A Simple Parallel Algorithm for the Single-Source Shortest Path Problem on Planar Digraphs

Jesper L. Träff Christos D. Zaroliagis

MPI–I–96–1-012 June 1996

A Simple Parallel Algorithm for the Single-Source Shortest Path Problem on Planar Digraphs*

Jesper L. Träff Christos D. Zaroliagis

Max-Planck-Institut für Informatik

Im Stadtwald, D-66123 Saarbrücken, Germany

E-mail: {traff,zaro}@mpi-sb.mpg.de

June 3, 1996

Abstract

We present a simple parallel algorithm for the *single-source shortest path problem* in *planar digraphs* with nonnegative real edge weights. The algorithm runs on the EREW PRAM model of parallel computation in $O((n^{2\epsilon}+n^{1-\epsilon})\log n)$ time, performing $O(n^{1+\epsilon}\log n)$ work for any $0<\epsilon<1/2$. The strength of the algorithm is its simplicity, making it easy to implement, and presumably quite efficient in practice. The algorithm improves upon the work of all previous algorithms. The work can be further reduced to $O(n^{1+\epsilon})$, by plugging in a less practical, sequential planar shortest path algorithm. Our algorithm is based on a region decomposition of the input graph, and uses a well-known parallel implementation of Dijkstra's algorithm.

1 Introduction

The shortest path problem is a fundamental and well-studied combinatorial optimization problem with a wealth of practical and theoretical applications [1]. Given an n-vertex, m-edge directed graph $G=(V,E)$ with real edge weights, the *shortest path problem* is to find a path of minimum weight between two vertices u and v, for each pair u,v of a given set of vertex pairs; the weight of a u-v path is the sum of the weights of its edges. The weight of a shortest u-v path is called the *distance* from u to v. The shortest path problem comes in different variants depending on the given set of u,v vertex pairs, and the type of edge weights [1].

*This work was partially supported by the EU ESPRIT LTR Project No. 20244 (ALCOM-IT), and by the DFG project SFB 124-D6 (VLSI Entwurfsmethoden und Parallelität).

Although efficient sequential algorithms exist for many of these variants, there is a certain lack of efficient parallel algorithms; that is, of algorithms that perform *work* (total number of operations performed by the available processors) which is close to the number of operations performed by the best known sequential algorithm. Designing efficient parallel algorithms for shortest path problems constitutes a major open problem in parallel computing. One possible reason for the lack of such algorithms could be that most of the emphasis so far has been put on obtaining very fast (i.e. NC) algorithms. However, in almost all practical situations, where the number of available processors p is fixed and much smaller than the sizes of the problems at hand, the primary goal is to have a work-efficient (rather than very fast) parallel algorithm, since (in any case) the running time will be dominated by the work divided by p. Moreover, this seems to be of particular importance if such algorithms can be shown to have other practical merits (e.g. simplicity, ease of implementation).

An important variant of the shortest path problem is the *single-source* or *shortest path tree* problem: given G as above and a distinguished vertex $s \in V$, called the *source*, the problem is to find shortest paths from s to every other vertex in G. (It is well-known that these shortest paths form a tree rooted at s [1].) The single-source shortest path problem has efficient sequential solutions, especially when G has nonnegative edge weights. In this case, the problem can be solved by Dijkstra's algorithm in $O(m+n\log n)$ time using the Fibonacci heap or another priority queue data structure with the same resource bounds [2, 4, 6]. If in addition G is planar, then the problem can be solved optimally in $O(n)$ time [11].

In this paper we consider the single-source shortest path problem in planar digraphs with nonnegative real edge weights. Despite much effort, no sublinear time, work-optimal parallel algorithm has been devised even for this case. The best previous algorithm is due to Cohen [3] and runs in $O(\log^4 n)$ time using $O(n^{3/2})$ work on an EREW PRAM. There are two cases where the work is better. Both cases, however, require edge weights to be nonnegative integers and in one case the algorithm is not deterministic. More specifically, in [13] an $O(\text{polylog}(n)\log L)$-time, $O(n)$-processor *randomized* EREW PRAM algorithm was given, where L is the largest (integral) edge weight of G. In [11], a deterministic parallel algorithm was given that runs in $O(n^{2/3}\log^{7/3} n(\log n+\log D))$ time using $O(n^{4/3}\log n(\log n+\log D))$ work, where D is the sum of the integral edge weights. All of the above algorithms use (in one way or another) sophisticated data structures which make them difficult to implement.

In this paper we present a simple, easily implementable, parallel algorithm for the single-source shortest path problem on a planar digraph G with nonnegative real edge weights. By compromising on parallel running time, we achieve a (deterministic) algorithm which in terms of work-efficiency improves upon the previous algorithms. More precisely, our algorithm runs in $O((n^{2\epsilon}+n^{1-\epsilon})\log n)$ time and performs $O(n^{1+\epsilon}\log n)$ work on an EREW PRAM, for any $0<\epsilon<1/2$. For instance, a choice of $\epsilon=1/3$ improves the bounds in [11] by a logarithmic factor at least, while a choice of $\epsilon=1/4$ improves the work in [3] by a factor of $n^{1/4}$. The work of our algorithm can be further improved to $O(n^{1+\epsilon})$, if the sequential algorithm of [11] is used as a subroutine. However, we cannot claim that this version of the algorithm is easily implementable.

Like previous planar single-source shortest path algorithms, our algorithm is based on a so-called region decomposition of G [5], coupled with a reduction of the problem to a collection of shortest path problems on the regions of G. Given a region decomposition, our algorithm mainly consists in the concurrent application of Dijkstra's sequential single-source shortest path algorithm to the regions of the graph, followed by a final application of a simple parallel version of Dijkstra's algorithm to an auxiliary (non-planar) graph constructed using the shortest path information computed in the regions. By suitable copying of the edges of the graph, concurrent reading and writing can be avoided. For computing the region decomposition presupposed by our shortest path algorithm, we also give an explicit EREW PRAM implementation of the algorithm in [5]. This implementation (slightly more complicated than that of the shortest path algorithm) computes a specific representation of the region decomposition as required for the EREW PRAM implementation of the shortest path algorithm.

It is worth noting that the only routines needed by our algorithms are: (i) Dijkstra's algorithm (sequential and parallel version) implemented via any elementary heap data structure (e.g. binary heap); (ii) standard implementations of parallel prefix computations and sorting; and (iii) the parallel planar separator algorithm of Gazit and Miller [7] whose explicit EREW PRAM implementation is given in Section 4.

