Parallel Ear Decomposition Search (EDS)
And ST-Numbering In Graphs

by

Yael Maon [1]
Baruch Schieber [1,3]
Uzi Vishkin [1,2,3]

Ultracomputer Research Laboratory
Courant Institute of Mathematical Sciences
715 Broadway, 10th Floor
New York, NY 10003

Ultracomputer Note #102
Computer Science Department Technical Report #222
February, 1986

[1] Department of Computer Science School of Mathematical Sciences Tel Aviv University Tel Aviv, Israel 69978.

[2] Department of Computer Science Courant Institute of Mathematical Sciences New York University 251 Mercer St., New York, N.Y. 10012. The research of this author was supported by NSF grant NSF-DCR-8318874 and ONR grant N0014-85-K-0046.

[3] The research of these authors was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U. S. Department of Energy under contract number DE-AC02-76ER03077.

ABSTRACT

[LEC-67] linear time serial algorithm for testing planarity of graphs uses the linear time serial algorithm of [ET-76] for *st*-numbering. This *st*-numbering algorithm is based on depth-first search (DFS). A known conjecture states that DFS, which is a key technique in designing serial algorithms, is not amenable to poly-log time parallelism using "around linearly" (or even polynomially) many processors. The first contribution of this paper is a general method for searching efficiently in parallel undirected graphs, called *ear-decomposition search* (EDS).

The second contribution demonstrates the applicability of this search method. We present an efficient parallel algorithm for *st*-numbering in a biconnected graph. The algorithm runs in logarithmic time using a linear number of processors on a concurrent-read concurrent-write (CRCW) PRAM. An efficient parallel algorithm for the problem did not exist before. The problem was not even known to be in NC.

1. Introduction

We define the problems considered in this paper. For all these problems we use the same input.

Input. An undirected graph $G(V,E)$ and some specified edge $e = (s,t)$ in E. (Denote $n = |V|$ and $m = |E|$)

Let P_0 be the path that consists of the edge e. An *ear decomposition* of G *starting with* P_0 is a decomposition $E = P_0 \cup P_1 \cup \cdots \cup P_k$, where P_{i+1} is a simple path whose endpoints belong to $P_0 \cup \cdots \cup P_i$, but its internal vertices do not. A simple path P_i is called an *ear*. It is called an *open ear* if the two endpoints of P_i do not coincide, and a *closed* ear otherwise. An ear decomposition is called *open* if all its ears are open.

The ear decomposition problem. Find an ear decomposition starting with P_0.

The open ear decomposition problem. Find an open ear decomposition starting with P_0.

An one-to-one function f from V to $\{1,...,n\}$ is called an $st-numbering$ if it satisfies: (i) $f(s)=1$ and $f(t)=n$, and (ii) for each $v \in V-\{s,t\}$ there exist adjacent vertices v_1 and v_2 such that $f(v_1)<f(v)<f(v_2)$.

The st-numbering problem. Find an *st*-numbering of G.

A graph G has an open ear decomposition starting with an edge iff G is biconnected (see [W-32]). A graph G is biconnected iff it has an *st*-numbering for every given edge (s,t) (see [LEC-67]).

The model of parallel computation used in this paper is the concurrent-read concurrent-write (CRCW) parallel random access machine (PRAM). A PRAM employs p synchronous processors all having access to a common memory. A CRCW PRAM allows simultaneous access by more than one processor to the same memory location for either reads or writes. We use the following concurrent-write convention. In case several processors seek access to the same memory location for write purposes, one of them succeeds but we do not know in advance which. See [V-83] for a survey of results concerning PRAMs.

For each of the three problems mentioned above we give a parallel algorithm which runs in $O(\log n)$ time using $n+m$ processors on a CRCW PRAM. Alternative parallel implementations attaining optimal speed up where $m > n \log n$ are also given for each problem. These alternative algorithms run in time $O(\log n \log^{(2)} n \log^{(3)} n)$, where $\log^{(k)}$ is the k^{th} iterate of the log function.

[L-85] showed that the ear decomposition problem has a parallel algorithm which runs in poly-log time using a polynomial number of processors and is therefore in the class NC. A remark at the end of Section 2.1, explains why we believe that it was not known whether the *open* ear decomposition problem is in the class NC. We are also not aware of any parallel algorithm for *st*-numbering. Apparently, it was not even known if *st*-numbering is in NC. Here, we do not only determine that these two problem are in NC, but actually give very efficient parallel algorithms. Each of these algorithms runs in logarithmic time using a linear number of processors.

Serial graph algorithms use two main techniques for searching graphs: Depth-First Search (DFS) and Breadth-First Search (BFS). Both techniques seem to be not ideal for parallel computation. The most efficient poly-log time parallel implementations of BFS require a number of processors which is cubic in the number of vertices. Many researchers have conjectured that DFS is not even in NC. One of the most challenging tasks in parallel computation is to cope to with this apparent intractability of DFS. One approach is not to give up and try to get the most out of DFS as in [A-85]. This approach is interesting from the theoretical point of view since it bounds the limits of DFS in parallel. We believe, however, that when it comes to actually designing efficient parallel algorithms, a more realistic approach should be taken. That is, BFS and DFS should be replaced by new techniques which are amenable to parallel computation. This approach of looking for new techniques to replace DFS has been practiced in designing efficient parallel algorithms for biconnectivity [TV-85] and strong orientation [V-85]. Observe, however, that unlike our problems, it was trivial to establish the membership of biconnectivity and strong orientation in NC using simple transitive closure algorithms.

Ear-decomposition has the flavor of a *general* search technique in graphs. It arranges the vertices of the graph by partitioning them into paths. This enables further exploration of the graph in an "orderly" manner. We call this search technique $ear-decomposition\ search$ (EDS).

For more motivation for considering the ear decomposition problems, we refer the reader to [L-85].

The serial algorithm for *st*-numbering is considered as one of the classical serial graph algorithms (see [E-79]). The linear time algorithm for the problem [ET-76] is heavily based on DFS. The present paper copes with the apparent intractability of DFS for parallel computation by providing a new algorithm for the problem. The new algorithm is based on the new EDS technique for ear decomposition of a graph, thereby demonstrating its applicability. The algorithm is also based on refined insights into the *st*-numbering problem. Interestingly, our algorithm provides also a new linear time serial algorithm for *st*-numbering.

The *st*-numbering is used as an important component in several serial algorithms. Most known is the planarity testing algorithm of [LEC-67] which was improved to run in linear time, following the linear time algorithm for *st*-numbering of [ET-76]. Our algorithm gives hope for finding efficient parallel planarity testing using the approach of [LEC-67]. It is yet unresolved whether there exists an efficient parallel algorithm for planarity testing. Note that the involved planarity testing of [JS-82] still leaves much to be desired, as it runs on $O(\log^2 n)$ time using $O(n^{3.29} / \log^2 n)$ processors.

Another application for *st*-numbering is mentioned in [IR-84] for the following problem. Given an undirected biconnected graph find two spanning trees T_1 and T_2 which are rooted at the same vertex r so that the paths from r to v in T_1 and T_2 are vertex disjoint (except r and v). This problem is important for reliable communication as was indicated in that paper.

