SIAM J. COMPUT.
Vol. 17, No. 6, December 1988

012

ON FINDING LOWEST COMMON ANCESTORS: SIMPLIFICATION AND PARALLELIZATION*

BARUCH SCHIEBER† AND UZI VISHKIN†‡

Abstract. We consider the following problem. Suppose a rooted tree T is available for preprocessing. Answer on-line queries requesting the lowest common ancestor for any pair of vertices in T. We present a linear time and space preprocessing algorithm that enables us to answer each query in $O(1)$ time, as in Harel and Tarjan [*SIAM J. Comput.*, 13 (1984), pp. 338–355]. Our algorithm has the advantage of being simple and easily parallelizable. The resulting parallel preprocessing algorithm runs in logarithmic time using an optimal number of processors on an EREW PRAM. Each query is then answered in $O(1)$ time using a single processor.

Key words. parallel algorithms, tree algorithms, lowest common ancestors

AMS(MOS) subject classifications. 05C05, 68Q10, 68Q20, 68R10

1. Introduction. We consider the following problem. Given a rooted tree $T(V, E)$ for preprocessing, answer on-line LCA queries of the form, "Which vertex is the Lowest Common Ancestor (LCA) of x and y?" for any pair of vertices x, y in T. (Let us denote such a query LCA (x, y).) We present a preprocessing algorithm that runs in linear time and linear space on the serial RAM model. (For the definition of a random access machine (RAM) model see, e.g., [1].) Given this preprocessing, we show how to process each such LCA query in constant time.

We also consider parallelization of our algorithm. The model of parallel computation used is the exclusive-read exclusive-write (EREW) parallel random access machine (PRAM). A PRAM employs p synchronous processors all having access to a common memory. An EREW PRAM does not allow simultaneous access by more than one processor to the same memory location for either read or write purposes. See [11] for a survey of results concerning PRAMs.

Let Seq (n) be the fastest known worst-case running time of a sequential algorithm, where n is the length of the input for the problem at hand. A parallel algorithm that runs in $O(\text{Seq}\,(n)/p)$ time using p processors is said to have *optimal speedup* or, more simply, to be *optimal.* A primary goal in parallel computation is to design optimal algorithms that also run as fast as possible.

Our preprocessing algorithm is easily parallelized to obtain an optimal parallel preprocessing algorithm that runs in $O(\log n)$ time using $n/\log n$ processors on an EREW PRAM, where n is the number of vertices in T. Parallelizing the query processing is straightforward, provided read conflicts are allowed: k queries can be processed in $O(1)$ time using k processors.

---

* Received by the editors March 3, 1987; accepted for publication (in revised form) February 22, 1988. This research was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under contract DE-AC02-76ER03077.

† Department of Computer Science, School of Mathematical Sciences, Tel Aviv University, Tel Aviv, Israel 69978. Present address, T. J. Watson Research Center, IBM Research Division, Yorktown Heights, New York 10598.

‡ Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, New York, New York 10012. The research of this author was supported by National Science Foundation grant NSF-CCR-8615337, Office of Naval Research grant N00014-85-K-0046 and the Foundation for Research in Electronics, Computers and Communication, administered by the Israeli Academy of Sciences and Humanities.

In their extensive paper [5], Harel and Tarjan gave a serial algorithm for the same problem. The performance of their algorithm is the same as ours. However, our algorithm has two advantages: (1) It is considerably simpler in both the preprocessing stage and the query processing and (2) It leads to a simple parallel algorithm. Below, we discuss similarities and differences with respect to [5]. *Similarities*: Both algorithms use two basic observations: (1) It is possible to answer LCA queries in simple paths in constant time and (2) It is possible to answer LCA queries in complete binary trees in constant time. Both algorithms pack information regarding several vertices into a single $O(\log n)$ bits number. *Differences*: The subtler part of both algorithms is to show how to use the above two observations for answering an LCA query. In this part, our approach is completely different. In the preprocessing stage we compute a mapping from the vertices of the input tree T to the vertices of a complete binary B. The mapping has two properties: (i) All the vertices of T mapped into the same vertex in B form a path and (ii) For each vertex v in T, the descendants of v are mapped into descendants of the image of v in B. This mapping, together with some additional information, enables us to answer an LCA query in constant time. In [5], on the other hand, the vertices of the input tree T are mapped to the vertices of an arbitrary tree of logarithmic height, called the compressed tree. The preprocessing consists of a quite involved manipulation of this compressed tree. This manipulation includes partitioning the compressed tree into three plies and preprocessing each ply separately and also embedding the compressed tree in a complete binary tree.

