# TOWARDS OVERCOMING THE TRANSITIVE-CLOSURE BOTTLENECK: EFFICIENT PARALLEL ALGORITHMS FOR PLANAR DIGRAPHS

(EXTENDED ABSTRACT)

MING-YANG KAO
DEPARTMENT OF COMPUTER SCIENCE
DUKE UNIVERSITY
DURHAM, NORTH CAROLINA 27706

PHILIP N. KLEIN
DEPARTMENT OF COMPUTER SCIENCE
BROWN UNIVERSITY
PROVIDENCE, RHODE ISLAND 02912

**Abstract.** Currently, there is a significant gap between the best sequential and parallel complexities of many fundamental problems related to digraph reachability. This complexity bottleneck essentially reflects a seemingly unavoidable reliance on transitive closure techniques in parallel algorithms for digraph reachability. To pinpoint the nature of the bottleneck, we develop a collection of polylog-time reductions among reachability problems. These reductions use only linear processors and work for general graphs. Furthermore, for planar digraphs, we give polylog-time algorithms for the following problems: (1) directed ear decomposition, (2) topological ordering, (3) digraph reachability, (4) descendent counting, and (5) depth-first search. These algorithms use only linear processors and therefore reduce the complexity to within a polylog factor of optimal.

**1. Introduction.** In its simpliest form, the *digraph reachability* problem is to test whether a graph contains a directed path from a given vertex to another. The best sequential algorithms for the problem use simple graph searches and run in optimal linear time [3]. In contrast, all known polylog-time parallel algorithms for the problem compute the transitive closure of the given graph; the best of them currently uses $O(n^{2.376})$ processors for an $n$-vertex graph [21], [12], [7]. Thus, there is a significant gap between the best sequential and parallel complexities of the problem. The reliance on transitive closure techniques and the associated complexity bottleneck are not limited to the digraph reachability problem [21]. Indeed, they are also shared by many fundamental problems related to digraph reachability such as strongly connected components and directed spanning trees. Motivated by the fundamental nature of digraph reachability, a substantial amount of research work has been directed towards overcoming the seemingly unavoidable reliance on transitive closure. The ultimate goal is to design polylog-time algorithms that use only linear processors and therefore reduce the complexity to within a polylog factor of optimal.

The first two breakthroughs have come only very recently. For *planar* digraphs, Kao has shown that the strongly connected components can be found in $O(\log^3 n)$ time using $O(n)$ processors [18], [20]. For planar digraphs that are *strongly connected*, Kao and Shannon together have proved that a directed spanning tree can be computed in $O(\log^2 n)$ time using $O(n)$ processors [19], [20].

In this paper, we report further breakthroughs. We show that for planar digraphs the following five fundamental problems can be solved all in polylog time using a linear number of processors: (1) directed ear decomposition, (2) topological ordering, (3) digraph reachability, (4) descendent counting, and (5) depth-first search. Previously known NC algorithms for these problems all require far more than linear processors, even for planar digraphs. Among the five problems, depth-first search is the most difficult. In fact, depth-first search for general digraphs is not even known to be in *deterministic* NC [2], and the planar case has only very recently been shown by Kao to have a deterministic NC algorithm using $O(n^4)$ processors [17].

In addition to devising these linear-processor NC algorithms, we also take the first step in developing a general theory for understanding the nature of the transitive-closure bottleneck. We discover a collection of NC *reductions* among reachability-related problems that use only linear processors and do *not* depend on planarity at all. Our goal is to eventually establish a broad complexity hierarchy among reachability-related problems via linear-processor NC reductions, and then use this hierarchy to guide further research efforts towards overcoming the transitive-closure bottleneck.

We further outline our results as follows. A *directed ear decomposition* of a digraph is a partition of the edges into internally vertex-simple directed paths $P_1, \cdots, P_k$ such that (1) the two endpoints of $P_1$ are the same vertex and (2) the endpoints of each $P_i \neq P_1$ lie in some lower-indexed $P_j$'s but the internal vertices of $P_i$ are not in any lower-indexed $P_j$'s. These $P_i$'s are called *ears*. By simple induction, a directed ear decomposition exists if and only if the graph is strongly connected. The undirected version of ear decomposition has proved tremendously useful in parallel algorithms. For instance, it is the basis of efficient algorithms for *st*-numbering [26], triconnectivity [27], [34], [10], [9], 4-connectivity [16], and planarity [22], [33]. We expect

that directed ear decomposition will also be very useful. Indeed, it has been used by Kao for computing planar directed cycle separators [17] and by us for finding a topological ordering in the present paper. Lovász has previously given a directed ear decomposition algorithm for general digraphs that relies on breadth-first search and has a high complexity [25]. We give a parallel reduction for obtaining a directed ear decomposition from an arbitrary pair of directed spanning trees, one convergent, one divergent, and both rooted at the same vertex. This reduction works for general graphs and has an optimal complexity of $O(\log n)$ time using $O((n+e)/\log n)$ processors for a graph with $n$ vertices and $e$ edges. For strongly connected planar digraphs, because such a pair of spanning trees can be found in $O(\log^2 n)$ time using $O(n)$ processors [19], [20], the reduction implies an algorithm for computing a directed ear decomposition in $O(\log^2 n)$ time using $O(n)$ processors.

A *topological ordering* of a digraph is a linear ordering of its vertices such that every edge in the graph points from a lower-indexed vertex to a higher-indexed vertex in the linear ordering. A topological ordering exists if and only if the graph is a dag. In this paper, we give an algorithm that computes a topological ordering for an $n$-vertex planar dag in $O(\log^2 n)$ time using $O(n)$ processors. A key fact used in our algorithm is that the dual of a planar dag is a strongly connected digraph. This fact has played a crucial role in the linear-processor NC algorithms for planar strongly connected components and planar directed spanning trees [18], [19], [20]. Based on this fact, to find a topological ordering of a planar dag, we exploit the structure in the dual of the input graph. Because the input graph is acyclic, the dual graph is strongly connected and therefore has a directed ear decomposition. Such an ear decomposition is used to cut the plane into small regions with a useful boundary orientation. Correspondingly, the input graph is partitioned into small pieces with a useful ordering property. This partition induces an natural divide-and-conquer strategy and allows us to compute a topological ordering of the input graph by recursing on the small pieces in parallel.

The *digraph reachability* problem can be formulated in several different ways. For brevity, this outline focuses on the following two versions: the *multiple-source* (or *single-source*) reachability problem is to find all the vertices reachable through directed paths from a given set of vertices (respectively, a given single vertex). The single-source version is clearly a special case of the multiple-source version. For general graphs, these two versions are in effect identical because the multiple-source version can be reduced to the single-source version by merging the sources into a super-source. However, for planar graphs, such a reduction may destroy the planarity if the sources are not connected. This subtlety has also been a fundamental constraint in many other algorithms for planar graphs including the recent network flow algorithm by Miller and Naor [30]. For the purpose of future study, we address this fundamental issue in the context of *minor-closed* families; a family of graphs is called *minor-closed* if it is closed under deletions and contractions of edges [13]. We show that for a minor-closed family, the reachability problem can be transformed via linear-processor NC reductions to the problems of computing strongly connected components and topological ordering. It is a classic theorem that the family of planar graphs is minor-closed [13]. Consequently, for planar digraphs, this transformation implies that the digraph reachability problem is solvable in polylog time and linear processors using the strongly connected component algorithm by Kao [18] and the topological ordering algorithm of this paper. The best previously known NC algorithms for the multiple-source and single-source reachability problems require $O(n^2)$ processors and $O(n^{1.5})$ processors, respectively. These complexity bounds are obtained using the path algebra algorithms by Pan and Reif [31] in conjunction with the randomized planar undirected separator algorithm by Gazit and Miller [11]. As for smaller classes of planar digraphs, there have been several optimal NC algorithms for the digraph reachability problem all based on properties unique to the class in question; in particular, for planar $st$-digraphs, Vitter and Tamassia have given optimal algorithms that solve the digraph reachability problem as well as other problems [39].

The *descendent counting* problem is to compute for each vertex the number of vertices that can be reached from the vertex through directed paths. More generally, the problem is to sum the weights of the descendents of each vertex using commutative semigroup operations. This fundamental problem appears as a subproblem in many digraphs problems. In particular, it plays an important role in the parallel depth-first search algorithms for planar and general digraphs [17], [2]. In this paper, we show that for a rooted planar digraph, the descendent counting problem can be solved in polylog time using linear processors. The algorithm builds upon the planar reachability algorithms and employs separator-based accounting arguments. The algorithm is an essential component of our linear-processor NC algorithm for planar directed depth-first search.

