SIAM J. COMPUT.
Vol. 22, No. 2, pp. 221–242, April 1993

RECURSIVE STAR-TREE PARALLEL DATA STRUCTURE*

OMER BERKMAN† AND UZI VISHKIN†‡

Abstract. This paper introduces a novel parallel data structure called the recursive star-tree (denoted "*-tree"). For its definition a generalization of the $*$ functional is used (where for a function f $*$ $f(n) = \min\{i | f^{(i)}(n) \leq 1\}$ and $f^{(i)}$ is the ith iterate of f). Recursive *-trees are derived by using recursion in the spirit of the inverse Ackermann function.

The recursive *-tree data structure leads to a new design paradigm for parallel algorithms. This paradigm allows for extremely fast parallel computations, specifically, $O(\alpha(n))$ time (where $\alpha(n)$ is the inverse of the Ackermann function), using an optimal number of processors on the (weakest) concurrent-read, concurrent-write parallel random-access machine (CRCW PRAM).

These computations need only constant time, and use an optimal number of processors if the following nonstandard assumption about the model of parallel computation is added to the CRCW PRAM: an extremely small number of processors each can write simultaneously into different bits of the same word.

Applications include finding lowest common ancestors in trees by a new algorithm that is considerably simpler than the known algorithms for the problem, restricted domain merging, parentheses matching, and a new parallel reducibility.

Key words. parallel algorithms, parallel data structures, lowest common ancestors

C.R. subject classification. F. 2. 2

1. Introduction.

The model of parallel computation 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). We use the weakest Common CRCW PRAM model, in which only concurrent writes of the value one are allowed. Given two parallel algorithms for the same problem, one is *more efficient* than the other if (1) primarily, its time–processor product is smaller and (2) secondarily (but important), its parallel time is smaller. An *optimal* parallel algorithm is an algorithm whose time–processor product matches the sequential complexity of the problem (which in this paper is always linear). A *fully parallel* algorithm is a parallel algorithm that runs in constant time and uses an optimal number of processors. An *almost fully parallel* algorithm is a parallel algorithm that runs in time $\alpha(n)$ (the inverse of the Ackermann function) while using an optimal number of processors.

The notion of a fully parallel algorithm represents an ultimate theoretical goal for designers of parallel algorithms. Research on lower bounds for parallel computation (see references later) indicates that for nearly any interesting problem this goal is not achievable. These same results also preclude almost fully parallel algorithms for the same problems. Therefore, any result that approaches this goal is somewhat surprising.

*Received by the editors April 16, 1990; accepted for publication (in revised form) January 8, 1992. Based on "Recursive *-Tree Parallel Data-Structure" by O. Berkman and U. Vishkin which appeared in the 30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, October 30–November 1, 1989, pp. 196–202. ©1989 IEEE.

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

‡University of Maryland Institute for Advanced Computer Studies (UMIACS), University of Maryland, College Park, Maryland 20742; and Department of Electrical Engineering, University of Maryland, College Park, Maryland 20742; and the Department of Computer Science, Tel Aviv University, Tel Aviv, Israel 69978. The work of this author was partially supported by National Science Foundation grants CCR-8906949 and CCR-9111348.

The class of doubly logarithmic optimal parallel algorithms and the challenge of designing such algorithms is discussed in [BBG+89]. The class of almost fully parallel algorithms represents an even stricter demand.

There is a remarkably small number of problems for which there exist optimal parallel algorithms that run in $o(\log\log n)$ time. These problems include (a) OR and AND of n bits; (b) finding the minimum among n elements, where the input consists of integers in the domain $[1, \cdots, n^c]$ for a constant c (see [FRW84]); (c) $\log^{(k)} n$ coloring of a cycle, where $\log^{(k)}$ is the kth iterate of the log function and k is constant [CV86b]; (d) some probabilistic computational geometry problems [Sto88]; (e) matching a pattern string in a text string, following a preprocessing stage in which a table based on the pattern is built [Vis91].

Not only is the number of such upper bounds small, there is evidence that for almost any interesting problem an $o(\log\log n)$-time optimal upper bound is impossible. We mention time lower bounds for a few very simple problems. Clearly, these lower bounds apply to more involved problems. For brevity, only lower bounds for optimal speed-up algorithms are stated: (a) parity of n bits, for which the time lower bound is $\Omega(\log n/\log\log n)$, following from the lower bound of [Has86] for circuits in conjunction with the general simulation result of [SV84] or from [BH87]; (b) finding the minimum among n elements, for which the lower bound is $\Omega(\log\log n)$ on a comparison model [Val75]; (c) merging two sorted arrays of numbers, for which the lower bound is similar to (b) [BH85].

The main contribution of this paper is a parallel data structure called a *recursive *-tree*. This data structure also provides a new paradigm for parallel algorithms. There are two known examples for which tree-based data structures provide a "skeleton" for parallel algorithms: (1) balanced binary trees, where the depth of such a tree with n leaves is $\log n$, and (2) "doubly logarithmic" balanced trees, where the depth of such a tree with n leaves is $\log\log n$ (each node of a doubly logarithmic tree, whose rooted subtree has x leaves, has $\sqrt{x}$ children). Balanced binary trees are used in the prefix-sums algorithm of [LF80] (perhaps the most heavily used routine in parallel computation) and in many other logarithmic-time algorithms. [BBG+89] shows how to apply doubly logarithmic trees for guiding the flow of the computation in several doubly logarithmic algorithms (including some previously known algorithms). Similarly, the recursive *-tree data structure provides a new pattern for almost fully parallel algorithms.

To be able to list results obtained by application of *-trees, we define the following family of extremely slow-growing functions. Our definition is direct. A subsequent comment explains how this definition leads to an alternative definition of the inverse Ackermann function. For a more standard definition see [Tar75]. We note that such a direct definition is implicit in several "inverse-Ackermann-related" serial algorithms (e.g., [HS86]).

The inverse-Ackermann function. Consider a real function f. Let $f^{(i)}$ denote the ith 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 consider only functions for which this minimum is well defined). (The function $\log^*$ will thus be denoted $*\log$.)

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 $\alpha(n) = \min\{i \mid I_i(n) \leq i\}$.

Comment. Ackermann's function is defined following [HS86] as follows: $A_1(n) = 2n$ and $A_k(n) = A_{k-1}^{(n)}(1)$ for $k \geq 2$.

Note that I_k is actually the inverse of the kth recursion level of A, the Ackermann function, namely, $I_k(n) = \min\{i \mid A_k(i) \geq n\}$ or $I_k(A_k(n)) = n$. The definition of $\alpha(n)$ is equivalent to the more often used (but perhaps less intuitive) definition: $\min\{i \mid A_i(i) \geq n\}$.

*Applications of recursive *-trees.*

1. *The lowest-common-ancestor* (LCA) *problem.* Suppose a rooted tree T is given for preprocessing. The preprocessing should enable a single processor to process quickly queries of the following form: Given two vertices u and v, find their lowest common ancestor in T.

Results. (i) Preprocessing is in $I_m(n)$ time and uses an optimal number of processors. (The space complexity here and throughout the paper is linear unless otherwise specified.) Queries will be processed in $O(m)$ time, that is, $O(1)$ time for constant m. A more specific result is (ii) almost fully parallel preprocessing and $O(\alpha(n))$ time for processing a query. These results assume that the Euler tour of the tree and the level of each vertex in the tree are given. Without this assumption the time for preprocessing is $O(\log n)$ with an optimal number of processors, and each query can be processed in constant time. For a serial implementation the preprocessing time is linear and a query can be processed in constant time.

Our algorithm for the LCA problem is new and is based on an approach that is completely different from the serial algorithm of Harel and Tarjan [HT84] and the simplified and parallelizable algorithm of Schieber and Vishkin [SV88]. Its serial version is considerably simpler than these two algorithms. Specifically, consider the Euler tour of the tree and replace each vertex in the tour by its level. This gives a sequence of integers. Unlike previous approaches, the new LCA algorithm is based only on an analysis of this sequence of integers. In particular, answering an LCA query converts to finding the minimum over some range $[i, i+1, \cdots, j]$ of the sequence of levels. (We define the range-minima problem formally and discuss previous results for it in §3.) This provides another interesting example of the quest for parallel algorithms also enriching the field of serial algorithms. Algorithms for quite a few problems use an LCA algorithm as a subroutine. We mention some: (1) strong orientation [Vis85]; (2) computing an open ear decomposition and st-numbering of a biconnected graph [MSV86] (also, [FRT89] and [RR89] use as a subroutine an algorithm for st-numbering and thus also the LCA algorithm); (3) approximate string matching on strings [LV89] and on trees [SZ89] and retrieving information on strings from their suffix trees [AIL$^+$88].

2. *The all-nearest-zero-bit problem.* Let $A = (a_1, a_2, \cdots, a_n)$ be an array of bits. For each bit a_i find the nearest-zero bit both to its left and right.

Result. The algorithm is almost fully parallel.

A similar problem is considered in [CFL83], where the motivation is circuits.

3. *The parentheses-matching problem.* Suppose a legal sequence of parentheses is given. For each parenthesis find its mate.

Result. Assuming the level of nesting of each parenthesis is given, we have an almost fully parallel algorithm. Without this assumption $T = O(\log n / \log\log n)$ if an optimal number of processors is used.

Parentheses matching in parallel is considered in [AMW89], [BSV88], [BV85], and [DS83]. We are putting the algorithm for parentheses matching into a later paper [BV91] to keep the present paper to a reasonable length.

4. *Restricted-domain merging.* Let $A = (a_1, \cdots, a_n)$ and $B = (b_1, \cdots, b_n)$ be two

nondecreasing lists whose elements are integers drawn from the domain $[1, \cdots, n]$. The problem is to merge them into a sorted list.

Result. The algorithm is almost fully parallel.

Merging in parallel is considered in [Van89], [BH85], [Kru83], [SV81], and [Val75]. We presented the merging algorithm in another paper [BV90], where it is implemented on the less powerful concurrent-read, exclusive-write (CREW PRAM) model with the same bounds, and the restricted-domain limitation is somewhat relaxed by using unrelated techniques.

