Algorithmica (1992) 8:461–486

Algorithmica

Parallel Methods for Visibility and Shortest-Path Problems in Simple Polygons[1]

Michael T. Goodrich,[2] Steven B. Shauck,[3] and Sumanta Guha[4]

Abstract. In this paper we give efficient parallel algorithms for solving a number of visibility and shortest-path problems for simple polygons. Our algorithms all run in $O(\log n)$ time and are based on the use of a new data structure for implicitly representing all shortest paths in a simple polygon P, which we call the *stratified decomposition tree*. We use this approach to derive efficient parallel methods for computing the visibility of P from an edge, constructing the visibility graph of the vertices of P (using an output-sensitive number of processors), constructing the shortest-path tree from a vertex of P, and determining all-farthest neighbors for the vertices in P. The computational model we use is the CREW PRAM.

Key Words. Parallel algorithms, Computational geometry, Data structures, Visibility, Polygons, CREW PRAM.

1. Introduction.

Problems involving visibility and shortest paths in simple polygons form a well-studied class of problems in computational geometry (e.g., see [4], [6], [10], [16], [20], [22], [23], [27], [28], [30], and [39]). Given a simple polygon P, these problems deal with properties of P relative to the *geodesic* metric [27], [28], [44], which is defined so that the distance between two points p and q is the length of the shortest path between p and q that does not cross the boundary of P. If the shortest path between p and q is a straight line, then we say that p is *visible* from q.

We briefly review some of the previous sequential results for these types of problems (the reader is referred to [20] and [39] for other references). Some notable early examples of methods for solving such problems include the $O(n)$-time solution of ElGindy and Avis [22] for computing the visibility polygon from a point, the $O(n)$-time solution of Avis and Toussaint [10] for determining if a simple polygon is weakly visible from a distinguished edge e (i.e., if every point in P is visible from some part of e), and the $O(n \log n)$-time method of Lee and Preparata [34] for determining the shortest path between two points in a simple polygon.

[1] This research was announced in preliminary form in the *Proceedings of the 6th ACM Symposium on Computational Geometry*, 1990, pp. 73–82. The research of Michael T. Goodrich was supported by the National Science Foundation under Grants CCR-8810568 and CCR-9003299, and by the NSF and DARPA under Grant CCR-8908092.

[2] Department of Computer Science, Johns Hopkins University, Baltimore, MD 21218, USA.

[3] Westinghouse Electric Corporation, P.O. Box 746, M/S 463, Baltimore, MD 21203, USA.

[4] Department of Electrical Engineering and Computer Science, University of Wisconsin at Milwaukee, Milwaukee, WI 53201, USA.

Received October 6, 1990; revised April 28, 1991. Communicated by Mikhail J. Atallah.

More recently, Chazelle and Guibas [16] have shown that the portion of a simple polygon weakly visible from an edge in $O(n \log n)$ time can be computed. In fact, they show that a data structure can be constructed in $O(n \log n)$ time that can be used to determine the first point on P that is hit by a query ray $\vec{r}$ in $O(\log n)$ time. These time complexities are improved to $O(\tau(n) + n)$ by Guibas *et al.* [28], where $\tau(n)$ is the time complexity of polygon triangulation [24], [48]. Addressing the shortest-path problem more specifically, Guibas and Hershberger [27] show that a data structure can be constructed in $O(\tau(n) + n)$ time that can be used to compute the length of the shortest path between two points in P in $O(\log n)$ time. In [30] Hershberger shows that the visibility graph for the vertices of P in $O(\tau(n) + n + k)$ time, where k is the number of edges, can be constructed. Recall that this is the graph defined so that (v, w) is an edge if and only if v is visible from w. Recently, Chazelle [14] has shown that $\tau(n)$ is $O(n)$, implying that all these time complexities involving $\tau(n)$ are linear.

In this paper we investigate the parallel complexity of solving visibility and shortest-path problems for simple polygons. In this domain the number of previous results is not as large as in the sequential case. In [7] Atallah *et al.* show that the portion of the plane visible from a point in a collection of line segments can be computed in $O(\log n)$ time using $O(n)$ processors. The number of processors needed for this problem was improved by Atallah and Chen [6] to $O(n/\log n)$ for the case when the line segments form a simple polygon. In [23] ElGindy and Goodrich show that the shortest path between two points in a simple polygon can be determined in $O(\log n)$ time using $O(n)$ processors, and the shortest-path tree from a vertex v (the union of all shortest paths from v to other vertices) can be constructed in $O(\log^2 n)$ time using $O(n)$ processors.

In this paper we give efficient parallel algorithms for a number of visibility and shortest-path problems for simple polygons. All of our algorithms run in $O(\log n)$ time with an efficient number of processors. The need for these new methods is based on the observation that a straightforward parallelization of any of the previous sequential methods seems to be either impossible or require $O(\log^2 n)$ time. Specifically, given a simple polygon P, we derive the following results:

Shortest-path queries. We show how to build a data structure, called the *stratified decomposition tree*, in $O(\log n)$ time using $O(n)$ processors ($O(n/\log n)$ processors if P is triangulated), that allows us to construct an implicit representation of the shortest path between two points inside P in $O(\log n)$ time.

