OPTIMAL DOUBLY LOGARITHMIC PARALLEL ALGORITHMS BASED ON FINDING ALL NEAREST SMALLER VALUES

Omer Berkman * *Baruch Schieber* † *Uzi Vishkin* ‡

Abstract

The *all nearest smaller values* problem is defined as follows. Let $A = (a_1, a_2, \ldots, a_n)$ be n elements drawn from a totally ordered domain. For each a_i, $1 \leq i \leq n$, find the two nearest elements in A that are smaller than a_i (if such exist): the left nearest smaller element a_j (with $j < i$) and the right nearest smaller element a_k (with $k > i$). We give an $O(\log\log n)$ time optimal parallel algorithm for the problem on a CRCW PRAM. We apply this algorithm to achieve optimal $O(\log\log n)$ time parallel algorithms for four problems: (i) Triangulating a monotone polygon, (ii) Preprocessing for answering range minimum queries in constant time, (iii) Reconstructing a binary tree from its inorder and either preorder or postorder numberings, (vi) Matching a legal sequence of parentheses.
We also show that any optimal CRCW PRAM algorithm for the triangulation problem requires $\Omega(\log\log n)$ time.

*Dept. of Computing, King's College London, The Strand, London WC2R 2LS, England. Part of this work was carried out while this author was at University of Maryland Institute for Advanced Computer Studies (UMIACS), College Park, MD 20742; and Tel Aviv University, Tel Aviv, Israel 69978. Partially supported by NSF grant CCR-8906949.

†IBM Research Division, T.J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598.

‡University of Maryland Institute for Advanced Computer Studies (UMIACS), and Dept. of Electrical Engineering, University of Maryland, College Park, MD 20742; and Tel Aviv University, Tel Aviv, Israel 69978. Partially supported by NSF grants CCR-8906949 and CCR-911348, ONR grant N00014-85-K-0046, the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy under contract number DE-AC02-76ER03077 and the Foundation for Research in Electronics, Computers and Communication, administered by the Israeli Academy of Sciences and Humanities.

1. Introduction

The *all nearest smaller values* problem (abbreviated *ANSV*) is defined as follows. Let $A = (a_1, a_2, \ldots, a_n)$ be an array of elements from a totally ordered domain. For each a_i, $1 \leq i \leq n$, find the nearest element to the left of a_i and the nearest element to the right of a_i that are less than a_i, if such elements exist. In other words, for each $1 \leq i \leq n$, find the maximal $1 \leq j < i$, and the minimum $i < k \leq n$ such that $a_j < a_i$ and $a_k < a_i$. We say that a_j is the *left match* and a_k is the *right match* of a_i.

The models of parallel computation used in this paper are the concurrent-read concurrent-write (CRCW) parallel random access machine (PRAM), and the the concurrent-read exclusive-write (CREW) PRAM. A PRAM employs p synchronous processors all having access to a common memory. A CRCW PRAM allows simultaneous access by more than one processor to the same memory location for both read and write operations. We assume a weak CRCW PRAM model in which several processors may attempt to write simultaneously at the same location only if they write the *same* value. (This model is called Common CRCW PRAM.) A CREW PRAM allows simultaneous access by more than one processor to the same memory location only for read operations. See [EG88], [KR90] and [Vis91] for surveys of results concerning PRAMs.

A parallel algorithm attains *optimal speedup* if its time-processor product is (asymptotically) the same as the time complexity of the best known sequential algorithm for the same problem. A parallel algorithm is *optimal* if this product is (asymptotically) equal to the lower bound on the time complexity of any sequential algorithm for the problem. A primary goal in parallel computation is to design optimal algorithms that also run as fast as possible.

We present a parallel algorithm for the ANSV problem, which runs in $O(\log \log n)$ (doubly logarithmic) time using $n / \log \log n$ processors on a CRCW PRAM. This is optimal since the ANSV problem has a simple linear time serial algorithm.

We argue that the ANSV problem is fundamental. First, two elementary problems: merging two sorted lists [BH85, Kru83] and finding the maximum of n elements [Val75] can be reduced to this problem. However, it still has the same serial and parallel *tight* complexity bounds as each of them. Second, the $O(\log \log n)$ time optimal algorithm for ANSV implies doubly logarithmic optimal algorithms for four problems:

1. Triangulating a monotone polygon. For this problem we also show that any CRCW PRAM algorithm for triangulating an n-vertex monotone polygon requires $\Omega(\log \log n)$ time when $O(n \log^c n)$ processors are available, for any fixed constant $c > 0$.
2. Preprocessing for answering range minimum queries in constant time. Given an array of

n real numbers $A = (a_1 \ldots, a_n)$, a range minimum query requests the minimum element in a sub-array $A_{i,j} = (a_i, \ldots, a_j)$, for some $1 \leq i \leq j \leq n$. The goal is to preprocess that array so that any range query can be answered in constant time using a single processor. Parallelizing the query processing is straightforward: k queries can be processed in constant time using k processors on a CREW PRAM.

3. Reconstructing a binary tree from its inorder and either preorder or postorder numberings.

4. Matching parentheses. Given a legal sequence of left and right parentheses and the level of nesting of each parenthesis, find the mate for each parenthesis. In case the levels of nesting are not given, we can still match all parentheses in $O(\log n / \log\log n)$ time employing an optimal number of processors, by first finding the nesting levels using the parallel prefix-sums algorithm of Cole and Vishkin [CV89].

The doubly logarithmic optimal algorithms for these problems make them *highly parallelizable problems* as classified in [BBG$^+$89]. The fact that the ANSV problem captures intrinsic difficulties in each of these problems which are from different domains, as well as the fact that merging and maximum finding can be reduced to ANSV support our claim that it is indeed a fundamental problem. Recently, [KLP89] applied our ANSV algorithm to obtain an optimal parallel algorithm for forest matching.

In many logarithmic time algorithms on arrays, the computation is guided by a complete binary tree whose leaves correspond to the elements of the input array. See, e.g., the prefix sums algorithm of [Sto75] or [LF80]. The computation in our doubly logarithmic algorithm is guided by a *balanced doubly logarithmic height tree* to be defined later.

Below, we discuss each of our four applications.

The problem of triangulating a monotone polygon received considerable attention in the literature. For instance, Garey et al. [GJPT78] gave a linear time serial algorithm for the problem, and Fournier and Montuno [FM84] achieved similar performance for triangulating a one-sided monotone polygon (OSMP). Our parallel algorithm implies new serial algorithms for these two problems. Interestingly, for the OSMP problem our serial algorithm is simpler. Each of the parallel methods of Aggarwal et al. [ACG$^+$85], Yap [Yap88] and Atallah and Goodrich [AG86] (given for a CREW PRAM) for triangulating a simple polygon uses, as its main subroutine, an algorithm for triangulating an OSMP. Their bounds for triangulating an OSMP is $O(\log n)$ time using n processors. Goodrich [Goo89] gave an algorithm for the problem of triangulating a monotone polygon on a CREW PRAM in $O(\log n)$ time using $n/\log n$ processors. This algorithm is quite involved. A variant of our algorithm matches this bound but is considerably simpler.

Gabow, Bentley and Tarjan [GBT84] observe that a range minimum search over any sub-array can be reduced to answering a lowest common ancestor (LCA) query in the *Cartesian tree* data structure introduced in [Vui80]. Using this observation they give a linear time sequential preprocessing algorithm for answering range minimum queries in constant time. Our preprocessing algorithm combines this serial approach with a preprocessing algorithm of [AS87]. Note that the binary minimum operation is not a *general* semi-group operation. Yao [Yao82] and Alon and Schieber [AS87] show that on-line retrieval of information on each sub-array relative to a general semi-group operation needs non-constant time if only linear amount of work is invested in the preprocessing stage. This lower bound result prohibits generalization of our results to any semi-group operations.

We mention two recent papers that need range minimum (or maximum) search: [ALV90] on scaled string matching, and [RV88] on parallel triconnectivity. Other applications of the algorithm are given in [GBT84].