5. *Almost fully parallel reducibility*. Let A and B be two problems. Suppose that any input of size n for problem A can be mapped into an input of size $O(n)$ for problem B. Such a mapping from A to B is an *almost fully parallel reducibility* if it can be realized by an almost fully parallel algorithm.

Given a convex polygon, the *all-nearest-neighbors* (*ANN*) problem is to find for each vertex of the polygon its nearest (Euclidean) neighbor. Using almost fully parallel reducibilities, we prove the following lower bound for the ANN problem: Any CRCW PRAM algorithm for the ANN problem that uses $O(n \log^c n)$ (for any constant c) processors needs $\Omega(\log \log n)$ time. We note that this lower bound is proved in [SV90] by using a considerably more involved technique.

Fully parallel results. For our fully parallel results we introduce the CRCW-bit PRAM model of computation. In addition to the above definition of the CRCW PRAM, we assume that a few processors can each write simultaneously into different bits of the same word. Specifically, in our algorithms this number of processors is very small and never exceeds $O(I_d(n))$, where d is a constant. Therefore, the assumption strikes us as quite reasonable from the architectural point of view. We believe that the cost for implementing a step of a PRAM on a feasible machine is likely to absorb implementation of this assumption at no extra cost. Nevertheless, we do not advocate adopting the CRCW-bit PRAM as a theoretical substitute for the CRCW PRAM.

Specific fully parallel results.

1. *The lowest-common-ancestor problem*. The preprocessing algorithm is fully parallel if we assume that the Euler tour of the tree and the level of each vertex in the tree are given. A query can be processed in constant time.

2. *The all-nearest-zero-bit problem*. The algorithm is fully parallel.

3. *The parentheses-matching problem*. If we assume that the level of nesting of each parenthesis is given, the algorithm is fully parallel.

4. *Restricted-domain merging*. The algorithm is fully parallel.

Results 3 and 4 are not explicitly given in [BV91] and [BV90]. However, they can be derived from the series of fast algorithms in these papers in a fashion similar to the way the fully parallel algorithm of §4.4 in the current paper is derived from the series of algorithms of Lemma 4.2.1 below.

We elaborate on where our fully parallel algorithms use the CRCW-bit new assumption. The algorithms work by mapping the input of size n into n bits. Then, given any constant d, we derive a value $x = O(I_d(n))$. The algorithms proceed by forming n/x groups of x bits each. Informally, our problem is then to pack all x bits of the same group into a single word and solve the original problem with respect to an input of size x in constant time. This packing is exactly where our almost fully parallel CRCW PRAM algorithms fail to become fully parallel and the CRCW-bit assumption is used. We believe that it is of theoretical interest to determine ways of avoiding such packing and thereby get fully parallel algorithms on a CRCW PRAM without the CRCW-bit assumption and suggest this as open problem.

A repeating motif in the present paper is putting restrictions on the domain of problems. This is the case for the all-nearest-zero-bit problem and the restricted-domain merging problem. Perhaps our more interesting applications concern problems whose input domain is *not explicitly restricted*. This is the case for the LCA problem and the parentheses-matching problem. However, as part of the design of our algorithms for these respective problems, we identified a few subproblems whose input domains are restricted: The level of vertices in the Euler tour is the restricted input for the LCA subproblem, and the level of nesting of each parenthesis is the input for the parentheses-matching subproblem. The Euler tour and prefix sums (by which the level of nesting is computed) are basic techniques for trees and arrays, respectively. A lower bound on their running times is $\Omega(\log n/\log\log n)$. One of the aims of this paper is to determine the added difficulty of each of the two problems, LCA and parentheses matching, beyond the use of these basic (and simple) techniques.

The rest of this paper is organized as follows. Section 2 describes the recursive *-tree data structure, and §3 recalls a few basic problems and algorithms. In §4 the parallel algorithms for LCA and the all-nearest-zero bit are presented. A sequential version of the LCA algorithm, which is considerably simpler than the parallel version, is also outlined. The almost fully parallel reducibility is presented in §5, and §6 discusses how to efficiently compute the functions used in this paper.

2. The recursive *-tree data structure. Let n be a positive integer. We define inductively a series of $\alpha(n) - 1$ trees. For each m, $2 \le m \le \alpha(n)$, a balanced tree with n leaves, denoted $BT(m)$, 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 (*see Fig.* 1). We start with the definition of the *-tree $BT(2)$. $BT(2)$ is simply a complete binary tree with n leaves.

Number of children per node	Number of leaves per node	Level of the tree
$I_1^{(0)}(n)/I_1^{(1)}(n) = 2$	$I_1^{(0)}(n) = n$	1
$I_1^{(1)}(n)/I_1^{(2)}(n) = 2$	$I_1^{(1)}(n) = \frac{n}{2}$	2
$I_1^{(2)}(n)/I_1^{(3)}(n) = 2$	$I_1^{(2)}(n) = \frac{n}{4}$	3
0	1	$*I_1(n) + 1$ $= I_2(n) + 1$ $= \log n + 1$

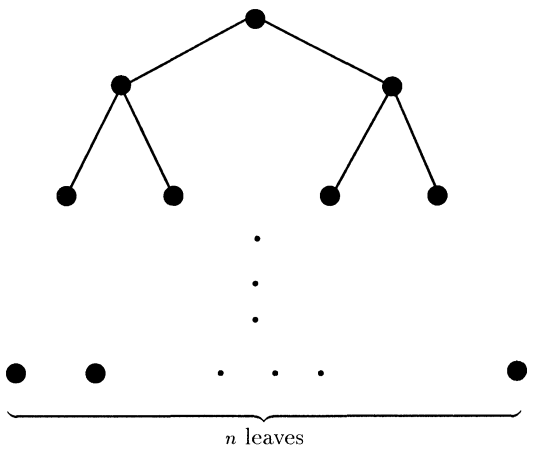

FIG. 1. *A BT* (2) *tree.*

The inductive step (*see Fig.* 2). 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 $I^{(l-1)}_{m-1}(n)/I^{(l)}_{m-1}(n)$ children (we define $I^{(0)}_{m-1}(n)$ to be n). The total number of leaves in the subtree rooted at node v is $I^{(l-1)}_{m-1}(n)$. We refer to the part of the $BT(m)$ tree described so far as the *top recursion level* of $BT(m)$ (denoted $TRL-BT(m)$ for brevity). In addition, node v recursively contains 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	$*I_{m-1}(n) + 1$ $= I_m(n) + 1$

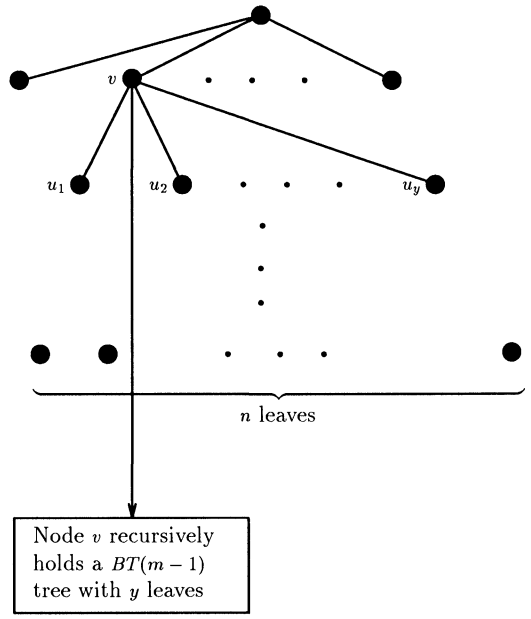

FIG. 2. *BT (m)—mth recursive * −tree.*

There is one key idea that enables our algorithms to be as fast as they are. When the mth tree $BT(m)$ is used 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.

Similar computational structures have appeared in a few contexts. See [AS87] and [Yao82] for generalized range-minima computations, [HS86] for Davenport–Schinzel sequences, and [CFL83] for circuits.

3. Basics. We need the following problems and algorithms.

The Euler-tour technique. Consider a tree $T = (V, E)$ rooted at some vertex r. The Euler-tour technique enables us to compute several problems on trees in logarithmic

time and optimal speedup (see also [TV85] and [Vis85]). The technique is summarized below.

Step 1. For each edge $(v \rightarrow u)$ in T we add its antiparallel edge $(u \rightarrow v)$. Let H denote the new graph. Since the in-degree and out-degree of each vertex in H are the same, H has an Euler path that starts and ends in the root r of T. Step 2 computes this path into a vector of pointers D of size $2|E|$, where for each edge e of H, $D(e)$ gives the successor edge of e in the Euler path.

Step 2. For each vertex v of H we do the following. Let the outgoing edges of v be $(v \rightarrow u_0), \cdots, (v \rightarrow u_{d-1})$. Then $D(u_i \rightarrow v) := (v \rightarrow u_{(i+1) \bmod d})$ for $i = 0, \cdots, d-1$. Vector D now represents an Euler circuit of H. The "correction" $D(u_{d-1} \rightarrow r) :=$ end-of-list (where the out-degree of r is d) gives an Euler path that starts and ends in r.

Step 3. In this step we apply list ranking to the Euler path. This results in ranking the edges so that the tour can be stored in an array. Similarly, we can find for each vertex in the tree its distance from the root. This distance is called the *level* of vertex v. These and other applications of list ranking appear in [TV85]. This list ranking can be performed in logarithmic time with an optimal number of processors by the method of [AM88], [CV86a], or [CV89].

Comments.

1. In §4 we assume that the Euler tour is given in an already ranked form. There we systematically replace each edge (u, v) in the Euler tour by the vertex v. We then add the root of the tree to the beginning of this new array. Suppose a vertex v has l children. Then v appears $l + 1$ times in our array.

2. We note that while advancing from a vertex to its successor in the Euler tour, the level may either increase by one or decrease by one.

Finding the minimum for restricted-domain inputs. Given an array $A = (a_1, a_2, \cdots, a_n)$ where each a_i is an integer between 1 and n, find the minimum value in A.

