



ELSEVIER

Discrete Applied Mathematics 57 (1995) 11–28

**DISCRETE
APPLIED
MATHEMATICS**

Almost fully-parallel parentheses matching

Omer Berkman^{a, 1, 2}, Uzi Vishkin^{b, c, d, *, 2}^a*Department of Computer Science, King's College London, The Strand, London WC2R 2LS, UK*^b*University of Maryland Institute for Advanced Computer Studies (UMIACS), College Park, MD 20742, USA*^c*Department of Electrical Engineering, University of Maryland, College Park, MD, USA*^d*Tel Aviv University, Tel Aviv, Israel*

Received 28 June 1991; revised 27 July 1993

Abstract

The *parentheses matching problem* is considered. Suppose we are given a balanced sequence of parentheses and wish to find for each parenthesis its mate. Assuming the levels of nesting of each parenthesis are given, we present an algorithm that runs in $O(\alpha(n))$ time using an optimal number of processors (where $\alpha(n)$ is the inverse of Ackermann's function) on the CRCW PRAM. Without this assumption the running time becomes $O(\log n / \log \log n)$.

1. Introduction

The parentheses matching problem is defined as follows. An array containing a balanced sequence of n parentheses is given. For each parenthesis we want to find its mate. Assuming the levels of nesting of each parenthesis are given, we present an algorithm that runs in $O(\alpha(n))$ time using an optimal number of processors. Without this assumption the running time becomes $O(\log n / \log \log n)$ using an optimal number of processors.

Quite a few algorithms (for several different problems) rely on a routine for the parentheses matching problem. Perhaps the most notable of these algorithms are algorithms for expression evaluation. Parallel parentheses matching has also received considerable attention. For instance, see [1, 2, 5, 8, 9, 14, 17, 18, 20, 21]. Algorithms that treat the case where the level of nesting of each parenthesis is given are those of Berkman et al. [5] and Schieber [18]. Both run in $O(\log \log n)$ time using an optimal number of processors if levels of nesting of each parenthesis is given. We note that it is easy to reduce the parity problem to the parentheses matching problem in constant time (the reduction is given for completeness in Section 3) and thus the lower bound of

* Corresponding author.

¹ Part of the work was carried out while the author was at UMIACS and at Tel Aviv University.

² Partially supported by NSF grants CCR-8906949 and 9111348.

$\Omega(\log n / \log \log n)$ time for parity using a polynomial number of processors [3] implies the same lower bound for parentheses matching if levels of nesting are not provided. However, prefix sums (by which levels of nesting are computed) is a basic and general technique. One of the aims of this paper is to investigate what is the added difficulty of parentheses matching beyond the use of this basic technique.

The model of parallel computation which is used in this paper is the concurrent-read concurrent-write (CRCW) parallel random access machine (PRAM). We assume that several processors may attempt to write at the same memory location only if they are seeking to write the same value (the so-called, common-CRCW PRAM), as in [19]. We use the weakest common-CRCW PRAM model, in which only concurrent writes of the value one are allowed.

A *work-optimal* (or, simply, *optimal*) parallel algorithm is an algorithm whose work (i.e., time–processor product) complexity matches the sequential complexity of the problem (which in this paper is linear). A *fully-parallel* algorithm is a parallel algorithm that runs in constant time using an optimal number of processors. An *almost fully-parallel* algorithm is a parallel algorithm that runs in $O(\alpha(n))$ (the inverse of Ackermann function) time using an optimal number of processors. We refer the reader to [6, 23] for a discussion on the difficulty and theoretical importance of designing such extremely fast parallel algorithms.

Section 2 reduces the parentheses matching problem to a “numbers-matching” problem. Section 3 overviews a few known techniques. Section 4 presents an algorithm for the numbers-matching problem.

2. Parentheses matching

Given a balanced sequence of n parentheses, we wish to find for each left parenthesis its matching right parenthesis. More specifically, the input is an array of size n containing for each parenthesis (1) a binary flag marking whether it is a left or right parenthesis, and (2) its level of nesting.

We define a new problem that instead of matching parentheses matches numbers. Section 4 is devoted to deriving parallel algorithms for this numbers-matching problem. These algorithms imply an almost fully-parallel algorithm for the parentheses matching problem (see Corollary 2.1).

The numbers-matching problem

Input: An integer k and an array $A = (a_1, a_2, \dots, a_n)$ of integers, such that $|a_i - a_{i+1}| \leq k$. In words, the difference between each a_i , $1 \leq i < n$, and its successor a_{i+1} is at most k . This bound on the difference between two successive elements is called a *restricted-domain* assumption. The parameter k need not be a constant.

Output: Find for each a_i the nearest element to its left that is *smaller than or equal to* a_i and the nearest to its right that is *smaller than* a_i . That is, for each $1 \leq i \leq n$, find the maximal $1 \leq j < i$, and the minimal $i < k \leq n$ such that $a_j \leq a_i$ and $a_k < a_i$. We say

that a_j is the *left match* and a_k is the *right match* of a_i . If the left (or right) match of a_i does not exist, then the output with respect to a_i is defined to be -1 .

We make the simplifying assumption that \sqrt{k} and all other quantities needed in the paper are always integers. It is demonstrated in [6] that handling the general case does not change the complexity of the algorithm in terms of “big Oh”.

Example. Let $A = (a_1, \dots, a_{15}) = (6, 2, 7, 9, 14, 7, 9, 11, 7, 13, 4, 16, 12, 15, 8)$. The right match of $a_6 = 7$ is $a_{11} = 4$ and its left match is $a_3 = 7$.

Theorem 2.1. *Consider the numbers-matching problem, where k , the bound on the difference between two successive elements in A , is constant. The algorithm, presented in Section 4, is almost fully-parallel.*

Corollary 2.1. *There exists an almost fully-parallel algorithm for the parentheses matching problem.*

Proof. Let $A = (a_1, a_2, \dots, a_n)$ be the levels of nesting of our parentheses. For each pair of matching parentheses, we observe that: (1) their level of nesting is the same; and (2) the nesting level of each parenthesis between them is larger. The parentheses matching algorithm consists of solving the numbers-matching problem with respect to A , where the difference between two adjacent numbers in A is one (i.e., $k = 1$). For each right parenthesis its left mate is simply the left match in the numbers-matching problem. The corollary follows. \square

Berkman et al. [5] defined the numbers-matching problem and gave a parallel algorithm for a nonrestricted-domain version of it. The algorithm runs in $O(\log \log n)$ time using $n/\log \log n$ processors. It is easy to see that an algorithm for the numbers-matching problem implies an algorithm for finding the minimum among n elements. Thus, the $\Omega(\log \log n)$ time lower bounds for finding the minimum apply to the numbers-matching problem, as well. Such lower bounds were given in [22] for a comparison model, and in [15, 4] for a PRAM where the domain from which the elements are drawn is large. The rest of this paper deals with solving our restricted-domain variant of the numbers-matching problem. The lower bounds above do not apply to this variant.