In Section 2 we present a parallel algorithm for computing an ear-decomposition and an open ear-decomposition. In Section 3 we use the open ear decomposition to compute *st*-numbering of a biconnected graph G. The numbering is done in two stages: (1) We orient the edges of the input graph. As a result we get a directed acyclic graph such that each of its vertices is on a path from s to t. (2) We topologically sort of this digraph to get the *st*-numbering. The whole subtlety of the algorithm lies in the orientation of Stage 1. Through a careful case analysis we perform this orientation based on considerations which are essentially local. We were surprised by the fact that these local considerations were sufficient. Another intriguing comment is that one of the interesting open problems in parallel computation is whether there is a poly-log time algorithm for topological sort that uses linearly many processors. We have examples where our topological sort algorithm fails on some acyclic digraphs. However, we can prove that this topological sort algorithm will work correctly on all acyclic digraphs which can be obtained as outcome of the first stage.

2. Ear-Decomposition In Parallel

Let $G(V,E)$ be an undirected biconnected graph. We give an algorithm for finding an open ear decomposition of G starting with (s,t), where $(s,t) \in E$. A parallel implementation of this algorithm is described at the end of the section.

Definitions: Let $T(V,E)$ be a tree which is rooted at t.

(1) For $v \in V$, $LEVEL(v)$ is the length, counting edges, of the unique path in T from t to v.

(2) For $v \in V$, $F(v)$ is the father of v in T.

(3) For $u,v \in V$, $LCA(u,v)$ is the lowest common ancestor of u and v in T.

(4) Let $u,v \in V$, where u is not an ancestor of v. We denote by $e_{u,v}$ the first edge in the unique path in T from $LCA(u,v)$ to u.

In order to compute an open ear-decomposition, we first find an ear-decomposition of G which is not necessarily open. This is described in the first subsection below. The second subsection shows how to modify this ear-decomposition into an open ear-decomposition.

2.1. Finding an ear-decomposition

Instead of designing a new algorithm for finding an ear decomposition, we use a parallel algorithm for a different problem which was given in [V-85]. There, the algorithm assigns directions to the edges of a connected bridgeless undirected graph so that the resulting directed graph is strongly connected. We note that we obtain the same ear-decomposition as in [L-85]. However, our parallel algorithm is more efficient. For completeness of the presentation, we outline below the algorithm for finding an ear-decomposition.

Step 1. Find a spanning tree $T(V,E_T)$ rooted at t of G, such that (s,t) will be the only tree edge whose endpoint is t. This is done as follows: First, find a spanning tree of the subgraph induced by the vertices $V - \{t\}$. Finally, add the vertex t and the edge (s,t) to the tree. (*Remark:* The fact that G is biconnected implies that the subgraph induced by $V - \{t\}$ is connected.) Step 1 is the only step in this subsection which is not completely identical to the algorithm of [V-85]. There, any spanning tree will do.

Edges in $E_T - \{(s,t)\}$ will be referred to as tree edges. Edges in $E - E_T$ will be referred to as non-tree edges. The edge (s,t) will have a special status.

In Step 2 we assign a number to each non-tree edge.

Step 2.

(a) For each $v \in V$, compute $LEVEL(v)$ and $F(v)$.

For each non-tree edge (u,v) compute $LCA(u,v)$.

(b) Assume that each edge $e \in E$ has a serial number $1 \le SERIAL(e) \le m$. For each non-tree edge e, let $NUMBER(e) = (LEVEL(LCA(e)), SERIAL(e))$.

We define a lexicographic order $<_L$ on these numbers as follows. Let e, e' be non-tree edges. $NUMBER(e) <_L NUMBER(e')$ if $NUMBER(e).1 < NUMBER(e').1$, or $NUMBER(e).1 = NUMBER(e').1$ and $NUMBER(e).2 < NUMBER(e').2$, where $NUMBER(x).i$ stands for the i^{th} coordinate of the pair $NUMBER(x)$.

Let f be a non-tree edge. We call the simple cycle which is formed by f together with the edges in E_T the *cycle of f*.

Step 3. Each tree edge e "considers" all non-tree edges f such that e is in their cycle. Among them e selects an edge f whose $NUMBER(f)$ is minimal (according to $<_L$) to be its *master edge*. The edge f is denoted $MASTER(e)$.

Proposition 2.1.1. Each non-tree edge e together with all the tree edges which selected it as their master edge form a simple path or a simple cycle. We call this path or cycle the *ear of e*.

Observe that the order $<_L$ on the non-tree edges induces an order on the defined ears. In addition, define the edge (s,t) as the first ear.

Proposition 2.1.2. The defined ears and the order yield an ear-decomposition.

Note that some of these ears may be simple cycles and therefore the resulted ear-decomposition may be not open. A closed ear may occur when a non-tree edge (u,v) is a master edge of all the tree edges in its cycle. (For an example where an arbitrary order on the serial numbers of the non-tree edges imply such a closed ear, see edge n_2 in Fig. 2.1, assuming that the serial number of n_2 is smaller than that of n_1.)

Remark: Unlike a remark in Lovasz's paper, Fig. 2.1 demonstrates that it is possible to get a closed ear whose endpoint is neither the root nor a separating vertex. The open ear decomposition proposed in [L-85] is based on this erroneous remark.

2.2. Finding an open ear-decomposition

In order to have a decomposition into *open* ears we have to define a subtler order on the non-tree edges. We show how to do it by reordering the non-tree edges which have the same *LCA*. Specifically, this reordering is achieved by changing only the second component in the pair $(LEVEL(LCA(e)),SERIAL(e))$. First, we take a closer look at the situation where a closed ear occurs. Let $(u,v) \in E - E_T$ and $x = LCA(u,v)$ and suppose $x \neq u$. If $v = x$ then the ear of (u,v) is closed iff $e_{u,v}$ chooses (u,v) as its master. Otherwise (i.e., $v \neq x$ and hence $e_{v,u}$ is defined), the ear of (u,v) is closed iff both $e_{u,v}$ and $e_{v,u}$ choose (u,v) as their master. Suppose, for instance, that $e_{u,v}$ (or $e_{v,u}$) is on the cycle of a non-tree edge *f* such that $LEVEL(LCA(f)) < LEVEL(x)$. Here, no matter how unfortunate we are with the initialization of *SERIAL*, the ear of *e* cannot be closed since (u,v) will never be the master of $e_{u,v}$. Therefore, we can characterize all the difficult cases, as the ones in which $LCA(MASTER(e_{u,v})) = LCA(MASTER(e_{v,u})) = x$ if $v \neq x$ and $LCA(MASTER(e_{u,v})) = x$ otherwise. (*MASTER*(.) is the same as in the previous subsection.) This leads to the following observation, which motivated us in changing the above ear-decomposition into an open ear-decomposition.

Observation. Consider any assignment of numbers into *SERIAL* of the non-tree edges so that for each non-tree edge (u,v), at least one of $e_{u,v}$ or $e_{v,u}$ does not choose (u,v) as its master. Then, the above ear-decomposition must be open.

This change is done using an auxiliary (not necessarily connected) graph H_x for every vertex *x* in $V - \{t\}$.