The *depth-first search* problem is to construct a forest that corresponds to performing depth-first search in a given graph starting from specified vertices [38], [3]. For lexicographic depth-first search, Reif shows that the problem is P-complete even for general undirected graphs [35]. For unordered depth-first search, Smith

gave the first deterministic NC algorithm for planar undirected graphs [37]; the processor complexity for this case was later shown to be linear by Ja'Ja and Kosaraju [15] and by He and Yesha [14]. Aggarwal and Anderson give the first randomized NC algorithm for general undirected graphs [1]. Kao gives the first deterministic NC algorithm for planar directed graphs [17]. Aggarwal, Anderson, and Kao give the first randomized NC algorithm for general directed graphs [2]. While these results have placed depth-first search in NC, they all have very high complexity except those for planar undirected graphs. Thus, it remains an open problem to find truly efficient parallel algorithms for depth-first search. For planar digraphs, this fundamental open problem has in part motivated the study of strongly connected components [18], directed spanning trees [19], and all four problems discussed above. With this paper, we move one step closer to the final goal. We give a deterministic NC algorithm for planar directed depth-first search that uses only linear processors and thus achieves a complexity to within a polylog factor of optimal. As expected, this result uses all linear-processor NC algorithms for planar digraphs highlighted above.

The above discussion has outlined the key results of this paper. The following sections proceed to provide the details. In section 2, we review basics of planar graphs. In sections 3 through 7, we discuss directed ear decomposition, topological ordering, digraph reachability, descendent counting, and depth-first search, respectively.

**2. Basics for planar graphs.** A *planar* digraph is one that can be drawn on a plane such that the edges in the drawing intersect only at common ends [13], [5]. A drawing of a planar digraph can be specified by the clockwise cyclic order of edges incident with each vertex. Such a specification is called a *combinatorial embedding* and is useful for algorithmic purposes. Klein and Reif give the first linear-processor NC algorithm for finding a combinatorial embedding [22]. Ramachandran and Reif have very recently reduced the complexity of finding an embedding to optimal $O(\log n)$ time using $O(n/\log n)$ processors for an $n$-vertex graph [33]. In the following discussion, we first review the definitions of faces, orientations, and duals for planar digraphs, and then quote related theorems from previous work.

Let $G$ be a connected embedded planar digraph. If the edges and vertices of $G$ are deleted from the embedding plane of $G$, then the plane is divided into disconnected regions. Each region is called a *face* of $G$. The *boundary* of a face $f$ is the set of edges and vertices surrounding $f$. The *orientation* of a boundary edge with respect to $f$ is determined by an observer staying inside $f$ and walking along the boundary of $f$. The dual of $G$, denoted by $\tilde{G}$, is obtained by placing a vertex in each face of $G$ and turning each edge of $G$ counterclockwise by 90 degrees.

THEOREM 2.1. [18] *Let $G$ be a connected planar digraph. Then $G$ is strongly connected if and only if $\tilde{G}$ is acyclic.*

THEOREM 2.2. [18] *Let $G$ be a planar digraph with $n$ vertices. Then the strongly connected components of $G$ can be computed in $O(\log^3 n)$ time using $O(n)$ processors on a* CRCW PRAM.

THEOREM 2.3. [18] *Let $G$ be a planar dag with $n$ vertices. Let $G'$ be the planar digraph constructed from $G$ by contracting a connected subgraph into a single vertex. Then the strongly connected components of $G'$ can be computed in $O(\log^2 n)$ time using $O(n)$ processors on a* CRCW PRAM.

THEOREM 2.4. [19] *Let $G$ be a strongly connected planar digraph with $n$ vertices. Then a directed spanning tree of $G$ rooted at a specified vertex can be computed in $O(\log^2 n)$ time using $O(n)$ processors on a* CRCW PRAM.

The next theorem is concerned with graph separators. The notion of separators has been extremely useful in many divide-and-conquer graph algorithms. For the purpose of this paper, an *undirected separator* of a graph $G$ is a set $S$ of vertices such that the largest connected component in $G - S$ contains at most $\frac{2}{3} \cdot n$ vertices. In the theorem, let $G$ be a connected planar graph with $n$ vertices; let $T$ be an undirected spanning tree rooted at $r$.

THEOREM 2.5. *There exist two vertices $x, y \in G$ such that the vertices in the two tree paths of $T$ from $r$ to $x$ and from $r$ to $y$ form an undirected separator of $G$. Furthermore, such a pair of $x$ and $y$ can be found in $O(\log n)$ time using $O(n/\log n)$ processors*

*Proof.* Lipton and Tarjan give an existence proof [24]. Miller gives a linear-processor algorithm [29]. □

**3. Directed ear decomposition.** In this section, we show that finding a directed ear decomposition is optimally NC-equivalent to finding a *CD-pair* of spanning trees as defined below. A *convergent* (or *divergent*) spanning tree of a digraph is a directed spanning tree in which every edge points from child to parent (respectively, from parent to child). A *CD-pair* of spanning trees consists of a convergent spanning tree and a divergent spanning tree both rooted at the same vertex [20], [19]. Observe that a digraph has a CD-pair of spanning trees if and only if it is strongly connected. Also recall that a digraph has an ear decomposition if and only if it is strongly connected. Consequently, a digraph has an ear decomposition if and only if it has a CD-pair of spanning trees. These facts provide the basis for the optimal NC-equivalence. The following theorem formally states the equivalence.

THEOREM 3.1. *For a strongly connected digraph with $n$ vertices and $e$ edges, given* CD*-pair of spanning trees, an ear decomposition can be computed in $O(\log n)$ time using $O((n+e)/\log n)$ processors. Conversely, given an ear decomposition, a* CD*-pair of spanning trees can be computed in the same complexity. The algorithms are deterministic and run on an* EREW PRAM.

THEOREM 3.2. *For a strongly connected planar digraph with $n$ vertices, a directed ear decomposition can be computed in $O(\log^2 n)$ time using $O(n)$ processors. The algorithm is deterministic and runs on a* CRCW PRAM.

*Proof.* By Theorems 3.1 and 2.4. □

We proceed to prove Theorem 3.1 by describing the optimal NC-equivalence. To facilitate the description, we first elaborate on the definition of a directed ear decomposition. Let $G$ be a digraph and let $r$ be a vertex in $G$. An ear *sequence* of $G$ rooted at $r$ is a sequence $P_1, \cdots, P_k$ of directed paths in $G$ such that (the *endpoint condition*) each endpoint of each $P_i$ either is $r$ or lies in a lower-indexed $P_j$. These $P_i$'s are called *ears*. Note that since $P_1$ is the lowest-indexed ear, both endpoints of $P_1$ must be $r$. Also note that ears are not necessarily simple. In fact, our discussion involves *internally* simple path and *half*-simple paths. A *half*-simple path is a directed path formed by concatenating a pair of simple paths. An *internally* simple path is a directed path in which an internal vertex appears only once in the entire path but the two endpoints may be the same vertex. An ear sequence is further called an ear *cover* of $G$ if the ears contain all vertices of $G$. Finally, an ear cover is called an ear *decomposition* of $G$ if the following three conditions are met: (1) (the *simplicity condition*) each ear is internally simple, (2) (the *intersection condition*) each ear $P_i \neq P_1$ intersects lower-indexed ears *only* at the endpoints of $P_i$, and (3) (the *partition condition*) each edge of $G$ occurs exactly once in the ear cover.

The reduction from an ear decomposition to a CD-pair is based on the following simple observations [20], [19]. A convergent spanning tree can be found by deleting the *first* edge of each ear. Symmetrically, a divergent spanning tree can be found by deleting the *last* edge of each ear. Both trees are rooted at the root of the given ear decomposition and thus form a CD-pair of spanning trees. By a straightforward implementation, this simple reduction can be done deterministically in $O(\log n)$ time using $O((n+e)/\log n)$ processors on an EREW PRAM [20], [19].

In the remainder of this section, we describe the reduction from a CD-pair to an ear decomposition. Let $G$ be a strongly connected digraph. Let $C$ and $D$ be a CD-pair of spanning trees for $G$ rooted at vertex $r$. An ear decomposition for $G$ is built from $C$ and $D$ in four stages as follows. Stage 1 decomposes $C$ and $D$ into an ear cover of $G$ such that each ear is a half-simple path formed by a tree path from $C$ and another from $D$. Stage 2 partitions each half-simple ear into smaller internally simple ears, satisfying the simplicity condition. Stage 3 further partitions each internally simple ear into even smaller ones such that the intersection condition is satisfied. Finally, to satisfy the partition condition, Stage 4 adds to the ear cover all missing edges and deletes from the cover all redundant appearances of edges. To finish the proof of Theorem 3.1, it suffices to show that each stage takes only $O(\log n)$ time using $O((n+e)/\log n)$ processors. We detail these stages and prove their complexity bounds in the next four lemmas respectively.