Bar-On and Vishkin [BV85] gave a logarithmic optimal parallel algorithm for parentheses matching and Anderson, Mayr and Warmuth [AMW89] achieved logarithmic time using a linear number of processors (on weaker models of computation). The problem of reconstructing a binary tree from its inorder and either preorder or postorder numberings was introduced in [Knu73] (§2.3.1, Page 329, Ex. 7). Burgdorff et al. [BLSZ87] gave an $O(n^2)$ serial algorithm for this problem.

The paper is organized as follows. In Section 2 we define and give an algorithm for the prefix minima problem which is an important subroutine in our ANSV algorithm. The algorithm for the ANSV problem is given in Section 3. In Section 4 we give upper and lower bounds for the problem of triangulating a monotone polygon. In sections 5, 6 and 7 we show how to apply the ANSV algorithm to the other problems.

2. The prefix minima problem

Let $A = (a_1, a_2, \ldots, a_n)$. The *prefix minima* of A are $p_1, \ldots, p_n$, where p_i, is the minimum among $a_1, \ldots, a_i$. Similarly, the *suffix minima* of A are $s_1, \ldots, s_n$, where s_i, is the minimum among $a_i, \ldots, a_n$.

We describe a recursive algorithm for finding the prefix minima of an array A. It runs in $O(\log\log n)$ time using $n/\log\log n$ processors on a CRCW PRAM. A similar algorithm was given in [Sch87].

Suppose we have n^2 processors. We show that in this case both the ANSV and the prefix minima problems can be solved in constant time. We begin by presenting the *1-color minimiza-*

tion problem of [FRW88] and the constant time n processors algorithm for it. This algorithm is used to get the constant time ANSV algorithm. This last algorithm will serve as a subroutine in the constant time n^2 processors prefix minima algorithm as well as in the doubly logarithmic time ANSV algorithm of the next section.

The 1-color minimization problem. The input to the problem is an array of n elements whose value is either zero or one. The output is the minimum index of the element whose value is one. Fich, Ragde and Wigderson [FRW88] proposed the following constant time algorithm for this problem: Partition the input array into $\sqrt{n}$ successive subarrays each of length $\sqrt{n}$. For each such subarray, find, in constant time using $\sqrt{n}$ processors, if it has a one. Then, using n processors, apply the constant time algorithm of Shiloach and Vishkin [SV81] for finding the first of those subarrays that has a one. Finally, reapply this algorithm for finding the minimum index of a one in this subarray.

The constant time n^2 processors ANSV algorithm. Allocate n processors to each a_i. The jth processor, $1 \leq j \leq n$, that is allocated to a_i, sets $c_{ij} := 1$ if $a_j < a_i$. Otherwise, it sets $c_{ij} := 0$. Now, the largest $j < i$ such that $c_{ij} = 1$ is a_i's left match. This j can be found using the algorithm for the 1-color minimization problem. The right match for each element a_i is found similarly.

The constant time n^2 processors prefix minima algorithm. First, solve the ANSV problem with respect to A, using the above algorithm. Then, for each a_i, $1 \leq i \leq n$, let $j \leq i$ be the largest index such that a_j does not have a left match. Then, a_j is the minimum value among $a_1, \ldots, a_i$. This j can be found by the algorithm for the 1-color minimization problem.

The doubly logarithmic prefix minima algorithm. We now give a recursive procedure that finds the prefix minima of $A = (a_1, a_2, \ldots, a_n)$ in $O(\log\log n)$ time using n processors. We later show how to reduce the number of processors to $n/\log\log n$.

Step 1: Partition A into $\sqrt{n}$ subsets of $\sqrt{n}$ elements each.

Step 2: Find the prefix minima of each subset using $\sqrt{n}$ processors, recursively. Denote the minimum in subset i by b_i. Let $B = (b_1, \ldots, b_{\sqrt{n}})$.

Step 3: Find the prefix minima of B using the constant time algorithm presented above.

Step 4: Let a_j be an element in subset i. The minimum among $a_1, \ldots, a_j$ is the minimum of two precomputed values: the minimum among $b_1, \ldots, b_{i-1}$ and the minimum among

$a_{(i-1)\sqrt{n}+1}, \ldots, a_j$. (Note that $a_{(i-1)\sqrt{n}+1}$ is the first element in subset i.)

Steps 1, 3 and 4 can be implemented in constant time using n processors. Since the depth of the recursion is $O(\log\log n)$, the algorithm runs in $O(\log\log n)$ time using n processors.

The optimal $O(\log\log n)$ algorithm is derived in a standard fashion from the above algorithm. First, partition A into $n/\log\log n$ subsets of $\log\log n$ elements each. Then, using one processor per subset, find the prefix minima in each subset. Second, use the above algorithm to solve the prefix minima problem with respect to the minimum in each of the $n/\log\log n$ subsets. Third, extend this into prefix minima for all elements in a way similar to Step 4.

The balanced doubly logarithmic height tree. Considering the non-recursive description of the (non-optimal) prefix minima algorithm, it turns out that its flow (as well as the flow of most of the rest of our algorithms) is guided by a balanced doubly logarithmic height tree. First, we define the tree. The leaves of the tree correspond to the n inputs of the problem. Any internal node of height $h > 1$ has $2^{2^{h-2}}$ children. An internal node of height one has two children. This implies that the number of leaves in the rooted subtree of any internal node of height $h > 1$ is $2^{2^{h-1}}$ (that is, the square of the number of its children). Clearly, the height of the tree is at most $\log\log n + 1$.
The prefix minima algorithm can be viewed as a bottom up computation in this doubly logarithmic tree. For each one of the nodes at a given height, we compute the prefix minima with respect to the elements associated with the leaves of its subtree. This is done using the prefix minima computed for its children.

3. The All Nearest Smaller Values (ANSV) problem

In this section, we give an algorithm for solving the ANSV problem in $O(\log\log n)$ time using $n/\log\log n$ processors.

3.1. Preliminaries

To gain some insight into the ANSV problem we note two known problems that can be solved (in parallel) using a (parallel) algorithm for the ANSV problem. The first problem is finding the minimum (maximum) of n elements [Val75]. The ANSV problem is at least as hard as the problem of finding the minimum of n elements since the minimum is simply the unique element with no left or right matches. The second problem is merging [BH85, Kru83]. We show that the same holds for the merging problem.

Reduction from merging to ANSV. Let $A = (a_1, a_2, \ldots, a_n)$ and $B = (b_1, b_2 \ldots, b_n)$ be two monotonically increasing arrays, that are to be merged. The reduction to the ANSV problem is done by constructing an array $C = (a_1, \ldots, a_n, b_n, \ldots, b_1)$. We solve the ANSV problem with respect to C. If b_j is the nearest smaller right match of a_i then the location of a_i in the merged list is $i + j$. Similarly, we find the location of each of the b_k's using its nearest *non-larger* left match.

Finally, we show that using an algorithm for the ANSV problem, it is possible to find for each element in an input vector $C = (c_1, \ldots, c_n)$ its nearest *non-larger* elements. Let $D = ((c_1, 1), (c_2, 2), \ldots, (c_n, n))$. Solve the ANSV problem with respect to D where comparisons are made *lexicographically*. Suppose (c_j, j) is the left match of (c_i, i) for some $2 \leq i \leq n$. Then, c_j is the nearest element to the left of c_i which is not larger than c_i. Finding the nearest non-larger element to the right of each element in C can be achieved by defining $E = ((c_1, n), (c_2, n-1), \ldots, (c_n, 1))$, and then solving the ANSV problem with respect to E.

3.2. The logarithmic time algorithm

In this subsection we give an algorithm for the ANSV problem that takes $O(\log n)$ time using $n/\log n$ processors on a CREW PRAM. In the next subsection we show how to extend it to run in $O(\log\log n)$ time using $n/\log\log n$ processors on a CRCW PRAM.

From now on assume that all elements in A are distinct. It can be shown that this assumption is without loss of generality using a construction similar to the one given above in the reduction from merging to ANSV. Also, assume for simplicity that n is a power of 2. Construct a complete binary tree with n leaves each corresponding to an element of the input array A. Then, find for each internal node of the tree the minimum value of its descendant leaves. This can be implemented to run in $O(\log n)$ time using $n/\log n$ processors by constructing the tree level by level from the leaves to the root. Now, for each element a_i find its left and right matches as follows.