Step 4. For every vertex *x* in $V - \{t\}$, we construct the bipartite undirected graph $H_x = (V_x, E_x)$, as follows:

$V_x = \{ [u,v] \mid (u,v) \in E - E_T \ and\ LCA(u,v) = x\} \cup \{ [u,F(u)] \mid F(u) = x\}$.

In words, the vertices V_x are all non-tree edges whose *LCA* is *x* (to be called *non-tree vertices*) and all tree edges connecting *x* with a son in the tree (to be called *tree vertices*).

Let $[u,v]$ be a non-tree vertex in V_x such that *u* is not an ancestor of *v* in *T*. Let $e_{u,v}$ be the tree edge (w,x). Then the edge $([u,v],[w,x])$ is in E_x.

The graph H_s which relates to the graph *G* of Fig. 2.1 is given in Fig. 2.2.

Step 5. For each graph H_x, compute its connected components and a spanning forest.

Main Lemma. Consider a spanning tree of a connected component of one of these auxiliary graphs H_x. Then, at least one of its tree vertices $[w,x]$ (where $x = F(w)$) must satisfy $LEVEL(LCA(MASTER(w,x))) < LEVEL(x)$.

Relevance of Main Lemma. Let $[u,v]$ be a non-tree vertex in H_x and let (w,x) be a tree vertex in the spanning tree of the connected component of $[u,v]$ in H_x whose existence is guaranteed by the Main Lemma. Consider the path leading from $[w,x]$ to $[u,v]$ in this spanning tree. This path alternates between tree and non-tree vertices. Let us denote it t_1 $(=[w,x]), n_1, t_2, n_2, \ldots, t_\alpha, n_\alpha$ $(=[u,v])$, where t_i stands for a tree vertex and n_i for a non-tree vertex. Observe that the edge (represented by) t_i belongs to the cycle of (the edge represented by) n_i, for $1 \le i \le \alpha$, and to the cycle of n_{i-1}, for $2 \le i \le \alpha$. We aim that none of the ears of $n_1, n_2, \ldots, n_\alpha$ will be closed. In view of the above Observation, we simply take care that the edge of each tree vertex t_i will not select the edge of the non-tree vertex n_i as its master. For this, Step 6 below numbers the non-tree vertices on this path in ascending order (i.e., so that $SERIAL(n_1) < \cdots < SERIAL(n_\alpha)$). This implies that for $2 \le i \le \alpha$, the edge of t_i will not select the edge of n_i as its master. In addition, the Main Lemma implies that $LEVEL(LCA(MASTER(w,x))) < LEVEL(x)$, implying that t_1 which belongs to the cycle of n_1 is not included in the ear of n_1.

Proof of Main Lemma. Take any spanning tree as in the Lemma. The proof proceeds as follows. We define a non empty subgraph G_1 of G which is a function of this spanning tree. We show that if the Lemma does not hold with respect to this spanning tree then $G_1 \ne G$ and G_1 is a biconnected component of G. This would contradict the biconnectivity of G and therefore the Lemma is correct. Let us define $G_1(V_1, E_1)$. Recall that all the tree vertices in the spanning tree are of the form $(v, F(v))$, where $F(v) = x$. The vertices of G_1 are: (1) All the vertices v such that $[v,x]$ is a tree vertex in the spanning tree. (2) All vertices of G which are descendants of the vertices in Item (1) in the spanning tree of G which was found in Step 1. (3) x itself.

G_1 is induced by these vertices. That is, the edges of G_1 are all the edges of G which are incident to two vertices in G_1.

Suppose H_x has more than one connected component. Let $G_2(V_2, E_2)$ be a similar subgraph of G which is defined for the spanning tree of another connected component of H_x. We first show that it is impossible to have an edge (u,v) in E connecting any vertex u of $V_1 - \{x\}$ with any vertex v of $V_2 - \{x\}$. Clearly, $LCA(u,v)$ must be x itself. The edge $e_{u,v}$ is in the connected component of G_1 in H_x and $e_{v,u}$ is in the connected component of G_2 in H_x. Therefore, if (u,v) exists, then it must be connected to the tree vertices of $e_{u,v}$ and $e_{v,u}$ and therefore G_1 and G_2 are the same, contrary to our assumption.

Hence, if there is no edge (u,v) in E such that u is a vertex of $V_1 - \{x\}$ and $LEVEL(v) < LEVEL(x)$ in the tree of G, then the removal of x will disconnect G. This contradicts the biconnectivity of G.

Step 6.

(a) For the spanning tree of each connected component in each auxiliary graph H_x, find a tree edge (w,x) (where $F(w)=x$) such that $LEVEL(LCA(MASTER(w,x))) < LEVEL(x)$. (Its existence is guaranteed by the Main Lemma.) This tree edge will be referred to as *the edge* of its component.

(b) Root each such spanning tree at the edge of its component. Compute the numbering of the non-tree edges in a preorder traversal of the spanning tree. For every non-tree edge *e*, this numbering will be stored in $PREORDER(e)$.

(c) For each non-tree edge *e*, let $NEWNUMBER(e) = (LEVEL(LCA(e)), PREORDER(e))$.

For example, notice that the edge of the component given in Fig. 2.2 is t_1.

Step 7. Each tree edge reselects a master edge according to *NEWNUMBER* in the same way as the selection of a master edge in Step 3 above.

Proposition 2.2.1. Each non-tree edge *e* together with all the tree edges which selected it as their master edge form a simple path (which is not a cycle). This path is the *ear of e*.

Again, observe that the order $<_L$ on the non-tree edges induces an order on these ears. In addition, define the edge (s,t) as the first ear.

Proposition 2.2.2. These ears and order yield an open ear-decomposition.

Remarks:

(1) This algorithm can be extended to check if *G* is biconnected. In case that *G* is not biconnected it can be modified to compute the open ear-decomposition of each biconnected component.

(2) For the *st*-numbering algorithm, we may omit each non-tree edge *e* of *G* which forms an ear that consists of *e* only.

2.3. Parallel Implementation

We first present an implementation which runs in $O(\log n)$ time using $n + m$ processors. Appendix 1 presents an alternative implementation which runs in time $(\log n \log^{(2)} n \log^{(3)} n)$ using an optimal number of processors when $m > n \log n$.

Steps 1, 2 and 3 are essentially taken from [V-85]. These steps take $O(\log n)$ time using $n + m$ processors.

Remark. The algorithm of [V-85] implements steps 2 and 3 in three stages: (1) Each vertex *v* selects a single non-tree edge (u,v) such that $LCA(u,v)$ is the lowest common ancestor of $(v, v_1, \ldots, v_k)$, where $v_1, \ldots, v_k$ are all the vertices which are adjacent to *v* by an edge in $E - E_T$. (2) The original serial numbers of the non-tree edges, which were set arbitrarily, are modified in the following way. All the non-tree edges which were not selected in Stage 1 are assigned serial numbers which are greater than those assigned to the selected non-tree edges. (3) Each tree edge selects its master as in Step 3 above. In [V-85], it is shown that this, rather tedious, implementation is more efficient. It requires only *n* computations of *LCA* instead of *m* computations as in the direct implementation.