Fich, Ragde, and Wigderson [FRW84] gave the following parallel algorithm for the restricted-domain minimum-finding problem. It runs in $O(1)$ time with n processors. We use an auxiliary vector B of size n, which is all zero initially. Processor $i, 1 \leq i \leq n$, writes one into location $B(a_i)$. Now the problem is to find the leftmost one in B. Partition B into $\sqrt{n}$ equal-size subarrays. For each such subarray find in $O(1)$ time, using $\sqrt{n}$ processors, whether the subarray contains a one. Apply the $O(1)$ time algorithm of Shiloach and Vishkin [SV81] for finding the leftmost subarray of size $\sqrt{n}$ containing a one, using n processors. 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.

Range-minima problem. Given an array $A = (a_1, a_2, \cdots, a_n)$ of n real numbers, preprocess the array so that for any interval $[a_i, a_{i+1}, \cdots, a_j]$ the minimum over the interval can be retrieved in constant time by using a single processor.

We show how to preprocess A in constant (to be precise, $O(1/\epsilon)$) time with $n^{1+\epsilon}$ processors and $O(n^{1+\epsilon})$ space for any constant ϵ. The preprocessing algorithm uses the following naive parallel algorithm for the range-minima problem. Allocate n^2 processors to each interval $[a_i, a_{i+1}, \cdots, a_j]$, and find the minimum over the interval in constant time, as in [SV81]. This naive algorithm runs in constant time and uses n^4 processors and n^2 space.

The preprocessing algorithm. Suppose some constant ϵ is given. The output of the preprocessing algorithm can be described by means of a balanced tree T with n leaves, where each internal node has $n^{\epsilon/3}$ children. The root of the tree is at level one, and the

leaves are at level $3/\epsilon+1$. Let v be an internal node, and, let $u_1, \cdots, u_{n^{\epsilon/3}}$ be its children. Each internal node v has the following data:

1. $M(v)$, the minimum over its leaves (i.e., the leaves of its subtree);

2. For each $1 \leq i < j \leq n^{\epsilon/3}$ the minimum over the range $\{M(u_i), M(u_{i+1}), \cdots, M(u_j)\}$.

This requires $O(n^{2\epsilon/3})$ space per internal node and a total of $O(n^{1+\epsilon/3})$ space.

The preprocessing algorithm advances through the levels of the tree in $3/\epsilon$ steps, starting from level $3/\epsilon$. At each level each node computes its data by using the naive range-minima algorithm. Each internal node uses $n^{4\epsilon/3}$ processors and $O(n^{2\epsilon/3})$ space, and the whole algorithm uses $n^{1+\epsilon}$ processors, $O(n^{1+\epsilon/3})$ space, and $O(3/\epsilon)$ time.

Retrieval. Suppose we wish to find the minimum, MIN(i, j), over an interval $[a_i, a_{i+1}, \cdots, a_j]$. Let LCA$(a_i, a_j)$ be the lowest common ancestor of leaf a_i and leaf a_j in the tree T. There are two possibilities.

(i) LCA(a_i, a_j) is at level $3/\epsilon$. The data at LCA(a_i, a_j) give MIN(i, j).

(ii) LCA(a_i, a_j) is at level $< 3/\epsilon$. Let x be the number of edges in the path from a_i (or a_j) to LCA(a_i, a_j) (i.e., LCA(a_i, a_j) is at level $3/\epsilon + 1 - x$). Using the tree T, we can represent interval $[i, j]$ as a disjoint union of $2x - 1$ intervals whose minima were previously computed. Formally, let $r(i)$ denote the rightmost leaf of the parent of a_i in T, and, let $l(j)$ denote the leftmost leaf of the parent of a_j in T. MIN$[i, j]$ is the minimum among three numbers.

1. MIN$[i, r(i)]$. The data at the parent of a_i give this information.

2. MIN$[l(j), j]$. The data at the parent of a_j give this information.

3. MIN$[r(i) + 1, l(j) - 1]$. Advance to level $3/\epsilon$ of the tree to get this information recursively.

Complexity. Retrieval of MIN(i, j) takes constant time: The first and second numbers can be looked up in $O(1)$ time. Retrieval of the third number takes $O(3/\epsilon - 1)$ time.

Below we review previous results for the range-minima problem. This review is appropriate since the main technical effort of §4 is a parallel algorithm for an instance of the range-minima problem. [GBT84] gives a linear time algorithm for preprocessing that allows constant-time query retrieval. We discuss these results further in Remark 4.1. An $O(\log \log n)$-time optimal parallel preprocessing algorithm for the range-minima problem is given in [BSV88]. The following generalization of the range-minima problem is considered in [Yao82] and [AS87]. The input array A contains semigroup elements, and a query over the range $[i, j]$ requests the product $a_i \cdot a_{i+1} \cdot \cdots \cdot a_j$, where "$\cdot$" is the semigroup product operator. Both papers give a sequential algorithm for the problem that runs in $O(n\alpha(n))$ time and allows $O(\alpha(n))$ time for query retrieval. In addition, they show that $\Omega(\alpha(n))$ is needed if only linear time is allowed for preprocessing. [AS87] also gives a tradeoff between the sequential time needed for preprocessing and the time for answering a query. The algorithm we give for our instance of the range-minima problem (given in §4) benefits from this approach. However, a serial implementation of our preprocessing algorithm needs only linear time.

4. LCA algorithm. The input to this problem is a rooted tree $T = (V, E)$. Let $n = 2|V| - 1$. We assume that we are given a sequence of n vertices $A = [a_1, \cdots, a_n]$, which is the Euler tour of the input tree, and that we know for each vertex v its level, LEVEL(v), in the tree.

Recall the range-minima problem defined in §3. Below we give a simple reduction from the LCA problem to a restricted-domain range-minima problem, which is an instance of the range-minima problem in which *the difference between each two successive*

numbers for the range-minima problem is exactly one. The reduction takes $O(1)$ time and uses n processors. An algorithm for the restricted-domain range-minima problem is given later, implying an algorithm for the LCA problem.

4.1. Reducing the LCA problem to a restricted-domain range-minima problem. Let v be a vertex in T. Denote by $l(v)$ the index of the leftmost appearance of v in A and by $r(v)$ the index of its rightmost appearance. For each vertex v in T it is easy to find $l(v)$ and $r(v)$ in $O(1)$ time with n processors by using the following (trivial) observations: $l(v)$ is where $a_{l(v)} = v$ and $\text{LEVEL}(a_{l(v)-1}) = \text{LEVEL}(v) - 1$; $r(v)$ is where $a_{r(v)} = v$ and $\text{LEVEL}(a_{r(v)+1}) = \text{LEVEL}(v) - 1$. The claims and corollaries below provide guidelines for the reduction.

CLAIM 1. *Vertex u is an ancestor of vertex v if and only if $l(u) < l(v) < r(u)$.*

COROLLARY 1. *Given two vertices u and v, a single processor can find in constant time whether u is an ancestor of v.*

COROLLARY 2. *Vertices u and v are unrelated (namely, neither is u an ancestor of v nor is v an ancestor of u) if and only if either $r(u) < l(v)$ or $r(v) < l(u)$.*

CLAIM 2. *Let u and v be two unrelated vertices. (By Corollary 2 we may assume without loss of generality that $r(u) < l(v)$.) Then the* LCA *of u and v is the vertex whose level is minimal over the interval $[r(u), l(v)]$ in A.*

The reduction. Let $\text{LEVEL}(A) = [\text{LEVEL}(a_1), \text{LEVEL}(a_2), \cdots, \text{LEVEL}(a_n)]$. Claim 2 shows that after we perform the range-minima preprocessing algorithm with respect to $\text{LEVEL}(A)$, a query of the form $\text{LCA}(u, v)$ becomes a range-minimum query. Observe that the difference between the level of each pair of successive vertices in the Euler tour (and thus each pair of successive entries in $\text{LEVEL}(A)$) is exactly one and therefore the reduction is to the restricted-domain range-minima problem as required.

Remark 4.1. In [GBT84] it is observed that the problem of preprocessing an array so that each range-minimum query can be answered in constant time (this is the range-minima problem defined in §3) is equivalent to the LCA problem. It gives a linear-time algorithm for the former problem by using an algorithm for the latter. This does not look very helpful: We know to solve the range-minima problem on the basis of the LCA problem, and, conversely, we know to solve the LCA problem on the basis of the range-minima problem. Nevertheless, using the restricted-domain properties of our range-minima problem, we show that this cyclic relationship between the two problems can be broken and thereby lead to a new algorithm.

4.2. The restricted-domain range-minima algorithm. Below we define a restricted-domain range-minima problem that is slightly more general than the problem for LEVEL (A). The more general definition enables recursion in the algorithm below. The rest of this section shows how to solve this problem.

The restricted-domain range-minima problem. We are given an integer k and an array $A = (a_i, a_2, \cdots, a_n)$ 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. The parameter k need not be a constant. Preprocess the array $A = (a_1, a_2, \cdots, a_n)$ so that any query $\text{MIN}[i, j]$, $1 \leq i < j \leq n$, requesting the minimum element over the interval $[a_i, \cdots, a_j]$, can be processed quickly with a single processor.

Comment 1. We make the simplifying assumption that $\sqrt{k}$ and all other quantities needed in the paper are always integers. For example, $\log n$ and $n/\log n$ are assumed to be integers. This implies that the root of $BT(3)$ with n leaves has $n/\log n$ children, each with *exactly* $\log n$ leaves in its rooted subtree. To demonstrate that handling the general case is only a matter of technicality let us consider as an example the *-tree

$BT(3)$. When $BT(3)$ is used in the algorithm below with, say, r leaves, we need to store at each internal node an array that contains all leaves of the node. The space is thus $O(r)$ per level and $O(r \log^* r)$ overall. We sketch how to treat general r: The number of children of the root of $BT(3)$ will be $\lceil r/\log r \rceil$, each with $\lceil \log r \rceil$ leaves. This implies that additional $O(r/\log r)$ space is needed for all nodes at level 2 of the tree. At level 3 the additional space needed is $O(\lceil r/\log r \rceil \cdot \lceil \log \log r \rceil)$. Overall, the additional space required is linear. To facilitate memory access we will allocate to each level of $BT(3)$ an additional $O(r)$ space, which is more than is actually needed. This will double the space and number of processors needed by our algorithms but will not change the complexity in terms of "big Oh."