LEMMA 3.3 (STAGE 1). *Given $C$ and $D$, an ear cover for $G$ with half-simple ears can be computed in $O(\log n)$ time using $O(n/\log n)$ processors. Furthermore, the ear cover contains at most $2(n-1)$ edges.*

*Proof.* This stage works exclusively with the edges in $C$ and $D$. Let $x_1, \cdots, x_k$ be the leaves of $D$ in an arbitrary order. The goal is to construct an ear cover $R_1, \cdots, R_k$ for $G$ with $R_i$ corresponding to $x_i$. $R_i$ is a half-simple path formed by a directed tree path $A_i$ in $D$ that ends at $x_i$ and a directed tree path $B_i$ in $C$ that starts at $x_i$. The $A_i$'s and $B_i$'s are defined as follows. $A_i$ is the tree path between $x_i$ and the lowest ancestor $a_i$ of $x_i$ in $D$ such that $a_i$ either is $r$ or lies in some lower-indexed $A_j$. Note that a vertex is considered an ancestor of itself. Also note that $a_1$ must be $r$ because $A_1$ has the lowest index. In fact, $A_1$ is simply the tree path between the root $r$ and the leaf $x_1$. The $B_i$'s are constructed in the same way. $B_i$ is the tree path between $x_i$ and the lowest ancestor $b_i$ of $x_i$ in $C$ such that $b_i$ either is $r$ or lies in some lower-indexed $B_j$. Again $b_1$ is actually $r$ because $B_1$ has the lowest index. Notice that unlike $A_i$, $B_i$ can be a single-vertex path without any edge because the $x_i$'s are the leaves of $D$ but not necessarily leaves of $C$. We now verify that the $R_i$'s form an ear cover with half-simple ears. Because $D$ is divergent, $A_i$ ends at $x_i$; because $C$ is convergent, $B_i$ starts from $x_i$. Therefore $R_i$ is a directed path. Because $A_i$ and $B_i$ are simple paths, $R_i$ is half-simple. From the definitions of $a_i$ and $b_i$, the $R_i$'s clearly form an ear sequence of $G$ rooted at $r$. Because the $x_i$'s are the leaves of $D$, the $A_i$'s actually partition $D$. Thus, the $R_i$'s contain all the vertices of $G$. As for the complexity, the idea for computing the $a_i$'s and $A_i$'s is to use tree contraction techniques [28] to compute for each vertex $v$ the lowest-indexed $x_i$ that is a descendant of $v$ in $D$. This descendant information can then be used to identify $a_i$ and $A_i$. The $b_i$'s and $B_i$'s are processed in the same way. □

LEMMA 3.4 (STAGE 2). *Given an ear cover for $G$*

*with half-simple ears and at most $2(n-1)$ edges, an ear cover for $G$ with internally simple ears can be computed in $O(\log n)$ time using $O(n/\log n)$ processors. Furthermore, the new ear cover also contains at most $2(n-1)$ edges.*

*Proof.* Let $R_1, \cdots, R_k$ be the given ear cover. This stage works exclusively with the edges in the $R_i$'s. For each $R_i$, the goal is to partition the edges of $R_i$ into a sequence of internally simple directed paths $S_{i,1}, \cdots, S_{i,t_i}$ such that the endpoints of $S_{i,1}$ are those of $R_i$, and for each $h > 1$, each endpoint of $S_{i,h}$ lies in a lower-indexed $S_{i,h'}$. Intuitively, $S_{i,1}, \cdots, S_{i,t_i}$ form an ear cover for $R_i$ rooted at the endpoints of $R_i$. Based on this intuition, it is easy to see that the $S_{i,j}$'s form an ear cover for $G$ with internally simple ears under the lexicographic order induced by the double-index $(i,j)$. After the $S_{i,j}$'s and $t_i$'s are computed, the double-indexing can be converted into single-indexing by parallel prefix computation [8] in $O(\log n)$ time using $O(n/\log n)$ processors. To finish the proof, it suffices to describe how to efficiently compute $S_{i,1}, \cdots, S_{i,t_i}$ for $R_i$. For notational brevity, the index $i$ is omitted in the following discussion. Let $R$ be formed by two simple directed paths $A$ from $a$ to $x$ and $B$ from $x$ to $b$. The ears $S_1, \cdots, S_h$ are constructed from $A$ and $B$ in the same way as an ear cover is from a CD-pair in the proof for Lemma 3.3. More precisely, let $y_1, \cdots, y_g$ be the vertices shared by $A$ and $B$ in the order of appearance in $A$. Note that $y_g = x$. Now, consider $A$ a divergent tree rooted at $a$ and consider $B$ a convergent tree rooted at $b$. Let $P_h$ be the tree path in $A$ between $y_h$ and the lowest ancestor $p_h$ of $y_h$ in $A$ such that $y_h$ either is $a$ or lies in a lower-indexed $P_{h'}$. Let $Q_h$ be the tree path in $B$ between $y_h$ and the lowest ancestor $q_h$ of $y_h$ in $B$ such that $y_h$ either is $b$ or lies in some lower-indexed $P_{h'}$. Let $S_h$ be the directed path formed by concatenating $P_h$ and $Q_h$ at $y_h$. □

The following two lemmas are easy to prove.

LEMMA 3.5 (STAGE 3). *Given an ear cover for $G$ with internally simple ears and at most $2(n-1)$ edges, an ear cover for $G$ satisfying the simplicity and intersection conditions can be computed in $O(\log n)$ time using $O(n/\log n)$ processors. Furthermore, the new ear cover also contains at most $2(n-1)$ edges.*

LEMMA 3.6 (STAGE 4). *Given an ear cover for $G$ that contains at most $2(n-1)$ edges and satisfies the simplicity and intersection conditions, an ear decomposition for $G$ can be computed in $O(\log n)$ time using $O((n+e)/\log n)$ processors.*

**4. Topological ordering.** In this section, we give an efficient parallel algorithm that computes a topological ordering for a planar dag. As already highlighted, this algorithm relies on the planar orientation structure described in Theorem 2.1 and builds on the ear decomposition algorithm developed in §3. The following theorem formally states the complexity of the algorithm.

THEOREM 4.1. *For a planar dag with $n$ vertices, a topological ordering can be computed in $O(\log^2 n)$ time using $O(n)$ processors on a* CRCW PRAM.

To prove the theorem, we describe the topological ordering algorithm as follows. Let $G$ be an acyclic digraph; the goal is to find a topological ordering of $G$. A new notion is in order. A *topological segmentation* of $G$ is a partition of the vertices into a sequence of sets $V_1, \cdots, V_k$ such that each edge in $G$ either points from some $V_i$ to $V_i$ itself or points from a lower-indexed $V_i$ to a higher-indexed $V_j$. Intuitively, a topological segmentation is an approximation to a topological ordering. The approximation scheme is based on two immediate facts. First, for any topological segmentation, there always exists a consistent topological ordering, one in which the vertices of a lower-indexed $V_i$ are placed all before those of a higher-indexed $V_j$. Second, if each $V_i$ contains exactly one vertex, then the topological segmentation is actually a topological ordering of $G$. Therefore, to find a topological ordering of $G$, it suffices to starts with a coarse topological segmentation and then iteratively refine the segmentation until it becomes a topological ordering. In the following discussion, we give such a refinement process; for ease of understanding, we first explain the refinement process as a sequential algorithm and then discuss how to parallelize the process.