Finding the edges of the H_x's in Step 4 can be done in $O(\log n)$ time using $n + m$ processors. Note that the total number of edges in all auxiliary graphs is $O(n + m)$. Step 5 takes $O(\log n)$ time using $n + m$ processors using the parallel connectivity algorithm of [SV-82]. Step 6(a) (i.e., finding the edge of each component) needs $O(1)$ time using n processors, since we check for each tree edge whether it satisfies the inequality of Step 6(a). Step 6(b) requires rooting of each spanning tree and computation of preorder numbering. Using the Euler tour technique on trees, which has already been used extensively in our above references to [V-85], this can be done in $O(\log n)$ time using $n + m$ processors.

(*Remark:* Step 6(b) requires an adjacency list representation of the spanning forest. To obtain adjacency lists from a list of l edges, one may consider applications of the sorting algorithm of [AKS-83] that uses l processors to sort l elements in time $O(\log l)$ (when edges are compared according to a lexicographic order). Since we only need to sort numbers which are in a restricted domain we can apply the recent randomized radix sort algorithm of [R-85] which achieves almost-surely the same asymptotic time (with much more acceptable constants) using only $l/\log l$ processors. A third possibility is to perform this sorting in time $O(\log l)$ and l processors, using an adaptation of the simple notion of "orthogonal trees" (see [Th-83]). However, this takes more space.)

Step 6(c) needs $O(1)$ time using m processors. Step 7 can be implemented in the same way as Step 3 above. (Recall that for the *st*-numbering we may omit the edges which form an ear of length one. This can be done in $O(\log n)$ time using $m/\log n$ processors by applying a variation of a parallel "prefix sums" algorithm. See [FL-80].)

3. ST-Numbering In Parallel

In this section we describe how to compute in parallel the *st*-numbering of a biconnected graph G, given an open ear-decomposition of G.

The input of our algorithm is as follows:

(1) A graph $G(V,E)$ and a specified edge $(s,t) \in E$.

(2) An *open* ear decomposition $G = P_0 \cup P_1 \cup \cdots \cup P_l$ such that $P_0 = (s,t)$ and the endpoints of each P_i are in P_j and P_k, where $i > j \geq k \geq 0$. (Observe that j and k depend on i, $j = j(i), k = k(i)$.) The decomposition is given in the following form. Each edge in P_i knows the index i, and has pointers to its neighbor in P_i leading to P_j and to its neighbor in P_i leading to P_k. Unless otherwise stated, whenever we say an ear, we will mean one of these open ears.

Definitions:

(1) The endpoint of an ear P_i, which belongs to P_j (resp. P_k), is denoted by $L(P_i)$ (resp. $R(P_i)$). (Recall that, in the above notation, $j \geq k$.) In addition, s (resp. t) is defined to be $L(P_0)$ (resp. $R(P_0)$).

(2) The vertex in an ear P_i, $i \geq 1$, which is adjacent to $L(P_i)$ (resp. $R(P_i)$), is denoted by $LS(P_i)$ (resp. $RS(P_i)$).

(3) The vertex $L(P_i)$ is called the *anchor* of P_i.

(4) A vertex v of an ear P_i (denoted by $v \in P_i$) will be called an *internal* vertex of P_i, if v is neither $L(P_i)$ nor $R(P_i)$.

Remark. In our algorithm for *st*-numbering, each ear is "responsible for providing" the numbering of its internal vertices only. Therefore, we can ignore ears P which include only one edge. The ear P_0, whose only edge is (s,t), is the only ear of this type which is not ignored. This means that we have $O(n)$ relevant edges in the graph G.

In the next subsection we give a high-level description of the algorithm. A description of the parallel implementation is given in the sequel.

3.1. High-level description of the algorithm

The numbering is done in two stages. In Stage 1 we orient the edges of the input graph so that the resulting directed graph is acyclic and each of its vertices is on a path from *s* to *t*. In Stage 2 we compute a topological sort of this digraph. This results in assigning numbers from 1 to *n* to the vertices of the digraph. These numbers yield a valid *st*-numbering.

Remark: The topological sort of Stage 2 is applicable to digraphs obtained in Stage 1, but not to general acyclic digraphs.

For the orientation of the edges of Stage 1 we use the $ear-tree$ E_T which is defined as follows:

$V_T = \{P_i \mid P_i$ is an ear of $G\}$. That is, every ear is a vertex in *ET*.
$E_T = \{(P_i, P_j) \mid L(P_i)$ is an internal vertex of P_j, where $i \geq 1\}$.

Observation: *ET* is a directed tree, rooted at P_0.

The descendance relation in the tree *ET* extends naturally to the vertices of *G*. In particular, if an ear P_i is a descendant of an ear P_l (it might be P_i itself) in *ET*, we say that each vertex in P_i is a *descendant* of $L(P_l)$, the anchor of P_l. Conversely, such $L(P_l)$ is called an $anchor-ancestor$ of each vertex in P_i.

In order to motivate the orientation of the ears in Stage 1, which is quite involved, we start by presenting the main ideas of Stage 2 (the numbering algorithm). This will be followed by a description of Stage 1.

The principles of the numbering algorithm (Stage 2).

We assume that each ear P_i ($i \geq 1$) was oriented, in Stage 1, either from $L(P_i)$ to $R(P_i)$ or from $R(P_i)$ to $L(P_i)$. The ear P_0 was oriented from *s* to *t*.

Definition: An ear P_i which is directed from $L(P_i)$ to $R(P_i)$ (resp. from $R(P_i)$ to $L(P_i)$) is called an *outgoing* (resp. *incoming*) ear of $L(P_i)$.

We give two rules which imply a unique numbering of the vertices of *G*. These rules imply actually a serial algorithm which provides this numbering. We refer often in our presentation to it as the serial numbering algorithm. Parallel implementation of both stages is discussed in the next subsection.

Rule 1 relates to the relative numbering of the internal vertices of an ear.

Definition: Let P_i be an ear, and let u, v be internal vertices of P_i. We say that *u precedes v* if, when moving on P_i according to its direction, we encounter *u* before encountering *v*.

Rule 1. Let P_i be an ear, and let u, v be internal vertices such that *u* precedes *v*. The *st*-numbering of *u* and each of its descendants is smaller than the *st*-numbering of *v* and each of its descendants.

Rule 1 guarantees that for each ear P_i, $i \geq 1$, all its internal vertices, excluding $LS(P_i)$ and $RS(P_i)$, are assigned a valid *st*-numbering. This holds, since each of these vertices will have a neighbor in P_i whose number is smaller and a neighbor in P_i whose number is greater than its own number. Hence, the difficulty in providing *st*-numbering is reduced to assigning a valid *st*-numbering to $LS(P_i)$ and $RS(P_i)$, for each ear P_i. Note however, that if P_i is an incoming (resp. outgoing) ear with more than one internal vertex, then Rule 1 guarantees that $LS(P_i)$ will have a neighbor whose number is smaller (resp. greater) than its own number, and $RS(P_i)$ will have a neighbor whose number is greater (resp. smaller) than its own number.

Rule 2 relates to the relative numbering of ears which have the same anchor.

