RANDOMIZED RANGE-MAXIMA IN NEARLY-CONSTANT PARALLEL TIME [1]

Omer Berkman [2] — King's College London

Yossi Matias [3] — AT&T Bell Laboratories

Uzi Vishkin [4] — University of Maryland & Tel Aviv University

Abstract. Given an array of n input numbers, the *range-maxima* problem is that of preprocessing the data so that queries of the type "what is the maximum value in subarray $[i..j]$" can be answered quickly using one processor. We present a randomized preprocessing algorithm that runs in $O(\log^* n)$ time with high probability, using an optimal number of processors on a CRCW PRAM; each query can be processed in constant time by one processor. We also present a randomized algorithm for a parallel comparison model. Using an optimal number of processors, the preprocessing algorithm runs in $O(\alpha(n))$ time with high probability; each query can be processed in $O(\alpha(n))$ time by one processor. (As is standard, $\alpha(n)$ is the inverse of Ackermann function.) A constant time query can be achieved by some slowdown in the performance of the preprocessing stage.

Key words. parallel algorithms; randomized algorithms; PRAM; comparison model; range maximum; prefix maximum.

Subject classifications. 68Q20.

---

[1] APPEARED IN *J. COMPUTATIONAL COMPLEXITY* 2 (1992), 350-373.

[2] Dept. of Computer Science, King's College London, The Strand, London WC2R 2LS, England.

[3] AT&T Bell Laboratories, 600 Mountain Ave., Murray Hill, NJ 07974-0636. Part of research was done while the author was at the Institute for Advanced Computer Studies, University of Maryland (UMIACS) and at the Department of Computer Science, Tel Aviv University, Israel.

[4] Institute for Advanced Computer Studies and Department of Electrical Engineering, University of Maryland, College Park, MD 20742. Also with the Department of Computer Science, Tel Aviv University, Israel.

1. Introduction

Given an array of n input numbers, the *range-maxima* problem is the problem of preprocessing the data so that queries of the type "what is the maximum value in subarray $[i..j]$?" can be answered quickly using one processor. The *prefix-maxima* problem is that of computing the maximum value in subarray $[1..j]$ for all $j = 1, \ldots, n$. Algorithms for the range-maxima problem solve the prefix-maxima problem as well. We mention two recent papers that need range maxima (or minima) search: [3] on scaled string matching, and [27] on parallel triconnectivity. Other applications of the algorithm are given in [15].

Using n processors in the deterministic comparison model, the maximum problem, and hence the prefix-maxima and the range-maxima problems, have an $\Omega(\log\log n)$ time lower bound [32]. Reischuk [28] showed that randomization enables computing the maximum in constant time; however, for the prefix-maxima and range-maxima problems with general input, the best deterministic algorithms were so far also the best randomized algorithms: [7] gave a deterministic $O(\log\log n)$ time preprocessing algorithm with linear number of operations for the range-maxima problem, and constant time query retrieval. This improved on Alon and Schieber [2] whose algorithm takes $O(n\alpha(n))$ operations and $O(\log\log n)$ time, but for a more general problem. Gil and Rudolph [18] gave an $O(\log\log n)$ time prefix-maxima algorithm using n processors. Schieber [29] gave an $O(\log\log n)$-time algorithm for prefix maxima that uses $n/\log\log n$ processors.

1.1. Results. We present range-maxima algorithms for two models of parallel computation. (The algorithms also explicitly solve the prefix-maxima problem with respect to the input.) Both algorithms use an optimal number of processors, i.e., the time-processor product is $O(n)$ for the preprocessing stage, and each query is processed by one processor. The preprocessing time is with probability $\geq 1 - 2^{-n^c}$ for some constant $c > 0$ ("almost surely"); the query-processing time is worst case. We have

- *Algorithm for the PRAM model:* Preprocessing in $O(\log^* n)$ time and query retrieval in constant time.
- *Algorithms for the comparison model:* Preprocessing in $O(I_m(n))$ time and query retrieval in $O(m)$ time, for $m = 1, \ldots, \alpha(n)$. The function $I_m(\cdot)$ is the inverse of a function at the m'th level of the primitive recursive hierarchy, and the function $\alpha(\cdot)$ is the inverse Ackermann function. In particular, this gives $O(\alpha(n))$ time for both preprocessing and query response.

Thus, the results improve the running times of the best randomized PRAM and comparison model algorithms (for both prefix maxima and range maxima) from $O(\log\log n)$ to $O(\log^* n)$ and $O(\alpha(n))$, respectively.

SIGNIFICANCE. The $O(\log^* n)$ result is for a problem that is a substantial generalization of the problem of computing the maximum over an array of numbers: The output enables constant-time retrieval of *any* of $O(n^2)$ data rather than providing a single datum. The $O(\alpha(n))$ result is a first example where the star-tree data structure of [8] is used to obtain such a result on *unrestricted* input. Also, this is the first example of an $O(\alpha(n))$ *randomized* result. (This latter result is only for the comparison model, however.) These results add to a growing knowledge base of "nearly constant-time" parallel algorithms (see [33]).

1.2. Related work. We review below related results for the range-maxima problem and for the prefix-maxima problem.

SEQUENTIAL ALGORITHMS. Gabow, Bentley, and Tarjan [15] gave a linear-time preprocessing algorithm for range maxima that results in constant-time query retrieval. For the prefix-maxima problem there is a trivial linear-time algorithm.

RANGE MAXIMA WITH RESTRICTED INPUT. [8] considered the range-maxima problem for restricted input where the difference between each a_i and its successor a_{i+1} is at most k; i.e., $|a_i - a_{i+1}| \leq k$, for $i = 1, \ldots, n-1$. They use it to solve the lowest-common-ancestor problem on trees, assuming that the input is given with some additional information. For a constant k, their preprocessing algorithm runs in $O(\alpha(n))$ parallel time using $n/\alpha(n)$ processors. They provide a trade-off between preprocessing time and query-processing time.