Since the main advantage of our algorithm is its simplicity which makes it easy to implement and presumably also efficient in practice (i.e. capable of giving good speedups in parallel machines with a modest number of processors), we intend to incorporate the algorithm into a library of basic PRAM algorithms and data structures, called PAD [10], currently under development.[1]

The rest of the paper is organized as follows. In the next section we give definitions and state some preliminary results about separators and decompositions of planar graphs. Our planar single-source shortest path algorithm is given in Section 3, while Section 4 presents the implementation details for obtaining in parallel the region decomposition needed for the shortest path algorithm. Concluding remarks are offered in Section 5.

2 Preliminaries

Let for the remainder of this paper $G = (V, E)$ be a directed planar graph with nonnegative, real edge weights, $n = |V|$ vertices and $m \leq 3n - 6 = O(n)$ edges. In the following, when we speak about separator properties of G, we are referring to the *undirected version* of G obtained by ignoring the direction of the edges. When we speak of shortest paths, however, we take the direction of edges into account.

Definition 2.1 *A* separator *of a graph $H = (V_H, E_H)$ is a subset C of V_H whose removal partitions V_H into two (disjoint) subsets A and B such that any path from a vertex in A to a vertex in B contains at least one vertex from C.*

Lipton and Tarjan [14] showed that planar graphs have small separators.

[1]The goal of PAD is to provide an organized collection of basic parallel algorithms and data structures, and to investigate the use of the PRAM as a high-level parallel programming model. The experiments so far are encouraging and many basic PRAM algorithms have been implemented (e.g. prefix computations, merge-sort, list ranking, Euler tour, tree contraction, parallel 2-3 trees).

Theorem 2.1 (Planar separator theorem) *Let $G = (V, E)$ be an n-vertex planar graph with nonnegative costs on its vertices summing up to one. Then, there exists a separator S of G which partitions V into two sets V_1, V_2, such that $|S| = O(\sqrt{n})$ and each of V_1, V_2 has total cost at most $2/3$.*

We shall call such a separator S, *a $\frac{1}{3}$-$\frac{2}{3}$ separator of G*. The cost of a subset V' of V, denoted by $wt(V')$, is defined as the sum of the costs of the vertices of V'.

Definition 2.2 *([5]) A* region decomposition *of a graph G is a division of the vertices of G into* regions*, such that each vertex is either* interior*, belonging to exactly one region, or* boundary*, and shared among at least two regions. For any integer $1 \leq r \leq n$, an r*-division *is a region decomposition of G into $\Theta(n/r)$ regions such that each region has at most $c_1 r$ vertices and at most $c_2\sqrt{r}$ boundary vertices, for some constants c_1 and c_2.*

By recursively applying the planar separator theorem, Frederickson [5] gave a sequential $O(n \log n)$ time algorithm for computing an r-division. Our single-source shortest path algorithm – like many others, see e.g. [11] – is based on Frederickson's r-division of a planar graph. An explicit parallel implementation of Frederickson's approach for computing an r-division, in a specific representation necessary for our shortest path algorithm, is given in Section 4. This implementation is based on recursive applications of the optimal parallel algorithm of Gazit and Miller [7] for finding a $\frac{1}{3}$-$\frac{2}{3}$ separator in a planar graph, whose explicit implementation is also given.

The other two main subroutines used by our algorithm are: (a) Dijkstra's sequential algorithm (see for instance [1]). We shall denote a call of the algorithm on a digraph H with source vertex s as Seq-Dijkstra(s, H). (b) A parallel version of Dijkstra's algorithm [4], applied to a digraph $G' = (V', E')$, and running in time $O(m'/p + n' \log n')$ using $p \leq m'/n'$ EREW PRAM processors, where $n' = |V'|$ and $m' = |E'|$.

The parallelization of Dijkstra's algorithm, called *parallel Dijkstra*, is straightforward, and obtained by doing distance label updates in parallel. The idea is as follows. Let each of the p processors have a private heap supporting insert and decrease-key operations in constant time, and find and delete-min in $O(\log n)$ time, all in worst-case [2, 4]. Assume that a vertex of minimum tentative distance has been selected and broadcast to the p processors before the start of the next iteration. The adjacency list of the selected vertex is divided into p equal sized segments, such that the distance labels of the adjacent vertices can be updated in parallel in $O(d/p)$ time, d being the degree of the selected vertex. Each processor inserts (or decreases the key of) the vertices it has updated in its private heap, and selects, again from its private heap, a vertex of minimum distance label. By a prefix-minimum operation the processors collectively determine the vertex of globally minimum distance label for the next iteration. The selected vertex is removed from all the heaps in which it is present, and the next iteration can start. It is easily verified that the algorithm runs on the EREW PRAM in the stated time bound, and is work-optimal for $p \leq m'/(n' \log n')$. The algorithm is easy to implement: the heaps are local to each processor, so a sequential implementation can be reused, the only parallel operation needed being the prefix-minimum computation. We shall denote a call of the parallel Dijkstra algorithm on G' with source vertex s as Par-Dijkstra(s, G').

It should be noted that any heap (e.g. a binary heap), with $O(\log n)$ worst-case time for any heap operation, suffices for our purposes. As we shall see in Section 3, the work performed, $O(m' \log n')$, by such an implementation of parallel Dijkstra is asymptotically smaller than the work performed by the other steps of our algorithm (because $m' = O(n)$).

3 The planar shortest path algorithm

In this section we present our parallel algorithm for solving the single-source shortest path problem on a planar digraph G with nonnegative edge weights. We assume that G is provided with an r-division (see Definition 2.2). In Section 4 we will show how such an r-division can be found.

Let $s \in V$ be the source vertex. Our algorithm works as follows. Inside every region compute, for every boundary vertex v, a shortest path tree rooted at v. These single-source computations are done concurrently using Dijkstra's sequential algorithm. For the region containing s an additional single-source computation starting at s is performed, if s is not a boundary vertex. Then, G is contracted to a graph G' having as vertices the source vertex s and all boundary vertices of the decomposition of G, and having edges between any two boundary vertices belonging to the same region (of G) with weight equal to their distance inside the region (if a path does not exist, the corresponding edge weight is set to ∞). Furthermore, there are edges from s to the boundary vertices of the region containing s, say R_1, with weight equal to their distance from s in R_1. In G' a single-source shortest computation is performed, using the parallel Dijkstra algorithm, producing shortest paths from s to all other vertices of G', that is, to all boundary vertices of G. Finally, the shortest paths and distances from s to the rest of the vertices in G (i.e. to all the interior vertices of the regions) are computed in parallel, using for each (interior) vertex the shortest path information obtained for the boundary vertices of the region it belongs to. The implementation details of our algorithm follow.