Comment 2. In case the minimum in the interval is not unique, find the minimum element in $[a_i, \cdots, a_j]$ whose index is smallest ("the leftmost minimum element"). Throughout this section, whenever we refer to a minimum over an interval we always mean the leftmost minimum element. Finding the leftmost minimum element (and not just the minimum value) will serve us later.

We start by constructing recursively a series of $\alpha(n) - 1$ parallel preprocessing algorithms for our range-minima problem.

LEMMA 4.2.1. *The algorithm for $2 \leq m \leq \alpha(n)$ runs in cm time, for some constant c, using $nI_m(n) + \sqrt{kn}$ processors and $o(nI_m(n))$ space. The preprocessing algorithm results in a table. With this table any range-minimum query can be processed in cm time with one processor. In addition, the preprocessing algorithm finds explicitly all prefix minima and suffix minima. Therefore, there is no need to do any processing for prefix-minima or suffix-minima queries.*

Our optimal algorithms, whose efficiencies are given in Theorem 4.3.1, are derived from this series of algorithms. We describe the series of preprocessing algorithms. We give first the base of the recursive construction and later the recursive step.

The base of the recursive construction (the algorithm for $m = 2$). To provide intuition for the description of the preprocessing algorithm for $m = 2$ we present first its output and how the output can be used for processing a range-minimum query.

Output of the preprocessing algorithm for $m = 2$:

(1) For each consecutive subarray $a_{j \log^3 n+1}, \cdots, a_{(j+1) \log^3 n}$, $0 \leq j \leq n/\log^3 n - 1$, we keep a table. The table enables constant-time retrieval of any range-minimum query within the subarray.

(2) We have array $B = b_1, \cdots, b_{n/\log^3 n}$, consisting of the minimum in each subarray.

(3) We have a complete binary tree $BT(2)$, whose leaves are $b_1, \cdots, b_{n/\log^3 n}$. Each internal node v of the tree holds an array P_v with an entry for each leaf of v. Consider prefixes that span between $l(v)$, the leftmost leaf of v, and a leaf of v. Array P_v has the minima over all these prefixes. Node v also holds a similar array S_v. For each suffix that spans between a leaf v and $r(v)$, the rightmost leaf of v, array S_v has its minimum.

(4) We have two arrays of size n each; one contains all prefix minima and the other contains all suffix minima with respect to A.

Remark. The reason behind the (seemingly arbitrary) grouping of array A into subarrays of $\log^3 n$ elements each is as follows: The main part of the preprocessing algorithm for $m = 2$ (which is given below) deals with constructing output element (3). To maintain processor count within our desired bounds, the input to this part (which is array B) must be of size $O(n/\log^3 n)$.

LEMMA 4.2.2. *Let m be 2. Then $I_m(n) = \log n$. The preprocessing algorithm runs in $2c$ time for some constant c using $n \log n + \sqrt{kn}$ processors and $o(n \log n)$ space. Retrieval of a query* MIN$[i, j]$ *takes $2c$ time with one processor.*

How does one retrieve a query MIN$[i, j]$ in constant time? There are two possibilities.

(i) a_i and a_j belong to the same subarray (of size $\log^3 n$). MIN(i, j) is computed in $O(1)$ time by using the table that belongs to the subarray.

(ii) a_i and a_j belong to different subarrays. Let right(i) denote the rightmost element in the subarray of a_i and let left(j) denote the leftmost element in the subarray of a_j. MIN$[i, j]$ is the minimum among three numbers:

1. MIN[i,right(i)], the minimum over the suffix of a_i in its subarray;
2. MIN[left$(j), j$], the minimum over the prefix of a_j in its subarray;
3. MIN[right$(i) + 1$, left$(j) - 1$].

The retrieval of the first and second numbers is similar to possibility (i) above. Denote $i_1 = \lceil i/\log^3 n \rceil + 1$ and denote $j_1 = \lceil j/\log^3 n \rceil - 1$. Retrieval of the third number is equivalent to finding the minimum over interval $[b_{i_1}, \cdots, b_{j_1}]$ in B, which is denoted MIN$_B[i_1, j_1]$.

Let x be the lowest common ancestor of b_{i_1} and b_{j_1}, let x_1 be the child of x that is an ancestor of b_{i_1}, and let x_2 be the child of x that is an ancestor of b_{j_1}. MIN$_B[i_1, j_1]$ is the minimum of two numbers:

1. MIN$_B[i_1, r(x_1)]$, the minimum over the suffix of b_{i_1} in x_1 (we obtain this from S_{x_1});
2. MIN$_B[l(x_2), j_1]$, the minimum over the prefix of b_{j_1} in x_2 (we obtain this from P_{x_2}).

How to find x, x_1, and x_2 in constant time remains to be shown. It is observed in [HT84] that the lowest common ancestor of two leaves in a complete binary tree can be found in constant time with one processor. (The idea is to number the leaves from 0 to $n - 1$. Given two leaves i_1 and j_1, it suffices to find the most significant bit in which the binary representation of i_1 and j_1 are different in order to get their lowest common ancestor.) Thus x (and thereby also x_1 and x_2) can be found in constant time. Constant-time retrieval of the MIN(i, j) query follows.

The preprocessing algorithm for $m = 2$.

Step 1. Partition A into subarrays of $\log^3 n$ elements each. Allocate $\log^4 n$ processors to each subarray, and apply the preprocessing algorithm for range minima given in §3 (for $\epsilon = \frac{1}{3}$). This uses $\log^4 n$ processors and $O(\log^3 n \log^{1/3} n)$ space per subarray and $n \log n$ processors and $o(n \log n)$ space overall.

Step 2. Take the minimum in each subarray to build array B of size $n/\log^3 n$. The difference between two successive elements in B is at most $k \log^3 n$.

Step 3. Build $BT(2)$, a complete binary tree whose leaves are the elements of B. For each internal node v of $BT(2)$ we keep an array. The array consists of the values of all leaves in the subtree rooted at v. So the space needed is $O(n/\log^3 n)$ per level and $O(n/\log^2 n)$ for all levels of the tree. We allocate to each leaf at each level $\sqrt{k} \log^2 n$ processors, and the total number of processors used is thus $\sqrt{k}n$.

Step 4. For each internal node find the minimum element over its array. If the minimum element is not unique, the leftmost one is found. We apply the constant- time algorithm mentioned in Remark 3.1. Consider an internal node of size r. After subtracting the first element of the array from each of its elements, we get an array whose elements range between $-kr \log^3 n$ and $kr \log^3 n$. The size of the range, which is $2kr \log^3 n + 1$, does not exceed the square of the number of processors, which is $r\sqrt{k} \log^2 n$, and the algorithm of Remark 3.1 can be applied.

Step 5. For each internal node v we compute P_v (S_v is computed similarly). That is, for each leaf b_i of v we need to find MIN$_B[l(v), i]$ (the minimum over the prefix of b_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 b_i. Let u_1 and u_2 denote the left and right children of u, respectively. If b_i belongs to (the subtree rooted at) u_2, then level l contributes the minimum over u_1. If b_i belongs to u_1, then level l does not contribute anything (actually, level l contributes a large default value so that the minimum computation is not affected). Finally, b_i is also included in the list. This minimum computation can be done in constant time with $\log^2 n$ processors by the algorithm of [SV81]. Note that all prefix minima and all suffix minima of B are computed (in the root) in this step. We note that since $BT(2)$ is a complete binary tree, node u (the ancestor at level $l-1$ of b_i) can be easily found in constant time with one processor ([HT84]).

Step 6. For each a_i we find its prefix minimum and its suffix minimum with respect to A in constant time by using one processor. Let b_j be the minimum representing the subarray of size $\log^3 n$ containing a_i. The minimum over the prefix of a_i with respect to A is the minimum between the prefix of b_{j-1} with respect to B and the minimum over the prefix of a_i with respect to its subarray.

This completes the description of the recursion base: Items (1), (2), (3), and (4) of the output were computed (respectively) in steps 1, 2, 5, and 6 above. The complexity of the recursion base is specified by $O(1)$ time by using $n \log n + \sqrt{k}n$ processors and $o(n \log n)$ space. Lemma 4.2.2 follows.

The recursion step. The algorithm is presented in a way similar to that of the recursion base.

Output of the mth preprocessing algorithm.

(1) For each consecutive subarray $a_{jI_m^3(n)+1}, \cdots, a_{(j+1)I_m^3(n)}$, $0 \leq j \leq n/I_m^3(n) - 1$, we keep a table. The table enables constant-time retrieval of any range-minimum query within the subarray. (The notation $I_m^3(n)$ means $(I_m(n))^3$, where $I_m(n)$ is defined earlier.)

(2) We have array $B = b_1, \cdots, b_{n/I_m^3(n)}$, consisting of the minimum in each subarray.

(3) We have $TRL\text{–}BT(m)$, the top recursion level of (the recursive *-tree) $BT(m)$, whose leaves are $b_1, \cdots, b_{n/I_m^3(n)}$. Each internal node v of $TRL\text{–}BT(m)$ holds an array P_v and array S_v with an entry for each leaf of v. These arrays hold (as in the binary tree for $m = 2$) prefix minima and suffix minima with respect to the leaves of v.

(4) Let $u_1, \cdots, u_y$ be the children of v, an internal node of $TRL-BT(m)$. Denote by $\text{MIN}(u_i)$ the minimum over the leaves of node u_i, $1 \leq i \leq y$. Each such node v has recursively the output of the $(m-1)$th preprocessing algorithm with respect to the input $\text{MIN}(u_1), \cdots, \text{MIN}(u_y)$.

(5) We have two arrays of size n each; one contains all prefix minima and the other contains all suffix minima with respect to A.

How can we retrieve a query $\text{MIN}[i, j]$ in cm time? We distinguish two possibilities.

(i) a_i and a_j belong to the same subarray (of size $I_m^3(n)$). $\text{MIN}(i, j)$ is computed in $O(1)$ time by using the table that belongs to the subarray.

