INFORMATIQUE THÉORIQUE ET APPLICATIONS

J. KATAJAINEN
C. LEVCOPOULOS
O. PETERSSON

Space-efficient parallel merging

Informatique théorique et applications, tome 27, nº 4 (1993), p. 295-310.

<http://www.numdam.org/item?id=ITA_1993__27_4_295_0>

Informatique théorique et Applications/Theoretical Informatics and Applications
(vol. 27, n° 4, 1993, p. 295 à 310)

SPACE-EFFICIENT PARALLEL MERGING (*)

by J. Katajainen (¹), C. Levcopoulos (²) and O. Petersson (³)

Communicated by I. Wegener

Abstract. – *The problem of designing space-efficient parallel merging algorithms is examined. It is shown that two sorted sequences of lengths m and n,* $m \leqq n$, *can be merged in* $O(n/p+\log n)$ *time on an EREW PRAM with p processors, using only a constant amount of extra storage per processor. After a slight modification, the algorithm runs on a DCM (Direct Connection Machine) within the same resource requirements. Moreover, using similar techniques, it is shown that merging can be accomplished in* $O(n/p+\log\log m)$ *time on a CREW PRAM with p processors, and* $O(1)$ *extra space per processor. Our algorithms use a sequential algorithm for in-place merging as a subroutine; if this is stable, the parallel algorithms are stable as well.*

Résumé. – *Le problème de la conception d'un algorithme de fusion parallèle performant en espace est étudié. Il est montré que deux suites triées de longueur m et n,* $m \leqq n$ *peuvent être fusionnée en temps* $O(n/p+\log n)$ *sur un modèle PRAM à lecture et écriture exclusives avec p processeurs, en utilisant une quantité de mémoire supplémentaire constante par processeur. Après de légères modifications, l'algorithme tourne sur une machine à connexion directe (DCM) avec les mêmes conditions sur les ressources. De plus, en utilisant des techniques similaires, il est montré que la fusion peut être réalisée en temps* $O(n/p+\log\log m)$ *sur un modèle PRAM à lecture concurrente et écriture exclusive avec p processeurs, et une mémoire supplémentaire en* $O(1)$ *par processeur. Nos algorithmes utilisent comme sous-routine un algorithme séquentiel de fusion en place; si celui-ci est stable alors l'algorithme parallèle est aussi stable.*

1. INTRODUCTION

The merging problem is classical in computer science. Sequentially, it is straightforward to merge two sorted sequences of lengths m and n into one sorted sequence in $O(m+n)$ time. Standard solutions usually require a working storage of size at least $\min\{m, n\}$. We consider solving the problem

---

(*) Received November 1991, accepted June 1992.

(¹) Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen East, Denmark. On leave from Department of Computer Science, University of Turku, Lemminkäisenkatu 14 A, SF-20520 Turku, Finland.

(²) Department of Computer Science, Lund University, Box 118, S-221 00 Lund, Sweden.

(³) Department of Mathematics, Statistics and Computer Science, Växjö University, 5-35195 Växjö, Sweden.

0988-3754/93/04 295 16/$3.60/

space-efficiently, *i. e.*, using few auxiliary memory cells. The merging problem can be defined as follows:

Merging Problem: Given an array $A[1\,..\,m+n]$ containing two sorted sequences, the first in locations $1\,..\,m$ and the second in locations $m+1\,..\,m+n$, $m \leqq n$, rearrange A such that it forms one sorted sequence.

Kronrod [15] (see also [21]) devised the first merging algorithm that runs in linear time and uses only a constant amount of extra space. This algorithm is, however, complicated and unstable, *i. e.*, the relative order of equal elements is not necessarily the same in the output as in the input. Recently, research efforts have focused on simpler, more practical, and stable algorithms for merging in linear time and constant extra space [11, 12, 18, 21].

In this paper we consider space-efficient solutions of the merging problem in the following synchronous parallel machines:

DCM (Direct Connection Machine [17]; also called seclusive PRAM [27])

A DCM is a complete network of processors (RAMs), each equipped with a memory module. Every processor may access every memory module but several processors are not allowed to access the same memory module simultaneously.

EREW PRAM (Exclusive-Read-Exclusive-Write Parallel RAM)

In an EREW PRAM processors share a common memory but several processors are not allowed to access the same memory location simultaneously.

CREW PRAM (Concurrent-Read-Exclusive-Write Parallel RAM)

A CREW PRAM is like an EREW PRAM with the exception that simultaneous reading is allowed.

In these machine models each processor can perform usual arithmetical and logical operations as well as memory accesses in constant time. For a more detailed discussion on these and other models of parallel computation, the reader is referred to any textbook, e.g., [13, 20], or survey, e.g., [8, 14, 27], on parallel algorithms.