Visibility from an edge. We show how to determine the subpolygon of P consisting of all points that are weakly visible from some distinguished edge e in $O(\log n)$ time using $O(n)$ processors.

Visibility graph. We show how to construct the visibility graph for the vertices of P in $O(\log n)$ time using $O(n \log n + k/\log n)$ processors, where k is the number of edges in the graph. Our method involves exploiting a duality between the problem of computing all visibility-graph edges that intersect a diagonal of P and the problem of computing all intersections between two sets of nonintersecting segments. We develop a new parallel method for solving this intersecting pairs problem in $O(\log n)$ time using $O(n + k/\log n)$ processors, which is optimal, where k is the number of answers.

Applications. We give a number of other applications that utilize our shortest-

paths data structure. These include computing the shortest-path tree from a vertex, computing the convex ropes for P, and finding all-farthest neighbors for the vertices of P.

All of our results are in the CREW PRAM model, the synchronous parallel model where processors share a common memory in which simultaneous access to a memory cell is allowed only if all the accesses are reads. Results that use an output-sensitive number of processors assume the weakest form of processor-allocation, namely, that processors must be allocated *globally* [26]. In this scheme we allow r new processors to be allocated in step t only if we have already constructed an r-element array that stores pointers to the r tasks these processors are to begin performing in step $t + 1$.

In the next section we give some preliminary definitions and observations. In Section 3 we describe the stratified decomposition tree and how it can be used to answer shortest-path queries. In Section 4 we show how to compute visibility from an edge, and in Section 5 we show how to use our method for visibility from an edge to construct the visibility graph for the vertices of P. We give several applications of our approach in Section 6.

2. Preliminaries. In this section we review a number of geometric observations concerning concepts relevant to visibility and shortest-path problems. We also introduce some miscellaneous algorithmic techniques, which we employ at various places in our solutions. We begin with some geometric observations.

2.1. Point-Line Duality. One of the main tools in our constructions involves the well-known concept of point-line duality [20], [21], [29], [37], [39], [43]. In particular, we use the framework of the two-sided plane [29], [43]. The *two-sided plane* is an extension of the normal plane in that each point in the plane becomes two points in the two-sided plane, one on the "top side" and one on the "bottom side." Each line in the classical plane can map to one of two lines in the two-sided plane, which differ only in orientation. The two-sided plane has a well-known dualization where a point p maps to a line $\mathscr{D}_p$, where, if we let d denote the distance from p to the origin, then $\mathscr{D}_p$ is the line perpendicular to l at a distance $1/d$ from the origin on the opposite side of the origin from p. $\mathscr{D}_p$ is defined to be oriented so that the origin is on its left (resp. right) relative to the top side of the plane if p is on the top (resp. bottom) side of the plane. The dualization maps lines to points by using the reverse of the above, thus the dualization is self-inverting, i.e., $\mathscr{D}_{\mathscr{D}p} = p$. This dualization maps convex polygons to convex polygons [29], [43].

Using this point-line duality, we can change a line-oriented problem to a point-oriented problem, and in so doing, possibly gain some insights that allow us to apply a more efficient (point-oriented) algorithm.

2.2. The Polygon-Cutting Theorem and Decomposition Trees. The last algorithmic topic we discuss in this section involves the use of another type of duality; namely, the relationships between planar graphs and their graph-theoretic planar duals.

We begin with some definitions. Let T be a height-balanced binary tree, and,

for each node v, let n_v denote the number of descendants v has (including itself). Given a positive integer, d, we define the *d-contraction* of T to be the tree T' that results by contracting to a single node each subtree rooted at a node v, provided both of v's children have less than d descendants while v has at least d descendants. The notion of a d-contraction plays heavily in several of our solutions.

As in many of the previous sequential algorithms, all of our algorithms assume that the input polygon P has been *triangulated*, i.e., P has been augmented by adding diagonals (between vertices of P) so as to partition the interior of P into triangles. If P is not given in this form, then we apply the following lemma, discovered independently by Goodrich [25] and Yap [50]:

LEMMA 2.1 [25], [50]. *Given a simply polygon P, we can triangulate P in $O(\log n)$ time using $O(n)$ processors in the CREW PRAM model.*

Given a triangulation of P, we define a tree D to be the (graph-theoretic) planar dual of the triangulation, excluding the exterior face of P. Since D is built upon a triangulation of a simple polygon, it has degree 3. By the polygon-cutting theorem of Chazelle [12], there is an edge of D that, when removed, divides D into two subtrees such that the subpolygon associated with each subtree has at least one-third and most two-thirds as many vertices as P. If we then recursively repeat this division on each subtree we define a binary tree, T, where each node v in T represents a subtree of D which we denote by D_v, and v's children represent the two pieces D_v was cut into. For each node v in T we let n_v denote the number of vertices in P_v. We refer to the tree T as the *centroid decomposition tree* for P.