We now detail the sequential version of the refinement process. First of all, the coarsest segmentation of $G$ is $G$ itself. This trivial segmentation can be refined into a segmentation consisting of two subsets as follows. Let $\tilde{G}$ denote the dual of $G$. From Theorem 2.1, $\tilde{G}$ is strongly connected and thus has a directed ear decomposition $\tilde{E}_1, \cdots, \tilde{E}_k$. Because $\tilde{E}_1$ is actually a vertex-simple directed cycle in $\tilde{G}$, the ear $\tilde{E}_1$ divides the plane into two disconnected regions. Let $A$ be the region such that $E_1$ runs counterclockwise as observed from inside $A$; symmetrically, let $B$ be the region such that $E_1$ runs clockwise as observed from inside $B$. Notice that the faces of $\tilde{G}$ are divided between $A$ and $B$. Because the faces of $\tilde{G}$ correspond to the vertices in $G$, the vertices of $G$ are also divided between $A$ and $B$. Let $V_A$ and $V_B$ be the sets of vertices of $G$ in $A$ and $B$, respectively. Because the edges of $\tilde{E}_1$ run counterclockwise with respect to $A$, the edges of $G$ between $V_A$ and $V_B$ all go from $V_A$ to $V_B$. This unidirectional property ensures that there are no directed paths in $G$ from $V_B$ to $V_A$. Thus, the sequence $V_A, V_B$ forms a topological segmentation of $G$. This segmentation is finer than the trivial segmentation $G$ itself. To continue the refinement process, we add $\tilde{E}_2$ to the discussion. From the planarity of $\tilde{G}$ and from the intersection condition of an ear decomposition, $\tilde{E}_2$ is either within $A$ or within

$B$. Without loss of generality, assume that $\tilde{E}_2$ is within $A$. From the endpoint condition of an ear sequence, $\tilde{E}_2$ divides $A$ into smaller regions. From the simplicity and intersection conditions of an ear decomposition, $A$ is divided into exactly two regions $C$ and $D$ such that $\tilde{E}_2$ runs counterclockwise with respect to $C$ and runs clockwise with respect to $D$. Using the same analysis as for $E_1$, the set of vertices of $G$ inside $A$ are divided into two sets; let $V_C$ and $V_D$ be the sets of the vertices inside $C$ and $D$, respectively. Because $\tilde{E}_2$ runs counterclockwise with respect to $C$, the edges of $G$ between $V_C$ and $V_D$ all go from $V_C$ to $V_D$. This unidirectional property ensures that there are no directed paths from $V_D$ to $V_C$. Thus, the sequence $V_C, V_D$ is a topological segmentation for the subgraph of $G$ induced by $V_A$. In sum, the sequence $V_C, V_D, V_B$ is a topological segmentation for $G$. If this process is iterated until the last ear is also considered, then from the partition condition of an ear decomposition, every region contains only one face of $\tilde{G}$ and thus only one vertex of $G$. Consequently, the resulting topological segmentation is a genuine topological ordering for $G$.

The refinement process can be parallelized by applying a bisection strategy to the ear decomposition $\tilde{E}_1, \cdots, \tilde{E}_k$; a similar strategy has been used by Kao to compute daisy graphs for planar directed depth-first search [17]. Let $\tilde{G}'$ denote the graph formed by $\tilde{E}_1, \cdots, \tilde{E}_{\lceil k/2 \rceil}$. By induction, $\tilde{G}'$ divides the plane into $\lceil k/2 \rceil + 1$ regions $R_1, \cdots, R_{\lceil k/2 \rceil + 1}$. Let $V_i$ be the set of vertices of $G$ inside $R_i$; let $G_i$ be the subgraph of $G$ induced by $V_i$. Furthermore, let $\tilde{\Pi}_i$ be the set of ears $\tilde{E}_j$' inside each $R_i$. These $V_i$'s, $G_i$'s, and $\tilde{\Pi}_i$'s can be found using undirected connected component algorithms [36]. By induction, the $V_i$'s can be arranged into a topological segmentation $T'$ for $G$. To refine $T'$ into a topological ordering of $G$, it suffices to first compute a topological ordering $T_i$ for each $G_i$ and then substitute the sequence $T_i$ for the set $V_i$ in the segmentation $T'$. After the substitution, the vertices in the resulting topological ordering can be reindexed by parallel prefix computation. To finish describing the parallelization, it suffices to show that $T', T_1, \cdots, T_{\lceil k/2 \rceil}$ can be recursively and independently computed in parallel. It can shown that the problem of computing $T'$ involves only $\tilde{E}_1, \cdots, \tilde{E}_{\lceil k/2 \rceil}$, and the problem of computing each $T_i$ involves only $\tilde{\Pi}_i$. In other words, each of these subproblems involves no more than half of the original ears; consequently the depth of recursion is at most $\lceil \log k \rceil$. To elaborate on the recursion for $T'$, let $G'$ be the graph constructed from $G$ by contracting each $V_i$ into a single vertex. Notice that $\tilde{G}'$ is in effect the dual of $G'$ and the sequence $\tilde{E}_1, \cdots, \tilde{E}_{\lceil k/2 \rceil}$ is an ear decomposition of $\tilde{G}'$. More importantly, a topological *segmentation* of $G$ formed by the $V_i$'s is in effect a topological *ordering* for $G'$. Consequently, the problem of computing $T'$ is one of computing a topological ordering for $G'$. To elaborate on the recursion for $T_i$, let $\tilde{X}_i$ be the subgraph of $\tilde{G}$ formed by the edges and vertices of the faces inside $R_i$. Let $\tilde{G}_i$ be the graph constructed from $\tilde{X}_i$ by contracting the boundary of $R_i$ into a single vertex. Let $\tilde{\Delta}_i$ be the set of paths obtained by contracting into a single vertex all endpoints of $\tilde{\Pi}_i$ on the boundary of $R_i$. Observe that the dual of $G_i$ is in effect $\tilde{G}_i$. Furthermore, $\tilde{\Delta}_i$ forms an ear decomposition for $\tilde{G}_i$; thus, an ear decomposition for $\tilde{G}_i$ with a small number of ears is easily obtained without recomputation from scratch. This finishes the proof of Theorem 4.1.

**5. Digraph reachability.** In addition to the multiple-source and single-source reachability problems, we also consider the following third version: the *single-source single-sink* reachability problem is to find all the vertices that can reach a specified vertex and at the same time can be reached from another specified vertex. In this section, we describe linear-processor NC reductions from the problems of computing digraph reachability to those of computing strongly connected components and topological ordering. As highlighted in the Introduction, these reductions apply to any minor-closed family of digraphs. All these reductions employ divide-and-conquer strategies based on the given oracles for computing strongly connected components and topological ordering. In the following discussion, we first abstract two useful ideas from the divide-and-conquer strategies, and then detail the reductions for the three reachability problems.

Throughout the following discussion, let $G$ denote the input graph to the reachability problems for a minor-closed family. Without loss of generality, we assume that $G$ is a dag. Otherwise, using the given oracle for strongly connected components, we can find these components in $G$ and then contract each component into a single vertex. The resulting dag not only contains all information for the digraph reachability in the original graph but also remain in the given minor-closed family. The oracle for strongly connected components is also used repeatedly elsewhere. In contrast, the oracle for topological ordering is used only once for an entire reduction. Therefore, our discussion assumes that the input dag $G$ is equipped with a topological ordering. The NC reductions then manipulate $G$ and this ordering; the topological orderings for the subsequent versions of $G$ are derived efficiently from the initial ordering. The only two operations used to manipulate the input graph are those of partitioning a graph and contracting a connected subgraph. Such a partition actually consists of a sequence of edge deletions, and such a contraction consists of a sequence of edge contractions. Consequently, all subsequent versions of the input graph remain in the

given minor-closed family.

For convenience, a few notations are in order. Let $n$ and $e$ denote the number of vertices and edges in $G$. Let $T_{sc}(\cdot)$ and $P_{sc}(\cdot)$ denote the time and processor complexities for computing the strongly connected components of a graph in the given minor-closed family. Given a digraph $H$ and two sets of vertices $A$ and $B$, let $R_H(A,B)$ denote the set of vertices in $H$ that can be reached from $A$ and can reach $B$ through directed paths. In particular, $R_H(H,B)$ is the set of vertices that can reach $B$, and $R_H(A,H)$ is the set of vertices that can be reached from $A$.

The first key idea for divide-and-conquer is captured in the next lemma. The lemma describes a crucial step for recovering the reachability information encoded in small-size subproblems.

LEMMA 5.1. *Let $H$ be a dag. Let $t$ be a vertex in $H$. Let $S$ be a subset of $R_H(H,t)$. If $H$, $t$, $R_H(H,t)$, and $S$ are given, then $R_H(S,t)$ can be found in $O(\log n + T_{sc}(n))$ time using $O(n+e+P_{sc}(n,e))$ processors.*

*Proof.* $R_H(S,t)$ can be found in two stages. The first stage is to compute a convergent tree $T$ of $H$ that is rooted at $t$ and consists of the vertices in $R_H(H,t)$. The tree $T$ is built as follows. For each vertex $u \in R_H(H,t) - \{t\}$, choose an edge $u \rightarrow v$ in $H$ such that $v$ is also in $R_H(H,t)$. Because $H$ is a dag, these chosen edges form a tree with the desired properties. The second stage proceeds to identify $R_H(S,t)$ as follows. First identify all tree paths in $T$ between $t$ and $S$. Then contract all these paths into a single vertex. Let $H'$ denote the resulting graph. Let $C'$ denote the strongly connected component in $H'$ that contains $S$ and $t$. Because $S$ and $t$ are contracted into the same vertex in $H'$ and because the contracted vertices are in $R_H(S,t)$, the component $C'$ actually consists of $R_H(S,t)$. Because the contracted paths are all connected to $t$, the graph $H'$ is in the given minor-closed family. Consequently, $C'$ can be found using the given oracle for strongly connected components. As for the complexity, it is straightforward to implement the above two stages in $O(\log n+T_{sc}(n))$ time and $O(n+e+P_{sc}(n,e))$ processors using well-known fundamental parallel algorithms. □

Using the above idea, we can now describe the reduction for the single-source single-sink reachability problem. The complexity of the reduction is stated in the next theorem.

