Fast Integer Merging on the EREW PRAM

Torben Hagerup Mirosław Kutyłowski

MPI–I–92–115 April 1992

Fast Integer Merging on the EREW PRAM

Torben Hagerup Mirosław Kutyłowski

MPI–I–92–115 April 1992

“Das diesem Bericht zugrunde liegende Vorhaben wurde mit Mitteln des Bundesministers für Forschung und Technologie (Betreuungskennzeichen ITR 9102) gefördert. Die Verantwortung für den Inhalt dieser Veröffentlichung liegt beim Autor.”

Fast Integer Merging on the EREW PRAM

Torben Hagerup*

Max–Planck–Institut für Informatik,
Im Stadtwald, D-W-6600 Saarbrücken, Germany.
Email: torben@mpi-sb.mpg.de

Mirosław Kutyłowski†

Fachbereich Mathematik–Informatik and Heinz–Nixdorf–Institut,
Universität–GH Paderborn, Postfach 1621, D-W-4790 Paderborn, Germany.
Email: mirekk@uni-paderborn.de

Abstract. We investigate the complexity of merging sequences of small integers on the EREW PRAM. Our most surprising result is that two sorted sequences of n bits each can be merged in $O(\log\log n)$ time. More generally, we describe an algorithm to merge two sorted sequences of n integers drawn from the set $\{0,\ldots,m-1\}$ in $O(\log\log n+\log m)$ time using an optimal number of processors. No sublogarithmic merging algorithm for this model of computation was previously known. The algorithm not only produces the merged sequence, but also computes the rank of each input element in the merged sequence. On the other hand, we show a lower bound of $\Omega(\log\min\{n,m\})$ on the time needed to merge two sorted sequences of length n each with elements in the set $\{0,\ldots,m-1\}$, implying that our merging algorithm is as fast as possible for $m=(\log n)^{\Omega(1)}$. If we impose an additional stability condition requiring the ranks of each input sequence to form an increasing sequence, then the time complexity of the problem becomes $\Theta(\log n)$, even for $m=2$. Stable merging is thus harder than nonstable merging.

1 Introduction

A PRAM is a synchronous parallel machine with a global memory accessible to all processors. The EREW (*exclusive-read exclusive-write*) PRAM is the weakest member of the PRAM family; it disallows concurrent access to the same memory cell by more than one processor, even for reading purposes. The resulting difficulties of interprocessor communication preclude superfast algorithms for even very simple problems: Lower time bounds of $\Omega(\log n)$ for the EREW PRAM were established by Cook, Dwork and Reischuk [7] for the problem of computing the OR of n bits (this bound holds even for the stronger CREW PRAM, which allows concurrent reading),

*Supported in part by the Deutsche Forschungsgemeinschaft, SFB 124, TP B2, VLSI Entwurfsmethoden und Parallelität, and in part by the ESPRIT II Basic Research Actions Program of the EC under contract No. 3075 (project ALCOM).

†Supported by Deutsche Forschungsgemeinschaft grant ME 872/1–4. Present address: Institute of Computer Science, University of Wrocław, Przesmyckiego 20, PL-51-151 Wrocław, Poland. Email: mirekk@plwruw11.bitnet

by Beame, Kik and Kutyłowski [1] for the problem of creating n copies of an input bit, and by Snir [11], under certain assumptions, for the problem of *searching*, i.e., of locating a value in a sorted table of size n. In fact, except for certain algorithms given by Snir [11] with running times down to $\Theta(\sqrt{\log n})$, the only EREW PRAM algorithms known to the authors that use $o(\log n)$ time on inputs of size n are either for trivial problems (e.g., compute the OR of n bits, at most one of which is a 1), or have the property that each output bit is determined by a small set of input bits that is very easy to compute (e.g., the ruling-set algorithm of Cole and Vishkin [6]). Against this background one might strongly suspect that merging two sorted sequences of length n each also requires $\Omega(\log n)$ time on the EREW PRAM. Indeed, since merging a sequence of length 1 with a sorted sequence of length n is surely no more difficult than merging two sorted sequences of length n each, such a result follows from the lower bound of Snir mentioned above. Note, however, that the lower bound for merging derived from Snir's result is unsatisfactory for two reasons. First, Snir places restrictions on the values that can be stored in memory cells and on the internal workings of individual processors. Second, his proof uses Ramsey theory and therefore breaks down if the values that can occur in the input sequences are drawn from a restricted domain. As shown by Snir, the restrictions placed on memory cells and processors are essential in the sense that without them, the searching problem can be solved faster than in $\Theta(\log n)$ time. The same can be seen to hold if the entries in the table to be searched are drawn from a sufficiently small domain, but neither effect is apparent for the more difficult problem of merging. We here answer the question of whether sublogarithmic merging on the EREW PRAM is possible in some circumstances and explore the fine boundary between what can and what cannot be done. Since the results obtained turn out to be very sensitive to the exact definition of merging, we back off for a while to provide such definitions.