Throughout the paper, p denotes the number of processors employed, and m and n, $m \leqq n$, denote the lengths of the sorted sequences to be merged. A parallel algorithm that solves some problem of size n in $T(n)$ time using p processors is said to be *cost optimal* if $p \cdot T(n)$ matches the time bound achieved by the fastest known sequential algorithm for the problem considered. Thus, a parallel merging algorithm is cost optimal if it uses $O(n/p)$ time since in the worst case $\Theta(n)$ operations (data moves) are necessary. We further say that a parallel algorithm is *cost-space optimal* if it is cost optimal and uses only a constant amount of extra space per processor. (Guan and Langston [9] denoted this concept by time-space optimality.) We proceed by reviewing the known results on parallel merging.

Snir [25] proved that, regardless of the number of processors available, $\Omega(\log n)$ time is necessary for merging two sequences on an EREW PRAM, and hence, on a DCM as well. This lower bound is valid even if the shorter sequence contains a single element. (Observe that, for this result to apply, the word length of the extra storage locations is bounded.) On the other hand, if concurrent reads are allowed it is possible to merge in constant time with mn processors [24]. However, if the number of processors is $O(n\log^{\alpha} n)$, for any fixed constant α, $\Omega(\log\log m)$ is a lower time bound even if concurrent writes are allowed [5, 22].

By adapting Batcher's [3] bitonic or odd-even merging, one can solve the merging problem on a bounded degree network (more precisely, on a shuffle-exchange network) in $O((n\log n)/p+\log n)$ time [26], which is not cost-optimal. Recently, several cost-optimal parallel algorithms for merging on an EREW PRAM have been proposed [2, 4, 7, 10]. These algorithms run in $O(n/p+\log n)$ time, which is cost-optimal whenever $p\leqq n/\log n$. We note that all these algorithms require a workspace of size $\Theta(n)$ independently of the number of processors, and hence, they are not cost-space optimal. On stronger PRAM models merging can be done in $O(n/p+\log\log m)$ time. Kruskal [16] proved this result for parallel computation trees, but it holds for CREW PRAMs as well. Table 1 summarizes our knowledge about parallel merging.

TABLE 1

State of the art when merging two sequences of lengths m *and* n, $m\leqq n$.

Model	# of processors	Time	Reference
Shuffle-exchange	$p\leqq m+n$	$O((n\log n)/p+\log n)$	[3, 26]
EREW PRAM	$p<\infty$	$\Omega(n/p+\log n)$	[25]
EREW PRAM	$p\leqq m+n$	$O(n/p+\log n)$	[2, 4, 7, 9, 10]
CREW PRAM	$p\geqq n$	$O(\log m/\log(p/n))$	[24]
CREW PRAM	$p\leqq n\log^{\alpha} n$	$\Omega(n/p+\log\log m)$	[5]
CREW PRAM	$p\leqq m+n$	$O(n/p+\log\log m)$	[16]
CRCW PRAM	$p\leqq n\log^{\alpha} n$	$\Omega(n/p+\log\log m)$	[22]

Recently, Guan and Langston [9] reported the first cost-space optimal EREW PRAM merging algorithm, which runs in $O(n/p+\log n)$ time. In this paper we present a new, simpler algorithm that runs within the same resource bounds. That is, we prove

THEOREM 1: *Two sorted sequences of respective lengths* m *and* n, $m \leqq n$, *can be merged in* $O(n/p+\log n)$ *time on an EREW PRAM using* $O(1)$ *extra space per processor, which is cost-space optimal for* $p \leqq n/\log n$.

An advantage with the new algorithm is that it can be implemented cost-space optimally on the weaker DCM as well:

THEOREM 2: *Two sorted sequences of respective lengths* m *and* n, $m \leqq n$, *can be merged in* $O(n/p+\log n)$ *time on a DCM using* $O(1)$ *extra space per processor, which is cost-space optimal for* $p \leqq n/\log n$.

More importantly, our algorithm can be applied to solve an open problem posed by Guan and Langston, namely whether cost-space optimal merging can be accomplished in sublogarithmic time when concurrent reads are allowed:

THEOREM 3: *Two sorted sequences of respective lengths* m *and* n, $m \leqq n$, *can be merged in* $O(n/p+\log\log m)$ *time on a CREW PRAM using* $O(1)$ *extra space per processor, which is cost-space optimal for* $p \leqq n/\log\log m$.

We would like to emphasize that all our algorithms can be made stable. Also, the reader should note that they match the lower time bounds stated in Table 1, and so they are the fastest possible cost-space optimal parallel algorithms.

The organization of the paper is as follows. In Section 2, after some preliminary definitions, we introduce the paradigm upon which our algorithms are based. We also give an EREW PRAM implementation that uses $O(n)$ extra space. In Section 3 we present a cost-space optimal implementation of the EREW PRAM algorithm. A more detailed analysis of the algorithm, given in Section 4, shows that it runs on a DCM as well, proving Theorem 2. In Section 5 we turn to the faster cost-space optimal CREW PRAM implementation, and prove Theorem 3. The paper is closed with some concluding remarks in Section 6.