(ii) a_i and a_j belong to different subarrays. Again, let $\text{right}(i)$ denote the rightmost element in the subarray of a_i, and let $\text{left}(j)$ denote the leftmost element in the subarray of a_j. $\text{MIN}[i, j]$ is the minimum among three numbers:

1. $\text{MIN}[i, \text{right}(i)]$;
2. $\text{MIN}[\text{left}(j), j]$;
3. $\text{MIN}[\text{right}(i) + 1, \text{left}(j) - 1]$.

The retrieval of the first and second numbers is similar to possibility (i) above. De-

note $i_1 = \lceil i/I_m^3(n) \rceil + 1$, and denote $j_1 = \lceil j/I_m^3(n) \rceil - 1$. Retrieval of the third number is equivalent to finding the minimum over interval $[b_{i_1}, \cdots, b_{j_1}]$ in B, which is denoted $\text{MIN}_B[i_1, j_1]$.

Let x be the lowest common ancestor of b_{i_1} and b_{j_1} in $TRL-BT(m)$, let $x_{\beta(i_1)}$ be the child of x that is an ancestor of b_{i_1}, and let $x_{\beta(j_1)}$ be the child of x that is an ancestor of b_{j_1}. $\text{MIN}_B[i_1, j_1]$ is (recursively) the minimum among three numbers:

1. $\text{MIN}_B[i_1, r(x_{\beta(i_1)})]$, the minimum over the suffix of b_{i_1} in $x_{\beta(i_1)}$ (we obtain this from $S_{x_{\beta(i_1)}}$);
2. $\text{MIN}_B[l(x_{\beta(j_1)}), j_1]$, the minimum over the prefix of b_{j_1} in $x_{\beta(j_1)}$ (we obtain this from $P_{x_{\beta(j_1)}}$);
3. $\text{MIN}_B[r(x_{\beta(i_1)}) + 1, l(x_{\beta(j_1)}) - 1]$ (this will be recursively derived from the data at node x).

The first two numbers are precomputed in TRL–$BT(m)$. The recursive definition of the third number implies that $\text{MIN}_B[i_1, j_1]$ is actually the minimum among $4(m-1)-2$ precomputed numbers and can thus be done in $\overline{c_1}m$ time for some constant $\overline{c_1}$. Below we show how to find the nodes x, $x_{\beta(i_1)}$, and $x_{\beta(j_1)}$ in constant time with one processor. This (together with the recursive definition of $BT(m)$) implies that over the retrieval procedure finding these nodes takes $\overline{c_2}m$ time for some constant $\overline{c_2}$ when one processor is used. We choose $c > \overline{c_1} + \overline{c_2}$, and the retrieval time is then cm, as required.

We first note that for each leaf of $TRL-BT(m)$, finding the child of the root that is its ancestor needs constant time if one processor is used: Recall that the number of leaves of $TRL-BT(m)$ is $n/I_m^3(n)$, its root has $(n/I_m^3(n))/I_{m-1}(n/I_m^3(n))$ children, and each of these children has $I_{m-1}(n/I_m^3(n))$ leaves. Thus the ancestor of a leaf b_i, $1 \le i \le n/I_m^3(n)$, of $TRL-BT(m)$ is the $\lceil i/I_{m-1}(n/I_m^3(n)) \rceil$th child of the root. Given two leaves of $TRL-BT(m)$, consider their ancestors among the children of the root. If these ancestors are different, we are done. Suppose these ancestors are the same.

Observe that for TRL–$BT(m)$ the same subtree structure is replicated at each child of the root. As part of the preprocessing algorithm we build one table with respect to this generic subtree structure. For each pair of leaves u and v of the generic subtree structure, we compute three items into a table: (1) their lowest common ancestor w; (2) the child f of w that is an ancestor of u; (3) the child g of w that is an ancestor of v. Since each child of the root has $I_{m-1}(n/I_m^3(n)) \le I_{m-1}(n)$ leaves, the size of the table is only $O(I_{m-1}^2(n))$.

How the table is computed remains to be shown. Consider an internal node w of the generic subtree structure, and suppose that its rooted subtree has r leaves. At node w each pair of leaves u, v is allocated to a processor. The processor determines in constant time if w is the LCA of u and v. This is done by finding whether the child of w that is an ancestor of u, denoted f, and the child of w that is an ancestor of v are different. If they are different, then w, f, and g are as required for the table. The number of internal nodes of the generic subtree is $O(I_{m-1}(n))$, and each has at most $I_{m-1}(n)$ leaves. Thus the number of processors needed for computing the table is $O(I_{m-1}^3(n))$.

The preprocessing algorithm for m. Inductively, we assume that we have an algorithm that preprocesses the array $A = (a_1, a_2, \cdots, a_n)$ for the range-minima problem in $c(m-1)$ time, using $nI_{m-1}(n) + \sqrt{k}n$ processors and $o(nI_{m-1}(n))$ space, where c is a constant, and that after this preprocessing any $\text{MIN}[i, j]$ query can be answered in $c(m-1)$ time. We construct an algorithm that solves the range-minima problem in $c_1 + c(m-1)$ time for some constant c_1, using $nI_m(n) + \sqrt{k}n$ processors and $o(nI_m(n))$ space. We have already shown that a query can be answered in $c_2 m$ time with one processor for some constant c_2. Selecting initially $c > c_1$ and $c > c_2$ implies that the algorithm runs in cm

time, using $nI_m(n) + \sqrt{kn}$ processors and $o(nI_m(n))$ space, and that a query can be answered in cm time.

Step 1. Partition A into subarrays of $I_m^3(n)$ elements each. Allocate $I_m^4(n)$ processors to each subarray, and apply the preprocessing algorithm for range minima given in §3 (for $\epsilon = \frac{1}{3}$). This uses $I_m^4(n)$ processors and $O(I_m^3(n)I_m^{1/3}(n))$ space per subarray and $nI_m(n)$ processors and $o(nI_m(n))$ space overall.

Step 2. Take the minimum in each subarray to build array B of size $n/I_m^3(n)$. The difference between two successive elements in B is at most $kI_m^3(n)$.

Step 3. Build $TRL-BT(m)$, the upper level of a $BT(m)$ tree whose leaves are the elements of B. The definition of $TRL-BT(m)$ implies that each internal node of $TRL-BT(m)$, whose rooted tree has r leaves, has $r/I_{m-1}(r)$ children. For each such internal node v of TRL–$BT(m)$ we keep an array. The array consists of the values of the r leaves of the subtree rooted at v. TRL–$BT(m)$ has $*I_{m-1}(n/I_m^3(n)) + 1 \leq I_m(n) + 1$ 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 $O(n/I_m^3(n))$ per level and $O(n/I_m^2(n))$ for all levels of $TRL-BT(m)$. We allocate to each leaf at each internal level $1 + \sqrt{k}I_m^2(n)$ processors, and the total number of processors used is thus $n/I_m^2(n) + \sqrt{kn}$ (which is less than $nI_m(n) + \sqrt{kn}$).

Step 4.1. For each internal node of $TRL-BT(m)$ find the minimum over its array. The difference between the minimum value and the maximum value in an array never exceeds the square of its number of processors, and we apply the constant-time algorithm mentioned in Remark 3.1 as in Step 4 of the recursion-base algorithm.

Step 4.2. We focus on internal node v having r leaves in $TRL-BT(m)$. Each of its $r/I_{m-1}(r)$ children contributes its minimum, and we preprocess these minima by using the assumed algorithm for $m-1$. The difference between adjacent elements is at most $kI_{m-1}(r)I_m^3(n)$. Thus this computation takes $c(m-1)$ time using $r + \sqrt{kI_m^3(n)}\, r$ processors. (To see this, simplify $(r/I_{m-1}(r))I_{m-1}(r) + \sqrt{kI_{m-1}(r)I_m^3(n)} \cdot (r/I_{m-1}(r))$, the processor count term for this problem, into $r + \sqrt{kI_m^3(n)}\, r/\sqrt{I_{m-1}(r)}$, which is less than $r + \sqrt{kI_m^3(n)}\, r$ processors.) This amounts to $n/I_m^3(n) + \sqrt{kI_m^3(n)}\, n/I_m^3(n)$ per level or a total of $n/I_m^2(n) + \sqrt{kn}/\sqrt{I_m(n)}$ processors, which is less than $n/I_m^2(n) + \sqrt{kn}$.

Step 5. For each internal node v we compute P_v (S_v is computed similarly). That is, for each leaf b_i of v we need to find $\mathrm{MIN}_B[l(v), i]$, the minimum over the prefix of b_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, $\mathrm{level}(v) < l \leq I_m(n) + 1$, of the tree contributes (at most) one number. Let u denote the ancestor at level $l-1$ of b_i, and let $u_1, \cdots, u_y$ denote its children, which are at level l. Suppose $u_j, j > 1$ is an ancestor of b_i. We take the prefix minimum over the leaves of $u_1, \cdots, u_{j-1}$. This prefix minimum is computed in the previous step (by the assumed algorithm for $m-1$). If u_1 is the ancestor of b_i, then level l contributes a large default value (as in Step 5 of the recursion-base algorithm). Finally, b_i is also added to the list. This minimum computation can be done in constant time by using $I_m^2(n)$ processors (by the algorithm of [SV81]). Note that (1) all prefix minima and all suffix minima with respect to B are computed (in the root) in this step, and (2) given a leaf b_i in $TRL-BT(m)$, finding a node w that is both a child of the root and an ancestor of b_i in constant time was solved in the context of the query retrieval. Node u is found similarly.

Step 6. For each a_i we find its prefix minimum and its suffix minimum with respect to A by using one processor in constant time. This is similar to Step 6 of the recursion base.

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, 4.2, and 6 above.

Complexity of the recursive step. In addition to application of the inductively assumed algorithm, Steps 1 through 6 take constant time and use $nI_m(n) + \sqrt{k}n$ processors and $o(nI_m(n))$ space. This totals cm time if $nI_m(n) + \sqrt{k}n$ processors and $o(nI_m(n))$ space are used. Together with Lemma 4.2.2, Lemma 4.2.1 follows.