Consider a dynamic LCA problem which, interspersed with the LCA queries, are on-line deletions and insertions of edges. Reference [5] also gives algorithms for some cases of this problem. We do not consider this problem in the present paper.

Our parallel algorithm improves on the following results. Tsin [9] gave two parallel algorithms for the LCA problem. In his first algorithm both the preprocessing stage and the query processing take logarithmic time with a linear number of processors. In his second algorithm the preprocessing stage takes $O(\log n)$ time using n^2 processors and processing a query takes $O(1)$ time using a single processor. Vishkin [12] includes a parallel algorithm for the LCA problem. The processing of an LCA query takes logarithmic time (as in the first algorithm of Tsin). The preprocessing stage takes $O(\log n)$ time using $n/\log n$ processors (as in the present paper).

Observe that using our parallel preprocessing algorithm we can process k off-line LCA queries in $O(\log n)$ time using $(n+k)/\log n$ processors, provided read conflicts are allowed. This affects the performance of parallel algorithms for three problems: (1) Given an undirected graph, orient its edges so that the resulting digraph is strongly connected (if such orientation is possible) [12]. (2) Computing an open-ear decomposition and *st*-numbering of a biconnected graph [8]. Using the new parallel connectivity and list-ranking algorithms of [3], it has become possible to solve each of these problems in logarithmic time using an optimal number of processors only when $m \geqq n \log n$, where n is the number of vertices and m is the number of edges in the input graph. Our off-line LCA computation enables us to extend the range of optimal speedup logarithmic time parallel algorithms for these problems to sparser graphs, where $m \geqq n \log^* n$ as in the above connectivity algorithm. (3) Approximate string matching [6]. The new parallel suffix tree construction of [7] together with the present parallel LCA computation lead to a considerable simplification of the parallel algorithm of [6]. This simplification has already been described in [2].

The paper is organized as follows. Section 2 gives a high-level description of the algorithm. Section 3 describes the preprocessing stage. In § 4 we show how to process LCA queries in T using the outcome of the preprocessing stage. Section 5 presents parallelization of our preprocessing stage.

2. High-level description. The entire algorithm is based on the following two observations (made also in [5]): (1) Had our input tree been a simple path, it would have been possible to preprocess it (by way of computing the distance of each vertex from the root, as explained below) and later answer each LCA query in constant time. (2) Had our input tree been a complete binary tree, it would have been possible to preprocess it (by way of computing its inorder number, as explained below) and later to answer each LCA query in constant time.

The preprocessing stage assigns a number INLABEL (v) to each vertex v in T. Motivated by observation (1), these numbers satisfy the following *Path-Partition Property*: The INLABEL numbers partition the tree T into paths, called INLABEL paths. Each INLABEL path consists of the vertices that have the same INLABEL number.

Let B be the smallest complete binary tree having at least n vertices. Our description identifies each vertex in B by its inorder number. Motivated by observation (2), the INLABEL numbers also satisfy the following *Descendance-Preservation Property*: The INLABEL numbers map each vertex v in T into the vertex INLABEL (v) in B, such that the descendants of v are mapped into descendants of INLABEL (v) in B (v is considered both a descendant and an ancestor of itself).

Consider a vertex v in T. By the Descendance-Preservation Property all the ancestors of v are mapped into ancestors of INLABEL (v). This implies that there are at most $\log n$ distinct numbers among the INLABEL numbers of all the ancestors of v. Later, we show how to record all these INLABEL numbers using a single string of $\log n$ bits. In the preprocessing stage we compute this string, for each vertex v in T, into ASCENDANT (v).

In the preprocessing stage we also compute the table HEAD. It contains the highest vertex in every INLABEL path.

Section 4 describes how to process a query LCA (x, y) for any pair of vertices x, y in T. The processing breaks into two cases. The simpler case is where x and y belong to the same INLABEL path. In the preprocessing stage we compute for each vertex v in T its distance from the root into LEVEL (v). So, LCA (x, y) is simply the vertex among x and y that is closer to the root. The more complicated case is where INLABEL $(x) \neq$ INLABEL (y). We proceed in four steps. In the first step, we find the LCA of INLABEL (x) and INLABEL (y) in the complete binary tree B, denoted by b. Let $z =$ LCA (x, y) in T. In the second step, we find INLABEL (z). INLABEL (z) is the lowest ancestor of b in B that is the INLABEL number of a common ancestor of x and y in T. For this, we use information provided by ASCENDANT (x) and ASCENDANT (y). In the third and fourth steps we find z in the INLABEL path defined by INLABEL (z). In the third step, we find the lowest ancestor of x, denoted $\hat{x}$, and the lowest ancestor of y, denoted $\hat{y}$, in the path defined by INLABEL (z) in T. This is done in an indirect fashion. Consider the path in B from INLABEL (z) to INLABEL (x). We derive from ASCENDANT (x) the first INLABEL number (i.e., vertex of B) of an ancestor of x in this path. Table HEAD gives the highest ancestor of x in T having this INLABEL number. Finally, $\hat{x}$ is the father in T of this ancestor. We find $\hat{y}$ similarly. In the fourth step we find z, which is simply the vertex among $\hat{x}$ and $\hat{y}$ that is closer to the root.