The basic search procedure: Suppose some processor is allocated to an element a_i. The processor finds the left match of a_i by climbing up the tree starting at a_i until it hits a node such that the value of its left sibling is smaller than a_i. Then it proceeds down the tree aiming at the rightmost leaf whose value is smaller than a_i.

Note that the basic search procedure implies an $O(\log n)$ time algorithm using n processors for our problem. Next, we show how to reduce the number of processors to $n/\log n$.

Step 1: Partition A into $n/\log n$ subsets of $\log n$ elements each.

Step 2: Allocate a processor to each subset and solve the ANSV problem serially with respect to the subset. This is done using a stack in the obvious way.

Step 3: Find the minimum, prefix minima and suffix minima in each subset. Let $b(i)$ denote the index of the minimum element in subset i.

Step 4: Find left and right matches for each of the $a_{b(i)}$'s using the basic search procedure.

In steps 5 and 6 below, we finish finding left and right matches for all elements. For this, we need some definitions and observations.

For $1 \leq j \leq n$, define $r(j)$ to be the index of the right match of a_j, $l(j)$ the index of its left match, $gr(j)$ the subset containing $a_{r(j)}$ and $gl(j)$ the subset containing $a_{l(j)}$.

Lemma 3.1: *For $1 \leq i \leq n/\log n$, suppose $r(b(i))$ exists and $gr(b(i)) \neq i+1$. Then, there exists a subset $i < k < gr(b(i))$ such that $gl(b(k)) = i$ and $gr(b(k)) = gr(b(i))$. Moreover, subset k is unique. (In words, there is a single subset k between subset i and subset $gr(b(i))$ such that the left match of $b(k)$ is in subset i and the right match is in subset $gr(b(i))$.) See Figure 3.1 (the subarrays A_1 and A_2 that appear in Figure 3.1 are defined later).*

[Insert Figures 3.1 and 3.2 here.]

Proof: Let $j = gr(b(i))$ and let k be such that $a_{b(k)} = \min\{a_{b(i+1)}, \ldots, a_{b(j-1)}\}$. Then, $a_{b(k)}$ has the minimum value among all elements of subsets $i+1, \ldots, j-1$, so $gl(b(k)) \leq i$ and $gr(b(k)) \geq j$. Since $gr(b(i)) = j$ and $i < k < j$, it follows that $a_{b(k)} > a_{b(i)} > a_{b(j)}$. Thus $gl(b(k)) = i$ and $gr(b(k)) = j$. Moreover, the definition of k implies that if $i < k' < k$, then $gr(b(k')) \leq k < j$ and if $k < k' < j$, then $gl(b(k')) \geq k > i$. Thus, k is unique. □

Lemma 3.2: *For $1 \leq i \leq n/\log n$, suppose $l(b(i))$ exists and $gl(b(i)) \neq i-1$. Then there exists a subset $gl(b(i)) < k < i$ such that $gr(b(k)) = i$ and $gl(b(k)) = gl(b(i))$. Moreover, subset k is unique.*

The proof of Lemma 3.2 is analogous to the proof of Lemma 3.1.

3.2.1. High–level description of steps 5 and 6

The idea is to reduce the problem of finding the remaining matches into many (actually, $2n/\log n$) merging problems that are solved separately, in parallel.

Characterizing the merging tasks. We characterize the pairs of subarrays of A for each of these merging problems, and then proceed to an algorithmic implementation. For each i, $1 \leq i \leq n/\log n$, there are two pairs: a *right* pair and a *left* pair.

Let us first describe and handle the right pair. See also Figures 3.1 and 3.2. There are three cases.

Case (i): The right match $r(b(i))$ does not exist.

Case (ii): The right match $r(b(i))$ exists and $gr(b(i)) = i + 1$.

Case (iii): The right match $r(b(i))$ exists and $gr(b(i)) > i + 1$ (satisfying the assumptions of Lemma 3.1).

In Case (i) the right pair is not defined. We now define the two subarrays of the right pair for Cases (ii) and (iii).

First subarray of a right pair: In Case (ii), the first subarray, denoted A_1, spans to the right from $a_{b(i)}$ until the end of subset i. See Figure 3.2. In Case (iii), the subarray is $A_1 = (a_{b(i)}, \ldots, a_{l(b(k))})$, where k is the index defined in Lemma 3.1. See Figure 3.1.

Second subarray of a right pair: In Case (ii), the second subarray, denoted A_2, spans to the right, from the beginning of subset $i+1$ until $a_{r(b(i))}$. See Figure 3.2. In Case (iii) the subarray is $A_2 = (a_{r(b(k))}, \ldots, a_{r(b(i))})$. See Figure 3.1.

Consider the subsequence of the first subarray that consists of all the elements of this subarray whose right match does not lie in subset i. We call it the *first subsequence.*

Observation 1a: An element of the first subarray is in the first subsequence if and only if it is smaller than all elements in subset i that are to its right.

This implies that the values of the elements in the first subsequence are monotonically increasing from left to right.

Similarly, consider the subsequence of the second subarray that consists of all the elements of this subarray for which the left match does not lie in subset $gr(b(i))$. We call it the *second subsequence.*

Observation 1b: An element of the second subarray is in the second subsequence if and only if it is smaller than all elements in subset $gr(b(i))$ that are to its left.

This implies that the values of the elements in the second subsequence are monotonically increasing from right to left.

Lemma 3.3: *The right matches of all elements in the first subsequence lie in the second subarray. (More precisely, they lie in the second subsequence.)*

Proof: Consider an element a_x in the first subsequence. Since $a_x \geq a_{b(i)}$, $r(x) \leq r(b(i))$. If $gr(b(i)) = i+1$ (Case (ii) above), then since $gr(x) \neq i$, we conclude that $r(x)$ lie in the second subarray. Otherwise, i.e., $gr(b(i)) \neq i+1$ (Case (iii) above), then since $a_x \leq a_{l(b(k))} < a_{b(k)}$ it must be that $r(x) \geq r(b(k))$. Since $r(x) \leq r(b(i))$, $r(x)$ is in the second subarray. $\square$

Lemma 3.4: *The left matches of all elements in the second subsequence (with the exception of $a_{r(b(i))}$) lie in the first subsequence.*

Lemma 3.4 is similar to Lemma 3.3 and its proof is omitted. This completes the characterization of the merging tasks.

Algorithmic implementation of the merging tasks. In Step 5.1 (respectively, Step 5.2), we find for each subset i, its right (respectively, left) pair of subarrays. Specifically, we find out whether we fall in Case (i) (by default), Case (ii) (in Step 5.1.1) or Case (iii) (in Step 5.1.2). For Case (iii), we identify the boundaries of these two subarrays, using the characterization of Lemma 3.1. In Step 6.1 we find right matches for the yet unmatched elements of the first subarray of right pairs and left matches for the yet unmatched elements of the second subarray of right pairs (i.e., right matches for the first subsequence and left matches for the second subsequence). In Step 6.2, we do the same for the subarrays of left pairs.

Step 5.1.1: Allocate a processor to each subset $1 \leq i \leq n/\log n$. If $gr(b(i)) = i+1$ (Case (ii) above), the processor determines the boundaries of the subarrays of the right pair of subset i.

Step 5.1.2: Allocate a processor to each subset $1 \leq k \leq n/\log n$. Let i be $gl(b(k))$. The processor allocated to subset k, checks whether k relates to i as defined in Lemma 3.1. Specifically, since the left match of $a_{b(k)}$ lies in subset i we need only check whether $gr(b(i)) = gr(b(k))$ in constant time. If so, the processor determines the boundaries of the subarrays of the right pair of subset i: The right boundary of the first subarray is $l(b(k))$ and the left boundary of the second subarray is $r(b(k))$.

The right matches for the subsequence of subarray A_1 and the left matches for the subsequence of subarray A_2, are found using a merging procedure. To facilitate application of merging we transform array A_1 into a non-decreasing array C_1, by taking its suffix minima. Similarly we transform A_2 into a non-increasing array C_2, by taking its prefix minima.