Each subtree D_v implicitly defines a subpolygon P_v of P: namely, the union of the triangles corresponding to the vertices of D_v. For each node v in T the *diagonal for v*, denoted e_v, is the diagonal in P that separates P_u and P_w where u and w are the children of v. Note that an edge e on the boundary of the polygon P_v is either an edge on the boundary of P or a diagonal of P in the triangulation. If an edge e on the boundary of P_v is a diagonal of P, we call e a *pseudoedge* of P_v.

OBSERVATION 2.2. *For any v in T, P_v has at most d_v pseudoedges, where d_v is the depth of v in T.*

Before we can describe our shortest-path data structure, we need to have a way to construct a centroid decomposition tree efficiently in parallel. The essential computation in this construction involves what is essentially a recursive determination of the centroid in a degree-3 tree, a computation that can be done in $O(n)$ time sequentially [12]. In fact, as Guibas *et al.* [28] show, we can perform the entire recursive decomposition of D to construct T in $O(n)$ time by maintaining the appropriate data structures. Unfortunately, a straightforward parallelization of their method leads to a running time of $O(\log^2 n)$ using $O(n/\log^2 n)$ processors, which is too slow for our purposes. Nevertheless, using the "accelerated centroid decomposition" scheme of Cole and Vishkin [18], we can achieve $O(\log n)$ time for the construction while using only $O(n/\log n)$ processors, giving us the following lemma:

LEMMA 2.3. *Given a triangulated simply polygon P, we can produce a centroid decomposition tree T for P in $O(\log n)$ time using $O(n/\log n)$ processors in the CREW PRAM model.*

3. Shortest-Path Queries. Having given our preliminary definitions and observations, in this section we show how to construct a data structure in parallel efficiently for quickly answering shortest-path queries. We begin by reviewing a well-known sequential result in this area.

LEMMA 3.1. *Given a triangulated n-node simple polygon P and two points p and q inside P, we can construct a binary tree representation of the shortest path between p and q inside P in $O(n)$ time.*

PROOF. The method of Lee and Preparata [34] can be used to construct a list representation of the path between p and q in P in $O(n)$ time. Given this list, it is a simple matter to construct a binary tree representation of this path. □

This result is clearly optimal if we do not allow the polygon P to be preprocessed for answering such queries. In our case, however, we do allow for preprocessing. Nevertheless, we apply the above lemma in our solution whenever we must solve a shortest-path query in a small ($O(\log n)$-sized) subpolygon. In particular, we make use of a d-contraction of T, the centroid decomposition tree for P. Note that, by the polygon-cutting theorem, for each leaf v in T', we have $d \leq n_v \leq 3d$ in such a case. The leaves of such a tree correspond to "small' subpolygons. For larger subpolygons we rely on the use of geometric structures known as hourglasses.

3.1. Hourglasses. An *hourglass* [23], [27] is a geometric structure defined on two diagonals of P. If $e = (a, d)$ and $f = (b, c)$ are the two diagonals, where the vertices occur in the order a, b, c, d in a clockwise listing of the vertices of P, then the hourglass between e and f, denoted $H(e, f)$, is the union of the shortest geodesic path from a to b and the shortest geodesic path from c to d. An hourglass is *open* if it consists of two disjoint inwardly convex chains. An hourglass is *closed* if it consists of two inwardly convex chains meeting at a point, followed by a path, followed by two inwardly convex chains again. (See Figure 1.)

We represent hourglasses as two chains, each of which is stored in the leaves of a height-balanced binary tree (e.g., an AVL-tree [35] or a red-black tree [46]). By storing labels at the internal nodes of these binary trees as in [38], we can derive the following lemmas:

LEMMA 3.2. *Given two hourglasses $H(e, f)$ and $H(f, g)$, where f is a shared edge, represented as above, a single processor can construct a representation of $H(e, g)$ in $O(\log n)$ time (without modifying the representation of $H(e, f)$ and $H(f, g)$), in the CREW PRAM model.*

