Journal of Parallel and Distributed Computing **60**, 1103–1124 (2000)
doi:10.1006/jpdc.2000.1646, available online at http://www.idealibrary.com on IDEAL®

A Simple Parallel Algorithm for the Single-Source Shortest Path Problem on Planar Digraphs[1]

Jesper L. Träff

C&C Research Laboratories, NEC Europe Ltd., Rathausallee 10, D-53757 Sankt Augustin, Germany
E-mail: traff@ccrl-nece.technopark.gmd.de

and

Christos D. Zaroliagis

Computer Technology Institute, and Department of Computer Engineering & Informatics, University of Patras, 26500 Patras, Greece
E-mail: zaro@ceid.upatras.gr

Received April 1, 1997; revised December 3, 1999; accepted April 11, 2000

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 $O((n^{2\varepsilon}+n^{1-\varepsilon})\log n)$ time, performing $O(n^{1+\varepsilon}\log n)$ work for any $0<\varepsilon<1/2$. The strength of the algorithm is its simplicity, making it easy to implement and presumable quite efficient in practice. The algorithm improves upon the work of all previous parallel algorithms. Our algorithm is based on a region decomposition of the input graph and uses a well-known parallel implementation of Dijkstra's algorithm. The logarithmic factor in both the work and the time can be eliminated by plugging in a less practical, sequential planar shortest path algorithm together with an improved 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 weigths, 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

[1] This work was partially supported by the EU ESPRIT LTR Project 20244 (ALCOM-IT) and by the DFG Project SFB 124-D6 (VLSI Entwurfsmethoden und Parallelität). Part of this work was done while both authors were with the Max-Planck-Institut für Informatik, Saarbrücken, Germany, and while the second author was with the Department of Computer Science, King's College, University of London, UK.

0743-7315/00 $35.00

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 weigths [1].

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 emphasis has most often been put on obtaining very fast (i.e., NC) algorithms. However, in most 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 the running time in such cases will be dominated by the work divided by p. This is 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. 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, 5, 7]. If, in addition, G is planar, then the problem can be solved optimally in $O(n)$ time [14].

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 [4] and runs in $O(\log^4 n)$ time using $O(n^{3/2})$ work on an EREW PRAM[2]. There are two cases where the work is better. Both cases, however, require edge weights to be integers and in one case the algorithm is not deterministic. More specifically, in [16] an $O(\text{polylog}(n) \log L)$-time, $O(n)$-processor *randomized* EREW PRAM algorithm is given, where edge weights are assumed to be nonnegative integers with L being the largest edge weight of G. In [14], 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 absolute values of the integral edge weights (which may be negative). All of the above algorithms in one way or another use sophisticated data structures which make them difficult to implement. They all require that a plane embedding of the input graph is given.

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 previous algorithms. More precisely, our algorithm runs in $O((n^{2\varepsilon} + n^{1-\varepsilon}) \log n)$ time and performs $O(n^{1+\varepsilon} \log n)$ work on an EREW PRAM, for any $0 < \varepsilon < 1/2$. For instance, a

[2] By PRAM, we refer to the unit-cost PRAM model [11].

choice of $\varepsilon = 1/3$ improves the bounds in [14] by at least a logarithmic factor and the work bound in [4] by a factor of $n^{1/6}$. The bounds of our algorithm can be further improved to $O(n^{2\varepsilon} + n^{1-\varepsilon})$ time and $O(n^{1+\varepsilon})$ work, if the sequential algorithm of [14] as well as the parallel implementation of Dijkstra's algorithm in [3] are used as subroutines. 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 as introduced in [6], 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 (nonplanar) 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 EREW PRAM implementation of the algorithm in [6]. 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. As in previous algorithms it is assumed that a planar embedding of the input graph is given.

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 (segmented) parallel prefix computations and sorting; and (iii) the parallel planar separator algorithm of Gazit and Miller [8], whose explicit EREW PRAM implementation is given in Section 4.

The main advantage of our algorithm is its simplicity which makes it easy to implement, in the sense that its implementation is based on fundamental, well-understood routines (e.g., prefix computations, list ranking, sorting [11]) that are likely to be found in any library of parallel combinatorial algorithms. A possible environment for such an implementation is the PAD library of basic PRAM algorithms and data structures [12], currently under development[3]. It is worth mentioning that for a machine allowing concurrent reading, the algorithm can be considerably simplified in that a lot of copying of the input graph becomes superfluous.

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 given in Section 5. A preliminary version of this work appeared in [21].

[3] 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, and parallel 2-3 trees).

2. PRELIMINARIES

For the remainder of this paper let $G=(V, E)$ be a directed planar graph with nonnegative, real edge weights, $n=|V|$ vertices, and $m \leqslant 3n-6=O(n)$ 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 [17] showed that planar graphs have small separators.

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*. Let $wt(v)$ denote the cost of a vertex $v \in V$. Then the cost of a subset $V' \subseteq V$ will be denoted as $wt(V')=\sum_{u \in V'} wt(u)$.

DEFINITION 2.2 [6]. A region decomposition of a graph G is a division of the vertices of G into regions, such that: (i) each vertex is either interior, i.e., it belongs to exactly one region, or boundary, i.e., it is shared among at least two regions; and (ii) there is no edge between two interior vertices belonging to different regions. For any integer $1 \leqslant r \leqslant n$, 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 [6] 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., [14]—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 work-optimal parallel algorithm of Gazit and Miller [8] 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 [5], applied to a digraph $G'=(V', E')$ and running in time $O(m'p+n' \log n')$ using $p \leqslant m'/n'$ 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. (We assume that the reader is famililar with Dijkstra's algorithm [1].) 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, 5]. 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 into 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 the 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 $p \leqslant m'/(n' \log n')$. 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 path 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 distance 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, $1 \leqslant i \leqslant t$, $t = O(n/r)$. Let $V(R_i)$ (resp. $B(R_i)$) be the vertex set (resp. boundary vertex set) of R_i. Let $C = \bigcup_{1 \leqslant i \leqslant t} B(R_i)$ be the set of *all* boundary vertices and let $C(v)$ for $v \in C$ denote the set of regions to which the boundary vertex v belongs. Without loss of generality 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 input $G=(V, E)$ is represented as a collection of adjacency lists stored in an array A, such that vertices adjacent to the same vertex $u \in V$ form a consecutive segment of the array denoted by $A(u)$. The r-division is computed by the algorithm given in Section 4 and is provided with the following data structures to facilitate the necessary copying of the graph. Each vertex that is interior in a region R_i (i.e., a vertex in $V(R_i) - B(R_i)$) has a label denoting the region it belongs to. The set $B(R_i)$ of boundary vertices of a region R_i is represented as an array. The set C of all boundary vertices is also represented as an array. All adjacent vertices of a boundary vertex $v \in C$ that belong to the same region are assumed to form a consecutive segment of vertices in $A(v)$. Adjacent boundary vertices v' of $v \in C$ that belong to several regions are arbitrarily put into one such segment. Every vertex $v \in B(R_i)$ has two pointers to $A(v)$, pointing to the first and the last vertex in the (consecutive) segment of vertices that belong to R_i. Each $v \in C$ has a pointer to an array $C(v)$ containing all regions for which v is a boundary vertex. Finally, each boundary vertex $v \in B(R_i)$ has a pointer to the position of R_i in the array $C(v)$. The representation of the r-division is illustrated in Fig. 1.