PREFIX MAXIMA WITH RESTRICTED INPUT. [6] gave a prefix-maxima algorithm for input restricted to integers in the range $[1..s]$. The algorithm runs in $O(\log\log\log s)$ time and $O(n)$ operations, using $O(ns^\epsilon)$ space. Using a parallel hashing algorithm [23, 17], this prefix-maxima algorithm can be implemented using only linear space (the time then increases by a factor of $O(\log^* n)$ with high probability). For the case $s = n$, [6] give an algorithm that takes $O(\log^* n)$ time and $O(n)$ operations, using $O(n)$ space. Hagerup [19] includes an $O(\log^* n)$ randomized algorithm for the prefix-maxima problem where the input is a random permutation, and therefore the expected number of *critical elements* is $O(\log n)$ (an element is *critical* if it equals the maximum over its prefix). Note that an arbitrary input may have as many as n critical elements.

Ordered chaining. Given an array A of 0's and 1's, the *ordered chaining* problem is to find for each location in A the nearest location that contains a 1. This problem is easier than prefix-maxima problem (it can be reduced to the prefix-maxima problem whose input is from $[1..n]$). Optimal deterministic algorithms that run in $O(\alpha(n))$ time on a PRAM were given by [8, 25].

Parallel lower bounds. Valiant [32] gave a lower bound of $\Omega(\log\log n)$ time for computing the maximum of n elements using n processors in a comparison model of parallel computation. This lower bound applies for the prefix-maxima problem; the proof does not hold if the input is restricted to a certain domain or for randomized algorithms. The same time lower bound for a deterministic PRAM assuming the input is drawn from a large domain is proven in [24] and [26]. Chandra, Fortune and Lipton [11] considered unbounded fan-in circuits for the prefix-carry problem. They showed that a circuit of depth m requires $\Omega(nI_m(n))$ size, implying an $\Omega(\alpha(n))$ lower bound for the depth of an n-size circuit for ordered chaining, and hence for prefix maxima. Chaudhuri [12] gave an $\Omega(\alpha(n))$ time lower bound for oblivious PRAM algorithms that use n processors to solve the ordered chaining problem.

Postscript. In an independent work, Dietz [14] considered a related problem of building a complete binary heap in parallel for an input array. He gives an $O(\log\log n)$ time deterministic algorithm using $n/\log\log n$ processors (this is optimal), and a randomized $O(\alpha(n))$ time (with high probability) algorithm using $n/\alpha(n)$ processors. Both algorithms are for the comparison model. Note that the heap data structure is insufficient to enable super fast retrieval of range maxima queries.

1.3. Outline. The rest of the paper is organized as follows. In Section 2 some definitions, notations, and basic facts are given. A basic randomized constant-time algorithm for range maxima that uses $n\log n$ processors is described in Section 3. The main algorithms, both for the comparison model and for the PRAM model, are described in Section 4. We conclude with some open problems in Section 5.

2. Basics

2.1. Models of computation. We use two models of parallel computation.

The PRAM model. Our main result is given for the concurrent-read concurrent-write (CRCW) PRAM model of computation. We assume the

Arbitrary-CRCW model, in which if two or more processors try to write into the same memory cell, one of them succeeds but we do not know in advance which one. It turns out, but is not shown in this paper, that our PRAM results actually hold for the weaker Tolerant-CRCW model, in which if two or more processors try to write (even the same value) into the same memory cell, the content of the cell does not change.

We define a *team* of size p to be a set of p processors with consecutive indices $i, i+1, \ldots, i+p-1$, 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".

Valiant's comparison model [32]. Only comparison steps are taken into account. For instance, processor allocation is given free of charge in this model. This model has provided yardsticks for the limits of comparison based parallel algorithms (in [32] for finding the maximum, in [10] for merging, and in [5, 1, 9] for sorting). In order to transform an upper bound within this model into a PRAM upper bound, issues of processor allocation have to be taken into account.

Definition 2.1. *Given an array $A[1..n]$, the* prefix *of $A[i]$, $1 \leq i \leq n$, is the subarray $A[1..i]$; the* suffix *of $A[i]$ is the subarray $A[i..n]$. The* prefix maximum *of an element in A is the maximum over its prefix. The* prefix maxima of A *is an array consisting of the n prefix maxima of A's elements. Given an array A, the* prefix maxima problem *is to compute the prefix maxima of A.*

Definition 2.2. *Given an array $A[1..n]$, let $[i,j]$ denote the integer interval $[i, i+1, \ldots, j-1, j]$, and $\max[i,j]$ denote the maximum of A over the subarray $A[i,j]$; i.e., $\max[i,j] = \max\{A[k] : k \in [i,j]\}$. A* range-maximum query *is given as a pair $\langle i,j \rangle$ and is answered by $\max[i,j]$. The* range-maxima problem *is to preprocess a given array A so that any range-maximum query can be processed within a certain time bound. (Typically, it is required that the processing time is constant or "nearly-constant".)*

2.2. Notations for probabilities. We say that a probability $p(n)$ is *n-polynomial* if $p \geq 1 - n^{-c}$, for some constant $c > 0$. We say that $p(n)$ is *n-exponential* if $p \geq 1 - 2^{-n^c}$, for some constant $c > 0$.

This notation classifies confidence in probabilistic analysis into polynomial and exponential. Precise degrees of polynomials and exponent parameters are of lesser significance. This somewhat coarse notation is convenient for a clearer presentation and analysis as explained below, and in [17].

Given a procedure that succeeds with n-polynomial probability, where n is the size of the problem, we will always be able to repeat it a constant number of times so that its success probability increases to $\geq 1 - n^{-c}$ for *any* constant c, with the only requirement being that a failure may be detected in constant time. (We will not indicate it explicitly in the text.) Indeed, a failure probability of n^{-c_1} is reduced to n^{-c} by $\frac{c}{c_1}$ repetitions. Thus, an algorithm which consists of polynomial number of procedures, each succeeding with n-polynomial probability, will succeed with n-polynomial probability, as well. Since our algorithms never take more than a polynomial number of operations, it will be sufficient to show that each step or procedure succeeds with n-polynomial probability. Clearly, if each step or procedure succeeds with n-exponential probability then so does the algorithm.

We use the following bounds on the tail of the binomial distribution.