3. Basics

We need the following reduction, problems, algorithms and definitions.

Reduction from parity to parentheses matching

Given an input $A = (a_1, a_2, \dots, a_n)$ for the parity problem, we construct an array B of size $8n$ which is an input to the parentheses matching problem as follows. (1) Each

“1” in A is replaced by two left parentheses and each “0” in A is replaced by one left and one right parenthesis to make an array B of size $2n$. (2) We concatenate to B an array of $2n$ right parentheses and concatenate the result to an array of $2n$ left parentheses (so B is now of size $6n$). (3) Finally, we reverse array A , replace each “1” by two right parentheses, replace each “0” by one left and one right parentheses and concatenate the result to B .

To sum up we get (i) $2n$ left parentheses, followed by (ii) a sequence of $2n$ parentheses corresponding to A , (iii) $2n$ right parentheses, and finally (iv) a sequence of $2n$ parentheses corresponding to the reverse of A . Consider the leftmost “1” in A . It yields two left parentheses in the sequence of item (ii). The mate of the first among these two left parentheses must be in the sequence of right parentheses of item (iii). Let i be the index of this mate in B . Then the number of ones in A is $(i - 4n)/2$. This implies the parity of A .

Finding the minimum for restricted-domain inputs

Input: Array $A = a_1, a_2, \dots, a_n$ of numbers. The *restricted-domain* assumption: each a_i is an integer between 1 and n .

Finding the minimum: Find the minimum value in A .

Fich et al. [10] gave the following four-step parallel algorithm for the restricted-domain minimum finding problem. It runs in $O(1)$ time using n processors. We use an auxiliary vector B of size n , that is all zero initially. (1) Processor i , $1 \leq i \leq n$, writes one into location $B(a_i)$. The problem now is to find the leftmost one in B . Partition B into \sqrt{n} equal size subarrays. (2) For each such subarray find in $O(1)$ time, using \sqrt{n} processors, if it contains a one. (3) Apply the $O(1)$ time algorithm of Shiloach and Vishkin [19] for finding the leftmost subarray of size \sqrt{n} containing one, using n processors. (4) Finally, reapply this latter algorithm for finding the index of the leftmost one in this subarray.

Remark 3.1. This algorithm can be readily generalized to yield $O(1)$ time for inputs between 1 and p^c , where $c > 1$ is a constant, as long as $p \geq n$ processors are used.

The prefix-minima and suffix-minima problems

Definitions: Let $A = (a_1, a_2, \dots, a_n)$. The *prefix minima* of A are the following n numbers. For each i , $1 \leq i \leq n$, we take the minimum value among a_1, a_2, \dots, a_i . The *suffix minima* of A are the following n numbers. For each i , $1 \leq i \leq n$, we take the minimum value among a_1, a_{i+1}, \dots, a_n .

Constant-time algorithms for numbers matching and prefix minima

Suppose we have n^2 processors. We show that in this case both the numbers-matching and the prefix-minima problems can be solved in $O(1)$ time.

Teams of processors: Below, and later in the paper, it will be convenient to use “teams of processors”. A *team* of size p is a set of p processors with consecutive indices

$i, i + 1, \dots, i + p - 1$, for some i and p . A team of size p is always (implicitly) assumed to have an allocated array of size $O(p)$. A team can operate as an independent “sub-PRAM”.

Constant-time numbers-matching algorithm: We allocate a team of n processors to each a_i . Processor j , $i + 1 \leq j \leq n$, that is allocated to a_i , marks $c_j := 1$ if $a_j < a_i$. Otherwise, it marks $c_j := 0$. Now, the smallest $j > i$ such that $c_j = 1$ is a_i 's right match. This j can be found using the part of Fich et al.'s [10] algorithm above that deals with finding the leftmost one. The left match for each element a_i can be found similarly.

Constant-time prefix-minima algorithm: We solve the numbers-matching problem with respect to A , using the constant-time numbers-matching algorithm presented above. For each 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 over the prefix a_1, a_2, \dots, a_i . This j can be found by [10] as before.

Remark 3.2. It is easy to extend the algorithms for the numbers-matching and prefix-minima problems to get algorithms for both problems that run in constant time (specifically $O(1/\varepsilon)$ time) and use $n^{1+\varepsilon}$ processors for any constant ε .

The inverse-Ackermann function

Consider a real function f . Let $f^{(i)}$ denote the i th iterate of f . (Formally, we denote $f^{(1)}(n) = f(n)$ and $f^{(i)}(n) = f(f^{(i-1)}(n))$ for $i > 1$.) Next, we define the $*$ (pronounced “star”) functional that maps the function f into another function $*f$: $*f(n) = \min\{i \mid f^{(i)}(n) \leq 1\}$ (we only consider functions for which this minimum is well defined). (Note that the function \log^* will be denoted $*\log$ using our notation. This change is for notational convenience.)

We define inductively a series I_d of slow-growing functions from the set of integers that are larger than 2 into the set of positive integers. (i) $I_1(n) = \lceil n/2 \rceil$ and (ii) $I_d = *I_{d-1}$ for $d \geq 2$. The first three in this series are familiar functions: $I_1(n) = \lceil n/2 \rceil$, $I_2(n) = \lceil \log n \rceil$ and $I_3(n) = *\log n$. The inverse-Ackermann function is $\alpha(n) = \min\{i \mid I_i(n) \leq i\}$.

Comment. Ackermann's function is defined following [12] as follows: $A_1(n) = 2n$ and $A_d(n) = A_{d-1}^{(n)}(1)$ for $d \geq 2$.

It is interesting to note that I_d is actually the inverse of the d th recursion level of A , the Ackermann function. Namely: $I_d(n) = \min\{i \mid A_d(i) \geq n\}$ or $I_d(A_d(n)) = n$. The definition of $\alpha(n)$ is equivalent to the more often used (but perhaps less intuitive) definition: $\min\{i \mid A_i(i) \geq n\}$.

The recursive star-tree data structure

Let n be positive integer. We define inductively a series of $\alpha(n) - 1$ trees. For each $2 \leq m \leq \alpha(n)$ a balanced tree with n leaves, denoted $BT(m)$ (for balanced tree), is defined. For a given m , $BT(m)$ is a recursive tree in the sense that each of its nodes holds a tree of the form $BT(m - 1)$.

The base of the inductive definition: We start with the definition of the star-tree $BT(2)$. $BT(2)$ is simply a complete binary tree with n leaves.