From recursion to algorithm. The recursive procedure in Lemma 4.2.1 translates easily into a constructive parallel algorithm in which 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) $I^{(i)}_{m-1}(x)$ for $1 \leq x \leq n$ and $1 \leq i \leq I_m(x)$. Let us illustrate how processor allocation is done in Step 3 above. We use $n/I^2_m(n) + \sqrt{k}n$ processors. The processors are partitioned into $I_m(n)$ groups of $n/I^3_m(n) + \sqrt{k}n/I_m(n)$ 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 are computed and stored in a table at the beginning of the algorithm. Section 6 discusses their computation.

4.3. The optimal parallel algorithms. In this subsection we show how to derive a series of optimal parallel algorithms from the series of algorithms described in Lemma 4.2.1. Theorem 4.3.1 gives a general tradeoff result between the running time of the preprocessing algorithm and the retrieval time. Corollary 4.3.1 emphasizes results in cases for which the retrieval time for a query is constant. Corollary 4.3.2 points at an interesting tradeoff instance in which the retrieval time bound is increased to $O(\alpha(n))$ and the preprocessing algorithm runs in $O(\alpha(n))$ (i.e., it become almost fully parallel).

THEOREM 4.3.1. *Consider the range-minima problem where k, the bound on the difference between two successive elements in A, is constant. For each $2 \leq m \leq \alpha(n)$ we present a parallel preprocessing algorithm that runs in time $O(I_m(n))$ and uses an optimal number of processors and optimal linear space. Query retrieval time is $O(m)$ if one processor is used.*

COROLLARY 4.3.1. *When m is constant the preprocessing algorithm runs in $O(I_m(n))$ time if $n/I_m(n)$ processors are used. Retrieval time is $O(1)$ if one processor is used.*

COROLLARY 4.3.2. *When $m = \alpha(n)$ the preprocessing algorithm runs in $O(\alpha(n))$ time if $n/\alpha(n)$ processors are used. Retrieval time is $O(\alpha(n))$ if one processor is used.*

We describe below the optimal preprocessing algorithm for m as per Theorem 4.3.1.

Step 1. Partition A into subarrays of $I^2_m(n)$ elements each, allocate $I_m(n)$ processors to each subarray, and find the minimum in the subarray. This can be done in $O(I_m(n))$ time. Put the $n/I^2_m(n)$ minima into an array B.

Step 2. Out of the series of preprocessing algorithms of Lemma 4.2.1, apply the algorithm for m to B, where k', the difference between two successive elements of B, is $O(I^2_m(n))$. This takes $O(m)$ time if $\sqrt{k'}\, n/I^2_m(n) + n/I_m(n)$ processors and $o(n/I_m(n))$ space are used. This can be simulated in $O(m)$ time by using $n/I_m(n)$ processors and $o(n/I_m(n))$ space.

Step 3. Preprocess each subarray of $I^2_m(n)$ elements so that a range-minimum query within the subarray can be retrieved in $O(1)$ time. This is done by using the following parallel variant of the range-minima algorithm of Gabow, Bentley, and Tarjan (GBT) [GBT84].

Range minimum: a parallel variant of the GBT *algorithm.* Consider the general range-minima problem as defined in §3, with respect to an input array $C = (c_1, c_2, \cdots, c_n)$. We overview a preprocessing algorithm that runs in $O(\sqrt{n})$ time using $\sqrt{n}$ processors, so that a range-minimum query can be processed in constant time.

(1) Partition array C into $\sqrt{n}$ subarrays $C_1, \cdots, C_{\sqrt{n}}$, each with $\sqrt{n}$ elements.

(2) Apply the GBT linear time serial algorithm separately to each C_i, taking $O(\sqrt{n})$ time and using $\sqrt{n}$ processors.

(3) Let $\overline{c_i}$ be the minimum over C_i. Apply the GBT algorithm to $\overline{C} = (\overline{c_1}, \cdots, \overline{c_{\sqrt{n}}})$ in $O(\sqrt{n})$ time, using a single processor.

It should be clear that any range-minimum query with respect to C can be retrieved in constant time by at most three queries with respect to the tables built by the above applications of the GBT algorithm. The complexity of the preprocessing algorithm is specified by $O(I_m(n))$ time if $n/I_m(n)$ processors are used. Retrieval of a range-minimum query takes $O(m+1)$ time, which is $O(m)$ time if one processor is used. Theorem 4.3.1 follows.

4.4. The fully parallel algorithms. Consider the restricted-domain range-minima problem in which k, the bound on the difference between adjacent elements, is constant. In this subsection we present a fully parallel preprocessing algorithm for the problem on a CRCW-bit PRAM that provides for constant time processing of a query. Theorem 4.4.1 gives the general result achieved in this subsection, including tradeoff among parameters. Corollary 4.4.1 summarizes the fully parallel result.

Let d be an integer $2 < d \leq \alpha(n)$. The model of parallel computation is the CRCW-bit PRAM with the assumption that up to $I_d(n)$ processors may write simultaneously into different bits of the same memory word.

THEOREM 4.4.1. *The preprocessing algorithm takes $O(d)$ time if n processors and linear space are used. Retrieval of a query* MIN(i, j) *takes $O(d)$ time if one processor is used.*

Remark. Theorem 4.4.1 represents a tradeoff between the time for the preprocessing algorithm and query retrieval, on one hand, and the number of processors that may write simultaneously into different bits of the same memory word, on the other.

COROLLARY 4.4.1. *For a constant d the algorithm is fully parallel and query retrieval can be done in constant time with one processor.*

Step 1. Partition A into $n/I_d(n)$ subarrays of $I_d(n)$ elements each. For each subarray find the minimum in $O(1)$ time and $I_d(n)$ processors. For this we apply the constant-time algorithm mentioned in Remark 3.1 as in Step 4 of the recursion-base algorithm. Put the $n/I_d(n)$ minima into an array B. The difference between two successive elements in B is at most $kI_d(n)$.

Step 2. Out of the series of preprocessing algorithms of Lemma 4.2.1 apply the algorithm for d to B, where k', the difference between two successive elements of B, is $O(I_d(n))$. This takes $O(d)$ time when $\sqrt{k'}\, n/I_d(n) + n$ processors and $o(n)$ space are used and can be simulated in $O(d)$ time by using n processors and $o(n)$ space.

Suppose we know to retrieve a range-minimum query within each of the subarrays of size $I_d(n)$ in constant time. It should be clear how a query MIN(i, j) can then be retrieved in $O(d)$ time. Theorem 4.4.1 would follow. Thus it remains to show how to preprocess the subarrays of size $I_d(n)$ in constant time such that a range-minimum query within a subarray can be retrieved in constant time. These subarrays are preprocessed in Steps 3.1, 3.2, and 3.3.

Step 3.1. For each subarray subtract the value of its first element from each element of the subarray. Observe that after this subtraction the value of the first element is zero and the difference between each pair of successive elements remains at most k. Step 3.2

constructs a table with the following information: For any $I_d(n)$-tuple $(c_1, \cdots, c_{I_d(n)})$, where $c_1 = 0$ and the difference between each pair of successive c_i values is at most k, the table has an entry. This entry gives all $I_d(n)(I_d(n)-1)/2$ range minima with respect to this $I_d(n)$-tuple.

Step 3.2. All n processors together build a table. Each entry of the table corresponds to one possible allocation of values to the $I_d(n)$-tuple. The entry provides all $I_d(n)(I_d(n)-1)/2$ range minima for this allocation. Observe that the number of possible allocations is $(2k(I_d(n)-1)+1)^{I_d(n)-1}$. To see this we note that each element in an $I_d(n)$-tuple assumes an integer value in the range $[-k(I_d(n)-1)\cdots k(I_d(n)-1)]$. Our table will have $y \geq (2k(I_d(n)-1)+1)^{I_d(n)-1}$ entries. It will be more convenient to defer the statement of the exact value for y to the comment below. By using n processors (or even fewer) the table can be built in $O(1)$ time. We sketch below how the table is built. Consider any integer x in the range $[1, y]$, and suppose x is given in binary representation. We break the binary (i.e., bit) representation of x into $I_d(n)-1$ bit strings of equal length. Each of these bit strings should be long enough to represent integers in the range $[-k(I_d(n)-1), k(I_d(n)-1)]$ (namely, a range of $2k(I_d(n)-1)+1$ integers). (It is enough to have $\lceil \log(2k(I_d(n)-1)+1) \rceil$ bits in each bit string and therefore a total of $(I_d(n)-1)(\lceil \log(2k(I_d(n)-1)+1) \rceil)$ bits for representing the original value of x. So $y = 2^{(I_d(n)-1)(\lceil \log(2k(I_d(n)-1)+1) \rceil)}$.) These integers are then put in an array X (of size $I_d(n)-1$). It is easy to see that for any $I_d(n)$-tuple (whose first entry is fixed to zero) for which the difference between any two consecutive locations is at most k, there exists an integer x in the range $[1, y]$ whose array X is equal to locations 2 to $I_d(n)$ in the $I_d(n)$-tuple. For each entry x, $1 \leq x \leq y$, in our table we compute separately each range minimum with respect to array X of x. Given a range $[i, j]$ the minimum over $[X_i \cdots X_j]$ is done in constant time with $(j-i)^2$ processors by the algorithm of [SV81]. Since $I_d(n) \leq \log^* n$, the n available processors are more than enough for the necessary computations above.

Step 3.3. The only difficulty is to identify the table entry for our $I_d(n)$-tuple, $c_1, \cdots, c_{I_d(n)}$, since once we reach the entry the table already provides the desired range minimum. We allocate to each subarray $I_d(n)$ processors. For each subarray we have a word in our shared memory with $(I_d(n)-1)\lceil \log(2k(I_d(n)-1)+1) \rceil$ bits. Processor $i, 1 < i \leq I_d(n)$, writes c_i (which is an integer from $[-k(I_d(n)-1)\cdots k(I_d(n)-1)]$) starting in bit number $(i-2)\lceil \log(2k(I_d(n)-1)+1) \rceil$ of the word belonging to its subarray (bit zero is the least significant). As a result, this word has a sequence of integers from the range $[-k(I_d(n)-1), k(I_d(n)-1)]$ that yields the desired entry in our table. Note that exactly $I_d(n)$ processors write to different bits of the same memory word. Theorem 4.4.1 follows.

