Optimal Parallel String Algorithms: Sorting, Merging and Computing the Minimum

T. Hagerup

MPI–I–93–152 November 1993

Optimal Parallel String Algorithms: Sorting, Merging and Computing the Minimum

T. Hagerup

MPI–I–93–152 November 1993

Optimal Parallel String Algorithms: Sorting, Merging and Computing the Minimum

Torben Hagerup[*]

Abstract: We study fundamental comparison problems on strings of characters, equipped with the usual lexicographical ordering. For each problem studied, we give a parallel algorithm that is optimal with respect to at least one criterion for which no optimal algorithm was previously known. Specifically, our main results are:

- Two sorted sequences of strings, containing altogether n characters, can be merged in $O(\log n)$ time using $O(n)$ operations on an EREW PRAM. This is optimal as regards both the running time and the number of operations.
- A sequence of strings, containing altogether n characters represented by integers of size polynomial in n, can be sorted in $O(\log n/\log\log n)$ time using $O(n\log\log n)$ operations on a CRCW PRAM. The running time is optimal for any polynomial number of processors.
- The minimum string in a sequence of strings containing altogether n characters can be found using (expected) $O(n)$ operations in constant expected time on a randomized CRCW PRAM, in $O(\log\log n)$ time on a deterministic CRCW PRAM with a program depending on n, in $O((\log\log n)^3)$ time on a deterministic CRCW PRAM with a program not depending on n, in $O(\log n)$ expected time on a randomized EREW PRAM, and in $O(\log n\log\log n)$ time on a deterministic EREW PRAM. The number of operations is optimal, and the running time is optimal for the randomized algorithms and, if the number of processors is limited to n, for the nonuniform deterministic CRCW PRAM algorithm as well.

Key words: parallel algorithms, sorting, merging, computing the minimum, strings.

1 Introduction

The recent surge of interest in computational biology has revitalized the area of string processing, the strings of interest being chiefly those encoded by molecules such as DNA. It is widely recognized that the computational demands of computational biology are such that the design of efficient parallel algorithms for its core tasks will play a major role. We contribute to the body of knowledge concerning parallel string processing.

One of the basic concerns of computational biology is to maintain a large data base of strings or objects resembling strings, the bulk of operations on which consists of searches of various kinds. In order to enable fast searching, one may choose to organize the data base

[*]Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany. Supported by the ESPRIT Basic Research Actions Program of the EC under contract No. 7141 (project ALCOM II).

as a sorted sequence, in which case the basic ordering operations of merging and sorting become very important. This presupposes that a linear ordering is imposed on the set of strings. No such linear ordering is truly natural, and different linear orderings may be preferable in different situations. As a general-purpose linear ordering, however, the standard lexicographical ordering seems as good as any. This paper studies the fundamental operations of merging, sorting and finding the minimum, as they apply to strings equipped with the lexicographical ordering.

The first problem studied is that of merging two sorted sequences of strings containing altogether n characters. The best previous algorithms for this problem [20] run in $O(\log n)$ time on the CRCW PRAM and in $O((\log n)^2)$ time on the EREW PRAM, in either case using $O(n)$ operations. We show how to achieve the $O(\log n)$ time bound of the old CRCW PRAM algorithm on the weaker EREW PRAM, while preserving the linear time-processor product. This could be viewed as somewhat surprising for the following reason: All well-known efficient EREW PRAM merging algorithms for the standard setting of constant-size objects operate in $\Omega(\log n)$ rounds of comparisons, but a comparison between two long strings needs $\Omega(\log n)$ time. Although certainly not a true lower bound, this argument would seem to indicate some difficulty in obtaining an efficient EREW PRAM string merging algorithm with a running time below $\Theta((\log n)^2)$. The new algorithm is not similar to standard EREW PRAM algorithms for merging sequences of constant-size objects, and its specialization to 1-character strings is a new optimal EREW PRAM algorithm for this task.

The new algorithm is optimal not only as regards the number of operations executed, but also as regards the running time, on both the EREW PRAM and the CREW PRAM. This follows from the fundamental lower bound of Cook *et al.* [15], which states that the computation of the OR of n bits needs $\Omega(\log n)$ time on a CREW PRAM with any number of processors, in conjunction with the following simple reduction of the computation of OR to string merging: Given n bits $b_1, \ldots, b_n$, construct the strings $X = (b_1, b_2, \ldots, b_n, 0)$ and $Y = (0, 0, \ldots, 0, 1)$ of length $n + 1$ each and merge the 1-element sequences consisting of X and Y, respectively. X will precede Y in the output sequence (i.e., be lexicographically smaller than Y) exactly if the OR of $b_1, \ldots, b_n$ is zero.

The lower bound of $\Omega(\log n)$ does not apply to the CRCW PRAM, and a much faster solution is indeed possible: Since the CRCW PRAM can compare any two strings in constant time using a linear number of processors (Lemma 7.1), it is easy to merge two sorted sequences of strings containing altogether n characters in constant time with n^2 processors by comparing every string in each sequence with all strings in the other sequence (given a suitable input representation). It is not known, however, whether there is a sublogarithmic string merging algorithm for the CRCW PRAM with optimal speedup.

The second problem considered is that of sorting a sequence of strings containing altogether n characters, whereby the characters are assumed to be represented by integers of size polynomial in n. For the CRCW PRAM, the best previous algorithm [25] uses $O(\log n)$ time and $O(n \log\log n)$ operations. We improve this to $O(\log n/\log\log n)$ time, still with $O(n \log\log n)$ operations. Noting that the string sorting problem generalizes the problem of sorting integers of polynomial size (consider each integer as a 1-character string), we can conclude from the lower bound of Beame and Hastad [7] that the new time bound of $O(\log n/\log\log n)$ is optimal for any polynomial number of processors. The time-processor product of $O(n \log\log n)$ is not known to be optimal. In the context of the well-studied integer sorting problem, however, it has stood unchallenged for six years, so that a bound below $\Theta(n \log\log n)$ would be quite surprising. Furthermore, any improvement in integer sorting would translate directly into a corresponding improvement in string sorting.

A variation of the CRCW PRAM string sorting algorithm yields corresponding algorithms for the CREW and EREW PRAMs. These have running times of $O((\log n)^{3/2}(\log\log n)^{1/2})$ and bounds on the number of operations executed around $O(n(\log n)^{1/2})$.

The final problem studied is that of computing the minimum among strings containing altogether n characters. We are not aware of any previous parallel algorithm for this problem, for which we provide a variety of results, all needing (expected) $O(n)$ operations: Constant expected time on a randomized CRCW PRAM, either $O(\log\log n)$ or $O((\log\log n)^3)$ time on a deterministic CRCW PRAM, depending on whether or not we allow nonuniformity, $O(\log n)$ expected time on a randomized EREW PRAM, and $O(\log n \log\log n)$ time on a deterministic EREW PRAM. In all cases the number of operations is clearly optimal. The result obtained for the randomized EREW PRAM is optimal for the randomized EREW and CREW PRAMs as regards time as well; this follows from [16, Theorem 5], which states that the time complexities of any boolean function on the deterministic and randomized CREW PRAMs agree to within a constant factor. The time bound of the nonuniform deterministic CRCW PRAM algorithm is optimal for any algorithm that uses at most n processors; following an $\Omega(\log\log n)$ bound for the parallel comparison-tree model given by Valiant [38], this was proved by Fich *et al.* [17].

2 Preliminaries

A *string* is a finite tuple $X = (x_1, \ldots, x_k)$ of *characters* drawn from an *alphabet* Σ equipped with a total order $<$ that can be evaluated in constant time by a single processor for any given pair of arguments. We shall occasionally write X simply as $x_1 \cdots x_k$. The integer k is called the *length* of X and is denoted by $|X|$. For ease in stating resource bounds, we assume that the empty string does not appear in the input to our algorithms. Given a string $X = (x_1, \ldots, x_k)$, take $x_{k+1} = x_{k+2} = \cdots = \flat$, where $\flat$ is a symbol not occurring in Σ ("blank"), and extend the total order $<$ from Σ to $\Sigma \cup \{\flat\}$ by declaring $\flat$ to precede every element in Σ, i.e., $\flat < x$ for all $x \in \Sigma$. Given two strings $X = (x_1, x_2, \ldots)$ and $Y = (y_1, y_2, \ldots)$, a *position in which X and Y differ* is an integer $j \in \mathbb{N}$ such that $x_j \neq y_j$. The total order on Σ induces a *lexicographical order* on the set of all strings. Let $X = (x_1, x_2, \ldots)$ and $Y = (y_1, y_2, \ldots)$ be distinct strings. Following [20], we define the *similarity* $sim(X, Y)$ of X and Y as the most significant position in which X and Y differ, i.e., $sim(X, Y) = \min\{j \in \mathbb{N} : x_j \neq y_j\}$. Now X precedes Y in the lexicographical order if and only if $x_j < y_j$, where $j = sim(X, Y)$.

The representation of a sequence of m strings is assumed to consist of two parts. The characters of each string X are stored in $|X|$ consecutive cells, not overlapping those of any other string, in a *character table*; in addition, the positions of the first and last characters of

X in the character table are given in the form of a *string descriptor*, and the descriptors of all m strings are provided together and in order in an array of size m. When we speak about merging or sorting sequences of strings, what we mean is rearranging the string descriptors to create a new sequence that is sorted with respect to the lexicographical order. Given a sequence $\mathcal{X} = (X_1, X_2, \ldots)$ or a (multi-)set $\mathcal{X} = \{X_1, X_2, \ldots\}$ of strings and a string Y, the *rank* of Y in $\mathcal{X}$, $rank(Y, \mathcal{X})$, is defined as the cardinality of the set $\{i : 1 \leq i \leq |\mathcal{X}|$ and $X_i \leq Y\}$.

For a definition of the various PRAM models, see, e.g., [24]. The variant of the CRCW PRAM used in this paper is the ARBITRARY PRAM, on which in each concurrent writing some (unknown) processor succeeds.

3 Merging

This section describes how to merge two sorted sequences of strings, containing altogether n characters, in $O(\log n)$ time using $O(n)$ operations on an EREW PRAM.