3. The preprocessing stage. The outcome of the preprocessing stage consists of labels that are assigned to the vertices of T and a look-up table, called HEAD. The label of each vertex v in T consists of three numbers: INLABEL (v), ASCENDANT (v), and LEVEL (v).

We start with computing INLABEL (v), for each vertex v in T. This is done in two steps. After a discussion of these two steps we show how to implement them.

Let PREORDER (v) be the serial number of v in preorder traversal of T and SIZE (v) be the number of vertices in the subtree rooted at v. Definition of preorder traversal can be found, e.g., in [1, pp. 54–55].

Step 1. Compute PREORDER (v) and SIZE (v).

We note that the PREORDER numbers of the vertices in the subtree rooted at v range between PREORDER (v) and PREORDER $(v)+$SIZE $(v)-1$, and therefore, the closed interval [PREORDER (v), PREORDER $(v)+$SIZE $(v)-1$] is called the *interval of* v.

In Step 2 we consider the binary representation of the (integer) numbers in the interval of v. We remark that throughout this paper we alternately refer to numbers and to their binary representations. No confusion will arise.

Step 2. Find the (integer) number that has the maximal number of rightmost "0" bits in the interval of v. This number is assigned to INLABEL (v).

For an example of computations described in this section see Fig. 3.1.

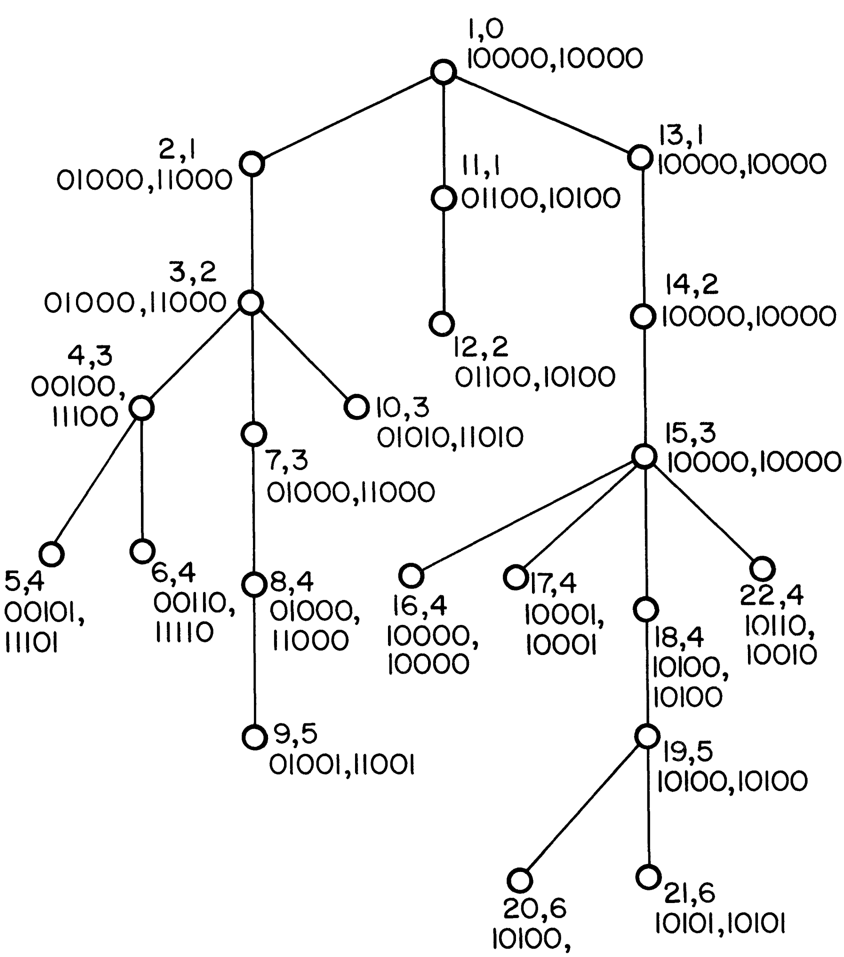

FIG. 3.1. *Example. A tree with four numbers*: PREORDER, LEVEL, INLABEL, *and* ASCENDANT *at each vertex.* (*The last two numbers are given in binary representation.*)

Discussion. We show that the INLABEL numbers satisfy the two properties defined in the high-level description of the previous section.