In the case where a boundary vertex belongs to many regions the segmentation of the adjacency lists of the boundary vertices allows a processor for each region to be associated with each such boundary vertex while still avoiding concurrent read or write operations.

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
2. Computation of shortest paths inside regions
3. Computation of shortest path tree inside R_1
4. Contraction of G into G' and shortest path tree computation in G'
5. Computation of shortest path tree in G

End of algorithm.

We now turn to the description of the implementation of each individual step.

1. *Initialization.* We start by making $|B(R_i)|$ copies of every region R_i. This is needed to avoid concurrent memory accesses in Step 2, when we compute shortest paths inside every region. Let R_i^k denote the kth copy of region R_i, $1 \leqslant k \leqslant |B(R_i)|$, which will be associated with the kth boundary vertex v^k of $B(R_i)$. This association is needed in Step 2 when we will perform shortest path tree computations in R_i^k rooted only at v^k. With every $u \in V(R_i)$, two arrays $D_u^i[1:|B(R_i)|]$ and $P_u^i[1:|B(R_i)|]$

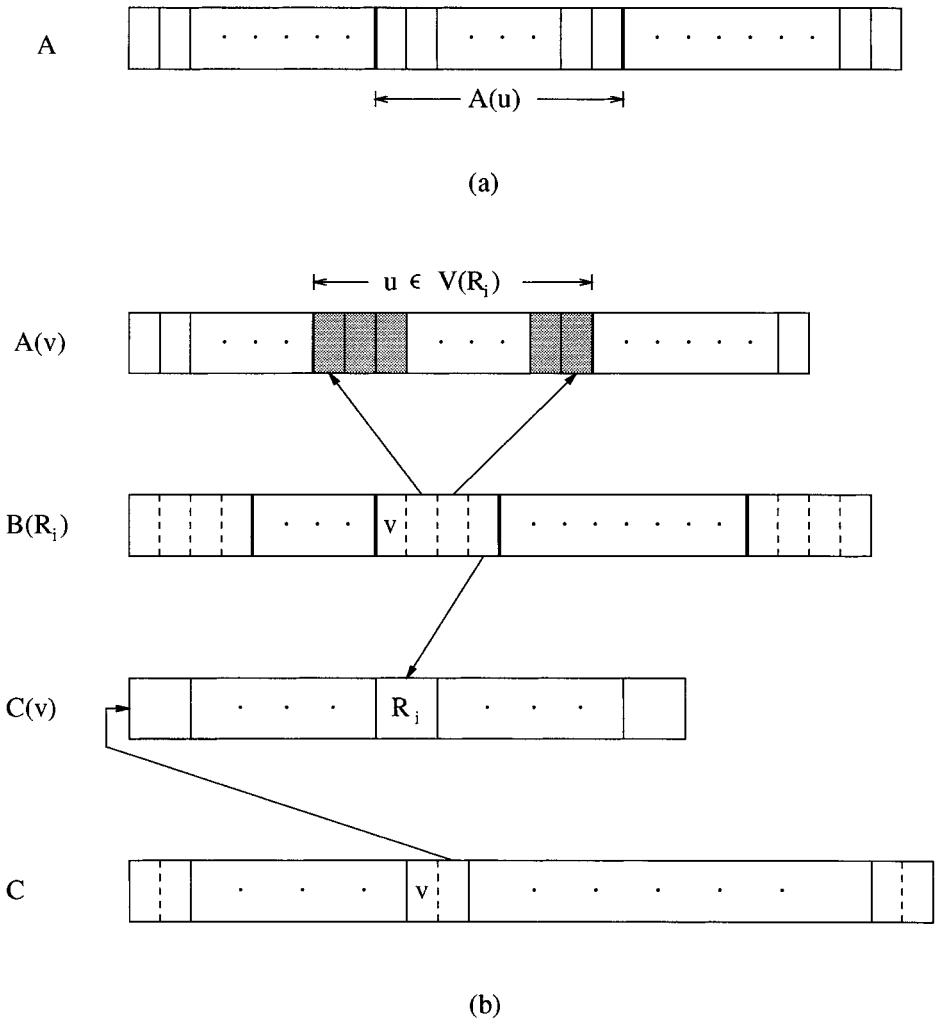

FIG. 1. Data structures needed for the representation of the r-division. (a) All vertices adjacent to $u \in V$ form a consecutive segment $A(u)$ in A. (b) The data structures involving a boundary vertex $v \in B(R_i)$.

are associated. The entry $D_u^i[k]$ stores the distance of a shortest v^k-u path in R_i^k, while the entry $P_u^i[k]$ stores the parent of u in a shortest path tree in R_i^k rooted at v^k.

1.01 **for all** $u \in V(R_i) - B(R_i)$, $1 \leqslant i \leqslant t$, **do in parallel**
1.02 Make $|B(R_i)|$ copies of $A(u)$;
1.03 **od**
1.04 **for all** $v \in B(R_i)$, $1 \leqslant i \leqslant t$ **do in parallel**
1.05 Make $|B(R_i)|$ copies of the segment of $A(v)$ containing the vertices belonging to R_i;
1.06 **od**
1.07 **for all** $u \in V(R_i)$, $1 \leqslant i \leqslant t$ **do in parallel**
1.08 Allocate space for the arrays $D_u^i[1 : |B(R_i)|]$ and $P_u^i[1 : |B(R_i)|]$;
1.09 **od**

1.10 **for all** $u \in V(R_i)$, $1 \leqslant i \leqslant t$, $1 \leqslant k \leqslant |B(R_i)|$ **do in parallel**
1.11 **if** $u = v^k$ **then** $D_u^i[k] = 0$ **else** $D_u^i[k] = \infty$;
1.12 $P_u^i[k] = null$;
1.13 **od**

Steps 1.01–1.03 and 1.04–1.06 are carried out by segmented prefix computations. We will discuss the implementation of Steps 1.01–1.03. The implementation of Steps 1.04–1.06 is done in a similar way. An array is constructed which for each vertex stores the number of copies to be made, namely $|B(R_i)|$ for $u \in V(R_i) - B(R_i)$. By prefix summation the size of a new vertex array A' is computed. This array is divided into segments $A'(u)$ which, for each $u \in V(R_i) - B(R_i)$, can hold the $|B(R_i)|$ copies of u. A pointer to $A(u)$ is stored in the first position of $A'(u)$. By a segmented prefix computation over A' each of these pointers are broadcast to each copy of u. In a similar fashion the adjacency array A is copied into an array A'' such that each $u \in V(R_i) - B(R_i)$ has $|B(R_i)|$ copies of each of its adjacent vertices. Let $A'(u)[k]$ denote the position in $A'(u)$ that stores the pointer for the kth copy, u^k, of u. Now the lth adjacent vertex of u^k in $A''(u)$ can be found by offsetting $A'(u)[k]$ by $|B(R_i)|\,(l-1)+k$ positions. Thus, each copy of u can access its own copy of u's adjacent vertices without any concurrent reading.