The inductive step (see Fig. 1): For $m, 3 \leq m \leq \alpha(n)$, we define $BT(m)$ as follows. $BT(m)$ has n leaves. The number of levels in $BT(m)$ is $*I_{m-1}(n) + 1 (= I_m(n) + 1)$. The root is at level 1 and the leaves are at level $*I_{m-1}(n) + 1$. Consider a node v at level $1 \leq l \leq *I_{m-1}(n)$ of the tree. Node v has $I_{m-1}^{(l-1)}(n)/I_{m-1}^{(l)}(n)$ children (recall that we define $I_{m-1}^{(0)}(n)$ to be n). The total number of leaves in the subtree rooted at node v is $I_{m-1}^{(l-1)}(n)$. We refer to the part of the $BT(m)$ tree described so far as the *top recursion level of $BT(m)$* (denoted for brevity as $TRL - BT(m)$). In addition, node v contains recursively a $BT(m - 1)$ tree. The number of leaves in this tree is exactly the number of children of node v in $BT(m)$.

An important feature of the star-tree data structure is the following. When the m th tree $BT(m)$ is employed to guide the computation, we invest $O(1)$ time on the top recursion level for $BT(m)$. Since $BT(m)$ has m levels of recursion, this leads to a total of $O(m)$ time. This is the parallel computational paradigm that was proposed in [6] and is used by the numbers-matching algorithm below. Similar parallel computational structures appeared in [7, 16].

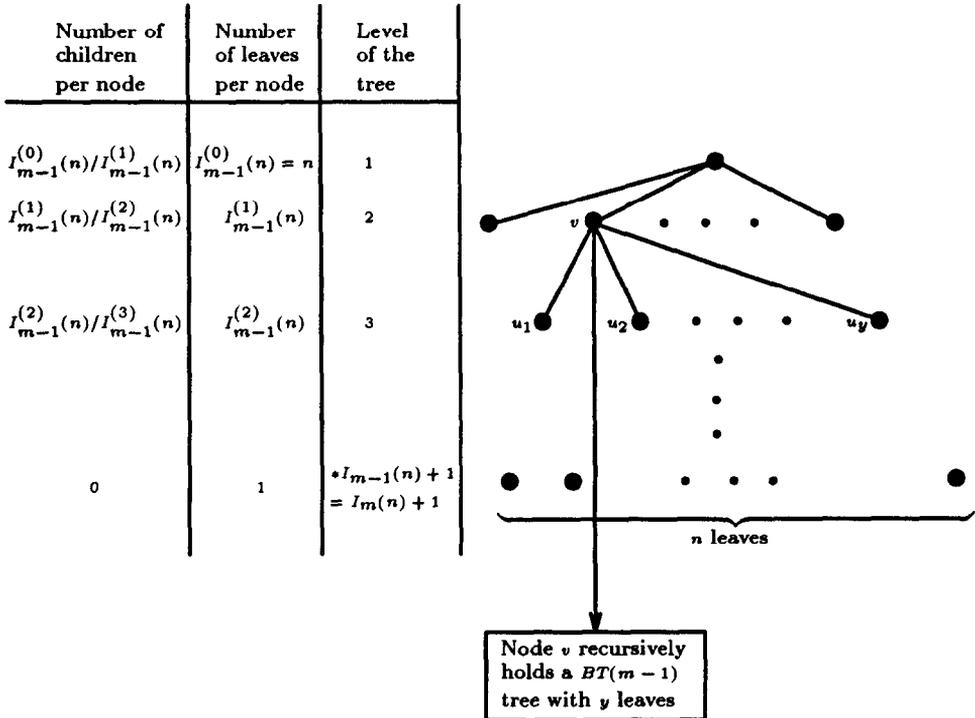


Fig. 1. $BT(m)$ – m th recursive star-tree.

4. Numbers-matching algorithm

4.1. High-level description

For proving Theorem 2.1 we construct recursively a series of $\alpha(n) - 1$ parallel algorithms for the numbers-matching problem. The role of the m th algorithm, $2 \leq m \leq \alpha(n)$, in this series becomes clear in the following lemma.

Lemma 4.1. *For $2 \leq m \leq \alpha(n)$, numbers matching with respect to array A can be solved in $2(m - 1)$ time pulses, using $nI_m(n) + \sqrt{k}n$ processors; each pulse takes constant time. Included within this complexity is the computation of prefix minima and suffix minima with respect to A .*

The algorithm for $m = \alpha(n)$ leads to an almost fully-parallel algorithms for the numbers-matching problem where k is a constant, thereby proving Theorem 2.1.

In the algorithm below we embed the numbers-matching problem on the star-tree data structure. This embedding is done by plugging in the following two-part *building block* at each internal node of the recursive star-tree data structure.

Let $D = (d_1, d_2, \dots, d_\delta)$ be an array of numbers, and let D_1, D_2, \dots, D_r be a partition of array D into subarrays of δ/r numbers each. The two-part building block solves the following three problems with respect to array D : (1) the numbers-matching problem itself; (2) prefix minima; and (3) suffix minima. This (triplicate) problem is marked $NMPS(D)$. The prefix-minima and suffix-minima problems are included since they are needed for our recursive description.

The two-part building block

1. *INTERNAL MATCHING*: For each $1 \leq i \leq r$ solve $NMPS(D_i)$. That is, for each number in D find its matches if they are within its subarray (justifying the name internal matching).

2. *EXTERNAL MATCHING*: This has three steps. (i) Pick $d_{m(i)}$, the minimum element for each D_i , $1 \leq i \leq r$, as a representative of D_i (if the minimum is not unique the leftmost minimum is taken) and let $B = (d_{m(1)}, d_{m(2)}, \dots, d_{m(r)})$. (ii) Solve $NMPS(B)$, the triplicate problem with respect to the representatives of the subarrays. (iii) For each number in D find its matches if they are outside its subarray (justifying the name external matching). Also, compute prefix minima and suffix minima with respect to D .

Below we describe an algorithm that realizes Lemma 4.1 for m , $m > 2$, assuming the existence of an algorithm that realizes the lemma for $m - 1$. Section 4.3 presents an algorithm for the case $m = 2$; this completes the recursive description of the algorithm, and the proof of Lemma 4.1.

4.1.1. Overview of the algorithm for $m > 2$

Input: Array $A = (a_1, a_2, \dots, a_n)$ of numbers where $|a_i - a_{i+1}| \leq k$.

Output: $NMPS(A)$.

Preview: The algorithm (for m) has three stages. In Stage 1 we apply the two-part building block and thereby reduce the size of the problem. In Stage 2, which is the main part of the algorithm, the recursive star-tree is used to solve the reduced problem. In Stage 3 we complete the solution of $NMPS(A)$.

