

Planar Separators and Parallel Polygon Triangulation*

Michael T. Goodrich[†]

Abstract

We show how to construct an $O(\sqrt{n})$ -separator decomposition of a planar graph G in $O(n)$ time. Such a decomposition defines a binary tree where each node corresponds to a subgraph of G and stores an $O(\sqrt{n})$ -separator of that subgraph. We also show how to construct an $O(n^\epsilon)$ -way decomposition tree in parallel in $O(\log n)$ time so that each node corresponds to a subgraph of G and stores an $O(n^{1/2+\epsilon})$ -separator of that subgraph. We demonstrate the utility of such a separator decomposition by showing how it can be used in the design of a parallel algorithm for triangulating a simple polygon deterministically in $O(\log n)$ time using $O(n/\log n)$ processors on a CRCW PRAM.

Keywords: Computational geometry, algorithmic graph theory, planar graphs, planar separators, polygon triangulation, parallel algorithms, PRAM model.

1 Introduction

Let $G = (V, E)$ be an n -node graph. An $f(n)$ -separator is an $f(n)$ -sized subset of V whose removal disconnects G into two subgraphs G_1 and G_2 each of size at most $2n/3$ [37]. Typically, separator finding is used to drive divide-and-conquer algorithms [7, 38], where one finds an $f(n)$ -separator of G , dividing G into G_1 and G_2 , and then recurses on each G_i . Such a recursive decomposition is called an $f(n)$ -separator decomposition of G [7]. It produces a binary *decomposition tree* for G , where each node v is associated with a subgraph G_v of G and an $f(|G_v|)$ -sized subset of the nodes of G_v that decompose G_v into two pieces, which are of size at most $2|G_v|/3$ each and are associated with v 's children.

This decomposition tree often corresponds to the calling structure of a divide-and-conquer algorithm for G , where, in order to optimize the running time of the “marry” step in this algorithm, one desires that the separators be as small as possible. Of course, if G is a tree, then this is easy, for there is a node, called the *centroid*, that is itself a separator [10], and

*This research was announced in preliminary form in *Proc. 24th ACM Symp. on Theory of Computing*, 1992, 507–516.

[†]This research was supported by NSF Grants CCR-9003299 and IRI-9116843, and by NSF/DARPA Grant CCR-8908092. Author's address: Dept. of Computer Science, The Johns Hopkins University, Baltimore, MD 21218. Email: goodrich@cs.jhu.edu.

its removal disconnects G into trees that may then be recursively decomposed. Moreover, one can construct a centroid decomposition of such a G in $O(n)$ time sequentially (e.g., see [27]) or in parallel in $O(\log n)$ time using $O(n/\log n)$ processors [15].

If, on the other hand, G is a planar graph, then, in what is now a classic result in algorithmic graph theory, Lipton and Tarjan [37] show that G has an $O(\sqrt{n})$ -separator that can be found in $O(n)$ time. This, of course, immediately leads to an $O(n \log n)$ -time algorithm for constructing an $O(\sqrt{n})$ -separator decomposition of such a graph G , a result that has been used to solve a number of problems in VLSI layout, computational geometry, and algorithmic graph theory (e.g., see [7, 11, 20, 36, 38]).

In this paper we show how to construct an $O(\sqrt{n})$ -separator decomposition of a planar graph G in $O(n)$ time. Our method is based on a recursive emulation of Lipton and Tarjan’s algorithm, except that we use additional data structures to implement each level of the recursion in $o(n)$ time so as to achieve an optimal $O(n)$ running time for the entire decomposition. These data structures include standard binary search trees (such as red-black trees [28, 43]) as well as the more-sophisticated link-cut tree data structure of Sleator and Tarjan [42, 43]. Throughout the recursive decomposition we maintain the breadth-first spanning (BFS) tree [2, 17] of each individual piece the graph, so as to avoid the recomputation of BFS trees as would be required by simple recursive applications of Lipton and Tarjan’s algorithm. We implement this BFS tree maintenance by augmenting the individual pieces with “deactivated” nodes “left over” from previous levels in the recursion. To achieve fast running times for locating separators we also maintain the *inverse* of each BFS tree, which is the tree formed by the graph-theoretic duals of the non-tree edges. Indeed, this *tree interlacing* technique of maintaining a spanning tree and its inverse to speed up dynamic graph maintenance is central to our method. Incidentally, this technique seems to be quite powerful, as it was also employed recently by Eppstein *et al.* [19] for fast maintenance of dynamic minimum spanning trees and by Goodrich and Tamassia [26] for maintaining dynamic planar subdivisions for fast planar point location.

Interestingly, our tree interlacing approach also leads to improved methods for finding many-way separators in parallel. In this case, the problem is to find a small-sized separator that divides G into $O(n^\epsilon)$ disjoint subgraphs, each of size $O(n^{1-\epsilon})$. We show that a many-way $O(n^{1/2+\epsilon})$ -separator can be found in $O(\log n)$ time using $O(n/\log n)$ processors assuming one is given a BFS tree as part of the input (otherwise, it requires $O(\log n)$ time using $O(n^3)$ processors [31, 32]). Our model of computation is the parallel random access machine (PRAM), the synchronous shared-memory parallel model in which simultaneous concurrent reads or writes are either allowed or disallowed, depending upon the submodel designation. This contrasts with the fastest previous parallel algorithm, due to Miller [39], which finds an $O(\sqrt{n})$ -sized binary (cycle) separator in these same bounds. Note that one could iteratively use Miller’s method to find a separator similar to ours, but this would require $O(\log^2 n)$

time. There is also a parallel separator-finding method due to Gazit and Miller [23] that has a more efficient processor bound than our method, but it runs in $O(\log^3 n)$ time (hence, would run in $O(\log^4 n)$ time to find a separator such as ours).

Our $O(n^3)$ processor bound might seem excessively high, but, by following an approach similar to that used in the sequential algorithm by Chazelle [11], we show how to use this inefficient algorithm to design an optimal parallel method for polygon triangulation: the problem of augmenting a simple polygon P with diagonals so that each internal face in the resulting planar subdivision is a triangle [22]. This problem was first studied in the parallel setting by Aggarwal *et al.* [1], and is a problem with many applications (e.g., see [11, 25, 27]). Our method runs in $O(\log n)$ time using $O(n/\log n)$ processors in the deterministic PRAM model where simultaneous concurrent reads and writes are allowed (the *CRCW* PRAM model [31, 32]), where we assume concurrent write conflicts are resolved arbitrarily. This matches the work bound of the best sequential method, due to Chazelle [11], and improves the previous parallel method, due to Clarkson, Cole, and Tarjan [12], which runs in $O(\log n \log \log n \log^* n)$ expected time and has an expected $O(n)$ work bound on a randomized *CRCW* PRAM. It also improves the previous best deterministic methods, which run in $O(\log n)$ time using $O(n)$ processors¹ [24, 46].

As mentioned above, our triangulation method is based on an approach similar to that used in the sequential method of Chazelle [11]. Specifically, we use a divide-and-conquer approach to construct a *submap* of P , that is, a partitioning of P into subpolygons of size $O(n^\delta)$, for some $\delta < 1$, which are then refined to form a triangulation. Our method differs from Chazelle’s approach in some important ways, however. For example, Chazelle is able to use the sequential *contour tracing* paradigm to traverse polygonal chains while performing “local” ray shooting operations, whereas our method depends upon the design of a parallel method for “global” ray shooting operations (this is where many-way separators come in). Another important difference is that our method is based on an n^ϵ -way divide-and-conquer paradigm, whereas Chazelle’s method is based on the simpler binary divide-and-conquer paradigm. This allows us to achieve our $O(\log n)$ running time, but this also requires that the “marry” step in our divide-and-conquer method be more complicated than that of a slower binary approach.

In the next section we present our linear-time method for finding a separator decomposition of a planar graph. In Section 3 we show how to use some of the insights in our sequential method to design an efficient parallel many-way separator-finding algorithm. In Section 4 we show how to triangulate a simple polygon, and we conclude in Section 5.

¹These methods are optimal, however, if P is allowed to contain polygonal holes, which we do not allow.

2 Separator Decomposition

Suppose we are given an n -node plane graph G , i.e., an n -node graph embedded on a sphere so that no two edges cross [8]. Moreover, let us assume that each face, including the external face, has three edges—we refer to such a graph as being *triangulated*. We also assume that G is *simple*, i.e., no two edges e and f are incident upon the same vertices.

We assume that G is represented so that the adjacencies for any vertex v are stored in cyclic order around v in a circular doubly-linked list. For example, G could be represented using the “winged edge” structure of Baumgart [4], the “quad edge” structure of Guibas and Stolfi [29], or the “doubly-connected edge list” structure of Muller and Preparata [40, 41]. In this section we present our linear-time method for constructing an $O(\sqrt{n})$ -separator decomposition of G .

2.1 Our Approach

Before we give our method, however, let us review the approach taken by Lipton and Tarjan [37], which we will emulate. Let T be a rooted BFS tree for G . Let $L(i)$ denote the set of nodes at level i in T , and let $\mathcal{F}(e)$ denote the fundamental cycle determined by T and some non-tree edge e . It is easy to see that any $L(i)$ is a separator, since T is BFS tree, and any $\mathcal{F}(e)$ is a separator, since T is a spanning tree for an embedded planar graph. Lipton and Tarjan’s separator theorem [37] can be viewed as an elegant method for pitting these two kinds of separators against each other in order to find an $O(\sqrt{n})$ -sized separator.

For completeness, we present a simplified version of their approach here. Let l_1 be the level in T such that $\sum_{i=0}^{l_1-1} |L(i)| < n/2$ but $\sum_{i=0}^{l_1} |L(i)| \geq n/2$. Search up T at most \sqrt{n} levels from l_1 to locate a level $l_0 \leq l_1$ (which must exist) such that $|L(l_0)| \leq \sqrt{n}$. Similarly, search down T from l_1 at most \sqrt{n} levels to locate a level $l_2 \geq l_1$ such that $|L(l_2)| \leq \sqrt{n}$. “Cut” T at the levels l_0 and l_2 , dividing it into three “pieces”, G_1 , G_2 , and G_3 . (See Figure 1.) Cutting the nodes at a level involves the removal of all nodes on that level together with all of their incident edges, of which there can be at most $O(n)$, since G ’s being planar implies that $|E|$ is $O(n)$ [8]. If the middle piece, G_2 , is of size at most $2n/3$, then we’re done. So suppose $|G_2| > 2n/3$. Create a new root r and join r to all the roots of subtrees created when we cut T at l_0 . This creates a spanning tree T_2 of G_2 that has depth at most $2\sqrt{n}$. Lipton and Tarjan show that there is a fundamental cycle $\mathcal{F}(e)$ in G_2 (whose size must be $O(\sqrt{n})$) that separates G_2 into two graphs of size at most $2n/3$. Adding this cycle to the nodes in $L(l_0)$ and $L(l_2)$ gives us an $O(\sqrt{n})$ -separator for G , and all this can easily be implemented in $O(n)$ time. This approach, of course, results in a running time of $O(n \log n)$ for constructing a separator decomposition of G .

We show how to reduce the running time for finding such a separator decomposition to

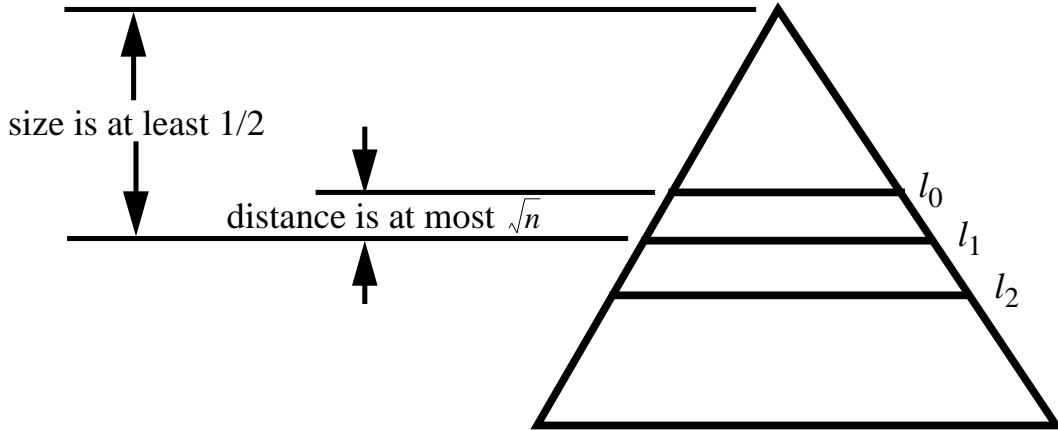


Figure 1: The levels cut for a $O(\sqrt{n})$ separator.