2. *Computation of shortest paths inside regions.* For each boundary vertex v^k of R_i, a single-source shortest path problem with source v^k is solved in R_i^k using Dijkstra's sequential algorithm. During the execution of the algorithm, each time a boundary vertex v^j, $j \neq k$, of R_i is selected, only the segment of its adjacent vertices belonging to R_i is scanned.

2.01 **for all** $v^k \in B(R_i)$, $1 \leqslant i \leqslant t$, $1 \leqslant k \leqslant |B(R_i)|$ **do in parallel**
2.02 Run Seq-Dijkstra(v^k, R_i^k);
2.03 **for all** $u \in V(R_i^k)$ **do in parallel**
2.04 Store the distance of a shortest v^k-u path in $D_u^i[k]$;
2.05 Store the parent of u in the shortest path tree rooted at v^k in $P_u^i[k]$;
2.06 **od**
2.06 **od**

3. *Computation of shortest path tree inside* R_1. 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 $D^1[1 : |V(R_1)|]$ (resp. $P^1[1 : |V(R_1)|]$); $\forall x \in V(R_1)$, $D^1[x]$ stores the distance of a shortest s-x path in R_1, and $P^1[x]$ stores a pointer to the parent of x in the shortest path tree in R_1 rooted at s.

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. *Contraction of* G *into* G' *and shortest path tree computation in* G'. Contract G to a graph $G' = (V', E')$ having the source vertex s and all boundary vertices of G as its vertices. For any two boundary vertices v^k and v^j belonging to the same region R_i there is an edge in G' from v^k to v^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, $\forall x \in V'$, $D'[x]$ stores the distance from s to x in G' and $P'[x]$ stores a pointer to the parent of x 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' = C ∪ {s}; E' = ∅;
4.02  for all 1 ≤ i ≤ t, 1 ≤ k, j ≤ |B(R_i)| do in parallel
4.03        for all pairs v^k, v^j ∈ B(R_i) do in parallel
4.04              Add edge (v^k, v^j) to E' with weight equal to D^i_{v^j}[k];
4.05        od
4.06  od
4.07  if s ∉ B(R_1) then
4.08        for all v ∈ B(R_1) do in parallel
4.09              Add edge (s, v) to E' with weight equal to D^1[v];
4.10        od
4.11  G' = (V', E');
4.12  Run Par-Dijkstra(s, G'), resulting in arrays D'[·] and P'[·]
```

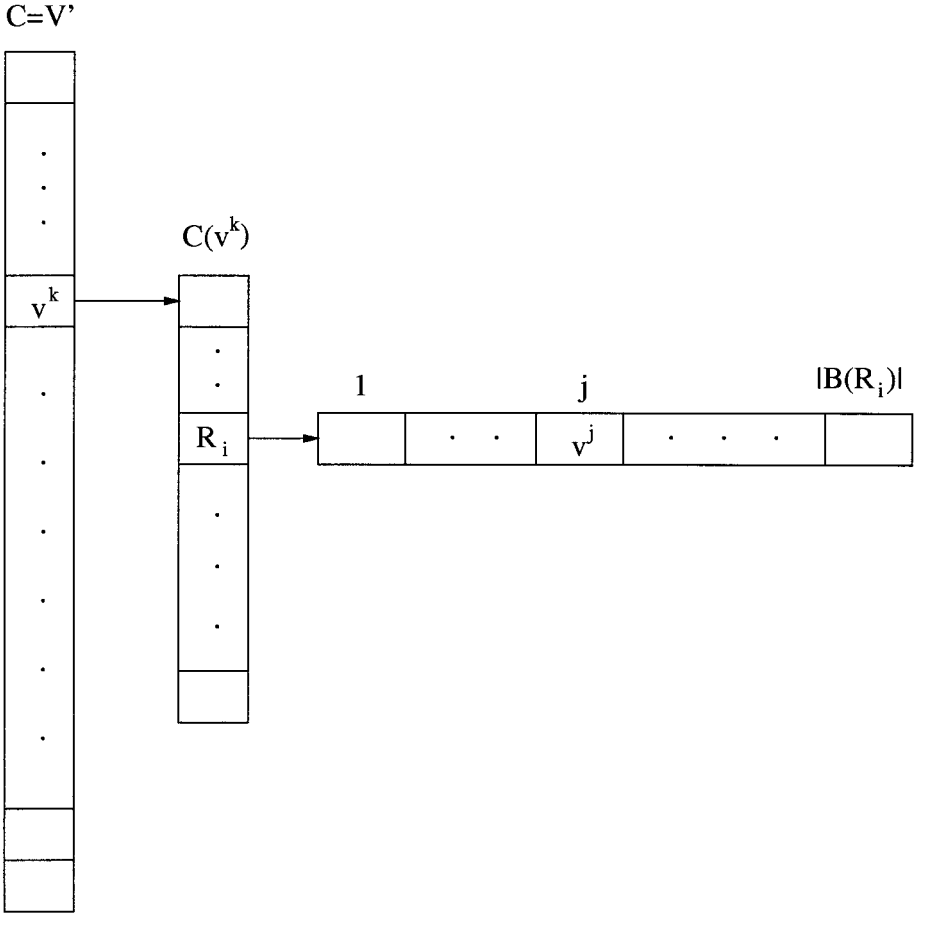

FIG. 2. Creation of E'.

The adjacency list representation of E' (Steps 4.04 and 4.09) is constructed as follows. Consider each array $C(v^k)$ which contains all different regions to which a boundary vertex $v^k \in C$ belongs. To each entry R_i of this array, we associate an array of size $|B(R_i)|$ (see Fig. 2). Store the edge (v^k, v^j) in the jth position of this array. Now, E', represented as an array in which all edges adjacent to v^k form a consecutive segment, can be constructed by a prefix computation on the union of the $C(v^k)$ arrays along with their associated arrays. If s is not a boundary vertex, it is trivial to augment E' with a new segment containing all edges (s, v) such that $v \in B(R_1)$. Note that E' may contain multiple edges, namely in the case where boundary vertices v^k and v^j both belong to the same regions, but this affects neither the correctness nor the complexity of running the parallel Dijkstra algorithm in Step 4.12. Each time a pointer $P'[x]$ is updated, $x \in V'$, we store together with the parent vertex of x the region to which the edge $(P'[x], x)$ belongs. This allows us in Step 5 to recover the parent pointers for the required shortest path tree in G.

5. *Computation of the shortest path tree in G.* To compute the requested shortest path tree T_s in G rooted at s, we have to find the shortest path distance in G from s to every vertex (boundary or interior) $x \in V$ and store it in $D[x]$, as well as the parent pointer for every x in the shortest path tree in G rooted at s and store it in $P[x]$.

To compute distances and parent pointers from s to interior vertices we do the following. For each interior vertex $u \in V(R_i) - B(R_i)$, of a region R_i, scan through its distance array D^i_u to find the boundary vertex $v^k \in B(R_i)$ which minimizes the sum of the distance from s to v^k (as computed in Step 4) and the distance from v^k to u (as computed in Step 2). Store this minimum distance in $D[u]$. 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^k, except for the special case where $s \in V(R_1) - B(R_1)$ and u is an interior vertex of R_1. These computations are done in Steps 5.10–5.13, except for the handling of the special case which is done in Steps 5.21–5.25 and discussed below.

Distances from s to every boundary vertex have been computed in Step 4 and stored in the array D'. Hence, all we have to do for the boundary vertices is to compute parent pointers. For each boundary vertex $v^k \in B(R_i)$, we look up its parent $w^l = P'[v^k]$ in the shortest path tree T'_s in G' rooted at s and check whether the edge (w^l, v^k) belongs to R_i or not (this information was saved by the parallel Dijkstra algorithm in Step 4). If the edge (w^l, v^k) does not belong to R_i, then we do nothing, because the shortest s-v^k path does not pass through R_i. Otherwise, if the edge belongs to R_i, we distinguish between the cases $w^l \neq s$ and $w^l = s$. In the former case, we have that $D[v^k] = D'[v^k]$ and the parent of v^k in T_s is the parent of v^k in the shortest path tree in R_i rooted at w^l which is stored in $P^i_{v^k}[l]$. In the latter case, we further check if s is a boundary vertex or not. If $s \in B(R_1)$, then the edge (s, v^k) belongs to R_1 and hence we return to the former case, i.e., $D[v^k] = D'[v^k] (= D^1[v^k])$ and $P[v^k] = P^1_{v^k}[l]$. If $s \notin B(R_1)$, $D[v^k] = D'[v^k] (= D^1[v^k])$ as before, but the parent of v^k in T_s is equal to $P^1[v^k]$ (as computed in Step 3). These computations, concerning the boundary vertices, are done in Steps 5.14–5.19.

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 5.21–5.25.