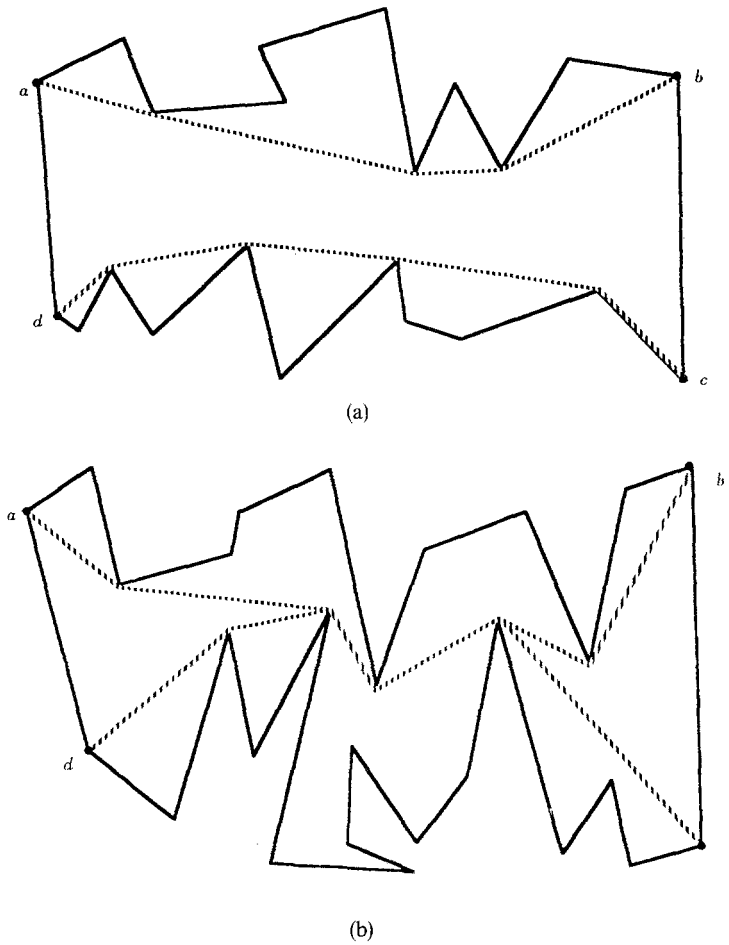


Fig. 1. An open (a) and closed (b) hourglass.

PROOF. The essential operation in "concatenating" two hourglasses is the determination of the (supporting and crossing) tangent lines between the four convex chains involved (see [23] and [27]). We can use the binary search method of Overmars and Van Luewen [38] to find each such tangent in $O(\log n)$ time. Given these tangents it is then a simple matter to split out the irrelevant portions of the chains $H(e,f)$ and $H(f,g)$, and concatenate the relevant portions to form $H(e,g)$. Since we are representing hourglasses in height-balanced binary trees, we can avoid the modification of $H(e,f)$ and $H(f,g)$ by creating new tree nodes to replace any that would have otherwise wished to modify. Clearly, this adds at most $O(\log n)$ nodes. □

In our parallel scheme we must also be able to concatenate chains of hourglasses, however.

LEMMA 3.3. *Given a chain of hourglasses* $H(e_1, e_2)$, $H(e_2, e_3), \ldots, H(e_{m-1}, e_m)$, *where each consecutive pair of hourglasses shares a common edge, we can construct a representation of* $H(e_1, e_m)$ *in* $O(\log n)$ *time and* $O(m \log n)$ *additional space using* $O(m^2)$ *processors in the CREW PRAM model, where* n *is the total size of all the hourglasses.*

PROOF. The main idea is to determine the surviving portions of each hourglass and then concatenate them to form $H(e_1, e_m)$. This resembles the approach of Atallah and Goodrich [8] and Aggarwal *et al.* [3] for computing convex hulls, but is necessarily more involved, since hourglasses are more complicated than convex hulls. Let $B_{i,j}$ (resp. $T_{i,j}$) denote the bottom (resp. top) chain of $H(e_i, e_j)$. Our approach is as follows. We determine, for each $B_{i,i+1}$, the surviving portion of $B_{i,i+1}$ in $B_{i,m}$, as well as the edge r_i, connecting this surviving portion to the next vertex (to the right) in $B_{i,m}$. We then determine, for each $B_{i,i+1}$, the surviving portion of $B_{i,i+1}$ in $B_{1,i+1}$, as well as the edge l_i, connecting this surviving portion to the next vertex (to the left) in $B_{1,i+1}$. We then compare l_i and r_i to determine if any of $B_{i,i+1}$ survives in $B_{1,m}$. A similar computation determines the surviving portion of $T_{i,i+1}$ in $T_{1,m}$. The details are as follows:

1. We assign a processor to each pair (i, j) and use this processor to determine the two tangent lines defined by $B_{i,i+1}$ and $H(e_j, e_{j+1})$, where $j \geq i$. Let $xr_{i,j}$ denote the cross tangents from $B_{i,i+1}$ to $T_{j,j+1}$ and let $sr_{i,j}$ denote the supporting tangent from $B_{i,i+1}$ to $B_{j,j+1}$. If $H(e_i, e_{i+1})$ or $H(e_j, e_{j+1})$ is closed, then we use the "funnel" portion of the hourglass that is closer to the other hourglass for this tangent-finding procedure. Also, if $H(e_i, e_{i+1})$ or $H(e_j, e_{j+1})$ obscures e_{i+1} from e_j, then we let $xr_{i,j}$ and $sr_{i,j}$ be undefined.