LEMMA 1. *The* INLABEL *numbers satisfy the Path-Partition Property.*

Proof. Observe that the intervals of the sons of v must be pairwise disjoint. Therefore, INLABEL (v) belongs to the interval of at most one son of v. Denote such a son by u. By the selection of the INLABEL numbers (Step 2), INLABEL(u) = INLABEL(v) (if u exists), and for any other son w of v, INLABEL$(w) \neq$ INLABEL (v). This implies the *Path-Partition Property* of the INLABEL numbers. □

LEMMA 2. *The* INLABEL *numbers satisfy the Descendance-Preservation Property.*

Proof. Let d be any descendant of v in T. We show that INLABEL(d) is a descendant of INLABEL (v) in the complete binary tree B. (Recall that our description identifies each vertex in B by its inorder number, thus proving the lemma.) Consider any two vertices b and c in B. We first give a necessary and sufficient condition for c to be a descendant of b in B and then show that INLABEL(d) and INLABEL(v) satisfy this condition. Let $l = \lfloor \log n \rfloor$[1] and i be the number of rightmost "0" bits in b. That is, b consists of $l-i$ leftmost bits followed by a single "1" and i "0"s.

CLAIM. *A vertex c is a descendent of b if and only if* (1) *the $l-i$ leftmost bits of c are the same as the $l-i$ leftmost bits of b, and* (2) *the number of rightmost* "0" *bits in c is at most i.*

Proof. Let b_L and b_R be the left and right sons of b, respectively. It is not difficult to see the following: (i) b_L consists of the $l-i$ leftmost bits of b followed by a single "0", a single "1", and $i-1$ "0"s; and (ii) b_R consists of the $l-i$ leftmost bits of b followed by two "1"s and $i-1$ "0"s. These facts readily imply both directions of our claim.

For an example of a complete binary tree and its inorder numbering see Fig. 3.2.

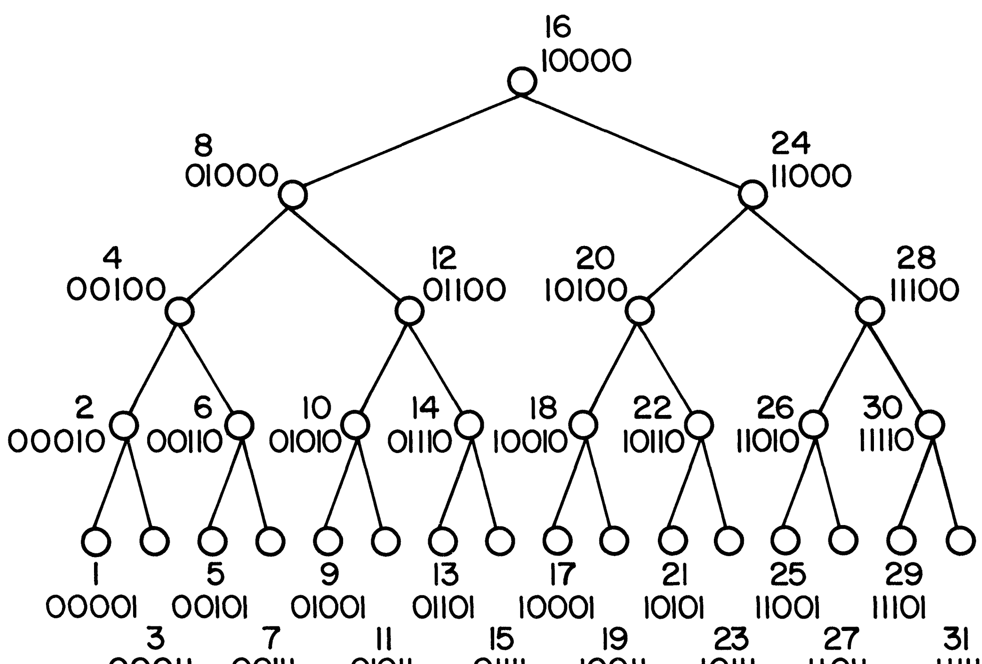

FIG. 3.2. *Example. Inorder numbering of the complete binary tree with* 31 *vertices.* (*The numbers are given also in binary representation.*)

[1] The base of all logarithms in this paper is two.