4.5. The all-nearest-zero-bit problem. The following corollary of Theorems 4.3.1 and 4.4.1 is needed for §5.

COROLLARY 4.5.1. *The all-nearest-zero-bit problem is almost fully parallel. On the* CRCW-*bit* PRAM *the all-nearest-zero-bit problem is fully parallel.*

Proof. Recall that the algorithm for the restricted-domain range-minima problem computes all suffix minima. Recall also that if the minimum over an interval is not unique, the leftmost minimum is found. Thus if we apply the restricted-domain range-minima algorithm (with the difference between successive elements at most one) with respect to A, then the minimum over the suffix of entry $i+1$ gives the nearest zero to the right of entry i. Thus the all-nearest-zero bit is actually an instance of the restricted-domain range-minima problem (with the difference between successive elements at most

one). It follows that the almost fully parallel and the fully parallel algorithms for the latter apply for the all-nearest-zero-bit problem as well. □

4.6. A simple sequential LCA algorithm. In this subsection we outline a sequential variant of the restricted-domain range-minima problem for which k, the difference between adjacent elements, is one. Together with the reduction of §4.1, this gives a sequential algorithm for the LCA problem.

We first describe two preprocessing procedures for the range-minima problem: (i) Procedure I takes $O(n \log n)$ time for an input array of length n. No assumptions are needed regarding the difference between adjacent elements. (A similar procedure is used in [AS87].) (ii) Procedure II takes exponential time. After each of these preprocessing procedures, query retrieval takes constant time. Second, the sequential linear-time range-minima algorithm is described. Finally, we show how to retrieve a range-minimum query in constant time.

Procedure I. Build a complete binary tree whose leaves are the elements of the input array A. Compute (and keep) for each internal node all prefix minima and all suffix minima with respect to its leaves.

Procedure I clearly runs in $O(n \log n)$ time. Given any range $[i, j]$, the range-minimum query with respect to $[i, j]$ can be processed in constant time, as follows. (1) Find the lowest node u of the binary tree such that the range $[i, j]$ falls within its leaves. This range is the union of a suffix of the left child of u and a prefix of the right child of u. The minima over the suffix and prefix were computed by Procedure I. (2) The answer to the query is the minimum among these two minima.

Procedure II. We use the assumption that the difference between any two adjacent elements of the input array A is exactly one. A table similar to the table built in Step 3 of §4.4 is built as follows. We assume without loss of generality that the value of the first element of A is zero (since otherwise we can subtract from every element in A the value of the first element without affecting the answers to range-minima queries). Then the number of different possible input arrays A is 2^{n-1}. The table will have a subtable for each of these 2^{n-1} possible arrays. For each possible array the subtable will store the answer to each of the $n(n-1)/2$ possible range queries. The time to build the table is $O(2^n n^2)$, and $O(2^n n^2)$ space is needed.

The linear-time range-minima preprocessing algorithm follows.

• For each of the subsets $a_{i \log n+1}, \cdots, a_{(i+1) \log n}$ for $0 \leq i \leq n/\log n - 1$, find its minimum and apply Procedure I to an array of these $n/\log n$ minima.

• Separately, for each of the subsets $a_{i \log n+1}, \cdots, a_{(i+1) \log n}$ for $0 \leq i \leq n/\log n - 1$, do the following. Partition each such subset into smaller subsets of size $\log \log n$, and find the minimum in each smaller subset; apply Procedure I to these $\log n/\log \log n$ minima.

• Run Procedure II to build the table required for an (any) array of size $\log \log n$. For each of the subsets $a_{i \log \log n+1}, \cdots, a_{(i+1) \log \log n}$ for $0 \leq i \leq n/\log \log n - 1$, identify its subtable.

The time (and space) for each step of the preprocessing algorithm is $O(n)$.

Consider a query requesting the minimum over a range $[i, j]$. We show how to process it in constant time. The range $[i, j]$ can easily be presented as the union of the following (at most) five ranges: $[i, x_1]$, $[x_1 + 1, y_1]$, $[y_1 + 1, y_2]$, $[y_2 + 1, x_2]$, and $[x_2 + 1, j]$, where (1) $[i, x_1]$ (and $[x_2 + 1, j]$) falls within a single subset of size $\log \log n$—its minimum is available in its subtable, (2) $[x_1 + 1, y_1]$ (and $[y_2 + 1, x_2]$) is the union of subsets of size $\log \log n$ and falls within a single subset of size $\log n$—its minimum is available from the application of Procedure I to the subset of size $\log n$, and (3) $[y_1 + 1, y_2]$ is the union of

subsets of size $\log n$—its minimum is available from the first application of Procedure I. So the minimum over range $[i, j]$ is simply the minimum of these five minima.

5. Almost fully parallel reducibility. We demonstrate how to use the $*$-tree data structure for reducing a problem A to another problem B by an almost fully parallel algorithm. We apply this reduction for deriving a parallel lower bound for problem A from a known parallel lower bound for problem B.

Given a convex polygon with n vertices, the *all-nearest-neighbors* (*ANN*) problem is to find for each vertex of the polygon its nearest (Euclidean) neighbor.

THEOREM 5.1. *Any* CRCW PRAM *algorithm for the* ANN *problem that uses* $O(n \log^c n)$ *(for any constant* c*) processors needs* $\Omega(\log \log n)$ time.

Proof. We give below an almost fully parallel reduction from the problem of merging two sorted lists of length n each to the ANN problem with $O(n)$ vertices. This reduction together with the following lemma imply the theorem.

LEMMA 5.1. *Merging two sorted lists of length* n *each by using* $O(n \log^c n)$ *(for any constant* c*) processors on a* CRCW PRAM *requires* $\Omega(\log \log n)$ time.

A remark in [SV90] implies that Borodin and Hopcroft's [BH85] lower bound for merging in a parallel comparisons model can be extended to yield the lemma.

Proof of Theorem 5.1 *(continued).*

The reduction (see Fig. 3). Let $A = (a_1, a_2, \cdots, a_n)$ and $B = (b_1, b_2, \cdots, b_n)$ be two increasing lists of numbers that we wish to merge. Assume, without loss of generality, that the numbers are integers and that $a_1 = b_1$, $a_n = b_n$. (The lower bound for merging assumes that the numbers are integers.) Consider the following *auxiliary* problem: For each $1 \leq i \leq n$ find the minimum index j such that $b_j > a_i$. The position of a_i in the merged list is $i + j - 1$, and therefore an algorithm for the auxiliary problem (together with a similar algorithm for finding the positions of the b_i numbers in the merged list) suffices for the merging problem.

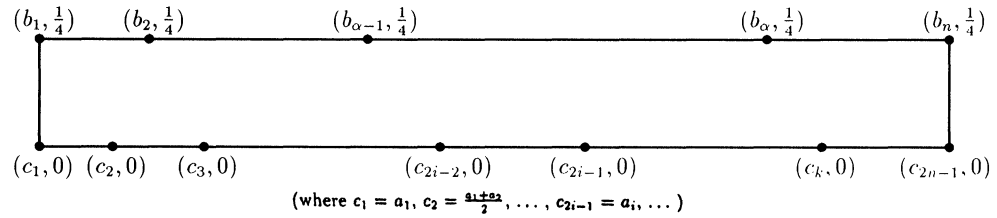

FIG. 3. *Almost fully parallel reducibility construction.*

We give an almost fully parallel reduction from the auxiliary problem to the ANN problem with respect to the following convex polygon. Let $(c_1, c_2, \cdots, c_{2n-1}) = (a_1, (a_1 + a_2)/2, a_2, (a_2 + a_3)/2, \cdots, a_{n-1}, (a_{n-1} + a_n)/2, a_n)$. The numbers $c_1, c_2, \cdots, c_{2n-1}$ form an increasing list. The convex polygon is $(c_1, 0), (c_2, 0), \cdots, (c_{2n-1}, 0), (b_n, \frac{1}{4}), (b_{n-1}, \frac{1}{4}), \cdots, (b_1, \frac{1}{4})$.

In [SV90] a similar construction is given, and the lower-bound proof then follows by (nontrivial) Ramsey theoretic arguments.

Let $D[1, \cdots, 2n-1]$ be a binary vector. Each vertex $(c_l, 0)$ finds its nearest neighbor with respect to the convex polygon (by using a "supposedly existing" algorithm for the ANN problem) and assigns the following into vector D.

If the nearest vertex is of the form $(b_k, \frac{1}{4})$

Then $D(l) := 0$

Else $D(l) := 1$

Next we apply to vector D the almost fully parallel algorithm for the nearest-zero-bit problem of §4.5. Finally, we show how to solve our auxiliary problem with respect to every element a_i. We break into two cases concerning the nearest neighbor of $(a_i, 0)$ $(= (c_{2i-1}, 0))$.

Case (i). The nearest neighbor of $(a_i, 0)$ is a vertex $(b_\alpha, \frac{1}{4})$. Then the minimum index j such that $b_j > a_i$ is either α or $\alpha + 1$. A single processor can determine the correct value of j in $O(1)$ time.

Case (ii). Otherwise, then $D(2i-1) = 1$. The nearest-zero computation gives the smallest index $k > 2i - 1$ for which $D(k) = 0$. Let the nearest neighbor of $(c_k, 0)$ be $(b_\alpha, \frac{1}{4})$. Then $j = \alpha$ is the minimum index for which $b_j > a_i$. □

6. Computing various functions. We need to compute certain functions in our algorithms. For each $2 < m \leq \alpha(n)$ we need $I^{(i)}_{m-1}(n)$ for all $1 \leq i \leq I_m(n)$. Fortunately, the function parameters that we are actually concerned with are small relative to n. For instance, to compute $I_3(n) = \log^* n$, it is enough to compute $\log^* (\log \log n)$ since $\log^* n = \log^* (\log \log n) + 2$.

We show only how to compute $I_m(n)$ for each $2 \leq m \leq \alpha(n)$. Computation of $I^{(i)}_{m-1}(n)$ for $2 < m \leq \alpha(n)$ and all $1 \leq i \leq I_m(n)$ is similar. Our computation works by induction on m.

The inductive hypothesis. Let $x = \log \log n$. We know how to compute the following values in $O(m-1)$ time by using $o(n/\alpha(n))$ processors: (1) $I_{m-1}(n)$ and (2) $I_{m-1}(y)$ for all $1 \leq y \leq \log \log n$.