In order to understand the algorithm, it is useful first to recall the standard $O(\log\log n)$-time merging algorithm for the CREW PRAM [38, 12, 29]. The workhorse of the standard algorithm is a trivial subroutine that merges two sequences of length n each in constant time using n^2 processors by comparing every element in each sequence with every element in the other sequence. A more efficient algorithm for merging sequences of length n each uses the fast subroutine for merging sequences of $\sqrt{n}$ equally-spaced representatives from the original sequences, and then recursively merges the subsequences between successive representatives; the complete algorithm uses $O(\log\log n)$ time and n processors. A final refinement reduces the number of processors to obtain an algorithm with optimal speedup.

The top-level structure of our approach is similar. The fast and wasteful core subroutine is characterized in Lemma 3.3; it no longer takes constant time and no longer is trivial. The operation count of the fast subroutine is linear in the total number of characters involved, but slightly worse than quadratic in the number of strings. We subsequently derive an algorithm with the same properties, except that the dependence of the number of operations on the number of strings becomes strictly quadratic (Lemma 3.4). An algorithm with a slightly superlinear operation count is described in Lemma 3.5. From this, we finally obtain an algorithm with optimal speedup (Theorem 3.6).

We now describe the fast subroutine. A central concern in this subroutine is to compute the rank of a string Y in a sorted sequence $\mathcal{X} = (X_1, \ldots, X_m)$, where $m \geq 2$. Without loss of generality we will assume that of the strings $Y, X_1, \ldots, X_m$, none is a prefix of another; this property can be ensured by appending a suitable end marker to each string. Imagine that we begin by constructing a *digital search tree* for the set $\{X_1, \ldots, X_m\}$ (see [28] or [31] for a discussion of digital search trees, and Fig. 1 below for an example). Recall that a digital search tree for $\{X_1, \ldots, X_m\}$ is a rooted tree $T_{\mathcal{X}}$ whose nodes are the prefixes of strings in $\{X_1, \ldots, X_m\}$; the root is the empty string ε, and the parent of a nonempty string X in $T_{\mathcal{X}}$ is the string obtained from X by removing its last character. It is useful to consider each edge in $T_{\mathcal{X}}$, say, between a node v and its parent u, to be labeled by the unique string w

that, appended to u as a suffix, yields v (i.e., $v = uw$); in the present context, each edge label w is a single character. We also assume that the children of each node in $T_{\mathcal{X}}$ are ordered lexicographically from left to right.

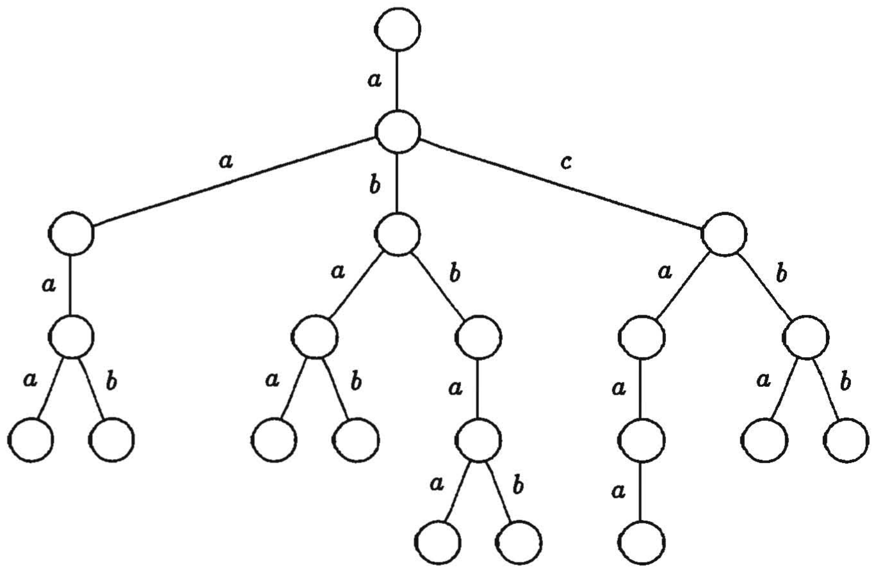

Fig. 1. A digital search tree for the strings *aaaa*, *aaab*, *abaa*, *abab*, *abbaa*, *abbab*, *acaaa*, *acba*, *acbb*.

It is very easy to determine the rank of Y in $\mathcal{X}$ via a sequential search in $T_{\mathcal{X}}$. For $j = 0, 1, \ldots, |Y|$, let Y_j be the prefix of Y of length j. Then, starting at $Y_0 = \varepsilon$, repeatedly move from Y_j to Y_{j+1} in $T_{\mathcal{X}}$, stopping at the first node Y_{j_0} such that Y_{j_0+1} is not a node in $T_{\mathcal{X}}$ (see Fig. 2). At this point insert Y in $T_{\mathcal{X}}$ and observe that the rank of Y in $\mathcal{X}$ is the number of leaves in $T_{\mathcal{X}}$ strictly to the left of the path in $T_{\mathcal{X}}$ from the root to Y; this number is easily obtained through a preorder traversal of $T_{\mathcal{X}}$.

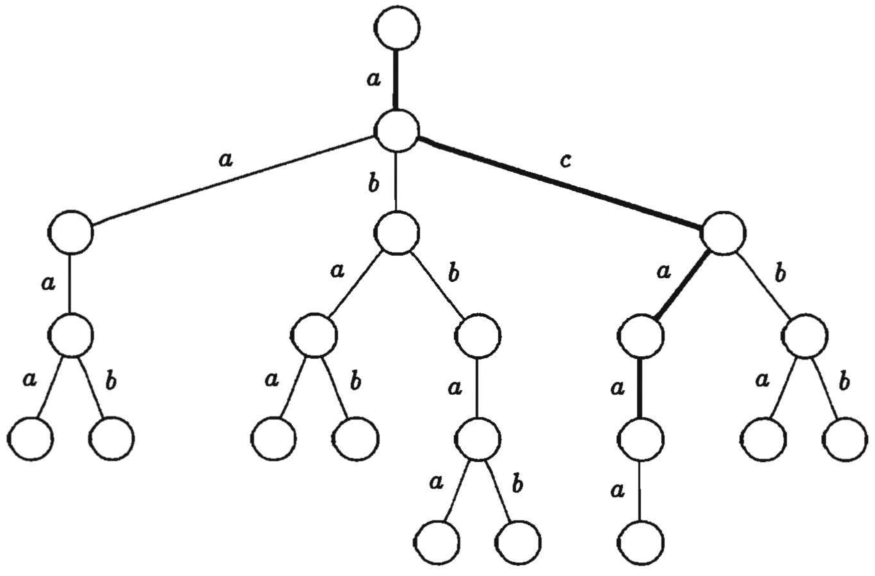

Fig. 2. The digital search tree of Fig. 1, with heavier lines denoting the path taken in a sequential search for the string *acaab*.

Computing the rank of Y in $\mathcal{X}$ fast in parallel is also easy, at least if we allow concurrent reading. We begin by marking certain edges in $T_{\mathcal{X}}$. An edge between nodes u and $v = ux$, where $x \in \Sigma$, is marked exactly if the $|v|$th character of Y equals x, i.e., if the label of the edge matches the relevant character of Y (see Fig. 3). It is easy to see that the marked edges form disjoint paths in $T_{\mathcal{X}}$. The root of $T_{\mathcal{X}}$ belongs to exactly one of these paths, called the *root path*; note that the root path is precisely the path followed by a sequential search for Y. The nodes on the root path can be identified using repeated pointer doubling, after which Y can be inserted in $T_{\mathcal{X}}$ as in the sequential case. Again the rank of Y is the number of leaves strictly to the left of Y. This quantity can be computed through an application of the Euler tour technique, described in great detail by Tarjan and Vishkin [36]. Briefly, construct a linked list L that visits the leaves of $T_{\mathcal{X}}$ in the order from left to right, label each edge of L with 1 if it leads into a leaf of $T_{\mathcal{X}}$ and with 0 otherwise, and then use repeated pointer doubling applied to L to compute the sum of the labels of the edges of L to the left of Y.

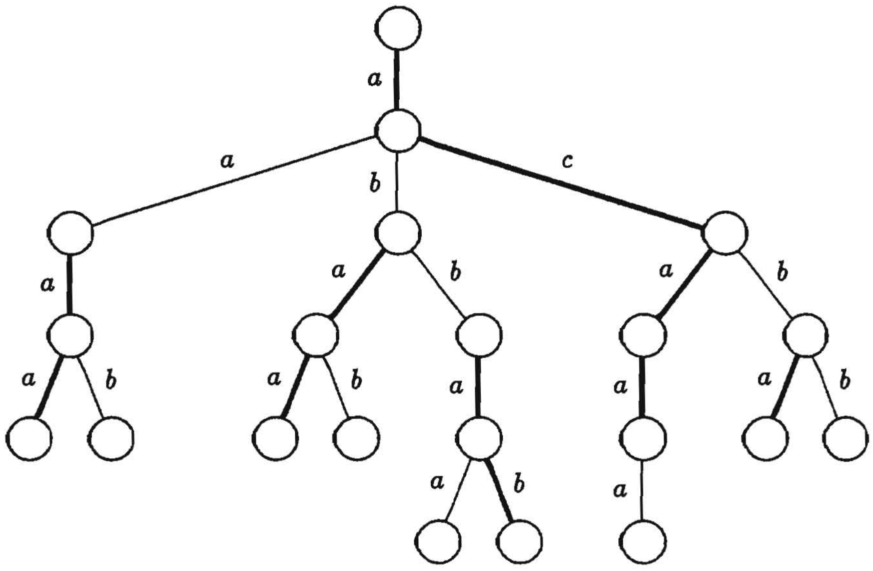

Fig. 3. The digital search tree of Fig. 1, with heavier lines denoting the edges marked in a parallel search for *acaab*.