2. BASIC ALGORITHM

2.1. Definitions

Let $X=\langle x_1, x_2, \ldots, x_m\rangle$ and $Y=\langle y_1, y_2, \ldots, y_n\rangle$ be two sequences, sorted in nondecreasing order. Recall that the sequences are given within a single array A, starting with X followed by Y. We use $X[l..h]$ to denote the subsequence $\langle x_l, x_{l+2}, \ldots, x_h\rangle$. (If $l>h$ then the subsequence is empty.)

We will divide input sequence X into *blocks* of b consecutive elements, where the block X_i is the subsequence $X[(i-1)b+1..\min\{ib, m\}]$, $i \leq \lceil m/b \rceil$. Observe that the last block might contain less than b elements. The first element of a block is called the *head* and the last the *tail*. Further, an X_i-*head* is the head of the block X_i.

To ensure stability, if two elements x_i in X and y_j in Y are equal, x_i is considered to be smaller. The following definitions are adopted from Cole [6]. For an element y_j in Y, *cross-rank* (y_j, X) is the number of elements in X that are smaller than y_j. The cross-rank concept can naturally be generalized to blocks. Let Y_k be a Y-block with head y_h. Now, *cross-rank* (Y_k, X) is the number of blocks in X having a head smaller than y_h. For convenience, let $x_0 = -\infty$ and $x_{m+1} = \infty$. If $x_i < y_j < x_{i+1}$, we say that x_i and x_{i+1} *straddle* y_j.

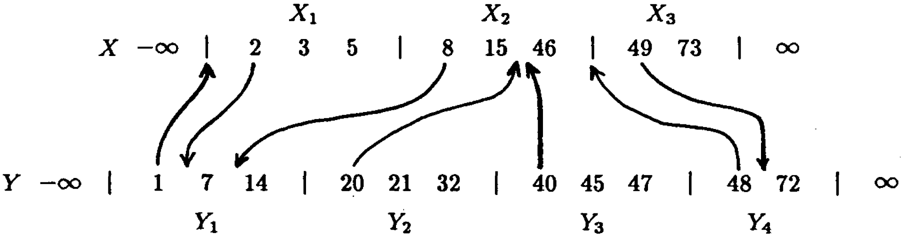
X_1
X_2
X_3
X $-\infty$ | 2 3 5 | 8 15 46 | 49 73 | ∞
Y $-\infty$ | 1 7 14 | 20 21 32 | 40 45 47 | 48 72 | ∞
Y_1
Y_2
Y_3
Y_4

Figure 1. – Two sequences divided into blocks. Arrows denote cross-ranks.

The above definitions are illustrated in Figure 1.

The cross-ranks of the heads divide the blocks into subblocks. Consider a block X_i with head x_h. Let x_k and x_{k+1} be the first pair of elements to the right of x_h straddling a Y-head. The subblock $X[(i-1)b+1..\min\{k, ib, m\}]$ is called the *heading subblock* of X_i. Further, let y_l and y_{l+1} be the two elements of Y that straddle x_h. Then block Y_j, to which y_l belongs, is called the *cross-block* of X_i. Moreover, let $y_{l'}$ and $y_{l'+1}$ be the element pair that straddles the head of X_{i+1} (for the last X-block, $l' = n$). The subblock $Y[l+1..\min\{l', jb, n\}]$ of Y_j is called the *accompanying subblock* of X_i. For example, in Figure 1 the heading subblock of X_2 is the subsequence $\langle 8, 15 \rangle$, and its accompanying subblock is the subsequence $\langle 14 \rangle$.

2.2. The paradigm

Our merging algorithms are based on the technique of blocking. This should not come as a surprise since previous sequential algorithms for in-place merging and parallel algorithms for merging are based on the same approach. The underlying paradigm in our algorithms is the one introduced by Shiloach and Vishkin [24]:

1. Divide the input sequences into blocks of $b=\lceil (m+n)/p \rceil$ elements and assign one processor per block.

2. Compute the cross-ranks for the blocks and their heads.

3. In parallel, merge the heading subblocks with their accompanying subblocks by a sequential algorithm.

4. Output the merged subblocks in the order given by the cross-ranks of the blocks.

Before moving on to an EREW PRAM implementation of the paradigm, we observe that the subblocks that are merged in Step 3 are, by definition, parts of the original blocks. Hence, the total length of a heading subblock and its accompanying subblock does not exceed $2b$.

In the next subsection we assume that $\Theta(n)$ extra space is available.

2.3. EREW PRAM implementation

Step 1 is implemented by broadcasting of the parameters m, n, and b to all processors. From these numbers and its own processor number each processor can compute the block it takes care of in constant time. The broadcasting can be done in $O(\log p)$ time on an EREW PRAM [1].

Were concurrent reads allowed, Step 2 could be accomplished in $O(\log n)$ time by p binary searches. This would give the cross-ranks of the heads, from which the cross-ranks of the blocks are readily obtained. To avoid concurrent reading, we proceed more cautiously in three steps:

2. Compute the cross-ranks for the blocks and their heads:

(*a*) Determine all cross-blocks.