Stage 1: Reducing the size of the problem

The set A is partitioned into $n_1 = n/I_m^3(n)$ subarrays A_1, A_2, \dots, A_{n_1} of size $I_m^3(n)$ each (where $I_m^3(n)$ means $(I_m^3(n))^3$). We solve the INTERNAL MATCHING part of the two-part building block with respect to A . Together with the EXTERNAL MATCHING part which is handled in Stages 2 and 3, this implies a solution to $NMPS(A)$. To solve $NMPS(A_i)$, $1 \leq i \leq n_1$, we use the constant-time algorithms for the prefix-minima and numbers-matching problems (where, for Remark 3.2, $\varepsilon = 1/3$).

Remark. The reason behind the (seemingly arbitrary) grouping of array A into subarrays of $I_m^3(n)$ elements each is to maintain processor count within our desired bounds in Stage 2 below.

In Stage 2 the star-tree $BT(m)$ is used to solve the triplicate problem with respect to an array $E = (e_1, e_2, \dots, e_{n_1})$ of numbers, where e_i is the minimum element in subarray A_i , $1 \leq i \leq n_1$; formally, this means solving $NMPS(E)$, which is item (ii) in the EXTERNAL MATCHING part.

Stage 2: Embedding the reduced problem on the recursive star-tree $BT(m)$

Input: Array $E = (e_1, e_2, \dots, e_{n_1})$ of numbers, $|e_i - e_{i+1}| \leq k_1 = kI_m^3(n)$.

Output: $NMPS(E)$.

Guided by $BT(m)$, this stage has $2(m-1)$ (time) pulses; each pulse takes constant time. The leaves of $TRL - BT(m)$, the top recursion level of $BT(m)$, are the elements of E . Each internal node of $TRL - BT(m)$ has an array with the values of its leaves. Consider now a node v at some level, l , $1 \leq l \leq I_m(n_1)$, of $TRL - BT(m)$. Node v has $\delta = I_{m-1}^{l-1}(n_1)$ leaves, and $r = \delta/I_{m-1}(\delta)$ children, v_1, v_2, \dots, v_r , in $TRL - BT(m)$. Let $D = (d_1, d_2, \dots, d_\delta)$ denote the array of the leaves of node v , let D_i , $1 \leq i \leq r$, denote the array of leaves of v_i , and let $\delta_i = I_{m-1}(\delta)$ denote the size of each D_i . Since the arrays D_1, D_2, \dots, D_r represent a partition of array D into subarrays of equal size, we can apply the two-part building block with respect to array D . Below we concentrate on solving the EXTERNAL MATCHING part with respect to (the array D of) node v .

The following subtle point is crucial to understanding the role of the two-part building block in our presentation. At the lowest level of the tree, subsets consist of a single leaf, and thus the match at this point must be external. At higher levels, the internal matches are all handled by nodes at lower levels. Therefore, the solution of EXTERNAL MATCHING with respect to all internal nodes of $TRL - BT(m)$, without ever solving any INTERNAL MATCHING, actually provides a solution to $NMPS(E)$.

EXTERNAL MATCHING with respect to v

Pulse 1 of the algorithm finds the minimum $d_{m(i)}$ in each D_i , $1 \leq i \leq r$, in parallel using the algorithm of Remark 3.1. This solves item (i) of the EXTERNAL MATCHING part. Let $B = (d_{m(1)}, d_{m(2)}, \dots, d_{m(r)})$. We now solve $NMPS(B)$ – item (ii) of the EXTERNAL MATCHING problem – by a recursive application of Lemma 4.1. That is, we apply the algorithm for $m - 1$ which uses inductively $2(m - 2)$ pulses (Pulse 2 to Pulse $2m - 3$). Note that the difference between adjacent entries of B is at most $k_1 \delta_1$. Thus, the recursive application needs $rI_{m-1}(r) + \sqrt{k_1 \delta_1} \cdot r$ processors which will be shown to be within the bounds of node v .

Pulse number $2m - 2$ which is the last pulse of the algorithm for m computes item (iii) of EXTERNAL MATCHING. This is done in four steps.

Step 1: Compute prefix minima and suffix minima with respect to D .

It remains to find for each number in D its matches if they are outside its subarray. For this we first find (in Step 2 below) the matches in D for each of the $d_{m(i)}$'s. For this computation note that if the right match of $d_{m(i)}$ in B is $d_{m(j)}$ then the right match of $d_{m(i)}$, in D lies in subarray D_j .

Step 2: Let the array of prefix minima with respect to subarray D_j be $P_j = (p_1, p_2, \dots, p_{\delta_j})$. Note that P_j is a nonincreasing array where the difference between any two adjacent elements in P_j is at most k_1 .

Step 2.1: We build a table that enables processing of the following *search* query with respect to P_j in constant time using one processor: Given any number x , $search_{P_j}(x)$ provides the minimal index l such that $p_l \leq x$.

Step 2.2: The right match of $d_{m(i)}$ (whose match in B is $d_{m(j)}$) is now provided by $search_{P_j}(d_{m(i)} - 1)$. Similar tables are built for the array of suffix minima of each D_j . They are used for finding left matches for the $d_{m(i)}$'s.

Step 3: Find for each d_γ , $1 \leq \gamma \leq \delta$, in D whose right (left) match is not in its subarray, the subarray D_j that contains its right match.

Step 4: Find the right match of each such d_γ within subarray D_j , using the query $search_{P_j}(d_\gamma - 1)$. The left match of d_γ is found similarly.

This finishes the description of EXTERNAL MATCHING with respect to v . It is done in parallel with respect to all nodes of $TRL - BT(m)$ and results in solving $NMPS(E)$, for the following reason. First note that prefix minima and suffix minima with respect to E were computed when Step 1 of pulse $2m - 2$ was applied to the root of $TRL - BT(m)$. Now, let v denote the lowest common ancestor of some e_γ , $1 \leq \gamma \leq n_1$, and its right match. The solution of EXTERNAL MATCHING at node v (and at node v only) will provide the right match of e_γ .

Stage 3: Completing the computation of $NMPS(A)$

We compute item (iii) of EXTERNAL MATCHING with respect to A and thereby complete the computation of EXTERNAL MATCHING (and thus of $NMPS(A)$). Specifically, we compute matches for each element in A if they are outside its subarray, and in addition compute the prefix minima and suffix minima with respect to A . To compute prefix minima and suffix minima with respect to A we use: (a) prefix minima and suffix minima within each A_i (that were computed in the INTERNAL MATCHING part of Stage 1) and (b) prefix minima and suffix minima with respect to E (that were computed as part of $NMPS(E)$ in Stage 2). Computation of matches for elements whose match is not in their subarray is done similar to Steps 2–4 of Stage 2 above.