Step 6.1: Allocate a processor to each subset $1 \leq i \leq n/\log n$. Processor i merges arrays C_1 and C_2 (reversed) into a sorted array C. The right matches of the elements in the first subsequence are computed as follows. Let l_1 and l_2 denote the number of elements in C_1 and C_2, respectively. Let y be an element in the first subsequence. Let r be y's index within C_1 and let j be its index within C. Then, the last $j - r$ elements of C_2 are smaller than y and the first $l_2 - (j - r)$ elements of C_2 are larger than y. Thus, the right match of y is the element of A_2 indexed $(l_2 - (j - r) + 1)$. The left matches of the elements in the second subsequence are computed similarly.

Detailed characterization of a left pair, respective observations and how they lead to Steps 5.2 and 6.2 for merging such pair are similar and therefore omitted. This completes the description of the algorithm.

Throughout the presentation we argued that whenever the algorithm finds left and right matches, it does so correctly. It remains to show that the algorithm finds matches for *all* elements.

Theorem 3.5: *Each element that has not been matched to the left or to the right by the end of Step 4 will be matched, if a match exists, by either the merges of Step 6.1 or the merges of Step 6.2.*

Proof: Let a_x be an element of some subset i where $x > b(i)$. Then, a_x is matched to its left within its subset in Step 2. Our goal is to show that a_x is matched also to its right by our algorithm.

There are two possibilities depending on the subset $j = gr(x)$ of the right match of a_x.

Case (i): $j = gr(b(i))$. In this case a_x belongs to the first subarray of the right pair of subset i and thus is matched by our algorithm.

Case (ii): $j \neq gr(b(i))$. Notice that in this case the left match of $a_{b(j)}$ is in subset i. Then, a_x belongs to the left pair of subset j and thus is matched.

The case $x < b(i)$ is handled analogously. □

We conclude

Theorem 3.6: *The parallel ANSV algorithm runs in $O(\log n)$ time using $n/\log n$ processors on a CREW PRAM.*

3.3. The doubly logarithmic time algorithm

We start with a high-level description to be followed by implementation details. The high-level description emphasizes several applications of a prefix minima routine. These applications enable the improvement to doubly logarithmic time.

3.3.1. High level description

Step 1: Partition A into $m = n/\log\log n$ subsets of $\log\log n$ elements each.

Step 2: Allocate a processor to each subset and solve the ANSV problem with respect to the subset sequentially.

Step 3: Find the minimum, prefix minima and suffix minima in each subset. Let $b(i)$ denote the index of the minimum element in subset i.

Step 4: Let $B = (a_{b(1)}, \ldots, a_{b(m)})$. Solve the ANSV problem with respect to B. For this, employ the balanced doubly logarithmic tree described in Section 2, referred to as the *PS–tree* (*Prefix–Suffix tree*). The leaves of the PS–tree are the elements of B. We refer to the elements associated with the leaves of the subtree rooted at an internal node as the elements of the node. The computation is done in three substeps.

- **Step 4.1:** Accumulate the following information for each internal node of the PS–tree: (1) The minimum of its elements. (2) An array consisting of the prefix minima with respect to its elements. (3) An array consisting of the suffix minima with respect to its elements.
- **Step 4.2:** For each internal node u, consider only its minimum element $min(u)$; and find a right (and left) match for $min(u)$ among the elements of the siblings of u, if it exists. The right match is found in two rounds. In the first round we identify the sibling whose subtree includes the right match. In the second round we find the right match itself. Finding the left match is similar.
- **Step 4.3:** Find the rest of the left and right matches using parallel merging routines, similar to the logarithmic time algorithm; however, the merging routines themselves will be parallel.

Step 5: Extend the ANSV solution for B into a solution of the ANSV problem for A.

Each of the above steps takes $O(\log\log n)$ time using $n/\log\log n$ processors, and thus the whole algorithm has the same bounds.

3.3.2. Implementation

We describe the implementation of steps 1–5 presented in the high level description. For the implementation we note that Lemma 3.1 and its symmetric counterpart Lemma 3.2 do not depend on the size of the subsets into which A is partitioned and thus will be used in what follows.

Steps 1–3 need no more explanation.

Implementation of Step 4.1 Recall that the input to this step is an array $B = (b_1, b_2, \ldots, b_m)$ where $m = n/\log\log n$. Consider some internal node v of the PS–tree of height at least two. (The implementation for internal nodes of height one is trivial.) We compute two arrays P_v and S_v for v, with an entry for each element of v. These entries are the prefix minima (resp. suffix minima) with respect to the elements of v.

For this we apply the (non optimal) doubly logarithmic prefix minima algorithm of Section 2 to array B. Recall that the recursive procedure given for this algorithm effectively constructs a PS-tree whose leaves are the elements of B and in addition computes array P_v for all nodes v of the PS-tree. This takes $O(\log\log n)$ time and uses $m = n/\log\log n$ processors. The array S_v for all nodes v is computed similarly, using the algorithm for the suffix minima problem.

Implementation of Step 4.2 The computation described with respect to u is done by the processors allocated to the parent of u. Suppose that v is the parent of u. Let r be the number of elements (i.e. leaves) of v; let $w_1, \ldots, w_{\sqrt{r}}$ be the children of v and $M = (min(w_1), \ldots, min(w_{\sqrt{r}}))$. Allocate r processors to v and apply the constant time ANSV algorithm of Section 2 to M. Now, if the right match in M of some $min(w_k)$ is $min(w_l)$ we conclude that the right match of $min(w_k)$ in v lies within the leaves of w_l. Over all nodes of the PS-tree, the above application of the ANSV algorithm runs in constant time and uses $m \cdot \log\log n = n$ processors, and thus can be simulated in $O(\log\log n)$ time using $n/\log\log n$ processors. This finishes the first round of Step 4.2. For the second round, we need to find the right match of $min(w_k)$ within the leaves of w_l. That is, we need to find the leftmost element in w_l that is smaller than $min(w_k)$. This can be done using the 1-color minimization algorithm in constant time using $\sqrt{r}$ processors. This sums to r processors over all $min(w_k)$, $1 \le k \le \sqrt{r}$. Over all nodes of the PS-tree, this takes constant time using n processors (and thus can be simulated in $O(\log\log n)$ time using $n/\log\log n$ processors).

Implementation of Step 4.3 The crux of the implementation of Step 4.3 is the way the merging procedures that find the right (or left) match for every element b_i of B are identified.

For each element b_i, the merging procedure that finds its right match is associated with the lowest level node v in the PS–tree such that both b_i and the right match of b_i are elements of v. Thus we need only identify node v. This is done in constant time and $\log\log n$ processors, one processor for each ancestor of b_i, or constant time and n processors for all elements in B using the following observation.

Observation 2: Let u be an ancestor of b_i. The right match of b_i is not an element of u if and only if b_i is smaller than all elements in u that occur to its right. Analogously, the left match of b_i is not an element of u if and only if b_i is smaller than all elements in u that occur to its left.

Consider again some node v of the PS-tree and let its children be $w_1, \ldots, w_{\sqrt{r}}$ as before. We now find the right matches of all elements x, such that x is an element of w_k for some k, $1 \leq k \leq \sqrt{r}$, and the right match of x is an element of v but not of w_k; (in view of Observation 2, node v will take care of finding the right match for element x;) this is done in parallel at each node of the tree.

Following Step 4.2, we have the left and right matches for the minimum element with respect to each of the $\sqrt{r}$ children of v, provided that this match (right or left) is located within the elements of v. This is essentially the same situation as after Step 4 in Section 3.2, where the children of v here play the role of the subsets of Section 3.2. Thus, using similar ideas to steps 5 and 6 in Section 3.2 we can proceed. Steps 5.1 and 5.2 of the algorithm given in Section 3.2 stay exactly as they are. The only change is in steps 6.1 and 6.2 where the merging of pairs of subarrays will be performed in parallel rather than by a serial merging algorithm. Specifically, the right pair (as defined in section 3.2) of a subset consists of two subarrays: the values of the left subarray are obtained by suffix-minima computation and the values of the right subarray are obtained by prefix-minima computation. Each subarray contains no more than $\sqrt{r}$ elements. We merge each pair of subarrays using the optimal doubly logarithmic merging algorithm of [Kru83]. This is done in $O(\sqrt{r})$ operations and at most $O(\log\log n)$ time for each pair of subarrays. Over all children of v this takes $O(\log\log n)$ time using $r/\log\log n$ processors and thus $O(\log\log n)$ time and $n/\log\log n$ processors over all nodes of the PS-tree. Analogously, we handle the left pair of each subset.