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 $D'[v^k]$ have to be made for each boundary vertex v^k (Steps 5.01–5.04). To avoid concurrent reading of the parent pointers in array P' in Step 5.16, a copy of $P'[v]$ is made for each of the $|C(v)|$ regions to which the boundary vertex $v \in C$ belongs (Steps 5.05–5.08). This copying of D' and P' is done by segmented prefix computations.

5.01 **for all** $v^k \in B(R_i)$, $1 \leqslant i \leqslant t$, $1 \leqslant k \leqslant |B(R_i)|$ **do in parallel**
5.02 Make $|V(R_i) - B(R_i)|$ copies of $D'[v^k]$;
5.03 Let $D'_u[v^k]$ denote the uth copy of $D'[v^k]$ for $u \in V(R_i) - B(R_i)$;
5.04 **od**
5.05 **for all** $v \in C$ **do in parallel**
5.06 Make $|C(v)|$ copies of $P'[v]$;
5.07 Let $P'_i[v]$ denote the copy of $P'[v]$ for region R_i;
5.08 **od**
5.09 **for all** $1 \leqslant i \leqslant t$ **do in parallel**
5.10 **for all** $u \in V(R_i) - B(R_i)$ **do in parallel**
5.11 $D[u] = \min_{v^k \in B(R_i)} \{D'_u[v^k] + D^i_u[k]\}$;
5.12 $P[u] = P^i_u[k]$;
5.13 **od**
5.14 **for all** $v^k \in B(R_i)$ **do in parallel**
5.15 $D[v^k] = D'[v^k]$;
5.16 Let $w^l = P'_i[v^k]$;
5.17 **if** edge (w^l, v^k) belongs to R_i **then**
5.18 **if** $w^l = s$ and $s \notin B(R_1)$ **then** $P[v^k] = P^1[v^k]$ **else** $P[v^k] = P^i_{v^k}[l]$;
5.19 **od**
5.20 **od**
5.21 **if** $s \notin B(R_1)$ **then**
5.22 **for all** $u \in V(R_1) - B(R_1)$ **do in parallel**
5.23 **if** $D^1[u] < D[u]$ **then**
5.24 $D[u] = D^1[u]$; $P[u] = P^1[u]$;
5.25 **od**

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 start with the correctness of the algorithm. We first 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_i only through the boundary vertices of R_i. Hence, computing shortest paths between any two boundary vertices v^k, v^j in R_i, or between s and any $w \in B(R_1)$, and then substituting the shortest v^k-v^j (resp. s-w) path in R_i (resp. R_1) by an edge (v^k, v^j) (resp. (s, w)) with weight equal to their distance in R_i (resp. R_1), resulting in graph G', preserves shortest paths from s to every boundary vertex in G. But these are exactly the shortest path computations performed in Step 4, based on the shortest paths computed inside every region in Steps 2 and 3. To complete the proof of the claim it remains to show that shortest paths inside every region are correctly computed in Steps 2 and 3, which reduces to showing that the arbitrary placement to only one segment in $A(v)$ of adjacent boundary vertices v' of $v \in C$ that may belong to several regions does not affect correct shortest path computation between v and v' in the relevant regions. We will show this for Step 2 (a similar proof holds for Step 3). Consider the edge (v, v'). Since G is planar, this edge can belong to at most two regions, say R_i and R_j. Let v' be arbitrarily put into the segment regarding R_i, i.e., (v, v') is assumed to belong only to R_i. Since (v, v') belongs only to R_i and R_j, it suffices to show that the arbitrary placement of (v, v') in R_i does not affect the shortest v-v' path that stays entirely within the subgraph induced by $V(R_i) \cup V(R_j)$ and that this information is available at the beginning of Step 4. Such a shortest v-v' path will be the minimum weight path between two paths: a shortest v-v' path in R_i and a shortest v-v' path in R_j. The former path is computed in Step 2 by the shortest path tree computation in R_i (which includes (v, v')) with root v. The latter path is computed in Step 2 by the shortest path tree computation in R_j with root v. By the construction of G', the weights of both paths will be present in G' as two (v, v') edges, one carrying the weight of the former path and the other carrying the weight of the latter path. As mentioned in Step 4, this affects neither the correctness nor the complexity of the parallel Dijkstra algorithm. Hence, the shortest paths computations in Step 4 are correct and consequently 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_i. Clearly, to find the shortest s-u path in G, it suffices to find the boundary vertex v of R_i which minimizes the sum of the s-v distance in G (computed correctly in Step 4) and the v-u distance in R_i (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 5.09–5.20. 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 5.09–5.20, 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 5.21–5.25. Hence, the second claim is true and consequently correctness has been established.

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 $O((n/r)\, r \sqrt{r}) = O(n \sqrt{r})$ work.

In Step 2, $O(\sqrt{r})$ concurrent single-source shortest path computations are performed in each region which has $O(r)$ vertices and edges. Each single-source shortest path computation takes $O(r \log r)$ time and work. Hence, Step 2 takes over all regions $O(r \log r)$ time and requires $O((n/r)\sqrt{r}\,(r \log r)) = O(n\sqrt{r} \log r)$ work. One additional single-source shortest path computation may be needed in Step 3 taking $O(r \log r)$ time and work. To construct the contracted graph $G' = (V', E')$ in Step 4, all we have to do is to generate E', since $V' = C \cup \{s\}$. As explained in the algorithm, creation of E' involves a prefix computation on an array of total size

$$\sum_{v \in C,\, R_i \in C(v)} |C(v)| \cdot |B(R_i)| = O\left(\sum_{v \in C} |C(v)| \sqrt{r}\right) = O(|C| \sqrt{r})$$
$$= O((n/\sqrt{r})\sqrt{r}) = O(n).$$

Thus, creating E' takes $O(\log n)$ time and $O(n)$ work. The resulting graph G' has $O(n/\sqrt{r})$ vertices and $O(n)$ edges. On G' 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 $C(v)$ is $O(n/\sqrt{r})$; both copying operations take $O(\log n)$ time. The loop in Steps 5.10–5.13 needs $O(\log r)$ time and $O(n\sqrt{r})$ work. 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 $r = n^{2\varepsilon}$, for any $0 < \varepsilon < 1/2$, we have:

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