(*b*) For every block, make a copy of its cross-block.

(*c*) For every block, find the cross-rank of its head inside its cross-block.

In Step $2a$ we first copy the heads of all the blocks of both sequences into new arrays, X' and Y'. The original positions of the heads are recorded. Then X' and Y' are merged using Batcher's odd-even merging algorithm [3], a brief description of which is outlined after the remaining substeps of Step 2 have been spelled out. Given the merged array of the heads, the cross-blocks are easily determined: if the head of block $X_i (Y_j)$ ends up in position k in the merged array, its cross-block is $Y_{k-i} (X_{k-j})$.

After Step $2a$ each processor knows which cross-block it wants to search. Since several blocks can have the same cross-block we make one copy of it for each processor that aims to search it. We have p processors, divided into consecutive groups, and processors within each group request a copy of the same block. Each processor first detects whether it starts or ends its group by checking the cross-blocks of its neighbors. Then the identities of the first and last processor in each group are communicated to the other processors within the group by a group prefix, or summing by groups, computation [23], which takes $O(\log p)$ time. Thereafter, each processor computes its relative position within the group.

We have thus reduced the problem to that of making p_i copies of b elements, using p_i processors, where $\sum p_i = p$. For $b = 1$, this is an instance of the parallel prefix problem (where the first element is equal to the element that is to be copied and the other elements are zeroes), which can be solved in $O(\log p_i)$ time using only $p_i/\log p_i$ processors on an EREW PRAM [8]. Employing all the p_i processors, we can thus copy $\log p_i$ elements simultaneously. Hence, our copying problem can be solved in time $O(\lceil b/\log p_i \rceil \cdot \log p_i)$, which is $O(b + \log p_i)$, for each group of processors. We conclude that Step $2b$ can be performed in $O(b + \log p)$ time, which is $O(b + \log n)$. Observe that this solution requires $O(b)$ extra storage per processor, which sums up to $O(n)$.

Now Step $2c$ can be carried out without read conflicts by letting each processor seach for the cross-rank of its head sequentially within its own copy of its cross-block, which takes $O(b)$ time.

To complete the description of Step 2, let us sketch Batcher's odd-even merging algorithm. This algorithm recursively merges the elements located at odd positions in X' with those located at odd positions in Y'. Similarly, the subsequences consisting of elements located at even positions are merged recursively. The output sequence is then obtained by shuffling the sequences returned by the recursive calls and possibly swapping pairs of adjacent elements. Since there is one processor per block, the merge can be performed in $O(\log p)$ time on an EREW PRAM. The reader should observe that Batcher's odd-even merging can be implemented stably [19]. Moreover, only linear, that is, $O(p)$, extra space is required.

To summarize, Step 2 takes $O(\log p + b + \log n)$ time, which is $O(b + \log n)$.

In the following the X-blocks and the Y-blocks are handled analogously, and so, for simplicity, we concentrate on the X-blocks. Suppose Y_j is the cross-block of X_i. Before starting the sequential merges in Step 3 it remains to determine the ends of the heading subblock of X_i and its accompanying subblock. We distinguish two cases, which are illustrated in Figure 2.

Case 1: X_i and X_{i+1} have the same cross-block.

Then the heading subblock of X_i is the entire block X_i, and the end of X_i's accompanying subblock is determined by the cross-rank of the head of X_{i+1}. Note that no concurrent reads occur.

Case 2: X_i and X_{i+1} have different cross-blocks.

The end of the heading subblock can now be determined from the cross-rank of the head of Y_{j+1}. In this case, the end of the accompanying subblock coincides with the tail of X_i's cross-block Y_j. As in the previous case, no concurrent reads are required.

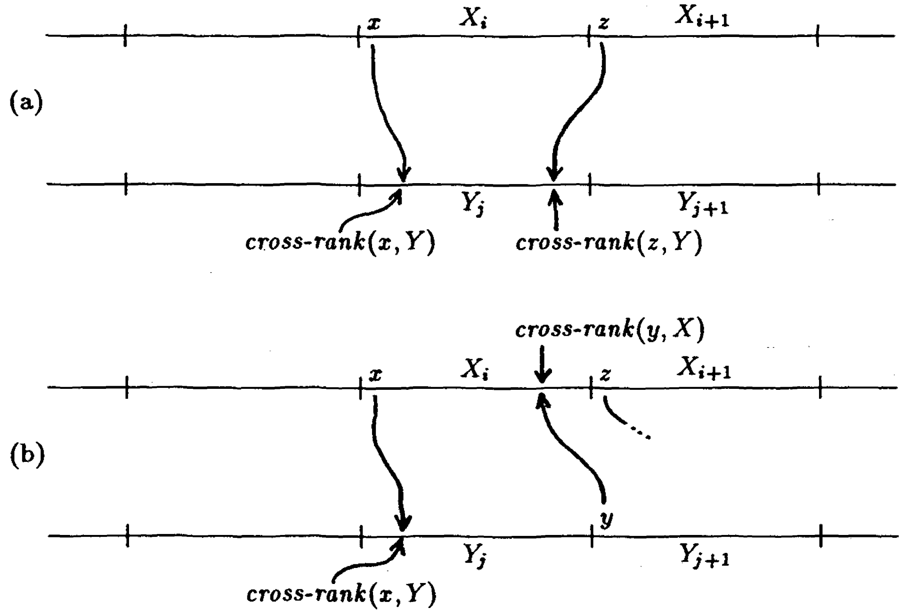

Figure 2. — Illustration of (a) Case 1 and (b) Case 2.