2. Let us concentrate our attention on a specific hourglass chain, $B_{i,i+1}$. For each $j > i$ we compute the tangent line XR_j, which is the "rightmost" cross tangent $xr_{i,j'}$, touching B_i and some $T_{j'}$, for $i \leq j' \leq j$. By "rightmost" we mean the tangent touching B_i farthest to the right, and, if there are ties, the one among those with smallest slope. Intuitively, this is the most restrictive cross tangent that would be encountered in concatenating $H(e_i, e_{i+1})$ with any $H(e_{j'}, e_{j'+1})$, where $i \leq j' \leq j$. Also, by convention, we let an undefined tangent be the most restrictive tangent in this sense. Note that all the XR_j values for $H(e_i, e_{i+1})$ can easily be computed in $O(\log n)$ by a simple parallel prefix computation. By a similar computation, for each $j > i$, we determine SR_j, the leftmost supporting tangent $sr_{i,j'}$, touching B_i and some $B_{j'}$, for $i < j' \leq j$. If SR_j is to the right of XR_j for all $j > i$, then $H(e_i, e_m)$ is open to the right of $H(e_i, e_{i+1})$, for this implies that there is a vertex on $H(e_i, e_{i+1})$ that has a clear line-of-sight to e_m. Thus, we take r_i to be SR_m in this case. (See Figure 2(a).) Otherwise, let j be first index greater than i such that SR_j is to the left of XR_j. In this case we take r_i to be sr_j if $sr_j = SR_j$, or xr_j if $xr_j = XR_j$. (See Figure 2(b).)

3. Perform a computation similar to Step 2 to compute tangents XL_j and SL_j for each $B_{i,i+1}$ (analogous to XR_j and SR_j), and, more importantly, the tangent l_i, the edge of $B_{1,i+1}$ tangent to $B_{i,i+1}$.

4. If l_i and r_i cross, then there is no vertex of $B_{i,i+1}$ on $B_{1,m}$. If l_i and r_i do not cross, however, then $B_{1,m}$ contains the chain formed by l_i, followed by the subchain

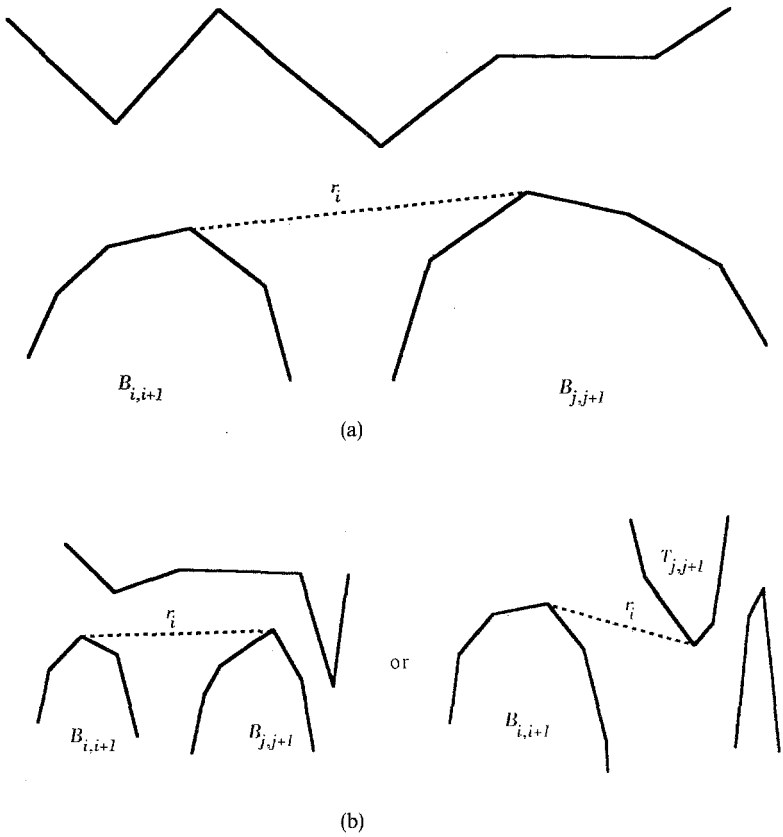


Fig. 2. The tangent edge to the right of $B_{i,i+1}$. In case (a) it is to a $B_{i,i+1}$ and $H(e_i, e_m)$ is open through $H(e_i, e_{i+1})$. In case (b) it can either be to a $B_{i,i+1}$ or to a $T_{i,i+1}$ and $H(e_i, e_m)$ is closed through $H(e_i, e_{i+1})$.

of $B_{i,i+1}$ from l_i's point of tangency to r_i's point of tangency, followed by r_i. (See Figure 3.)

5. We perform a computation analogous to Steps 1–4 to compute the surviving portion of $T_{i,i+1}$, if there is any, as well as the edges adjacent to such a surviving portion.