4.2. Detailed description

Inductively, we assume that we have an algorithm that solves the numbers-matching problem for the array $A = (a_1, a_2, \dots, a_n)$ in $c(m-1)$ time using $nI_{m-1}(n) + \sqrt{kn}$ processors, where c is a constant. We construct an algorithm that solves the numbers-matching problem in $c_1 + c(m-1)$ time for some constant c_1 , using $nI_m(n) + \sqrt{kn}$ processors. Selecting initially $c > c_1$ implies that the algorithm runs in cm time using $nI_m(n) + \sqrt{kn}$ processors.

Stage 1

We partition A into $n_1 = n/I_m^3(n)$ subarrays A_1, A_2, \dots, A_{n_1} of $I_m^3(n)$ elements each, allocate a team of $I_m^4(n)$ processors to each subarray and solve the numbers-matching problem, the prefix-minima problem and the suffix-minima problem within the subarray. This can be done in constant time using the algorithms of Remark 3.2 (for $\varepsilon = 1/3$). It uses $I_m^4(n)$ processors per subarray and $nI_m(n)$ processors overall. This computation solves INTERNAL MATCHING within each subarray and provides the minimum in each subarray which is the input (i.e., array E) to Stage 2.

Stage 2

We build $TRL - BT(m)$, the top recursion level of a $BT(m)$ tree whose leaves are the elements of $E = (e_1, \dots, e_{n/I_m^2(n)})$. $TRL - BT(m)$ has $*I_{m-1}(n/I_m^3(n)) \leq I_m(n)$ levels, excluding the level of the leaves. We allocate a team of $I_m(n)(1 + \sqrt{kI_m^2(n)})$ processors to each leaf, $1 + \sqrt{kI_m^2(n)}$ for each of its ancestors. The total amount of processors used is $n/I_m^2(n) + \sqrt{kn}$ (which is less than $nI_m(n) + \sqrt{kn}$).

Pulse 1 (Computation of the minimum element of the D_i 's): We find the minimum over each D_i , $1 \leq i \leq r$, using the constant algorithm of Remark 3.1. The processors needed for this computation are taken from the child v_i of v that corresponds to subarray D_i . (Recall that the size of D_i is $\delta_1 = I_{m-1}(\delta)$, that δ is the size of D , and that the difference between adjacent elements of D_i is $kI_m^3(n)$.) After subtracting the first element of the array D_i from each of its elements, we get an array whose elements range between $-\delta_1 I_m^3(n)$ and $k\delta_1 I_m^3(n)$. The size of the range, which is $2k\delta_1 I_m^3(n) + 1$,

does not exceed the square of the number of processors at node v_i , which is $(\delta_1 + \sqrt{k}\delta_1 I_m^2(n))^2$, and the algorithm of Remark 3.1 can be applied.

Pulses 2 through $2m - 3$ (Computation of item (ii)): We solve $NMPS(B)$ by applying the assumed algorithm from $m - 1$. The processors needed for this computation are taken from the allocation of node v . We have shown that this computation uses $rI_{m-1}(r) + \sqrt{k_1\delta_1} \cdot r$ processors where $r = \delta/I_{m-1}(\delta)$ is the size of B , $k_1 = kI_m^3(n)$ is a bound on the difference between adjacent elements of B and $\delta_1 = I_{m-1}(\delta)$ is the size of each D_i . Thus the number of processors used is

$$rI_{m-1}(r) + \sqrt{k_1\delta_1} \cdot r = \frac{\delta}{I_{m-1}(\delta)} I_{m-1} \left(\frac{\delta}{I_{m-1}(\delta)} \right) + \sqrt{kI_{m-1}(\delta)I_m^3(n)} \frac{\delta}{I_{m-1}(\delta)}$$

$$\leq \delta + \sqrt{kI_m^3(n)} \delta / \sqrt{I_{m-1}(\delta)} \leq \delta + \sqrt{k}\delta I_m^2(n),$$

which is the number of processors at node v .

Pulse $2m - 2$ (Computation of item (iii)):

Step 1: We compute prefix minima with respect to D (suffix minima are computed similarly). That is, for each leaf e_i of v we need to find the minimum over the prefix of e_i with respect to the leaves of node v . For this, the minimum among the following list of at most $*I_{m-1}(n) + 1 = I_m(n) + 1$ numbers is computed: Each level l , $level(v) < l \leq I_m(n) + 1$, of the tree contributes (at most) one number. Let u denote the ancestor at level $l - 1$ of e_i and let u_1, \dots, u_j denote its children, which are at level l . Suppose u_j , $j > 1$, is an ancestor of e_i . We take the prefix minimum over the leaves of u_1, \dots, u_{j-1} . This prefix minimum was computed above (by the assumed algorithm for $m - 1$). If u_1 is the ancestor of e_i then level l does not contribute anything (actually, level l contributes a large default value so that the minimum computation is not affected). Finally, e_i is also added to the list. This minimum computation can be done in constant time using $I_m^2(n)$ processors (by the algorithm of [19]). Note that: (1) all prefix minima and all suffix minima with respect to E are computed (in the root) in this step; and (2) given a leaf e_i in $TRL - BT(m)$, finding node u (the ancestor at level $l - 1$ of e_i) is easy since $TRL - BT(m)$ is balanced.

Step 2.1 (Handling search queries): We describe a procedure that handles search queries for a generic nonincreasing array of integers $M = (m_1, m_2, \dots, m_q)$ such that the difference between any two adjacent elements of M is at most k' (formally $m_i \geq m_{i+1} \geq m_i - k'$). The query $search_M(x)$ requests the minimal index l , $1 \leq l \leq q$, such that $m_l \leq x$ (if such an index exists). We show how to preprocess array M with $\sqrt{k'q}$ processors in constant time, so that each query can be processed by a single processor in constant time.

By way of motivation, assume that we have $k'q$ processors for preprocessing, where k' processors are standing by each element m_i . Possible query values are all x , $m_1 \geq x \geq m_1 - k'(q - 1)$ (since $m_q \geq m_1 - k'(q - 1)$). Construct in $O(1)$ time an array (of size $O(k'q)$), such that for each such value of x the array has the index for $search_M(x)$. Specifically, processor j , $0 \leq j \leq k' - 1$, of element m_i enters the answer

i for query $search_M(m_i + j)$, unless $m_i + j \geq m_{i-1}$. A variant of this simple construction uses only $\sqrt{k'q}$ processors, as shown below.

We build two kinds of arrays:

(i) A single array B , of size $\leq \sqrt{k'q}$. Array B will have all answers for query values $m_1 \geq x \geq m_q$ which are integer multiples of $\sqrt{k'}$. It is easy to compute array B in constant time by assigning $\sqrt{k'}$ processors to each element m_i , $1 \leq i \leq q$.