$O(n)$. We achieve this improvement by retaining more information from recursive call to recursive call, so as to implement each level of the recursion in $o(n)$ time. To achieve this we augment T and G with a few data structures.

2.2 The Underlying Data Structures

For each level i in the BFS tree T we store the nodes of $L(i)$ in a dynamic binary search tree $B(i)$, ordered from left to right in the order that their corresponding nodes in T would be visited in an in-order traversal of T . We also maintain a dynamic search tree B that is built upon the $B(i)$'s, stored by increasing level numbers. The specific data structure one uses is not crucial (e.g., a red-black tree [28, 43] will do) so long as, in addition to the usual operations of *Search*, *Insert*, and *Delete*, it also supports the following operations:

- *Split*(B, x): split the tree B into two trees, B_1 and B_2 , such that every element in B_1 is less than x and every element in B_2 is greater than or equal to x .
- *Join*(B_1, B_2): join the trees B_1 and B_2 into one, provided that each element in B_1 is less than every element in B_2 .

In each internal node μ in $B(i)$ (or B) we store the number of the leaf descendants of μ . Maintaining this information for the $B(i)$'s subject to the above operations can easily be done in $O(\log n)$ time per operation (e.g., see [2, 17, 43]).

In addition to its adjacency information, we represent T using the *link-cut* tree data structure of Sleator and Tarjan [42, 43]. This data structure represents an arbitrary rooted tree, such as T , as a collection of paths joined by edges not on any of these paths. The extra edges are not considered to be on the paths, so we refer to the edges on the distinguished paths as *solid* and the other edges as *dashed*. The solid paths are stored in binary search

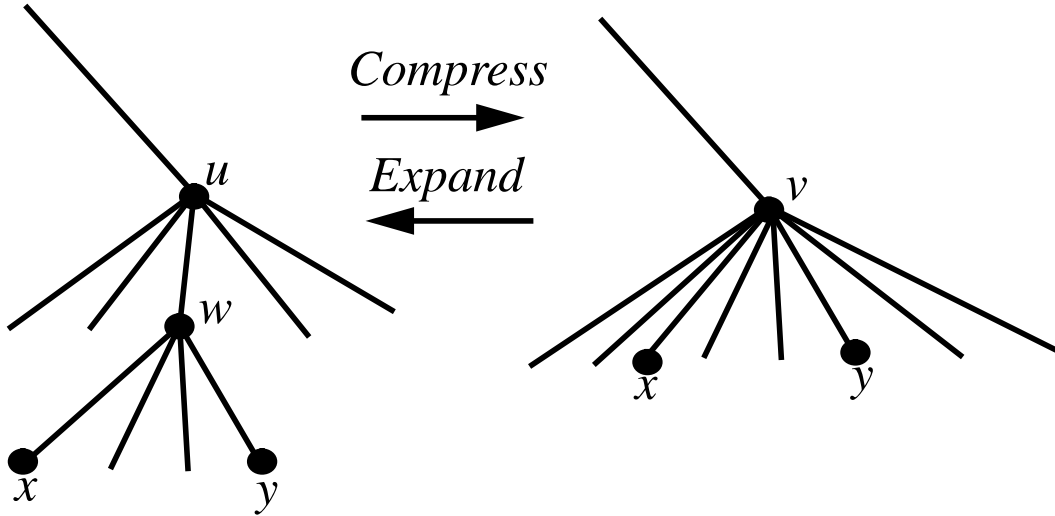


Figure 2: The operations *Compress* and *Expand*.

trees, with each dashed edge (v, w) being represented as a pointer from the root of the tree for the v to the record for w in the tree for the solid path to which w belongs. Sleator and Tarjan show how to implement this structure to support $O(\log n)$ -time execution of the following operations:

- *Cut* (T, e) : separate T at an edge $e \in T$ into two trees T_1 and T_2 .
- *Link* (T_1, v, T_2) : join trees T_1 and T_2 into a single tree by making the root of T_2 a child of v .
- *Path-query* (v) : produce a binary search tree representation of the path from v to the root (which may then be searched using any standard binary-search tree query method).

In addition, as shown by Eppstein *et al.* [19], this data structure also supports the following operations (see Figure 2):

- *Compress* (u, w) : compress the edge (u, w) in T so as to identify u and w as a single vertex, v . The parent of w becomes the parent of v and the children of u and the children of w (except for u itself) become the children of v (with the same relative in-order ordering).
- *Expand* (v, x, y) : expand the node v into two nodes u and w , such that u is the parent of w , with w taking all of v 's children between x and y , inclusive, and u taking v 's other children.

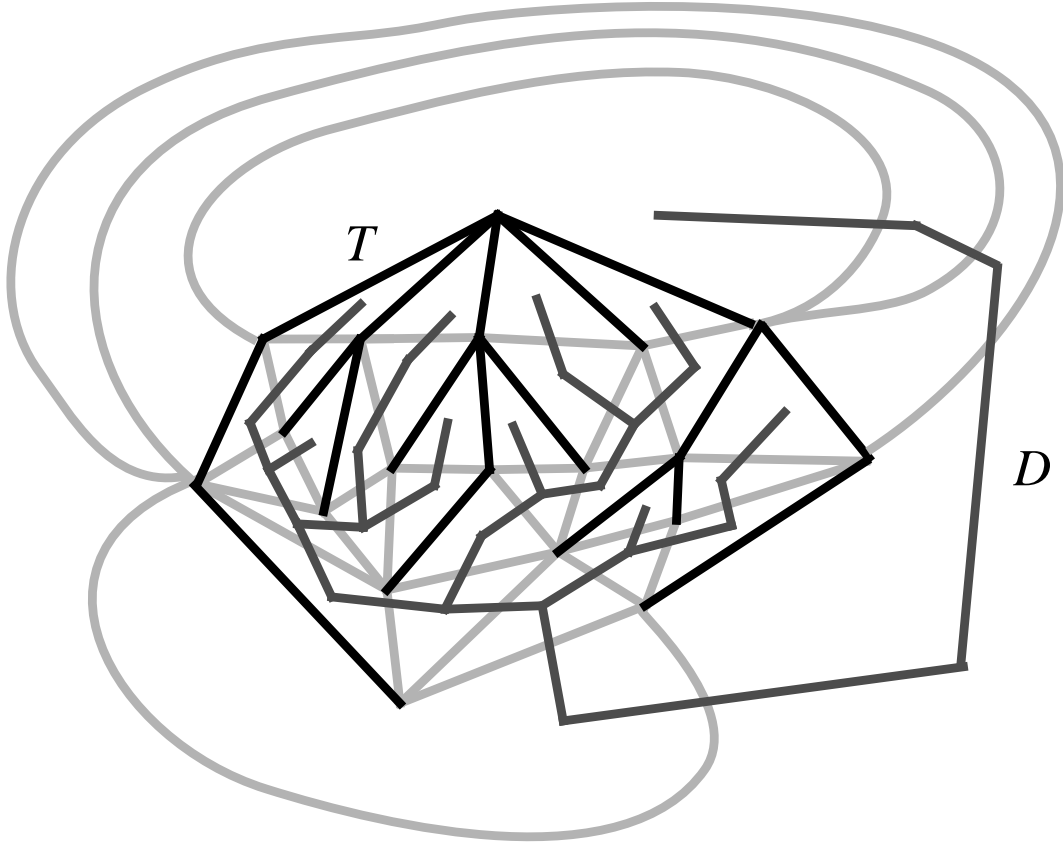


Figure 3: The BFS tree T and its inverse D .

We also augment the binary trees representing solid paths so that we can compute the number of nodes on a path, strictly to the left of a path, or strictly to the right of a path in $O(\log n)$ time. This is easily accomplished by associating appropriate values with the leaves of our solid trees and summing these values in internal nodes. Given this modification, it is a simple matter to compute the number of nodes inside, outside, and on a fundamental cycle determined by some non-tree edge (v, w) in $O(\log n)$ time.

Finally, we also represent a spanning tree of the graph-theoretic dual of G . In particular, as mentioned in the introduction, we maintain a tree D such that each edge of D is the graph-theoretic dual of a non-tree edge in G . (See Figure 3.) As with T , we maintain D using the link-cut tree data structure of Sleator and Tarjan [42, 43].

We initialize our recursive computation by constructing each of these data structures. This includes the construction of the BFS tree T , its graph-theoretic inverse, D , the $L(i)$ level lists, as well as the underlying data structures that include the link-cut tree representations of T and D , and the binary search trees B and the $B(i)$. This can all be done in $O(n)$ time (see [26, 28, 42, 43]).

2.3 Separating G into G_1 , G_2 , and G_3

Our method for performing recursive separator, then, is to use these structures to find an $O(\sqrt{n})$ -separator and then remove the separator vertices while maintaining these data structures in the resulting “pieces.” In order for this approach to run in $O(n)$ time over all the recursive calls we require that each separation step run in $o(n)$ time. To achieve this requirement we will perform the separation of G so as to maintain the data structures B , T , and D in the resulting subgraphs. We do this by removing some, but not necessarily all, vertices in G that belong to the separator. We refer to the separator vertices that remain in a subgraph as “deactivated” vertices.

We must take care, however, that the total size of all the subgraphs still be linear. We fulfill this requirement by forcing the total number of edges in all the subgraphs to be at most that of the original graph. Unfortunately, this requires that we relax our assumption about G being a simple triangulated plane graph (for this will only be guaranteed to be true for the initial G). Instead, we will inductively assume only that G is a triangulated plane graph, i.e., each face is a triangle. Interestingly, a familiar property for simple plane graphs, still holds for non-simple triangulated plane graphs:

Lemma 2.1: *Let $G = (V, E)$ be a possibly non-simple triangulated plane graph. Then $|E| = 3|V| - 6$.*

Proof: The proof is essentially the same as that for the similar relationship that holds for simple plane graphs (e.g., see [8] (p. 144)). \square

So, let us begin our method by assuming inductively that we are given the above B , T , and D structures representing G . Let n denote the number of active vertices in G , let k denote the number of deactivated vertices in G . If $k > n$, then we perform a simple traversal of G to remove all the deactivated vertices in G together with their adjacencies. Since G is a triangulated plane graph, such a traversal of G takes $O(n + k) = O(k)$ time (by Lemma 2.1). In this case, we then re-triangulate all the (simple) plane graph components, constructing the B , T , and D structures for each, and proceed on each independently. Thus, for the remainder of this discussion, let us assume that $k \leq n$ and, since G is a triangulated plane graph, that the number of edges in G is $O(n)$.

We search in B to locate the level l_1 (as defined in Section 2.1) in $O(\log n)$ time, and, from there, we iteratively search in B to locate the levels l_0 and l_2 in $O(\sqrt{n})$ additional time. Having found l_0 and l_2 , we must separate T (and G) by removing the nodes at levels l_0 and l_2 , creating G_1 , G_2 , and G_3 . Of course, we must also create the underlying data structures for G_1 , G_2 , and G_3 , as well (i.e., B_1 , B_2 , B_3 , T_1 , T_2 , T_3 , D_1 , D_2 , and D_3).