THEOREM 5.2 (SINGLE-SOURCE SINGLE-SINK REACHABILITY). *Given two vertices $s$ and $t$ in $G$ and a topological ordering for $G$, the set $R_G(s,t)$ can be found in $O(\log^2 n + T_{sc}(n)\cdot\log n)$ time using $O(n+e+P_{sc}(n))$ processors.*

*Proof.* First, two simplifying assumptions are in order. Let $v_1,\cdots,v_n$ be the given topological ordering of $G$. The vertices that appear before $s$ in the ordering cannot belong to $R_G(s,t)$; symmetrically, the vertices after $t$ cannot belong to $R_G(s,t)$. Thus, the following discussion assumes $v_1 = s$ and $v_n = t$. Because the problem is trivial for $n$ less than a small constant, the discussion further assumes that $n$ is greater than a constant big enough to cover all base cases. Under these assumptions, $R_G(s,t)$ is computed recursively in three stages as follows. Stage 1 prepares $R_G(s,t)$ for recursion. Let $V_s = \{v_1,\cdots,v_{\lceil n/2\rceil}\}$, and let $V_t = \{v_{\lceil n/2\rceil+1},\cdots,v_n\}$. Let $H_s$ (or $H_t$) be the subgraph of $G$ induced by $V_s$ (respectively, $V_t$). Let $C_s$ (or $C_t$) be the connected component of $H_s$ (respectively, $H_t$) that contains $s$ (respectively, $t$). Let $I$ be the subgraph of $G$ induced by $C_s \cup C_t$. Notice that there is no edge in $G$ from $C_s$ to $V_s - C_s$. Thus, the vertices in $V_s - C_s$ can be discarded without affecting $R_G(s,t)$. Symmetrically, the set $V_t - C_t$ can also be discarded. In sum, $R_G(s,t) = R_I(s,t)$. Stage 2 breaks the problem of computing $R_I(s,t)$ into two small-size subproblems. Let $I_s$ (or $I_t$) be the graph constructed from $I$ by contracting $C_t$ (respectively, $C_s$) into a single vertex $t'$ (respectively, $s'$). Observe that $I$ is a dag in the given minor-closed family. Furthermore, $C_s$ and $C_t$ form a topological segmentation of $I$, and are each connected in $I$. Therefore, $I_s$ and $I_t$ are dags in the given minor-closed family. Also, a topological ordering for each of $I_s$ and $I_t$ can be easily obtained from the given ordering of $G$. Thus, the reduction may recursively compute $R_{I_s}(s,t')$ and $R_{I_t}(s',t)$. The progress made by this stage is as follows. Because $C_s$ and $C_t$ form a topological segmentation of $I$, $R_{I_s}(s,t') - \{t'\}$ consists of the vertices in $C_s$ that can be reached from $s$ and can reach $C_t$. Symmetrically, $R_{I_t}(s',t) - \{s'\}$ consists of the vertices in $C_t$ that can be reached from $C_s$ and can reach $t$. Stage 3 recovers $R_I(s,t)$ from $R_{I_s}(s,t')$ and $R_{I_t}(s',t)$ in $O(\log n + T_{sc}(n))$ time and $O(n+e+P_{sc}(n))$ processors. Let $W_s = \{u \mid u$ is a vertex in $C_s$ such that there is an edge $e = u \rightarrow v$ in $I$ with $v \in R_{I_t}(s',t).\}$. Symmetrically, let $W_t = \{v \mid v$ is a vertex in $C_t$ such that there is an edge $e = u \rightarrow v$ in $I$ with $u \in R_{I_s}(s,t').\}$. Let $W = R_{I_s}(s,W_s) \cup R_{I_t}(W_t,t)$. It is straightforward to verify that $R_I(s,t) = W$. As for the complexity of this stage, $W_s$ and $W_t$ can be found by testing every edge in $I$ between $C_s$ and $C_t$. The set $W$ can be computed using Lemma 5.1. Therefore, the total complexity for the third stage is $O(\log n + T_{sc}(n))$ time and $O(n+e+P_{sc}(n))$ processors. □

THEOREM 5.3. *For a planar digraph with $n$ vertices, the single-source single-sink reachability problem can be solved in $O(\log^3 n)$ time using $O(n)$ processors on a* CRCW PRAM.

*Proof.* Notice that in the proof of Theorem 5.2, the strongly connected component oracle is applied

only to graphs that are built from a dag by contracting a connected subgraph. Thus, from Theorem 2.3, $T_{sc}(n) = O(\log^2 n)$ and $P_{sc}(n, e) = O(n)$. □

The second key idea for divide-and-conquer is the notion of a *bottleneck vertex* defined as follows. To define the notion, let $v_1, \cdots, v_n$ be the given topological ordering for $G$. Let $v_k$ be the highest-indexed vertex such that the subgraph induced by $\{v_k, \cdots, v_n\}$ contains a connected component with at least $n/2$ vertices. Let $B$ denote this large component. Let $A$ denote the set of vertices in $G - B$. Let $C_1, \cdots, C_p$ denote the connected components in the subgraph induced by $B - \{v_k\}$. Observe that $A$, $v_k$, and the $C_i$'s form a topological segmentation of $G$, and that there is an edge from $v_k$ to every $C_i$. This observation suggests the following terminology. The vertex $v_k$ is called the *bottleneck vertex*, the set $A$ the *pre-bottleneck subgraph*, the components $C_i$' the *post-bottleneck components*, and the triple $(A, v_k, B)$ the *bottleneck triple*. To formulate a general divide-and-conquer strategy, let $G'$ be the graph constructed from $G$ by contracting $B$ into a single vertex $t'$. Because $B$ is connected, $G'$ is in the given minor-closed family. Because $A$ and $B$ form a topological segmentation of $G$, the graph $G'$ is acyclic and a topological ordering for $G'$ can be easily obtained from the given ordering for $G$. The divide-and-conquer strategy consists of two stages. The first stage solves a reachability problem in $G'$ so as to find the representatives in $B$ of the sources of $A$. The reachability problem for $G'$ is of a simpler version than the version for $G$. More precisely, the multiple-source problem is reduced to the single-source problem, which is in turn reduced to the single-source single-sink problem. The second stage computes the reachability of $A$ and the $C_i$'s independently in parallel. These reachability problems are of the same version as that for $G$. Because $|A| \leq n/2$ and $|C_i| < n/2$, these subproblems are at most half the size of the original problem for $G$. This half-size property ensures that the depth of recursion is at most $\lceil \log n \rceil$. To control other aspects of the recursion complexity, the reductions for the reachability problems also satisfy the following three properties. First, the total size of graphs induced from $G$ at any recursion level is at most linear in the size of $G$; this property is for achieving the desired linear-processor complexity. Second, the graphs induced from $G$ for recursion are dags in the given minor-closed family. Third, a topological ordering for each of these induced graphs can be easily obtained from the given ordering for $G$. For the reduction theorems below, these recursion properties can be verified in the same way as in the proof of Theorem 5.2, and thus some of the verification details are omitted for brevity.

THEOREM 5.4 (SINGLE-SOURCE REACHABILITY). *Given a vertex $s$ in $G$ and a topological ordering for $G$, the set $R_G(s, G)$ can be found in $O(\log^3 n + T_{sc}(n) \cdot \log^2 n)$ time using $O(n + e + P_{sc}(n, e))$ processors.*

*Proof.* To compute $R_G(s, G)$, first find the bottleneck triple by binary search. Then there are three cases: $s \in B - \{v_k\}$, $s = v_k$, or $s \in A$. In the first two cases, the problem can be immediately reduced to half-size subproblems. We concentrate the discussion on the third case. Case (3): $s \in A$. First find $R_{G'}(s, t')$; this is a single-source single-sink reachability problem and can be solved using Theorem 5.2. Then find the the set $L$ of vertices in $R_{G'}(s, t') - \{t'\}$ that can reach $B$ via a single edge. There are two subcases based on whether or not $L$ can reach $v_k$ via a single edge. Case (3a): $L$ can reach $v_k$. Let $H$ be the subgraph of $G$ formed by $L$ and $B$. Let $H'$ be the graph constructed from $H$ by contracting $L \cup \{v_k\}$ into a new vertex $s'$. Then $R_G(s, G) = R(s, A) \cup \{v_k\} \cup (R_{H'}(s', H') - \{s'\})$. Case (3b): $L$ cannot reach $v_k$. Let $H$ be the subgraph of $G$ formed by $L$ and $B - \{v_k\}$. Let $H'$ be the graph constructed from $H$ by contracting $L$ into a new vertex $s'$. Then $R_G(s, G) = R_A(s, A) \cup (R_{H'}(s', H') - \{s'\})$. In either subcase, $R_{H'}(s', H')$ can be broken into small pieces, as is $R_B(s, B)$ in Case (2). □

THEOREM 5.5. *For a planar digraph with $n$ vertices, the single-source reachability problem can be solved in $O(\log^4 n)$ time using $O(n)$ processors on a* CRCW PRAM.