Implementation of Step 5. Assign a processor to each subset i, $1 \leq i \leq m$. Recall that $b(i)$ is the index of the minimum element in subset i; and that Step 4 computed the left and right matches of each element of B within B. The goal of Step 5 is to find left and right matches for each element of A (within A).

Let $b(j)$ be the index of the right match of $a_{b(i)}$ within B. Then, the right match of $a_{b(i)}$ in

A is the leftmost element in subset j that is smaller than b_i and can be found in $O(\log \log n)$ time using the processor of subset i. In total, $n/\log \log n$ processors are used. Handling the left match of $a_{b(i)}$ is similar.

Finally, we show how to find matches in A for *all* elements in A. This is similar to the implementation of steps 5 and 6 in Section 3.2. However since the subsets are of size $\log \log n$ each, the computation will take $O(\log \log n)$ time using one processor for each subset or $O(\log \log n)$ time using $n/\log \log n$ processors overall.

We showed,

Theorem 3.7: *The parallel ANSV algorithm runs in* $O(\log \log n)$ *time using* $n/\log \log n$ *processors on a CRCW PRAM.*

4. Triangulating a monotone polygon

A polygonal chain $Q = (q_1, \ldots, q_m)$ is said to be *monotone* if the vertices $q_1, \ldots, q_m$ are in increasing (or decreasing) order by the x-coordinate. A *monotone polygon* is composed of two monotone polygonal chains: the *upper* and *lower* chains. We assume without loss of generality that the upper chain goes from the vertex with minimum x-coordinate to the vertex with maximum x-coordinate. A *one-sided monotone polygon (OSMP),* is a monotone polygon whose upper (or lower) chain is a straight line. We call this straight line the *distinguished* edge of an OSMP.

4.1. The upper bound

Let $P = (v_0, v_1, \ldots, v_{n-1})$ be a monotone polygon. The algorithm for triangulating a monotone polygon has two stages. In the first stage we decompose P into one-sided monotone polygons (OSMP's). In the second stage we show how to triangulate an OSMP.

4.1.1. Decomposition into one-sided monotone polygons Assume for simplicity that no two vertices of P have the same x coordinate. The general idea for achieving decomposition into one-sided monotone polygons (OSMP's) is due to Aggarwal et al. [ACG+85] and Goodrich [Goo89].

Step 1: Merge the vertices of the lower chain and the vertices of the upper chain, according to their x-values using the parallel merging algorithm of Kruskal [Kru83]. This takes $O(\log \log n)$ time using $n/\log \log n$ processors.

Step 2: For each edge (v_i, v_{i+1}) of the lower chain, the segment of the upper chain consisting of those points whose x coordinates lie between v_i and v_{i+1} together with the edge (v_i, v_{i+1}) itself (as the distinguished edge) form a one-sided monotone polygon. Given the merged array of Step 1, the OSMP of an edge (v_i, v_{i+1}) of the lower chain is described by the subarrays of the output of Step 1 that extend from v_i to v_{i+1} (if not empty).

Step 3: For each edge (v_i, v_{i+1}) of the upper chain, the segment of the lower chain consisting of those points whose x coordinates lie between v_i and v_{i+1} together with the edge (v_i, v_{i+1}) itself (as the distinguished edge) form a one-sided monotone polygon. (The representation of an OSMP is similar to the previous step.)

4.1.2. Triangulating a one-sided monotone polygon We show how to triangulate each of the OSMP's. First, we show how to allocate the $n/\log\log n$ processors among the vertices. Then, we show how the triangulation is done using these processors. Consider the merged array of Section 4.1.1. The $n/\log\log n$ processors are partitioned among consecutive subarrays of $\log\log n$ vertices each. Consider, first, OSMP's whose entire vertices fall in the subarray of a single processor. The number of vertices of each such OSMP is at most $\log\log n$. In parallel, each processor applies a linear time serial algorithm (by the algorithms of [GJPT78] or [FM84]) to each such OSMP in its subarray. This takes $O(\log\log n)$ time.

Now, allocate the processors to OSMP's that extend into more than one subarray, so that a ratio of $\Omega(\frac{1}{\log\log n})$ between processors and number of vertices is maintained for each OSMP.

Let $Q = (u_0, u_1, \ldots, u_{m-1})$ be a one-sided monotone polygon. We assume without loss of generality that all vertices are above or on the distinguished edge (u_0, u_{m-1}), and that $x(u_0) < x(u_{m-1})$ and $y(u_0) \leq y(u_{m-1})$.

Step 1: Rotate Q so that the distinguished edge is be parallel to the x axis without moving vertex u_0. The rotation is done by an angle with absolute value smaller than $\frac{\pi}{2}$.

Step 2: For each $0 \leq i \leq m-1$, add the following edges: (i) An edge between u_i and u_j, where $j < i$ is the largest index for which u_j is *not higher* than u_i. (ii) An edge between u_i and u_k, where $k > i$ is the smallest index for which u_k is *lower* than u_i.

Step 2 applies the ANSV algorithm of the previous section as follows. The array A for the ANSV procedure is an array of pairs of numbers. $A = ((y(u_0), 0), (y(u_1), 1), \ldots, (y(u_{m-1}), m-1))$. The comparisons during the algorithm are done lexicographically. This implements Step 2.

Each of steps 1 and 2 above takes $O(\log\log n)$ time using $n/\log\log n$ processors.

Using Yap [Yap88], we conclude:

Theorem 4.1: *The above algorithm triangulates a monotone polygon in $O(\log\log n)$ time using $n/\log\log n$ processors.*

4.2. Lower bound for triangulating a monotone polygon

In this subsection we prove the following theorem:

Theorem 4.2: *Any parallel algorithm for triangulating a monotone polygon with n vertices on a CRCW PRAM that uses $O(n\log^c n)$ processors, for some constant $c>0$, requires $\Omega(\log\log n)$ time.*

Proof: We reduce the following merging problem to the problem of triangulating a monotone polygon. The input to the merging problem are two sorted lists of size n, with all $2n$ elements distinct. The output is the merged list. Using a technique of [MW85], [SV90] show that this merging problem requires $\Omega(\log\log n)$ time, if $O(n\log^c n)$ processors, for some constant $c>0$, are available. Theorem 4.2 follows. □

Let $A=(a_0,a_1,\ldots,a_{n-1})$, $B=(b_0,b_1,\ldots,b_{n-1})$ be the lists we wish to merge both ordered from smallest to largest. We assume without loss of generality that
(i) $a_0<b_0$; (ii) $a_{n-1}>b_{n-1}$; (iii) $a_0>0$.

[Insert Figure 4.1 here.]

Let $\ell=a_{n-1}^2$. Define the following monotone polygon P. (See also Figure 4.1.) The upper chain of the polygon is

$$((a_0,0),(\tfrac{a_0+a_1}{2},1),(a_1,0),\ldots,(\tfrac{a_{n-2}+a_{n-1}}{2},1),(a_{n-1},0)).$$

The lower chain is

$$((a_0,0),(\frac{a_0+b_0}{2},-1),(b_0,\frac{\sqrt{b_0}}{\ell}),(\frac{b_0+b_1}{2},-1),(b_1,\frac{\sqrt{b_1}}{\ell}),\ldots$$
$$\ldots,(\frac{b_{n-2}+b_{n-1}}{2},-1),(b_{n-1},\frac{\sqrt{b_{n-1}}}{\ell}),(\frac{b_{n-1}+a_{n-1}}{2},-1),(a_{n-1},0)).$$

We explain this structure.

Let a_j be some element of the first list, $0\le j<n-1$. Let $b_i,b_{i+1},\ldots,b_r$ be all elements of the second list whose values are larger than a_j but smaller than a_{j+1}. Consider the triangle induced by the three successive vertices of the upper chain: $(a_j,0),(\frac{a_j+a_{j+1}}{2},1),(a_{j+1},0)$.