This separation step can be viewed as “cutting” G along the edges that join the vertices at levels l_0 and l_2 , respectively, dividing G into three parts. Let us concentrate on the

operations for the nodes in $L(l_0)$, as the method for the nodes in $L(l_2)$ is similar. Given an embedded plane graph G , as described above, we create B_1 , G_1 , T_1 , and D_1 as follows.

Step 1: Creating B_1 and G_1 . We perform a *Split* in B at the node for level l_0 , and remove all the nodes of $B(l_0)$. This creates B_1 and a binary tree B' that can then be split into B_2 and B_3 by a similar operation. To maintain the adjacency information in G we split v into two (non-adjacent) nodes v' and v'' , so that v' retains all of v 's adjacencies to nodes on levels l_0 and $l_0 - 1$ and v'' retains all of v 's adjacencies to nodes on levels l_0 and $l_0 + 1$. We do not include any edges of the form (v', v'') , however. The nodes on levels 0 to $l_0 - 1$, together with the v' nodes, form the nodes of G_1 and the nodes on levels $l_0 + 1$ and greater, together with the v'' nodes, form nodes of the remaining piece of G (which will later be cut into G_2 and G_3).

Analysis of Step 1. Assuming that the adjacency list for each v in $L(l_0)$ is represented as a circular doubly-linked list, and we have separate pointers to the sublist of adjacencies to nodes on level $l_0 - 1$, the sublist adjacencies to nodes on the level l_0 , and the sublist of adjacencies to nodes on the level $l_0 + 1$ —we can perform all of these operations in constant time per node visited, plus an $O(\log n)$ charge for the search in B . Thus, this step can be implemented in $O(\sqrt{n} + k_0)$ time, where k_0 is the number of deactivated vertices in $L(l_0)$.

Step 2: Compressing the separator vertices in G_1 . In this step we compress each edge connecting v' vertices in $L(l_0)$. We mark the remaining vertices as “deactivated” vertices (there will be one for each connected component in the subgraph induced by $L(l_0)$), and we do not count such vertices in the number of vertices of G_1 . It is important to note that the number of edges in G_1 does not increase, but the result is that G_1 is a fully-triangulated plane (multi-) graph (except that there may now be some faces having only two edges). (See Figure 4.) If any such edge compression should create two parallel tree edges (on the same face), then we arbitrarily pick one of these parallel edges to no longer be a tree edge (so that we maintain T_1 as a tree), and we remove the non-tree edge from G_1 . Similarly, if we create two parallel non-tree edges, then we remove one of them arbitrarily. This maintains G_1 as a triangulated plane graph, and is also illustrated in Figure 4.

Analysis of Step 2. Assuming adjacency lists are partitioned as we needed for the analysis of Step 1, this step can be implemented in $O(m_0)$ time, where m_0 is the number of edges in the subgraph of G induced by the nodes in $L(l_0)$.

Step 3: Creating T_1 and T' . To create T_1 we perform a *Cut* on the edge from v to its parent in T , for each $v \in L(l_0)$. This separates T_1 from the nodes of $G_2 \cup G_3$. Let us therefore consider each such v node on level l_0 as the (deactivated) root of a subtree of T in $G_2 \cup G_3$, i.e., we view each as a v'' node. Unfortunately, there may be many such v'' nodes, implying that we do not have a single BFS tree for the nodes of $G_2 \cup G_3$. But we need such a BFS tree to implement the separation of G_2 and G_3 . Thus, we artificially form such a BFS tree by merging all the v'' nodes into a single node r (much as we did in Step 2), which

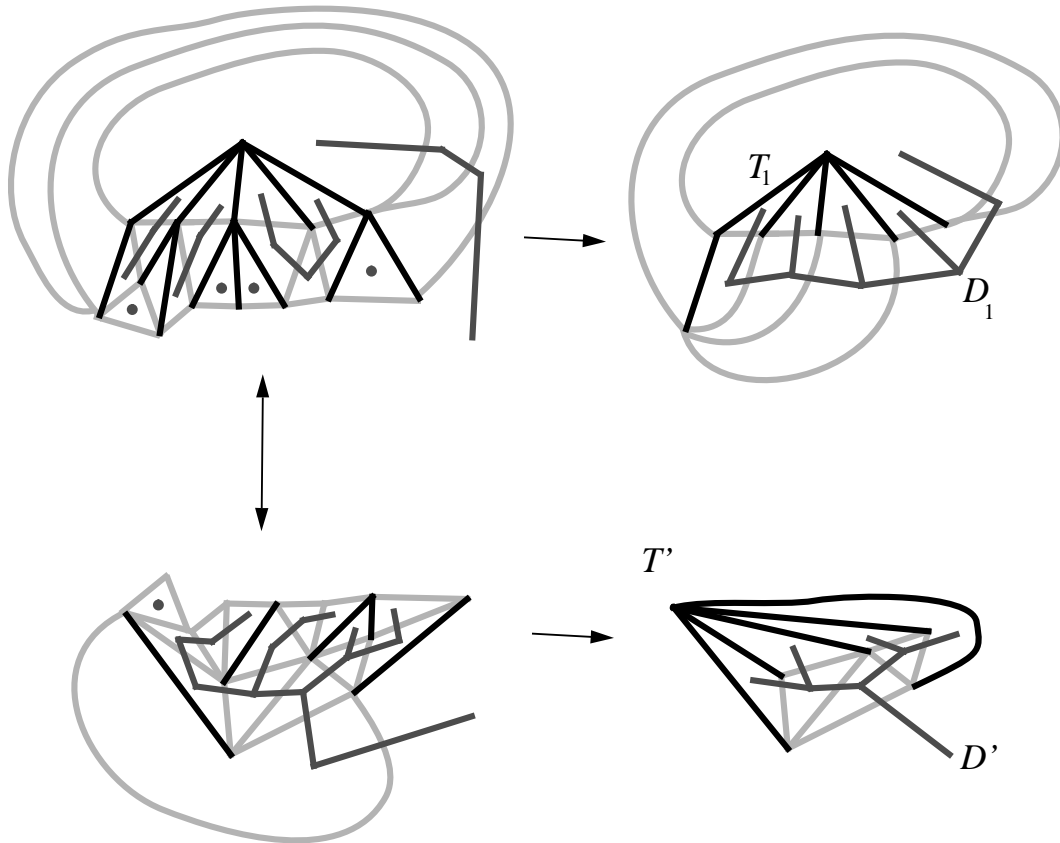


Figure 4: Separating G at l_0 and compressing the nodes of $L(l_0)$ in each resulting piece.

becomes the root of a new BFS tree T' for all of $G_2 \cup G_3$. Moreover, as Lipton and Tarjan show [37], the resulting graph, G' , remains a plane graph. And, as was the case in Step 2, we get that G' is a plane (multi-) graph, which in this case has the same number of edges as the original G' (although some faces may now have only two edges). (See Figure 4.) As in Step 2, we remove any non-tree edges that become parallel to tree edges as a result of this compression, so as to maintain G' as a triangulated plane graph.

Analysis of Step 3. First, we observe that cutting T_1 from T requires $O(\sqrt{n} + k_0)$ *Cut* operations and the root-merging process can be implemented using $O(\sqrt{n} + k_0)$ *Link* and *Compress* operations. Thus, the construction of T_1 and T' can be implemented in $O((\sqrt{n} + k_0) \log n)$ time.

Step 4: Creating D_1 . Of course, performing all the above changes to G and T necessitates that we update our representation for D , the inverse of T , so as to create D_1 and a dual spanning tree D' that can then be split into D_2 and D_3 . Certainly, for each non-tree edge e of G removed in constructing G_1 and T_1 (including multi-edges), we must perform the corresponding *Cut* operation along the edge of D dual to e , and remove any isolated nodes this creates. (See Figure 4.) In addition, during the compression of l_0 -level edges in G_1 in Step 2 we must connect the appropriate parts of D_1 via *Link* operations (note that we actually only need to do this when we create a parallel tree edge in T_1 that is then removed). Finally, each parallel edge removed in Step 3 requires a *Cut* and a *Link* to update the dual tree D' .

Analysis of Step 4. We perform at most $O(m_0)$ *Cut* and *Link* operations. Thus, the running time for this step is $O(m_0 \log n)$.

This completes our cutting operation along level l_0 . By a similar operation we may cut G' along level l_2 so as to create $B_2, B_3, G_2, G_3, T_2, T_3$, and D_2 , and D_3 . The only caveat is that G_1, G_2 , and G_3 may contain extra deactivated nodes and edges incident upon these nodes (and possibly even multiple edges), but we may nevertheless keep the total number of edges to be at most the number in G .

Let n_1, n_2 , and n_3 denote the number of active nodes in G_1, G_2 , and G_3 , respectively. Clearly, $n_1 + n_2 + n_3 < n$ and m_i is $O(n_i)$ for $i = 1, 2, 3$. Note at this point that we have $n_1 \leq n/2$ and $n_3 \leq n/2$. We are not done with the construction of our separator, however, if $n_2 > 2n/3$.

2.4 Finding a Fundamental Cycle in G_2

If $n_2 > 2n/3$, then we must emulate the second step in the approach of Lipton and Tarjan [37]—that of finding a separating fundamental cycle $\mathcal{F}(e)$ in G_2 , i.e., a fundamental cycle that minimizes the maximum number of nodes either inside or outside the cycle. Indeed, the search for such a cycle is the motivation for the maintenance of a link-cut representation

of the tree D . In this case we are interested in D_2 . For each node v in D_2 we let $e(v)$ denote the edge in G_2 that is dual to the edge from v to its parent in D_2 . Given any edge $e(v)$ we can use the link-cut representation of T_2 to compute the number of nodes of G_2 strictly inside $\mathcal{F}(e(v))$, on $\mathcal{F}(e(v))$, and strictly outside $\mathcal{F}(e(v))$ in $O(\log n)$ time. Moreover, using the method of Goodrich and Tamassia [26], we may locate a centroid node v in D_2 in $O(\log n)$ time. This centroid location is implemented by a simple search in the solid path in D_2 containing the root [26], and does not require any modifications to the standard link-cut tree representation [42]. Using the information obtained from the queries in T_2 , we can then perform a (temporary) cut at v and recurse on the appropriate subtree of D_2 . By repeating this procedure $O(\log n)$ times we will find a fundamental cycle $\mathcal{F}(e)$ in G_2 that minimizes the maximum number of nodes either inside or outside the cycle. We then reverse these temporary cuts so as to reconstruct D_2 . The total time for this search is $O(\log^2 n)$.

By construction, the total number of nodes on this cycle $\mathcal{F}(e)$ is $O(\sqrt{n})$. The computation that remains, then, is to cut G_2 along $\mathcal{F}(e)$, and update the underlying data structures for the two remaining pieces, G'_2 and G''_2 . We do this as follows:

Step 1: Splitting B_2 into B'_2 and B''_2 . The cycle $\mathcal{F}(e)$ intersects each level of T_2 in at most two nodes. Moreover, since each $B(l)$ is stored in in-order, $\mathcal{F}(e)$ divides $B(l)$ into at most three pieces, two of which are exterior to $\mathcal{F}(e)$, and one of which is interior to $\mathcal{F}(e)$. These pieces can be formed by performing a *Split* in $B_2(l(v))$ for each node v in $\mathcal{F}(e)$, where $l(v)$ denotes v 's level. This may be followed by a *Join* operation for each l to join the two exterior pieces for level l , creating at most two trees, $B'_2(l)$ and $B''_2(l)$, for each level l . We may then construct the tree B'_2 (resp., B''_2) by combining all the $B'_2(l)$'s (resp., $B''_2(l)$'s). The total time needed to implement these operations is $O(\sqrt{n} \log n)$. We do not include the nodes of $\mathcal{F}(e)$ in the resulting trees (although, as we show in the next step, these nodes will remain, albeit as deactivated nodes).