Both bounds of the above result 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 [14] and the call of parallel Dijkstra algorithm in Step 4 with the new implementation given in [3], which runs in $O(n)$ (resp. $O(n \log(m/n))$) time and $O(m \log n)$ work on a CREW (resp. EREW) PRAM. Since $m = O(n)$ in our case, we have the following.

THEOREM 3.3. *On an n-vertex planar graph the single-source shortest path problem can be solved in $O(n^{2\varepsilon} + n^{1-\varepsilon})$ time and $O(n^{1+\varepsilon})$ work on an EREW PRAM.*

4. OBTAINING THE REGION DECOMPOSITION IN PARALLEL

In this section we present an explicit EREW PRAM implementation of the algorithm in [6] for finding an r-division of a planar graph G. The main procedure is an algorithm for finding a separator in G. A simple, work-optimal parallel algorithm for the latter problem was given by Gazit and Miller [8]. Their algorithm is a clever parallelization of the sequential approach by Lipton and Tarjan [17] and runs in $O(\sqrt{n} \log n)$ time using $O(n)$ work on a CRCW PRAM. Randomized NC algorithms

for the problem of finding a small separator have been given in [9, 13]. For our purposes the slower, but simple and easily implementable algorithm is appropriate.

We start by giving the implementation on an EREW PRAM of the algorithm in [8], 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 [8]. 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. (W.l.o.g., we assume that G is connected.) The Lipton–Tarjan algorithm starts by choosing an arbitrary vertex $s\in V$ and then performing from s a breadth first search (BFS) in G. The vertices of V are assigned a *level* numbering (with s having level 0) with respect to the level they belong to in the BFS tree constructed. Let $\ell_{\max}$ be the maximum level computed and 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 $wt(\bigcup_{\ell\leqslant\ell_1} V(\ell))\geqslant 1/2$. Consequently, $wt(\bigcup_{\ell>\ell_1} V(\ell))\leqslant 1/2$. If $|V(\ell_1)|=O(\sqrt{n})$, then the algorithm stops since $V(\ell_1)$ is clearly the required separator. Otherwise, there must exist levels $\ell_0\leqslant\ell_1$ (*first cut*) and $\ell_2>\ell_1$ (*last cut*) such that $|V(\ell_0)|\leqslant\sqrt{n}$, $|V(\ell_2)|\leqslant\sqrt{n}$, $\ell_1-\ell_0<\sqrt{n}$, and $\ell_2-\ell_1<\sqrt{n}$. (Note that ℓ_2 may be the empty level $\ell_{\max}+1$.) 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 $wt(B)\leqslant 2/3$, then the required separator is $S=V(\ell_0)\cup V(\ell_2)$, V_1 is the heaviest of A, B, C, and V_2 is the union of the remaining two (lighter) 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 $V(\ell_0)\cup V(\ell_2)\cup S'$, $|S|=O(\sqrt{n})$, V_1 is the heavier part of B, and V_2 is the union of A, C and the lighter part of B. Clearly, both V_1 and V_2 will have cost at most 2/3.

To find S', construct a planar graph G_B as follows. Delete from G all vertices in $C\cup V(\ell_2)$ (along with their incident edges) and shrink all vertices in $A\cup V(\ell_0)$ to a single vertex s' with weight zero, i.e., replace all vertices in A with s' and add edges from s' to every vertex in $V(\ell_0+1)$. It is easy to verify that G_B has a spanning tree T of diameter at most $2(\ell_2-\ell_0-1)+1\leqslant 2\sqrt{n}$: s' is the root of T and all vertices in G_B have BFS distance at most $(\ell_2-\ell_0-1)$ from s'. Then, the required separator S' can be found by working on the dual graph of G_B. The bound on $|S'|$ comes from the bound in the diameter of T.

The problem of computing efficiently in parallel a separator S' in G_B of size $O(\sqrt{n})$ has been solved in [18]. More specifically such an S' can be found in $O(\log n)$ time and $O(n)$ work on an EREW PRAM if G_B is provided with a spanning tree of diameter $O(\sqrt{n})$.

The main difficulty in parallelizing the Lipton–Tarjan 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 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 each augmented level is connected only to the next; otherwise, G_B may not have a spanning tree of small diameter and/or each level of the augmented BFS will not be a separator. That is, for $0 \leqslant \ell < \ell_{\max}$, there exists at least one edge (x, y) such that $x \in V(\ell)$ and $y \in V(\ell + 1)$, and there is no edge (w, z) such that $w \in V(\ell)$ and $z \in V(\ell + i)$, for $2 \leqslant i \leqslant \ell_{\max}$. This specific connectedness is achieved by taking augmentation vertices in preorder from a spanning tree of G.

It follows by the above discussion that there are two main problems to be solved: (i) find the levels ℓ_0 (first cut), ℓ_1 (middle level), and ℓ_2 (last cut); and (ii) find a spanning tree of diameter $O(\sqrt{n})$ in G_B.

The bulk of the work in the Gazit–Miller algorithm is in the solution of the first problem. The solution of the second problem is a by-product.

The three levels ℓ_0, ℓ_1, and ℓ_2 computed by the Gazit–Miller approach may not be the same as those computed by the Lipton–Tarjan algorithm; however, they will have similar properties. That is, $\ell_0 \leqslant \ell_1 < \ell_2$, $wt(\bigcup_{\ell < \ell_1} V(\ell)) < 1/2$, $|V(\ell_0)| \leqslant 2\sqrt{n}$, $|V(\ell_2)| \leqslant \sqrt{n}$, $\ell_1 - \ell_0 < \sqrt{n}$, and $\ell_2 - \ell_1 < \sqrt{n}$.

The high-level description of the Gazit–Miller algorithm is similar to that of the Lipton–Tarjan approach. PHASE A performs the augmented BFS and computes levels ℓ_0 and ℓ_1, while PHASE B computes another set of levels and also finds level ℓ_2.

Algorithm Gazit–Miller.
Input: Embedded (connected) 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, $|S| \leqslant 7\sqrt{n}$, and each of V_1, V_2 has total cost at most 2/3.
Method:
01. Run the INITIALIZATION PHASE;
02. Run PHASE A;
03. **if** $|V(\ell_1)| \leqslant 7\sqrt{n}$ **then**