Proposition 4.3: *Vertices* $(b_i, \frac{\sqrt{b_i}}{\ell}), (b_{i+1}, \frac{\sqrt{b_{i+1}}}{\ell}), \ldots, (b_r, \frac{\sqrt{b_r}}{\ell})$ *lie inside this triangle.*

Definition 1: *Vertices* u *and* v *are said to be (mutually)* visible *in the presence of segments in the plane, if the segment* $\overline{uv}$ *does not intersect any other segment.*

Proposition 4.4: *Let* $i < k < r$.

1. *The only vertices that are visible from vertex* $(b_k, \frac{\sqrt{b_k}}{\ell})$ *(in addition to its adjacent vertices* $(\frac{b_{k-1}+b_k}{2}, -1)$ *and* $(\frac{b_k+b_{k+1}}{2}, -1)$*) are* $(b_{k-1}, \frac{\sqrt{b_{k-1}}}{\ell})$, $(b_{k+1}, \frac{\sqrt{b_{k+1}}}{\ell})$ *and* $(\frac{a_j+a_{j+1}}{2}, 1)$.
2. *The only vertices that are visible from* $(b_i, \frac{\sqrt{b_i}}{\ell})$ *(in addition to its adjacent vertices) are* $(a_j, 0)$, $(b_{i+1}, \frac{\sqrt{b_{i+1}}}{\ell})$ *and* $(\frac{a_j+a_{j+1}}{2}, 1)$.
3. *The only vertices that are visible from* $(b_r, \frac{\sqrt{b_r}}{\ell})$ *(in addition to its adjacent vertices) are* $(b_{r-1}, \frac{\sqrt{b_{r-1}}}{\ell})$, $(a_{j+1}, 0)$, *and* $(\frac{a_j+a_{j+1}}{2}, 1)$.

Proposition 4.5: *Vertex* $(a_j, 0)$ *is above the straight line connecting* $(\frac{b_{i-1}+b_i}{2}, -1)$ *to* $(b_i, \frac{\sqrt{b_i}}{\ell})$, *and vertex* $(a_{j+1}, 0)$ *is above the straight line connecting* $(b_r, \frac{\sqrt{b_r}}{\ell})$ *to* $(\frac{b_r+b_{r+1}}{2}, -1)$.

Proposition 4.6: *No three vertices from the sequence* $(b_i, \frac{\sqrt{b_i}}{\ell}), (b_{i+1}, \frac{\sqrt{b_{i+1}}}{\ell}), \ldots, (b_r, \frac{\sqrt{b_r}}{\ell})$ *lie on the same straight line.*

We first show how these propositions imply the reduction. From Propositions 4.3 and 4.5 we conclude that the polygon P is simple. That is, no two edges of the polygon intersect. The polygon is clearly monotone. Consider any triangulation T of P. Let a_j and $b_i, \ldots, b_r$ be as above.

Claim: Each vertex $(b_k, \frac{\sqrt{b_k}}{\ell})$, for $i \leq k \leq r$, must be connected by an edge in T to $(\frac{a_j+a_{j+1}}{2}, 1)$.

Proof of Claim: We prove the claim for $i < k < r$. The proofs for $k = i$ and $k = r$ are similar. Observe that the vertex $(b_k, \frac{\sqrt{b_k}}{\ell})$ must be connected in T to at least one of the vertices visible from it. Now, suppose that $(b_k, \frac{\sqrt{b_k}}{\ell})$ is not connected to $(\frac{a_j+a_{j+1}}{2}, 1)$ in T. Therefore, it must be connected to either $(b_{k-1}, \frac{\sqrt{b_{k-1}}}{\ell})$ or $(b_{k+1}, \frac{\sqrt{b_{k+1}}}{\ell})$. Assume without loss of generality that it is connected to $(b_{k+1}, \frac{\sqrt{b_{k+1}}}{\ell})$. Consider the triangle in T lying above the edge connecting these two vertices. From Proposition 4.6 we conclude that the edge must also be an edge of the triangle, and cannot just be a segment of such an edge. (REMARK: In our definition of triangulation we allow degeneracy in the following sense. If several vertices lie on the same straight line then the segment connecting these vertices may be an edge of the triangulation.)

The third vertex in this triangle must be above and also must be visible from both $(b_k, \frac{\sqrt{b_k}}{\ell})$ and $(b_{k+1}, \frac{\sqrt{b_{k+1}}}{\ell})$. The only such vertex is $(\frac{a_j+a_{j+1}}{2}, 1)$. The Claim follows. $\square$

The edge of a triangulation T that connects a vertex $(b_k, \frac{\sqrt{b_k}}{\ell})$, for $i \leq k \leq r$, with $(\frac{a_j+a_{j+1}}{2}, 1)$ reveals that the largest element of the first list that is smaller than b_k is a_j. This yields the ranking of all elements of the second list for the merging problem. By substituting the role played by lists A and B in the reduction, we can obtain ranking for all elements in A. This ends the reduction of the merging problem into the triangulation problem.

It remains to verify the propositions.

Proof of Proposition 4.3: We show that the vertices $(b_i, \frac{\sqrt{b_i}}{\ell}), (b_{i+1}, \frac{\sqrt{b_{i+1}}}{\ell}), \ldots, (b_r, \frac{\sqrt{b_r}}{\ell})$ are under the line connecting $(a_j, 0)$ and $(\frac{a_j+a_{j+1}}{2}, 1)$. Showing that these vertices are also under the line connecting $(\frac{a_j+a_{j+1}}{2}, 1)$ and $(a_{j+1}, 0)$ is similar. Consider the straight line connecting $(a_j, 0)$ and $(\frac{a_j+a_{j+1}}{2}, 1)$. The y-value of this line at $x = b_k$, for $i \leq k \leq r$, is at least $\frac{2}{\sqrt{\ell}}$. This is since $a_j \geq 0$, $\frac{a_{j+1}-a_j}{2} \leq \frac{\sqrt{\ell}}{2}$ and $b_k - a_j \geq 1$. However since $\frac{\sqrt{b_k}}{\ell} < \frac{2}{\sqrt{\ell}}$, vertex $(b_k, \frac{\sqrt{b_k}}{\ell})$ is under the line connecting $(a_j, 0)$ and $(\frac{a_j+a_{j+1}}{2}, 1)$. $\square$

Proof of Proposition 4.4: We observe that all vertices $(b_k, \frac{\sqrt{b_k}}{\ell})$ lie on a positive convex curve. That is, the segment connecting two of these vertices must lie under the curve. Each of the three items under this proposition follows from this observation and Proposition 4.3. $\square$

Proof of Proposition 4.5: We observe that the y-value of the line connecting $(\frac{b_{i-1}+b_i}{2}, -1)$ and $(b_i, \frac{\sqrt{b_i}}{\ell})$ at $x = a_j$ is negative. The first claim in Proposition 4.5 follows. The proof of the second claim is similar. $\square$

Proof of Proposition 4.6: Immediate from the fact that the vertices $(b_i, \frac{\sqrt{b_i}}{\ell}), (b_{i+1}, \frac{\sqrt{b_{i+1}}}{\ell}), \ldots, (b_r, \frac{\sqrt{b_r}}{\ell})$ lie on the curve $y = \frac{\sqrt{x}}{\ell}$. $\square$

Thus, we have shown a reduction from the problem of merging two sorted lists into the problem of triangulating a monotone polygon and thereby proved Theorem 4.2.

5. Range minimum search

The *range minimum searching* problem is defined as follows. Preprocess an array of n real numbers $A = (a_1 \ldots, a_n)$ so that on-line range minimum queries of the form "Which is the minimum element in the sub-array $A_{i,j} = (a_i, \ldots, a_j)$?" for any $1 \leq i \leq j \leq n$, can be answered in constant time. (Such queries are denoted $MIN(i, j)$.)

5.1. The logarithmic time algorithm

In this section we describe an optimal logarithmic time CREW PRAM preprocessing algorithm for the range minimum searching problem. Without loss of generality we assume that the elements in A are distinct.

The preprocessing algorithm is based on the sequential algorithm of [GBT84]. First, we recollect this algorithm.

The preprocessing algorithm of [GBT84] uses the *Cartesian tree* data structure introduced in [Vui80].