Step 2: Splitting G_2 into G'_2 and G''_2 . We divide G_2 along the edges of $\mathcal{F}(e)$ making each node on $\mathcal{F}(e)$ a deactivated node in both G'_2 and G''_2 , in a manner similar to that used in the previous subsection. In addition, so as to maintain the same number of edges in each piece, we compress each pair of nodes on $\mathcal{F}(e)$ that are on the same level in T_2 . We perform this compression in G'_2 , as well as in G''_2 (viewing G_2 as being drawn on a sphere makes the compressions in these two graphs symmetric operations). (See Figure 5.) Also, as in the previous subsection, any time we create two parallel edges we simply remove one of them. This step is easily accomplished in $O(\sqrt{n})$ time, since each vertex stores its adjacencies in a doubly-linked circular list.

Splitting T_2 into T'_2 and T''_2 . Creating the BFS trees for G'_2 and G''_2 is accomplished by performing $O(\sqrt{n})$ *Expand*, *Compress*, *Cut*, and *Link* operations. We perform an *Expand* operation for each node v on $\mathcal{F}(e)$ to expand v into an edge (v', v'') so as to separate its adjacencies on or inside $\mathcal{F}(e)$ (which go with v') from its adjacencies outside $\mathcal{F}(e)$ (which go

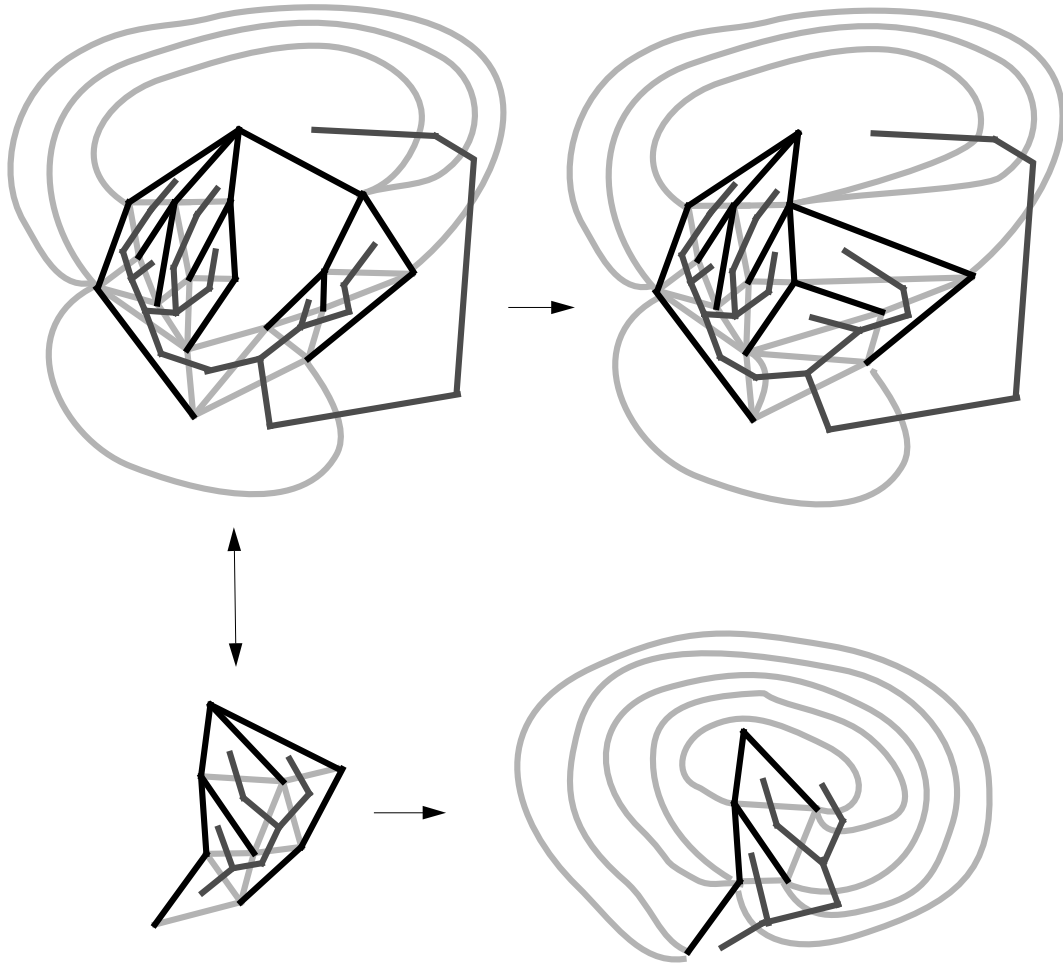


Figure 5: Separating G_2 along $\mathcal{F}(e)$ compressing the nodes of $\mathcal{F}(e)$ at the same level in each resulting piece.

with v''). We then perform a *Cut* for each such expanded edge (v', v'') , and we link each node v'' with w'' , such that w was the parent of v in T_2 . Finally, we compress the nodes on the same level on $\mathcal{F}(e)$, each of which can be done with a *Cut*, *Link*, and *Compress* operation. An example pair of resulting trees is illustrated in Figure 5 using bold edges. The total time for all of this is $O(\sqrt{n} \log n)$.

Splitting D_2 into D'_2 and D''_2 . In this case, our final update operation of splitting D_2 into D'_2 and D''_2 is quite trivial. We simply cut D_2 at the dual to e . (See Figure 5.) This takes $O(\log n)$ time and completes the construction of the separator.

As in the previous subsection, we left some deactivated nodes and some of their adjacencies in order to efficiently perform the separation of G_2 into G'_2 and G''_2 . Note, however, that the number of edges in G'_2 and G''_2 is the same as the number in G_2 .

Our method for constructing a separator decomposition, then, is to recursively iterate the above procedures for each of the resulting pieces.

2.5 Analysis

Let us therefore analyze the total running time of this method. First, note that deactivated nodes are not counted in our search for a separator; so our separator will still be of size $O(\sqrt{n})$ and will still separate the active nodes in G into two pieces whose number of active nodes is least $n/3$ and at most $2n/3$, where n denotes the number of active nodes in G .

We will use an “accounting” argument in our analysis. In particular, any time we mark a node v as deactivated, then we imagine that we give v $\lceil \log n \rceil$ credits, each of which is worth $O(1)$ computation steps. Since we are allowing at least $O(\sqrt{n} \log n)$ time for our separation running time, this accounting trick only increases the running time charged to a separation computation by a constant factor (since there are only $O(\sqrt{n})$ nodes receiving such credits). So, consider the case where $k > n$ and we must “purge” away the deactivated nodes. In this case we spend $O(k)$ time, all of which can be charged to the credits “stored” by deactivated nodes, for after this operation completes there are no deactivated nodes left. So, next consider the time spent for an l_0 -level separation. It takes $O(m_0 \log n)$ time, where m_0 is the number of edges of G induced by the nodes in $L(l_0)$ (note that $\sqrt{n} + k_0 \leq m_0$, since G is connected). Clearly, each of the k_0 deactivated nodes in $L(l_0)$ were deactivated in a fundamental-cycle separation (at a higher level in the recursion). Moreover, because of the way we compress nodes after a fundamental-cycle separations, the adjacencies of any deactivated node in $L(l_0)$ is a subset of the adjacencies for at most two nodes in the original graph. Since this is an induced subgraph of a simple plane graph, we have that m_0 is $O(\sqrt{n} + k_0)$. Thus, the time needed for an l_0 -level separation is actually $O((\sqrt{n} + k_0) \log n)$. But we may inductively assume each deactivated node in $L(l_0)$ contains $\log n$ credits, each worth $O(1)$ computation steps. Thus, we may account for the running time of this l_0 -level

separation as being $O(\sqrt{n} \log n)$. A similar argument establishes a similar bound for an l_2 -level separation. Since we have already established that a fundamental-cycle separation takes $O(\sqrt{n} \log n)$ time, we may therefore characterize the running time of our method by the following recurrence relation:

$$T(n) \leq T(n') + T(n'') + b\sqrt{n} \log n,$$

where $b > 0$ is a constant, $n' + n'' < n$, and $n/3 \leq n' \leq n'' \leq 2n/3$. We may easily show by induction that

$$T(n) \leq cn - c\sqrt{n} \log n,$$

for some constant $c > 0$ (where the negative term is added to make the induction easier). This gives us the following theorem:

Theorem 2.2: *Given an n -vertex simple triangulated plane graph G , one can construct an $O(\sqrt{n})$ -separator decomposition of G in $O(n)$ time.*

Incidentally, this theorem immediately implies that one can perform the VLSI embedding of a planar graph in $O(n)$ time using the algorithms of [7, 36]. Moreover, in a manner similar to the approach of Lipton and Tarjan [37], it is straightforward to generalize our methods to weighted graphs. In this case one is given a planar graph G with nonnegative vertex weights summing to W , and one desires a decomposition based on the recursive construction of a $O(\sqrt{n})$ -weighted-separator, i.e., a separator of size $O(\sqrt{n})$ that divides G into two pieces of weight at most $2W/3$. We leave the details to the reader, but note that one possibility would be to modify our method so that in addition to dividing based on the number of nodes in G one can also be dividing by weights associated with the nodes in G (e.g., one can follow a separation based on node counts by a separation based on weights). This approach can be used to derive the following theorem:

Theorem 2.3: *Let G be an n -vertex simple triangulated plane graph with nonnegative vertex weights. Then one can construct an $O(\sqrt{n})$ -weighted-separator decomposition of G in $O(n)$ time.*

3 Many-Way Separators

Interestingly, our tree-based approach to finding planar separators also carries over into the parallel setting. In this section we give our method for finding small-sized separators that divide G into many similarly-sized subgraphs. In particular, our method finds a separator of size $O(n^{1/2+\epsilon})$ that divides G into $O(n^\epsilon)$ subgraphs of size $O(n^{1-\epsilon})$. Assuming we are given a BFS spanning tree as part of the input, our method runs in $O(\log n)$ time using $O(n/\log n)$

processors in the deterministic PRAM model where concurrent reads are allowed but writes must be exclusive (the *CREW* PRAM model [31, 32]). If we are not given the BFS spanning tree, then our method runs in $O(\log n)$ time using $O(n^3)$ processors on a *CRCW* PRAM (e.g., see [31, 32] for a method for constructing a BFS tree in these bounds).

3.1 Many-way Trees

Since our method produces a separation of G into many subgraphs it gives rise to a decomposition tree that is not binary. So, before we describe our method, let us make a few observations about non-binary trees. Let T be a balanced rooted tree, and, for each node v in T , let n_v denote the number leaves in T that are descendants of v (including v itself if it is a leaf). We say that T is a *globally $f(n)$ -way tree* if each node v has at most $f(n)$ children, and we say that T is a *locally $f(n)$ -way tree* if each node v has at most $f(n_v)$ children. Note that the height of a globally $f(n)$ -way tree is $O(\log n / \log f(n))$, whereas the height of a locally $f(n)$ -way tree is determined by the recurrence $h(n) = h(n/f(n)) + 1$. For example, given a constant $0 < \epsilon < 1$, if F_1 is a globally n^ϵ -way tree and F_2 is a locally n^ϵ -way tree, then F_1 has height $O(1/\epsilon) = O(1)$ whereas F_2 has height $\Theta(\log \log n)$.

An important property of a balanced tree T is that it allows for a canonical representation for intervals. In particular, let the leaves of T be numbered left-to-right $1, 2, \dots, n$, and let each internal node v of T be associated with the interval $[a, b]$ that spans v 's descendants. As in the (binary) segment tree data structure of Bentley and Wood [5], we say that an interval $[c, d]$ *covers* a node v if $[c, d]$ contains the interval for v but does not contain the interval for v 's parent. An interval $[c, d]$ can therefore be represented by the union intervals in T that it covers. Note that any interval $[c, d]$ can cover at most $O(f(n) \log n / \log f(n))$ nodes in a globally $f(n)$ -way tree and at most a number in a locally $f(n)$ -way tree that is determined by the recurrence $g(n) = g(n/f(n)) + 2f(n)$. Continuing our example, note that an interval $[c, d]$ can cover at most $O(n^\epsilon)$ nodes in either F_1 or F_2 .