Given the boundaries of the subblocks to be merged, Step 3 is carried out in $O(b)$ time, by letting each processor merge its heading subblock and its accompanying subblock using any linear sequential merging algorithm.

Finally, the results of the sequential merges are assigned into a new array, as follows. Let x_h be the head of X_i. Then the merged subsequence is written starting from location $(i-1)b+1+\textit{cross-rank}(x_h, Y)$ and onwards, which takes constant time per element.

By the above discussion, the four steps of the algorithm run in time $O(\log p)$, $O(b+\log n)$, $O(b)$, and $O(b)$, respectively, which totals $O(n/p+\log n)$.

3. COST-SPACE OPTIMAL EREW PRAM IMPLEMENTATION

The space-consuming parts of the algorithm described in the previous section are the copying in Step 2 b and the reporting of the output in Step 4. In this section we show how both these steps can be implemented using only $O(p)$ extra space. That is, we prove Theorem 1.

3.1. Cross-block copying

The reason for copying was to avoid concurrent reads when searching for the cross-ranks of heads within their cross-blocks. As we saw in the previous section, taking one copy per processor is too space consuming, requiring $O(n)$ extra space. We show that a smaller number of copies is indeed sufficient. Each group of processors having the same cross-block is divided into subgroups, and only a single copy is taken for each subgroup.

After having computed the cardinality of each group of processors and communicated the appropriate information within the groups, as described in Section 2.3, the groups are divided into subgroups of b consecutive processors each. The first member of each subgroup, called the *leader*, checks whether its subgroup is full, that is, whether it contains b processors. If this is the case, the leader marks itself. Then we perform the cross-block copying, given in Section 2.3, for the marked leaders only. Note that, the number of copies taken is $O(p/b)$, and hence, the total amount of extra space used is $O(p)$. Since the division into subgroups and leader finding in the subgroups can be done in $O(\log p)$ time, it follows that the entire copying runs in $O(b+\log n)$ time.

Those subgroup leaders that were marked now possess their own copy of their cross-block, and each one has $b-1$ other subgroup members following it in sorted order by their heads. Leaders that were not marked will make use of the original cross-blocks directly. Next, each subgroup leader finds the cross-ranks for the heads in its subgroup by scanning its cross-block sequentially. That is, it merges its cross-block and the sorted sequence defined by the heads of the blocks in its subgroup, but rather than moving any elements, it just determines the cross-ranks of the heads. This step is easily done in $O(b)$ time.

We conclude that Step 2 can be implemented in $O(n/p+\log n)$ time and $O(p)$ extra space on an EREW PRAM.

3.2. Producing the output

So far we have assumed that there is a separate array available for writing the output. We next show how to produce the resulting sequence within the input array A. The main idea is to move the elements towards their correct positions stepwise, and delay the sequential merges. The paradigm in Section 2.2 is modified as follows:

3. Permute the blocks such that the heads form a non-decreasing sequence.
4. Move the accompanying X-subblocks beside their heading Y-subblocks.
5. In parallel, merge each heading Y-subblock with its accompanying X-subblock by a sequential algorithm.
6. Move the accompanying Y-subblocks beside their heading X-subblocks.
7. In parallel, merge each heading X-subblock with its accompanying Y-subblock by a sequential algorithm.

Step 3 is trivial if all blocks are of equal size b. At time step j, $1 \leqq j \leqq b$, each processor moves the j-th element in its block, where the destination is easily calculated from its own block number and its cross-block (which is known after Step 2). If the last blocks of the input sequences have less than b elements, we have to be careful so that we do not overwrite any elements. To facilitate the moves each processor should know the size and the head of the non-full blocks, which are broadcasted in $O(\log p)$ time. Basically, we have two cases. (See Figure 3.) Let x_h be the head of the last X-block and y_h that of the last Y-block.

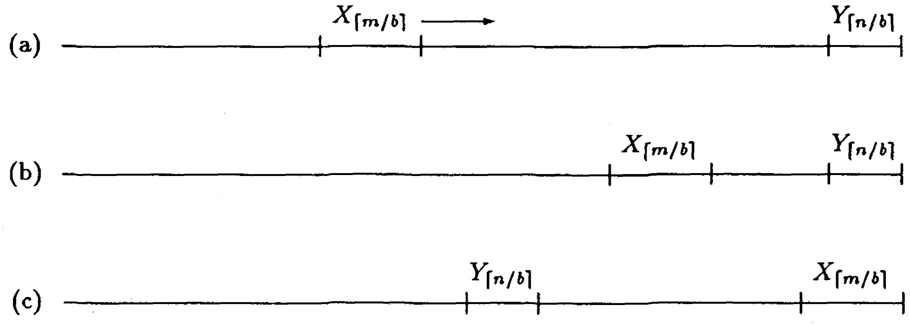

Figure 3. – Illustration of the block permutation: (a) input, (b) Case 1, and (c) Case 2.