Algorithm Planar single-source shortest path.

Input: A weighted planar digraph $G = (V, E)$, a distinguished source vertex $s \in V$, and an r-division of G into regions R_i, $1 \leq i \leq t$, $t = O(n/r)$. Let $V(R_i)$ (resp. $B(R_i)$) be the vertex set (resp. boundary vertex set) of R_i. Let $B = \bigcup_{1 \leq i \leq t} B(R_i)$ be the set of all boundary vertices and let $B(v)$, for $v \in B$, denote the set of regions to which the boundary vertex v belongs. W.l.o.g. assume that $s \in V(R_1)$. (If s is a boundary vertex, then pick R_1 arbitrarily from the regions to which s belongs.) The r-division is computed by the algorithm given in Section 4, and is provided with the following information. Each set $B(R_i)$ is represented as an array, and every interior vertex (i.e. a vertex in $V(R_i) - B(R_i)$) has a label denoting the region it belongs to. There is also an array containing all boundary vertices $v \in B$, and for each such v there is an array of length $|B(v)|$ containing the regions for which v is a boundary vertex. Each boundary vertex $u \in B(R_i)$ has a pointer to its position in the array $B(u)$. All adjacent vertices of a boundary vertex $v \in B$ that belong to the same region are assumed to form a consecutive segment of vertices in the adjacency list of v. Finally, every vertex $u \in B(R_i)$ has two pointers to u's adjacency list, pointing to the first and the last vertex in the (consecutive) segment of vertices that belong to R_i.

Remark: The segmentation of the adjacency lists of the boundary vertices allows a processor to be associated with each boundary vertex of R_i and thus avoiding concurrent read or write operations in cases where a boundary vertex belongs to many regions.

Output: A shortest path tree in G rooted at s. The shortest path tree is returned in arrays $D[1:n]$ and $P[1:n]$. The distance from s to v is stored in $D[v]$ and the parent of v in the shortest path tree is stored in $P[v]$.

Method:

1. INITIALIZATION

Comment: We make $|B(R_i)|$ copies of every region R_i. This is needed to avoid concurrent memory accesses in Step 2. Let R_i^k denote the k-th copy of region R_i which will be associated with the k-th boundary vertex $v_i^k \in B(R_i)$. With every $v \in V(R_i)$, a distance (resp. parent) array $D_v[1:|B(R_i)|]$ (resp. $P_v[1:|B(R_i)|]$) is associated. For boundary vertex $v_i^k \in B(R_i)$ the k-th entry $D_v[k]$ (resp. $P_v[k]$) will be used when a shortest path from v_i^k to v is computed.

for all $1 \le i \le t$ **do in parallel**
    for all $1 \le k \le |B(R_i)|$ **do in parallel**
        for all $v \in V(R_i) - B(R_i)$ **do in parallel**
            Make a copy of the adjacency list of v and add it to R_i^k;
            $D_v[k] = \infty$; $P_v[k] = null$;
        od
        for all $v \in B(R_i)$ **do in parallel**
            Make a copy of the segment of the adjacent vertices of v belonging to R_i,
                and add it to R_i^k;
            $D_v[k] = \infty$; $P_v[k] = null$;
        od
    od
od

2. SHORTEST PATHS INSIDE REGIONS

Comment: For each boundary vertex v_i^k of R_i, a single-source shortest path problem is solved in R_i using Dijkstra's sequential algorithm. Each time, during the execution of the algorithm, if a boundary vertex v_i^j of R_i is selected, then only the segment of its adjacent vertices belonging to R_i is scanned.

for all $1 \le i \le t$ **do in parallel**
    for all $v_i^k \in B(R_i)$ **do in parallel**
        Run Seq-Dijkstra(v_i^k, R_i^k);
    od
od

Comment: After this step, in every region R_i, the distance from each boundary vertex $v_i^k \in B(R_i)$ to each $u \in V(R_i)$ is stored in $D_u[k]$, and a pointer to the parent of u in the shortest path tree rooted at v_i^k is stored in $P_u[k]$.

3. Shortest path tree inside R_1

Comment: If s is not a boundary vertex, solve the single-source shortest path problem inside R_1 with source vertex s, resulting in a distance (resp. parent) array $D^1[v]$ (resp. $P^1[v]$), for all $v \in V(R_1)$.

3.01 **if** $s \notin B(R_1)$ **then** run Seq-Dijkstra(s, R_1), resulting in arrays $D^1[\cdot]$ and $P^1[\cdot]$;

4. Contract G

Comment: Contract G to a graph G' having the source vertex s and all boundary vertices of G as its vertices. For any two boundary vertices v_i^k and v_i^j belonging to the same region R_i there is an edge in G' from v_i^k to v_i^j with weight equal to their distance in R_i. If s is not a boundary vertex, then add edges from s to all boundary vertices of R_1 with weights equal to the distances found in Step 3. The single-source shortest path problem is then solved on G' with source s, using the parallel Dijkstra algorithm, resulting in a distance (resp. parent) array $D'[1 : |V'|]$ (resp. $P'[1 : |V'|]$), where $D'[v]$ stores the distance from s to v in G' and $P'[v]$ stores a pointer to the parent of v in the shortest path tree in G' rooted at s. After this step the distance from s to each boundary vertex of G has been computed.