FACT 2.3. (CHERNOFF BOUNDS [13, 4]) *Let x be random variable with a binomial distribution and let $\overline{x} = \mathbf{E}(x)$. Then, for $0 < \epsilon < 1$ we have $(1-\epsilon)\overline{x} \leq x \leq (1+\epsilon)\overline{x}$ with $(\epsilon^2\overline{x})$-exponential probability.*

2.3. A thinning-out principle. Suppose that we have n "active" elements and we want to turn them into non-active ones; suppose also that we have at hand a procedure that, using r processors, makes an active element non-active with r-polynomial probability. The thinning-out principle applies this procedure to each of the active elements simultaneously and *independently*. The number n' of remaining active elements is bounded by a binomial variable with parameters (n, r^{-c}), where $c > 0$ can be any constant. Thus, by Chernoff bounds (Fact 2.3), $n' < \frac{3}{2}n/r^c$ with n/r^c-exponential probability (by taking $\epsilon = 1/2$), and the number of active elements has indeed been thinned-out with high probability. This paradigm, where high success probability is converted into a decrease in the number of active elements, was used in fast randomized algorithms [16] and in the log-star paradigm in [23].

In this paper we use the thinning-out principle as follows. We have many subproblems of "small" size, say r, that we try to solve simultaneously. We use two constant time subroutines: a randomized subroutine in which r processors can solve a subproblem with r-exponential probability, and a deterministic subroutine which uses r^b processors, with $b = 2$, 3, or 4 (depending on the subproblem). There are initially n/r subproblems, each allocated with r processors. The randomized subroutine is used for each subproblem, solving it in constant time with r-exponential probability. By using Chernoff bounds, it is shown that the number of unsolved sub-problems is at most n/r^4, with appropriately high probability. To each subproblem r^b processors are allocated (with $b = 2$, 3 or 4); they can solve the subproblem deterministically in constant time. In the comparison model the allocation is given for free. To implement the allocation on a PRAM, we map each representative of an unresolved subproblem into a unique cell in an array of size n/r^b, where r^b processors are standing by. [23] showed how to compute such a mapping using a *linear approximate compaction* algorithm:

FACT 2.4. (LINEAR APPROXIMATE COMPACTION [23]) *Assume that we are given an array of n processors, and a subset $S \subset [1..n]$ of distinguished processors such that $|S| \leq m$. Then each distinguished processor can be allocated with a private cell from an array B of size $4m$ in $\log^* m$ time with n-exponential probability. If $m \log^{(i)} m \leq n$ and B is of size $m \log^{(i)} m$ then the allocation can be done in $O(i)$ time with n-exponential probability, for any $i = 1, \ldots \log^* n$.*

2.4. Constant-time computations of maximum and of range-maxima. The following constant-time algorithms are used as subroutines in our algorithms.

FACT 2.5. ([28]) *The maximum of n given elements can be computed in constant time, using n processors, with n-exponential probability.*

Given $n^{1+\epsilon}$ processors, for any constant $\epsilon > 0$, the maximum over n given elements can be easily computed deterministically in constant time [30, 32]. This algorithm can be generalized for range-maxima preprocessing:

FACT 2.6. ([8]) *Assume that we are given an array A of size n, $n^{1+\epsilon}$ processors, and $O(n^{1+\epsilon})$ space, for any constant $\epsilon > 0$. Then, the array A can be preprocessed for range-maxima (deterministically) in constant time. Specifically, the time for preprocessing is $O(1/\epsilon)$. A query can be processed in constant time using a single processor.*

3. The basic range-maxima algorithm

Lemma 3.1. *Given n elements and $n \log n$ processors, preprocessing takes constant time, with n-exponential probability. Range-maxima query retrieval will take constant time.*

Assume that n elements $a_1, \ldots, a_n$ are given in an array A. First we present the output of the preprocessing algorithm, and show how the output can be used for processing a range-maximum query. This is followed by the basic preprocessing algorithm.

3.1. Output of the basic preprocessing algorithm. Array A is partitioned into subarrays $A_1, \ldots, A_{n/\log^k n}$ where $A_j = [a_{(j-1)\log^k n+1}, \ldots, a_{j \log^k n}]$, (i.e., each subarray is of size $\log^k n$), where $k > 0$ is some constant to be determined later. The output consists of local data structures, one for each subarray, and a global data structure for the $n/\log^k n$ subarrays, as follows:

(1) A local table for each subarray A_j. The table enables constant time retrieval of any range-maximum query within the subarray.

(2) Array $B = b_1, \ldots, b_{n/\log^k n}$ consisting of the maximum value in each subarray, i.e., $b_j = \max\{A_j\}$.

(3) A complete binary tree whose leaves are (the values) $b_1, \ldots, b_{n/\log^k n}$. For each internal node v of the tree let $L(v)$ be the subarray consisting of the (values of the) leaves of v. Node v also holds two more arrays: (i) array P_v, consisting of the prefix maxima of $L(v)$; and (ii) array S_v, consisting of the suffix maxima of $L(v)$.

(4) Two arrays of size n each, one containing all prefix-maxima and the other all suffix-maxima with respect to A.

In fact, item (4) is needed only for the recursive statement of the main algorithm in the next section.

3.2. Constant-time query retrieval. Let $\max[i, j]$ be a query to be processed. There are two possibilities:

(i) a_i and a_j belong to the same subarray (of size $\log^k n$): $\max[i, j]$ is computed in $O(1)$ time using the local table that belongs to the subarray.

(ii) a_i and a_j belong to different subarrays. We elaborate below on possibility (ii).

Let right(i) denote the rightmost element in the subarray of a_i and left(j) denote the leftmost element in the subarray of a_j. Then $\max[i,j]$ is the maximum among three numbers:

1. $\max[i, \text{right}(i)]$, the maximum over the suffix of a_i in its subarray
2. $\max[\text{left}(j), j]$, the maximum over the prefix of a_j in its subarray
3. $\max[\text{right}(i)+1, \text{left}(j)-1]$.

The retrieval of the first and second numbers is similar to possibility (i) above. Denote $i' = \lceil i/\log^k n\rceil + 1$ and $j' = \lceil j/\log^k n\rceil - 1$. Then, a_i is in subarray $A_{i'-1}$ and a_j is in subarray $A_{j'+1}$. We discuss retrieval of the third number. This is equivalent to finding the maximum over interval $[b_{i'},\ldots,b_{j'}]$ in B, which is denoted $\max_B[i',j']$.

Let LCA(i',j') be the lowest common ancestor of $b_{i'}$ and $b_{j'}$, $v(i')$ be the child of LCA(i',j') that is an ancestor of $b_{i'}$, and $v(j')$ be the child of LCA(i',j') that is an ancestor of $b_{j'}$. Then $\max_B[i',j']$ is the maximum among two numbers:

1. $\max_B[i', \text{right}(v(i'))]$, the maximum over the suffix of $b_{i'}$ in $v(i')$. We get this from $S_{v(i')}$.
2. $\max_B[\text{left}(v(j')), j']$, the maximum over the prefix of $b_{j'}$ in $v(j')$. We get this from $P_{v(j')}$.