(ii) Several arrays, each of size $\sqrt{k'}$. Consider a query $search_M(x)$, and recall that array B solves the query $j = search_M(x_1)$, where $x_1 = \lceil x/\sqrt{k'} \rceil \cdot \sqrt{k'}$ is an integer multiple of $\sqrt{k'}$. By element m_j , we store an array C_j of size $\leq \sqrt{k'}$. Array C_j will have all answers for query values $x_1 \geq y > x_1 - \sqrt{k'}$. There will be $\sqrt{k'}$ processors standing by each element m_i , $1 \leq i \leq q$, and they will compute in $O(1)$ time all arrays C_j (for every element m_j such that $j = search_M(x)$ for some integer multiple x of $\sqrt{k'}$). Specifically, the processors that will participate in the computation of C_j belong to all elements m_j, m_{j+1}, \dots, m_i that lie between the same two multiples of $\sqrt{k'}$ as m_j itself. Implementation details are omitted.

In our case the array P_j is of size δ_1 and the difference between adjacent elements of P_j is $k_1 = kI_m^3(n)$. Thus, the preprocessing for $search$ query retrieval takes constant time using $\sqrt{kI_m^3(n)}\delta_1$ processors, less than the amount of processors at node v_j (which is $\delta_1 + \sqrt{k}\delta_1 I_m^3(n)$).

Step 2.2: We need to answer one $search$ query with respect to the minimum $d_{m(i)}$ of each of the subarrays D_i , $1 \leq i \leq r$. This can be done in constant time using a single processor per query and constant time using r processors (that are available at node v) overall.

To compute matches for all elements in D whose matches are not in their respective subarrays (Step 3) we need a few definitions and observations from [5]. Let $r(i)$ denote the index of the right match of $d_{m(i)}$ in D and let $l(i)$ denote the index of the left match of $d_{m(i)}$.

Lemma 4.2 (Berkman et al. [5]). (a) Suppose $r(i)$ exists and the index of the subarray of $r(i)$ is larger than $i + 1$. Then, there exists a subarray $g > i$, such that $l(g)$ belongs to subarray i and $r(g)$ belongs to the subarray of $r(i)$. Moreover, subarray g is unique (see Fig. 2).

(b) (Symmetric to item (a)) Suppose $l(i)$ exists and the index of the subarray of $l(i)$ is smaller than $i - 1$. Then, there exists a subarray $h < i$, such that $r(h)$ belongs to subarray i and $l(h)$ belongs to the subarray of $l(i)$. Subarray h is unique.

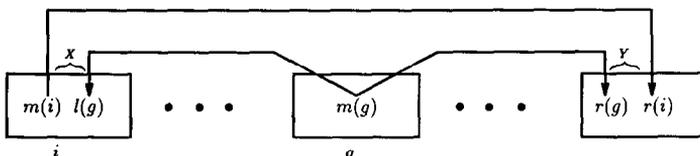


Fig. 2.

Suppose the assumptions for both (a) and (b) of Lemma 4.2 hold. (Cases where these assumptions do not hold are easy and will be mentioned later.) For each $m(i)$, $1 \leq i \leq r$, we define two pairs of subarrays: a *right* pair and a *left* pair. We describe in detail only the right pair of $m(i)$. Let g be as in Lemma 4.2. The first subarray of such a pair spans to the right of $m(i)$ until $l(g)$ (subarray X in Fig. 2). The second subarray of the right pair spans to the left of $r(i)$ until $r(g)$ (subarray Y in Fig. 2). The important observations are as follows.

We define the *first subsequence* to be the subsequence of the first subarray that consists of every element of this subarray whose right match does not lie within the subarray. Similarly, we define the *second subsequence* to be the subsequence of the second subarray that consists of every element of this subarray whose left match does not lie within the subarray.

Observation 1. *Suppose element d_j belongs to the first subarray of $m(i)$. Then d_j belongs to the first subsequence of $m(i)$ if and only if d_j is less than or equal to the minimum over the suffix of d_{j+1} in subarray i . Similarly, if element d_j belongs to the second subarray of $m(i)$ then d_j belongs to the second subsequence of $m(i)$ if and only if d_j is less than the minimum over the prefix of d_{j-1} in the subarray of $r(i)$.*

Corollary. *The values of the elements in the first subsequence are monotonically non-decreasing from left to right. The values of the elements in the second subsequence are monotonically increasing from right to left.*

Observation 2. *The right matches of all elements in the first subsequence of $m(i)$ lie in the second subsequence of $m(i)$. The left matches of all elements in the second subsequence of $m(i)$ (with the exception of $r(i)$) lie in the first subsequence of $m(i)$.*

Proof. Consider an element d_j , $m(i) \leq j \leq l(g)$, of the first subsequence. From Observation 1 we have that d_j is less than or equal to all elements $d_{j+1}, \dots, d_{l(g)}$. The definition of the numbers-matching problem implies that $d_{l(g)}$ is less than or equal to $d_{m(g)}$ and that $d_{m(g)}$ is less than all elements $d_{l(g)+1}, \dots, d_{m(g)-1}$. It follows that d_j is less than or equal to all elements $d_{j+1}, \dots, d_{m(g)}$. Similarly it can be shown that d_j is less than or equal to all elements $d_{m(g)+1}, \dots, d_{r(g)-1}$. On the other hand, $d_{m(i)}$, and thus also d_j , is larger than $d_{r(i)}$. This implies that the right match of d_j must be in the second subarray. It is easy to see that the right match must belong to the second subsequence. This proves the first part of Observation 2; the proof of the second part is similar. \square

We are now ready to present the implementation of Step 3.

Step 3: For each subarray g , $1 \leq g \leq r$, of D , we do the following.

(a1) Let i be the index of the subarray of $l(g)$. A processor allocated to subarray g checks whether the subarray of $r(i)$ is the same as the subarray of $r(g)$; namely, we check whether $m(g)$ relates to $m(i)$ as in Lemma 4.2(a). If yes, the processor determines the boundaries of the subarrays of the right pair of $m(i)$. Symmetrically, boundaries of left pairs are found using Lemma 4.2(b).

(a2) We allocate a team of $\delta/r (= I_{m-1}(\delta))$ processors to each subarray i , $1 \leq i \leq r$ (overall δ processors are allocated). Processor t , $1 \leq t \leq \delta/r$, of subarray i checks (using Observation 1) whether element t of the subarray belongs to the first subsequence of the right pair of subarray i . If yes, the processor marks this element with the index of the subarray of $r(i)$ (telling the element that its right match is in the subarray of $r(i)$). Similarly, each element of the second subsequence of the right pair of subarray i is marked ‘ i ’ (telling the element that its left match is in subarray i). Elements belonging to subsequences of left pairs are marked symmetrically.

Remark. In case where instead of the assumption of Lemma 4.2(a), we have $r(i) = i + 1$, the boundaries of the first subarray of the right pair of $m(i)$ include all elements of subarray i that are to the right of $m(i)$. The boundaries of the second subarray include all elements of subarray $i + 1$ that are to the left of $r(i)$. Similar considerations apply to the left pair of $m(i)$.