4.01 $V' = (\bigcup_{1 \le i \le t} B(R_i)) \bigcup \{s\}$; $E' = \emptyset$;
4.02 **for all** $1 \le i \le t$ **do in parallel**
4.03   **for all** pairs $v_i^k, v_i^j \in B(R_i)$ **do in parallel**
4.04   Add edge (v_i^k, v_i^j) to E' with weight equal to $D_{v_i^j}[k]$;
4.05   **od**
4.06 **od**
4.07 **if** $s \notin B(R_1)$ **then**
4.08   **for all** $v_1^k \in B(R_1)$ **do in parallel**
4.09   Add edge (s, v_1^k) to E' with weight equal to $D^1[k]$;
4.10   **od**
4.11 $G' = (V', E')$;
4.12 Run Par-Dijkstra(s, G'), resulting in arrays $D'[\cdot]$ and $P'[\cdot]$;

Comment: The adjacency list representation of E' (Steps 4.04 and 4.09) is constructed as follows. An array of size $|B(R_i)|$ is associated with each boundary vertex $v_i^k \in B(R_i)$; the edge (v_i^k, v_i^j) is stored in the j-th position of this array. Now recall that vertex $v = v_i^k$ belongs to different regions. Using the array $B(v)$ of the regions to which v belongs, an array containing the edges of E', in which all edges adjacent to v form a consecutive segment, can be constructed by a prefix computation. Note that this representation of E' may contain multiple edges, namely in the case where boundary vertices v_i^k and v_i^j both belong to the same regions, but this does not affect neither the correctness nor the complexity of running the parallel Dijkstra algorithm in Step 4.12. (The latter is true, because every edge in G' belongs only to one region.) Moreover, each time a pointer $P'[v]$ is updated, $v \in V'$, we store together with the parent vertex of v the region to which the edge $(P'[v], v)$ belongs. This allows us in Step 5 to recover the parent pointers for the required shortest path tree in G.

5. Final Step

Comment: A shortest path tree T_s in G rooted at s is now computed as follows.

For each interior vertex $u \in V(R_i) - B(R_i)$, of a region R_i, scan through its distance array and find the boundary vertex v_i^k which minimizes the sum of the distance from s to v_i^k (as computed in Step 4) and the distance from v_i^k to u (as computed in Step 2). The parent of u, $P[u]$, in T_s is the parent of u in the shortest path tree in R_i rooted at v_i^k. These computations, concerning the interior vertices of the regions, are done in Steps 5.10-5.13.

For each boundary vertex $v_i^k \in B(R_i)$ we look up its parent $v_j^l = P'[v_i^k]$ in the shortest path tree T'_s in G' rooted at s. If the edge (v_j^l, v_i^k) belongs to R_i (this information was saved by the parallel Dijkstra algorithm in Step 4), and v_j^l is not the source vertex, then the s-v_i^k distance is simply the s-v_i^k distance in G', and the parent of v_i^k in T_s is the parent of v_i^k in the shortest path tree in R_i rooted at v_j^l which is stored in $P_{v_i^k}[l]$. Note that if the parent of v_i^k in T'_s happens to be the source vertex s, and s is not a boundary vertex, then the required distance and parent of v_i^k in T_s are those stored in $D^1[v_i^k]$ and $P^1[v_i^k]$, respectively, as computed in Step 3. These computations, concerning the boundary vertices, are done in Steps 5.14-5.22.

Finally, in the case where s is not a boundary vertex, the distance and parent information computed so far for the interior vertices in R_1 may not be correct, because $D[u]$, for $u \in V(R_1) - B(R_1)$, stores the weight of a (shortest) s-u path passing through at least one boundary vertex and the actual shortest s-u path may stay entirely in R_1. This is rectified by updating $D[u]$ (resp. $P[u]$) to $D^1[u]$ (resp. $P^1[u]$) in the case where $D^1[u] < D[u]$. These computations, regarding the interior vertices of R_1, are done in Steps 5.23-5.27.

A preprocessing step is necessary in order to avoid concurrent memory accesses. To avoid concurrent reading of the array D' in Step 5.11, $|V(R_i) - B(R_i)|$ copies of each value $D'[v_i^k]$ has to be made for each boundary vertex v_i^k (Steps 5.01-5.05). To avoid concurrent reading of the parent pointers in array P' in Step 5.15, a copy of $P'[v]$ is made for each of the $|B(v)|$ regions to which the boundary vertex $v \in B$ belongs (Steps 5.06-5.08).

When Step 5 is completed, distances and parent pointers for all vertices $v \in V$ are stored in arrays $D[1:n]$ and $P[1:n]$ as required.

5.01 **for all** $1 \le i \le t$ **do in parallel**
5.02   **for all** $v_i^k \in B(R_i)$ **do in parallel**
5.03   Make $|V(R_i) - B(R_i)|$ copies of $D'[v_i^k]$, and
            let $D'_u[v_i^k]$ denote the u-th copy of $D'[v_i^k]$ for $u \in V(R_i) - B(R_i)$;
5.04   **od**
5.05 **od**
5.06 **for all** $v \in B$ **do in parallel**
5.07   Make $|B(v)|$ copies of $P'[v]$, and let $P'_i[v]$ denote the copy of $P'[v]$ for region R_i;
5.08 **od**
5.09 **for all** $1 \le i \le t$ **do in parallel**
5.10   **for all** $u \in V(R_i) - B(R_i)$ **do in parallel**
5.11 $D[u] = \min_{v_i^k \in B(R_i)}\{D'_u[v_i^k] + D_u[k]\}$;
5.12 $P[u] = P_u[k]$;
5.13   **od**
5.14   **for all** $v_i^k \in B(R_i)$ **do in parallel**
5.15   Let $v_j^l = P'_i[v_i^k]$;
5.16   **if** edge (v_j^l, v_i^k) belongs to R_i **then**

5.17 **if** $v_j^l = s$ and $s \notin B(R_1)$ **then**
5.18 $D[v_i^k] = D^1[v_i^k]$; $P[v_i^k] = P^1[v_i^k]$;
5.19 **else**
5.20 $D[v_i^k] = D'[v_i^k]$; $P[v_i^k] = P_{v_i^k}[l]$;
5.21 **od**
5.22 **od**
5.23 **if** $s \notin B(R_1)$ **then**
5.24 **for all** $u \in V(R_1) - B(R_1)$ **do in parallel**
5.25 **if** $D^1[u] < D[u]$ **then**
5.26 $D[u] = D^1[u]$; $P[u] = P^1[u]$;
5.27 **od**

End of algorithm.

Theorem 3.1 *The single-source shortest path problem in an n-vertex planar digraph, with nonnegative edge weights, can be solved in $O((r + n/\sqrt{r}) \log n)$ time using $O(n\sqrt{r} \log n)$ work on the EREW PRAM.*

Proof. We first argue about the correctness of the algorithm. We claim that the shortest paths from s to all boundary vertices in G have been correctly computed after Step 4. A shortest s-v path in G, where v is a boundary vertex, consists of a sequence of shortest subpaths each one belonging to a region of G. A path can enter and leave a region R only through the boundary vertices of R. Hence, computing shortest paths between any two boundary vertices v_1, v_2 in R and then substituting the shortest v_1-v_2 path in R by an edge (v_1, v_2) with weight equal to their distance in R, resulting in graph G', preserves shortest paths from s to every boundary vertex in G. But these are exactly the shortest paths computed in Step 4, and hence the claim is true.