We show the inductive claim (the claim itself should be clear) by assuming the inductive hypothesis. The inductive base is given later. First, we describe (informally) the computation of $I_m(x)$ in $O(1)$ (additional) time using $o(n/\alpha(n))$ processors. Consider all permutations of the numbers $1, \cdots, x$. The number of these permutations is (much) less than n. The idea is to identify a permutation that provides the sequence $[x, I_{m-1}(x), I^{(2)}_{m-1}(x), \cdots, I^{(k)}_{m-1}(x) = 1, \cdots]$. So if $I_{m-1}(p_i) = p_{i+1}$ for all $0 \leq i \leq k-1$, we conclude that $I_m(x) = k$. We can check this condition in $O(1)$ time with x processors per permutation by using the ability of the CRCW PRAM to find the AND of x bits in $O(1)$ time. The total number of processors is $o(n/\alpha(n))$. We make two remarks: (1) Computing $I_m(y)$ for all $1 \leq y \leq \log \log n$, the rest of the inductive claim, in $O(1)$ time with $o(n/\alpha(n))$ processors is similar; (2) there are easy ways for finding all permutations in $O(1)$ time by using the number of available processors.

We finish by showing the inductive base. We compute $\log n$ in $O(1)$ time and with $o(n/\alpha(n))$ processors as follows. If n is given in a binary representation, then the index of the leftmost one is $\log n$. Following [FRW84], this can be computed in $O(1)$ time by using as many processors as the number of bits of a number. By iterating this we obtain $\log^{(2)} n$. Finally, we find $\log y$ for all $1 \leq y \leq \log \log n$. The number of processors used for this computation is $o(n/\alpha(n))$.

Acknowledgments. We are grateful to Pilar de la Torre and to Baruch Schieber for fruitful discussions and helpful comments.

REFERENCES

[AIL+88] A. APOSTOLICO, C. ILIOPOULOS, G. M. LANDAU, B. SCHIEBER, AND U. VISHKIN, *Parallel construction of a suffix tree with applications*, Algorithmica, 3 (1988), pp. 347–365.

[AM88] R. J. ANDERSON AND G. L. MILLER, *Deterministic parallel list ranking*, in Lecture Notes in Computer Science 319, Springer-Verlag, Berlin, New York, 1988, pp. 81–90.

[AMW89] R. J. ANDERSON, E. W. MAYR, AND M. K. WARMUTH, *Parallel approximation algorithms for bin packing*, Inform. and Comput., 82 (1989), pp. 262–277.

[AS87] N. ALON AND B. SCHIEBER, *Optimal preprocessing for answering on-line product queries*, Tech. Report TR 71/87, Moise and Frida Eskenasy Institute of Computer Science, Tel Aviv University, Tel Aviv, Israel, 1987.

[BBG+89] O. BERKMAN, D. BRESLAUER, Z. GALIL, B. SCHIEBER, AND U. VISHKIN, *Highly-parallelizable problems*, in Proc. 21st Annual ACM Symposium on Theory of Computing, 1989, pp. 309–319.

[BH85] A. BORODIN AND J. E. HOPCROFT, *Routing, merging, and sorting on parallel models of computation*, J. Comput. System Sci., 30 (1985), pp. 130–145.

[BH87] P. BEAME AND J. HASTAD, *Optimal bounds for decision problems on the* CRCW PRAM, in Proc. 19th Annual ACM Symposium on Theory of Computing, 1987, pp. 83–93.

[BSV88] O. BERKMAN, B. SCHIEBER, AND U. VISHKIN, *Some doubly logarithmic optimal parallel algorithms based on finding nearest smaller values*, Tech. Report UMIACS-TR-88-79, University of Maryland Institute for Advanced Computer Studies, College Park, MD, 1988; also IBM Res. Report, Computer Sciences, RC 14128 (#63291); J. Algorithms, to appear.

[BV85] I. BAR-ON AND U. VISHKIN, *Optimal parallel generation of a computation tree form*, ACM Trans. Prog. Lang. and Systems, 7 (1985), pp. 348–357.

[BV90] O. BERKMAN AND U. VISHKIN, *On parallel integer merging*, Tech. Report UMIACS-TR-90-15.1 (revised version), University of Maryland Institute for Advanced Computer Studies, College Park, MD, 1990; Inform. and Comput., to appear.

[BV91] O. BERKMAN AND U. VISHKIN, *Almost fully parallel parentheses matching*, Tech. Report UMIACS-TR-91-103, University of Maryland Institute for Advanced Computer Studies, College Park, MD, 1991.

[CFL83] A. K. CHANDRA, S. FORTUNE, AND R. J. LIPTON, *Unbounded fan-in circuits and associative functions*, in Proc. 15th Annual ACM Symposium on Theory of Computing, 1983, pp. 52–60.

[CV86a] R. COLE AND U. VISHKIN, *Approximate and exact parallel scheduling with applications to list, tree and graph problems*, in Proc. 27th IEEE Annual Symposium on Foundations of Computer Science, 1986, pp. 478–491.

[CV86b] ———, *Deterministic coin tossing with applications to optimal parallel list ranking*, Inform. and Control, 70 (1986), pp. 32–53.

[CV89] ———, *Faster optimal parallel prefix sums and list ranking*, Inform. and Comput., 81 (1989), pp. 334–352.

[DS83] E. DEKEL AND S. SAHNI, *Parallel generation of postfix and tree forms*, ACM Trans. on Prog. Lang. and Systems, 5 (1983), pp. 300–317.

[FRT89] D. FUSSELL, V. RAMACHANDRAN, AND R. THURIMELLA, *Finding triconnected components by local replacements*, Lecture Notes in Computer Science 372, Springer-Verlag, Berlin, New York, 1989.

[FRW84] F. E. FICH, P. L. RAGDE, AND A. WIGDERSON, *Relations between concurrent-write models of parallel computation* (preliminary version), in Proc. 3rd ACM Symposium on Principles of Distributed Computing, 1984, pp. 179–189; also SIAM J. Comput., 17 (1988), pp. 606–627.

[GBT84] H. N. GABOW, J. L. BENTLEY, AND R. E. TARJAN, *Scaling and related techniques for geometry problems*, in Proc. 16th Annual ACM Symposium on Theory of Computing, 1984, pp. 135–143.

[Has86] J. HASTAD, *Almost optimal lower bounds for small depth circuits*, in Proc. 18th Annual ACM Symposium on Theory of Computing, 1986, pp. 6–20.

[HS86] S. HART AND M. SHARIR, *Nonlinearity of Davenport–Schinzel sequences and generalized path compression schemes*, Combinatorica, 6 (1986), pp. 151–177.

[HT84] D. HAREL AND R. E. TARJAN, *Fast algorithms for finding nearest common ancestors*, SIAM J. Comput., 13 (1984), pp. 338–355.

[Kru83] C. P. KRUSKAL, *Searching, merging, and sorting in parallel computation*, IEEE Trans. Comput., C-32 (1983), pp. 942–946.

[LF80] R. E. LADNER AND M. J. FISCHER, *Parallel prefix computation*, J. Assoc. Comput. Mach., 27 (1980) pp. 831–838.

[LV89] G. M. LANDAU AND U. VISHKIN, *Efficient parallel and serial approximate string matching*, J. Algorithms, 10 (1989), pp. 157–169.

[MSV86] Y. MAON, B. SCHIEBER, AND U. VISHKIN, *Parallel ear decomposition search (EDS) and st-numbering in graphs*, Theoret. Comput. Sci., 47 (1986), pp. 277–298.

[RR89] V. RAMACHANDRAN AND J. H. REIF, *An optimal parallel algorithm for graph planarity*, in Proc. 30th IEEE Annual Symposium on Foundations of Computer Science, 1989, pp. 282–287.

[Sto88] Q. F. STOUT, *Constant-time geometry on PRAMs*, in Proc. International Conference on Parallel Processing, 1988, pp. 104–107.

[SV81] Y. SHILOACH AND U. VISHKIN, *Finding the maximum, merging, and sorting in a parallel computation model*, J. Algorithms, 2 (1981), pp. 88–102.

[SV84] L. STOCKMEYER AND U. VISHKIN, *Simulation of parallel random access machines by circuits*, SIAM J. Comput., 13 (1984), pp. 409–422.

[SV88] B. SCHIEBER AND U. VISHKIN, *On finding lowest common ancestors: Simplification and parallelization*, SIAM J. Comput., 17 (1988), pp. 1253–1262.

[SV90] ———, *Finding all-nearest neighbors for convex polygons in parallel: A new lower bound technique and a matching algorithm*, Discrete Appl. Math., 29 (1990), pp. 97–111.

[SZ89] D. SHASHA AND K. ZHANG, *New algorithms for the editing distance between trees*, in Proc. 1st ACM Symposium on Parallel Algorithms and Architectures, 1989, pp. 117–126, to appear in J. Algorithms as *Fast algorithms for the unit cost editing distance between trees*.

[Tar75] R. E. TARJAN, *Efficiency of a good but not linear set union algorithm*, J. Assoc. Comput. Mach., 22 (1975), pp. 215–225.

[TV85] R. E. TARJAN AND U. VISHKIN, *An efficient parallel biconnectivity algorithms*, SIAM J. Comput., 14 (1985), pp. 862–874.

[Val75] L. G. VALIANT, *Parallelism in comparison problems*, SIAM J. Comput., 4 (1975), pp. 348–355.

[Van89] A. VAN GELDER, *PRAM processor allocation: A hidden bottleneck in sublogarithmic algorithms*, IEEE Trans. Comput., 38 (1989), pp. 289–292.

[Vis85] U. VISHKIN, *On efficient parallel strong orientation*, Inform. Process. Lett., 20 (1985), pp. 235–240.

[Vis91] ———, *Deterministic sampling–a new technique for fast pattern matching*, SIAM J. Comput., 20 (1991), pp. 22–40.

[Yao82] A. C. YAO, *Space–time tradeoff for answering range queries*, in Proc. 14th ACM Symposium on Theory of Computing, 1982, pp. 128–136.