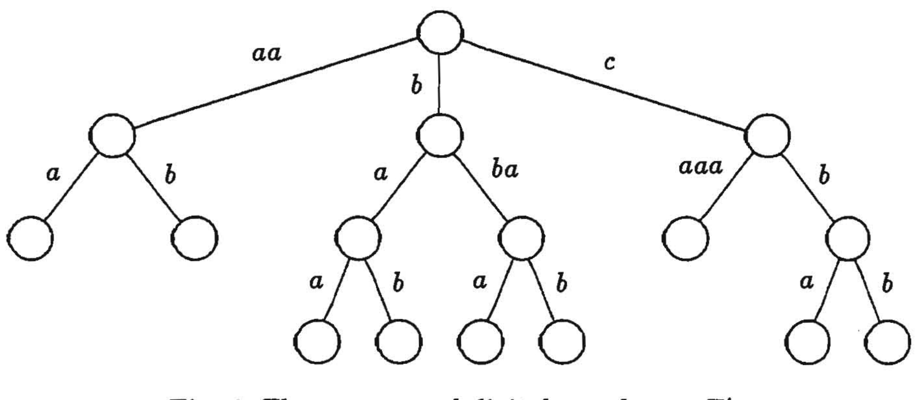

Fig. 4. The compressed digital search tree $T'_{\mathcal{X}}$.

We do not actually construct the full digital search tree $T_{\mathcal{X}}$, which may be too large for our purposes. Instead we construct a compressed digital search tree of size $O(m)$.

Let $T'_{\mathcal{X}}$ be the tree obtained from $T_{\mathcal{X}}$ by removing the nodes with exactly one child (including the root, if it has exactly one child) in any order, while after each removal of a nonroot node v making the former child of v a child of the former parent of v (see Fig. 4). Each edge in $T'_{\mathcal{X}}$ corresponds to a path in $T_{\mathcal{X}}$, and each edge label consists of one or more characters.

A parallel search in $T'_{\mathcal{X}}$ can proceed much as in $T_{\mathcal{X}}$. Again certain edges are marked. An edge between nodes u and $v = uxw$, where $x \in \Sigma$ and $w \in \Sigma^*$, is marked exactly if the $(|u| + 1)$st character of Y equals x, i.e., only the first character of an edge label is checked against the query string Y (see Fig. 5). Again the marked edges form disjoint paths, and the nodes on the root path can be determined via repeated pointer doubling.

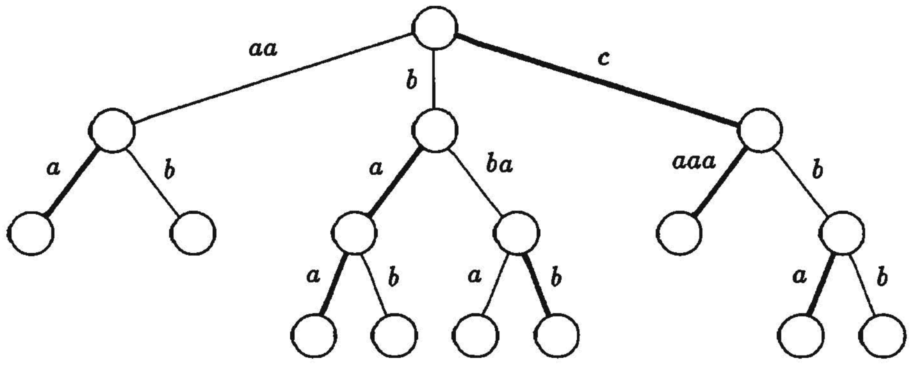

Fig. 5. The digital search tree of Fig. 4, with heavier lines denoting the edges marked in a parallel search for *acaab*.

A crucial observation is that although a sequential search in $T_{\mathcal{X}}$ may not follow the root path π in $T'_{\mathcal{X}}$ (more precisely, the corresponding path in $T_{\mathcal{X}}$) in its entirety (because of the incomplete checking of edge labels against the query string mentioned above), it will follow an initial part of π and then stop (i.e., it never leaves π); this includes the case in which the sequential search never gets as far as to the root of $T'_{\mathcal{X}}$. We can therefore insert Y in $T'_{\mathcal{X}}$, which allows us to determine the rank of Y in $\mathcal{X}$ as before, in the following way: Let w be the last node on π, whereby π is considered to be directed away from the root, and let X_l be any leaf descendant of w. Then determine $s = sim(Y, X_l)$. If $s = |w| + 1$, Y should become a new child of w. Otherwise ($s \leq |w|$) use binary search on π to determine the first node v on π with $|v| \geq s$ and note that Y should become a second child of a new node with v as its other child and the former parent of v, if any, as its parent.

The tree $T'_{\mathcal{X}}$ is still not quite what we want; we aim for an even smaller data structure. To motivate this, note that although the number of nodes in $T'_{\mathcal{X}}$ is $O(m)$, the total length of its edge labels may not be $O(m)$. Fortunately, the edge labels were introduced merely for the sake of explanation. Before we describe the construction of a slimmed-down version of $T'_{\mathcal{X}}$ that contains only the truly essential information, it is useful to take a look at a version of a data structure introduced by Vuillemin [39].

Given a sequence $a_1, \ldots, a_n$ of n distinct elements drawn from a totally ordered universe, the *Cartesian tree* of $a_1, \ldots, a_n$ is the (possibly empty) tree T on the node set $\{a_1, \ldots, a_n\}$ defined inductively as follows: (1) If $n = 0$, T is the empty tree; (2) If $n \geq 1$, the root of T

is $a_{i_0} = \min\{a_1, \ldots, a_n\}$, and the left and right subtrees of the root are the Cartesian trees of the sequences $a_1, \ldots, a_{i_0-1}$ and $a_{i_0+1}, \ldots, a_n$, respectively.

LEMMA 3.1 *For all integers $n \geq 2$, the Cartesian tree of a sequence $a_1, \ldots, a_n$ of n distinct elements drawn from a totally ordered universe can be constructed in $O(\log n)$ time using $O(n)$ operations on an EREW PRAM.*

PROOF For $i = 1, \ldots, n$, define a *nearest smaller* of a_i as an input element $a_j < a_i$ such that $a_k > a_i$ for every integer k strictly between i and j. For $i = 1, \ldots, n$, let $N(a_i)$ be the set of nearest smallers of a_i; it is easy to see that $|N(a_i)| \in \{0, 1, 2\}$. The key observation is that unless a_i is the smallest input element and hence the root of the Cartesian tree T of $a_1, \ldots, a_n$, the parent of a_i in T is $\max N(a_i)$, for $i = 1, \ldots, n$ [8]. The construction of T therefore reduces to the computation of the set of nearest smallers of each input element. As shown by Kim [27, Theorem 2.2] and Wagener [40, Theorem 3.1], the latter problem can be solved in $O(\log n)$ time using $O(n)$ operations (the algorithm of [40] was formulated for the CREW PRAM, but can be implemented without loss on the EREW PRAM [41]). □

In order to obtain our final data structure $T''_{\mathcal{X}}$, begin by computing $s_i = sim(X_i, X_{i+1})$, for $i = 1, \ldots, m-1$. Then construct the Cartesian tree T of the sequence $(s_1, 1), \ldots, (s_{m-1}, m-1)$, whereby pairs are compared using the lexicographical ordering (see Fig. 6).

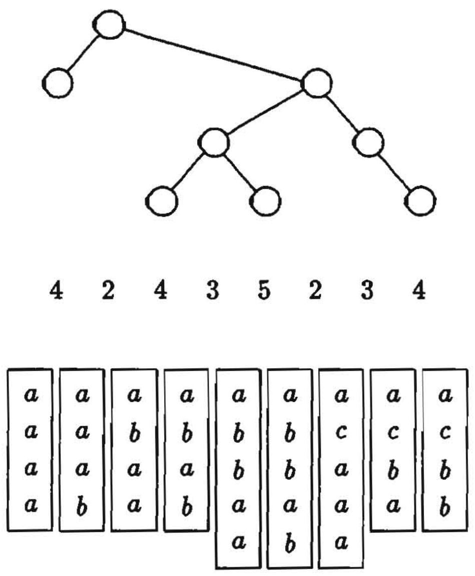

Fig. 6. From bottom to top: The sorted sequence of example strings (written vertically), the sequence of similarities between consecutive strings, and the Cartesian tree of these similarities.

Calling the nodes of T *internal*, obtain $T''_{\mathcal{X}}$ from T by adding *external* leaves as follows: Add an external left child to each node of T that has no left child, and add an external right child to each node of T that has no right child. The result is shown in Fig. 7 for our running example.

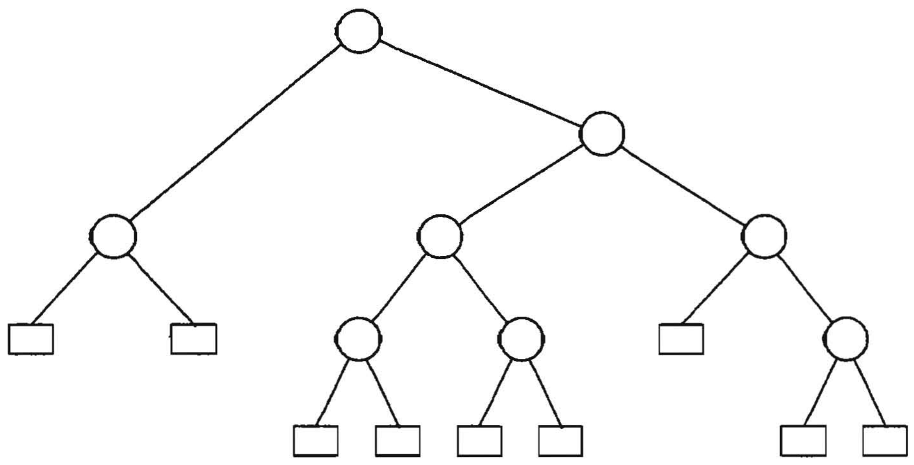

Fig. 7. The Cartesian tree of Fig. 6, with external leaves added. Internal and external nodes are shown as circles and rectangles, respectively.

It is easy to see that $T''_{\mathcal{X}}$ contains exactly m (external) leaves. We can say more, however. The tree in Fig. 7 is isomorphic to that in Fig. 4, except that each node in Fig. 4 with $k \geq 3$ children (there is precisely one such node) corresponds in Fig. 7 to a path of $k - 1$ internal nodes connected to descendants via exactly k edges. From the construction of $T'_{\mathcal{X}}$ and $T''_{\mathcal{X}}$, this property can be seen to hold in general. This makes it relatively simple to translate our search algorithm from $T'_{\mathcal{X}}$ to $T''_{\mathcal{X}}$.