A sequence $(x_1, \ldots, x_n)$ of integers is called *sorted* if $x_1 \leq \cdots \leq x_n$. Given two sorted sequences X and Y of integers, we denote by $X \cup Y$ the sorted sequence of integers that contains as many occurrences of i, for each integer i, as X and Y put together. We consider several different merging problems. Common to all of these is that the input consists of two sorted sequences $X = (x_1, \ldots, x_n)$ and $X' = (x_{n+1}, \ldots, x_{2n})$ with elements drawn from a set $\{0, \ldots, m-1\}$, where n and m are positive integers known to all processors; without loss of generality we always assume that $n \geq 4$ and $m \geq 2$. The *value-merging* problem simply calls for the construction of the sequence $X \cup X'$. *Rank-merging*, on the other hand, provides a description of how to construct $X \cup X'$ from the input sequences. More precisely, the rank-merging problem is to compute a permutation r of $\{1, \ldots, 2n\}$, called a *rank vector* and represented by the sequence $(r(1), \ldots, r(2n))$, such that $x_{r^{-1}(1)} \leq x_{r^{-1}(2)} \leq \cdots \leq x_{r^{-1}(2n)}$. When wanting to be explicit about the parameters n and m, we speak of (n, m)-value-merging and (n, m)-rank-merging. Furthermore, when the meaning is clear from the context, we will omit the qualifiers "value" and "rank".

A procedure for (n, m)-rank-merging clearly implies a procedure for (n, m)-value-merging with the same resource bounds (up to constant factors). Value-merging at first glance might seem more natural than rank-merging. The latter is needed, however, whenever the actual objects to be merged are records containing other information in addition to the integer keys according to which the records are to be merged, since value-merging the keys alone does not allow the full records to be put in order. The definition of rank-merging given above is precise, but often cumbersome. In most proofs to follow, we therefore employ a less precise but more convenient informal terminology that we now introduce. The key point is to consider the input elements not as integers, of which several may be identical, but as pairwise distinct objects with several attributes. An input element x_i has a *value*, the integer x_i, and an *original position*, i, and the goal of rank-merging is to mark each input element with a *rank* from the set $\{1, \ldots, 2n\}$ such that distinct input elements receive distinct ranks and smaller elements receive smaller ranks. Equivalently, the goal is to store the input element of rank i in position i of an output sequence of length $2n$, for $i = 1, \ldots, 2n$. In the formal definition, the rank of x_i of course corresponds to $r(i)$.

Consider the easiest among the merging problems introduced above, that of $(n,2)$-value-merging, i.e., given two sorted sequences X and X' of n bits each, construct the sorted sequence of $2n$ bits that contains exactly as many 0's as X and X' put together. Despite its seeming simplicity, this is a fascinating problem whose complexity is still unresolved. Observe first that it is easy to solve the problem in constant time on a CREW PRAM. If a processor is associated with each bit in X and each such processor inspects its own bit and the bit following it, if any, exactly one processor, detecting the change from 0 to 1 in X, will know the number of 0's in X. The number of 0's in X' can be determined in the same way, and the sum of the two counts can be broadcast to each of $2n$ processors, each of which can then easily produce one output bit. Since only the broadcasting part of this algorithm uses concurrent reading, it is also evident that even an EREW PRAM can compute any desired bit of the output in constant time. Computing all output bits faster than in $\Theta(\log n)$ time, however, is a challenging problem, and both authors expended considerable energy trying to prove that it cannot be done. The lower bound of Beame, Kik and Kutyłowski cited above shows that any EREW PRAM algorithm that uses the same general approach as the above CREW PRAM algorithm is doomed to fail, since broadcasting the total number of 0's to sufficiently many processors will require $\Omega(\log n)$ time.

Our main result is that not only $(n,2)$-value-merging, but in fact $(n,2)$-rank-merging can be done in $O(\log\log n)$ time on an EREW PRAM. An interesting feature of the algorithm is that, intuitively speaking, a significant portion of the computation is carried out by processors operating on incorrect data. A major concern in the design of the algorithm was to prevent such processors from interfering with the useful part of the computation, e.g., by causing concurrent reading or writing. We also extend the algorithm to solve the general (n,m)-rank-merging problem, in which case the running time becomes $O(\log\log n + \log m)$, and reduce the number of processors used to obtain an algorithm that executes $O(n)$ operations, which is optimal. On the other hand, we prove that (n,m)-value-merging requires $\Omega(\log \min\{n,m\})$ time, thereby showing our algorithm to be as fast as possible for $m = (\log n)^{\Omega(1)}$ (assume that we switch to a standard $O(\log n)$-time algorithm for $m \geq n$). Our discovery that integer merging is easier than general merging on the EREW PRAM to some extent parallels what is known for the CREW PRAM. General merging can be done in $O(\log\log n)$ time on the CREW PRAM with an optimal number of processors [9] and there is a corresponding lower bound [4], whereas an algorithm that runs in $O(\log\log\log m)$ time using an optimal number of processors is known for the case of input numbers drawn from the set $\{0, \dots, m-1\}$ [2].

