$$\left(\frac{N}{h+1} * k + \frac{N}{h+1} - 1 \right) \geq (|A'| + |B'|), \quad (1)$$

$$(|A'| + |B'|) \geq \left(\frac{N}{h+1} * k - \frac{N}{h+1} + 1 \right), \quad (2)$$

$$\left(\frac{N}{h+1} * (h+1-k) + \frac{N}{h+1} - 1 \right) \geq (|A''| + |B''|), \quad (3)$$

$$(|A''| + |B''|) \geq \left(\frac{N}{h+1} * (h+1-k) - \frac{N}{h+1} + 1 \right). \quad (4)$$

When one of the partitions attains the lower bound in inequalities (1) or (2), the other attains the upper bound. The partition with lower bound can be partitioned into subarrays such that each partition has $O(N/(h+1))$ elements (by induction hypothesis). Let us assume (without loss of generality) that $(|A'| + |B'|)$ attains the upper bound, that is, $(|A'| + |B'|) = (N/(h+1) * k + N/(h+1) - 1)$.

By induction hypothesis A' and B' can be partitioned into k subarray pairs such that $(|A_i| + |B_i|) = O((N/(h+1) * k + N/(h+1) - 1)/k) = O(N/(h+1) + N/(h+1) * k - 1/k) = O(N/(h+1))$. ■

LEMMA 6. PROCEDURE PARTITION-3 terminates in $O(\log p \log N)$ time.

Proof. We know that PARTITION-3 calls PROCEDURE DIVIDE once and that recursively PARTITION-3 calls itself twice in one parallel-time step. Thus, to show that PARTITION-3 terminates in $O(\log p \log N)$ time, it is sufficient to show that there are at most $O(\log p)$ parallel recursive-calls to PROCEDURE PARTITION-3. The calls to PARTITION-3 can be represented as a labeled binary tree. The node labels are the distance of the nodes from the root.

By Lemma 3, it is known that the maximum size of a subarray pair at level 1 (of the procedure call tree) is $\frac{3}{4}N$, at level 2 it is $\frac{3}{4}(\frac{3}{4}N) = (\frac{3}{4})^2N$. In general, at level i maximum size of any pair of subarrays is $(\frac{3}{4})^iN$. Thus, at level $\log_{4/3} p$, the maximum size of any subarray pair is $(\frac{3}{4})^{\log_{4/3} p}N = N/p = O(N/p)$.

Thus, the subarray size would be $O(N/p)$ after at most $O(\log p)$ parallel calls to PROCEDURE PARTITION-3. Hence, the proof of the lemma. ■

THEOREM 1. *PROCEDURE PARTITION-3 divides a pair of arrays A and B into p pairs of subarrays in $O(\log p \log N)$ -time. Also, the size of each pair of subarray is $O(N/p)$.*

4. MERGING AND SORTING ALGORITHMS

The partitioning algorithms presented in previous section are used in this section to design parallel merging and sorting algorithms. First, partitioning algorithm is invoked and the arrays are partitioned into p subarray pairs, and then these subarray pairs are merged in parallel as in [2]. It is worth mentioning that the merging algorithm that uses PROCEDURE PARTITION-3 for partitioning have to compute the index of the beginning position of each processor in the merged array C . This can be done using any partial-sum computation algorithm [3].

PROCEDURE MERGE-1(A, B, C, P);

PARTITION-1(A, B, P, I, J);

do in parallel for all $i, 1 \leq i \leq p$

 Merge A_i and B_i using processor P_i and store
 into C starting at location $((i-1)*(m+n)p)+1$;

PROCEDURE MERGE-1 takes $O(N/p + \log N)$ time on CREW PRAM and is optimal for $p \leq N/\log N$. In PROCEDURE MERGE-1, when PROCEDURE PARTITION-1 is replaced by PROCEDURE PARTITION-2, the resulting merging algorithm, say PROCEDURE MERGE-2, takes $O(N/p + \log p \log N)$ time on EREW PRAM and is optimal for $p \leq N/\log^2 N$. A third parallel merging algorithm is obtained when PROCEDURE PARTITION-3 is used. The algorithm is presented next.

PROCEDURE MERGE-3(A, B, C, P);

PARTITION-3(A, B, P, I, J);

do in parallel for $i, 1 \leq i \leq p$

 Processor P_i counts the number of elements in A_i and B_i

 (* Let N_i be the number of elements

 in A_i and B_i and $\sum_{j=1}^i N_j = X_i$ *)

 Compute X_i s, the partial sums;

 Merge A_i and B_i and store into C starting

 at location $X_{i-1} + 1$

Computation of X_i s in the preceding procedure can be performed in $O(\log p)$ time (see [3]). All other computations can be performed in $O(N/p + \log p \log N)$ time on EREW PRAM and thus, MERGE-3 is optimal when $p \leq N/\log^2 N$.

Using merging algorithms presented here parallel sorting can be performed as follows: Divide the list of elements to be sorted into p equal-size sublists and sort each sublist sequentially by one processor using some efficient sequential sorting algorithm. Then, in parallel merge two list into one successively, until a single sorted list is obtained. To see that the implementation of this sorting algorithm can be achieved in $O((N/p)\log(N/p) + (N/p + \log p \log N)\log(N/p))$ time is straightforward. The sorting algorithm is optimal for $p \leq N/\log^2 N$ on the EREW PRAM.