The first step is the edge marking. Every internal node (s_i, i) in $T''_{\mathcal{X}}$ marks the edge leading to its left child if the s_ith character of Y is no larger than the s_ith character of X_i, and the edge leading to its right child otherwise. To see that this makes sense, note that the node (s_i, i) may be viewed as representing the fact that X_i and X_{i+1} "separate" at character position s_i. The resulting marking may not correspond to our marking of $T'_{\mathcal{X}}$, but it preserves the crucial property that the sequential search in $T_{\mathcal{X}}$ follows an initial part of the root path and then stops. The nodes on the root path can be identified in $T''_{\mathcal{X}}$ exactly as in $T'_{\mathcal{X}}$, and Y is compared to the string X_l such that the lth leaf in $T''_{\mathcal{X}}$ is the last node on the root path. A binary search is used to determine the last internal node $u = (s_i, i)$ with $s_i \leq sim(Y, X_l)$ on the root path (if there is no such node, let u be the root), after which Y can be (conceptually) inserted as a new child of u. We omit the straightforward details.

Remark If the full uncompressed digital search tree $T_{\mathcal{X}}$ of the input strings is actually desired, it can easily be derived from $T''_{\mathcal{X}}$ using $O(\log m)$ time and $O(n)$ operations, essentially by expanding edges into paths. Noting that the construction of $T''_{\mathcal{X}}$ needs $O(\log n)$ time and $O(n)$ operations, we see that $T_{\mathcal{X}}$ can be constructed in $O(\log n)$ time using $O(n)$ operations on an EREW PRAM. The problem of constructing a digital search tree for a given sorted sequence of m strings containing a total of n characters was also considered by JáJá *et al.* [26, Theorem 4.1], who achieve $O((\log m)^2/\log\log m)$ time and $O(m \log m + n)$ operations on a CRCW PRAM. Our result is clearly superior except when m is much smaller than n, in which case the running time of [26] may be $o(\log n)$. A time bound of $o(\log n)$ is not possible on the EREW PRAM, even for $m = 2$. Moving to the CRCW PRAM, however, we can easily achieve $O(\log m)$ time together with $O(n)$ operations. This is because more than

$\Theta(\log m)$ time is needed in the EREW PRAM algorithm only for computing the similarities of consecutive input strings, which can certainly be done in $O(\log m)$ time on a CRCW PRAM (Lemma 7.1).

Before proceeding with the main development, we define the *segmented broadcast* problem and recall a well-known fact following from a reduction of segmented broadcasting to (generalized) prefix summation.

DEFINITION *For all integers $n \geq 1$, the segmented broadcast problem of size n is, given an array A of n cells, some of which are marked, to store in each cell of A a copy of the value in the nearest marked cell to its left (assume that the leftmost cell of A is always marked).*

LEMMA 3.2 *For all integers $n \geq 2$, segmented broadcast problems of size n can be solved in $O(\log n)$ time using $O(n)$ operations on an EREW PRAM.*

LEMMA 3.3 *For all integers $n \geq 2$ and $m_x, m_y \geq 1$, two sorted sequences of m_x and m_y strings, containing altogether n characters, can be merged in $O(\log n)$ time using $O(n + m_x m_y + (m_x + m_y)\log n)$ operations on an EREW PRAM.*

PROOF Let the two input sequences be $\mathcal{X} = (X_1, \ldots, X_{m_x})$ and $\mathcal{Y} = (Y_1, \ldots, Y_{m_y})$ and assume that no input string is a prefix of another input string. It suffices to show how to compute the rank of Y_i in $\mathcal{X}$, for $i = 1, \ldots, m_y$. Since the algorithm for this task was essentially described above, we mainly need to bound the time and the number of operations and to show how to avoid concurrent reading. A time bound of $O(\log n)$ will be obvious for all steps of the algorithm and will not be mentioned explicitly in the following.

Begin by computing $s_i = sim(X_i, X_{i+1})$, for $i = 1, \ldots, m_x - 1$, which needs $O(n)$ operations. Then construct the Cartesian tree T of $(s_1, 1), (s_2, 2), \ldots, (s_{m_x-1}, m_x - 1)$ and obtain $T''_{\mathcal{X}}$ from T by adding the external leaves. According to Lemma 3.1, this can be done using $O(m_x)$ operations. Now create m_y copies of the single tree $T''_{\mathcal{X}}$ constructed so far. Since the size of $T''_{\mathcal{X}}$ is $O(m_x)$, this needs $O(m_x m_y)$ operations. As a result, each string in $\mathcal{Y}$ can work on its own copy of $T''_{\mathcal{X}}$. Consider therefore one particular string Y in $\mathcal{Y}$ and its associated copy of $T''_{\mathcal{X}}$.

In order to mark one of the edges to its children, each internal node in $T''_{\mathcal{X}}$ needs to compare a character in Y with a character in a string in $\mathcal{X}$. The latter character can be considered to be built into $T''_{\mathcal{X}}$ and therefore poses no problem of concurrent reading. Likewise, the *position* of the relevant character in Y can be built into $T''_{\mathcal{X}}$, whereas the actual character in that position clearly depends on Y. In order to access the characters in Y without concurrent reading, essentially carry out the standard simulation of one step of a CREW PRAM on an EREW PRAM. Specifically, first sort the nodes in $T''_{\mathcal{X}}$ by the position in Y that they want to inspect, which partitions the nodes into segments of nodes that want to access the same cell. Then let only the first node in each segment carry out the actual reading and use segmented broadcasting (Lemma 3.2) to distribute the value read to all other nodes in the segment. Except for the initial sorting, the simulation needs $O(m_x)$ operations, which sums to $O(m_x m_y)$ operations over all strings Y. The sorting takes $O(m_x \log m_x) = O(m_x \log n)$ operations, which is too much for being summed over all strings Y. Once again, however, the sorting is preprocessing that needs to be done only once and whose outcome can be built into $T''_{\mathcal{X}}$.

The depth of each node in $T''_{\mathcal{X}}$ can be obtained via the Euler tour technique. With this precomputed information available to guide the pointer doubling, it is not too difficult to identify the root path using $O(m_x)$ operations, which sums to $O(m_x m_y)$ over all strings Y. This informs each string Y of a string X_l with which it is to be compared. Note that it suffices to provide each string Y with a private copy of the first $\min\{|X_l|, |Y|+1\}$ characters of X_l — any remaining characters of X_l are of no relevance to the comparison. We will show how to do this without concurrent reading. Divide each of the requested strings X_l into *blocks* of $\lfloor \log n \rfloor$ characters each, except for one block that may be smaller, and let each requesting string Y generate a *read request* for each block that it wants. Since the total length of the strings Y is bounded by n, the total number of read requests is $O(n/\log n + m_y)$. We can therefore sort the read requests by the requested blocks using $O(n + m_y \log n)$ operations. Similarly as in the simulation of concurrent reading earlier in the proof, we now use segmented broadcasting to satisfy the read requests using $O(n + m_y \log n)$ operations. The difference to the earlier setting is that here the number of read requests is a factor of at least $\log n$ smaller than the desired operation bound, whereas each request is for at most $\log n$ characters. Running the standard algorithm $\lfloor \log n \rfloor$ times in parallel performs as required.

The remainder of the algorithm is easy. The binary search on the root path of $T''_{\mathcal{X}}$ is done sequentially and independently for each string Y, and the final Euler tour computation can be moved to a precomputation step; the precomputation should mark each node in $T''_{\mathcal{X}}$ with the position of its first and last leaf descendants in the ordering of all leaves from left to right.

□

The algorithms described in the three next proofs are instances of a generic merging algorithm, a variant of which appears in [20]. The generic algorithm inputs two sorted sequences $\mathcal{X}$ and $\mathcal{Y}$ of pairwise distinct strings and then executes the following steps:

Step 1 (Choose representatives)
Construct subsequences $\mathcal{X}' = (X_1, \ldots, X_p)$ and $\mathcal{Y}' = (Y_1, \ldots, Y_q)$ of *representatives* of $\mathcal{X}$ and $\mathcal{Y}$, respectively, marking each representative with its original position.

Step 2 (Merge representatives)
Merge $\mathcal{X}'$ and $\mathcal{Y}'$ into a sequence $\mathcal{Z} = (Z_1, \ldots, Z_{p+q})$.

Step 3 (Prepare for ranking representatives)
For $i = 1, \ldots, p+q$, determine $\mathit{rank}(Z_i, \mathcal{X}')$. Then, for $i = 0, \ldots, p$, compute $\mathcal{Z}'_i$ and $\mathcal{X}'_i$ as the subsequences of $\mathcal{Z}$ and $\mathcal{X}$, respectively, of strings of rank i in $\mathcal{X}'$.

Step 4 (Rank representatives)
For $i = 0, \ldots, p$, merge $\mathcal{Z}'_i$ and $\mathcal{X}'_i$.

Step 5 (Set up subproblems)
For $i = 1, \ldots, p+q$, determine $\mathit{rank}(Z_i, \mathcal{X})$. Use the outcome to compute the rank in $\mathcal{Z}$ of each string in $\mathcal{X}$. Then, for $i = 0, \ldots, p+q$, construct $\mathcal{X}_i$ as the subsequence of $\mathcal{X}$ of strings of rank i in $\mathcal{Z}$.

Steps 3′–5′ Repeat Steps 3–5, but with $\mathcal{X}$, $\mathcal{X}'$, $\mathcal{X}_i$ and $\mathcal{X}_i'$ replaced by $\mathcal{Y}$, $\mathcal{Y}'$, $\mathcal{Y}_i$ and $\mathcal{Y}_i'$, respectively, and with p and q interchanged. This constructs $\mathcal{Y}_i$ as the subsequence of $\mathcal{Y}$ of strings of rank i in $\mathcal{Z}$, for $i = 0, \ldots, p+q$.

Step 6 (Solve subproblems)
For $i = 0, \ldots, p+q$, merge $\mathcal{X}_i$ and $\mathcal{Y}_i$ into a sequence $\mathcal{Z}_i$.