We now claim that Step 5 computes correct shortest paths from s to every vertex in G. This is true for the boundary vertices, as shown above, except for the updating of the parent pointers whose correctness can be easily verified by the description of Step 5. Now, let u be an interior vertex of a region R. Clearly, to find the shortest s-u path in G, it suffices to find the boundary vertex v of R which minimizes the sum of the s-v distance in G (computed correctly in Step 4) and the v-u distance in R (computed in Step 2). (If s is a boundary vertex, then for some regions the former distance is zero.) This is exactly the computation performed in Steps 5.09-5.22. A special handling must be done in the case where s, belonging to region R_1, is not a boundary vertex of R_1. Then, it can be easily verified that the shortest s-u path, where u is an interior vertex of R_1, either stays entirely in R_1, or passes through at least one boundary vertex of R_1. The former path is computed in Step 3, while the latter one in Steps 5.09-5.22, as described above. Clearly, the path of minimum weight between these two paths is the required shortest s-u path in G. This computation is performed by Steps 5.23-5.27. Hence, the (second) claim is also true.

From the description of the algorithm, it is clear that all steps can be done without concurrent read or write. The complexity of the algorithm is as follows. In Step 1, $O(\sqrt{r})$ copies of $O(r)$ edges are made within each region, using a prefix computation. Hence, Step 1 takes $O(\log n)$ time and $O(n\sqrt{r})$ work. The shortest path computations in Step 2 take $O(r \log r)$

time and require $O(n\sqrt{r}\log r)$ work. One additional single-source shortest path computation may be be needed in Step 3 taking $O(r \log r)$ time. The contraction of the graph in Step 4 results in a graph of $O((n/r)\sqrt{r}) = O(n/\sqrt{r})$ vertices and $O((\sqrt{r})^2(n/r)) = O(n)$ edges, on which the single-source shortest path problem is solved in parallel in $O((n/\sqrt{r})\log n)$ time and $O(n \log n)$ work, using the parallel Dijkstra algorithm. Finally, in Step 5, copying the D' values takes $O(n\sqrt{r})$ work, and copying the P' values takes $O(n/\sqrt{r})$ work, since the total size of the lists $B(v)$ is $O(n/\sqrt{r})$; both copying operations take $O(\log n)$ time. The remainder of Step 5 can be done in constant time and $O(n)$ work. Hence, the total time taken by the algorithm is $O(r \log r + (n/\sqrt{r})\log n)$, and the total work performed is $O(n\sqrt{r}\log n)$. □

By letting $r = n^{2\epsilon}$, for any $0 < \epsilon < 1/2$, we have:

Theorem 3.2 *On an n-vertex planar graph the single-source shortest path problem can be solved in $O(n^{2\epsilon}\log n + n^{1-\epsilon}\log n)$ time and $O(n^{1+\epsilon}\log n)$ work.*

Choosing either $\epsilon = 1/4$ or $\epsilon = 1/3$ we get:

Corollary 3.1 *On an n-vertex planar graph the single-source shortest path problem can be solved in $O(n^{3/4}\log n)$ time and $O(n^{5/4}\log n)$ work, or in $O(n^{2/3}\log n)$ time and $O(n^{4/3}\log n)$ work.*

The work bound of the above results can be improved by a logarithmic factor, if we substitute the calls of the sequential Dijkstra algorithm in Step 2 with the linear-time algorithm for planar digraphs [11].

Corollary 3.2 *On an n-vertex planar graph the single-source shortest path problem can be solved on an EREW PRAM (i) in $O(n^{2\epsilon}\log n + n^{1-\epsilon}\log n)$ time and $O(n^{1+\epsilon})$ work; (ii) $O(n^{3/4}\log n)$ time and $O(n^{5/4})$ work; (iii) in $O(n^{2/3}\log n)$ time and $O(n^{4/3})$ work.*

4 Obtaining the region decomposition in parallel

In this section we present an explicit EREW PRAM implementation of the algorithm in [5] for finding an r-division of a planar graph G. The main procedure is an algorithm for finding a separator in G. A simple and optimal parallel algorithm for the latter problem was given by Gazit and Miller [7]. Their algorithm is a clever parallelization of the sequential approach by Lipton and Tarjan [14], and runs in $O(\sqrt{n}\log n)$ time using $O(n)$ work on a CRCW PRAM.

We start by giving the implementation on an EREW PRAM of the algorithm in [7], running in $O(\sqrt{n}\log n)$ time and performing $O(n \log n)$ work. We also include a proof of correctness by reproving and simplifying some lemmata used in [7]. Then, in Section 4.2, we give the implementation of the algorithm for finding the r-division. (For simplicity, we relax in the following the constant in the size of the separator.)

4.1 The Gazit-Miller separator algorithm

In order to better understand how the Gazit-Miller algorithm works, we have to recall the Lipton-Tarjan approach.

Let $G = (V, E)$ be an embedded planar graph. The Lipton-Tarjan algorithm starts by choosing an arbitrary vertex $s \in V$ and then performing from s a BFS (breadth first search) in G. The vertices of V are assigned a *level* numbering (with s having level 0) w.r.t. the level they belong to in the BFS tree constructed. Let $V(\ell)$ be the set of vertices at level ℓ. The crucial property of BFS is that every $V(\ell)$ is a separator of G. Let ℓ_1 be the *middle level*, i.e. $wt(\bigcup_{\ell<\ell_1} V(\ell)) < 1/2$, but $wt(\bigcup_{\ell\leq\ell_1} V(\ell)) \geq 1/2$. Consequently, $wt(\bigcup_{\ell>\ell_1} V(\ell)) < 1/2$. If $|V(\ell_1)| = O(\sqrt{n})$, then the algorithm stops since $V(\ell_1)$ is clearly the required separator. Otherwise, there are levels $\ell_0 \leq \ell_1$ (*first cut*) and $\ell_2 > \ell_1$ (*last cut*) such that $|V(\ell_0)| \leq \sqrt{n}$, $|V(\ell_2)| \leq \sqrt{n}$, $\ell_2 - \ell_0 \leq \sqrt{n}$, and ℓ_0 (resp. ℓ_2) is the largest (resp. smallest) such level. Removal of the first and last cuts partitions V into three sets: $A = \bigcup_{\ell<\ell_0} V(\ell)$, $B = \bigcup_{\ell_0<\ell<\ell_2} V(\ell)$, and $C = \bigcup_{\ell>\ell_2} V(\ell)$. If $wt(B) \leq 2/3$, then the required separator is $S = V(\ell_0) \cup V(\ell_2)$, V_1 is the largest of A, B, C, and V_2 is the union of the remaining two (smaller) sets. However, if $wt(B) > 2/3$, then B has to be further split. Since $wt(A) + wt(C) < 1/3$, it suffices to find a separator S' of B with $O(\sqrt{n})$ vertices such that each part into which B is separated has cost at most $2/3$. For if we have it, then the required separator S is $V(\ell_0) \cup V(\ell_2) \cup S'$ and $|S| = O(\sqrt{n})$, V_1 is the larger part of B, and V_2 is the union of A, C and the smaller part of B. Clearly, both V_1 and V_2 will have cost at most $2/3$.