It is frequently desirable to impose an additional restriction on the rank vector r obtained by merging two sorted sequences X and X' of length n each. Requiring that $r(1) < \cdots < r(n)$ and $r(n+1) < \cdots < r(2n)$ (informally, the relative order of the elements in X and X' is to be preserved), we obtain the problem of *stable (rank-) merging*. Stable merging clearly is no easier than unadorned rank-merging, which we will also call *unstable merging*. Our fast algorithm solves the unstable merging problem only and cannot be extended to stable merging. In fact, we prove a lower time bound of $\Omega(\log n)$ for stable $(n, 2)$-merging. Stable merging is hence more difficult than unstable merging. Although such a result might seem natural for randomized algorithms (compare, e.g., the integer sorting algorithm of Rajasekaran and Reif [10]), in our deterministic setting it is somewhat unexpected.

2 Lower bounds

In this section we first show that (n, m)-value-merging needs $\Omega(\log \min\{n, m\})$ time on the EREW PRAM. We then give an $\Omega(\log n)$ lower time bound for stable merging. All of our lower bounds allow any number of processors and memory cells, arbitrary local computation, the storage of arbitrary values in memory cells, and nonuniformity. They are derived from the following result, proved by Beame, Kik and Kutyłowski [1] and valid under the same assumptions.

Lemma 2.1 *Every EREW PRAM algorithm that takes as input a single bit x and stores the value of x in n distinct fixed memory cells has a running time of $\Omega(\log n)$.*

Corollary 2.2 *In order to show that an EREW PRAM algorithm runs in $\Omega(\log t)$ time, it suffices to exhibit two inputs that differ in just one component (i.e., the contents of exactly one input cell differ), but whose associated outputs, as computed by the algorithm, differ in t components.*

We first use Corollary 2.2 to show the lower bound on (n, m)-value-merging.

Theorem 2.3 *(n, m)-value-merging requires $\Omega(\log \min\{n, m\})$ time on the EREW PRAM.*

Proof: Let $t = \min\{n, m\}$ and consider the n-element sequences $X = (1, 2, \dots, t-2, t-1, t-1, \dots, t-1)$, $Y_1 = (0, t-1, t-1, \dots, t-1)$ and $Y_2 = (t-1, t-1, \dots, t-1)$. Viewed as input to a merging algorithm, (X, Y_1) and (X, Y_2) differ in exactly one component, but $X \cup Y_1 = (0, 1, \dots, t-3, t-2, t-1, \dots, t-1)$ and $X \cup Y_2 = (1, 2, \dots, t-2, t-1, t-1, \dots, t-1)$ differ in their first $t-1$ components. The theorem now follows from Corollary 2.2. $\square$

Section 4 presents an (n, m)-rank-merging algorithm that runs in $O(\log\log n + \log m)$ time. By Theorem 2.3, this is as fast as possible for $m = (\log n)^{\Omega(1)}$. For $m = (\log n)^{o(1)}$ the complexity of the problem is unknown.

In the remainder of this section we consider stable $(n, 2)$-merging. We first show a lower time bound of $\Omega(\log n)$ for the *strictly stable merging* of two sorted sequences $X = (x_1, \dots, x_n)$ and $X' = (x_{n+1}, \dots, x_{2n})$, which we define to be the problem of computing a permutation r that satisfies the following condition in addition to those of stable merging: $r(i) < r(j)$ for all integers $i$ and $j$ with $1 \le i \le n < j \le 2n$ such that $x_i = x_j$ (i.e., in the case of ties, elements in X are to be considered smaller). Since this result is subsumed by our next lower bound and mostly serves as a warm-up to the latter, we shall not formulate it as a theorem.

Suppose that an algorithm for strictly stable merging maps the input sequences $X_0 = (0, 0, \ldots, 0)$ and $X' = (0, 0, \ldots, 0)$ to the rank vector r_0, and the input sequences $X_1 = (0, 0, \ldots, 1)$ and X' to the rank vector r_1. Then $(r_0(n+1), \ldots, r_0(2n)) = (n+1, n+2, \ldots, 2n)$, but $(r_1(n+1), \ldots, r_1(2n)) = (n, n+1, \ldots, 2n-1)$. The desired lower bound again follows from Corollary 2.2.

As announced above, we can extend the last result to (not necessarily strict) stable merging. Note how the above proof breaks down in the absence of strict stability.

Theorem 2.4 *Stable $(n, 2)$-merging requires $\Omega(\log n)$ time on the EREW PRAM.*

Proof: Consider $2n+1$ inputs $(X_0, Y_0), \ldots, (X_{2n}, Y_{2n})$ defined as follows: For $i = 0, \ldots, n$, X_i consists of $n-i$ zeros followed by $i$ ones, and Y_i consists of n ones. For $i = 0, \ldots, n$, X_{n+i} consists of n ones, and Y_{n+i} consists of $i$ zeros followed by $n-i$ ones. Given a stable merging algorithm, let r_i be the rank vector computed by the algorithm on the input sequences (X_i, Y_i), for $i = 0, \ldots, 2n$, and let $\Phi_i = \sum_{j=1}^{n} r_i(j)$, that is, Φ_i is the sum of the ranks of the elements of X_i in $X_i \cup Y_i$. In order to gain intuition for the remainder of the proof, consider an experiment consisting of $2n$ steps. In the $i$th step, for $i = 1, \ldots, 2n$, we replace (X_{i-1}, Y_{i-1}) by (X_i, Y_i) as input to the merging algorithm and observe the resulting change in Φ_i.