Step 7 (Combine solutions of subproblems)
Concatenate $\mathcal{Z}_0, \ldots, \mathcal{Z}_{p+q}$ in this order.

The correctness of the generic algorithm is readily established. If the input strings contain a total of n characters, Steps 3, 5 and 7 are easily executed in $O(\log n)$ time using $O(n)$ operations, and Steps 3′–5′ are analogous to Steps 3–5; the details are left to the reader, who should note that the segmented broadcasting of Lemma 3.2 comes in handy more than once. The implementation of the remaining steps is discussed in the proofs to follow.

Given a set S' of elements of a finite sequence S, define the *maximum gap* of S' in S as the maximum length of a contiguous subsequence of S that contains no element of S'.

LEMMA 3.4 *For all integers $n \geq 2$ and $m_x, m_y \geq 1$, two sorted sequences of m_x and m_y strings, containing altogether n characters, can be merged in $O(\log n)$ time using $O(n + m_x m_y)$ operations on an EREW PRAM.*

PROOF Without loss of generality assume that $m_x \geq m_y$. It is easy to see that the algorithm of Lemma 3.3 performs as required unless $m_y \leq \log n$. The case $m_y \leq \log n$ can be handled using the following somewhat degenerate form of the generic algorithm: In Step 1, choose a set of $O(n/\log n)$ representatives from $\mathcal{X}$ whose maximum gap in $\mathcal{X}$ is $O(\log n)$ and declare every element of Y to be a representative. In Step 2, merge the representatives using the algorithm of Lemma 3.3. This needs $O(\log n)$ time and $O(n)$ operations. In Step 4, again use the algorithm of Lemma 3.3. Since at most $\log n$ of the sequences Z_i' actually include an element of Y and every merging problem involves $O(\log n)$ strings, $O(n)$ operations certainly suffice. Furthermore, since every element of Y is a representative, Steps 3′–5′ and 6 are not needed — Step 5 already establishes the rank of every element in the opposite sequence. □

LEMMA 3.5 *For all integers $n \geq 4$, two sorted sequences, containing altogether n characters, can be merged in $O(\log n)$ time using $O(n \log\log n)$ operations on an EREW PRAM.*

PROOF We describe a recursive instance of the above generic algorithm. Let the two sequences to be merged be $\mathcal{X}$ and $\mathcal{Y}$ and take $m = |\mathcal{X}| + |\mathcal{Y}|$. There are two cases. If $m^2 \leq n$ (Case 1), apply the algorithm of Lemma 3.4, which takes $O(\log n)$ time and uses $O(n + m^2) = O(n)$ operations. Otherwise (Case 2) execute the generic algorithm according to the following specification:

Step 1 Choose a set of $O(\sqrt{m})$ representatives from $\mathcal{X}$ whose maximum gap in $\mathcal{X}$ is $O(\sqrt{m})$; analogously, choose a set of $O(\sqrt{m})$ representatives from $\mathcal{Y}$ whose maximum gap in $\mathcal{Y}$ is $O(\sqrt{m})$.

Step 2 Merge the representatives using the algorithm of Lemma 3.4. This needs $O(\log n)$ time and uses $O(n + \sqrt{m}\sqrt{m}) = O(n)$ operations.

Step 4 Solve every merging problem using the algorithm of Lemma 3.4. Since the total size of the sequences $\mathcal{Z}'_0, \ldots, \mathcal{Z}'_p$ is $O(\sqrt{m})$, while each sequence $\mathcal{X}'_i$ is of size $O(\sqrt{m})$, this needs $O(\log n)$ time and $O(n + \sqrt{m}\sqrt{m})$ operations.

Step 6 Solve the subproblems recursively.

For $m, n \in \mathbb{N}$, denote by $T(m,n)$ the time taken by the above algorithm to merge two sequences of at most m strings each and containing altogether at most n characters. Case 1 in the algorithm requires $O(\log n)$ time, whereas Case 2 is easily seen to need $O(\log n) + T(c\sqrt{m}, n)$ time, for some constant $c > 0$. Since Case 2 is not entered unless $n < m^2$ and hence $\log n = O(\log m)$, there is a constant $c > 0$ such that

$$T(m,n) \le \max\{c \log n, c \log m + T(c\sqrt{m}, n)\}.$$

This recurrence solves to $T(m,n) = O(\log n)$. The depth of recursion is $O(\log\log m) = O(\log\log n)$, and each recursive level uses $O(n)$ operations, for a total of $O(n \log\log n)$ operations. $\square$

Theorem 3.6 *For all integers $n \ge 4$, two sorted sequences, containing altogether n characters, can be merged in $O(\log n)$ time using $O(n)$ operations on an EREW PRAM.*

Proof We again use an instance of the generic algorithm. The steps of interest are implemented as follows:

Step 1 Choose $O(n/\log\log n)$ representatives from $\mathcal{X}$ whose maximum gap in $\mathcal{X}$ is $O(\log\log n)$ and that contain $O(n/\log\log n)$ characters; it is easy to see that such representatives exist and can be found in $O(\log n)$ time using $O(n)$ operations. Compute representatives from $\mathcal{Y}$ in the obvious analogous way.

Step 2 Merge the representatives using the algorithm of Lemma 3.5.

Step 4 Solve the merging problems using the algorithm of Lemma 3.4. The total number of strings in $\mathcal{Z}'_0, \ldots, \mathcal{Z}'_p$ is $O(n/\log\log n)$ and each $\mathcal{X}'_i$ contains $O(\log\log n)$ strings, so that this takes $O(\log n)$ time and needs $O(n)$ operations.

Step 6 Each subproblem to be solved comprises $O(\log\log n)$ strings. If the strings in a subproblem contain fewer than $\lceil \log\log n \rceil^2$ characters, solve the problem sequentially, which takes $O((\log\log n)^2)$ time. Otherwise solve it in $O(\log n)$ time using the algorithm of Lemma 3.4. In either case, the number of operations needed is proportional to the number of characters in the subproblem.

$\square$

4 Sorting on the CRCW PRAM

The first fast and efficient parallel algorithm for string sorting on the CRCW PRAM, designed only for the case of strings of equal length, was given by Vaidyanathan *et al.* [37]. Their basic insight was that two adjacent character positions (such as the two most significant positions) of the strings to be sorted can be combined into a single character position as follows: Within each string, interpret the substring consisting of the two characters in the chosen positions as a single integer in an order-preserving manner (e.g., according to a suitable positional system), then sort the substrings using an integer sorting algorithm, and finally replace each substring by its rank in the resulting sorted sequence, viewed as a 1-character string over the alphabet $\{1, \ldots, n\}$. This replaces two character positions by a single position, as desired, without altering the relative order of the strings to be sorted.

The above algorithm can be applied in parallel to any set of disjoint pairs of adjacent character positions, whereby it makes no difference whether the substrings originating in a particular pair of character positions are sorted separately or together with all other substrings; in the interest of simplicity, we will assume the latter. The number of character positions can therefore be reduced from l to $\lceil l/2 \rceil$ in the time needed by a single integer sorting. Since the initial number of character positions is no larger than n, carrying out at most $\lceil \log n \rceil$ iterations as described replaces the original input strings by 1-character strings with the same relative order, which can be determined in one final integer sorting. Since the total number of characters in the strings to be sorted drops by a constant factor in each iteration, the operation count of the whole algorithm is within a constant factor of that of the first iteration. If the character-combining subroutine is implemented using the algorithm of Bhatt *et al.* [10], which sorts n integers in $O(\log n/\log\log n)$ time using $O(n \log\log n)$ operations, the complete string sorting algorithm of Vaidyanathan *et al.* runs in $O((\log n)^2/\log\log n)$ time using $O(n \log\log n)$ operations.

Following the demonstration by Vaidyanathan *et al.* that efficient parallel string sorting is feasible, the basic scheme was improved in two directions. Hagerup and Petersson [20] adapted the algorithm to the case of strings of different lengths without compromising the resource bounds of the original algorithm. Subsequently JáJá and Ryu [25] observed that it suffices to execute $\Theta(\log\log n)$ character-combining iterations, since the resulting reduction in the problem size by a factor of $\Omega(\log n)$ allows one to switch to a less efficient, but faster comparison-based sorting algorithm. Based on this, they derived the best algorithm predating the present paper, which works in $O(\log n)$ time using $O(n \log\log n)$ operations.

Taking the algorithm of JáJá and Ryu as our starting point, we introduce additional improvements that reduce the running time to the optimal $O(\log n/\log\log n)$, still with $O(n \log\log n)$ operations. The basic idea is to replace the integer sorting subroutine by a subroutine for so-called *padded integer sorting*, which turns out to be just as useful for the present application. Since padded integer sorting can be done much faster than standard integer sorting, we can carry out many more than $\Theta(\log\log n)$ character-combining iterations. This allows us to build up a significant *processor advantage*, i.e., the problem size drops far below the number of available processors, and this in turn enables us to speed up the final comparison-based sorting from $O(\log n)$ to $O(\log n/\log\log n)$.

Before turning to the details of the algorithm, we discuss the concept of padded (integer) sorting, which was introduced by MacKenzie and Stout [30]. Following Hagerup and Raman [21], to *padded-sort* n keys with padding factor $\lambda \geq 0$ is to output them in sorted order in an array of size at most $(1+\lambda)n$, unused cells in the output array being filled with a special *null* value. The following was proved in [22, Theorem 20]:

LEMMA 4.1 *For all integers $n \geq 4$, n integers of size polynomial in n can be padded-sorted with constant padding factor in $O((\log n)^{1/2}(\log\log n)^{3/2})$ time using $O(n\log\log n)$ operations on a CRCW PRAM.*

In the algorithm of JáJá and Ryu and its precursors, the sorting of the 2-character substrings serves to determine the rank of each substring within the set of all substrings. Since a lower bound of $\Omega(\log n/\log\log n)$ applies to the computation of ranks, while we intend to be significantly faster, we must substitute a different quantity for the rank of a substring. To this end, note that the only properties of the rank function relevant to us are (1) equal substrings have equal ranks, (2) smaller substrings have smaller ranks, and (3) ranks are integers of size polynomial in n. Instead of the rank of a substring, we can therefore use its *pseudo-rank*, defined relative to a particular padded-sorted array of the substrings as the position in the padded-sorted array of the first occurrence of the substring.