Lemma 4.3 (Berkman et al. [5]). *Let d_j be an element in subarray i and let l be the index of the subarray in which the right match of d_j lies. Then d_j belongs either to (the first subsequence of) the right pair of $m(i)$ or to the left pair of $m(l)$.*

Corollary. *Step (a2) results in (correctly) marking all elements whose matches (right or left) are not in their subarrays.*

The number of processors used in Step (a1) is $r = \delta/I_{m-1}(\delta)$. The number of processors used in Step (a2) is δ . Overall Step 3 takes constant time using δ processors (which are available at node v).

Step 4: The complexity of this step is constant time using δ processors (since there are at most δ queries).

Stage 3

We compute prefix minima and suffix minima with respect to A as follows: Consider some j , $1 \leq j \leq n$, and let A_i be the subarray containing a_j . The minimum over the prefix of a_j with respect to A is the minimum between the prefix e_{i-1} with respect to array E (computed as part of $NMPS(E)$ in Stage 2) and the minimum over the prefix of a_j with respect to A_i (computed in Stage 1). Suffix minima with respect to A are computed similarly. Matches for elements whose match is not in their subarray are computed similar to Steps 2–4 of Pulse $2m - 2$ of Stage 2. We analyze the running times of these three steps. Building the tables for *search* queries within each subarray A_i , $1 \leq i \leq n_1$ (Step 2), uses here $\sqrt{k}I_m^3(n)$ processors for each subarray and \sqrt{kn} processors overall. Answering all *search* queries (Step 3) takes constant time using $n/I_m^3(n)$ processors. Extending the numbers-matching solution for matches outside the subarrays (Step 4) takes constant time using n processors.

Complexity of the recursive step: In addition to application of the inductively assumed algorithm, the other steps of the algorithm take constant time using $nI_m(n) + \sqrt{kn}$ processors. This totals cm time using $nI_m(n) + \sqrt{kn}$ processors.

4.3. The recursion base (numbers-matching algorithm for $m = 2$)

Lemma 4.4. *Let m be 2. Then $I_m(n) = \log n$. The algorithm runs in time $2c$ for some constant c using $n \log n + \sqrt{kn}$ processors.*

The details of the algorithm for $m = 2$ are very similar to those of the algorithm for general m . We give below only the most crucial details.

Stage 1: Reducing the size of the problem

We partition A into $n_2 = n/\log^3 n$ subarrays of size $\log^3 n$, solve the NMPS problems with respect to each subarray and construct array $E = (e_1, e_2, \dots, e_{n_2})$, where e_i is the minimum element in subarray i , $1 \leq i \leq n_2$.

Stage 2: Solving NMPS(E)

Input: Array $E = (e_1, e_2, \dots, e_{n_2})$ of numbers, $|e_i - e_{i+1}| \leq k_2 = k \log^3 n$.

We build $BT(2)$, a complete binary tree, whose leaves are the elements of E and allocate a team of $\log n(1 + \sqrt{k} \log^2 n)$ processors to each leaf, $1 + \sqrt{k} \log^2 n$ processors for each of its ancestors. The total amount of processors used is $n/\log^2 n + \sqrt{kn}$ (which is less than $n \log n + \sqrt{kn}$. Consider a node v of $BT(2)$ with δ leaves $D = (d_1, d_2, \dots, d_\delta)$. Let v_1 and v_2 be the left and right children of v , respectively, and let D_1 and D_2 be their respective arrays of leaves. Note that D_1 and D_2 are subarrays of D of size $\delta/2$ each. The algorithm has two (time) pulses, each takes constant time.

Pulse 1: Find the minimum over D_1 (and D_2) as in the algorithm for m : The difference between the minimum value and the maximum value in D_1 does not exceed the square of the number of processors of v_1 and the algorithm of Remark 3.1 can be applied.

Pulse 2: It has three steps.

Step 1: We compute prefix minima with respect to D . That is, for each leaf e_i of v , we need to find the minimum over the prefix of e_i in v . For this, the minimum among the following list of (at most) $\log n + 1$ numbers is computed. Denote the level of v in the binary tree by $level(v)$. Each level l , $level(v) < l \leq \log n + 1$, of the tree contributes (at most) one number. Let u denote the ancestor at level $l - 1$ of e_i . Let u_1 and u_2 denote the right and left children of u , respectively. If e_i belongs to (the subtree rooted at) u_2 then level l contributes the minimum over u_1 . If e_i belongs to u_1 then level l contributes a large default value (as in Step 1 of the algorithm for m). Finally, e_i is also included in the list. This minimum computation can be done in constant time using $\log^2 n$ processors by the algorithm of [19]. Note that: (1) all prefix minima and all suffix minima of E are computed (in the root) in this step; and (2) since $BT(2)$ is a complete binary tree, node u can be easily found in constant time using one processor (see also [11]).

It remains to find for each number in D its matches if they are outside its subarray. Doing this for all descendent nodes of v will imply a solution to $NMPS(E)$.

Step 2: Let the array of prefix minima with respect to subarray D_2 be $P_2 = (p_1, p_2, \dots, p_{\delta/2})$. We build a table that enables processing *search* query with respect to P_2 in constant time using one processor. (Given any number x , $search_{P_2}(x)$ provides the minimal index l such that $p_l \leq x$.) The right match of an element d_i , $1 \leq i \leq \delta/2$, of D whose right match is not in D_1 is provided by $search_{P_2}(d_i - 1)$. Similar tables are built for the suffix minima of D_1 . They are used for finding left matches for the elements d_i , $\delta/2 + 1 \leq i \leq \delta$, whose left match is not in D_2 . Building the tables for search queries takes constant time using $\sqrt{k \log^3 n} \cdot \delta/2$ processors, less than the amount of processors at node v_1 (which is $\delta/2 + \sqrt{k} \cdot \delta/2 \cdot \log^2 n$).

Step 3: Answer a *search* query for each element d_i , $1 \leq i \leq \delta$, of D whose right match is not in d_i 's subarray. We need to answer (at most) δ queries with respect to D . This takes constant time using δ processors (that are available at node v).

This finishes the description of the computation with respect to v . It is done in parallel with respect to all nodes of $BT(2)$ and results in solving $NMPS(E)$. Extending the solution to A (Stage 3) is done in the same way as in the algorithm for m .

Complexity of the recursion base: $O(1)$ time using $n \log n + \sqrt{kn}$ processors.

Lemma 4.4 follows.

Together with Lemma 4.4, Lemma 4.1 follows.