Definition 2 ([Vui80]): *The* Cartesian tree *for an array* $A = (a_1, \ldots, a_n)$ *of* n *distinct real numbers is a binary tree with vertices labeled by the numbers. The root has label* a_m*, where* $a_m = \min\{a_1, \ldots, a_n\}$*. Its left subtree is a Cartesian tree for* $A_{1,m-1} = (a_1, \ldots, a_{m-1})$*, and its right subtree is a Cartesian tree for* $A_{m+1,n} = (a_{m+1}, \ldots, a_n)$*. (The tree for an empty sub-array is the empty tree.)*

REMARK: Throughout this section, we identify the nodes of a Cartesian tree with their labels.

Preprocessing: Construct the Cartesian tree for A. This can be done in linear time as shown in [GBT84]. Apply the linear time preprocessing algorithm for answering LCA queries in trees given in [HT84] (see also, [SV88, BV89]).

Query processing: From the recursive definition of the Cartesian tree it readily follows that $MIN(i, j)$ is the LCA of a_i and a_j in the Cartesian tree. Thus, each range minimum query can be answered in constant time by answering the corresponding LCA query in the Cartesian tree.

To parallelize the above algorithm we show how to construct the Cartesian tree optimally in logarithmic time. The parallel algorithm is different from the sequential algorithm given by [GBT84] which does not seem to be amenable for efficient parallelism. We construct the Cartesian tree using a proper algorithm for the ANVS problem, and then apply the optimal logarithmic time parallel preprocessing algorithms for answering LCA queries of [SV88] or [BV89]. This would give a logarithmic time preprocessing algorithm for the range minimum searching problem.

Let a_i be an element in A. Recall that the left (resp. right) match of a_i is the nearest element to its left (resp. right) with a smaller value, if such exists.

Claim: The parent of a vertex a_i in the Cartesian tree for A is the *larger* among its left and right matches in A, if such exist. The vertex a_i is a right child if its parent is its left match, and a left child otherwise.

Proof: Our proof focuses on a representative case only. Assuming the left and right matches of a_i exist, denote the left match by a_ℓ and the right match by a_r. Suppose that $a_r < a_\ell$. Consider the definition of Cartesian tree with respect to the subtree rooted at a_ℓ. Its right subtree must be the Cartesian tree for subarray $(a_{\ell+1}, a_{\ell+2}, \ldots, a_{r-1})$. The minimum over this subarray is a_i and therefore a_i is the root of this Cartesian tree and the right child of a_ℓ. The proofs for the cases where $a_r > a_\ell$ and where either a_ℓ or a_r do not exist are similar. □

To conclude the parallel preprocessing algorithm we compute the left and right matches for each element a_i in A using our ANSV algorithm.

5.2. The doubly logarithmic time algorithm

The doubly logarithmic time CRCW PRAM algorithm is based on the $O(n \log n)$ time preprocessing algorithm given in [AS87]. Let us recall this algorithm.

Preprocessing: Without loss of generality assume that n is a power of two. Construct a complete binary tree T with n leaves, and associate the elements of A with the leaves of T, in order. For each vertex v of T having leaves $L_v = (a_{i2^k+1}, \ldots, a_{(i+1)2^k})$ in its rooted subtree, compute the prefix minima and suffix minima with respect to L_v. That is, compute $P_v(q) = \min\{a_{i2^k+1}, \ldots, a_{i2^k+q}\}$, and $S_v(q) = \min\{a_{(i+1)2^k-q+1}, \ldots, a_{(i+1)2^k}\}$, for $q = 1, \ldots, 2^k$.

Query processing: The query $MIN(i,j)$ is answered as follows. Find w, the LCA of a_i and a_j in T. Let v and u be the left and right children of w, respectively. Let ℓ be the index of the leftmost leaf in the subtree of u. Then, $MIN(i,j) = \min\{S_v(\ell - i), P_u(j - \ell + 1)\}$. The LCA of any two vertices in T can be found using the inorder numbering of T, as shown in [HT84].

The above sequential algorithm is easily parallelized to run in $O(\log \log n)$ time using $n \log n$ processors. This is done by allocating to each vertex v of T having ℓ leaves ℓ processors, and then computing the prefix and suffix minima using these processors.

The number of processors can be reduced by cascading the logarithmic time algorithm given above as follows.

Step 1: Partition the array A into blocks of size $\log n \log \log n$. (The last block maybe of a smaller size.) For each block, perform the logarithmic time algorithm given in the previous subsection. This takes $O(\log\log n)$ time and $\log n$ processors for each block, totaling $n/\log\log n$ processors.

Step 2: Let B be the array consisting of the minimum elements in each block. Perform the above doubly logarithmic time algorithm on the array B. Since the size of B is $O(n/(\log\log n \log n))$, this requires $n/\log\log n$ processors.

Given this preprocessing a query $MIN(i,j)$ is answered as follows. If a_i and a_j are in the same block then $MIN(i,j)$ can be retrieved using the preprocessing done in Step 1. Otherwise, suppose that a_i and a_j are in blocks indexed $b(i)$ and $b(j)$, respectively. Let $right(i)$ be the index in A of the rightmost element in a_i's block, and let $left(j)$ be the index in A of the leftmost element in a_j's block. Compute $MIN(i, right(i))$ and $MIN(left(j), j)$ using the preprocessing done in Step 1. If $b(i)+1 < b(j)$, then compute the $MIN(right(i)+1, left(j)-1)$ using the preprocessing done in Step 2. (Notice that $MIN(right(i)+1, left(j)-1)$ is the minimum element in blocks $b(i)+1, \ldots, b(j)-1$.) Clearly,

$$MIN(i,j) = \min\{MIN(i, right(i)), MIN(right(i)+1, left(j)-1), MIN(left(j), j)\}.$$

6. Reconstructing a binary tree from its traversals

The input to this problem is a binary tree $T(V,E)$ where $V = \{1,\ldots,n\}$, described by two arrays:

1. An array $P = (p_1, p_2, \ldots, p_n)$ containing the numbering of the nodes in a preorder traversal. (That is, the preorder number of vertex i is p_i.)
2. An array $I = (i_1, i_2, \ldots, i_n)$ containing the numbering of the nodes in inorder traversal.

The output is the binary tree in the following format. For each node $1 \le i \le n$ pointers to its left child and its right child, if such exist, or a null pointer otherwise.

First, compute the arrays P^{-1} and I^{-1}. The array P^{-1} is an array containing in each location $1 \le i \le n$ the number j for which $P(j) = i$. The array I^{-1} is defined in the same way with respect to I.

Now, compute an array $\overline{P} = (x_1, x_2, \ldots, x_n)$ where $x_i = P(I^{-1}(i))$ for $1 \le i \le n$. In words, $\overline{P}$ is an array containing the preorder of the vertices according to their inorder. This array can be constructed in constant time using n processors in the obvious way.

Lemma 6.1: *Let $1 \le i \le n$ and let $j = P^{-1}(P(i)+1)$ (in words, j is the node visited immediately after i in the preorder traversal). If $I(j) < I(i)$ then j is the left child of i. Otherwise, i does not have a left child.*

Proof: By our assumption $P(j) = P(i) + 1$. Properties of the preorder traversal imply that there are three possibilities:

Possibility (i): j is the left child of i.

Possibility (ii): j is right child of i and i does not have a left child.

Possibility (iii): j is the right child of some ancestor k of i, i is in the left subtree of k and i does not have any children.

In Possibility (i) $I(j) < I(i)$. In Possibilities (ii) and (iii) $I(j) > I(i)$. The Lemma follows. □

Using Lemma 6.1 it is straightforward to find the left child for each node if it exists (or indicate it does not exist) in constant time using a single processor. Thus, we can find the left child of all nodes in the tree in $O(\log\log n)$ time using $n/\log\log n$ processors.