We now describe the new string sorting algorithm at three successively more detailed levels, using stepwise refinement in order not to overwhelm the reader with implementation details before the overall idea is clear.

First description. The algorithm consists of three *phases*. Phase 1 executes a number of *rounds*, in each of which characters are paired and the input size is reduced as in the algorithm of [37]. A complication not present in the original algorithm is due to strings that, after a number of rounds, consist of a single character. Such strings must be removed from the process, since character-pairing operations applied to them no longer yield any reduction in the problem size, i.e., their continued presence could ruin the operation bound. Each round therefore separates out a sorted sequence of 1-character strings that do not participate in subsequent rounds. Phase 2 sorts those strings that survive Phase 1, i.e., those that are not removed in any round. The third phase, starting with the sorted sequence produced in Phase 2, gradually merges back the sequences removed in Phase 1 in the reverse order of their removal, which produces the final sorted output sequence.

Second description. Each round of Phase 1 consists of the following: First the strings remaining from previous rounds are padded-sorted by their first characters with constant padding factor according to Lemma 4.1, whereby in the case of ties strings of length 1 are considered smaller, and the strings of length 1 are removed and saved for Phase 3. Each of the remaining strings is replaced by the corresponding sequence of substrings of length 2 each; in order to do this for the strings of odd length as well, a suffix consisting of one '$\sharp$' character is first appended to each such string. Then the substrings of length 2 are padded-sorted with constant padding factor, and each substring is replaced (in its position in the original strings) by its pseudo-rank. This creates the set of input strings for the next round.

It is easy to see that any two successive rounds reduce the total number of characters by a factor of at least 2. We fix the number of rounds at $N = 2\lceil (\log n)^{1/3} \rceil$. Then, by Lemma 4.1, all rounds can be executed in $O(N(\log n)^{1/2}(\log\log n)^{3/2}) = o(\log n/\log\log n)$ time using $O(n \log\log n)$ operations, and this reduces the number of characters to at most $n/2^{N/2}$. At the beginning of Phase 2, we can therefore allocate $k = 2^{8\lceil (\log n)^{1/4} \rceil}$ processors to each remaining character. This allows us to execute any set of pairwise comparisons between strings in constant time, provided that no string participates in more than k comparisons at a time. The task therefore is to find an algorithm for standard comparison-based sorting that works in $O(\log n/\log\log n)$ time without ever comparing an element to more than k other elements simultaneously. Such an algorithm can be derived from the AKS network [1]. Recall that the AKS network is a sorting network, in the sense of Knuth [28], with n data lines and $O(\log n)$ levels of comparators (see Fig. 8).

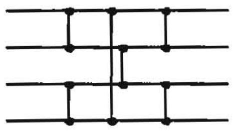

Fig. 8. A sorting network with 4 inputs and 3 levels of comparators.

If the AKS network is divided into *blocks* of at most $r = \lfloor \frac{1}{3} \log\log k \rfloor$ successive levels each (a related idea was used in [5]), fan-in considerations imply that each output of a block is a function of at most 2^r inputs to the block, and fan-out considerations imply that at most 2^r outputs of the block depend on any particular input. Within each block, therefore, each fixed input is compared only to inputs in a fixed set of size at most $2^{2r} \le k$, so that an algorithm that processes one block at a time and carries out all such potential comparisons within each block is suitable for being used in Phase 2. We argue below that there is a CRCW PRAM algorithm of this kind that processes each block in constant time, which yields a total running time of $O(\log n/r) = O(\log n/\log\log n)$.

Phase 3 starts with the sequence $\mathcal{X}$ of strings sorted in Phase 2 and then, for $i = N, \ldots, 1$, merges into $\mathcal{X}$ the sorted sequence of strings removed in Round i of Phase 1. We will ensure that each comparison relevant to this merging can be carried out in constant time by one processor, so that any standard merging algorithm can be used. Since standard merging problems involving n elements can be solved in $O(\log\log n)$ time using $O(n)$ operations [29], it is then easy to see that the time and the number of operations needed for Phase 3 are dominated by those consumed in Phase 1.

Third description. We now discuss the remaining implementation details. A first issue is how to compute the pseudo-ranks needed in Phase 1. We reduce this problem to the segmented broadcast problem defined in Section 3. The following lemma, due to Berkman and Vishkin [9] and Ragde [34], shows that segmented broadcasting can be done faster on the CRCW PRAM than on the EREW PRAM. The authors mentioned actually provide much faster ("inverse Ackermann") running times, but the weaker form given here suffices for our purposes.

LEMMA 4.2 *For all integers $n \geq 4$, segmented broadcast problems of size n can be solved in $O(\log\log n)$ time using $O(n)$ operations on a CRCW PRAM.*

Following the padded-sorting of substrings in a round in Phase 1, the corresponding pseudo-ranks can be determined via two applications of Lemma 4.2. In the first application, each substring learns the position of the nearest substring to its left, if any (recall that *null* entries may intervene), which enables it to decide whether it is a first occurrence. In the second application, each substring that is not a first occurrence is informed of the position of the nearest first occurrence to its left, which is its pseudo-rank. By Lemma 4.2, the cost of computing pseudo-ranks is dominated by that of the preceding padded sorting.

In order to actually capitalize on the geometric reduction in the problem size during Phase 1, we need to compact the remaining strings and characters into arrays of size linear in their number after each round. This problem is formalized as the *interval allocation* problem below, after which we give a lemma showing that the resources needed for the interval allocation are also dominated by those used by the preceding padded sorting.

DEFINITION *For all integers $n \geq 1$, the interval allocation problem of size n is, given n nonnegative integers $a_1, \ldots, a_n$, to compute an upper bound $\widehat{s}$ on $s = \sum_{j=1}^{n} a_j$ with $\widehat{s} = O(s)$ and to allocate (i.e., compute the offsets of) n nonoverlapping subarrays of sizes $a_1, \ldots, a_n$ of a base array of $\widehat{s}$ cells.*

LEMMA 4.3 [19] *For all integers $n \geq 4$, interval allocation problems of size n can be solved in $O((\log\log n)^3)$ time using $O(n)$ operations on a CRCW PRAM.*

We next show that the operation of the AKS network can indeed be simulated in $O(\log n/\log\log n)$ time, as claimed above. Because of the large processor advantage available, we will usually not worry about the number of processors needed. Specifically, we bound the number of processors needed in the most processor-intensive substep of the algorithm and leave the consideration of the remaining substeps to the reader. Recall that $k = 2^{8\lceil(\log n)^{1/4}\rceil}$ and $r = \lfloor\frac{1}{3}\log\log k\rfloor$.

Working through the descriptions of the AKS network given, e.g., by Chvátal [13], Paterson [32] and Pippenger [33], one can see that (a reasonable graph representation of) an m-input AKS network can be constructed in $O(\log m/\log\log m)$ time on a CRCW PRAM using $O(m\log m)$ operations. Furthermore, each comparator in the network can be labeled with its level number, so that it is easy, in our case, to divide the network into $O(\log n/\log\log n)$ blocks of at most r successive levels each.

Using a sequential search, each output line of a block can now determine the set of at most 2^r input lines of the block whose values affect it, and place requests for all pairwise comparisons between these values with the input lines; this device is necessary because the comparisons will be carried out by processors associated with the input strings, i.e., values. Furthermore, simulating the operation of part of the block for each of the at most $2^{2^{2r}}$ possible outcomes of all pairwise comparisons between these input values in parallel, $2^{2^{2r}}$ processors associated with the output line can construct a table mapping each such possible outcome, encoded as an integer, to the input line whose value will appear on the output line. The part of the sorting algorithm described so far can be viewed as preprocessing.

Presented with a set of input values, a block first carries out those comparisons that were requested during the preprocessing, and each output line obtains the outcome of those comparisons that affect it. Using a team of 2^{2r} processors for each of the at most $2^{2^{2r}}$ possible outcomes, it then converts the outcome of the comparisons to the corresponding integer, performs a lookup in the table discussed above, and produces its value in constant time. The sorting hence takes constant time per block and $O(\log n/\log\log n)$ time altogether, as required. The number of processors needed is dominated by the $2^{2^{2r}}$ teams of 2^{2r} processors each for each output line of a block, a total of $O(\log n \cdot 2^{2^{2r}} \cdot 2^{2r}) = o(k)$ processors per string to be sorted.

We finally take a closer look at the reintroduction in Phase 3 of the 1-character sequence removed in Round i of Phase 1, for some $i \in \{1, \ldots, N\}$, the goal being to show that every comparison between a "new" string (one that is currently being reintroduced) and an "old" string (one that was reintroduced earlier or that was never removed in Phase 1) can be carried out in constant time by a single processor. Note that there was a time when the relevant comparisons were easy, namely after the padded-sorting of strings by their first characters in Round i of Phase 1; at that time comparing two strings could have been done simply by comparing their positions in the array A_i resulting from the padded-sorting by first characters. We therefore save A_i and use it to facilitate comparisons in Phase 3. All that is required is that before reintroducing the "new" strings removed in Round i of Phase 1, we mark both "old" and "new" strings with their positions in A_i. This can be done in constant time using a number of operations proportional to the size of A_i; by the geometric decrease in the number of remaining strings during Phase i, this sums to $O(n)$ operations over all rounds.