Definition: Let *v* be an internal vertex of some ear. We define a total order on the incoming and outgoing ears of *v* as follows:

(i) An incoming ear *I* of *v* is $before$ another incoming ear I' of *v* iff the edge $(v, LS(I))$ is $before$ $(v, LS(I'))$ in the adjacency list of *v* in *G*. (Any other arbitrary order on the edges will do. We only need some consistent order.)

(ii) An outgoing ear *O* of *v* is $before$ another outgoing ear O' of *v* iff the edge $(v, LS(O))$ appears $after$ $(v, LS(O'))$ in the adjacency list of *v* in *G*.

(iii) Each incoming ear *I* of *v* is $before$ each outgoing ear *O* of *v*.

Remark: Note that the internal order defined on the outgoing ears is *opposite* to the internal order defined on the incoming ears. Interestingly, this subtle definition is essential for the correctness of the algorithm.

Rule 2. Let *v* be an internal vertex of some ear.

Rule 2.1. The *st*-numbering assigned to each of the vertices belonging to incoming ears of *v* is smaller than the *st*-numbering assigned to *v*.

Rule 2.2. The *st*-numbering assigned to *v* is smaller than the *st*-numbering assigned to each of the vertices belonging to outgoing ears of *v*.

Rule 2.3. If an ear *P* of *v* is before another ear P' of *v* then the *st*-numbering assigned to each of the internal vertices in *P* is smaller than the *st*-numbering assigned to each of the internal vertices in P'.

Consider now a serial implementation of the numbering which obeys Rule 2. Upon arriving at a vertex *v*, we first assign numbers to each of its incoming ears (and their descendants) in order. Then we number *v*, and finally we assign numbers to each of its outgoing ears (and their descendants) in order. (We use the order defined above.)

Rules 2.1 and 2.2 guarantee that for each incoming (resp. outgoing) ear P_i which have more than one internal vertex, $LS(P_i)$ is assigned a valid *st*-numbering; Since the number of $L(P_i)$ is greater (resp. smaller) than the number of $LS(P_i)$; Recall that Rule 1 implies that the number of the neighbor of $LS(P_i)$ in P_i is smaller (resp. greater) than the number of $LS(P_i)$. Therefore, the difficulty in providing *st*-numbering is further reduced to assigning valid *st*-numbering to $RS(P_i)$, for each ear P_i. This will be the task of the orientation algorithm (which will make use of Rule 2.3).

We already mentioned that Rules 1 and 2 define a unique numbering of the vertices of the oriented graph. This serial numbering algorithm starts at ear P_0, which is directed from *s* to *t*, and arrives to each ear through the ear-tree *ET*. An ear is always "entered" through its anchor.

Fig. 3.1(a) gives an example of a graph with directions on the ears. Specification of $R(.)$ for all ears is omitted. Fig. 3.1(b) demonstrates the result of an application of the numbering process.

The orientation of the ears (Stage 1).

We should remember that the orientation of the ears (Stage 1), which is described below, is aimed at the numbering algorithm (Stage 2), which was outlined above. As was mentioned before, the main task of the orientation algorithm is to guarantee that, for each ear P_i, the number which will be assigned to $R(P_i)$ is *consistent* with the number which will be assigned to $RS(P_i)$. Specifically, this means that: (i) If P_i gets a direction from $L(P_i)$ to $R(P_i)$ then the task is to take care that the number of $R(P_i)$ will be greater than the number of $RS(P_i)$. (ii) If P_i gets a direction from $R(P_i)$ to $L(P_i)$ then the task is to take care that the number of $R(P_i)$ will be smaller than the number of $RS(P_i)$.

The relation between the numbers assigned to $R(P_i)$ and $RS(P_i)$ is determined by the following event: Will the serial numbering algorithm visit the (internal vertices of the) ear P_i before or after visiting the vertex $R(P_i)$? Our crucial observation is that we can make this event depend only on the direction which will be given to some *single* ear P_m. As will become clear later, P_m is always closer than P_i to P_0 in the ear-tree. (P_m is not necessarily an ancestor of P_i.) This ear P_m will be called the *hinge* of P_i. (Observe that *m* is a function of *i*, $m = m(i)$.)

Definition: We say that the ear P_i gets the *same* direction as its hinge P_m, if our orientation algorithm implies that both are directed either from $L(.)$ to $R(.)$ or from $R(.)$ to $L(.)$. Similarly, the ear P_i gets a direction *opposite* to P_m, if the algorithm implies that one of them is directed from $L(.)$ to $R(.)$ iff the other is directed from $R(.)$ to $L(.)$.

Below we describe how the hinge P_m is found, and how to determine whether P_i will have the same or the opposite direction as P_m. Since this description is not short we show first Step 2 (the last step) of Stage 1. That is, how to determine the orientation of each ear, once we know for each ear its hinge and its direction with respect to the hinge.

We define the *hinge−tree* $HT(V_{HT}, E_{HT})$ as follows:

$V_{HT} = \{P_i \mid P_i \text{ is an ear of } G\}$. That is, every ear is a vertex in *HT*.

$E_{HT} = \{(P_i, P_m) \mid P_m$ is the hinge of $P_i\}$

Fact: *HT* is a directed tree rooted at P_0.

Step 2. Let P_0 be directed from $L(P_0)$ to $R(P_0)$ (i.e., from s to t). For each $i \geq 1$, consider the ear P_i and the path from P_i to P_0 in *HT*. Count the number of ears on this path which get a direction opposite to their hinge. If this number is even then P_i is directed from $L(P_i)$ to $R(P_i)$; otherwise, it is directed from $R(P_i)$ to $L(P_i)$.

Step 1. Finding the hinge $\mathbf{P}_m$ of ear $\mathbf{P}_i$.

Definition: We say that a vertex $v \in P_i$ appears to the *left* of $u \in P_i$ on P_i if v appears prior to u when moving on P_i from $L(P_i)$ to $R(P_i)$.

For finding the hinge of P_i, we consider several cases.

Case 1. The vertices $R(P_i)$ and $L(P_i)$ are both on the same ear P_j, see Fig. 3.2.

Instructions: In this case, the ear P_j is set to be the hinge of P_i, and P_i gets the same direction as P_j iff $L(P_i)$ appears to the left of $R(P_i)$ in P_j.

Discussion: Notice that we use the fact that the given ear-decomposition of G is open, therefore, $R(P_i) \neq L(P_i)$.

Since this is the first case we consider, we elaborate on it slightly. In later cases, the justification of the choice of the hinge and of the relation between the orientations of an ear and its hinge will be shorter.

In Fig. 3.2(a), P_i gets the same direction as P_j. On the other hand, in Fig. 3.2(b) it gets a direction opposite to P_j. To justify this orientation, recall the numbering algorithm. Assume that the situation is as in Fig. 3.2(a). The ear P_i will be directed from $L(P_i)$ to $R(P_i)$ iff P_j will be directed from $L(P_j)$ to $R(P_j)$. Now if both conditions hold, the number assigned to $RS(P_i)$ will be smaller than that of $R(P_i)$, and if none holds then the number assigned to $RS(P_i)$ will be greater than that of $R(P_i)$. This yields a consistent numbering for $RS(P_i)$. Similarly, assume that the situation is as in Fig. 3.2(b). The ear P_i will be directed from $L(P_i)$ to $R(P_i)$ iff P_j will be directed from $R(P_j)$ to $L(P_j)$. If both conditions hold, the number assigned to $RS(P_i)$ will be smaller than that of $R(P_i)$, and if none holds then the number assigned to $RS(P_i)$ will be greater than that of $R(P_i)$. This will provide, again, a consistent numbering for $RS(P_i)$.