Case 1: $x_h \leqq y_h$.

The blocks are permuted in two phases. First, $\mathrm{X}_{|m/b|}$ is moved to its correct position; that is, the Y-blocks with a head smaller than x_h are moved over it. (The Y-blocks with a head greater than x_h remain untouched.) This takes $O(b)$ time. Second, the blocks to the left of $\mathrm{X}_{\lceil m/b \rceil}$, which are all of size b, are permuted as described above.

Case 2: $x_h > y_h$.

Again, we permute the blocks in two phases. First, all the X-blocks with a head greater than y_h are moved over $Y_{\lceil n/b \rceil}$, which brings $Y_{\lceil n/b \rceil}$ into its correct position. Second, the equal-sized blocks located to the left of $Y_{\lceil n/b \rceil}$ are permuted as before.

Hence, the blocks can be permuted in $O(b + \log p)$ time and constant extra space per processor.

Let $X_i\, Y_j\, Y_{j+1} \ldots Y_{j+k}\, X_{i+1}$ be a series of blocks in the sequence obtained after Step 3. Then all the Y-blocks have their accompanying subblocks in X_i. In Step 4 we move these X-subblocks such that each of the Y-blocks is followed by its accompanying subblock. (See Figure 4).

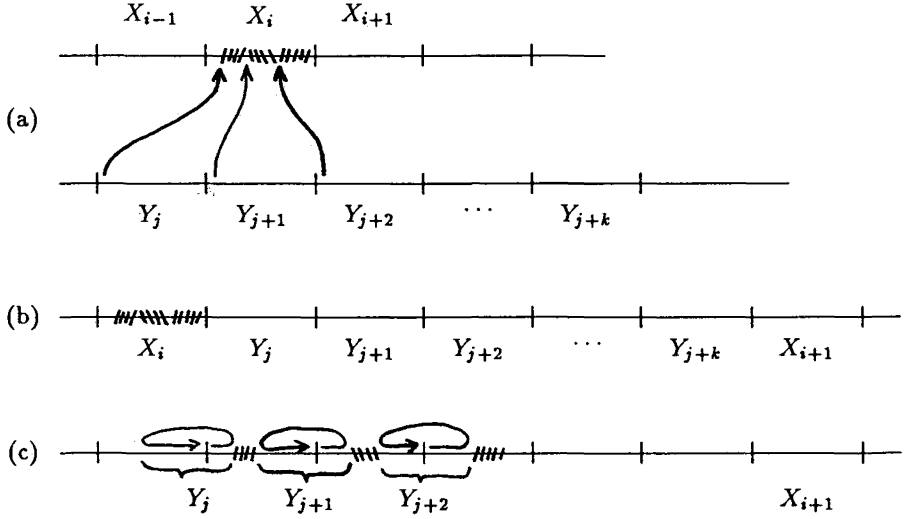

Figure 4. – Moving the accompanying X-subblocks beside their heading Y-subblocks. (a) the cross-ranks; (b) after Step 3; and (c) after Step 4.

After Step 2 each processor knows its cross-block and the cross-rank of its head. From these parameters each Y-processor can calculate how many positions to the left its block shall be moved in order to make room for its accompanying subblock (as well as for the accompanying subblocks of the Y-blocks preceding it). If the number of positions is greater than zero we say that the Y-processor is *active*. Moreover, an active processor is called *last-active* if its right neighbor is an X-processor or a non-active Y-processor. The blocks are moved as follows. At each time step j, $1 \leqq j \leqq b$, each last-active Y-processor reads the j-th last element x from its cross-block and stores this in its local memory. Then each active processor moves the j-th last element of its Y-block b positions leftwards. Finally, each last-active processor writes the element x in the location occupied by the just moved element in its own block.

When running the above procedure, the Y-blocks get split into two pieces U and V, where the elements in U are greater than those in V. Each processor should maintain pointers to the head and the tail of these pieces. The original order is then restored by swapping the pieces U and V, which can be done in-place in $O(b)$ time by first reversing U and V separately and then reversing their concatenation. This concludes the description of Step 4.

Step 5 is performed in $O(b)$ time and constant extra space per processor using any linear time in-place merging algorithm.

In Step 6 the merged subblocks are viewed as Y-subblocks, after which Steps 6 and 7 are analogous to Steps 4 and 5. Hence, Steps 4, 5, 6 and 7 take $O(b+\log p)$ time and $O(p)$ extra space. The discussion above and in the previous subsection concludes the proof of Theorem 1.