It remains to show how to find LCA(i',j'), $v(i')$, and $v(j')$ in constant time. It was observed in [20] that the lowest common ancestor of two leaves in a complete binary tree can be found in constant time using one processor. (The idea is to number the leaves from 0 to $n-1$; given two leaves i' and j', it suffices to find the most significant bit in which the binary representation of i' and j' are different, in order to get the level of their lowest common ancestor.) Thus LCA(i',j'), and therefore also $v(i')$ and $v(j')$, can be found in constant time. Constant time retrieval of $\max[i,j]$ query follows.

3.3. The preprocessing algorithm. The preprocessing algorithm has four steps:

Step 1. Local preprocessing: For each sub-array A_j of size $\log^k n$ allocate a team of $\log^{k+1} n$ processors and preprocess the sub-array for range maxima. This can be done in constant time using Fact 2.6.

Step 2. Take the maximum b_j in each subarray A_j to build array B of size $n/\log^k n$.

Step 3. *Global preprocessing:* Preprocess array B for range maxima, using procedure Global Preprocessing below.

Step 4. Compute the prefix maxima for array A: If element a_i is in sub-array A_j, then the prefix-maximum of i is the maximum between the prefix-maximum of i in A_j and the prefix-maximum of b_{j-1} in B. Compute the suffix maxima in a similar manner.

PROCEDURE Global Preprocessing. We first describe the procedure with n-polynomial success probability, and then extend it to n-exponential success probability.

(1) Build a complete binary tree over array B; allocate $\log^{k+1} n$ processors to each element in B ($\log^k n$ per each of its ancestors).

(2) For each internal node v compute the maximum $m(v)$ over its array of leaves $L(v)$ as follows. Replace each leaf in $L(v)$ by $\log^k n$ copies — one per allocated processor; put these copies into an array $L_1(v)$ (of size $|L(v)| \cdot \log^k n$) and apply the algorithm of Fact 2.5 to $L_1(v)$. As is shown in Claim 3.2 below, making $\log^k n$ copies of each leaf enables this step to run in constant time with n-polynomial probability.

(3) For each internal node v compute the prefix maxima and suffix maxima over its array $L(v)$, as follows. Consider a leaf l in the array of v. Let $LS_v(l)$ be the set of left siblings of the nodes which are on the path from l to v. The leaves in arrays $L(u)$ of nodes u in $LS_v(l)$, together with l itself, are exactly the same leaves as the leaves in the prefix of l in $L(v)$. Therefore, the prefix maximum of l in $L(v)$ is the maximum over $\{m(u) : u \in LS_v(l) \cup l\}$. We allocate $(\log n)^2$ processors to l, and compute its prefix maximum in constant time, using Fact 2.6. (Note that $|LS_v(l)| \leq \log n$.)

CLAIM 3.2. *In Step 2 of Procedure* Global Preprocessing, *all maxima computations terminate in constant time with n-polynomial probability.*

PROOF. Consider a level j in the binary tree in which each node v has an array of size L $(= |L(v)| = 2^{j-1})$ and an extended array $L_1(v)$ of size L_1 $(= |L_1(v)| = 2^{j-1} \cdot \log^k n)$. In level j there are $n/L\log^k n$ nodes, and using Fact 2.5 the maximum can be computed for each node in constant time with $(L\log^k n)$-exponential probability. The probability that *all* maxima computations in level j take constant time is therefore $\geq 1 - (n/L\log^k n) \cdot 2^{-(L\log^k n)^\epsilon} \geq 1 - n/\log^k n \cdot 2^{-\log^{k\epsilon} n}$ for some constant $\epsilon > 0$, which is n-polynomial for any constant $k > 1/\epsilon$. This implies that the maxima computations for all nodes in all levels will also be completed in constant time, with n-polynomial probability. □

EXTENSIONS FOR n-EXPONENTIAL PROBABILITY. We use the thinning-out principle. Consider a level in the binary tree in which each internal node v has an array of size L. As in the proof of Claim 3.2, the maximum computation for v was done in Step (2) with $(L\log^k n)$-exponential probability. Let δ be some constant, $1/4 > \delta > 0$. If $L \geq n^\delta$ then the maximum computation with respect to v takes constant time with n-exponential probability. It follows that all maxima computations in this level take constant time with n-exponential probability (since the number of nodes is polynomial). Assume therefore that $L < n^\delta$. By Chernoff bounds (Fact 2.3) the number of failing nodes (in the level) is at most n/L^3 with (n/L^3)-exponential probability, and hence with n-exponential probability. We therefore add the following step:

(2b) Allocate in constant time L^2 processors to each failing node in the level, and compute the maximum for each allocated node in constant time by comparing each of the $\binom{L}{2}$ possible pairs in parallel. For nodes with $L \leq \log^{k/2} n$, L^2 processors are already allocated (see Step 1). For all other nodes the allocation is done using linear approximate compaction (Fact 2.4), as follows. Let S be the set of failing nodes, and let $m = n/L^3$. Assume that $|S| \leq m$—this holds with n-exponential probability. Since $L \geq \log n$ we have $m\log m \leq n/L^2 < n$. Therefore, we can use linear approximate compaction to map S injectively into an array B of size $m\log m$, where each cell represents a team of size L^2. By Fact 2.4, the allocation takes $O(1)$ time, with n-exponential probability.

We have

CLAIM 3.3. *Following Step 2b, all maxima computations terminate with n-exponential probability.*

4. The main algorithm

THE INVERSE-ACKERMANN FUNCTION. We define the inverse-Ackermann function as in [8]. (For a more standard definition see [31].) 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 $*f$: $*f(n) = minimum\{i \mid f^{(i)}(n) \leq 1\}$ (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_k 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) $I_k = *I_{k-1}$ for $k \geq 2$. 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 $\alpha(n) = minimum\ \{i \mid I_i(n) \leq i\}$.
Comment. Ackermann's function is defined following [21] as follows: $A_1(n) = 2n$ and $A_k(n) = A_{k-1}^{(n)}(1)$ for $k \geq 2$. It is interesting to note that I_k is actually the inverse of the k^{th} recursion level of A, the Ackermann function. Namely: $I_k(n) = minimum\ \{i \mid A_k(i) \geq n\}$ or $I_k(A_k(n)) = n$. The definition of $\alpha(n)$ is equivalent to the more often used (but perhaps less intuitive) definition: $minimum\ \{i \mid A_i(i) \geq n\}$.