We have proved

THEOREM 4.4 *For all integers $n \geq 4$, a sequence of strings, containing altogether n characters represented by integers of size polynomial in n, can be sorted in $O(\log n/\log\log n)$ time using $O(n \log\log n)$ operations on a CRCW PRAM.*

More generally, we have

THEOREM 4.5 *Suppose that n integers of size polynomial in n can be sorted in $O(t(n))$ time using $O(nq(n))$ operations on a CRCW PRAM, for all $n \in \mathbb{N}$ and for nondecreasing functions $t, q : \mathbb{N} \to \mathbb{N}$. Then for all $n \in \mathbb{N}$, a sequence of strings, containing altogether n characters represented by integers of size polynomial in n, can be sorted in $O(t(n))$ time using $O(nq(n))$ operations on a CRCW PRAM.*

5 Sorting with a General Alphabet

The algorithm in the previous section assumes that characters are represented by integers of size polynomial in n, the total number of characters. Apart from being motivated by practical considerations, this serves to make the problem interesting. In a comparison-based setting in which information about characters can be obtained only through pairwise comparisons, the usual lower bounds for sorting apply, i.e., the sorting requires $\Omega(n \log n)$ operations, and $\Theta(n \log n)$ operations can be achieved only together with a running time of $\Omega(\log n)$ [3, 5, 11]. On the other hand, a variant of the idea of Vaidyanathan *et al.* [37] used in the previous section makes it easy to carry out the sorting in $O(\log n)$ time using $O(n \log n)$ operations: First sort the n characters present in the input, then replace all characters in the input strings by their ranks in the sorted sequence, which does not alter the relative order of the strings, and finally apply the sorting algorithm developed in the previous section for the case of integer characters.

JáJá *et al.* [26] investigate the question in finer detail by introducing the number of input strings as a second complexity parameter and show that m strings containing altogether n characters can be sorted in $O((\log m)^2/\log\log m)$ time using $O(m \log m + n)$ operations on a CRCW PRAM. The time bound results from the execution of $O(\log m)$ iterations, each of which solves merging and prefix summation problems of size $O(m)$. We can obtain an immediate improvement by noting that the $O(\log m/\log\log m)$-time exact prefix summation can be replaced by the faster approximate prefix summation of [22, Corollary 9]. The only nontrivial observation needed is that padded-sorted sequences can be merged with the aid of the algorithm of Lemma 4.2.

THEOREM 5.1 *For all integers $n, m \geq 8$, m strings containing altogether n characters (drawn from a general alphabet) can be sorted in $O(\log m (\log\log m)^4/\log\log\log m)$ time using $O(m \log m + n)$ operations on a CRCW PRAM.*

6 Sorting on the CREW and EREW PRAMs

Our CREW and EREW PRAM algorithms for string sorting are similar to the CRCW PRAM algorithm described in Section 4. It is advantageous, however, to implement Phase 2 using a sorting algorithm based on multiway merging, rather than on the AKS network. Suppose, as in Section 4, that Phase 2 is carried out with k processors allocated to each character. Starting with each string forming a 1-element sequence by itself, we then repeatedly merge sequences in disjoint groups of k sequences each. After $\lceil \log n/\log k \rceil$ merging steps, only a single sorted sequence remains.

In order to merge the k sequences in a group, merge all pairs of sequences. This yields the rank of each string in each of the k sequences, and its rank in the combined sequence can be obtained by adding these; with the rank of each string available, of course, the merging can be completed in constant time. Since each character can contribute one processor to each merging in which it participates, Theorem 3.6 implies that each merging step can be executed in $O(\log n)$ time, and hence that the complete sorting algorithm works in $O((\log n)^2/\log k)$ time. We will balance this contribution with the time needed to obtain a processor advantage of k.

In dealing with the substrings in Phase 1, we revert from padded sorting to standard sorting. For the CREW PRAM algorithm, we rely on an algorithm by Albers and Hagerup [2] that sorts n integers in $O(\log n \log\log n)$ time using $O(n(\log n)^{1/2})$ operations. Applying this algorithm in $\Theta((\log n)^{1/2}/(\log\log n)^{1/2})$ successive rounds in Phase 1 uses $O(n(\log n)^{1/2})$ operations, takes $O((\log n)^{3/2}(\log\log n)^{1/2})$ time and allows us to execute Phase 2 with $\log k = \Omega((\log n)^{1/2}/(\log\log n)^{1/2})$. We hence have

THEOREM 6.1 *For all integers $n \geq 4$, a sequence of strings, containing altogether n characters represented by integers of size polynomial in n, can be sorted in $O((\log n)^{3/2}(\log\log n)^{1/2})$ time using $O(n(\log n)^{1/2})$ operations on a CREW PRAM.*

In the case of the EREW PRAM algorithm, we fix some parameters differently. The situation is more complicated, since the best known integer sorting algorithm for the EREW PRAM [2] exhibits a tradeoff between speed and efficiency: For all t with $\log n \log\log n \leq t \leq (\log n)^{3/2}/(\log\log n)^{1/2}$, n integers can be sorted in $O(t)$ time using $O(n(\log n)^2/t)$ operations. Our best strategy is to begin by operating the algorithm at the slowest point of its tradeoff curve, where it uses $O((\log n)^{3/2}/(\log\log n)^{1/2})$ time and $O(n(\log n)^{1/2}(\log\log n)^{1/2})$ operations. After each group of 4 rounds, however, the problem size has decreased by a factor of at least 4, and we can allow the algorithm to run a factor of 2 faster (if it is not already running in $O(\log n \log\log n)$ time). This ensures that both the total time and the total number of operations consumed by the algorithm over all rounds will be within a constant factor of the corresponding resource bounds for the first round, except that each round takes at least $\Theta(\log n \log\log n)$ time. We fix the number of rounds at the same value as for the CREW PRAM algorithm, which can easily be seen to result in the same overall time bound.

THEOREM 6.2 *For all integers $n \geq 4$, a sequence of strings, containing altogether n characters represented by integers of size polynomial in n, can be sorted in $O((\log n)^{3/2}(\log\log n)^{1/2})$ time using $O(n(\log n)^{1/2}(\log\log n)^{1/2})$ operations on an EREW PRAM.*

Just as for the CRCW PRAM algorithm (cf. Theorem 4.5), any improvement in integer sorting on the CREW or EREW PRAM will yield an improvement in string sorting on the same machine. We omit the details.

7 Computing the Minimum

We describe several related algorithms for the task of computing the minimum string among strings containing altogether n characters. They center around the following ideas.

On the CRCW PRAM, computing the minimum among strings of the same length is essentially the same as computing the minimum among constant-size objects: When an algorithm for the standard setting compares two objects, a derived algorithm for the string setting can compare the two corresponding strings in constant time (Lemma 7.1). The same is true if the input strings, although not of the same length, have lengths that differ by at most a constant factor. We can therefore partition the input strings into $O(\log n)$ *groups* such that the lengths of any two strings in the same group differ by at most a factor of 2 and compute the minimum within each group. This leaves us with only one candidate string in each group, and the minimum among these can be found in constant time. The main outstanding problem is to rearrange the input strings so that the strings forming each group occur together, which is necessary for processing the group efficiently.

It is well-known that the minimum of two strings can be determined in constant time on a CRCW PRAM. We include a proof for the sake of completeness.

LEMMA 7.1 *For all integers $n, m \geq 1$, the similarity of two distinct strings of n and m characters can be computed in constant time on a CRCW PRAM with $k = \min\{n, m\}$ processors. Hence two strings of n and m characters can be compared within the same resource bounds.*

PROOF Let the input strings be $X = (x_1, \ldots, x_n)$ and $Y = (y_1, \ldots, y_m)$ and compute b_j, for $j = 1, \ldots, k$, as follows: $b_j = 1$ if $x_j \neq y_j$, and $b_j = 0$ otherwise. If $b_1 = \cdots = b_k = 0$, $sim(X, Y) = k + 1$. Otherwise $sim(X, Y) = \min\{j : 1 \leq j \leq k \text{ and } b_j = 1\}$, and this quantity can be computed in constant time with k processors using the algorithm of Fich *et al.* [18, Theorem 1]. □

The following lemma shows that input strings of the same length are easy to handle.

LEMMA 7.2 *For all integers $n, m \geq 4$ such that m divides n, the minimum among n/m strings of m characters each can be computed in $O(\log\log n)$ time using $O(n)$ operations on a CRCW PRAM.*

PROOF Without loss of generality assume that the input strings are pairwise distinct. Carry out $\lceil \log\log n \rceil$ preprocessing steps, in each of which the remaining strings are compared in pairs, with at most one string left out. Within each pair, the maximum is discarded. Since each stage reduces the number of remaining strings by at least a constant factor, Lemma 7.1 shows that the preprocessing can be carried out in $O(\log\log n)$ time using $O(n)$ operations. As a result, we can associate m processors with each remaining string. We now simulate the standard minimum-finding algorithm [35], which works in $O(\log\log n)$ time: Whenever the standard algorithm uses one processor to compare two constant-size objects, we use m processors to compare the two corresponding strings in constant time (Lemma 7.1). This computes the minimum string in $O(\log\log n)$ time. □

Another easy case is that in which the lengths of the input strings form a geometric series.

LEMMA 7.3 *Let $n \in \mathbb{N}$ and let $\mathcal{X}$ be a sequence of strings, containing altogether n characters, such that for $i = 1, 2, \ldots, \lfloor \log n \rfloor$, $\mathcal{X}$ contains at most one string whose length lies in the interval $[2^{i-1}, 2^i)$. Then the minimum string in $\mathcal{X}$ can be found in constant time using $O(n)$ operations on a CRCW PRAM.*

PROOF Comparing a string of length m in $\mathcal{X}$ with all shorter strings in $\mathcal{X}$ can be done in constant time using $O(m)$ operations. Doing this for all strings in $\mathcal{X}$ uses constant time and $O(n)$ operations, and the minimum is easily deduced from the outcome of the comparisons. □

We can now combine Lemmas 7.2 and 7.3 to obtain a general algorithm that works in logarithmic time.