6. Given all the surviving portions of the hourglasses $H(e_1, e_2), \ldots, H(e_{m-1}, e_m)$. We can compute all the pieces of $T_{1,m}$ and $B_{1,m}$ by concatenating these chains via list-ranking[5] and parallel prefix computations. This completes the construction. Since each of the above steps can easily be performed in $O(\log n)$ time using $O(m^2)$ processors, this also completes the proof. □

[5] A list-ranking procedure determines the rank of each element in a linked list, and can be implemented in $O(\log n)$ time using $O(n/\log n)$ processors [5], [17].

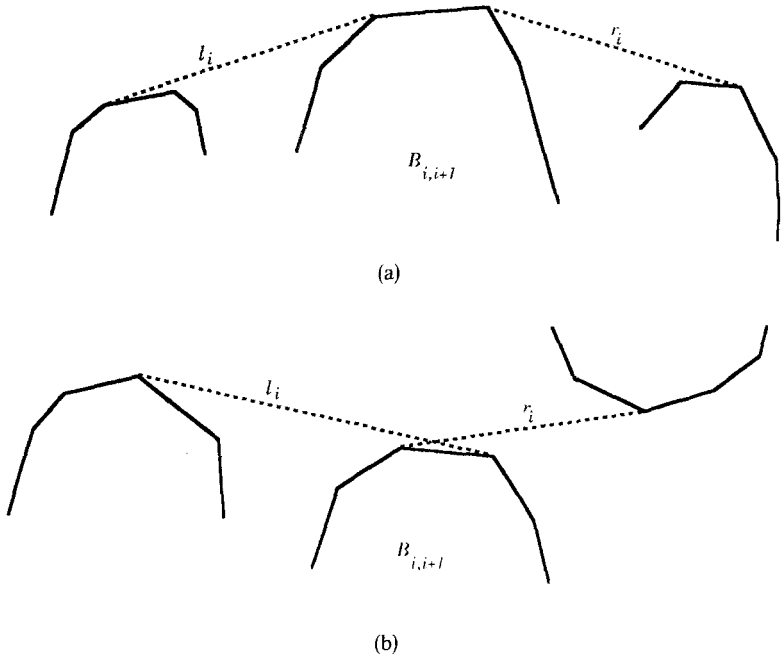


Fig. 3. The edges l_i and r_i do not cross in (a); they do cross in (b).

The hourglass structure provides the foundation upon which we build an efficient, but relatively slow, parallel method for preprocessing P, which we describe next. (We subsequently improve this approach using the statified decomposition tree.)

3.2. The Factor Graph. Suppose we are given a triangulated polygon P, the dual graph D of this triangulation, and a centroid decomposition tree T for P. We augment T as in [16] and [27] so that there is an edge (v, w) added to T for each pair of nodes v and w such that e_v is a pseudoedge on P_w (note that v must be an ancestor of w in this case). The edges of T together with these new edges defines the *factor graph* of P [27], which we denote $\hat{T}$ (see Figure 4).

The following lemma characterizes the size of $\hat{T}$.

LEMMA 3.4 [16]. *A node v in T at depth d_v has*

(a) *at most one edge in $\hat{T}$ going to a node at depth d, if d is smaller than d_v, and*
(b) *at most two edges in $\hat{T}$ going to nodes at depth d, if d is larger than d_v.*

Therefore, each node of $\hat{T}$ has degree at most $O(\log n)$, and all of $\hat{T}$ has size $O(n)$.

We can use the factor graph for shortest-path queries simply by constructing a representation of the hourglass $H(e_v, e_w)$ for each edge (v, w) in $\hat{T}$. Such a data structure can be built in $O(\log^2 n)$ time using $O(n)$ processors by a simple "bottom-up" procedure, which is essentially a parallelization of the high-level description

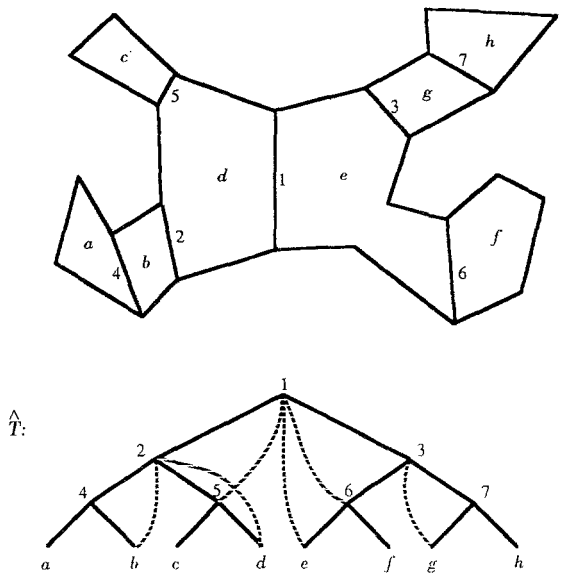


Fig. 4. The factor graph.

of a method given by Guibas and Hershberger [27]. In particular, we start with the triangles corresponding to the leaves in T, and construct a (trivial) hourglass for each edge (v, w) of $\hat{T}$ where w is a leaf (i.e., P_w is a triangle). Then we proceed up the tree, level by level, where we construct an hourglass representation for each edge (v, w) such that w is on the current level. Each such hourglass can be defined by concatenating two existing hourglasses; hence, be Lemma 3.2, each stage can be implemented in $O(\log n)$ time using $O(n)$ processors. Thus, the entire computation can be implemented in $O(\log^2 n)$ time and $O(n \log n)$ space using $O(n)$ processors. Given two query points p and q inside P, we can use this structure to build a (binary tree) representation of the shortest path between p and q inside P by concatenating at most $O(\log n)$ hourglasses stored in $\hat{T}$ (e.g., see [27]).