We return to the proof of Lemma 2. Let i be the number of rightmost "0" bits in INLABEL (v). Since INLABEL (d) belongs to the interval of v and INLABEL (v) has the maximal number of rightmost "0" bits in this interval, the number of rightmost "0" bits in INLABEL (d) is at most i. The $l-i$ leftmost bits are the same for all numbers in the interval. In particular, the $l-i$ leftmost bits in INLABEL (d) are the same as the $l-i$ leftmost bits in INLABEL (v). This implies that INLABEL (d) is the descendant of INLABEL (v) in B. Lemma 2 follows. □

Implementation. Step (1) is implemented in linear time and linear space, using preorder traversal of T. Given PREORDER (v) and SIZE (v), for each vertex v in T, Step (2) is implemented in constant time per vertex in two substeps.

Step 2.1. Compute $\lfloor \log [(\text{PREORDER}\ (v)-1)\ \textbf{xor}\ (\text{PREORDER}\ (v)+\text{SIZE}\ (v)-1)] \rfloor$ into i. Let us explain this. The bitwise logical exclusive OR (denoted **xor**) of PREORDER $(v)-1$ and PREORDER $(v)+$ SIZE $(v)-1$ assigns "1" to each bit in which PREORDER $(v)-1$ and PREORDER $(v)+$ SIZE $(v)-1$ differ. The floor of the (base two) logarithm gives the index of the leftmost bit of difference (counting from the rightmost bit whose index is zero). Note that the bit-indexed i must be "0" in PREORDER $(v)-1$ and "1" in PREORDER $(v)+$ SIZE $(v)-1$, since the second number is larger.

Step 2.2 shows how to "compose" INLABEL (v). For this, we need two observations. (1) The $l-i+1$ leftmost bits of INLABEL (v) are the same as the $l-i+1$ leftmost bits in PREORDER $(v)+$ SIZE $(v)-1$. (2) The i other bits in INLABEL (v) are "0"s.

Step 2.2. Compute $2^i \lfloor (\text{PREORDER}\ (v)+\text{SIZE}\ (v)-1)/2^i \rfloor$ into INLABEL (v). This assigns the $l-i+1$ leftmost bits in PREORDER $(v)+$ SIZE $(v)-1$ to the $l-i+1$ leftmost bits in INLABEL (v) and "0"s to the other bits of INLABEL (v).

Remark. The above computation is based on PREORDER numbering of the vertices of T. This numbering has the property that the numbers assigned to the subtree rooted at any vertex of T provide a consecutive series of integers. In fact, any alternative numbering having this property (e.g., POSTORDER, INORDER) will produce INLABEL numbers that will be suitable for our preprocessing stage.

We proceed to the computation of the ASCENDANT numbers. The general idea is that for each vertex v, the single number ASCENDANT (v) will record the INLABEL numbers of *all* the ancestors of v in T. We observe that, from the viewpoint of vertex v the INLABEL number of each of its ancestors can be fully specified by the index of its rightmost "1". This is so because the bits that are to the left of this "1" are the same as their respective bits in INLABEL (v). Like the INLABEL numbers, ASCENDANT (v) is also an $(l+1)$-bit number. Denote the binary representation of ASCENDANT (v) by the binary sequence $A_l(v), \cdots, A_0(v)$. We set $A_i(v)=1$ only if i is the index of a rightmost "1" in the INLABEL number of an ancestor of v in T. To compute the ASCENDANT numbers, we scan the vertices of T from its root r down to its leaves (use, for instance, Breadth-First Search). We start with ASCENDANT $(r)=2^l$. Consider an internal vertex v in T and let $F(v)$ be the father of v in T. If INLABEL $(v)=$ INLABEL $(F(v))$ then we assign ASCENDANT $(F(v))$ to ASCENDANT (v); otherwise, we assign ASCENDANT $(F(v))+2^i$ to ASCENDANT (v), where i is the index of the rightmost "1" in INLABEL (v). It can be easily verified that i is given by $\log(\text{INLABEL}\ (v)-[\text{INLABEL}\ (v)\ \textbf{and}\ (\text{INLABEL}\ (v)-1)])$, where **and** denotes bitwise logical AND.

Recall that LEVEL (v), for each vertex v in T, is the distance, counting edges, of the path from v to the root r. Computation of the LEVEL numbers is straightforward and can be done using, e.g., Breadth-First Search.

Recall that Fig. 3.1 gives an example of the labels.