To construct S', remove from G all vertices in A and C (along with their incident edges) and add to the resulting graph the vertex s with edges from s to all vertices in $V(\ell_0 + 1)$. Call the new graph G_B. The crucial property that gives the required size for S' is that G_B has a spanning tree T of diameter $\leq 2\sqrt{n}$. Let E_T be the edges of T and let E_T^* be their corresponding dual edges in the dual graph $G_B^* = (V_B^*, E_B^*)$ of G_B. Then, the set of edges $E_B^* - E_T^*$ (i.e. the duals of the non-tree edges of T) form a spanning tree of G_B^*. By observing that every non-tree edge $e = (u, v)$ of T forms a cycle, say $C(e)$, with the unique u-v path in T and by working on T^*, we can compute for each $C(e)$ its size as well as the total cost of the vertices which are strictly inside $C(e)$. This information can be computed in $O(n)$ time by performing a bottom-up traversal of T^*. It is not hard to see that there exists a cycle $C(e)$ which is the required separator S'.

The difficulty in parallelizing the above approach is the computation of the BFS tree rooted at (an arbitrary vertex) s: either one has to pay in time ($O(n)$) resulting in a parallel algorithm with actually no speedup, or one has to pay in work (close to $O(n^3)$) which makes the parallel algorithm highly work-inefficient. In order to avoid the expensive BFS computation, Gazit and Miller proposed a different partitioning of V into levels. Their approach is summarized as follows: perform a normal BFS, but if at some level there are only a few vertices then "augment" its size by adding more vertices into it. This so-called *augmented BFS* must be done in a way such that all augmented levels are connected between them, otherwise G_B may not have a small diameter. Connectedness is achieved by taking augmentation vertices in preorder from a spanning tree of G.

By the description of Lipton-Tarjan algorithm, it suffices to find the levels ℓ_0 (first cut), ℓ_1 (middle level) and ℓ_2 (last cut). The problem of finding a separator S' in G_B can be solved by choosing one of the following approaches: (a) a straightforward parallelization of the Lipton-Tarjan approach which takes $O(\sqrt{n})$ time and $O(n)$ work, since T and T^* have depth $O(\sqrt{n})$; (b) a fast parallelization of approach (a) in $O(\log n)$ time and $O(n)$ work using parallel tree contraction [9]; (c) the approach described in [15] and which also takes $O(\log n)$ time and $O(n)$ work. For our purposes, the approach (a) is the most appropriate.

Hence, the bulk of the work in the Gazit-Miller algorithm is the computation of the three levels ℓ_0, ℓ_1 and ℓ_2. (Note that these levels may not be the same as those computed by the Lipton-Tarjan algorithm; however, they will have the same crucial properties.) The implementation details of the algorithm follow.

Algorithm Gazit-Miller.
Input: Embedded planar graph $G = (V, E)$ with nonnegative costs on its vertices summing up to one.
Output: A partition of V into three sets V_1, V_2, S, such that S is a separator of G, $|S| = O(\sqrt{n})$ and each of V_1, V_2 has total cost at most 2/3.
Method:
1. Run the INITIALIZATION PHASE;
2. Run PHASE A;
3. **if** $|V(\ell_1)| \leq 4\sqrt{n}$ **then** stop
   else run PHASE B;
4. **if** $wt(V_B) \leq (2/3)$ **then** stop
   else find a $\frac{1}{3}$-$\frac{2}{3}$ separator in G_B;

End of algorithm.

In view of the preceding discussion, it suffices to describe the implementation of the three phases in the first three steps of the algorithm. The INITIALIZATION PHASE is as follows:

INITIALIZATION PHASE:
1. Initialize arrays $A[1:n]$ and $A'[1:\sqrt{n}]$;
2. Find a spanning tree T of G rooted at an arbitrary vertex s;
3. Compute the preorder numbering, $pre(\cdot)$, of the vertices in T;
4. **for all** $v \in T$ **do in parallel** $A[pre(v)] = v$;

END OF INITIALIZATION PHASE.

In Step 4 of the INITIALIZATION PHASE, the vertices of G are stored into an array A w.r.t. their preorder number. In the following, A will be used as a stack data structure.

Lemma 4.1 *The* INITIALIZATION PHASE *of Gazit-Miller algorithm runs in $O(\log^2 n)$ time using $O(n \log n)$ work on an EREW PRAM.*

Proof. It is clear that Step 1 can be done in $O(\log n)$ time and $O(n)$ work, while Step 4 in $O(1)$ time and $O(n)$ work. Step 3 takes $O(\log n)$ time and $O(n)$ work, using parallel tree contraction [9]. Step 2 takes $O(\log^2 n)$ time and $O(n \log n)$ work using the (very simple) algorithm of [16]. (Note that for the latter step, there exists an $O(\log n \log^* n)$-time, $O(n)$-work EREW PRAM algorithm [8]; however, this algorithm does not seem to be as simple as the algorithm of [16].) $\square$

PHASE A finds levels ℓ_0 and ℓ_1 and is implemented as follows.