4. COST-SPACE OPTIMAL DCM IMPLEMENTATION

In this section we show that the EREW PRAM algorithm given in the previous section works on a DCM within the same resource bounds. That is, we prove Theorem 2. The easiest way of understanding this result is to regard a DCM as a special kind of an EREW PRAM, in which the shared memory is divided into p modules of b consecutive locations, and where each module may only be accessed by one processor at a time. We assume that the input sequences are initially partitioned into blocks of b elements (possibly except two blocks that are smaller than b), and distributed among the processors such that the i-th processor has the i-th block in its memory module.

It is readily seen that our applications of Batcher's merging, broadcasting, and prefix computation can be performed on a DCM as efficiently as on an EREW PRAM, since during these computations only one datum per memory module participates. The critical parts of the algorithm presented in Section 3 are those where several simultaneous accesses are made within the same block; that is, when copying cross-blocks in Step 2b and when performing the sequential merges in Steps 5 and 7. Let us therefore consider these steps in greater detail.

Consider a group of processors aiming to access the same cross-block. First, the first marked leader in the group copies the cross-block, letting the processors in its subgroup store one element each. Then the i-th processor, $1 \leqq i \leqq b$, in each full subgroup obtains a copy of the element stored by the i-th processor in the first subgroup by a broadcasting operation. Now, each full subgroup possesses its own copy of the cross-block – not stored by the leader of the subgroup as in the previous section, but scattered among the processors within the subgroup. Non-full subgroups will make use of the original cross-blocks. The cross-rank determination is then performed as explained in Section 3.1. Since the first copy of a cross-block is taken in $O(b)$ time and as the following broadcasting takes $O(\log p)$ time, the entire cross-block copying runs in $O(b+\log n)$ time. Moreover, note that each processor stores at most one extra element.

During the sequential merges it might happen that several processors attempt to access the same memory module simultaneously. This is, however, easy to avoid. Consider the merges performed in Step 5 (see Figure 4). No merging task spans more than $2b$ consecutive memory locations and only the last task in each Y-group can have less than b elements. It thus suffices to first perform every second merge, and when these are ready start the remaining merges.

Together with the analysis in the previous section, the above discussion concludes the proof of Theorem 2.

5. FAST COST-SPACE OPTIMAL CREW PRAM IMPLEMENTATION

It is interesting to observe that the presented techniques can be used to design a fast cost-space optimal CREW PRAM merging algorithm, which runs in time $O(n/p+\log\log m)$ and $O(1)$ extra space per processor (Theorem 3). The algorithm is a hybrid of the fast merging algorithm proposed by Borodin and Hopcroft [5, 16] and our cost-space optimal EREW PRAM algorithm. For the sake of clarity, we repeat the steps of the algorithm, although they can be extracted from the already presented ones.

1. Divide the input sequences into blocks of $b=\lceil (m+n)/p \rceil$ elements and assign one processor per block.
2. Compute the cross-ranks for the blocks and their heads.
3. Permute the blocks such that the heads from a non-decreasing sequence.
4. Move the accompanying X-subblocks beside their heading Y-subblocks.
5. In parallel, merge each heading Y-subblock with its accompanying X-subblock by a sequential algorithm.
6. Move the accompanying Y-subblocks beside their heading X-subblocks.
7. In parallel, merge each heading X-subblock with its accompanying Y-subblock by a sequential algorithm.

Step 1 takes constant time on a CREW PRAM.

In Step 2 we apply the Borodin-Hopcroft merging algorithm [5, 16] to merge the sequences X' and Y' defined by the heads. This algorithm first determines the cross-ranks of every $\sqrt{\lceil X' \rceil}$-th element of X' in Y' using a brute force algorithm. These cross-ranks divide Y' into $\sqrt{\lceil X' \rceil}$ parts, which are merged with their associated parts of X' recursively. Using a linear number of processors in the number of elements, the running time of the Borodin-Hopcroft merging algorithm is doubly-logarithmic in the length of the shortest input sequence, and it uses linear extra space in the number of elements. Hence, since we are only merging p elements, it runs in time $O(\log\log(m/b))$, which is $O(\log\log m)$, and $O(p)$ extra space. As described in Section 2.3, the cross-blocks are easily determined once the heads have been merged. Then each processor determines the cross-rank of its head by a sequential search in its cross-block. The sequential search takes $O(b)$ time, and thus, Step 2 requires $O(b+\log\log m)$ time and constant extra space per processor.

Recall the implementation of Steps 3, 4, 5, 6, and 7 in Section 3.2. The only phases consuming more than $O(b)$ time are those when broadcasting is needed. This takes constant time on a CREW PRAM, and thus, all these steps require $O(b)$ time and constant extra space per processor.

Note that the only part of the algorithm requiring non-trivial processor allocation is the application of the Borodin-Hopcroft merging algorithm, which indeed can be implemented on a CREW PRAM [5].