3.2 Constructing Many-way Separators

Let T be a BFS spanning tree for G , and let $0 \leq \epsilon \leq 1/2$ be a given constant. We begin our method for constructing an $O(n^{1/2+\epsilon})$ -sized many-way separator for G in parallel by constructing each $L(i)$ in the form of an array representing a listing of the nodes on level i , ordered by increasing in-order numbers². We refer to this computation as the construction of a *level linking* of T , which we can do efficiently by the following lemma.

²We assume the adjacencies for any node v in T are cyclically ordered, and this in-order number is consistent with this ordering.

Lemma 3.1: *Let T be an n node rooted tree. One can construct a level linking of T in $O(\log n)$ time using $O(n/\log n)$ processors on a CREW PRAM.*

Proof: Using the Euler-tour technique on trees [31, 32, 45], we construct a list L of the nodes in T listed in the order they would be traversed in a recursive tree traversal (such as the in-order traversal), so long as the children of a node v are “visited” according to their cyclic ordering around v . It is important to note that a node in L is included *each* time it would be visited in such a traversal, not just when it would be assigned its, say, in-order number. With each node v in this list we associate its level number $l(v)$, which can be computed by another application of the Euler-tour technique, along with calls to the well-known *list ranking* [3, 14] and parallel prefix [34, 35] techniques³ (see also [31, 32]). All of this can be implemented in $O(\log n)$ time using $O(n/\log n)$ processors. Let v be a node in T , and let v_{first} and v_{last} respectively denote the first and last copies of v in L . Notice that v ’s left neighbor in $L(i)$ corresponds to the node left of v_{first} in L that is nearest to v_{first} and has level label at least as large as $l(v_{\text{last}})$. Similarly, v ’s right neighbor in $L(i)$ corresponds to the node right of v_{last} in L that is nearest to v_{last} and has level label at least as large as $l(v_{\text{last}})$. The problem of locating all such neighbors for every node in L is known as the *all nearest larger values* problem, can, as shown by Berkman *et al.* [6], this can be solved in $O(\log n)$ time using $O(n/\log n)$ processors on a CREW PRAM. Linking each node v to its neighbors in $L(i)$ gives us a linked-list representation of each $L(i)$. A final application of list ranking [3, 14], then, gives us each $L(i)$ represented in an array. \square

Given the $L(i)$ lists we build a binary tree B “on top” of these lists, as we did in the previous section. Let s_i denote the number of nodes on levels $1, 2, \dots, i$. The tree B allows us to perform “binary searches” on the s_i values. We allocate $O(n^\epsilon)$ processors to the task of finding each level i such that the interval $(s_{i-1}, s_i]$ contains a multiple of $\lceil n^{1-\epsilon} \rceil$. Call such levels *starter* levels. We then assign $O(n^{1/2})$ processors to each starter level i to locate the levels i' and i'' nearest to i such that i' and i'' contain at most $2\lceil \sqrt{n} \rceil$ nodes. Call these levels the *cutter* levels. Note that such levels must exist within a distance of $\sqrt{n}/2$ of each starter level, by a simple pigeon-hole argument. We have to take a bit of care here to avoid concurrent writes, which could occur if the computations for different i ’s simultaneously discover the same cutter level. This difficulty is easily overcome, however, by limiting the search for the cutter level nearest to level i to the interval of levels spanned by i ’s predecessor and successor starter levels. Given the cutter levels, we disconnect G along the nodes of each of these levels, removing each node at such a level, as well as all its adjacencies. This can be implemented in $O(\log n)$ time using $O(n/\log n)$ processors, and it decomposes G into $O(n^\epsilon)$ subgraphs G_1, G_2, \dots, G_m , such that each G_i either has size $O(n^{1-\epsilon})$ or depth $O(n^{1/2})$.

³Recall that a list ranking determines for each node x in a linked list A the rank of x in A , and a parallel prefix sum determines for each element x in an array A the sum of the elements that precede x in A .

Let n_i denote the number of nodes in G_i . If n_i is $O(n^{1-\epsilon})$, then we are done. So, suppose n_i is over this threshold. That is, G_i has depth $O(n^{1/2})$. We treat each such G_i in parallel. Let l_i denote the top level of G_i . We create a “dummy” node r and make each node v on level l_i , which is currently the root of a subtree in G_i , be a child of r . This representation gives us a BFS tree, T_i , for this augmented G_i , and can easily be constructed in $O(\log n)$ time using $O(n_i/\log n)$ processors, given the array $L(l_i)$. We then construct the inverse of T_i , which we denote D_i (as in Section 2.2). This too can easily be constructed in $O(\log n)$ time using $O(n_i/\log n)$ processors.

Note that a centroid of D_i determines a non-tree edge e of G_i that in turn determines a fundamental cycle $\mathcal{F}(e)$ that has at most $2n_i/3$ nodes either inside or outside⁴ $\mathcal{F}(e)$. We therefore form a centroid decomposition of D_i using the accelerated centroid decomposition method of Cole and Vishkin [15], which can be implemented for G_i in $O(\log n)$ time using $O(n_i/\log n)$ processors. This gives us a binary decomposition tree A such that each node μ in A corresponds to a node in G_i that forms a centroid of the piece of T_i one gets by cutting T_i at the nodes associated with μ 's ancestors. For each μ in A , we can determine a non-tree edge e_μ in G_i that in turn determines a fundamental cycle $\mathcal{F}(e_\mu)$ in G_i . Each such $\mathcal{F}(e_\mu)$ contains in its interior (or exterior) a constant fraction of the number of nodes in the piece of G_i one gets by “cutting out” the fundamental cycles associated with μ 's ancestors. Thus, there is a level l in A of size $O(n^\epsilon)$ such that the nodes on level l and higher determine a decomposition of G_i into $O(n^\epsilon)$ pieces, each of size at most $\lceil n^{1-\epsilon} \rceil$. Since G_i has depth $O(n^{1/2})$ the total size of all these fundamental cycles is $O(n^{1/2+\epsilon})$. Collecting the nodes on these cycles with the nodes on the cutter levels, then, gives us our separator. Let us, therefore, summarize our results so far with the following lemma:

Lemma 3.2: *Supposed one is given an n -node embedded planar graph G , a BFS spanning tree T on G , and a parameter $0 < \epsilon < 1/2$. Then one can construct an $O(n^{1/2+\epsilon})$ -size separator of G that divides G into $O(n^\epsilon)$ subgraphs of size $O(n^{1-\epsilon})$. This construction can be implemented in $O(\log n)$ time using $O(n/\log n)$ processors on a CREW PRAM.*

Iteratively applying this lemma immediately gives us the following theorem:

Theorem 3.3: *Supposed one is given an n -node embedded planar graph G and a parameter $0 < \epsilon < 1/2$. Then one can construct a globally $O(n^\epsilon)$ -way $O(n^{1/2+\epsilon})$ -size separator decomposition of G in $O(\log n)$ time using $O(n^3)$ processors on a CRCW PRAM.*

⁴The alert reader will note that $\mathcal{F}(e)$ actually bounds the number of *triangles* on either side of it. This causes no difficulties, however, since the number of nodes inside or outside of $\mathcal{F}(e)$ is bounded by the number of triangles that are respectively inside or outside $\mathcal{F}(e)$ (indeed, the number of nodes may be significantly smaller than the number of triangles).

Proof: The method is a straightforward recursive application of the previous lemma. The large processor bound comes from the best known bound for constructing a BFS tree in $O(\log n)$ time (e.g., see [31, 32]). \square

We give a non-trivial application of this result in the following section.

4 Parallel Polygon Triangulation

Suppose we are given a simple polygon P . The problem we address in this section is that of augmenting P with diagonal edges so as to decompose P 's interior into triangles. Before we give our method, however, let us first address a problem that will arise repeatedly in our method.

4.1 Point Location in a Jordan Tessellation

Suppose we are given an m -edge Jordan tessellation J of the plane, that is, a subdivision of \mathbb{R}^2 with simple, closed curves (where two curves to share a portion of their do allow two). Suppose further that each edge in the Jordan tessellation is a piece-wise linear curve, and let n denote the total number of linear pieces in J . (See Figure 6.) Given a point p , a *horizontal ray-shooting query* for p is to determine the first point of J , called the *shadow* of p , that is hit by a horizontal ray emanating from p . We assume that we have the following *ray-shooting oracle*:

- *Ray-shooting oracle.* There is an oracle that allows us to perform a horizontal ray shoot against a single Jordan curve in $O(\log n)$ time using $\rho(n)$ processors (where $\rho(n)$ is a measure of the “complexity” of the Jordan curves that make up J).

In a fashion analogous to a sequential structure used by Chazelle [11], in this subsection we show how to use the parallel separator decomposition theorem (3.3) to design a data structure that allows for arbitrary horizontal ray shooting queries to be performed in $O(\log n)$ time using $O(\rho(n)m^{1/2+\epsilon})$ processors.

Let D be the graph-theoretic dual of J , that is, the graph formed by associating a vertex with each face of J and adding an edge (v, w) for each pair of faces whose boundaries overlap. (See Figure 6.) Apply Theorem 3.3 to D to produce a globally m^ϵ -way separator decomposition tree T for D , for some constant $0 < \epsilon < 1/2$. The height of this tree is $O(1/\epsilon) = O(1)$. In addition to the separator stored at each node v , we also store at v the $O(m^{1/2+\epsilon})$ Jordan curves that are dual to the nodes of D stored at v .

Given a horizontal ray \vec{r} , we perform a ray shoot for \vec{r} as follows. Using $O(\rho(n)m^{1/2+\epsilon})$ processors we perform a ray shoot against all the Jordan curves stored at the root of T . This

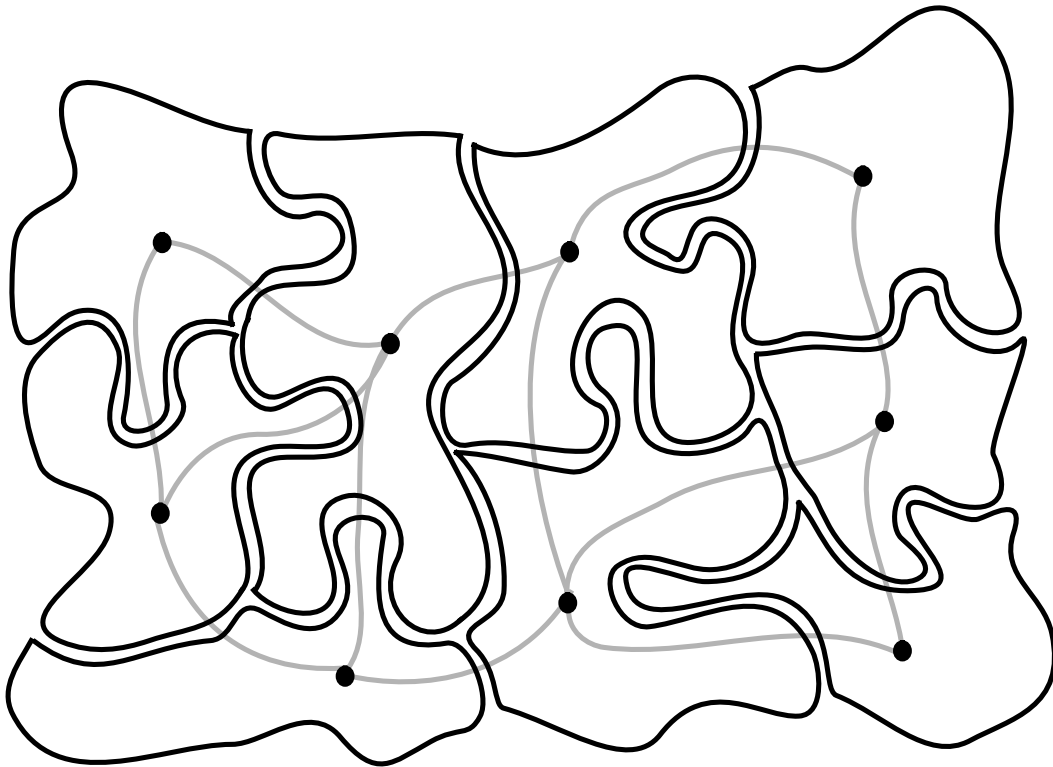


Figure 6: A Jordan Tessellation and its dual graph (excluding the external face).

takes $O(\log n)$ time, by assumption, and returns the horizontal *shadow* of the head of \vec{r} on each curve stored at the root of T . We then compute the shadow point nearest to the head of \vec{r} , and determine the subregion we traverse just before hitting this point. This takes an additional $O(\log m)$ time using $O(m^{1/2+\epsilon})$ processors (without using concurrent writes), and it informs us of the child w of v in T at which we may now recurse to complete the ray shoot for \vec{r} . If w is not a leaf, then we repeat this test at w . Since there are $O(1)$ levels in T , this procedure clearly runs in $O(\log n)$ time. Therefore, we have the following theorem:

Theorem 4.1: *Suppose one is given an m -edge planar Jordan tessellation J . Suppose further that there exists an oracle that can perform a horizontal ray shooting query for a single Jordan curve in J in $O(\log n)$ time using $\rho(n)$ processors. Then one can construct a data structure for J in $O(\log n)$ time using $O(m^3)$ processors on a CRCW PRAM that allows $O(\rho(n)m^{1/2+\epsilon})$ CREW PRAM processors to perform horizontal ray shooting queries in J in $O(\log n)$ time, where ϵ is any constant such that $0 < \epsilon < 1/2$.*

This theorem plays an important role in our method for polygon triangulation, which we now describe.