Case 2. The vertices $L(P_i)$ and $R(P_i)$ belong to different ears.

Assume that $L(P_i)$ is in P_j and $R(P_i)$ is in P_k, and let the lowest common ancestor of P_j and P_k in the ear-tree be P_α.

Case 2.1. P_α is neither P_j nor P_k, see Fig. 3.3.

Let v_j (resp. v_k) be the anchor-ancestor of P_j (resp. P_k) in P_α.

Case 2.1.1. $v_j \neq v_k$, see Fig. 3.3(a).

Instructions: (Similar to Case 1.) The ear P_α is set to be the hinge of P_i, and P_i gets the same direction as P_α iff v_j appears to the left of v_k in P_α.

Establishing the validity of the numbering of $RS(P_i)$ in this case is similar to Case 1.

Case 2.1.2. $v_j = v_k$, see Fig. 3.3(b).

Instructions: Consider the first edge $(P_\alpha, P_{j'})$ on the path in *ET* from P_α to P_j. This edge corresponds to an edge of *G* such that one of its endpoints is v_j. Denote this edge by e_j. In the same way, $P_{k'}$ and e_k are defined. Since $v_j = v_k = v$, both edges appear in the adjacency list of *v* in *G*. If e_j appears before e_k in this list, then the hinge of P_i is set to be $P_{j'}$ and P_i gets a direction opposite to $P_{j'}$. If e_k appears before e_j, then the hinge of P_i is set to be $P_{k'}$ and P_i gets the same direction as $P_{k'}$.

Discussion: The following claim shows that also in this case the vertex $RS(P_i)$ is assigned a valid *st*-numbering.

Claim 1. The ear P_i is directed from $L(P_i)$ to $R(P_i)$ (i.e., P_i is an outgoing ear) iff the number assigned to $RS(P_i)$ is smaller than the number assigned to $R(P_i)$.

To prove Claim 1 we need the total order defined on the incoming and outgoing ears of a vertex *v*. This is since this order determines the order of numbering the vertices in the serial numbering algorithm (by Rule 2.3.). The crucial point is that Claim 1 holds *regardless* of the directions which will actually be given to $P_{j'}$ and $P_{k'}$.

Proof of Claim 1. We have two possibilities to consider.

Possibility (a): The edge e_j appears before the edge e_k in the adjacency list of *v*.

In Possibility (a), $P_{j'}$ and each of its descendants are numbered before $P_{k'}$ and each of its descendants iff $P_{j'}$ is an incoming ear (regardless of the direction of $P_{k'}$). But, the instructions of Case 2.1.2. imply that $P_{j'}$ is an incoming ear iff P_i is an outgoing ear. Recall now that $RS(P_i)$ is a descendant of $P_{j'}$, and $R(P_i)$ is a descendant of $P_{k'}$. Combining the above together we obtain that P_i is an outgoing ear iff the number assigned to $RS(P_i)$ is smaller than the number assigned to $R(P_i)$, as required.

Possibility (b): The edge e_k appears before the edge e_j in the adjacency list of *v*.

In Possibility (b), $P_{j'}$ and each of its descendants are numbered before $P_{k'}$ and each of its descendants iff $P_{k'}$ is an outgoing ear (regardless of the direction of $P_{j'}$). But, the instructions of Case 2.1.2. imply that $P_{k'}$ is an outgoing ear iff P_i is an outgoing ear. Recall, again, that $RS(P_i)$ is a descendant of $P_{j'}$, and $R(P_i)$ is a descendant of $P_{k'}$. Combining the above together we obtain that P_i is an outgoing ear iff the number assigned to $RS(P_i)$ is smaller than the number assigned to $R(P_i)$, completing the proof of the claim.

Case 2.2. P_α is P_k, see Fig. 3.4. (Note that P_α can not be P_j as this implies that $j<k$, while the definition of $L(P_i)$ and $R(P_i)$ implies that $j \geq k$.)

Let *v* be the anchor-ancestor of P_i in P_α. Using the same arguments as above, we get the following.

Case 2.2.1. $v \neq R(P_i)$, see Fig. 3.4(a).

Instructions: In this case the hinge of P_i is set to be P_α, and P_i gets the same direction as P_α iff v appears to the left of $R(P_i)$ in P_α.

Establishing the validity of the numbering of $RS(P_i)$ in this case is similar to Case 1.

Case 2.2.2. $v = R(P_i)$, see Fig. 3.4(b).

Instructions: Let $P_{j'}$ be the first ear on the path in ET from P_α to P_j. The hinge of P_i is set to be $P_{j'}$, and P_i gets a direction opposite to $P_{j'}$.

Discussion: The ear P_i will be directed from $L(P_i)$ to $R(P_i)$ iff $P_{j'}$ will be directed from $R(P_{j'})$ to $L(P_{j'})$. If both conditions hold, the number assigned to $RS(P_i)$ will be smaller than that of $R(P_i)(= L(P_{j'}))$, and if none holds then the number assigned to $RS(P_i)$ will be greater than that of $R(P_i)(= L(P_{j'}))$. This will provide a consistent numbering for $RS(P_i)$.

This completes Step 1 of Stage 1.

The above presentation shows that all the vertices, excluding s and t, will be given a valid *st*-numbering. In order to establish correctness of the algorithm it remains to show that s and t are also assigned a valid *st*-numbering.

Claim 2. The algorithm assigns the number 1 to the vertex s, and the number n to the vertex t.

Proof of Claim 2. Recall that P_0 is directed from s to t. Consider an ear P_i which is a son of P_0 in the ear-tree. The definition of $L(P_i)$ and $R(P_i)$ implies that both $L(P_i)$ and $R(P_i)$ must belong to P_0. (This definition states that if $L(P_i)$ is in P_j and $R(P_i)$ is in P_k then $j \geq k$.) This means that all the ears P_i such that $L(P_i) = s$ (resp. $L(P_i) = t$) must satisfy $R(P_i) = t$ (resp. $R(P_i) = s$). Thus, by Case 1 of the orientation algorithm, for all P_i such that $L(P_i) = s$, P_i is an outgoing ear, and for all P_i such that $L(P_i) = t$, P_i is an incoming ear. Hence, from Rules 2.1 and 2.2, it follows that 1 is assigned to s, and n is assigned to t.

3.2. Parallel Implementation

We describe a parallel implementation of the two stages of the *st*-numbering algorithm.

Stage 1. In Stage 1 we start by constructing the ear tree ET. Let l be the number of ears that have at least one internal vertex. Observe the $l \leq n$. Finding the edges of ET from the output of the open ear decomposition algorithm can be done in $O(1)$ time using n processors. Computing the adjacency lists of ET needs $O(\log n)$ time with n processors using integer sorting algorithms, e.g. [AKS-83]. References to other possible sorting algorithms may be found in the remark at the end of Section 2. Finding the hinges requires l computations of lowest common ancestor in ET. This can be done in $O(\log l)$ time using l processors, see [V-85]. The other computations of Stage 1, including the computation of the direction of the ears based on the hinge tree, can also be done in $O(\log l)$ time using l processors. Therefore, Stage 1 takes $O(\log n)$ time using n processors.