Since $\Phi_0 = \sum_{j=1}^{n} j$ and $\Phi_{2n} = \sum_{j=1}^{n}(n+j)$, we have $\Phi_{2n} - \Phi_0 = n^2$. It follows that for some i_0 with $1 \le i_0 \le 2n$, $\Phi_{i_0} - \Phi_{i_0-1} \ge n/2$. Let $t = \sqrt{n/2}$ and consider two possibilities:

Case 1: $\max_{1 \le j \le n}(r_{i_0}(j) - r_{i_0-1}(j)) \le t$. In this case at least t elements of X_i change their rank in step i_0 of the experiment, i.e., $(r_{i_0}(1), \ldots, r_{i_0}(n))$ and $(r_{i_0-1}(1), \ldots, r_{i_0-1}(n))$ differ in at least t components. Since the input sequences (X_{i_0}, Y_{i_0}) and (X_{i_0-1}, Y_{i_0-1}) differ in exactly one component, the lower bound now follows from Corollary 2.2 and the fact that $\log t = \Omega(\log n)$.

Case 2: *For some $j$ with $1 \le j \le n$, $r_{i_0}(j) > r_{i_0-1}(j) + t$.* Note that in a stable merging of two sequences X and Y, the rank of the $j$th element of X is precisely $j$ plus the number of elements in Y of smaller rank. Hence the element of X_i that increases its rank by more than t in step i_0 of the experiment changes its relative order with respect to more than t elements of Y_i. By another application of the same principle, each of these more than t elements of Y_i changes its rank in the same step of the experiment. The lower bound now follows as in Case 1. □

If even the requirement of stability is relinquished, merging becomes substantially easier, as demonstrated in the following sections.

3 Bitonic ranking

For $k \in I\!N$ and $b_1, b_2, \ldots, b_k \in \{0, 1\}$, we call a sequence of 0's and 1's a $b_1 b_2 \cdots b_k$-*sequence* if it belongs to the set described by the regular expression $b_1^* b_2^* \cdots b_k^*$, i.e., if it consists of zero or more occurrences of b_1, followed by zero or more occurrences of b_2, etc. 010- and 101-sequences are collectively said to be *bitonic*. In this section we describe an algorithm for *bitonic ranking*, i.e., given a bitonic sequence $(x_1, \ldots, x_n)$, compute a permutation r of $\{1, \ldots, n\}$ such that $x_{r^{-1}(1)} \leq \cdots \leq x_{r^{-1}(n)}$. Note, as before, that ranking can be used to sort, i.e., to actually arrange the input elements in sorted order, and that the opposite is true provided that each element is marked with its original position. Since the concatenation of a sorted bit sequence with the reverse of another such sequence is bitonic, rank-merging of sorted bit sequences reduces to bitonic ranking.

Theorem 3.1 *When $n \geq 4$ is a power of 2, a bitonic sequence of length n can be ranked in $O(\log\log n)$ time on an EREW PRAM using $O(n)$ processors and $O(n)$ space.*

Proof: We show Theorem 3.1 by induction on n, i.e., by giving a recursive algorithm that performs as stated. It turns out to be necessary for the inductive argument to make an additional assumption about the rank vector r computed by the recursive algorithm. Recall that every permutation can be written as a *product of cycles*, i.e., as the composition of a number of cyclic permutations. We inductively assume that r has a particularly simple cycle structure, namely that it can be written as a product of *disjoint transpositions*, i.e., of cyclic permutations of disjoint sets of size 2.

Given a bitonic input sequence of length n, where $n \geq 4$ is a power of 2, let $s = 2^{\lfloor \log n/2 \rfloor}$ and $t = n/s$ and consider the input sequence to be stored in an $s \times t$ *tableau* $\mathcal{A}$ in column-major order, i.e., the first s elements in the sequence are stored, from top to bottom, in the first column of the tableau, the next s elements are stored in the next column, etc. Number the rows and columns of all tableaus consecutively starting at 1. It is easy to see that each row of $\mathcal{A}$ is bitonic, so that the rows of $\mathcal{A}$ can be sorted by recursive applications of the algorithm (or trivially, if $t = 2$). The resulting tableau $\mathcal{B}$ has at most one column containing distinct values (i.e., both 0's and 1's). In the following assume that $\mathcal{B}$ has such a column, called the *critical column*; it will be easy to see that the case in which $\mathcal{B}$ has no critical column is handled correctly.

Since the critical column of $\mathcal{B}$ is bitonic, the most natural way to solve the original problem would be first to sort the rows of $\mathcal{A}$ to obtain $\mathcal{B}$, and then to sort the critical column of $\mathcal{B}$, which results in a sorted sequence (stored in column-major order). However, this would give a recurrence relation for the running time $T(n)$ of the algorithm on input of size n of the approximate form $T(n) = 2T(\sqrt{n}) + \Omega(1)$, which solves to $T(n) = \Omega(\log n)$. Since this is clearly too slow, we must proceed differently. Our approach is to sort the critical column of $\mathcal{B}$ *before* $\mathcal{B}$ has been constructed, i.e., while the rows of $\mathcal{A}$ are being sorted! In this way we obtain an algorithm that uses only constant time in addition to a number of recursive calls on arguments of size $O(\sqrt{n})$, all of which can be started simultaneously; the running time is $O(\log\log n)$, as desired.