6. CONCLUSIONS

We have shown how merging can be done cost-space optimally on DCMs, EREW PRAMs, and CREW PRAMs. Furthermore, in the first two models our algorithms achieve the best possible time bounds with a finite number of processors. The latter one is the (asymptotically) fastest possible using $O(n \log^{\alpha} n)$, for any fixed constant α, processors. Moreover, the parallel merging algorithms are stable, provided the sequential in-place merging algorithm used as a subroutine is stable. We note that the merging algorithms naturally lead to cost-space optimal sorting algorithms.

ACKNOWLEDGEMENTS

The first author would like to thank Hristo Djidjev and Torben Hagerup for helpful discussions on the subject of this paper.

REFERENCES

1. S. G. Akl, *The Design and Analysis of Parallel Algorithms*, Prentice-Hall, Englewood Cliffs, N.J., 1989.
2. R. J. Anderson, E. W. Mayr and M. K. Warmuth, Parallel approximation algorithms for bin packing, *Inform. and Comput.*, *82*, 1989, pp. 262-277.
3. K. E. Batcher, Sorting networks and their applications, *Proc. of AFIPS Spring Joint Computer Conf.*, 1968, pp. 307-314.
4. G. Bilardi and A. Nicolau, Adaptive bitonic sorting: An optimal parallel algorithm for shared memory models, *SIAM J. Comput.*, *18*, 1989, pp. 216-228.
5. A. Borodin and J. E. Hopcroft, Routing, sorting, and merging on parallel models of computation, *J. Comput. System Sci.*, *30*, 1985, pp. 130-145.
6. R. Cole, Parallel merge sort, *SIAM J. Comput.*, *17*, 1988, pp. 770-785.
7. E. Dekel and I. Ozsvath, Parallel external merging, *J. Parallel Distr. Comput.*, *6*, 1989, pp. 623-635.
8. D. Eppstein and Z. Galil, Parallel algorithmic techniques for combinatorial computation, *Annual Reviews in Computer Science*, *3*, 1988, pp. 233-283.
9. X. Guan and M. A. Langston, Time-space optimal parallel merging and sorting, *IEEE Trans. Comput.*, *40*, 1991, pp. 596-602.
10. T. Hagerup and C. Rüb, Optimal merging and sorting on the EREW PRAM, *Inform. Process. Lett.*, *33*, 1989, pp. 181-185.
11. B.-C. Huang and M. A. Langston, Practical in-place merging, *Comm. ACM*, *31*, 1988, pp. 348-352.
12. B.-C. Huang and M. A. Langston, Fast stable merging and sorting in constant extra space, *Proc. Internat. Conf. on Computing and Information*, 1989, pp. 71-80.
13. J. JáJá, *An Introduction to Parallel Algorithms*, Addison-Wesley, Reading, MA, 1992.
14. R. M. Karp and V. Ramachandran, A survey of parallel algorithms for shared memory machines, In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. North-Holland, Amsterdam, The Netherlands, 1990.
15. M. A. Kronrod, An optimal ordering algorithm without a field of operation, *Dokladi Akademia Nauk SSSR*, *186*, 1969, pp. 1256-1258.

16. C. P. KRUSKAL, Searching, merging, and sorting in parallel computation, *IEEE Trans. Comput.*, *C-32*, 1983, pp. 942-946.
17. C. P. KRUSKAL, L. RUDOLPH and M. SNIR, A complexity theory of efficient parallel algorithms, *Theoret. Comput. Sci.*, *71*, 1990, pp. 95-132.
18. H. MANNILA and E. UKKONEN, A simple linear-time algorithm for in situ merging, *Inform. Process. Lett.*, *18*, 1984, pp. 203-208.
19. F. P. PREPARATA, New parallel-sorting schemes, *IEEE Trans. Comput.*, *C-27*, 1978, pp. 669-673.
20. M. J. QUINN, *Designing Efficient Algorithms for Parallel Computers*, North-Holland, New York, 1987.
21. J. SALOWE and W. STEIGER, Simplified stable merging tasks, *J. Algorithms*, *8*, 1987, pp. 557-571.
22. B. SCHIEBER and U. VISHKIN, Finding all nearest neighbors for convex polygons in parallel: A new lower bound technique and a matching algorithm, *Discrete Applied Math.*, *29*, 1990, pp. 97-111.
23. J. T. SCHWARTZ, Ultracomputers, *ACM Trans. Program. Lang. Syst.*, *2*, 1980, pp. 484-521.
24. Y. SHILOACH and U. VISHKIN, Finding the maximum, merging, and sorting in a parallel computational model, *J. Algorithms*, *12*, 1981, pp. 88-102.
25. M. SNIR, On parallel searching, *SIAM J. Comput.*, *15*, 1985, pp. 688-708.
26. H. S. STONE, Parallel processing with the perfect shuffle, *IEEE Trans. Comput.*, *C-20*, 1971, pp. 153-161.
27. L. G. VALIANT, General purpose parallel architectures, In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. North-Holland, Amsterdam, The Netherlands, 1990.