We conclude by describing how to compute the Table HEAD. HEAD (k) contains the vertex closest to the root in the path consisting of all vertices whose INLABEL number is k. HEAD (k) is sometimes called the *head* of the INLABEL path k. Computation of the table HEAD is trivial. For each vertex v, such that INLABEL $(v) \neq$ INLABEL $(F(v))$ we assign v to HEAD (INLABEL (v)). This, again, takes linear time and linear space.

A general implementation remark. The time bounds of both the preprocessing stage and the query processing depend on the ability to perform multiplication, division, powers-of-two computation, bitwise AND, base-two discrete logarithm, and bitwise exclusive OR in constant time. If these operations are not part of the machine's repertoire, look-up tables for each missing operation are prepared in linear time and linear space as part of the preprocessing stage. These tables will be used to perform the missing operations in $O(1)$ operations in the repertoire.

4. Processing LCA queries. In this section we show how to answer LCA queries using the outcome of the preprocessing stage.

Consider a query LCA (x, y), for any pair of vertices x, y in T. (To illustrate the presentation the reader is referred to Fig. 3.1.) There are two cases.

Case A. INLABEL (x) = INLABEL (y). It must be that x and y are in the same INLABEL path. We conclude that LCA (x, y) is x if LEVEL $(x) \leqq$ LEVEL (y) and y otherwise.

Case B. INLABEL $(x) \neq$ INLABEL (y). Let z be LCA (x, y). We find z in four steps:

Step 1. Find b, the LCA of INLABEL (x) and INLABEL (y) in the complete binary tree B, as follows. Let i be the index of the rightmost "1" in b. Since b is a common ancestor of INLABEL (x) and INLABEL (y) in B, i must satisfy the following two conditions. (1) The $l-i$ leftmost bits in INLABEL (x) and in INLABEL (y) are the same as these bits in b. (2) The index of the rightmost "1" in INLABEL (x) and in INLABEL (y) is at most i. Since b is the *lowest* common ancestor of INLABEL (x) and INLABEL (y) in B, i is the *minimum* index satisfying both conditions. We distinguish three *cases*.

Case (1). INLABEL (x) is an ancestor of INLABEL (y). Let i_1 be the index of the rightmost "1" in INLABEL (x). Note that in this case the $l-i_1$ leftmost bits in INLABEL (x) and in INLABEL (y) are the same and that the index of the rightmost "1" in INLABEL $(y) < i_1$. Hence, i equals i_1.

Case (2). INLABEL (y) is an ancestor of INLABEL (x). Similar to Case (1), i is the index of the rightmost "1" in INLABEL (y).

Case (3). Not cases (1) and (2). In this case i is the *minimum* index such that the $l-i$ leftmost bits in INLABEL (x) and INLABEL (y) are the same.

We can deal with all three cases at once by simply taking i to be the maximum among the following: the index of the leftmost bit in which INLABEL (x) and INLABEL (y) differ; the index of the rightmost "1" in INLABEL (x); and the index of the rightmost "1" in INLABEL (y). b consists of the $l-i$ leftmost bits in INLABEL (x) (or INLABEL (y)) followed by a single "1" and i "0"s.

In Step 2 we find INLABEL (z) (where z is LCA (x, y)). The Descendance-Preservation Property of the INLABEL numbers implies that INLABEL (z) is a common ancestor of INLABEL (x) and INLABEL (y). Notice that INLABEL (z) is not necessarily b, the *lowest* common ancestor of INLABEL (x) and INLABEL (y). This is so because the vertices in T mapped into b are not necessarily ancestors of x

or y. However, it is not difficult to see that INLABEL (z) is the lowest ancestor of b in B that is the INLABEL number of an ancestor of both x and y in T.

Step 2. Find INLABEL (z). For this we find the index of the rightmost "1" in INLABEL (z), denoted by j. Since z is a common ancestor of x and y in T, $A_j(x)=1$ and $A_j(y)=1$. Since INLABEL (z) is the *lowest* ancestor of b that is a common ancestor of x and y, the index j must be the index of the *rightmost* "1" in $A_l(x), \cdots, A_i(x)$ and $A_l(y), \cdots, A_i(y)$. INLABEL (z) consists of the $l-j$ leftmost bits of INLABEL (x) (or INLABEL (y)) followed by a single "1" and j "0"s.

In the next steps we find z, the lowest vertex in the path defined by INLABEL (z) that is a common ancestor of x and y in T. For this we find $\hat{x}$, the lowest ancestor of x in the path defined by INLABEL (z) and $\hat{y}$, the lowest ancestor of y in this same path. z is the highest vertex among these two vertices.