PHASE A:

01. $\ell = 0$; $V(0) = \{s\}$;
02. **while** $wt(\bigcup_{\ell' < \ell} V(\ell')) < 1/2$ **do** (* main loop *)
03. NEXT-LEVEL$(\ell, V(\ell))$;
04. $\ell = \ell + 1$; $j = 2\ell + 1$; (* Now ℓ represents the next level *)
05. **while** $|V(\ell)| < j$ **do** (* augment level ℓ *)
06. Pop the top $\sqrt{n}$ elements from A and store them temporarily into an array A';
07. Mark in A' those vertices that belong to any level $i < \ell$;
08. Using a parallel prefix computation, remove the marked vertices from A' and count the number, R, of the remaining vertices;
09. $\rho = \min\{j - |V(\ell)|, R\}$;
10. Add the first ρ elements of A' to $V(\ell)$ and push the remaining $R - \rho$ to the top of A;
11. **od** (* augment level ℓ *)
12. **if** $|V(\ell)| = j$ **then** $\ell_0 = \ell$;
13. **od** (* main loop *)
14. $\ell_1 = \ell$;

END OF PHASE A.

The procedure NEXT-LEVEL is a straightforward parallelization of one BFS step.

Procedure NEXT-LEVEL$(\ell, V(\ell))$

1. $V(\ell + 1) = V(\ell)$;
2. Replace every $u \in V(\ell + 1)$ by the list of its adjacent vertices;
3. Remove from $V(\ell + 1)$ all vertices belonging to any level $i < \ell + 1$, using a parallel prefix computation;
4. Remove all duplicate vertices, using sorting;

End of procedure.

Lemma 4.2 PHASE A *of Gazit-Miller algorithm runs in* $O(\sqrt{n} \log n)$ *time using* $O(n \log n)$ *work on an EREW PRAM.*

Proof. We first argue about the correctness of PHASE A. We claim that for any $0 \leq \ell \leq \ell_1$, the subgraph induced on $\bigcup_{0 \leq i \leq \ell} V(i)$ is connected: a vertex v is added in $V(i)$, $0 < i \leq \ell$, either after the execution of the NEXT-LEVEL procedure and hence it is adjacent to a vertex in $V(i-1)$ (in which case the claim is true), or v has been picked from the array A. The fact that the vertices in A are stored w.r.t. their preorder number in T and that $v \in V(i)$ imply that all vertices with smaller preorder number than v are already in $\bigcup_{0 \leq j \leq i} V(j)$, and hence v is adjacent to at least one vertex in $\bigcup_{0 \leq j \leq i} V(j)$ (i.e. its parent in T). Note also that due to Step 07 there is no edge that crosses two or more levels implying that every $V(\ell)$ is a separator.

The total number of vertices at the end of PHASE A are $|\bigcup_{0 \leq \ell \leq \ell_1} V(\ell)| \geq 1 + \sum_{\ell=1}^{\ell_1} (2\ell + 1) = (\ell_1 + 1)^2$. On the other hand, $|\bigcup_{0 \leq \ell \leq \ell_1} V(\ell)| = k \leq n$. Consequently, the total number of levels, ℓ_1, computed in PHASE A is at most $\sqrt{k} - 1 < \sqrt{n}$. This also implies that $|V(\ell_0)| = 2\ell_0 + 1 \leq 2\sqrt{k} - 1 < 2\sqrt{n}$, and from the description of the algorithm it is clear that this is the largest such level ℓ_0.

Let us now discuss the resource bounds of PHASE A. We claim that the total number of iterations of the main-loop is bounded by $2\sqrt{n}$. To see this, it suffices to show that the inner while-loop (augment level) is executed at most $2\sqrt{n}$ times overall executions of the main-loop. Consider an iteration of the inner while-loop and call it *proper* if $R < j - |V(\ell)|$. Clearly, there are at most $\sqrt{n}$ proper iterations in total. On the other hand, observe that a non-proper iteration is always the last iteration of the inner while-loop and as a consequence $V(\ell)$ has the required size. Since the total number of levels is $< \sqrt{n}$, the total number of non-proper iterations is also $< \sqrt{n}$. Hence, the claim is true.

It is easy to verify that the dominating step (w.r.t. the resource bounds) in every iteration is the execution of procedure NEXT-LEVEL which takes $O(\log n)$ time and $O(|N(V(\ell))| \log n)$ work on an EREW PRAM, where $N(V(\ell)) = \{u : (u, v) \in E \text{ and } v \in V(\ell)\}$. Consequently, PHASE A runs in $O(\sqrt{n} \log n)$ time using $O(n \log n)$ work on an EREW PRAM. □

PHASE B of Gazit-Miller algorithm computes level ℓ_2 and is implemented as follows.

PHASE B:
1. $\ell = \ell_1$; $k = |\bigcup_{0 \le i \le \ell_1} V(i)|$; $j = \sqrt{n-k}$;
2. **while** $|V(\ell)| > j$ **do**
3. NEXT-LEVEL$(\ell, V(\ell))$;
4. $\ell = \ell + 1$; $j = j - 2$;
5. **od**
6. $\ell_2 = \ell$;

END OF PHASE B.

Lemma 4.3 PHASE B *of Gazit-Miller algorithm runs in* $O(\sqrt{n} \log n)$ *time using* $O(n \log n)$ *work on an EREW PRAM.*

Proof. First observe that every $V(\ell)$, $\ell > \ell_1$, is a separator, because it is a level created by normal BFS (Step 3). It is clear that the total number of iterations, ℓ_2, in PHASE B is bounded by $\sqrt{n-k}$. Moreover, note that such a level ℓ_2 exists (i.e. the while-loop terminates), since otherwise $|V(\ell)| > j$, for all $\ell > \ell_1$, and consequently $|\bigcup_{\ell > \ell_1} V(\ell)| > n - k$, a contradiction. Therefore, $\ell_2 - \ell_0 \le \sqrt{k} + \sqrt{n-k} \le \sqrt{2n}$. Hence, G_B has (a spanning tree of) diameter at most $2\sqrt{2n}$. Moreover, $|V(\ell_2)| = \ell_2 - 2 < \sqrt{n}$ and from the description of the algorithm it is clear that this is the smallest such level ℓ_2. Hence, the correctness is established.

The resource bounds follow from the fact that $\ell_2 < \sqrt{n}$ and the resource bounds of executing procedure NEXT-LEVEL (see Lemma 4.2). □

The following theorem is an immediate consequence of the preceding discussion and Lemmata 4.1, 4.2 and 4.3.