Stage 2. We now describe how to implement the numbering rules in parallel. We construct a tree, whose preorder numbering obeys the numbering rules. This *numbering – tree NT* is defined as follows. Let v be a vertex of G. The vertex v induces two vertices v_{in} and v_{out} in NT. Also, v_{in} and v_{out} are both connected by an edge. The rest of the edges of NT are defined by the following rules:

(1) Assume that v is in P_i and let u be the neighbor of v in P_i which precedes v (if such a vertex exists). Then u_{out} is connected to v_{in}.

(2) For each incoming ear P_i whose anchor is v the vertex v_{in} is connected to w_{in}, where $w = RS(P_i)$.

For each outgoing ear P_j whose anchor is v the vertex v_{out} is connected to w_{in}, where $w = LS(P_i)$.

Fig. 3.5 illustrates the construction of NT for an example of a part of an ear-tree.

We now perform a preorder traversal of NT starting at the root s_{in}. We observe that the order in which vertices of the form v_{out} are visited during the traversal obeys the rules of the numbering algorithm. That is, for each v in G we first visit v_{in} and all the vertices of incoming ears whose anchor is v and their descendants. Then we visit v_{out}. Finally, we visit all the vertices of outgoing ears whose anchor is v and their descendants. The order in which the incoming (and outgoing) ears of v are visited is as defined in Rule 2. Note that for each vertex v in G, its vertices in NT are traversed in the order: first v_{in}, and then v_{out}.

During the traversal we assign preorder numbers to the vertices of the form v_{out}, ignoring all the vertices of the form v_{in}. These numbers provide the *st*-numbering of the graph G.

The construction of NT can be done in $O(\log n)$ time using n processors. The preorder numbering can be done using the Euler tour technique for computing tree functions given in [V-85] in $O(\log n)$ time using n processors.

To sum up, given a biconnected graph G, we present a parallel algorithm which finds an *st*-numbering of G. The first part of the algorithm finds an open ear-decomposition of G. This is done either in $O(\log n)$ time using $m + n$ processors, or in $O(\log n \log^{(2)} n \log^{(3)} n)$ time using $((n \log n + m)/\log n \log^{(2)} n \log^{(3)} n)$ processors. The second part finds an *st*-numbering of G in $O(\log n)$ time using n processors. Therefore, the whole *st*-numbering algorithm takes either $O(\log n)$ time using $n+m$ processors or $O(\log n \log^{(2)} n \log^{(3)} n)$ time using $((n \log n + m)/\log n \log^{(2)} n \log^{(3)} n)$ processors.

REFERENCES

[A-85] Anderson, R., ‘‘A Parallel Algorithm for the Maximal Path Problem’’, Proc. 17^{th} ACM Symp. on Theory of Computing (1985), pp. 33-37.

[AKS-83] Ajtai, M., Komlos, J. and Szemeredi, E., ‘‘An $O(n \log n)$ Sorting Network’’, Combinatorica 3 (1983), pp. 1-19.

[CV-86] Cole, R. and Vishkin, U., ‘‘Deterministic Coin Tossing and Accelerating Cascades: Micro and Macro Techniques for Designing Parallel Algorithms’’, Proc. 18^{th} ACM Symp. on Theory of Computing (1986), to appear.

[E-79] Even, S., ‘‘Graph Algorithms’’, Computer Science Press, 1979.

[ET-76] Even, S. and Tarjan, R.E., ‘‘Computing an *st*-numbering’’, Th. Comp. Science 2 (1976), pp. 339-344.

[FL-80] Fisher, M. and Ladner, L., ‘‘Parallel Prefix Computation’’, JACM 27 (1980), pp. 831-838.

[IR-84] Itai, A. and Rodeh, M., ‘‘The Multi-Tree Approach to Reliability in Distributed Networks’’, Proc. 25^{th} Symp. on Foundations of Comp. Science (1984), pp. 137-147.

[JS-82] Ja’Ja’, J. and Simon, J., ‘‘Parallel Algorithms in Graph Theory: Planarity Testing’’, SIAM J. Comput. 11 (1982), pp. 314-328.

[L-85] Lovasz, L., ‘‘Computing Ears and Branchings in Parallel’’, Proc. 26^{th} Symp. on Foundations of Comp. Science (1985), pp. 464-467.

[LEC-67] Lempel, A., Even, S. and Cederbaum, I., ‘‘An Algorithm for Planarity Testing of Graphs’’, Theory of Graphs, Int. Symp., Rome, July 1966. P. Rosenstiehl, Ed., Gordon and Breach, NY (1967), pp. 215-232.

[R-85] Reif, J.H., ‘‘An Optimal Parallel Algorithm for Integer Sorting’’, Proc. 26^{th} Symp. on Foundations of Comp. Science (1985), pp. 496-503.

[SV-82] Shiloach, Y. and Vishkin, U., ‘‘An $O(\log n)$ Parallel Connectivity Algorithm’’, J. of Algorithms 3 (1982), pp. 57-63.

[T-83] Thompson, C.D., ‘‘The VLSI Complexity of Sorting’’, IEEE Trans. Computers 32 (1983), pp. 1171-1184.

[TV-85] Tarjan, R.E. and Vishkin, U., ‘‘An Efficient Parallel Biconnectivity Algorithm’’, SIAM J. Comput. 14 (1985), pp. 862-874.

[V-83] Vishkin, U., ‘‘Synchronous Parallel Computation - a Survey’’, TR #71, Dept. of Computer Science, Courant Inst., NYU, 1983.

[V-85] Vishkin, U., ‘‘On Efficient Parallel Strong Orientation’’, Information Proc. Letters 20 (1985), pp. 235-240.

[W-32] Whitney, H., ‘‘Non-separable and Planar Graphs’’, Trans. Amer. Math. Soc. 34 (1932), pp. 339-362.

APPENDIX 1. An Alternative Parallel Implementation of the Open Ear-Decomposition Algorithm.

Observe that the total number of operations in the parallel implementation of Section 2 is $O(n \log n + m \log n)$, while the number of operations in the respective serial algorithm is $O(n + m)$. In the following, we present an $O(\log n \log^{(2)} n \log^{(3)} n)$ time implementation in which the total number of operations is only $O(n \log n + m)$. This is done using the new algorithms for connectivity and linked-list ranking of [CV-86]. Notice that Steps 1,4,5 and 6 of Section 2 are the only steps where $O(m \log n)$ operations are performed. (Recall the Remark concerning Steps 2 and 3 in Subsection 2.3.)

Step 1. Step 1 can be implemented in $O(\log n \log^{(2)} n \log^{(3)} n)$ time with $((n \log n + m)/\log n \log^{(2)} n \log^{(3)} n)$ processors using the connectivity algorithm of [CV-86].