THEOREM 5.6 (MULTIPLE-SOURCE REACHABILITY). *Given a set $S$ of vertices in $G$ and a topological ordering for $G$, the set $R_G(S, G)$ can be found in $O(\log^4 n + T_{sc}(n) \cdot \log^3 n)$ time using $O(n + e + P_{sc}(n))$ processors.*

*Proof.* $R_G(S, G)$ is computed in two stages. The first stage is to compute the set $Z$ of vertices in $B - \{v_k\}$ that can be reached from $S \cap (A \cup \{v_k\})$ via a directed path in $A \cup \{v_k\}$ and an edge from $A \cup \{v_k\}$ to $B - \{v_k\}$. Intuitively, $Z$ represents the sources of $A \cup \{v_k\}$ to the subgraph $B - \{v_k\}$. It is easy to compute $Z$ once we have computed $R_G(S, B)$. To do this, we first contract $B$ to a single vertex $t'$, obtaining $G'$. We compute $R_{G'}(A, t')$ using a single-sink reachability algorithm. By Lemma 5.1, we can then compute $R_G(S, t')$, which is $R_G(S, B)$. The second stage is to recursively compute $R_G(S, G) = R_A(S \cap A, A) \cup (S \cap \{v_k\}) \cup R_{B-\{v_k\}}((Z \cup S) \cap (B - \{v_k\}), B - \{v_k\})$. The first stage is more complicated than it would ideally be. The complication is due to the following subtle difficulty: because $B - \{v_k\}$ may not be connected, the set cannot be directly contracted to simplify $G$ without possibly destroying the membership of $G$ in the given minor-closed family. □

THEOREM 5.7. *For a planar digraph with $n$ vertices, the multiple-source reachability problem can be solved in $O(\log^5 n)$ time using $O(n)$ processors on a* CRCW PRAM.

**6. Descendent counting.** In this section, we give a linear-processor NC algorithm for planar descendant counting. The algorithm makes use of the two-path separator in Theorem 2.5 and builds upon the single-source reachability algorithm in §5. To precisely state the result, a few definitions are in order. A digraph is *rooted* at a vertex if that vertex can reach every other vertex via directed paths. Let $G$ be a rooted planar dag. Let $w(\cdot)$ be an assignment of weights to the vertices of $G$. The descendant counting problem for $G$ is to compute for each vertex $v$, the sum $\sigma(v)$ of the weights assigned to the descendents of $v$. A prototypical application of the algorithm is to count the descendents of each vertex, where all weights are 1. The algorithm can also compute $\sigma(v)$ in any commutative semigroup as long as binary addition takes constant time.

THEOREM 6.1. *Let $n$ denote the number of vertices in $G$. The descendant counting problem for $G$ can be solved in $O(\log^6 n)$ time using $O(n)$ processors on a* CRCW PRAM.

To prove the theorem, we detail the descendant counting algorithm in the following subsections. In §6.1, we discuss an easier counting problem. Given a directed path $P = v_1, \cdots, v_k$ in $G$, the *path subproblem* is to compute the sum $\sigma(v)$ only for each vertex $v$ in the path $P$. In §6.2, the solution to the path subproblem is used to solve the original problem. For ease of understanding, we describe a recursive algorithm for the descendant counting problem without addressing the issue of time and processor efficiency. In §6.3, we explain how to implement this basic algorithm so that the depth of recursion is polylog in the size of $G$. The key to ensuring such a small recursion depth is the use of the two-path separator given in Theorem 2.5. In §6.4, we modify the basic algorithm so that the processor complexity is linear in the size of $G$.

**6.1. The path subproblem.** The path subproblem has the following simple solution.

P1 For each vertex $v_i$, determine the set $R(v_i)$ of descendents of $v$ that are *not* descendents of $v_i$'s successor $v_{i+1}$.

P2 For each $1 \le i \le k$, let $f(v_i)$ be the sum of weights of vertices in $R(v_i)$.

P3 For each $1 \le i \le k$, compute $\sigma(v_i) = \sum_{j=i}^{k} f(v_j)$.

The set $R(v_i)$ computed in step P1 is a variant of the *dangling subgraph* defined for directed depth-first search [2]. To compute these subgraphs, we use a divide-and-conquer technique. If $P$ contains only one vertex, we can solve the problem directly. Otherwise, first determine the set $A$ of descendents of $v_{\lceil k/2 \rceil}$ as a single-source reachability problem. Then recurse in parallel on two subproblems, one for the subgraph $A$ and the subpath $v_{\lceil k/2 \rceil}, \cdots, v_k$, and the other for the subgraph $G - A$ and the subpath $v_1, \cdots, v_{\lceil k/2 \rceil - 1}$. Once step P1 has been carried out, step P2 is easy because the sets $R(v_i)$'s are disjoint. Step P3 can then be implemented using parallel prefix computation. This completes the description of our solution to the path subproblem.

LEMMA 6.2. *The path subproblem can be solved in $O(\log^5 n)$ time using $O(n)$ processors on a* CRCW PRAM.

**6.2. A basic algorithm.** Now we use the solution to the path subproblem in our recursive solution to the original problem. At each level of recursion, we are given a vertex-weighted graph $G$ in which some of the vertices $v$ have already been assigned labels $\sigma(v)$; our task is then to assign labels to the remaining vertices of $G$. We proceed as follows.

D1 Find a directed path $P = v_1, \cdots, v_k$ in $G$.

D2 For each ..., compute $\sigma(v_i)$. This step can be carried out using the procedure for the path subproblem. Now the vertices of $P$ are all labelled.

D3 Identify the connected components $C_q$ of $G - P$ that contain unlabelled vertices.

D4 For each component $C_q$ in parallel,

D5 let $G_q$ be the subgraph of $G$ induced by $C_q \cup P$;

D6 for $1 \le i \le k$, let $f_q(v_i)$ be the sum of weights of proper descendents of $v_i$ in $G_q$ that are not descendents of $v_{i+1}$.

D7 For each component $C_q$, for $1 \le i \le k$, assign a new weight to the vertex $v_i$,

$$w_q(v_i) := w(v_i) + \sum_{q' \neq q} f_{q'}(v_i)$$

where the sum is over indices $q'$ of components different from $C_q$. The weight $w_q(v_i)$ is $w(v_i)$ plus the sum of weights of descendents of $v_i$ that lie in components other than $C_q$ and are not reachable from $v_{i+1}$. Weights $w_q(v)$ for vertices not in $P$ are the same as the previous weights $w(v)$.

D8 Recurse on each $G_q$ with weights $w_q(\cdot)$.

The correctness of the procedure is straightforward to verify. Let us focus on a particular subgraph $G_q$. When components $C_{q'}$ are stripped away from $G$ leaving only the graph $G_q$, they leave their mark in the form of updated weights on the boundary $P$. These new weights ensure that the sum of weights of descendents of each vertex $v \in C_q$ in the graph $G_q$ is the same as in the graph $G$.

Now we consider implementation. Steps D2 and D6 can be executed using the techniques of the path subproblem, and step D3 can be done using any undirected components algorithm [36].

**6.3. Limiting the recursion depth.** The key to the algorithm's efficiency is a careful choice of the path $P$ in step D1. At the top level, before commencing the

recursion, we choose a divergent directed tree $D_0$ of the initial graph $G_0$. We will maintain the invariant that, at every level of recursion, for each subgraph $G$ being recursed on, the subgraph $D_0[G]$ induced by $G$ is in fact itself a directed tree with the same root as $D_0$. We call this the *directed tree invariant.*

Given a tree $T$ and a vertex $u$, let $P_u(Tree)$ denote the root-to-$u$ path in the tree. At each level of recursion, we choose as our path $P$ of step D1 a root-to-vertex path $P_u(D_0[G])$ of the induced directed tree $D_0[G]$. Consider the resulting subgraphs $G_q = G[C_q \cup P]$ obtained in step D5. The directed tree invariant states that the subgraph $D_0[G_q]$ of $D_0$ induced on each subgraph is itself a directed tree. To prove this invariant, let $v$ be any vertex of $G_q$; we claim that the parent of $v$ is also in $G_q$. If the parent of $v$ is in $P$, certainly the parent is in $G_q$, for $G_q$ contains $P$. Otherwise, the parent is a vertex of $G-P$ that is adjacent to $v$, and hence a vertex in the same component $C_q$ as $v$. But $C_q$ is contained in $G_q$, so the claim is proven.