Theorem 4.1 *Let* $G = (V, E)$ *be an* n*-vertex planar graph with nonnegative costs on its vertices summing up to one. Then, a partition of* V *into three sets* V_1, V_2, S*, such that* S *is a separator of* G, $|S| = O(\sqrt{n})$*, and each of* V_1, V_2 *has total cost at most* $2/3$*, can be computed in* $O(\sqrt{n} \log n)$ *time using* $O(n \log n)$ *work on an EREW PRAM.*

4.2 The parallel algorithm for finding an r-division

The algorithm for finding an r-division of a planar graph $G = (V, E)$, in the form required by the planar single-source shortest path algorithm (Section 3), is given below. The algorithm is based on recursive applications of Theorem 4.1.

Algorithm Parallel r-division.
Input: Planar graph $G = (V, E)$, parameter r and constants c_1, c_2.
Output: An r-division of G.
Method:

(* INITIALIZATION *)
1. $R = G$; $B(R) = \emptyset$; $B = \emptyset$;

(* MAIN PART *)
2. **if** $|V(R)| > c_1 r$ **then**
   run the Gazit-Miller algorithm on R with vertex cost $\frac{1}{|V(R)|}$,
   yielding partition $V_1(R), V_2(R)$ and $S(R)$
   else
   if $|B(R)| > c_2\sqrt{r}$ **then**
   run the Gazit-Miller algorithm on R with vertex cost 0
   for each $v \in V(R) - B(R)$ and vertex cost $\frac{1}{|B(R)|}$
   for each $v \in B(R)$, yielding partition $V_1(R), V_2(R)$ and $S(R)$
   else stop;
3. Infer regions R_i, $i = 1, 2$, induced on vertex set $V(R_i) = V_i(R) \cup S(R)$
   and having boundary vertex set $B(R_i) = (B(R) \cap V_i(R)) \cup S(R)$;
4. $B = B \cup B(R_1) \cup B(R_2)$;
5. Using a parallel prefix computation, split the adjacency list of each $v \in B(R)$
   into two parts, one containing the neighbors of v belonging to R_1,
   and the other the neighbors of v belonging to R_2;
6. For $v \in B(R_i)$, $i = 1, 2$, create pointers to the beginning and the end of
   the segment of the neighbors of v that belong to $V(R_i)$;
7. Run the MAIN PART recursively on each R_i, $i = 1, 2$;
8. Create for each $v \in B$ the array $B(v)$ of regions to which v belongs,
   using a parallel prefix computation in the segmented adjacency list of v;
9. For all regions R', and for each $v \in B(R')$, create a pointer to the position of R' in $B(v)$;

End of algorithm.

Theorem 4.2 *An r-division of an n-vertex planar graph G can be computed in $O(\sqrt{n}\log^2 n)$ time using $O(n\log^2 n)$ work on an EREW PRAM.*

Proof. The correctness can be easily verified from the description of the algorithm (see also [5]). Concerning the resource bounds, it can be easily checked that Steps 3, 4, 5, and 8 need $O(\log n)$ time and $O(n)$ work, while Steps 6 and 9 can be done in $O(1)$ time and $O(n)$ work. The dominating step for the resource bounds is Step 2, whose bounds follow from those of Theorem 4.1 and the fact that the depth of the recursion is $O(\log n)$. □

Note that both time and work required to find the r-division is within that of the shortest path algorithm (Theorem 3.2).

5 Final remarks

We presented a sublinear-time, work-efficient parallel algorithm for the single-source shortest path problem on planar digraphs. We believe that the main advantage of our algorithm is its simplicity and ease of implementation. The improvement in the work is based on a suitable choice of the parameters in the region decomposition which reduced the problem to computing a small collection of local shortest path information (inside every region) and then using this in computing global shortest path information from the source to every boundary vertex in the original graph. Coming down to linear work seems to be difficult, however.

It has tacitly been assumed that the input to the separator algorithm is a planar graph with an embedding. This of course begs the question of the existence of a parallel planarity testing and embedding algorithm, or of a parallel separator algorithm not requiring an embedding as part of the input. We are not aware of any parallel algorithm for the latter case. Work-efficient, NC algorithms for the former case have been given in [12, 17], but neither of these algorithms seems to be easily implementable. Designing a simple, easily implementable, parallel algorithm for planarity testing and embedding is an interesting open problem.

Acknowledgement. We are grateful to Hillel Gazit for providing us with [7].

References

[1] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows*. Prentice-Hall, 1993.

[2] Gerth Stølting Brodal. Worst-case efficient priority queues. In *Proc. 7th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 52–58, 1996.

[3] Edith Cohen. Efficient parallel shortest-paths in digraphs with a separator decomposition. In *Proc. 5th Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 57–67, 1993.

[4] James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert E. Tarjan. Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(11):1343–1354, 1988.

[5] Greg N. Frederickson. Fast algorithms for shortest paths in planar graphs with applications. *SIAM Journal of Computing*, 16(6):1004–1022, 1987.

[6] Michael L. Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[7] Hillel Gazit and Gary L. Miller. An $O(\sqrt{n}\log(n))$ optimal parallel algorithm for a separator for planar graphs. Unpublished manuscript, 1987.

[8] Torben Hagerup. Optimal Parallel Algorithms for Planar Graphs. *Information and Computation*, 84:71-96, 1990.

[9] Joseph JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

[10] Christoph W. Kessler and Jesper L. Träff. A library of basic PRAM algorithms and its implementation in FORK. In *Proc. 8th Symposium on Parallel Algorithms and Architectures (SPAA)*, to appear, 1996.

[11] Philip Klein, Satish Rao, Monika Rauch, and Sairam Subramanian. Faster shortest-path algorithms for planar graphs. In *Proc. 26th Symposium on Theory of Computation (STOC)*, pages 27–37, 1994.

[12] Philip N. Klein and John H. Reif. An efficient parallel algorithm for planarity. *Journal of Computer and System Sciences*, 37:190–246, 1988.

[13] Philip N. Klein and Sairam Subramanian. A linear-processor, polylog-time algorithm for shortest paths in planar graphs. In *Proc. 34th Symposium on Foundations of Computer Science (FOCS)*, pages 259–270, 1993.

[14] Richard J. Lipton and Robert Endre Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

[15] Gary L. Miller. Finding small simple cycle separators for 2-connected planar graphs. *Journal of Computer and System Sciences*, 32:265–279, 1986.

[16] Cynthia Phillips. Parallel graph contraction. In *Proc. 1st Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 148–157, 1989.

[17] Vijaya Ramachandran and John H. Reif. Planarity testing in parallel. *Journal of Computer and System Sciences*, 49(3):517–561, 1994.