04. $S = V(\ell_1)$; $V_1 = \bigcup_{\ell < \ell_1} V(\ell)$; $V_2 = V - V_1 - S$;
05. **else** run PHASE B;
06. Let A, B, C, and G_B as defined previously;
07. **if** $wt(B) \leqslant (2/3)$ **then**
08. $S = V(\ell_0) \cup V(\ell_2)$; $V_1 = \max_{wt}\{A, B, C\}$; $V_2 = (A \cup B \cup C) - V_1$;
09. **else** find a $\frac{1}{3}$–$\frac{2}{3}$ separator in G_B yielding partition
10. W_1, W_2, S', $|S'| \leqslant 4\sqrt{n}$, and $wt(W_1) \geqslant wt(W_2)$;
11. $S = V(\ell_0) \cup V(\ell_2) \cup S'$; $V_1 = W_1$; $V_2 = A \cup C \cup W_2$;
End of algorithm.

We will now describe the implementation details of the three phases. The INITIALIZATION PHASE is as follows:

INITIALIZATION PHASE:
1. Allocate 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 in an array A according to their preorder number. The array A will be used as a stack. At any given instant vertices will be stored in the segment $A[i:n]$, $1 \leqslant i \leqslant n$, with $A[i]$ being the stack top. After initialization lower numbered vertices will appear nearer the top. Vertices will pop off A in blocks, and by the preorder numbering the vertices in any such block will be connected to vertices in some previously popped block.

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

Proof. In Step 1 space for arrays A and A' is allocated and start addresses are broadcast to all processors. This takes $O(\log n)$ time and $O(n)$ work. Step 2 takes $O(\log^2 n)$ time and $O(n \log n)$ work using the very simple algorithm of [19]. (Note that for the latter step, there exists an $O(\log n \log^* n)$-time, $O(n)$-work EREW PRAM algorithm [10]; however, this algorithm does not seem to be as simple as the algorithm of [19].) Step 3 takes $O(\log n)$ time and $O(n)$ work, using parallel tree contraction [11]. Finally, Step 4 uses $O(1)$ time and $O(n)$ work. ∎

The implementation of PHASE A, which performs the augmented BFS and finds levels ℓ_0 and ℓ_1, is given next.

PHASE A:
01. $\ell = 0$; $V(0) = \{s\}$;
02. **while** $wt(\bigcup_{\ell' < \ell} V(\ell')) < 1/2$ **do** (* main loop *)
03. NEXT-LEVEL(ℓ, $V(\ell)$);
04. $\ell = \ell + 1$; $j = 2\ell + 1$;
05. **while** $|V(\ell)| < j$ **and** $A \neq \varnothing$ **do** (* augmented level ℓ *)
06. Pop the top $\sqrt{n}$ vertices from A and store these in array A';
07. Mark the vertices in A' that belong to any previous level $i < \ell$;
08. Remove the marked vertices from A' by parallel prefix computations 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 $R - \rho$ back onto A;
11. **od** (* augment level ℓ *)
12. **if** $|V(\ell)| < 2\sqrt{n} + 1$ **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(ℓ, $V(\ell)$)
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 the Gazit–Miller algorithm computes a set of levels* $0 \leqslant \ell \leqslant \ell_1$, ℓ_1 *being the last level, along with a special level* $\ell_0 \leqslant \ell_1$ *such that:*

(i) *Each* $V(\ell)$ *is a separator and the subgraph induced by* $\bigcup_{0 \leqslant i \leqslant \ell} V(i)$ *is connected.*

(ii) $wt(\bigcup_{\ell < \ell_1} V(\ell)) < 1/2$, $\ell_1 - \ell_0 < \sqrt{n}$, *and* $|V(\ell_0)| \leqslant 2\sqrt{n}$.

(iii) *Any spanning tree of the subgraph induced by* $\bigcup_{\ell_0 \leqslant i \leqslant \ell_1} V(i)$ *has diameter at most* $2\sqrt{n} - 1$.

(iv) *The whole computation takes* $O(\sqrt{n} \log n)$ *time using* $O(n \log n)$ *work on an EREW PRAM.*

Proof. (i) A vertex v is added to $V(i)$, $0 < i \leqslant \ell$, either because the NEXT-LEVEL procedure is executed or because it is picked from the array A. In the former case, v is adjacent to a vertex in $V(i-1)$. In the latter case, since vertices are chosen from A in preorder number, v must be adjacent to at least one vertex in $\bigcup_{0 \leqslant j \leqslant i} V(j)$ (i.e., its parent in T). Hence, in either case the subgraph induced on $\bigcup_{0 \leqslant i \leqslant \ell} V(i)$ is connected. Due to Steps 07 and 08 there is no edge that crosses two or more levels so every $V(\ell)$ is a separator.

(ii) The fact that $wt(\bigcup_{\ell < \ell_1} V(\ell)) < 1/2$ follows by the termination condition of the while-loop at Step 02. The total number of vertices k included in the sets $V(\ell)$ at the end of PHASE A is $k = |\bigcup_{0 \leqslant \ell \leqslant \ell_1} V(\ell)| \geqslant 1 + \sum_{\ell=1}^{\ell_1} (2\ell+1) = (\ell_1+1)^2$. On the other hand, $|\bigcup_{0 \leqslant \ell \leqslant \ell_1} V(\ell)| \leqslant n$. Consequently, the total number of levels, ℓ_1, computed in PHASE A is at most $\sqrt{k} - 1 < \sqrt{n}$, which implies that $\ell_1 - \ell_0 < \sqrt{n}$. By the condition in Step 12 it is clear that $|V(\ell_0)| \leqslant 2\sqrt{n}$.

(iii) The diameter of a spanning tree in the subgraph induced by $\bigcup_{\ell_0 \leqslant i \leqslant \ell_1} V(i)$ cannot be more than $\sum_{i=\ell_0}^{\ell_1} |V(i)| = \sum_{i=0}^{\ell_1} |V(i)| - \sum_{i=0}^{\ell_0 - 1} |V(i)| \leqslant n - \ell_0^2$. By the condition in Step 05, we must have that the augmented level $\ell_0 + 1$ has size $|V(\ell_0+1)| \geqslant 2\ell_0 + 3$. Moreover, since ℓ_0 is the last level with $|V(\ell_0)| < 2\sqrt{n} + 1$, it must also hold that $2\ell_0 + 3 \geqslant 2\sqrt{n} + 1$. This implies that $\ell_0 \geqslant \sqrt{n} - 1$. Hence, $\sum_{i=\ell_0}^{\ell_1} |V(i)| \leqslant n - (n - 2\sqrt{n} + 1) = 2\sqrt{n} - 1$.

(iv) We start by bounding the total number of iterations of the main-loop. We claim that this number is bounded by $2\sqrt{n}$. Clearly, if the number of iterations of the inner while-loop (augment level) is 0, then the main-loop will iterate at most $\ell_1 < \sqrt{n}$ times. Hence, it suffices to show that the inner while-loop is executed at most $2\sqrt{n}$ times over all 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, since at that point A has become empty. On the other hand, a nonproper 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 nonproper iterations is also $<\sqrt{n}$. Hence, the claim is true.