4.4. From recursion to algorithm

The recursive procedure in Lemma 4.1 translates easily into a constructive parallel algorithm where the instructions for each processor at each time unit are available. For such translation, issues such as processor allocation and computation of certain functions need to be taken into account. Since $TRL - BT(m)$ is balanced, allocating processors in the algorithm above can be done in constant time if the following functions are precomputed: (a) $I_m(x)$ for $1 \leq x \leq n$ and (b) $I_{m-1}^{(i)}(x)$ for $1 \leq x \leq n$ and $1 \leq i \leq I_m(x)$. Let us illustrate how the processor allocation which is described at the beginning of Stage 2 of the algorithm for m is actually done. We use $n/I_m^2(n) + \sqrt{kn}$ processors. The processors are partitioned into $I_m(n)$ groups of $n/I_m^3(n) + \sqrt{kn}/I_m(n)$ processors each. Each group is allocated to one level. Within a level, the processors in its group are partitioned equally among the nodes of the level. This will provide a sufficient number of processors to each node. Some processors are superfluous and simply remain idle. These same functions suffice for all other computations above. The functions will be computed and stored in a table at the beginning of the algorithm. Computation of the functions is given in [6].

4.5. The optimal algorithm

Consider the numbers-matching problem where k , the bound on the difference between successive entries in A , is constant. Below we give an almost fully-parallel algorithm for the problem. It implicitly applies the two-part building block.

Step 1: Partition the input array A into successive subarrays of $\alpha^2(n)$ elements each. Within each subarray solve the numbers-matching problem, the prefix-minima problem and the suffix-minima problem. This can be done in $O(\alpha(n))$ time using $\alpha(n)$ processors for each subarray (using, for instance, [13]). Overall this takes $O(\alpha(n))$ time using $n/\alpha(n)$ processors.

Step 2: Find the minimum in each subarray and put these minima into array B (of size $n/\alpha^2(n)$).

Step 3: Out of the series of numbers-matching algorithms of Lemma 4.1 apply the algorithm for $\alpha(n)$ to B , where k' , the difference between two successive elements of B , is $O(\alpha^2(n))$. This takes $O(\alpha(n))$ time using $n/\alpha^2(n) \cdot I_{\alpha(n)}(n) + \sqrt{k'} n/\alpha^2(n)$ processors. The definition of $\alpha(n)$ implies that $I_{\alpha(n)}(n) \leq \alpha(n)$ and the complexity of processors is thus $2n/\alpha(n)$. This can be simulated in $O(\alpha(n))$ time using $n/\alpha(n)$ processors.

Step 4: Find right and left matches in A for all elements in B .

Step 5: Find right and left matches for all elements in A .

The details of the last two steps are similar to Steps 2–4 of Stage 2 of the algorithm for m . Since k is constant the tables for *search* queries can be built in constant time using $\alpha^2(n)$ processors for each subarray and constant time using n processors overall. All other computations take constant time using (at most) n processors.

This concludes the proof Theorem 2.1.

References

- [1] R.J. Anderson, E.W. Mayr and M.K. Warmuth, Parallel approximation algorithms for bin packing, *Inform. and Comput.* 82 (1989) 262–277.
- [2] I. Bar-On and U. Vishkin, Optimal parallel generation of a computation tree form, *ACM Trans. Prog. Languages Systems* 7 (1985) 348–357.
- [3] P. Beame and J. Hastad, Optimal bounds for decision problems on the CRCW PRAM, in: *Proceedings of the 19th Annual ACM Symposium on Theory of Computing* (1987) 83–93.
- [4] O. Berkman, Y. Matias and P.L. Ragde, Triply-logarithmic upper and lower bounds for minimum, range minima, and related problems with integer inputs, in: *Proceedings of the 3rd Workshop on Algorithms and Data Structures* (1993).
- [5] O. Berkman, B. Schieber and U. Vishkin, Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values, *J. Algorithms* 14 (1993) 344–370.
- [6] O. Berkman and U. Vishkin, Recursive star-tree parallel data-structure, *SIAM J. Comput.* 22 (1993) 221–242.
- [7] A.K. Chandra, S. Fortune and R.J. Lipton, Unbounded fan-in circuits and associative functions, in: *Proceedings of the 15th Annual ACM Symposium on Theory of Computing* (1983) 52–60.
- [8] C.-Y. Chen, S.K. Das, G. Lewis and S. Prasad, Parentheses matching revisited: three parallel algorithms, Technical Report CRPDC-90-7, University of North Texas (1990).
- [9] E. Dekel and S. Sahni, Parallel generation of postfix and tree forms, *ACM Trans. Prog. Languages Systems* 5 (1983) 300–317.
- [10] F.E. Fich, P.L. Ragde and A. Wigderson, Relations between concurrent-write models of parallel computation, *SIAM J. Comput.* 1 (1988) 606–627.
- [11] D. Harel and R.E. Tarjan, Fast algorithms for finding nearest common ancestors, *SIAM J. Comput.* 13 (1984) 338–355.
- [12] S. Hart and M. Sharir, Non linearity of Davenport-Schinzel sequences and generalized path compression schemes, *Combinatorica* 6 (1986) 151–177.

- [13] R.E. Ladner and M.J. Fischer, Parallel prefix computation, *J. ACM* 27 (1980) 831–838.
- [14] R. Mattheyses and C.M. Fiduccia, Parsing Dyck languages on parallel machines, in: *Proceedings of the 20th Allerton Conference on Communication, Control and Computing* (1982) 272–280.
- [15] F. Meyer auf der Heide and A. Wigderson, The complexity of parallel sorting, in: *Proceedings of the 26th IEEE Annual Symposium on Foundation of Computer Science* (1985) 532–540.
- [16] P.L. Ragde, The parallel simplicity of compaction and chaining, in: *Proceedings of the 17th International Colloquium on Automata, Languages and Programming* (1990) 744–751.
- [17] D. Sarkar and N. Deo, Parallel algorithms for parentheses matching and generation of random balanced sequences of parentheses, in: *Lecture Notes in Computer Science*, 297 (Springer, Berlin, 1988) 970–984.
- [18] B. Schieber, Design and analysis of some parallel algorithms, Ph.D. Thesis, Department of Computer Science, Tel Aviv University (1987).
- [19] Y. Shiloach and U. Vishkin, Finding the maximum, merging, and sorting in a parallel computation model, *J. Algorithms* (1981) 88–102.
- [20] Y.N. Srikant and P. Shankar, A new parallel algorithm for parsing arithmetic infix expressions, *Parallel Comput.* 4 (1987) 291–304.
- [21] Y.N. Srikant and P. Shankar, Parallel parsing of programming languages, *Inform. Sci.* 43 (1987) 55–83.
- [22] L.G. Valiant, Parallelism in comparison problems, *SIAM J. Comput.* 4 (1975) 348–355.
- [23] U. Vishkin, Structural parallel algorithmics, in: *Proceedings of the 18th International Colloquium on Automata, Languages and Programming* (1991) 363–380.