We choose the root-to-vertex path in a way that ensures logarithmic depth of recursion. Recall from Theorem 2.5 that we can choose a pair $u, v$ of vertices such that each component of $G - P_u(D_0[G]) - P_v(D_0[G])$ is *small* in that it contains at most two-thirds of the unlabelled vertices of $G$. Having found such a pair $u, v$ of vertices, we let $P_u(D_0[G])$ be the path $P$ of step D1. Since the tree $D_0[G]$ is directed, the path $P$ is a directed path. Consequently, every subgraph $G_q$ of step D5 is small except possibly the one containing $v$. For that one large subgraph $G_q$, at the *next* level of recursion, we use the path $P_v(D_0[G])$ as $P$ in step D1. Consequently, every resulting subgraph of step D5 is small relative to $G$. This strategy ensures that the recursion depth is at most $2\log_{3/2}|G_0|$.

**6.4. Limiting the processor count.** It remains to ensure that, at every level of recursion, the sum of the sizes of all subgraphs being recursed on is linear in the size of the original graph. Unfortunately, that is not true of the algorithm as it stands; we must make a slight modification. The difficulty is that in constructing the graphs $G_q = G[C_q \cup P]$ from the components $C_q$, we duplicate each vertex of $P$ many times, once for each component $C_q$. The fix for this difficulty is to duplicate a vertex $v_i$ of $P$ only for components from which arcs enter $v_i$. This modification allows us to charge each duplication of $v_i$ to one of its incoming edges. In the remainder of this section, we outline this approach in greater detail.

Conceptually, the algorithm proceeds as follows. Immediately before the recursion step, the algorithm obtains a contracted version $G'_q$ from $G_q$ by contracting edges $(v_i, v_{i+1})$ of $P$ where the child $v_{i+1}$ has no incoming edge from $C_q$, and adding the weights of the identified vertices. Let $P_q$ be the resulting contracted version of $P$ in $G'_q$. This modification to the graph and the weights does not change the sum of weights of descendents of a vertex $v \in C_q$, as we now show. If the child $v_{i+1}$ was reachable from $v$ in $G_q$, then the parent $v_i$ was also reachable from $v$, because in $G_q$ the only arc entering $v_{i+1}$ comes from $v_i$. Hence identifying $v_{i+1}$ with $v_i$ and adding the weight of $v_{i+1}$ to that of $v_i$ does not change the sum of weights of descendents of $v$.

Because of the edge contractions in the modified algorithm, we must modify the directed tree invariant. We still maintain a convergent directed tree $D$ for $G$, but to do so we must contract edges of the directed corresponding to edge contractions in $G$.

We will presently describe how to efficiently implement the modified algorithm, but first we show that the modified algorithm achieves the desired goal: at each level of recursion, the sum of the sizes of all the graphs being recursed on is linear in the size of the original graph $G_0$. The argument has four parts. First we observe that the nontree edges of $G_0$ are partitioned among the various graphs being recursed on at a given level of recursion. This holds inductively because when the graph is decomposed into subgraphs, then only edges that could lie in several subgraphs are the edges of $P$, which are tree-edges. Hence the number of nontree edges overall is at most the number of nontree edges in $G_0$. Second, we infer that the number of duplicates of an original nonroot node $v \in G_0$ is bounded by the number of incoming nontree edges. This follows inductively from the modification of the algorithm: a nonroot node $v_i \in P$ appears in a subgraph $G'_q$ only if $G'_q$ contains a nontree edge entering $v_i$. Hence the total number of nonroot nodes overall is at most the number of nodes in $G_0$, plus the number of nontree edges in $G_0$. Third, we observe that there is at most one tree edge per duplicated nonroot node. Hence the number of tree edges overall is at most the number of nodes in $G_0$ plus the number of nontree edges in $G_0$. Fourth, we note that the number of duplicates of the root node at any level of recursion is bounded by the number of distinct subgraphs being recursed on at that level. In each such subgraph, there is at least one node that has not yet been labelled by $\sigma$, else there is no need to recurse on that subgraph. Such a node has never been duplicated. Thus the number of duplicates of the root is at most the number of original nodes of $G_0$. It follows, finally, that the total size of all subgraphs being recursed on at a given level is within a constant factor of the size of the original graph $G_0$.

It remains to describe how to efficiently implement the modified algorithm. Once the components $C_q$ have been identified, there are two tasks to perform before the recursions may commence. (1) The algorithm must

construct for each component $C_q$ the contracted path $P_q$ in which every node has an incoming edge from $C_q$. This task can be done using sorting and simple graph manipulations in $O(\log n)$ time using one processor per edge incident to the path $P$. (2) The algorithm must compute the new weights for nodes of $P_q$. We omit details; the key is the following well-known lemma.

LEMMA 6.3 (FOLKLORE). *One can process a sequence $\lambda_1, \ldots, \lambda_k$ of values in $O(\log k)$ time using $k$ processors so that subsequently, a single processor can in $O(\log k)$ time obtain the sum $\lambda_i + \lambda_{i+1} + \cdots + \lambda_j$ of any subsequence.*

**7. Depth-first search.** In the following discussion, we combine all the techniques we have developed up to now to give a linear-processor NC algorithm for depth-first search in planar directed graphs. The discussion is divided into three parts. First, we review the notion of directed separators defined by Kao [17]. The notion plays a crucial role in the NC algorithms for directed depth-first search [17], [2]. Next, we reformulate the key reduction theorems for depth-first search by Kao [17] and by Aggarwal, Anderson, and Kao [2]. Finally, we show that the techniques in the previous sections suffice to implement the depth-first search reductions in polylog time using linear processors for planar directed graphs.

Intuitively, a separator of a graph is a subgraph whose removal disconnects the graph into small pieces. This section follows the directed separator definition given by Kao [17]: a *separator* of an $n$-vertex directed graph $G$ is a set of vertices $S$ such that the largest strongly connected component in $G - S$ contains at most $\alpha \cdot n$ vertices for some constant $\alpha$ between zero and one. A directed *path* separator is a vertex-simple directed path whose vertices form a separator; a directed *multipath* separator is a set of vertex-disjoint vertex-simple directed paths whose vertices form a separator; a directed *cycle* separator is a vertex-simple directed cycle whose vertices form a separator. A single vertex is considered a cycle of length zero; thus, if the removal of a vertex separates a graph, the vertex is a cycle separator. For $\alpha = 1/2$, Kao has shown that every directed graph has a directed path separator and a directed cycle separator, and that these separators can be found efficiently in sequential and parallel computation [17].

The next two theorems rephrase related results from the papers on directed depth-first search by Kao [17] and by Aggarwal, Anderson, and Kao [17]. Both theorems apply to any given minor-closed family of digraphs. For notational brevity, we employ the following abbreviations. Let *scc*, *dst*, *ssssr*, *ssr*, *msr*, *dc*, and *dcs* stand for, respectively, strongly connected components, directed spanning trees in strongly connected graphs, single-source single-sink reachability, single-source reachability, multiple-source reachability, descendant counting for rooted digraphs, and directed cycle separators for strongly connected digraphs. In general, the abbreviations are composed of the first initials of the terms. For each abbreviation $x$, let $T_x(n)$ and $P_x(n)$ denote the time and processor complexities of the corresponding problem for an input graph with $n$ edges and vertices.

Let $T_{mer}(n) = (2 \cdot \lceil \log n \rceil + 3) \cdot (T_{scc}(n) + T_{dst}(n))$.

Let $P_{mer}(n) = P_{scc}(n) + P_{dst}(n)$.

THEOREM 7.1. *Let $G$ be a digraph of size $n$. Let $Q$ be a multipath separator of $G$ with $k$ disjoint paths. Given $G$ and $Q$, a directed cycle separator for $G$ can be found in $k \cdot T_{mer}(n)$ time using $P_{mer}(n)$ processors.*

*Proof.* The proof directly follows that of Theorem 3 and the discussion in §2.2 in [17]; a more detailed exposition of the same discussion is in §4.2 in [2]. $T_{mer}(n)$ and $P_{mer}(n)$ are the time and processor complexities of merging the two ends of a path or merging two ends from two paths. □

Let $T_{dan}(n) = T_{scc}(n) + T_{dc}(n) + T_{dcs}(n) + T_{ssssr}(n) + T_{dst}(n) + \log n \cdot T_{ssr}(n)$.