The execution of NEXT-LEVEL takes $O(\log n)$ time and $O(|N(V(\ell))| \log n)$ work on an EREW PRAM, where $N(V(\ell)) = \{(u, v) \in E : u \in V(\ell)\}$. Each execution of the inner while loop takes $O(\log n)$ time and $O(\sqrt{n})$ work. Consequently, PHASE A runs in $O(\sqrt{n} \log n)$ time using $O(n \log n)$ work on an EREW PRAM. ▮

We now turn to the implementation of PHASE B. It computes a set of levels (using normal BFS) and also computes the level ℓ_2.

PHASE B:
1. $\ell = \ell_1$; $k = |\bigcup_{0 \leqslant i \leqslant \ell_1} V(i)|$;
2. **while** $|V(\ell)| \geqslant \sqrt{n-k}$ **do**
3. NEXT-LEVEL(ℓ, $V(\ell)$);
4. $\ell = \ell + 1$;
5. **od**
6. **if** $V(\ell_1 + 1) = \varnothing$ **then** $\ell_2 = \ell_1 + 1$ **else** $\ell_2 = \ell$;

END OF PHASE B.

LEMMA 4.3. PHASE B *of the Gazit–Miller algorithm computes a set of levels* $\ell > \ell_1$ *along with a special level* $\ell_2 > \ell_1$ *such that:*

(i) $V(\ell_2)$ *is a separator,* $|V(\ell_2)| < \sqrt{n}$, *and* $\ell_2 - \ell_1 < \sqrt{n}$.

(ii) *The subgraph* G' *induced by* $(\bigcup_{\ell_1 \leqslant i < \ell_2} V(i)) \cup \{s''\}$, *where* s'' *is a special zero-weighted vertex connected to all vertices in* $V(\ell_1)$, *has a spanning tree of diameter at most* $2\sqrt{n} + 1$.

(iii) *The whole computation takes* $O(\sqrt{n} \log n)$ *time using* $O(n \log n)$ *work on an EREW PRAM.*

Proof. (i) If $V(\ell_1 + 1) = \varnothing$, or $V(\ell) = \varnothing$ at some iteration, then there are no additional BFS levels to be created, i.e., we have reached the maximum BFS level. In this case, $V(\ell_2) = \varnothing$ is a trivial separator. Otherwise, each $V(\ell)$, $\ell > \ell_1$, is a separator, since every level is created by Normal BFS (Step 3), and so is $V(\ell_2)$. In every iteration of the while-loop, $V(\ell) \geqslant \sqrt{n-k}$. Hence, the total number of iterations, $\ell_2 - \ell_1$, performed by the while-loop is bounded by $\sqrt{n-k} < \sqrt{n}$. It is also clear by the description of the algorithm that $|V(\ell_2)| < \sqrt{n}$, either because of the terminating condition of the while-loop or because $V(\ell_2)$ is the trivial separator (empty set).

(ii) Vertex s'' can be considered as replacing all vertices in level $\ell_1 - 1$ and below in a way analogous to vertex s' in the Lipton–Tarjan approach. The BFS levels ℓ_1 (created during PHASE A), $\ell_1 + 1, ..., \ell_2$ (created by PHASE B) can alternatively be seen as being produced by performing on G' a BFS rooted at s'', with s'' having level $\ell_1 - 1$. Consequently, all vertices are at a BFS distance of at most

$\ell_2 - \ell_1$ from s''. As in the Lipton–Tarjan approach, the spanning tree of G' is determined by the BFS process, has s'' as its root, and diameter at most $2(\ell_2 - \ell_1) + 1 < 2\sqrt{n} + 1$.

(iii) The resource bounds follow from the fact that the total number of iterations of the while-loop is at most $\sqrt{n}$ and the resource bounds of executing procedure NEXT-LEVEL (see Lemma 4.2). ▮

We are now ready to prove the following theorem.

THEOREM 4.1. *Let $G = (V, E)$ be an n-vertex planar graph with nonnegative costs on its vertices summing up to one. Then, the Gazit–Miller algorithm finds a partition of V into three sets V_1, V_2, S, such that S is a separator of G, $|S| \leqslant 7\sqrt{n}$, and each of V_1, V_2 has total cost at most 2/3. This partition can be computed in $O(\sqrt{n} \log n)$ time using $O(n \log n)$ work on an EREW PRAM.*

Proof. In view of the discussion preceding the Gazit–Miller algorithm, to prove the theorem it suffices to show that (a) the levels ℓ_0, ℓ_1, and ℓ_2 obey properties similar to those computed by the Lipton–Tarjan algorithm, and (b) that the subgraph G_B has a spanning tree of small diameter.

Part (a) follows immediately by Lemmata 4.2(i, ii) and 4.3(i). For part (b), recall that G_B is the graph induced on $(\bigcup_{\ell_0 < i < \ell_2} V(i)) \cup \{s'\}$, where s' has replaced all vertices in $\bigcup_{0 \leqslant i \leqslant \ell_0} V(i)$ and is adjacent to all vertices in $V(\ell_0 + 1)$. Hence, we can consider $V(\ell_0) = \{s'\}$. Compute any spanning tree T in $\bigcup_{\ell_0 \leqslant i \leqslant \ell_1} V(i)$. By Lemma 4.2(iii), its diameter is at most $2\sqrt{n} - 1$. Now, consider all nodes u of T that belong to level $V(\ell_1)$. Each such node u is also the root of a BFS tree, $T'(u)$, generated during PHASE B by the procedure NEXT-LEVEL in the graph induced by $\bigcup_{\ell_1 \leqslant i < \ell_2} V(i)$. In fact, each such u is a child of s'' in the implicit BFS tree construction used in the proof of Lemma 4.3(ii). Attach to each such node u in T the subtree $T'(u)$. Clearly, the resulting tree is a spanning tree of G_B. By Lemmata 4.2(iii) and 4.3(ii), the new tree has diameter not greater than $2\sqrt{n} - 1 + 2\sqrt{n} + 1 = 4\sqrt{n}$.

The worst-case bound on $|S|$ comes from Step 11 and is clearly bounded by $7\sqrt{n}$. The resource bounds follow easily by those of Lemmata 4.1, 4.2, and 4.3 and the fact that computing a spanning tree takes $O(\log^2 n)$ time and $O(n \log n)$ work by the algorithm of [19]. ▮

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:
1. $R = G$; $B(R) = \varnothing$; $C = \varnothing$;
2. SPLIT-REGION(R);

3. For all regions R' and all boundary vertices $v \in B(R')$, create pointers to the beginning and end of the segment of v-adjacent vertices belonging to R';
4. Create for each $v \in C$ the array $C(v)$ of regions to which v belongs, using a parallel prefix computation in the segmented adjacency list of v;
5. For all regions R', and for each $v \in B(R')$, create a pointer to the position of R' in $C(v)$;

End of algorithm.

Procedure SPLIT-REGION(R)

1. **if** $|V(R)| > c_1 r$ **then** (* split region *)
   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 (* split boundary *)