THE STAR-TREE DATA STRUCTURE. Let n be a 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 Figure 4.1). 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 $TRL(m)$). 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 $BT(m)$.

Number of children per node	Number of leaves per node	Level of the tree
$I^{(0)}_{m-1}(n)/I^{(1)}_{m-1}(n)$	$I^{(0)}_{m-1}(n) = n$	1
$I^{(1)}_{m-1}(n)/I^{(2)}_{m-1}(n)$	$I^{(1)}_{m-1}(n)$	2
$I^{(2)}_{m-1}(n)/I^{(3)}_{m-1}(n)$	$I^{(2)}_{m-1}(n)$	3
0	1	$*I_{m-1}(n) + 1$ $= I_m(n) + 1$

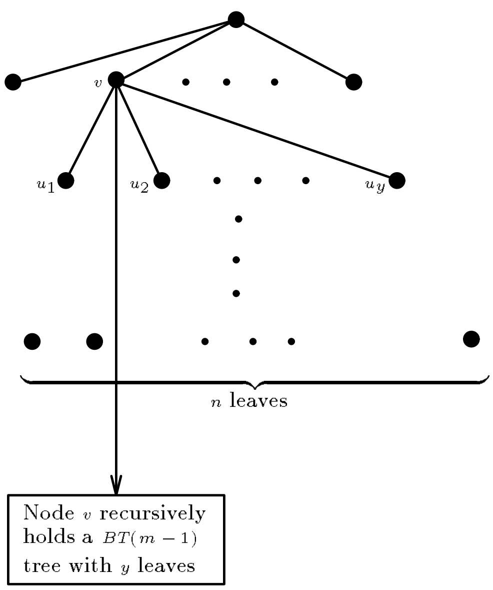

Figure 4.1: $BT(m)$ – m^{th} recursive star-tree

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 [8] and is used by the range-maxima algorithm below. The algorithm on the PRAM model uses only $BT(3)$ — the third recursion level (see Figure 4.2).

Number of children per node	Number of leaves per node	Level of the tree
$\log^{(0)}(n)/\log^{(1)}(n)$	$\log^{(0)}(n) = n$	1
$\log^{(1)}(n)/\log^{(2)}(n)$	$\log^{(1)}(n)$	2
$\log^{(2)}(n)/\log^{(3)}(n)$	$\log^{(2)}(n)$	3
0	1	$*\log(n) + 1$ $= \log^*(n) + 1$

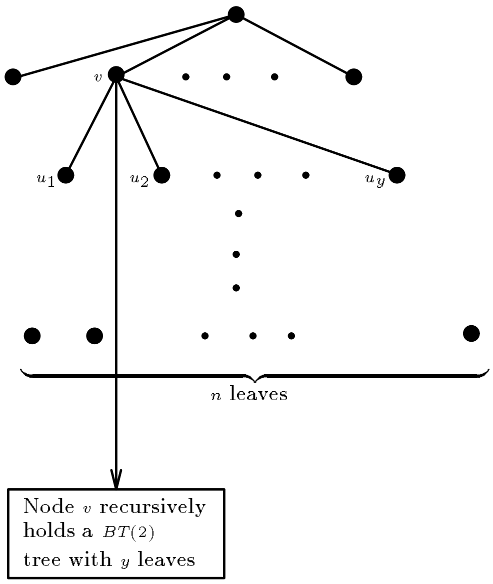

Figure 4.2: $BT(3)$ – the third recursive star-tree

Theorem 4.1 below gives the $O(\log^* n)$ time result for the PRAM model. Theorem 4.2 gives a general trade-off result between running time of the preprocessing algorithm and retrieval time on the comparison model.

Theorem 4.1. *There exists a PRAM preprocessing algorithm for the range-maxima problem that, using an optimal number of processors and linear space, takes $O(\log^* n)$ time with n-exponential probability. Subsequently, each query takes constant (worst case) time using one processor.*

THEOREM 4.2. *For any m, $m = 2, \ldots, \alpha(n)$, there exists a preprocessing algorithm for the range-maxima problem that runs in $O(m + I_m(n))$ time using an optimal number of processors in the comparison model with n-exponential probability. Subsequently, each query takes $O(m)$ (worst case) time using one processor.*

We give below a non-optimal preprocessing algorithm, and show how it leads to an optimal algorithm. The only difference between the algorithm for the PRAM model and the algorithm for the comparison model is that for the PRAM model we must explicitly implement allocation of processors to tasks. The only place such implementation is not straightforward is in Step 4 of the preprocessing algorithm below. We only know how to implement it within the stated bounds for $m = 3$. We therefore describe the algorithm for the comparison model for arbitrary m, and explain how to implement on a PRAM the allocations steps for $m = 3$.

LEMMA 4.3. *There exists a preprocessing algorithm that takes m time units, using $nI_m(n)$ processors and $O(nI_m(n))$ space with n-exponential probability, where a time unit consists of c steps, for a sufficiently large constant $c > 0$. Subsequently, each query takes m time units using one processor. In addition, the preprocessing algorithm explicitly finds prefix maxima and suffix maxima (and, therefore, a prefix-maximum or a suffix-maximum query can be answered in constant time using one processor). For $m = 3$ the algorithm works on the PRAM model.*

We present the algorithm for a given $m > 2$, assuming inductively that an algorithm for $m - 1$ is already given. The base of the induction is the basic algorithm of Section 3. The algorithm is presented in a similar way to that algorithm.

4.1. Output of the m^{th} preprocessing algorithm — $\mathsf{Table}(m)$. The output consists of five items:

(1) A table for each consecutive $I_m^k(n)$-size subarray $a_{jI_m^k(n)+1}, \ldots, a_{(j+1)I_m^k(n)}$, $0 \leq j \leq n/I_m^k(n) - 1$. The table enables constant time retrieval of any range-maximum query within the subarray. *Comment.* The notation $I_m^k(n)$ means $(I_m(n))^k$ where $I_m(n)$ is defined earlier.

(2) Array $B = b_1, \ldots, b_{n/I_m^k(n)}$ consisting of the maximum in each subarray.