We cannot actually compute the critical column of $\mathcal{B}$ and store it in a fixed place (this would require $\Omega(\log n)$ time). What we can do, however, is to construct two candidate sequences, one of which will equal the critical column C of $\mathcal{B}$. Note carefully that we take C to be simply a sequence of 0's and 1's; the additional information associated with the elements in the critical column of $\mathcal{B}$, such as their original positions, is not available. Begin by constructing the sequence $D = (d_1, \dots, d_s)$, where d_i is the number of 0's in the ith row of $\mathcal{A}$ (or $\mathcal{B}$), for $i = 1, \dots, s$. Computing d_i is easy since the ith row of $\mathcal{A}$ is bitonic: In the sequence of values there is at most one change from 0 to 1 and one change from 1 to 0, and d_i is a simple function of the positions of these changes, which can be determined in constant time by letting each pair of consecutive elements be inspected by a processor. If d is the index of the critical column, then for each $i \in \{1, \dots, s\}$ we have $d = d_i$ or $d = d_i + 1$. Hence if the parity of d were known, a processor P_i associated with the ith element of D could determine the exact value of d and hence the ith element of C. The parity of d being unknown, $P_1, \dots, P_s$ instead construct an "even candidate" $E = (e_1, \dots, e_s)$, equal to C if d is even, and an "odd candidate" $U = (u_1, \dots, u_s)$, equal to C if d is odd (see Fig. 1). In precise terms, if d_i is even, then P_i sets $e_i := 0$ and $u_i := 1$; otherwise it sets $e_i := 1$ and $u_i := 0$. Observe that E and U are both bitonic.

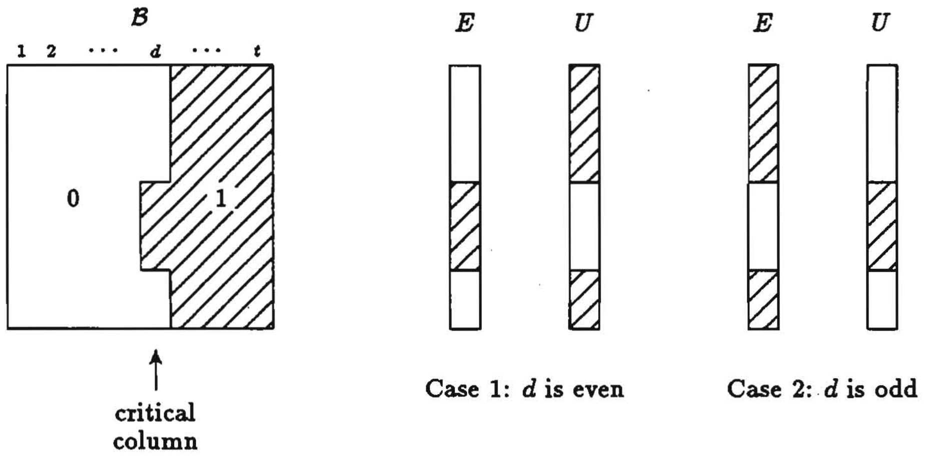

Fig. 1. The candidate critical columns E and U.

Recursively rank both E and U and recall that this produces permutations r_E and r_U of $\{1, \dots, s\}$ that sort E and U, respectively. Our remaining task is, using constant time after the construction of $\mathcal{B}$, to permute the elements in the critical column of $\mathcal{B}$, now marked with their original positions, according to the proper one of r_E and r_U (i.e., r_E if d is even, and r_U if d is odd). The problem again is that the parity of d is unknown. We might therefore apply both r_E and r_U, arguing that permuting elements in columns other than the critical column does no harm. More precisely, let P_i move the elements in position i of its candidate critical columns in $\mathcal{B}$, i.e., those indexed by d_i and $d_i + 1$ (introduce dummy columns numbered 0 and $t + 1$), according to r_E and r_U, with r_E operating in the column of even index and r_U operating in the column of odd index. However, while this applies one of the permutations r_E and r_U correctly in column d, the other permutation may be applied partly in column $d - 1$ and partly in column $d + 1$. This is because not all processors have the same two candidate critical columns, and it does not lead to the desired result (some positions in the tableau may afterwards hold either 0 or

2 elements). In order to counter this problem, we modify the procedure so that it moves an element to a new position only if the element in that position before the move has a different value (e.g., a 0 may be moved to a position previously holding a 1, but not to one previously holding a 0). This clearly prevents movement of elements outside of the critical column, as desired. Because r_E and r_U are products of disjoint transpositions, according to the inductive hypothesis, the modified procedure still has the effect of permuting the elements within the critical column – we have simply cancelled all interchanges of 0's with 0's and of 1's with 1's). Since the critical column is sorted and no other column is modified, the resulting tableau is sorted.