   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 return;
2. Infer regions R_i, $i = 1, 2$, induced by vertex sets $V(R_i) = V_i(R) \cup S(R)$ with boundary vertex sets $B(R_i) = (B(R) \cap V_i(R)) \cup S(R)$;
3. $C = C \cup B(R_1) \cup B(R_2)$;
4. Split the adjacency list of each $v \in B(R)$ by a parallel prefix computation into two parts containing the neighbors of v belonging to R_1, and the neighbors of v belonging to R_2; mark each neighbor vertex with the region to which it belongs; neighbors belonging to both R_1 and R_2 are put (arbitrarily) in the first part;
5. Run SPLIT-REGION(R_i) in parallel for $i = 1, 2$;

End of procedure.

THEOREM 4.2. *An r-division of an n-vertex planar graph G represented as required for the planar shortest path algorithm in Section* 3 *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 [6]). It is also easy to see that the required representation of the r-division is computed. In particular at each level of the recursion the adjacency lists of the boundary vertices are correctly split and the adjacent vertices marked with the region to which they belong. Thus, after the recursion each vertex adjacent to a boundary vertex has been marked with the region to which it finally belongs. The marks are used to create, for each boundary vertex $v \in B(R_i)$ of region R_i, pointers to the beginning and end of the segment of v's adjacency list of vertices belong to R_i. This information is then used to create for each boundary vertex v the array $C(v)$ of regions to which it belongs.

We now turn to the resource bounds. We start with procedure SPLIT-REGION. The depth of the recursion is $O(\log n)$. By Theorem 4.1 each iteration of Step 1 takes $O(\sqrt{n} \log n)$ time and $O(n \log n)$ work. Steps 2, 3, and 4 take $O(\log n)$ time and $O(n)$ work. Hence, procedure SPLIT-REGION, and consequently Step 2 of the

algorithm, takes in total $O(\sqrt{n}\log^2 n)$ time and $O(n\log^2 n)$ work on an EREW PRAM. The other steps of the algorithm are performed by segmented prefix operations (similar to the detailed descriptions given in Section 3) and can be done within $O(\log n)$ time and $O(n)$ work. This concludes the proof of the theorem. ▮

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 seperator 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 [15, 20], 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.

ACKNOWLEDGMENTS

We are indebted to the anonymous referees for several helpful comments and suggestions that improved considerably the presentation of the paper. We are also grateful to Hillel Gazit for providing us with [8].

REFERENCES

1. R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, "Network Flows," Prentice–Hall, New York, 1993.
2. G. S. Brodal, Worst-case efficient priority queues, *in* "Proc. 7th ACM-SIAM Symposium on Discrete Algorithms (SODA)," pp. 52–58, 1996.
3. G. S. Brodal, J. L. Träff, and C. D. Zaroliagis, A parallel priority queue with constant time operations, *J. Parallel Distrib. Comput.* **49** (1998), 4–21.
4. E. Cohen, Efficient parallel shortest-paths in digraphs with a separator decomposition, *J. Algorithms* **21** (1996), 331–357.
5. J. R. Discroll, H. N. Gabow, R. Shrairman, and R. E. Tarjan, Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation, *Comm. Assoc. Comput. Mach.* **31**, 11 (1988), 1343–1354.
6. G. N. Frederickson, Fast algorithms for shortest paths in planar graphs with applications, *SIAM J. Comput.* **16**, 6 (1987), 1004–1022.

7. M. L. Fredman and R. E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, *J. Assoc. Comput. Mach.* **34**, 3 (1987), 596–615.
8. H. Gazit and G. L. Miller, An $O(\sqrt{n}\log(n))$ optimal parallel algorithm for a separator for planar graphs, Unpublished manuscript, 1987.
9. H. Gazit and G. L. Miller, A parallel algorithm for finding a separator in planar graphs, *in* "Proceedings of the 28th IEEE Symposium on Foundations of Computer Science (FOCS)," pp. 238–248, 1987.
10. T. Hagerup, Optimal parallel algorithms on planar graphs, *Inform. and Comput.* **84** (1990), 71–96.
11. J. JáJá, "An Introduction to Parallel Algorithms," Addison–Wesley, Reading, MA, 1992.
12. C. W. Keßler and J. L. Träff, Language and library support for practical PRAM programming, *Parallel Comput.* **25**, 2 (1999), 105–135.
13. P. Klein, On Gazit and Miller's parallel algorithm for planar separators: Achieving greater efficiency through random sampling, *in* "5th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)," pp. 43–49, 1993.
14. P. Klein, S. Rao, M. Rauch, and S. Subramanian, Faster shortest-path algorithms for planar graphs, *in* "Proceedings of the 26th Symposium on Theory of Computation (STOC)," pp. 27–37, 1994.
15. P. N. Klein and J. H. Reif, An efficient parallel algorithm for planarity, *J. Comput. System Sci.* **37** (1988), 190–246.
16. P. N. Klein and S. Subramanian, A linear-processor, polylog-time algorithm for shortest paths in planar graphs, *in* "34th Symposium on Foundations of Computer Science (FOCS)," pp. 259–270, 1993.
17. R. J. Lipton and R. E. Tarjan, A separator theorem for planar graphs, *SIAM J. Appl. Math.* **36**, 2 (1979), 177–189.
18. G. L. Miller, Finding small simple cycle separators for 2-connected planar graphs, *J. Comput. System Sci.* **32** (1986), 265–279.
19. C. A. Phillips, Parallel graph contraction, *in* "Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures (SPAA)," pp. 148–157, 1989.
20. V. Ramachandran and J. H. Reif, Planarity testing in parallel, *J. Comput. System Sci.* **49**, 3 (1994), 517–561.
21. J. L. Träff and C. D. Zaroliagis, A simple parallel algorithm for the single-source shortest path problem on planar digraphs, *in* "Parallel Algorithms for Irregularly Structured Problems—IRREGULAR '96," Lecture Notes in Computer Science, Vol. 1117, pp. 183–194, Springer-Verlag, Berlin/New York, 1996.