(3) $TRL(m)$, the top recursion level of the recursive star-tree $BT(m)$, whose leaves are $b_1, \ldots, b_{n/I_m^k(n)}$. Each internal node v of $TRL(m)$ holds an array P_v and an array S_v with an entry for each leaf of v. These arrays hold (as in the binary tree for the basic algorithm) prefix maxima and suffix maxima with respect to the leaves of v.

(4) Let $u_1, \ldots, u_y$ be the children of v, an internal node of $TRL(m)$. Denote by $\max(u_i)$ the maximum over the leaves of node u_i, $1 \le i \le y$. Each such node v has recursively the output $\mathsf{Table}(m-1)$ of the $(m-1)^{st}$ preprocessing algorithm with respect to the input $\max(u_1), \ldots, \max(u_y)$. In some cases node v will have a table $\mathsf{Naive\text{-}table}$ of size y^2, consisting of answers to all possible range-maxima queries, instead of $\mathsf{Table}(m-1)$.

(5) Two arrays of size n each, one containing all prefix maxima and the other all suffix maxima with respect to A.

4.2. How to retrieve a query $\max[i,j]$ **in O(m) time.** We distinguish two possibilities:

(i) a_i and a_j belong to the same subarray A_l (of size $I_m^k(n)$). Then $\max[i,j]$ is computed in $O(1)$ time using the table that belongs to the subarray (output item (1)).

(ii) a_i and a_j belong to different subarrays. We elaborate on possibility (ii).

Again, let right(i) denote the rightmost element in the subarray of a_i and left(j) denote the leftmost element in the subarray of a_j. Then $\max[i,j]$ is the maximum among three numbers:

1. $\max[i, \mathrm{right}(i)]$.
2. $\max[\mathrm{left}(j), j]$.
3. $\max[\mathrm{right}(i)+1, \mathrm{left}(j)-1]$.

The retrieval of the first and second numbers is similar to possibility (i) above. Denote $i' = \lceil i/I_m^k(n) \rceil + 1$ and $j' = \lceil j/I_m^k(n) \rceil - 1$. Retrieval of the third number is equal to finding the maximum over interval $[b_{i'}, \ldots, b_{j'}]$ in B which is denoted $\max[i', j']$.

Let $\mathrm{LCA}(i', j')$ be the lowest common ancestor of $b_{i'}$ and $b_{j'}$ in $TRL(m)$, $v(i')$ be the child of $\mathrm{LCA}(i', j')$ that is an ancestor of $b_{i'}$, and $v(j')$ be the child of $\mathrm{LCA}(i', j')$ that is an ancestor of $b_{j'}$. Then $\max_B[i', j']$ is (recursively) the maximum among three numbers:

1. $\max_B[i', \text{right}(v(i'))]$, the maximum over the suffix of $b_{i'}$ in $v(i')$.

2. $\max_B[\text{left}(v(j')), j']$, the maximum over the prefix of $b_{j'}$ in $v(j')$.

3. $\max_B[\text{right}(v(i')) + 1, \text{left}(v(j')) - 1]$.

The first two numbers are precomputed in $TRL(m)$ (output item (3)). The first number can be found in $S_{v(i')}$ and the second number in $P_{v(j')}$. The third number will be recursively derived from $\mathsf{Table}(m-1)$ at node LCA(i', j'). It is the maximum among $O(m)$ precomputed numbers — $O(1)$ numbers from each recursion level. If instead of $\mathsf{Table}(m-1)$ there is $\mathsf{Naive\text{-}table}$ then the third number is trivially found in constant time. For the PRAM algorithm, we need to show how to find nodes LCA(i', j'), $v(i')$ and $v(j')$ in constant time. The idea, as presented in [8], is essentially as follows. Since $TRL(m)$ is balanced, it is sufficient to build a table that will enable us to find the LCA for any two leaves of *one* of the children of the root. The number of leaves of a child of the root is small enough that the table can be built in constant time.

4.3. The preprocessing algorithm for m**.** The algorithm is described inductively on m, the level of recursion.

The inductive hypothesis. There exists a preprocessing algorithm that takes $m-1$ time units, using $nI_{m-1}(n)$ processors with n-exponential probability, where a time unit consists of c steps, for a sufficiently large constant $c > 0$. Subsequently, each query takes $m-1$ time units using one processor. In addition, the preprocessing algorithm explicitly finds prefix maxima and suffix maxima.

The Base. The base is the basic range-maxima algorithm of Lemma 3.1.

The inductive step.

Step 1. Preprocess the local range-maxima problems:
Partition A into subarrays of $I_m^k(n)$ elements each. Allocate $I_m^{k+1}(n)$ processors to each subarray, and apply the (deterministic) constant-time preprocessing algorithm for range-maxima given in Fact 2.6.

Steps 2 and 3 below prepare the input for a global reduced-size problem that is handled in Step 4.

Step 2. Take the maximum in each subarray (computed in Step 1) to build array B of size $n/I_m^k(n)$.

Step 3. Build $BT(m)$ over array B:
Build $TRL(m)$, the top recursion level of a $BT(m)$ tree whose leaves are the elements of B. The definition of $TRL(m)$ implies that each internal node of $TRL(m)$, whose rooted tree has r leaves, has $r/I_{m-1}(r)$ children. For each such internal node v of $TRL(m)$ we keep an array. The array consists of the values of the r leaves of the subtree rooted at v. $TRL(m)$ has $*I_{m-1}(n/I_m^k(n))+1 \leq I_m(n)+1$ levels and thus at most $I_m(n)$ internal levels (where internal levels exclude the leaves but include the root). So, the space needed is $O(n/I_m^k(n))$ per level and $O(n/I_m^{k-1}(n))$ for all levels of $TRL(m)$. We allocate to each leaf at each internal level $I_m^k(n)$ processors and the total number of processors used is thus $nI_m(n)$.

Step 4. For each internal node of $TRL(m)$ build recursively Table$(m-1)$ *for array M, where $M[i]$ is the maximum over the leaves of the i^{th} child of the node. This step takes $m-1$ time units with n-exponential probability:*
We first compute for each internal node the maximum over its leaves in $TRL(m)$ in constant time. This is done for most nodes in Step 4.1 below. The remaining computation is done in Step 4.2 in constant time with n-exponential probability, using the thinning-out principle.