Step 4. Recall that in Step 4 we construct the auxiliary graphs. Note that for doing this we have to compute $LCA(u,v)$, $e_{u,v}$ and $e_{v,u}$ for each non-tree edge (u,v). Specifically, for each non-tree edge (u,v) it takes $O(\log n)$ time using one processor to compute $LCA(u,v)$. This implies a total of $O(m \log n)$ operations. We want to implement this step in only $O(n \log n + m)$ operations. We modify the auxiliary graphs of Subsection 2.2. This enables dispensing with explicit LCA computations for most of the non-tree edges in constructing the auxiliary graphs.

For every vertex x in $V - \{t\}$, we construct the bipartite undirected graph $H_x = (V_x, E_x)$ in the following way.

The vertices in V_x are of three types:

(1) All tree edges connecting x with a son in the tree (the *tree vertices*).

(2) (Let u be a vertex in G. We define $Ad-LCA(u)$ to be the lowest common ancestor in T of $(u, u_1, \ldots, u_l)$, where $u_1, \ldots, u_l$ are all the vertices which are adjacent to u by an edge in $E - E_T$.) All non-tree edges (u,v) such that $LCA(u,v) = x$ and x is also either $Ad-LCA(u)$ or $Ad-LCA(v)$. (These are the *non-tree vertices*.)

(3) A new vertex $\overline{x}$. (We call it the $hyper-vertex$.)

Let $[u,v]$ be a non-tree vertex in V_x such that u is not an ancestor of v in T. If x is $Ad-LCA(u)$ then the edge $([u,v],[w,x])$ (where $[w,x]$ is $e_{u,v}$) is in E_x. Otherwise, (i.e., when $x \neq Ad-LCA(u)$) the edge $([u,v],\overline{x})$ is in E_x. (Note that in this case $Ad-LCA(v)$ is a descendant of $Ad-LCA(u)$.)

For these modified auxiliary graphs we have the following Lemma.

Modified Main Lemma. Consider a spanning tree of a connected component of one of the auxiliary graphs H_x. Then either it contains $\overline{x}$ or at least one of its tree vertices $[w,x]$ (where $x = F(w)$) must satisfy $LEVEL(LCA(MASTER(w,x))) < LEVEL(x)$.

Notice that if the edge $([u,v],\overline{x})$ is in E_x then the ear of (u,v) must be open.

We describe how to construct these auxiliary graphs in parallel. First we compute $Ad-LCA(v)$ for each vertex v. This is done as in the implementation of Step 2 (given in [V-85]) in $O(\log n)$ using n

processors. For each vertex v, we also compute $Sub-Ad-LCA(v)$ which is the first vertex on the path from $Ad-LCA(v)$ to v. This also can be done in $O(\log\ n)$ time using n processors.

For a given non-tree edge (u,v), if $[u,v]$ is a vertex of an auxiliary graph H_x then x has to be either $Ad-LCA(u)$ or $Ad-LCA(v)$. This gives at most two auxiliary graphs in which $[u,v]$ can be a vertex. It takes $O(1)$ time using a single processor to identify these two auxiliary graphs.

Suppose $x = Ad-LCA(u)$. The vertex $[u,v]$ is in H_x only if $x = LCA(u,v)$. We find out whether $x = LCA(u,v)$ by checking whether v is a descendant of $Sub-Ad-LCA(u)$. [V-85] shows that given two vertices u and v in G, it is possible to determine whether v is a descendant of u in T in $O(1)$ time using one processor. (This is done using the preorder and postorder numbering of the vertices, which were computed in the implementation of Step 2 in [V-85].) Thus, we can determine whether $[u,v]$ is in H_x in $O(1)$ time using one processor. Suppose $x = Ad-LCA(u)$ and $[u,v]$ is a vertex in H_x. We show how to find the the adjacent vertices of $[u,v]$ in H_x. Recall that, by the definition of the auxiliary graphs, $[u,v]$ is connected to the tree vertex $[Sub-Ad-LCA(u),x]$. Suppose further that v is not an ancestor of u (i.e., $x \neq v$). If $x = Ad-LCA(v)$ then $[u,v]$ is connected by an edge to the tree vertex $[Sub-Ad-LCA(v),x]$, otherwise $[u,v]$ is connected to $\bar{x}$. So, finding the adjacent vertices of $[u,v]$ in H_x takes constant time using one processor.

The case $x = Ad-LCA(v)$ is treated similarly.

The parallel connectivity algorithm of [CV-86] needs an adjacency list representation of the auxiliary graphs H_x. We show how to obtain such representation. Note that naive sorting of the edges can not be applied within the complexity bounds that we seek.

The adjacency lists of the non-tree vertices can be obtained, as described above, in constant time using one processor.

We show how to compute the adjacency lists of the tree vertices. Let v be a vertex in G and let x be $Ad-LCA(v)$. Recall that a non-tree edge (u,v) is a non-tree vertex in H_x if and only if $x = LCA(u,v)$. Moreover, if $[u,v]$ is a non-tree vertex in H_x then it is connected by an edge to the tree vertex $[Sub-Ad-LCA(v),x]$. So, for each vertex v in G we consider all edges (u,v) such that $LCA(u,v) = Ad-LCA(v)$ as one block and apply a sorting algorithm to these blocks for obtaining the adjacency lists of the tree vertices. This sorting takes $O(\log\ n)$ time using n processors since we have only n blocks. Thinning out these blocks to include only relevant edges is easy and can be done by a parallel prefix sum computation (given the adjacency list representation of G) in $O(\log\ n)$ time using $m/\log n$ processors.

The computation of the adjacency list of the hyper-vertices is similar. Let v be a vertex in G and let x be $Ad-LCA(v)$. Recall that a non-tree edge (u,v) is connected by an edge to $\bar{x}$ in H_x if and only if $x = LCA(u,v) \neq Ad-LCA(u)$. So, for each vertex v in G we consider all edges (u,v) such that $LCA(u,v) = Ad-LCA(v) \neq Ad-LCA(u)$ as one block and proceed as for the tree vertices.

Step 5. In Step 5 we compute the connected components and a spanning forest of each auxiliary graph. Note that the total number of vertices in these graphs is $O(m)$. Thus, it seems that $O(m)$ operations will not be enough. However, we can perform only $O(m)$ operations using the following Fact.

Fact. The degree of each non-tree vertex in the auxiliary graphs is at most two.

Thus, we may shrink each non-tree vertex $[u,v]$ of degree two and the two edges emanating from it into a new edge connecting the two neighbors of $[u,v]$. The resulting graphs have $O(n)$ vertices and $O(m)$ edges, and hence we may apply the connectivity algorithm of [CV-86] to get the same results as in Step 1. Notice that the connected components and the spanning forest of the original graphs can be recovered from the the connected components and the spanning forest of the shrunken graphs in constant time using $O(m)$ processors.

Step 6. This step can be implemented using the Euler tour technique applying the new optimal ranking algorithm of [CV-86] in time $O(\log n \ \log^{(2)} n)$ using $m/\log n \ \log^{(2)} n$ processors.

In summary, we have shown an implementation which runs in $O(\log n \ \log^{(2)} n \ \log^{(3)} n)$ time with $((n \ \log n + m)/\log n \ \log^{(2)} n \ \log^{(3)} n)$ processors.