Lemma 7.4 *For all integers $n \geq 4$, the minimum among strings containing altogether n characters can be computed in $O(\log n)$ time using $O(n)$ operations on a CRCW PRAM.*

Proof Increase the length of each input string to the nearest larger power of 2 by padding it with the '$\natural$' character, after which there are only $O(\log n)$ different lengths. Then sort the strings by their lengths using the integer sorting algorithm of Cole and Vishkin [14, remark following Theorem 2.3] or Wagner and Han [42]. Since the number of distinct keys is $O(\log n)$, this takes $O(\log n)$ time and uses $O(n)$ operations. Then apply the algorithm of Lemma 7.2 separately to each group of strings of a common length, which takes $O(\log\log n)$ time and $O(n)$ operations. This leaves exactly one string in each length group, and the overall minimum can be determined in constant time using the algorithm of Lemma 7.3. □

The bottleneck in the above algorithm clearly is the initial integer sorting. In order to derive a faster algorithm, suppose that at some point $O(n/\log n)$ candidates for being the minimum are left. We can then use an interval allocation routine first to place the remaining candidate strings in an array of size $O(n/\log n)$, and subsequently independently for each length group to place the strings in that group in an array only a constant factor larger than the number of strings in the group, after which we can continue as in the proof of Lemma 7.4. In order to reduce the number of candidate strings to $O(n/\log n)$, partition the input strings into groups of $\Theta(\log n)$ strings each and use the algorithm of Lemma 7.4 to compute the minimum within each group of those strings in the group that are of length at most $\lceil \log n \rceil$. Because of the length restriction, this takes $O(\log\log n)$ time and uses $O(n)$ operations. It leaves a number of strings of length $\geq \log n$, plus at most one additional string per group. The total number of remaining strings is therefore $O(n/\log n)$, as desired.

It can be shown that interval allocation problems of size n can be solved in $O(\log\log n)$ time by a nonuniform algorithm using $O(n)$ operations. Complementing Lemma 4.3 with this result, we obtain

Theorem 7.5 *For all integers $n \geq 4$, the minimum among strings containing altogether n characters can be computed on a CRCW PRAM using $O(n)$ operations in $O((\log\log n)^3)$ time by a uniform algorithm, and in $O(\log\log n)$ time by a nonuniform algorithm.*

We finally develop a randomized algorithm. In the following discussion, "with high probability" will always mean with probability $1 - 2^{-n^{\Omega(1)}}$. Without loss of generality assume that the input strings are pairwise distinct.

The minimum of n constant-size objects can be found in constant time with n processors with high probability [4], and interval allocation problems of size n can be solved in constant time with n processors with high probability if the number of nonzero input values (in our case, strings to be compacted) is $O(n/\log n)$ [6, Corollary 4.3]. If we draw a random sample from the input strings by including each string with probability $1/\lfloor\sqrt{n}\rfloor$ and independently of all other strings, a well-known Chernoff bound (see, e.g., [23]) shows that with high probability the number of strings in the sample will be $O(\sqrt{n})$, so that we can use these facts to compute the minimum string in the sample in constant time with high probability. Subsequently comparing every input string with the minimum string X in the sample, we are left with the problem of computing the minimum only among the strings no larger than X. With high probability, the number of such strings is $O(n^{2/3})$; specifically, the probability that none of the k smallest input strings is included in the sample is at most $(1-1/\sqrt{n})^k \le e^{-k/\sqrt{n}}$, which for $k = \lfloor n^{2/3} \rfloor$ is negligible. The remaining problem can therefore also be solved in constant time with high probability.

THEOREM 7.6 *There is a constant $\epsilon > 0$ such that for all integers $n \ge 1$, the minimum among strings containing altogether n characters can be computed in constant time on an n-processor CRCW PRAM with probability at least* $1 - 2^{-n^\epsilon}$.

It is easy to simulate the algorithms of Theorems 7.5 and 7.6 on an EREW PRAM with a logarithmic slowdown, but uniformly and without any increase in the number of operations. We hence have

THEOREM 7.7 *For all integers $n \ge 4$, the minimum among strings containing altogether n characters can be computed on an EREW PRAM using $O(\log n \log\log n)$ time and $O(n)$ operations.*

THEOREM 7.8 *There is a constant $\epsilon > 0$ such that for all integers $n \ge 1$, the minimum among strings containing altogether n characters can be computed on an EREW PRAM using $O(\log n)$ time and $O(n)$ operations with probability at least* $1 - 2^{-n^\epsilon}$.

Acknowledgment

Useful discussions with Shiva Chaudhuri, Ola Petersson, Prabhakar Ragde and Rajeev Raman are gratefully acknowledged.

References

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

[2] S. Albers and T. Hagerup. Improved parallel integer sorting without concurrent writing. In *Proc. 3rd Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 463–472, 1992.

[3] N. Alon and Y. Azar. The average complexity of deterministic and randomized parallel comparison-sorting algorithms. *SIAM J. Comput.*, 17:1178–1192, 1988.

[4] N. Alon and N. Megiddo. Parallel linear programming in fixed dimension almost surely in constant time. In *Proc. 31st Ann. Symp. on Foundations of Computer Science*, pages 574–582, 1990.

[5] Y. Azar and U. Vishkin. Tight comparison bounds on the complexity of parallel sorting. *SIAM J. Comput.*, 16:458–464, 1987.

[6] H. Bast, M. Dietzfelbinger, and T. Hagerup. A perfect parallel dictionary. In *Proc. 17th International Symp. on Mathematical Foundations of Computer Science, Springer Lecture Notes in Computer Science, Vol. 629*, pages 133–141, 1992.

[7] P. Beame and J. Hastad. Optimal bounds for decision problems on the CRCW PRAM. *J. ACM*, 36:643–670, 1989.

[8] O. Berkman, B. Schieber, and U. Vishkin. Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. *J. Algorithms*, 14:344–370, 1993.

[9] O. Berkman and U. Vishkin. Recursive *-tree parallel data-structure. In *Proc. 30th Ann. Symp. on Foundations of Computer Science*, pages 196–202, 1989.

[10] P. C. P. Bhatt, K. Diks, T. Hagerup, V. C. Prasad, T. Radzik, and S. Saxena. Improved deterministic parallel integer sorting. *Inform. and Comput.*, 94:29–47, 1991.

[11] R. B. Boppana. The average-case parallel complexity of sorting. *Inform. Process. Lett.*, 33:145–146, 1989.

[12] A. Borodin and J. E. Hopcroft. Routing, merging, and sorting on parallel models of computation. *J. Comput. System Sci.*, 30:130–145, 1985.

[13] V. Chvátal. Lecture notes on the new AKS sorting network. DIMACS Tech. Report 92-29, Rutgers University, New Brunswick, NJ, 1992.

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

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

[16] M. Dietzfelbinger, M. Kutyłowski, and R. Reischuk. Exact time bounds for computing Boolean functions on PRAMs without simultaneous writes. In *Proc. 2nd Ann. ACM Symp. on Parallel Algorithms and Architectures*, pages 125–135, 1990.

[17] F. E. Fich, F. Meyer auf der Heide, P. Ragde, and A Wigderson. One, two, three ... infinity: Lower bounds for parallel computation. In *Proc. 17th Ann. ACM Symp. on Theory of Computing*, pages 48–58, 1985.

[18] F. E. Fich, P. Ragde, and A. Wigderson. Relations between concurrent-write models of parallel computation. *SIAM J. Comput.*, 17:606–627, 1988.

[19] T. Hagerup. Fast deterministic processor allocation. In *Proc. 4th Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 1–10, 1993.

[20] T. Hagerup and O. Petersson. Merging and sorting strings in parallel. In *Proc. 17th International Symp. on Mathematical Foundations of Computer Science, Springer Lecture Notes in Computer Science, Vol. 629*, pages 298–306, 1992.

[21] T. Hagerup and R. Raman. Waste makes haste: Tight bounds for loose parallel sorting. In *Proc. 33rd Ann. Symp. on Foundations of Computer Science*, pages 628–637, 1992.

[22] T. Hagerup and R. Raman. Fast deterministic approximate and exact parallel sorting. In *Proc. 5th Ann. ACM Symp. on Parallel Algorithms and Architectures*, pages 346–355, 1993.

[23] T. Hagerup and C. Rüb. A guided tour of Chernoff bounds. *Inform. Process. Lett.*, 33:305–308, 1990.

[24] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, Reading, Mass., 1992.

[25] J. F. JáJá and K. W. Ryu. An efficient parallel algorithm for the single function coarsest partition problem. In *Proc. 5th Ann. ACM Symp. on Parallel Algorithms and Architectures*, pages 230–239, 1993.

[26] J. JáJá, K. W. Ryu, and U. Vishkin. Sorting strings and constructing digital search trees in parallel. Tech. Report CS–TR–3073, University of Maryland, College Park, MD, 1993.

[27] S. K. Kim. Optimal parallel algorithms on sorted intervals. Tech. Report 90–01–04, Dept. of Computer Science and Engineering, University of Washington, Seattle, WA, 1990.

[28] D. E. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley, Reading, Mass., 1973.

[29] C. P. Kruskal. Searching, merging, and sorting in parallel computation. *IEEE Trans. Comput.*, C-32:942–946, 1983.

[30] P. D. MacKenzie and Q. F. Stout. Ultra-fast expected time parallel algorithms. In *Proc. 2nd Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 414–423, 1991.

[31] K. Mehlhorn. *Data Structures and Algorithms, Vol. 1: Sorting and Searching.* Springer-Verlag, Berlin, Germany, 1984.

[32] M. S. Paterson. Improved sorting networks with $O(\log N)$ depth. *Algorithmica*, 5:75–92, 1990.

[33] N. Pippenger. Communication networks. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. A, Algorithms and Complexity*, chapter 15, pages 805–833. Elsevier/The MIT Press, 1990.

[34] P. Ragde. The parallel simplicity of compaction and chaining. *J. Algorithms*, 14:371–380, 1993.

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

[36] R. E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM J. Comput.*, 14:862–874, 1985.

[37] R. Vaidyanathan, C. R. P. Hartmann, and P. K. Varshney. Optimal parallel lexicographic sorting using a fine-grained decomposition. Tech. Report SU-CIS-91-01, School of Computer and Information Science, Syracuse University, Syracuse, NY, 1991.

[38] L. G. Valiant. Parallelism in comparison problems. *SIAM J. Comput.*, 4:348–355, 1975.

[39] J. Vuillemin. A unifying look at data structures. *Comm. ACM*, 23:229–239, 1980.

[40] H. Wagener. Triangulating a monotone polygon in parallel. In *Proc. International Workshop on Computational Geometry, Springer Lecture Notes in Computer Science, Vol. 333*, pages 136–147, 1988.

[41] H. Wagener. Personal communication, October 1990.

[42] R. A. Wagner and Y. Han. Parallel algorithms for bucket sorting and the data dependent prefix problem. In *Proc. 1986 International Conf. on Parallel Processing*, pages 924–930, 1986.