Step 4.1 For each internal node find the maximum over its leaves in $TRL(m)$ in constant time with r-exponential probability, where r is the number of its leaves:
Apply the constant time optimal algorithm of Fact 2.5 using r processors (out of the $rI_m^k(n)$ processors of v).

Step 4.2 Complete the computation of maxima for all internal nodes, with n-exponential probability:
It is sufficient to show n-exponential success probability for each level individually. Consider a level in $TRL(m)$ whose nodes have r leaves each. The number of nodes in this level is at most n/r. By Chernoff bounds (Fact 2.3), the number of nodes for which Step 4.1 failed is at most n/r^2 with (n/r^2)-exponential probability which is n-exponential probability. Allocate r^2 processors to each failing node, and compute the maximum over its array deterministically in constant time by comparing in parallel each of the $\binom{r}{2}$ possible pairs.

PRAM implementation: For $m = 3$ ($I_m(n) = \log^* n$) there are $r = \log^{(i-1)} n'$ leaves at level i of $TRL(3)$, where $n' \leq n$ (specifically, $n' = n/(\log^* n)^k$) for $i = 1, \ldots, \log^* n$. We have n processors for each level. Therefore, for level i, where $r = \log^{(i-1)} n'$, the allocation can be done in $O(i)$ time with n-exponential probability, using the linear approximate compaction algorithm (Fact 2.4).

We now build for each internal node a Table$(m-1)$ for array M. This is done in Step 4.3 for most nodes in $m-1$ time units. The remaining computation is done in Step 4.4 in constant time with n-exponential probability, using the thinning-out principle.

Step 4.3 For each internal node build Table$(m-1)$ *for array* M *in* $m-1$ *time units; succeed for each node with* r*-exponential probability, where* r *is the number of its leaves:*
We focus on internal node v having r leaves in $TRL(m)$. Each of its $r/I_{m-1}(r)$ children contributes its maximum to build array M. We use recursively the (inductively assumed) algorithm for $m-1$ to build Table$(m-1)$ for M. Using r processors (out of the $rI_m^k(n)$ processors of v) and $m-1$ time units, Table$(m-1)$ is built for each node with r-exponential probability.

Step 4.4 Complete the computation of Table$(m-1)$ *for all internal nodes with* n*-exponential probability:*
If $r \geq n^{1/6}$ then we are done. Assume that $r < n^{1/6}$.

As before, it is sufficient to show n-exponential success probability for each level individually. Consider a level in $TRL(m)$ whose nodes have r leaves each. The number of nodes in this level is at most n/r. By Chernoff bounds (Fact 2.3), the number of nodes for which Step 4.3 failed is at most n/r^4 with (n/r^4)-exponential probability which is n-exponential probability. Allocate r^4 processors to each failing node, and build Naive-table in a straight forward manner in constant time.

PRAM implementation: Similar to the implementation in Step 4.2.

Step 5. For each leaf, compute deterministically its prefix maximum and suffix maximum with respect to the leaves of each of its ancestors:
For each internal node v we compute its prefix maxima (suffix maxima are computed similarly). That is, for each leaf b_i of v we need to find the maximum over the prefix of b_i with respect to the leaves of node v. For this, the maximum 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 b_i and let $u_1, \ldots, u_y$ denote its children, which are at level l. Suppose u_j is an ancestor of b_i. We take the prefix maximum over the leaves of $u_1, \ldots, u_{j-1}$. This prefix maximum is computed in Step 4

(recall that Table$(m-1)$ contains explicitly all prefix maxima and suffix maxima). Finally, b_i is also added to the list. This maximum computation can be done in constant time using $I_m^2(n)$ processors, (out of the $I_m^k(n)$ processors of b_i at v), by comparing each pair of elements. Note that all prefix maxima and all suffix maxima with respect to B are computed (in the root) in this step.

Step 6. Extend the prefix- and suffix-maxima computations from the reduced-size input B to the original input A:
For each a_i we find its prefix maximum and its suffix maximum with respect to A using one processor in constant time. This is similar to Step 4 of the basic algorithm.

This completes the description of the recursive step: Items (1), (2), (3), (4) and (5) of the output were computed (respectively) in steps 1, 2, 5, 4 and 6 above.

Complexity of the recursive step: In addition to the recursive call in Step 4 which takes $m-1$ time units, all steps take constant time which adds to at most one time unit. This totals m time units using $nI_m(n)$ processors. Lemma 4.3 follows.

4.4. The optimal algorithms. In this subsection, we show how to derive the optimal parallel algorithm of Theorem 4.1 and the series of optimal parallel algorithms of Theorem 4.2 from the series of algorithms described in Lemma 4.3.

Step 1. Reduce the size of the problems:
Partition A into subarrays of $I_m^2(n)$ elements each, allocate $I_m(n)$ processors to each subarray and find the maximum in the subarray. This can be done in $O(I_m(n))$ time.

Put the $n/I_m^2(n)$ maxima into an array A'.

Step 2. Apply the algorithm of Lemma 4.3 to the reduced-size problem:
Apply the preprocessing algorithm for m of Lemma 4.3 to A'. This takes $O(m)$ time using $n/I_m(n)$ processors.

Step 3. Preprocess each subarray (of $I_m^2(n)$ elements) so that range-maximum queries within the subarray can be answered in constant time:
Use a parallel variant of the range-maxima algorithm of [15] that was given in [8].

Complexity of the preprocessing algorithm: $O(I_m(n))$ time using $n/I_m(n)$ processors and $O(n)$ space. Retrieval of a range-maximum query takes $O(m)$ time using one processor.
Theorem 4.1 and Theorem 4.2 follow.

5. Conclusion and open problems

The constant time randomized algorithm for the maximum problem proved that randomization can beat deterministic lower bounds in parallel algorithms. In this paper we demonstrated that randomization is powerful enough to enable "nearly-constant" time optimal algorithms for problems which are much more general than the maximum problem. While only describing the most efficient algorithms, one can modify them to get other performances as well, such as a faster PRAM algorithm which uses more processors, or a simpler PRAM algorithm for which the query response is $O(\log^* n)$ time.

Our results suggest several open problems:

- Give a preprocessing algorithm for the PRAM in $O(\alpha(n))$ expected time or show that $\Omega(\log^* n)$ is optimal.
- Prove $\Omega(\alpha(n))$ expected time lower bound for the comparison model or give an $O(1)$ expected time algorithm.
- Recently, MacKenzie [22] gave a lower bound of $\Omega(\log^* n)$ expected time for the linear approximate compaction problem. This implies that the PRAM algorithm cannot be improved by having a better algorithm for linear approximate compaction — one of the main tools for processor allocation in super fast randomized algorithms. Can one find alternative tools for processor allocation?