At this point we have obtained a permutation r that sorts the input sequence. Because an element may be moved both by a transposition within a row and subsequently by a transposition within the critical column, however, r is not necessarily a product of disjoint transpositions. In order to satisfy the inductive assumption, a final constant-time computation described below replaces r by a product r' of disjoint transpositions that is equivalent to r, i.e., results in the same distribution of 0's and 1's.

Let G be an undirected graph with n vertices arranged in an $s \times t$ array and with each vertex corresponding in the natural way to a tableau position. There is an edge between two vertices if and only if elements in the corresponding positions are interchanged during the sorting procedure described above. G has "horizontal" edges, corresponding to interchanges within rows during the construction of B, and "vertical" edges, corresponding to interchanges within the critical column. Since only vertices in the critical column have incident vertical edges and no vertex is incident to more than one horizontal edge or to more than one vertical edge, it is easy to see that no connected component of G contains more than 4 vertices. Call a vertex of G a 01-vertex if the corresponding position initially contains a 0, but after the application of r contains a 1, and define a 10-vertex analogously. Since all "interaction" takes place within connected components of G, the number of 01-vertices within a connected component equals the number of 10-vertices. Now construct a new graph G' from G as follows: Within each connected component of G, replace all edges by new edges that form a perfect matching of the 01-vertices with the 10-vertices. G' in a natural way corresponds to a product of disjoint transpositions, which clearly is equivalent to r and can be taken as r'. Since connected components are treated independently and each component represents a problem of constant size, the construction of G' and r' from G can be carried out in constant time on an EREW PRAM if only we can associate a unique processor with each connected component of G. But since each nontrivial component contains exactly one vertical edge, this is easy: Associate a processor with each vertex of G and assign responsibility for a nontrivial component to the processor associated with the upper endpoint, say, of the vertical edge. This ends the description of the recursive algorithm, which can clearly be executed using $O(n)$ processors and $O(n)$ space. □

4 Optimal merging

In this section we extend the basic algorithm of the previous section in two directions: First we show how the algorithm can be modified to handle input numbers in the set

$\{0, \ldots, m-1\}$, for arbitrary $m \geq 2$. Then we reduce the number of processors used by the modified algorithm in order to obtain an algorithm with optimal speedup.

Lemma 4.1 *(n, m)-rank-merging problems can be solved on an EREW PRAM using $O(\log\log n + \log m)$ time, $O(nm)$ processors and $O(nm)$ space.*

Proof: Without loss of generality assume that $n \geq 4$ and that n is a power of 2 (to achieve this extend each input sequence by a suitable number of dummy elements larger than all other elements). Since nm processors can easily create m copies of the entire input in $O(\log m)$ time, it suffices to show how to compute the ranks of all occurrences of an arbitrary but fixed value $i \in \{0, \ldots, m-1\}$ using $O(\log\log n + \log m)$ time, n processors and $O(n)$ space in addition to a constant number of applications of the algorithm of Theorem 3.1 to inputs of size $O(n)$. This can be done as follows (see Fig. 2):

Given two sorted input sequences X and X' of length n each and with elements in the set $\{0, \ldots, m-1\}$, let Y be the concatenation of X with the reverse of X'. In order to compute ranks for all occurrences of some value $i \in \{0, \ldots, m-1\}$, first derive a 010-sequence V from Y by replacing each value $\leq i$ by 0 and each value $> i$ by 1. Derive another 010-sequence W from Y by replacing each value $< i$ by 0 and each value $\geq i$ by 1. Sort both V and W and form a new 010-sequence Z as the componentwise exclusive-or of the two resulting 01-sequences. For $j = 1, \ldots, 2n$, it is easy to see that Z contains a 1 in position j exactly if sorting Y places an i in position j, i.e., if $j \in R$, where R is the set of ranks to be distributed among the occurrences of i in Y. Marking each 1 in Z with its position in Z and subsequently sorting Z stores the elements of R in the last positions of a sequence S_R. In a similar fashion we store in the last positions of another sequence S_I (constructed as explained below) the elements of the set I of positions that contain an occurrence of i in the (unsorted) sequence Y. Matching corresponding positions in the two sequences S_I and S_R now defines a bijection between I and R, i.e., computes the desired ranks.

In order to compute S_I, first derive from each half of Y a 010-sequence by replacing each occurrence of i by 1 and each occurrence of a value different from i by 0. Mark each 1 in either half with its position in Y, sort the halves separately and concatenate the sorted first half with the reverse of the sorted second half. This creates a new 010-sequence, the sorting of which easily yields S_I. This ends the proof of Lemma 4.1. □

Before deriving an optimal algorithm from Lemma 4.1, we discuss the reasons why this is not as easy as it might seem. The standard way of merging optimally, given a nonoptimal subroutine for merging, is to begin by merging sequences of equally-spaced *representatives*. As a result of this computation, each representative knows its rank in the opposite sequence to within the distance between successive representatives. Each representative then determines its exact rank in the opposite sequence, typically by binary search in the relevant subsequence, which splits the original problem into a collection of small and independent subproblems that are easily solved in parallel. The binary search by the representatives is easy on the CREW PRAM, but on the EREW PRAM requires coordination between representatives. This coordination between representatives can be achieved by means of prefix