4.2 Our Polygon Triangulation Algorithm: An Overview

Suppose we are given a simple polygonal chain⁵ P . Following elegant conventions used by Chazelle [11], as well as Kirkpatrick, Klawe, and Tarjan [33], we view the edges of P as having two distinct sides and we view P as being embedded in a sphere. A *submap* of P is the planar (i.e., spherical) subdivision that is determined by adding edges, called *chords*, from some distinguished vertices to their shadow points, defined by performing horizontal ray shooting operations from each distinguished vertex in both directions, counting only chords as dual to edges. Because of the two-sided nature of P 's boundary, a horizontal ray only hits one side of an edge. Indeed, we store the shadow points on P in two lists, the shadows on the “left” side and the shadows on the “right” side, both of which are ordered along P . By adopting the convention that P is embedded in a sphere, we view horizontal ray shooting operations that “miss” P as actually wrapping around the sphere and hitting P from the other side. (See Figure 7.)

Following Chazelle’s approach [11], our method for constructing a triangulation of P is to construct a submap of P and then refine that submap into a *trapezoidal map*, that is, the decomposition formed by adding an edge from each vertex of P to its shadows on P 's boundary. By a well-known result of Fournier and Montuno [21], constructing a trapezoidal map is linear-time equivalent to polygon triangulation. Indeed, several recent triangulation

⁵We describe our method assuming a possibly open polygonal chain, since our method is based upon applying recursion to the edges of this chain.

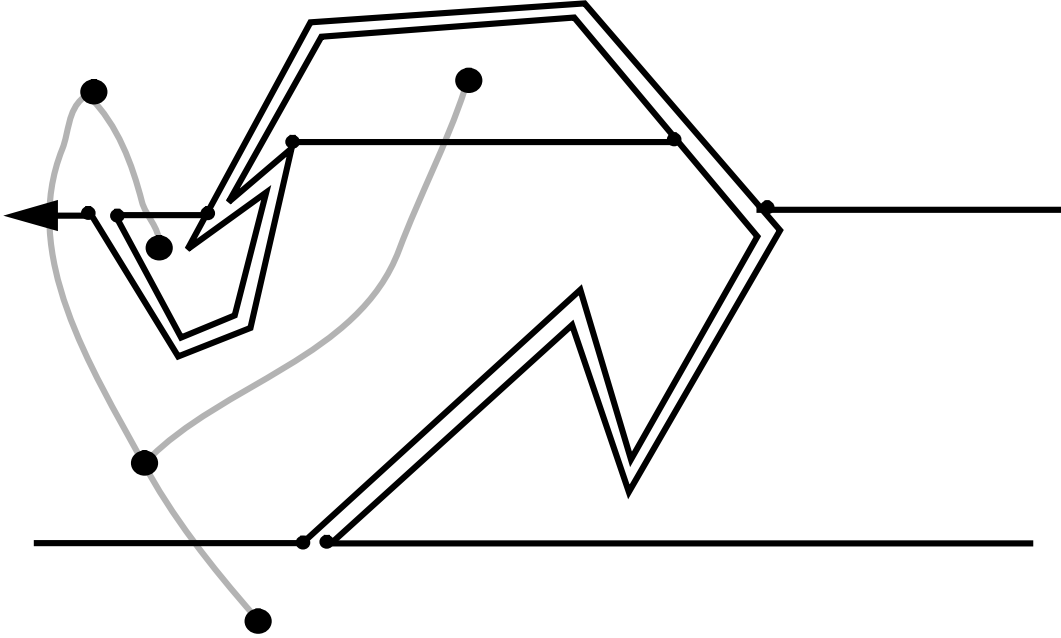


Figure 7: A 5-granular conformal submap and its dual tree.

algorithms (e.g., [11, 33, 44]) actually produce a trapezoidal map and then apply this result to construct a triangulation. In our case, we will construct a trapezoidal map and then apply an algorithm due to the author [24] to convert this trapezoidal map into a triangulation in parallel.

So, let us begin our discussion by describing the specific kind of submap we construct in the first phase of our algorithm. Let S be a submap of P , and let D be the graph-theoretic dual of S . Because of the two-sided nature of P , there are no edges in D that correspond to adjacencies that cross the boundary of P ; hence, D is a tree. Following the terminology of Chazelle [11], we say that S is *conformal* if D has degree at most 4. Each region R in S (and its associated node in D) is assigned a weight, where the *weight* of a region R is the maximum number of polygon edges on any arc in R , whereby *arc* we refer to a maximal continuous portion of P 's boundary. S is γ -*granular* if each of its regions has weight at most γ and compressing any edge e in D incident upon a node of degree 3, by removing the chord dual to e , would create a node with weight more than γ . (See Figure 7.) As the following lemma shows, conformality and granularity imply an “even distribution” of regions:

Lemma 4.2 (Chazelle [11]): *A γ -granular conformal submap of an n -edge polygonal curve P has $O(n/\gamma + 1)$ regions and each region is bounded by $O(\gamma)$ edges.*

We concentrate on submaps whose granularity is a function of n , namely, we are interested in n^δ -granular conformal submaps, where δ is some constant such that $0 < \delta < 1$. In addition

to the adjacency information for the nodes, arcs, and edges in such a submap, we require that it be augmented with a few important data structures. In particular, we say that an n^δ -granular conformal submap is *fully augmented* if it has the following data structures associated with it:

- *A (binary) centroid decomposition tree for D .* We have a binary centroid decomposition of D , the dual spanning tree D of S .
- *A ray-shooting data structure.* We have a data structure that allows for horizontal ray shooting in S to be performed in $O(\log n)$ time using $O(n^{\delta^2})$ processors.
- *A chain-cutting data structure.* We have a data structure that allows for chain-cutting operations to be performed in $O(\log n)$ time using $O(n^\epsilon)$ processors, for some constant $0 < \epsilon < 1$. Given a subchain P' of P , a *cutting* of P' is a partitioning of P' into $O(n^\epsilon)$ chains J_1, J_2, \dots, J_l such that each J_i has an associated fully augmented n_i^δ -granular conformal submap, where $n_i = |J_i|$.

Given an n -edge simple polygon P , the first phase in our algorithm is to construct a fully augmented n^δ -granular conformal submap of P , for some constant $0 < \delta < 1$, in $O(\log n)$ time using $O(n/\log n)$ processors. Our method is based on an n^ϵ -way divide-and-conquer merge where $0 < \epsilon < 1$ is a constant to be determined in the analysis⁶.

The structure of our algorithm is as follows:

1. We begin our algorithm by dividing P into subchains P_1, P_2, \dots, P_m of size $n^{1-\epsilon}$ each (so m is $O(n^\epsilon)$) and recursively constructing a (fully augmented) n_i^δ -granular submap S_i of each P_i in parallel, where $n_i = |P_i|$.
2. We merge all the submaps S_1, S_2, \dots, S_m into a single submap S' , which may not be conformal nor n^δ -granular. We implement this step by performing horizontal ray shooting queries for every chain endpoint in an S_i . This step requires $O(\log n)$ time using $O(n^{1-\delta+\delta^2+2\epsilon})$ processors.
3. We refine S' into a conformal submap S'' by adding extra chords as necessary. We implement this step by dividing each region in S' into $O(n^{3\epsilon})$ chains with fully augmented n^{δ^3} -granular conformal submaps, and then performing a parallel search for each pair of such chains to try to find a vertex on one chain that horizontally sees an edge on the other. This step requires $O(\log n)$ time using $O(n^{1-\delta+\delta^3+\delta^4+8\epsilon})$ processors.

⁶So as to relieve the suspense a bit, let us foreshadow here that $\epsilon = 1/80$ and $\delta = 7/10$ will prove to be acceptable values for these algorithmic parameters.

4. We contract S'' into an n^δ -granular conformal submap S by removing chords as necessary. We implement this step by a recursive procedure based on a centroid decomposition of the dual tree for S'' that runs in $O(\log n)$ time using $O(n^{1-\delta+4\epsilon})$ processors.
5. We conclude by constructing a ray shooting data structure for S and decomposition tree for the dual graph of S . We need not explicitly construct a chain-cutting data structure for S , however, for the recursion tree for our algorithm can be used for such operations. This is because our recursion tree is a locally n^ϵ -way tree, which implies, by an observation from Section 3.1, that any subchain of P covers $O(n^\epsilon)$ nodes in our recursion tree. This step runs in $O(\log n)$ time using $O(n^{3(1-\delta)})$ processors, and produces a structure that allows ray shooting queries to be performed in $O(\log n)$ time using $O(n^{(1-\delta)/2+\delta^4+3\epsilon})$ processors.

Let us, therefore, describe in detail how we implement each of these steps. Since Step 1 is the divide-and-recurse step, we begin with Step 2.

4.3 Merging Submaps

So, suppose we have a polygonal curve P that has been divided into $m = O(n^\epsilon)$ subcurves P_1, P_2, \dots, P_m of size $O(n^{1-\epsilon})$ each, such that, for each P_i , we have a fully augmented n_i^δ -granular conformal submap S_i , where $n_i = |P_i|$. In this section we describe how to merge all the submaps into a single submap in parallel.

For each endpoint p of a chain P_i , we perform a horizontal ray shoot with respect to each of the $O(n^\epsilon)$ other chains to determine the shadow point(s) for p with respect to P . The segments from all such p 's to their shadows define the chords in this new submap, S' , of P . For any such point p , this computation takes $O(\log n)$ time and, since each P_i has a fully augmented submap, it requires $O(n^{\delta^2})$ processors⁷ for each of the $O(n^\epsilon)$ ray shooting queries for p . Thus, since there are $O(n^{1-\delta+\epsilon})$ such endpoints, this step requires

$$O(n^{1-\delta+\delta^2+2\epsilon}) \tag{1}$$

processors in total. We then sort the shadow points with respect to their ordering around P to determine all the adjacencies between endpoints and shadow points, and also between consecutive shadow points. This can be implemented in $O(\log n)$ time using $O(n^{1-\delta+\epsilon})$ processors [13] (which is clearly dominated by (1)).

Lemma 4.3: *The number of Jordan curves (arcs) bounding any region in S' is $O(n^\epsilon)$.*

⁷For notational simplicity, we will often use n in place of $\max\{n_i : i = 1, 2, \dots, m\}$, even though this quantity is $O(n^{1-\epsilon})$. This convention will have only a marginal, albeit pessimistic, effect on our processor bounds, but it will allow us to avoid complicating the exponents in our analysis with a lot of inconsequential “ $1 - \epsilon$ ” terms. For example, the true processor bound here is $O(n^{(1-\epsilon)\delta^2})$.