Step 3. Find $\hat{x}$ and $\hat{y}$. We show how to find $\hat{x}$. $\hat{y}$ is found similarly. If INLABEL $(x)=$ INLABEL (z) then $\hat{x}=x$ and nothing has to be done. Suppose INLABEL $(x) \neq$ INLABEL (z). We set the following intermediate goal, as the main step toward finding $\hat{x}$: Find the son of $\hat{x}$ that is also an ancestor of x. Denote the vertex that we search by w and let k be the index of the rightmost "1" in INLABEL (w). It is not difficult to verify that k is the index of the leftmost "1" in $A_{j-1}(x), \ldots, A_0(x)$. So, we find k. Clearly, INLABEL (w) consists of the $l-k$ leftmost bits of INLABEL (x) followed by a single "1" and k "0"s. Observe that w is the head of its INLABEL path (since the INLABEL number of its father $\hat{x}$ is different from INLABEL (w)). Therefore, w is HEAD (INLABEL (w)) and our intermediate goal is achieved. Finally, $\hat{x}$ is the father of w.

Step 4. LCA(x, y) is $\hat{x}$ if LEVEL$(\hat{x}) \leqq$ LEVEL$(\hat{y})$ and $\hat{y}$ otherwise.

In the rest of this section we give additional implementation details required for the above processing.

Step 1. To find i, the index of the rightmost "1" in b, we do the following.

Step 1.1. Find i_1, the index of the rightmost "1" in INLABEL (x), and i_2, the index of the rightmost "1" in INLABEL (y). To find i_1 we compute $i_1 := \log(\text{INLABEL}(x) - [\text{INLABEL}(x)\ \textbf{and}\ (\text{INLABEL}(x)-1)])$, as in the ASCENDANT numbers computation of the previous section. i_2 is found similarly.

Step 1.2. Find i_3, the index of the leftmost bit in which INLABEL (x) and INLABEL (y) differ. To find i_3 we compute $i_3 := \lfloor \log[\text{INLABEL}(x)\ \textbf{xor}\ \text{INLABEL}(y)] \rfloor$. This is similar to Step 2.1 in the INLABEL numbers computation of the previous section.

i is the maximum among i_1, i_2, and i_3. Given i, b can be computed similarly to Step 2.2 in the INLABEL numbers computation.

Step 2. To find j we do the following steps.

Step 2.1. Compute the bitwise logical AND of ASCENDANT (x) and ASCENDANT (y) into COMMON.

Step 2.2. Compute $2^i \lfloor \text{COMMON}/2^i \rfloor$ into COMMON$_i$. COMMON$_i$ lists all the "1"s in both $A_l(x), \cdots, A_i(x)$ and $A_l(y), \cdots, A_i(y)$.

Step 2.3. j is the index of the rightmost "1" in COMMON$_i$. To find j we compute $j := \log(\text{COMMON}_i - [\text{COMMON}_i\ \textbf{and}\ (\text{COMMON}_i - 1)])$, as in the ASCENDANT numbers computation of the previous section.

The implementation of Step 3 uses the same techniques.

5. The parallel preprocessing algorithm. In this section we describe the parallel version of our preprocessing stage. It runs in $O(\log n)$ time using $n/\log n$ processors.

We make the following assumption regarding the representation of the input tree T. Its $n-1$ edges are given in an array, where the incoming edges of each vertex are grouped successively. By our definition of the tree T, its edges are directed towards the root.

Computing the labels in parallel. To compute the labels of the vertices in T we apply the Euler tour technique for computing tree functions, which was given in [10] and [12]. We will implement it, however, using the $O(\log n)$ time optimal parallel list-ranking algorithm of [3]. This list-ranking algorithm is designed for an EREW PRAM. It is based on expander graphs and its $O(\log n)$ time bound hides a constant that is not very small. We note that [4] recently gave an alternative list-ranking algorithm with the same time and processor efficiencies. This alternative algorithm is designed for a PRAM that allows simultaneous access to the same memory location for both read and write purposes (called CRCW PRAM). It is simpler and its $O(\log n)$ time bound requires a small constant.

Below, we first recollect the construction required for the Euler tour technique. We then show how to use it for computing the labels. The only reason we were forced to present anew the Euler tour technique is that the computation of the ASCENDANT numbers has not appeared elswhere.

Step 1. For each edge $(v \to u)$ in T we add its antiparallel edge $(u \to v)$. Let H denote the new graph.