Acknowledgements

We wish to thank the anonymous referees for their helpful comments regarding the presentation of this paper. The research was partially supported by NSF grants CCR-8906949, and CCR-9111348.

References

[1] N. Alon and Y. Azar, The average complexity of deterministic and randomized parallel comparison-sorting algorithms, *SIAM Journal on Computing*, **17** (1988), 1178–1192.

[2] N. Alon and B. Schieber, Optimal preprocessing for answering on-line product queries, *Tech. Rep. 71/87, Eskenasy Institute of Computer Sciences, Tel Aviv University, Ramat Aviv 69978, Israel*, 1987.

[3] A. Amir, G. M. Landau, and U. Vishkin, Efficient pattern matching with scaling, in *Proc. of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, 1990, 344–357.

[4] D. Angluin and L. G. Valiant, Fast probabilistic algorithms for hamiltonian paths and matchings, *Journal of Computer and System Sciences*, **18** (1979), 155–193.

[5] Y. Azar and U. Vishkin, Tight comparison bounds on the complexity of parallel sorting, *SIAM Journal on Computing*, **16** (1987), 458–464.

[6] O. Berkman, J. JáJá, S. Krishnamurthy, R. Thurimella, and U. Vishkin, Some triply-logarithmic parallel algorithms, in *Proc. of the 31st IEEE Annual Symp. on Foundation of Computer Science*, 1990, 871–881, also in Top-bottom routing around a rectangle is as easy as computing prefix minima, to appear in *SIAM Journal on Computing.*

[7] O. Berkman, B. Schieber, and U. Vishkin, Some doubly logarithmic parallel algorithms based on finding all nearest smaller values, *Tech. Rep. UMIACS-TR-88-79, Univ. of Maryland Inst. for Advanced Computer Studies*, 1988, to appear in *Journal of Algorithms* as Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values.

[8] O. Berkman and U. Vishkin, Recursive *-tree parallel data-structure, in *Proc. of the 30th IEEE Annual Symp. on Foundation of Computer Science*, 1989, 196–202, also in UMIACS-TR-90-40, Institute for Advanced Computer Studies, Univ. of Maryland, 1991, to appear in *SIAM Journal on Computing.*

[9] R. B. Boppana, The average-case parallel complexity of sorting, *Information Processing Letters*, **33** (1989), 145–146.

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

[11] A. K. Chandra, S. Fortune, and R. J. Lipton, Lower bounds for constant depth circuits for prefix problems, in *Proc. of 10th International*

Colloquium on Automata Languages and Programming, Springer LNCS 154, 1983, 109–117.

[12] S. Chaudhuri, Tight bounds for the chaining problem, in *3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, 1991, 62–70.

[13] H. Chernoff, A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations, *Annals of Math. Statistics*, **23** (1952), 493–507.

[14] P. F. Dietz, Heap construction in the parallel comparison tree model, in *Proc. of the Third Scandinavian Workshop on Algorithm Theory, Springer LNCS 621*, 1992, 140–150.

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

[16] J. Gil and Y. Matias, Fast hashing on a PRAM—designing by expectation, in *Proc. of the Second Annual ACM-SIAM Symposium on Discrete Algorithms*, 1991, 271–280.

[17] J. Gil, Y. Matias, and U. Vishkin, Towards a theory of nearly constant time parallel algorithms, in *Proc. of the 32nd IEEE Annual Symp. on Foundation of Computer Science*, 1991, 698–710.

[18] J. Gil and L. Rudolph, Counting and packing in parallel, in *Proc. of the International Conference on Parallel Processing and Applications*, 1986, 1000–1002.

[19] T. Hagerup, Fast parallel generation of random permutations, in *Proc. of 18th International Colloquium on Automata Languages and Programming, Springer LNCS 510*, 1991, 405–416.

[20] D. Harel and R. E. Tarjan, Fast algorithms for finding nearest common ancestors, *SIAM Journal on Computing*, **13(2)** (1984), 338–355.

[21] S. Hart and M. Sharir, Non linearity of davenport-schinzel sequences and generalized path compression schemes, *Combinatorica*, **6(2)** (1986), 151–177.

[22] P. D. MacKenzie, Load balancing requires $\Omega(\log^* n)$ expected time, in *Proc. of the Third Annual ACM-SIAM Symposium on Discrete Algorithms*, 1992, 94–99.

[23] Y. Matias and U. Vishkin, Converting high probability into nearly-constant time—with applications to parallel hashing, in *Proc. of the 23rd Ann. ACM Symp. on Theory of Computing*, 1991, 307–316, also in Tech. Rep. UMIACS-TR-91-65, Institute for Advanced Computer Studies, Univ. of Maryland, 1991.

[24] 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*, 1985, 532–540.

[25] P. L. Ragde, The parallel simplicity of compaction and chaining, in *Proc. of 17th International Colloquium on Automata Languages and Programming, Springer LNCS 443*, 1990, 744–751, to appear in *Journal of Algorithms*.

[26] P. L. Ragde, Towards lower bounds for parallel computation over moderate sized domains, Manuscript, 1990.

[27] V. Ramachandran and U. Vishkin, Efficient parallel triconnectivity in logarithmic parallel time, in *Proc. of the 3rd Aegean Workshop on Parallel Computing, Springer LNCS 319*, 1988, 33–42.

[28] R. Reischuk, Probabilistic parallel algorithms for sorting and selection, *SIAM Journal on Computing*, **14(2)** (1985), 396–409.

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

[30] Y. Shiloach and U. Vishkin, Finding the maximum, merging, and sorting in a parallel computation model, *Journal of Algorithms*, **2** (1981), 88–102.

[31] R. E. Tarjan, Efficiency of a good but not linear set union algorithm, *Journal of the ACM*, **22(2)** (1975), 215–225.

[32] L. G. Valiant, Parallelism in comparison problems, *SIAM Journal on Computing*, **4** (1975), 348–355.

[33] U. Vishkin, Structural parallel algorithmics, in *Proc. of 18th International Colloquium on Automata Languages and Programming, Springer LNCS 510*, 1991, 363–380.