Proof: Chazelle [11] shows that in any submap, S , the Jordan curves (not counting chords) bounding a region R appear in the order they occur on P . Thus, if R is a region in S' , R 's boundary contains a (possibly empty) collection of arcs from P_1 , followed by a (possibly empty) collection of arcs from P_2 , and so on, until it terminates with a (possibly empty) collection of arcs from P_m . Since the submaps for each of the P_i 's were conformal, each such P_i can contribute at most $O(1)$ arcs. Therefore, the total number of curves on R 's boundary is $O(n^\epsilon)$. \square

At this point in the algorithm we have a submap consisting of $O(n^{1-\delta+\epsilon})$ regions, where each region R is bounded by $O(n^\epsilon)$ chains of weight $O(n^\delta)$ each (because of the granularity of the recursively-computed submaps). This submap may not be conformal, however.

4.4 Achieving Conformality

So we must refine the submap S' to make it be conformal. Let R be a region in the submap. By the chain-cutting structure, we can divide each chain in R into $O(n^\epsilon)$ subchains, C_1, C_2, \dots , of size $O(n^\delta)$ each, such that each such subchain C_i has a fully augmented n^{δ^2} -granular submap built upon it. In fact, we make yet another application of chain-cutting, so as to divide each subchain C_i into $O(n^\epsilon)$ smaller chains, c_1, c_2, \dots , of size $O(n^{\delta^2})$ each, such that each c_j has a fully augmented n^{δ^3} -granular submap built upon it. This, of course, implies that each region R has been partitioned into $O(n^{3\epsilon})$ chains, each of which has a centroid decomposition tree and a ray-shooting data structure to go with it.

For each c_j we wish to determine if there is a vertex on c_j that can horizontally see some edge on c_k , for each other c_k . We call each such chord a *valid* chord. To locate all the valid chords, we perform a globally $O(n^\epsilon)$ -way search down the centroid decomposition tree B_j , for c_j , to drive a search for a visible edge on c_k , which we perform for each c_k in parallel. Of course, B_j is a binary tree, so we implement this by searching $\epsilon \log n$ levels down the decomposition tree for c_j and performing a “probe” for each of the $O(n^\epsilon)$ nodes on that level. The probe that we must perform in this case is that we have a horizontal chord ab , determined by a centroid in B_j , and we wish to find the first edge in R that is hit by the ray \vec{ab} (with a as the head). Given such a chord ab , we perform a ray shooting query against each of the $O(n^{3\epsilon})$ subchains in R . Each such ray-shooting query requires $O(\log n)$ time using $O(n^{\delta^4})$ processors (because of the ray-shooting structure that accompanies each c_k). By a local test with respect to this shadow point, we can determine which node in the decomposition tree from which to continue our search for a visible edge of c_k (which we determine for each c_k in parallel). This local test is similar to a binary-search test given by Chazelle [11] and is based on the fact that the edges bounding R occur in the same order they occur on P 's boundary. We leave the details of this test to the reader.

In searching down B_j for some c_k , we must perform $O(1)$ rounds of local tests, where a single round consists of $O(n^\epsilon)$ ray shooting queries being performed in parallel. The total overhead in setting up these searches requires only $O(\log n)$ time, and after performing these $O(1)$ probes, each of which requires $O(\log n)$ time, we will reach a leaf in B_j . Performing these internal-node probes for all the $O(n^{3\epsilon})$ chains in R requires $O(n^{\delta^4+4\epsilon})$ processors. Upon reaching a leaf node v in B_j in the search for a valid chord to c_k , we must then perform a ray shooting query for each vertex on the chain associated with v . There are $O(n^{\delta^3})$ vertices on this chain, and each requires $O(n^{\delta^4+3\epsilon})$ processors to answer its query. Therefore, we can implement the leaf-level searches for all the $O(n^{3\epsilon})$ chains in R in $O(\log n)$ time using $O(n^{\delta^3+\delta^4+6\epsilon})$ processors. This implies that we can implement all the searches (for both the internal-node and the leaf-node ray shooting queries) in $O(\log n)$ time using $O(n^{\delta^3+\delta^4+7\epsilon})$ processors. Performing the ray shooting queries for all the regions in S' , therefore, requires $O(\log n)$ time using

$$O(n^{1-\delta+\delta^3+\delta^4+8\epsilon}) \tag{2}$$

processors in total.

Lemma 4.4: *Adding all valid chords to S' using the above method creates a submap S'' that is conformal and contains at most $O(n^{1-\delta+4\epsilon})$ arcs.*

Proof: Suppose there is region R in S' with whose boundary contains the sequence of curves c_1, c_2, \dots, c_l , with $l > 4$. Chazelle [11] shows that in such a region there are a pair of curves c_j and c_k on R that contain two mutually horizontally-visible points with $|k - j| > 1$. But our method finds all pairs of curves c_j and c_k with two mutually horizontally-visible points and adds a chord for such a pair. Therefore, no such R can exist in S' . As for the total number of arcs, note that before we added all of these extra chords, we had $O(n^{1-\delta+\epsilon})$ regions in our submap. Moreover, by Lemma 4.3 and the fact that we decomposed each region twice using the chain-cutting structure, each region in this submap had $O(n^{3\epsilon})$ arcs. Since each region is topologically equivalent to a circle and the added chord topologically equivalent to non-intersecting chords in this circle, the total number of extra chords we can add per region is $O(n^{3\epsilon})$. \square

4.5 Achieving n^δ -Granularity

Having constructed a conformal submap S'' , we must then turn it into an n^δ -granular submap S . We do this by recursion on a centroid decomposition of the dual tree D'' of S'' . That is, we find a centroid edge e in D'' , disconnect D'' at e , recursively contract the two subtrees this creates, and then (by a local test) determine if we should compress e (by removing the

chord dual to e). After a preprocessing step that computes a centroid decomposition of D'' in $O(\log n)$ time [15], this can easily be implemented in $O(\log n)$ time, using

$$O(n^{1-\delta+4\epsilon}) \tag{3}$$

processors.

So, we now have an n^δ -granular conformal submap. We have only to augment it with the necessary data structures.

4.6 Augmenting the Submap

Recall that our submap must be augmented with a centroid decomposition tree for D , a ray-shooting data structure, and a chain-cutting data structure. Now that we have D , the dual tree for S , the first of these is fairly straightforward to construct in $O(\log n)$ time using

$$O(n^{1-\delta} / \log n) \tag{4}$$

processors [15].

The second structure, used for ray shooting queries, requires a little more effort, however. Each chain in S consists of $O(n^\delta)$ edges; hence, by the chain-cutting structure, each such chain can be partitioned into $O(n^\epsilon)$ subchains of size $O(n^\delta)$ each, such that each subchain has a fully augmented n^{δ^2} -granular conformal submap. In fact, each such subchain can be further partitioned into $O(n^\epsilon)$ chains of size $O(n^\delta)$ each, such that each has a fully augmented n^{δ^3} -granular conformal submap. This implies that we can perform a ray-shooting query against any Jordan chain in S in $O(\log n)$ time using $\rho(n) = O(n^{\delta^4+2\epsilon})$ processors. Actually, if one desires a more formal characterization of the running time of a query, then one can let $T_q(n)$ denote its running time, where $T_q(n) \leq T_q(n^\delta) + b \log n$, for some constant $b > 0$, which implies that $T_q(n)$ is $O(\log n)$.

This gives us a collection of Jordan chains for which we would like to apply Theorem 4.1 to derive a point-location structure, but that theorem requires that we view S as a Jordan tessellation; hence, we must merge the list of shadow points on the “left” side of P with the list of shadow points on the “right” side of P so as to produce a dual graph G for S (viewed as a Jordan Tessellation). Because S is an n^δ -granular conformal submap, G has size $O(n^{1-\delta})$; hence, applying Theorem 4.1 to the Jordan tessellation induced by G to construct a ray shooting data structure for S (in $O(\log n)$ time) requires

$$O(n^{3(1-\delta)}) \tag{5}$$

processors. This structure allows ray shooting queries to be answered in $O(\log n)$ time using

$$O(n^{(1-\delta)/2+\delta^4+3\epsilon}) \tag{6}$$

processors, also by Theorem 4.1. Note that in order to satisfy the induction invariant for the ray shooting query, we must choose the appropriate values for ϵ and δ so that (6) is $O(n^{\delta^2})$. Assuming such values can be chosen (which we show below), this completes our construction of an n^δ -granular conformal submap of P .

Let us, therefore, analyze the time and processor bounds for this method. The time bounds are determined by the recurrence relation $T(n) = T(n^{1-\epsilon}) + b \log n$, for some constant $b > 0$, which implies that $T(n)$ is $O(\log n)$. If we desire that our merging procedure uses only $O(n^{1-\alpha})$ processors for some constant $0 < \alpha < 1$, then the processor bounds specified above in Equations (1)–(5) imply the following non-redundant⁸ constraints for ϵ and δ :

$$1 - \delta + \delta^2 + 2\epsilon < 1 \tag{7}$$

$$1 - \delta + \delta^3 + \delta^4 + 8\epsilon < 1 \tag{8}$$

$$3(1 - \delta) < 1. \tag{9}$$

Moreover, we need to satisfy the induction invariant that the ray shooting data structure accompanying our submap requires only $O(n^{\delta^2})$ processors, which, by the above claim, implies that

$$(1 - \delta)/2 + \delta^4 + 3\epsilon \leq \delta^2. \tag{10}$$

There are, in fact, an infinite number of possible assignments for ϵ and δ that satisfy Equations (7)–(10). For example, one possibility is to set $\epsilon = 1/80$ and $\delta = 7/10$. Thus, we have the following:

Lemma 4.5: *Suppose one is given an n -vertex polygonal chain P partitioned into $O(n^\epsilon)$ subchains P_1, P_2, \dots, P_m , such that each P_i has an associated fully augmented n_i^δ -granular submap (where $n_i = |P_i|$), for some positive constants δ and ϵ that satisfy Equations (7)–(10). Then one can construct a fully augmented n^δ -granular conformal submap for P in $O(\log n)$ time using $O(n^{1-\alpha})$ processors in the CRCW PRAM model, for some constant $\alpha > 0$.*

Using this lemma to drive our divide-and-conquer algorithm, then, gives us the following theorem:

Theorem 4.6: *Given an n -vertex polygonal chain P , one can construct a fully augmented n^δ -granular conformal submap for P , for some positive constant $0 < \delta < 1$, in $O(\log n)$ time using $O(n/\log n)$ processors in the CRCW PRAM model, by the above n^ϵ -way divide-and-conquer algorithm.*

⁸Equations (3) and (4) are subsumed by the other constraints.

Proof: We have already established the time bound. If we let $W(n)$ denote the work performed by our algorithm then $W(n) \leq \sum W(n_i) + bn^{1-\alpha} \log n$, for some constants $b > 0$ and $0 < \alpha < 1$, where n_i is $O(n^{1-\epsilon})$. For the base case, when n is below some constant, then we construct a fully augmented n^δ -granular submap for P using the sequential algorithm of Chazelle [11]. This implies that $W(n)$ is $O(n)$. Having established the work bound to be $O(n)$, we may then make a simple application of Brent’s Theorem [9] (which is also known as the *work-time* scheduling principle [31]) to establish the processor bounds. In order to apply this theorem we must be able to satisfy two conditions: (1) we must be able to determine the number “real” computations being performed in each step, and (2) we must be able to map these computations to the $O(n/\log n)$ processors we are using in the simulation. In this case we can satisfy these conditions by either an ad-hoc method based on the recurrence relation for $W(n)$ or by applying the duration-unknown task scheduling method of Cole and Vishkin [14] or Cole and Zajicek [16]. \square

4.7 The Trapezoidal Map

Of course, we wish to construct a trapezoidal map, not merely an n^δ -granular conformal submap. That is, we desire the subdivision of P determined by the chords produced by a horizontal ray shooting operation from each vertex on P . In this section we show how to use the procedures outlined above to refine such a submap into a trapezoidal map. Our method runs in $O(\log n)$ time using $O(n/\log n)$ processors on a CRCW PRAM.