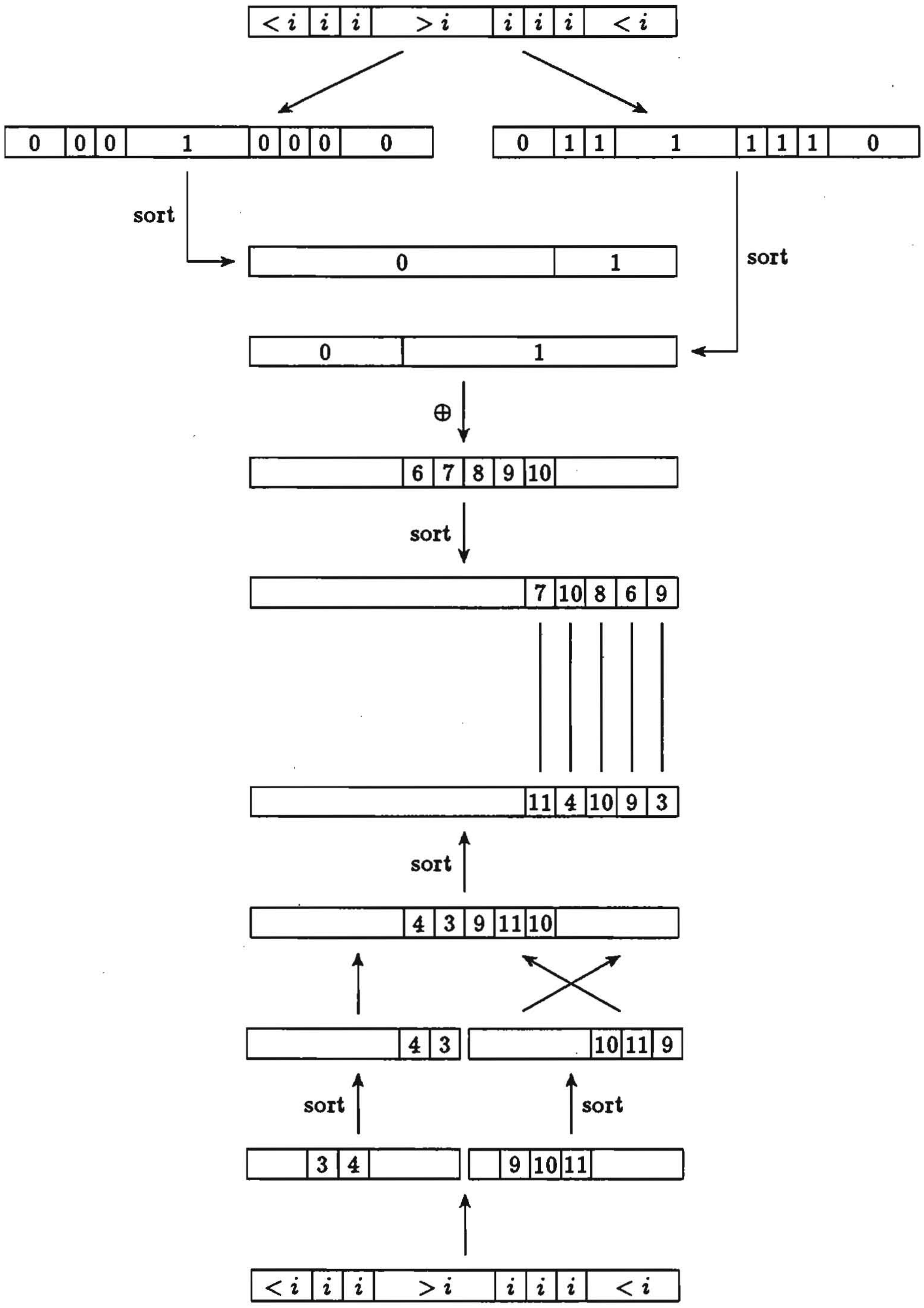

Fig. 2. Computing the ranks of all occurrences of i. Nonzero integers mark key values of 1.

summation [8]; in the present context, however, we do not have the time to carry out a prefix summation. While still merging sequences of representatives, we are therefore forced to proceed in an entirely different way.

Theorem 4.2 *Two sorted sequences of length n each and with elements drawn from $\{0,\ldots,m-1\}$ can be rank-merged on an EREW PRAM using $O(\log\log n + \log m)$ time, $O(n)$ operations and $O(n)$ space.*

Proof: As in the proof of Lemma 4.1 we can assume n to be a power of 2. We will also assume that $m \le n^{1/3}$. This is no restriction, since for $m > n^{1/3}$ the running time claimed is $O(\log n)$, so that general merging algorithms [3, 8] can be used. Choose $k \le n$ as a power of 2 with $k = \Theta(m \log n)$.