Guibas and Hershberger [27] show how to improve the sequential time and space complexity of this approach, but their methods do not seem to lead to a more time-efficient parallel algorithm. We can make a simple modification to improve the space and number or processors used by this method, however, without slowing down the time complexity for preprocessing or querying. Specifically, we can modify this computation by performing it on T', a d-contracted version of T, where $d = \lceil \log^2 n \rceil$. Before performing the construction we perform a preprocessing step, where we construct the factor graph, $\hat{T}'$ for T', and build a representation of the hourglass associated with each (v, w) in $\hat{T}'$, where e_v and e_w are both on a P_v such that v is a leaf; we call each such edge a *bottom* edge. By Lemma 3.4, $\hat{T}'$ has size $O(n/\log^2 n)$. Thus, we can assign a processor to each bottom edge (v, w) in $\hat{T}'$, and use Lemma 3.1 to construct a binary tree representation of the associated hourglass in $O(\log^2 n)$ time, since $n_v \leq 3\lceil \log^2 n \rceil$. We

then apply the bottom-up parallel method given in the previous paragraph. This results in a structure with $O(n)$ space (including the extra $O(\log n)$ for each edge in $\hat{T}'$), which can be constructed in $O(\log^2 n)$ time using only $O(n/\log^2 n)$ processors. It can still be used to answer shortest-path queries in $O(\log^2 n)$ time, however. If our query points p and q are in the same P_v, where v is a leaf, then we can simply apply Lemma 3.1 to construct a representation of the shortest path between p and q. Otherwise, if $p \in P_v$ and $q \in P_w$, with v and w being different leaves, then we can apply Lemma 3.1 to construct binary tree representations of the two hourglasses needed for the two leaves v and w in T' such that $p \in P_v$ and $q \in P_w$, and then use the method of [27] to concatenate these with another at most $O(\log n)$ hourglasses stored in $\hat{T}'$. Thus, we have the following:

LEMMA 3.5. *Given a triangulated n-node polygon P, we can construct an $O(n)$-space data structure that allows us to construct a binary tree representation of the shortest path between two points inside P in $O(\log^2 n)$ time. Moreover, this construction can be done in $O(\log^2 n)$ time and $O(n)$ space using $O(n/\log^2 n)$ processors in the CREW PRAM model.*

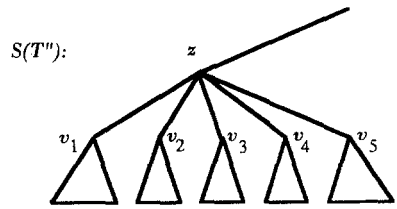
Just as Lemma 3.1 provided the foundation for this result, Lemma 3.5 provides the foundation for an even faster parallel method, which we describe next.

3.3. The Stratified Decomposition Tree. In this subsection we describe an $O(\log n)$-time method for constructing a data structure that can be used to answer shortest-path queries in $O(\log n)$ time. Our method is based on the use of a structure we call the *stratified decomposition tree.* Suppose we have a centroid decomposition tree T for P, and are also given an integer parameter, r. For reasons that will soon become apparent, we define the stratified decomposition tree on T'', a d-contraction of T, where $d = r\lceil \log^4 n \rceil$. We use a recursive definition to mark certain nodes in T'' as *boundary* nodes. The basis boundary node in T'' is the root. Given a recursively defined boundary node v, we define each w in the subtree of T'' rooted at v to be a boundary node if either of the following is satisfied:

- w is at distance $\lceil (\log(n_v/r))/4 \rceil$ from v.
- w is at distance less than $\lceil (\log(n_v/r))/4 \rceil$ from v, but w is a leaf.

All such boundary nodes w define the *boundary children* of v. Note that any v-to-leaf path in T'' must contain exactly one boundary child of v, and any boundary node v in T'' can have at most $O((n_v/r)^{1/4})$ boundary children. We define the boundary children of each nonleaf boundary child of v recursively. By repeating this definition until we include all the leaves of T'' as boundary nodes we define the *stratified decomposition tree* for P, which we denote $S(T'')$. The name of this structure is motivated by the "layering" of T'' that is imposed by the boundary nodes. (See Figure 5(a).) Given the tree T'', it is straightforward to determine all the boundary nodes in T'' by a top-down procedure that runs in $O(\log n)$ time using $O(n/\log n)$ processors.