Lemma 6.2: *Let $1 \le j \le n$ be the inorder number of some node v. Assume that $\overline{P}(j) \neq 1$ (i.e., v is not the root of T) and that v is not a left child of any node in T (so v must be a right child of some node in T). Let $\ell(j)$ denote the index of the left match (as defined in the ANSV problem) of x_j in $\overline{P}$. Then, v is the right child of $I^{-1}(\ell(j))$ (i.e., the node whose inorder number is $\ell(j)$).*

Proof: Consider the inorder traversal. Between the visit at the parent of v and the visit at v itself, we traverse the left subtree of v. The preorder numbering of each node in this left subtree is larger than $\overline{P}(j)$. On the other hand, the preorder numbering of the parent is less than $\overline{P}(j)$. Thus if we move in the array $\overline{P}$ from entry j to the left, the first index for which $\overline{P}$ is smaller than $\overline{P}(j)$ yields the parent of v, and v is the right child of this parent. □

By Lemma 6.2, the ANSV algorithm can be used for finding the parent of each node which is neither a left child of any node, nor the root of T. Using our ANSV algorithm we conclude:

Theorem 6.3: *The algorithm for reconstructing a binary tree from its preorder and inorder traversals runs in $O(\log\log n)$ time using $n/\log\log n$ processors on a CRCW PRAM.*

A similar algorithm can be designed for reconstructing a binary tree from its postorder and inorder traversals.

7. Parentheses matching

The input to this problem is a legal sequence of left and right parentheses and the level of nesting for each parenthesis. The output is the left mate of each right parenthesis.

We begin with the following simple observation:

Observation: Let i be the level of nesting of some left parenthesis and let j be the level of nesting of its right mate. Then, $i = j$ and the nesting levels of all left and right parentheses between them are larger.

Let $a_1, \ldots, a_n$ be the levels of nesting of the sequence. Define $A = ((a_1, 1), (a_2, 2), \ldots, (a_n, n))$. We now apply the ANSV algorithm with respect to A where comparisons are made lexicographically and take for each right parenthesis its left match. We conclude:

Theorem 7.1: *The algorithm for the parentheses matching problem runs in $O(\log\log n)$ time using $n/\log\log n$ processors on a CRCW PRAM.*

References

[ACG+85] A. Aggarwal, B. Chazelle, L. Guibas, C. O'Dunlaing, and C. Yap. Parallel computational geometry. In *Proc. of the 26th IEEE Annual Symp. on Foundation of Computer Science*, pages 468–477, 1985.

[AG86] M.J. Atallah and M.T. Goodrich. Efficient plane sweeping in parallel. In *Proc 2nd ACM Symp. on Computational Geometry*, pages 216–225, 1986.

[ALV90] A. Amir, G.M. Landau, and U. Vishkin. Efficient pattern matching with scaling. In *Proc. of the 1st ACM-SIAM Symp. on Discrete Algorithms*, pages 344–357, 1990.

[AMW89] R.J. Anderson, E.W. Mayr, and M.K. Warmuth. Parallel approximation algorithms for bin packing. *Information and Computation*, 82:262–277, 1989.

[AS87] N. Alon and B. Schieber. Optimal preprocessing for answering on-line product queries. Technical Report TR 71/87, The Moise and Frida Eskenasy Inst. of Computer Science, Tel Aviv University, 1987.

[BBG+89] O. Berkman, D. Breslauer, Z. Galil, B. Schieber, and U. Vishkin. Highly-parallelizable problems. In *Proc. of the 21st Ann. ACM Symp. on Theory of Computing*, pages 309–319, 1989.

[BH85] A. Borodin and J.E. Hopcroft. Routing, merging, and sorting on parallel models of computation. *J. Computer and System Sciences*, 30:130–145, 1985.

[BLSZ87] H.A. Burgdorff, S. Lajodia, F.N. Springsteel, and Y. Zalcstein. Alternative methods for the reconstruction of trees from their traversals. *Bit*, 27:134–140, 1987.

[BV85] I. Bar-On and U. Vishkin. Optimal parallel generation of a computation tree form. *ACM Trans. on Prog. Lang. and Systems*, 7:348–357, 1985.

[BV89] O. Berkman and U. Vishkin. Recursive *-tree parallel data-structure. In *Proc. of the 30th IEEE Annual Symp. on Foundation of Computer Science*, pages 196–202, 1989. To appear in *SIAM J. Comput.*

[CV89] R. Cole and U. Vishkin. Faster optimal parallel prefix sums and list ranking. *Information and Computation*, 81(3):334–352, 1989.

[EG88] D. Eppstein and Z. Galil. Parallel algorithmic techniques for combinatorial computation. *Ann. Rev. Comput. Sci.*, 3:233–283, 1988.

[FM84] A. Fournier and D.Y. Montuno. Triangulating simple polygons and equivalent problems. *ACM Trans. on Graphics*, 3:153–174, 1984.

[FRW88] F.E. Fich, P.L. Ragde, and A. Wigderson. Relations between concurrent-write models of parallel computation. *SIAM J. Comput.*, 17:606–627, 1988.

[GBT84] H.N. Gabow, J.L. Bentley, and R.E. Tarjan. Scaling and related techniques for geometry problems. In *Proc. of the 16th ACM Symp. on Theory of Computing*, pages 135–143, 1984.

[GJPT78] M.R. Garey, D.S. Johnson, F.P. Preparata, and R.E. Tarjan. Triangulating a simple polygon. *Information Processing Letters*, 7:175–179, 1978.

[Goo89] M.T. Goodrich. Triangulating a polygon in parallel. *J. Algorithms*, 10:327–351, 1989.

[HT84] D. Harel and R.E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.

[KLP89] Z.M. Kedem, G.M. Landau, and K.V. Palem. Optimal parallel prefix-suffix matching algorithm and applications. In *Proc. 1st ACM Symposium on Parallel Algorithms and Architectures*, pages 388–398, 1989.

[Knu73] D.E. Knuth. *The art of computer programming*, volume 1. Addison-Wesley,, 2 edition, 1973.

[KR90] R.M. Karp and V. Ramachandran. A survey of parallel algorithms for shared-memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume 1. MIT Press/Elsevier, 1990.

[Kru83] C.P. Kruskal. Searching, merging, and sorting in parallel computation. *IEEE Trans. on Comp*, C-32:942–946, 1983.

[LF80] R.E. Ladner and M.J. Fischer. Parallel prefix computation. *J. Assoc. Comput. Mach.*, 27:831–838, 1980.

[MW85] F. Meyer auf der Heide and A. Wigderson. The complexity of parallel sorting. In *Proc. of the 26th IEEE Annual Symp. on Foundation of Computer Science*, pages 532–540, 1985.

[RV88] V.L. Ramachandran and U. Vishkin. Efficient parallel triconnectivity in logarithmic parallel time. In *Proc. of AWOC 88, Lecture Notes in Computer Science* No. 319, pages 33–42. Springer-Verlag, 1988.

[Sch87] B. Schieber. *Design and analysis of some parallel algorithms.* PhD thesis, Dept. of Computer Science, Tel Aviv Univ., 1987.

[Sto75] H.S. Stone. Parallel tridiagonal equation solvers. *ACM Transactions on Mathematical Software*, 1(2):289–307, 1975.

[SV81] Y. Shiloach and U. Vishkin. Finding the maximum, merging, and sorting in a parallel computation model. *J. Algorithms*, 2:88–102, 1981.

[SV88] B. Schieber and U. Vishkin. On finding lowest common ancestors: simplification and parallelization. *SIAM J. Comput.*, 17(6):1253–1262, 1988.

[SV90] B. Schieber and U. Vishkin. Finding all nearest neighbors for convex polygons in parallel: a new lower bound technique and a matching algorithm. *Discrete Applied Math*, 29:97–111, 1990.

[Val75] L.G. Valiant. Parallelism in comparison problems. *SIAM J. Comput.*, 4:348–355, 1975.

[Vis91] U. Vishkin. Structural parallel algorithmics. In *Proc. of 18th ICALP, Lecture Notes in Computer Science*, No. 510. Springer–Verlag, 1991.

[Vui80] J. Vuillemin. A unified look at data structures. *Communications of the ACM*, 23:229–239, 1980.

[Yao82] A.C. Yao. Space-time tradeoff for answering range queries. In *Proc. of the 14th ACM Symp. on Theory of Computing*, pages 128–136, 1982.

[Yap88] C. Yap. Parallel triangulation of a polygon in two calls to the trapezoidal map. *Algorithmica*, 3:279–288, 1988.