Let $(x_1,\ldots,x_n)$ and $(x_{n+1},\ldots,x_{2n})$ be two sorted input sequences with elements drawn from $\{0,\ldots,m-1\}$ and divide each of these into *blocks* of k consecutive elements. For $i = 1,\ldots,2n/k$, call x_{ik} the *representative* of the block containing the elements $x_{(i-1)k+1},\ldots,x_{ik}$, and call the value of x_{ik} the *header* of the block. Use the algorithm of Lemma 4.1 to merge the sequences $(x_k, x_{2k},\ldots,x_n)$ and $(x_{n+k}, x_{n+2k},\ldots,x_{2n})$ of representatives. This needs $O(\log\log n + \log m)$ time and $O((n/k)m(\log\log n + \log m)) = O(n)$ operations and marks each representative with a rank in the set $\{1,\ldots,2n/k\}$. Multiply each rank by k or, equivalently, place the representatives in sorted order in equally-spaced positions of an output array A of size $2n$. Broadcast the rank of each representative to its entire block, which takes $O(\log k) = O(\log\log n + \log m)$ time, and subsequently move the block with its representative, i.e., for $i = 1,\ldots,2n/k$, if the (new) rank of x_{ik} is jk, then move x_{ik-l} to $A[jk-l]$, for $l = 0,\ldots,k-1$.

If A were sorted, it would implicitly define the desired ranks. It is easy to see, however, that this is not generally the case. Our remaining task therefore is to sort A. We do this by identifying a small set of *critical* elements that are potentially "out of order", extracting these, sorting them and putting them back in their original positions (but in a different order), after which A will turn out to be sorted.

Define a block to be *heterogeneous* if it contains at least two distinct values, and *homogeneous* if it contains just one value. Label each heterogeneous block with 'L' if its elements came from the "left" sequence $(x_1,\ldots,x_n)$, and otherwise with 'R'. For $i = 0,\ldots,m-1$, also label a block with '1' or '2' if it is the first or second homogeneous block in A, respectively, with header i. The labeling is easy to carry out using $O(\log k)$ time and $O(n)$ operations. Note that no two distinct blocks can have the same label *and* the same header. Define an element to be *critical* exactly if it belongs to a labeled block. Since there are only 4 distinct labels, m distinct headers and k elements in each block, it is easy to place all critical elements in an array of size $4km$ using $O(\log k)$ time and $O(n)$ operations. Now sort both the set of critical elements and the set of their original positions in A using Cole's algorithm [5]. This takes $O(\log(km)) = O(\log k)$ time and $O(km \log k) = O(n)$ operations (recall that $m \le n^{1/3}$). Match up corresponding elements in the two sorted sequences, i.e., place the ith smallest critical element in $A[j]$, for $i = 1, 2, \ldots$, where j is the ith smallest index of a cell in A that held a critical element before the sorting. We must prove that A is now sorted.

Since every element of a heterogeneous block is critical, the subsequence of the sequence stored in A consisting of all noncritical elements is clearly sorted. After the sorting of the critical elements, the same by definition applies to the subsequence consisting of all critical elements. Furthermore, since no noncritical element is preceded by a larger element before the sorting of the critical elements, the same condition holds after the sorting. The only remaining problem is hence that some noncritical element might be followed by a smaller critical element after the sorting. To see that this is impossible, note that before the sorting, a noncritical element with value i is followed by at most $2(k-1)$ smaller elements (each such element must be a nonrepresentative in a block with header $\geq i$ but containing a value smaller than i; each input sequence contributes at most one such block), while it is preceded by at least $2k$ critical elements with value i (those in the first two homogeneous blocks with header i); the number of positions available to critical elements before the noncritical element under consideration hence exceeds the number of positions required by smaller critical elements. □

References

[1] P. Beame, M. Kik and M. Kutyłowski, Information broadcasting by exclusive read PRAMs, submitted.

[2] O. Berkman, J. JáJá, S. Krishnamurthy, R. Thurimella and U. Vishkin, Some triply-logarithmic parallel algorithms, *in* Proc. 31st Annual IEEE Symposium on Foundations of Computer Science (1990), pp. 871–881.

[3] G. Bilardi and A. Nicolau, Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines, *SIAM J. Comput.* **18** (1989), pp. 216–228.

[4] A. Borodin and J. E. Hopcroft, Routing, merging, and sorting on parallel models of computation, *J. Comput. Syst. Sci.* **30** (1985), pp. 130–145.

[5] R. Cole, Parallel merge sort, *SIAM J. Comput.* **17** (1988), pp. 770–785.

[6] R. Cole and U. Vishkin, Deterministic coin tossing with applications to optimal parallel list ranking, *Inform. and Control* **70** (1986), pp. 32–53.

[7] S. Cook, C. Dwork and R. Reischuk, Upper and lower time bounds for parallel random access machines without simultaneous writes, *SIAM J. Comput.* **15** (1986), pp. 87–97.

[8] T. Hagerup and C. Rüb, Optimal merging and sorting on the EREW PRAM, *Inform. Process. Lett.* **33** (1989), pp. 181–185.

[9] C. P. Kruskal, Searching, merging, and sorting in parallel computation. *IEEE Trans. Comput.* **32** (1983), pp. 942–946.

[10] S. Rajasekaran and J. H. Reif, Optimal and sublogarithmic time randomized parallel sorting algorithms, *SIAM J. Comput.* **18** (1989), pp. 594–607.

[11] M. Snir, On parallel searching, *SIAM J. Comput.* **14** (1985), pp. 688–708.