EXAMPLE. Let $L = \{9, 1, 8, 6, 14, 3, 10, 12, 11, 13, 2, 7, 16, 4, 15, 5\}$ be the list to be sorted by processors P_1, P_2, P_3 , and P_4 . Initially, each of P_1, P_2, P_3 , and P_4 will sort one fourth of the list obtaining $A = \langle 1, 6, 8, 9 \rangle$, $B = \langle 3, 10, 12, 14 \rangle$, $C = \langle 2, 7, 11, 13 \rangle$, and $D = \langle 4, 5, 15, 16 \rangle$. Now, A and B will be partitioned by P_1 to obtain $A_1 = \langle 1, 6, 8 \rangle$, $A_2 = \langle 9 \rangle$, $B_1 = \langle 3 \rangle$, and $B_2 = \langle 10, 12, 14 \rangle$. Similarly, C and D will be partitioned by P_3 . Then, first merging phase will produce two sorted lists $E = \langle 1, 3, 6, 8, 9, 10, 12, 14 \rangle$ and $F = \langle 2, 4, 5, 7, 11, 13, 15, 16 \rangle$, which will be partitioned into $E_1 = \langle 1, 3 \rangle$, $E_2 = \langle 6, 8 \rangle$, $E_3 = \langle 9, 10, 12 \rangle$, $E_4 = \langle 14 \rangle$, $F_1 = \langle 2, 4 \rangle$, $F_2 = \langle 5, 7 \rangle$, $F_3 = \langle 11 \rangle$, and $F_4 = \langle 13, 15, 16 \rangle$. Finally, $P_i, 1 \leq i \leq 4$, will merge E_i and F_i to produce the sorted list.

5. CONCLUSION

In this paper, we have presented several algorithms for three comparison problems—selection, merging, and sorting. First, we have presented an $O(\log(\min(m, n, j)))$ -time sequential algorithm to select the j th-smallest element from two sorted arrays A and B of sizes m and n . This algorithm was then used to design two parallel algorithms to partition A and B into p subarrays, each of size $O((m+n)/p)$ on CREW PRAM as well as EREW PRAM models. These algorithms require $O(N/p + \log N)$ and $O(N/p + \log p \log N)$ time and are optimal for $p \leq N/\log N$ and $p \leq N/\log^2 N$, respectively. A third partitioning algorithm has also been presented that does not use the selection algorithm, but partitions A and B in $O(N/p + \log p \log N)$ time into p pairs of subarrays of size $O(N/p)$ on an EREW PRAM. Finally, these partitioning algorithms are used to obtain optimal parallel merging and sorting algorithms on CREW PRAMS and EREW PRAMS.

We would also like to point out that, although on the surface the sorting algorithm presented here may appear very similar to the one proposed by Akl

and Santoro in [2], a closer look will reveal that there are fundamental differences in the functioning and performance of the two key procedures (i.e., selection and partitioning) used by both algorithms. The selection algorithm of [2] (after the necessary corrective measures are taken) is an adaptation of a distributed selection algorithm of [11], and does not seem to be very efficient in this application. Secondly, while the algorithm proposed here will select the k th-smallest value, for any k , the one in [2] is for selecting the median only. Consequently, the partitioning algorithm of [2] partitions A and B into 2^i pairs and not into an arbitrary number of pairs, as is the case for the partitioning algorithms proposed here.

REFERENCES

1. S. G. Akl, *Parallel Sorting Algorithms*, Academic, Orlando, Fla., 1985.
2. S. G. Akl and N. Santoro, Optimal parallel sorting without memory conflicts, *IEEE Trans. Comput.* C-36 (11):1367–1369 (1986).
3. I. Bar-On and U. Vishkin, Optimal parallel generation of computation tree form, *ACM Trans. Prog. Lang. Syst.* 7(2):348–357 (1985).
4. G. Bilardi and A. Nicolau, Bitonic sorting with $O(n \log n)$ comparisons, in *Proc. Twentieth Annual Conf. on Information Sciences and Systems*.
5. D. Bitton, D. J. DeWitt, D. K. Hsiao, and J. Menon, A taxonomy of parallel sorting, *Comput. Surv.* 16:287–318 (1984).
6. A. Borodin and J. E. Hopcroft, Routing, merging, and sorting on parallel models of computation, in *Proc. 14th ACM Symp. Theory Comput.*, San Francisco, Cal., 1982, pp. 338–344.
7. R. Cole, Parallel Merge Sort, in *Proc. IEEE 1986 27th Symp. Found. Comput. Sci.*, Toronto, Canada, 1986, pp. 511–516.
8. D. M. Eckstein, Simultaneous memory accesses, Tech. Rep. 79-6, Dept. Comput. Sci., Iowa State Univ., Ames, IA, Aug. 1979.
9. D. E. Knuth, *The Art of Computer Programming*, Addison-Wesley, Reading, Mass., 1973, vol. 3.
10. C. P. Kruskal, Searching, merging and sorting in parallel computation, *IEEE Trans. Comput.* C-32:942–962 (1983).
11. M. Rodeh, Finding the median distributively, *J. Comput. Syst. Sci.* 24:162–167 (1982).
12. N. Santoro and J. B. Sidney, Order statistics on distributed sets, in *Proc. 20th Allerton Conf. Commun. Contr. Comput.*, Monticello, Ill., 1982, pp. 251–256.
13. Y. Shiloach and U. Vishkin, Finding the maximum, merging, and sorting in a parallel computation model, *J. Algorithms* 2:88–102 (1981).
14. L. G. Valiant, Parallelism in comparison problems, *SIAM J. Comput.* 4:348–355 (1975).
15. U. Vishkin, Synchronous parallel computation—a survey, Tech. Rep. 71, Courant Institute, New York University, New York, 1983.

Received 24 June 1988; revised 30 May 1989