Let $P_{dan}(n) = P_{scc}(n) + P_{dc}(n) + P_{dcs}(n) + P_{ssssr}(n) + P_{dst}(n) + P_{ssr}(n)$.

THEOREM 7.2. *Let $G$ be a digraph of size $n$. Then the depth-first search problem for $G$ can be solved in $\lceil \log n \rceil \cdot T_{msr}(n) + \lceil \log n \rceil^2 \cdot T_{dan}(n)$ time using $P_{msr}(n) + P_{dan}(n)$ processors.*

*Proof.* The proof directly follows that of Theorem 3.3 in [2]. In the complexity estimate, the terms $\lceil \log n \rceil \cdot T_{msr}(n)$ and $P_{msr}(n)$ account for breaking $G$ with several starting vertices into several rooted digraphs each with one starting vertex. The terms $T_{dan}(n)$ and $P_{dan}(n)$ account for using a cycle separator to break a rooted digraph into several rooted subgraphs. The term $\lceil \log n \rceil^2$ accounts for the fact that the breakup process is iterated at most $\lceil \log n \rceil^2$ times. □

We now prove the main results of this section.

THEOREM 7.3. *For a strongly connected planar digraph with $n$ vertices, a directed cycle separator can be found in $O(\log^4 n)$ time using $O(n)$ processors on a* CRCW PRAM.

*Proof.* Let $G$ denote the given graph. A directed cycle separator for $G$ is constructed in three steps as follows. Step 1 uses Theorem 2.4 to compute a directed spanning tree $T$ for $G$. Step 2 uses Theorem 2.5 to compute from $T$ a two-path undirected separator for $G$. Notice that because $T$ is a directed tree, this two-path *undirected* separator is also a two-path *directed* separator. Step 3 uses Theorem 7.1 to convert the two-path separator into a directed cycle separator. The total complexity of these steps follows the estimates in Theorems 2.4, 2.5, 7.1, and 2.2. □

THEOREM 7.4. *For a planar digraph with n vertices, the depth-first search problem can be solved in $O(\log^8 n)$ time using $O(n)$ processors on a* CRCW PRAM.

*Proof.* The proof follows Theorems 7.2, 5.7, 2.2, 6.1, 7.3, 5.3, 2.4, and 5.5. The most expensive subroutine is the descendant counting algorithm. □

**Acknowledgements.** We wish to thank Gary Miller, Greg Shannon, and Cliff Stein for many helpful and insightful discussions. The research of the first author was supported in part by NSF Grant CCR-8909323. The research of the second author was supported in part by ONR Grant N00014-88-K-0243 and DARPA Grant N00039-88-C0113. Most of the second author's work was done while he was at Aiken Computation Laboratory, Harvard University.

## REFERENCES

[1] A. AGGARWAL AND R. ANDERSON, *A random* NC *algorithm for depth first search*, Combinatorica, 8 (1988), pp. 1–12.

[2] A. AGGARWAL, R. ANDERSON, AND M. Y. KAO, *Parallel depth-first search in general directed graphs*, STOC, 1989, pp. 297–308.

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

[4] R. J. ANDERSON AND G. L. MILLER, *Deterministic parallel list ranking*, AWOC, 1988.

[5] J. BONDY AND U. MURTY, *Graph Theory with Applications*, North-Holland, 1976.

[6] R. COLE AND U. VISHKIN, *Optimal parallel algorithms for expression tree evaluation and list ranking*, AWOC, 1988.

[7] D. COPPERSMITH AND S. WINOGRAD, *Matrix multiplication via arithmetic progressions*, STOC, 1987, pp. 1–6.

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

[9] D. FUSSEL, V. RAMACHANDRAN, AND R. THURIMELLA, *Finding triconnected components by local replacements*, ICALP, 1989.

[10] D. FUSSEL AND R. THURIMELLA, *Separation pair detection*, AWOC, 1988, pp. 149–159.

[11] H. GAZIT AND G. L. MILLER, *A parallel algorithm for finding a separator in planar graphs*, FOCS, 1987, pp. 238–248.

[12] ———, *An improved parallel algorithm that computes the* BFS *numbering of a directed graph*, Inform. Process. Lett., 28 (1988), pp. 61–65.

[13] F. HARARY, *Graph Theory*, Addison-Wesley, 1969.

[14] X. HE AND Y. YESHA, *A nearly optimal parallel algorithm for constructing depth first spanning trees in planar graphs*, SIAM J. Comput., 17 (1988), pp. 486–491.

[15] J. JA'JA AND S. KOSARAJU, *Parallel algorithms for planar graphs and related problems*, IEEE Trans. Circuits and Systems, 35 (1988), pp. 304–311.

[16] A. KANEVSKY AND V. RAMACHANDRAN, *Improved algorithms for graph four-connectivity*, FOCS, 1987, pp. 252–259.

[17] M. Y. KAO, *All graphs have cycle separators and planar directed depth-first search is in* DNC, AWOC, 1988, pp. 53–63.

[18] ———, *Linear-processor* NC *algorithms for planar directed graphs* I: *strongly connected components*. Submitted, 1989.

[19] M. Y. KAO AND G. E. SHANNON, *Linear-processor* NC *algorithms for planar directed graphs* II: *directed spanning trees*. Submitted, 1989.

[20] ———, *Local reorientation, global order, and planar topology*, STOC, 1989, pp. 286–296.

[21] R. KARP AND V. RAMACHANDRAN, *A survey of parallel algorithms for shared-memory machines*. To appear in the *Handbook of Theoretical Computer Science*, North-Holland.

[22] P. N. KLEIN AND J. H. REIF, *An efficient parallel algorithm for planarity*, FOCS, 1986, pp. 465–477. Also in *Journal of Computer and System Sciences*, 37 (1988), pp. 190-246.

[23] S. R. KOSARAJU AND A. L. DELCHER, *Optimal parallel evaluation of tree-structured computations by raking*, AWOC, 1988.

[24] R. LIPTON AND R. TARJAN, *A separator theorem for planar graphs*, SIAM Journal on Algebraic and Discrete Methods, 36 (1979), pp. 177–189.

[25] L. LOVÁSZ, *Computing ears and branchings*, FOCS, 1985, pp. 464–467.

[26] Y. MAON, B. SCHIEBER, AND U. VISHKIN, *Parallel ear decomposition search* (EDS) *and st-numbering in graphs*, AWOC, 1986, pp. 34 – 45.

[27] G. MILLER AND V. RAMACHANDRAN, *A new graph triconnectivity algorithm and its parallelization*, STOC, 1987, pp. 335–344.

[28] G. MILLER AND J. REIF, *Parallel tree contractions and its applications*, STOC, 1985, pp. 478–489.

[29] G. L. MILLER, *Finding small simple cycle separators for 2-connected planar graphs*, Journal of Computer and System Sciences, 32 (1986), pp. 265–279.

[30] G. L. MILLER AND J. NAOR, *Flow in planar graphs with multiple sources and sinks*, FOCS, 1989, pp. 112–117.

[31] V. PAN AND J. H. REIF, *Extension of parallel nested dissection algorithm to the path algebra problems*, Tech. Rep. 9, Computer Science Department, State University of New York at Albany, 1985.

[32] ———, *Fast and efficient solution of path algebra problems*, Tech. Rep. 3, Computer Science Department, State University of New York at Albany, 1987.

[33] V. RAMACHANDRAN AND J. H. REIF, *An optimal parallel algorithm for graph planarity*, FOCS, 1989, pp. 282–287.

[34] V. RAMACHANDRAN AND U. VISHKIN, *Efficient parallel triconnectivity in logarithmic time*, AWOC, 1988, pp. 33 – 42.

[35] J. H. REIF, *Depth-first search is inherently sequential*, Inform. Process. Lett., 20 (1985), pp. 229–234.

[36] Y. SHILOACH AND U. VISHKIN, *An $O(\log n)$ parallel connectivity algorithm*, Journal of Algorithms, 3 (1982), pp. 57–67.

[37] J. R. SMITH, *Parallel algorithms for depth first search* I. *Planar graphs*, SIAM J. Comput., 15 (1986), pp. 814–830.

[38] R. TARJAN, *Depth-first search and linear graph algorithms*, SIAM J. Comput., 1 (1972), pp. 146–160.

[39] J. VITTER AND R. TAMMASIA, *Optimal parallel algorithms for transitive closure and point location in planar structures*, SPAA, 1989, pp. 299-408.