We begin by constructing a fully augmented n^δ -granular conformal submap S using the algorithm of the previous section. In fact, let us view this as a preprocessing step. Each region R in S consists of $O(1)$ polygonal chains C_1, C_2, \dots, C_l , each of which contains $O(n^\delta)$ edges. Because we have constructed a fully augmented submap, we can apply the chain-cutting structure to partition each such C_i into $O(n^\epsilon)$ subchains of size $O(n^\delta)$ each, such that each subchain has an associated n^{δ^2} -granular conformal submap. We may therefore apply Lemma 4.5 to construct an n^{δ^2} -granular conformal submap of R . Our algorithm, then, iteratively applies chain-cutting and Lemma 4.5 to the newly created regions until we have a trapezoidal map for P . If we view this iterative algorithm as a recursive procedure, then we may characterize the running time as $T(n) = T(n^\delta) + O(\log n)$, which implies that $T(n)$ is $O(\log n)$. Also, the work bound is characterized by $W(n) = \sum W(n_i) + n^{1-\alpha} \log n$, for some constants $b > 0$ and $0 < \alpha < 1$, which implies that $W(n)$ is $O(n)$. Thus, we have the following theorem:

Theorem 4.7: *Given an n -vertex simple polygon P , one can triangulate P in $O(\log n)$ time using $O(n/\log n)$ processors in the CRCW PRAM model.*

Proof: Apply the above algorithm to produce a trapezoidal map of P , and then apply the result of the author [24] to refine this into a triangulation. \square

An immediate consequence of this theorem is that it eliminates the bottleneck computation in the algorithm of Goodrich *et al.* [25] so that one can now preprocess a polygon P in $O(\log n)$ time using $O(n/\log n)$ processors so as to answer shortest path queries inside P in $O(\log n)$ time using a single processor.

5 Conclusion

We have given optimal algorithms for sequentially constructing planar separator decompositions and triangulating a simple polygon in parallel, both of which are problems with many applications. Indeed, our planar separator result has been used recently by Eppstein *et al.* [18] in improved methods for dynamic planar graph algorithms, and our parallel polygon triangulation result has been used recently by Hershberger [30] for improved parallel computational geometry algorithms.

Although our methods for solving these two problems are quite different, they are both based upon similar paradigms. One of these paradigms is the dynamic maintenance of a planar graph using a spanning tree and its inverse. Indeed, this tree interlacing technique has already proved useful for solving a number of other problems as well [19, 26]. The second main paradigm our two algorithms share is that they both employ the divide-and-conquer technique, but in a slightly non-standard way. In particular, in addition to the usual subproblem information passed to recursive calls, both of our algorithms also pass rather sophisticated data structures already built on the elements that comprise these subproblems. This technique is certainly not new to this paper, for it is used in such methods as Chazelle's algorithm for polygon triangulation [11] and the centroid decomposition algorithm of Guibas *et al.* [27], but its use in this paper provides further evidence of its power. It is sure to appear again in future divide-and-conquer algorithms.

References

- [1] A. Aggarwal, B. Chazelle, L. Guibas, C. Ó'Dúnlaing, and C. Yap, "Parallel Computational Geometry," *Algorithmica*, **3**(3), 1988, 293–328.
- [2] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *Data Structures and Algorithms*, Addison-Wesley (Reading, Mass.: 1983).
- [3] R.J. Anderson and G.L. Miller, "Deterministic Parallel List Ranking," *Lecture Notes in Computer Science, 319: 3rd Aegean Workshop on Computing, AWOC 88*, Springer-Verlag, 1988, 81–90.
- [4] B.G. Baumgart, "A Polyhedron Representation for Computer Vision," *Proc. 1975 AFIPS National Computer Conf.*, **44**, AFIPS Press, 1975, 589–596.

- [5] J.L. Bentley and D. Wood, “An Optimal Worst Case Algorithm for Reporting Intersections of Rectangles,” *IEEE Trans. on Computers*, **C-29**(7), 1980, 571–576.
- [6] O. Berman, D. Breslauer, Z. Galil, B. Schieber, and U. Vishkin, “Highly Parallelizable Problems,” *Proc. 21st ACM Symp. on Theory of Computing*, 1989, 309–319.
- [7] S.N. Bhatt and F.T. Leighton, “A Framework for Solving VLSI Graph Layout Problems,” *J. Comp. and Sys. Sci.*, **28**(2), 1984, 300–343.
- [8] J.A. Bondy and U.S.R. Murty, *Graph Theory with Applications*, North-Holland (New York: 1976).
- [9] R.P. Brent, “The Parallel Evaluation of General Arithmetic Expressions,” *J. ACM*, **21**(2), 1974, 201–206.
- [10] B. Chazelle, “A Theorem on Polygon Cutting with Applications,” *Proc. 23rd IEEE Symp. on Foundations of Computer Science*, 1982, 339–349.
- [11] B. Chazelle, “Triangulating a Simple Polygon in Linear Time,” *Disc. and Comp. Geom.*, **6**, 1991, 485–524.
- [12] K.L. Clarkson, R. Cole, and R.E. Tarjan, “Randomized Parallel Algorithms for Trapezoidal Diagrams,” *Int. J. of Computational Geometry and Applications*, **2**(2), 1992, 117–134.
- [13] R. Cole, “Parallel Merge Sort,” *SIAM J. Comput.*, **17**(4), 1988, 770–785.
- [14] R. Cole and U. Vishkin, “Approximate Parallel Scheduling, Part I: The Basic Technique with Applications to Optimal Parallel List Ranking in Logarithmic Time,” *SIAM J. Comput.*, **17**(1), 1988, 128–142.
- [15] R. Cole and U. Vishkin, “The Accelerated Centroid Decomposition Technique for Optimal Parallel Tree Evaluation in Logarithmic Time,” *Algorithmica*, **3**, 1988, 329–346.
- [16] R. Cole and O. Zajicek, “An Optimal Parallel Algorithm for Building a Data Structure for Planar Point Location,” *J. Par. and Dist. Comput.*, **8**, 1990, 280–285.
- [17] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*, MIT Press (Cambridge, Mass.: 1990).
- [18] D. Eppstein, Z. Galil, G. Italiano, T. Spencer, “Separator Based Sparsification for Dynamic Planar Graph Algorithms,” *Proc. 25th ACM Symp. on Theory of Computing*, 1993.
- [19] D. Eppstein, G.F. Italiano, R. Tamassia, R.E. Tarjan, J. Westbrook, and M. Yung, “Maintenance of a Minimum Spanning Forest in a Dynamic Planar Graph,” *J. of Algorithms*, **13**, 1992, 33–54.
- [20] G.N. Frederickson, “Fast Algorithms for Shortest Paths in Planar Graphs, with Applications,” *SIAM J. Comput.*, **6**, 1987, 1004–1022.
- [21] A. Fournier and D.Y. Montuno, “Triangulating Simple Polygons and Equivalent Problems,” *ACM Trans. on Graphics*, Vol. 3, No. 2, April 1984, pp. 153–174.
- [22] M.R. Garey, D.S. Johnson, F.P. Preparata, and R.E. Tarjan, “Triangulating a Simple Polygon,” *Information Processing Letters*, **7**(4), 1978, 175–179.
- [23] H. Gazit and G.L. Miller, “A Parallel Algorithm for Finding a Separator in Planar Graphs,” *Proc. 28th IEEE Symp. on Foundations of Computer Science*, 1987, 238–248.
- [24] M.T. Goodrich, “Triangulating a Polygon in Parallel,” *J. of Algorithms*, **10**, 1989, 327–351.

- [25] M.T. Goodrich, S. Shauck, and S. Guha, “Parallel Methods for Visibility and Shortest Path Problems in Simple Polygons,” *Proc. 6th ACM Symp. on Computational Geometry*, 1990, 73–82.
- [26] M.T. Goodrich and R. Tamassia, “Dynamic Trees and Dynamic Point Location,” *Proc. 23rd ACM Symp. on Theory of Computing*, 1991, 523–533.
- [27] L. Guibas, J. Hershberger, D. Leven, M. Sharir and R. Tarjan, “Linear Time Algorithms for Visibility and Shortest Path Problems Inside Triangulated Simple Polygons,” *Algorithmica*, **2**, 1987, 209–233.
- [28] L.J. Guibas and R. Sedgewick, “A Dichromatic Framework for Balanced Trees,” *Proc. 19th IEEE Symp. on Foundations of Computer Science*, 1978, 8–21.
- [29] L.J. Guibas and J. Stolfi, “Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams,” *ACM Transactions on Graphics*, Vol. 4, 1985, 75–123.
- [30] J. Hershberger, “Optimal Parallel Algorithms for Triangulated Simple Polygons,” *Proc. 8th ACM Symp. on Computational Geometry*, 1992, 33–42.
- [31] J. JáJá, *An Introduction to Parallel Algorithms*, Addison-Wesley (Reading, Mass.), 1992.
- [32] Karp, R.M., and Ramachandran, V., “Parallel Algorithms for Shared-Memory Machines,” in *Handbook of Theoretical Computer Science, Vol. A: Algorithms and Complexity*, ed., J. Van Leeuwen, The MIT Press (Cambridge, Mass.), 1990, 869–942.
- [33] D.G. Kirkpatrick, M.M. Klawe, and R.E. Tarjan, “Polygon Triangulation in $O(n \log \log n)$ time with Simple Data Structures,” *Proc. 6th ACM Symp. on Computational Geometry*, 1990, 34–43.
- [34] C. Kruskal, L. Rudolph, and M. Snir, “The Power of Parallel Prefix,” *Proc. 1985 IEEE Int. Conf. on Parallel Proc.*, 180–185.
- [35] R.E. Ladner and M.J. Fischer, “Parallel Prefix Computation,” *J. ACM*, 1980, 831–838.
- [36] C.E. Leiserson, “Area-Efficient Graph Layouts (for VLSI),” *Proc. 21st IEEE Symp. on Foundations of Computer Science*, 1980, 270–281.
- [37] R.J. Lipton and R.E. Tarjan, “A Separator Theorem for Planar Graphs,” *SIAM J. Appl. Math.*, **36**(2), 1979, 177–189.
- [38] R.J. Lipton and R.E. Tarjan, “Applications of a Planar Separator Theorem,” *SIAM J. Comput.*, **9**(3), 1980, 615–627.
- [39] G.L. Miller, “Finding Small Simple Cycle Separators for 2-Connected Planar Graphs,” *J. Comp. and Sys. Sci.*, **32**, 1986, 265–279.
- [40] D.E. Muller and F.P. Preparata, “Finding the Intersection of Two Convex Polyhedra,” *Theoretical Computer Science*, Vol. 7, No. 2, October 1978, 217–236.
- [41] F.P. Preparata and M.I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, New York, NY, 1985.
- [42] D.D. Sleator and R.E. Tarjan, “A Data Structure for Dynamic Trees,” *J. Comput. and Sys. Sci.*, **26**, 362–391, 1983.
- [43] R.E. Tarjan, *Data Structures and Network Algorithms*, SIAM, Philadelphia, PA, 1983.
- [44] R.E. Tarjan and C.J. Van Wyk, “An $O(n \log \log n)$ -time Algorithm for Triangulating a Simple Polygon,” *SIAM J. Comput.*, **17**, 1988, 143–178.

- [45] R.E. Tarjan and U. Vishkin, “Finding Biconnected Components and Computing Tree Functions in Logarithmic Parallel Time,” *SIAM J. Comput.*, **14**, 1985, 862–874.
- [46] C.K. Yap, “Parallel Triangulation of a Polygon in Two Calls to the Trapezoidal Map,” *Algorithmica*, **3**, 1988, 279–288.