Since the indegree and outdegree of each vertex in H are the same, H has an Euler path that starts and ends in r. Step 2 computes this path into the vector of pointers D, where for each edge e of H, $D(e)$ will have 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 $(v \to u_0), \cdots, (v \to u_{d-1})$.) $D(u_i \to v) := (v \to u_{(i+1) \bmod d})$, for $i = 0, \cdots, d-1$. Now D has an Euler circuit. The "correction" $D(u_{d-1} \to r) :=$ *end-of-list* (where the outdegree of r is d) gives an Euler path which starts and ends in r.

We show how to use the Euler path in order to find PREORDER (v), PREORDER (v) + SIZE $(v) - 1$, and LEVEL (v) for each vertex v in T.

Step 3. We assign two weights: $W_1(e)$ and $W_2(e)$ to each edge e in the Euler path as follows. (1) $W_1(e) = 1$ if e is directed from r (that is, if e is not a tree edge), and $W_1(e) = 0$ otherwise. (2) $W_2(e) = 1$ if e is directed from r, and $W_2(e) = -1$ otherwise.

Step 4. We apply twice an optimal logarithmic time parallel list-ranking algorithm to find for each e in H its (weighted) distance from the *start* of the Euler path: The first application is relative to the weights W_1 and the result is stored in DISTANCE$_1$ (e); the second application is relative to the weights W_2 and the result is stored in DISTANCE$_2$ (e). Consider a vertex $v \neq r$ and let u be its father in T. PREORDER (v) is DISTANCE$_1$ $(u \to v) + 1$, PREORDER (v) + SIZE $(v) - 1$ is DISTANCE$_1$ $(v \to u) + 1$, and LEVEL (v) is DISTANCE$_2$ $(u \to v)$. (These claims can be readily verified by the reader.)

Step 5. Given PREORDER (v) and PREORDER (v) + SIZE $(v) - 1$ for each vertex v in T we compute INLABEL (v) in constant time using n processors as in the serial algorithm.

Next, we show how to use the Euler path in order to find ASCENDANT (v) for each vertex v in T.

Step 6. We assign a (new) weight $W(e)$ to each edge e in the Euler path as follows. For each vertex $v \neq r$ we do the following. Let u be the father of v in T and let i be the index of the rightmost "1" in INLABEL (v). If INLABEL $(v) \neq$ INLABEL (u), we assign $W(u \to v) = 2^i$ and $W(v \to u) = -2^i$. The weight of all other edges is set to zero.

Step 7. We again apply a parallel list-ranking algorithm to find for each e in H its (weighted) distance from the start of the Euler path. Consider a vertex $v \neq r$ and let u be its father in T. ASCENDANT (v) is the distance of the edge $(u \rightarrow v)$ plus 2^l. Clearly, ASCENDANT $(r) = 2^l$.

We note that, given the labels, the table HEAD can be computed in constant time using n processors.

Complexity. Each of steps 4 and 7 needs $n/\log n$ processors and $O(\log n)$ time. Each of Steps 1, 2, 3, 5, 6 and the computation of HEAD needs n processors and $O(1)$ time and can be readily simulated by $n/\log n$ processors in $O(\log n)$ time. Thus, the parallel preprocessing stage can be done in a total $O(\log n)$ time using $n/\log n$ processors.

Acknowledgments. We are grateful to Noga Alon and Yael Maon for stimulating discussions. We also thank the anonymous referee for drawing our attention to [9].

REFERENCES

[1] A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] A. APOSTOLICO, C. ILIOPOULOS, G. M. LANDAU, B. SCHIEBER, AND U. VISHKIN, *Parallel construction of a suffix tree with applications*, Algorithmica Special Issue on Parallel and Distributed Computing, to appear.

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

[4] ———, *Faster optimal parallel prefix sums and list ranking*, TR 56/86, the Moise and Frida Eskenasy Institute of Computer Science, Tel Aviv University, Tel Aviv, Israel, 1986.

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

[6] G. M. LANDAU AND U. VISHKIN, *Introducing efficient parallelism into approximate string matching*, in Proc. 18th Annual ACM Symposium on Theory of Computing, 1986, pp. 220-230.

[7] G. M. LANDAU, B. SCHIEBER, AND U. VISHKIN, *Parallel construction of a suffix tree*, in Proc. 14th Internat. Colloquium on Automata Language and Programming, Lecture Notes in Computer Science 267, Springer-Verlag, Berlin, New York, 1987, pp. 314-325.

[8] 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.

[9] Y. H. TSIN, *Finding lowest common ancestors in parallel*, IEEE Trans. Comput., 35 (1986), pp. 764-769.

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

[11] U. VISHKIN, *Synchronous parallel computation—a survey*, TR-71, Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, New York, NY, 1983.

[12] ———, *On efficient parallel strong orientation*, Inform. Process. Lett., 20 (1985), pp. 235-240.