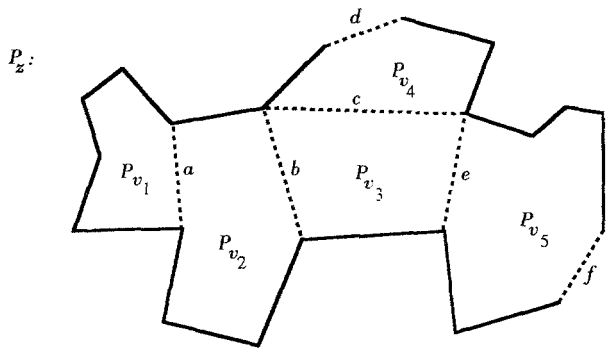
The tree $S(T'')$ provides the "skeleton" for our data structure, which is defined



(a)

Pseudo(z) = (d,f)

Pairs(z) = ((a,b),(a,c),(a,d),(a,e),(a,f),(b,a),(b,c),...)



(b)

Fig. 5. (a) The stratified decomposition tree $S(T'')$, and (b) its relationship to P.

so that for each boundary node in z in $S(T'')$ we have the following:

- A list of all the boundary children of z.
- A list, *Pairs*(z), of all pairs (e,f), where e is a pseudoedge on some P_v, where v is a child boundary node of z, and f is a pseudoedge on P_z or P_w, where w is another child boundary node of z.
- A representation of $H(e,f)$ for each pair (e,f) in *Pairs*(z).
- A list, *Pseudo*(z), of the pseudoedges on P_z, listed in counterclockwise order around P_z. With each edge e in P_z we store the two values of *sum*(x), the prefix summation, associated with the two places in the Euler-tour of D where the dual of e appears.
- A factor-graph data structure for each leaf of $S(T'')$.

Before we describe our method for constructing this structure, we make an important observation concerning its size. Clearly, the most space-consuming portion of this structure is that we store a representation of the hourglass $H(e,f)$ for each (e,f) in *Pairs*(z) for each boundary node z in $S(T'')$. (See Figure 5(b).) Note that for any such v we may need to store up to $O((n_z/r)^{1/4} + \log n)$ such hourglass representations, by Lemma 2.2; hence, we may need to store up to $O((n_z/r)^{1/2} +$

$(n_z/r)^{1/4} \log n + \log^2 n)$ such hourglass representations for all the children of z in $S(T'')$. Nevertheless, since T'' is an $r\lceil \log^4 n \rceil$-contraction of T, $\lceil \log^4 n \rceil \leq n_v/r$ for each v in T''. Thus, we can characterize the number of hourglass representations we need to store for the children of z as being $O((n_z/r)^{1/2})$ (this is the main reason for our choosing T'' as an $r\lceil \log^4 n \rceil$-contraction of T).

We begin our construction with a preprocessing step, where we apply Lemma 3.5 to construct a factor-graph-based shortest-path data structure on P_v for each leaf v in T''. This takes $O(\log^2 n_v) = O((\log r + \log\log n)^2)$ time and $O(n)$ space using $O(n/(\log r + \log\log n)^2)$ processors. We then apply the query result from Lemma 3.5 to construct a representation of each hourglass $H(e,f)$ such that e and f are pseudoedges on the boundary of P_v (i.e., (e,f) is in $Pairs(z)$) for each leaf v of T''. Since the number of such pairs (e,f) is $O(\log^2 n)$ for each v (by Lemma 2.2), this can be done in $O(\log^2 n_v) = O((\log r + \log\log n)^2)$ time using $O(n/(r \log^2 n))$ processors.

Our method for constructing the other needed hourglass representations is a recursive procedure, which we call by passing it the name of a boundary node z (we call it initially with z being the root). The problem is to construct a representation of $H(e,f)$ for each pair (e,f) in $Pairs(z)$. As observed above, there are at most $O((n_z/r)^{1/2})$ such pairs for z. In order to do this efficiently, we need a representation of each hourglass $H(e',f')$ such that e' and f' are pseudoedges on P_v for each child v of z. For each child v that is a leaf in T'', we already have these representations, by our preprocessing step. For each child v of z that is not a leaf, we recursively construct an hourglass representation for each pair of edges (e,f) in $Pairs(v)$. Having returned from this parallel recursive call, we assign $O((n_z/r)^{1/2})$ processors to each pair (e,f) in $Pairs(z)$—we use these processors to construct $H(e,f)$. Note that, since there are at most $O((n_z/r)^{1/2})$ pairs in $Pairs(z)$, this assignment requires only $O(n_z/r)$ processors. The group of processors assigned to the pair (e,f) determine the at most $O((n_z/r)^{1/4})$ hourglasses $H_1, H_2, \ldots, H_l$ that belong to children boundary nodes of z and must be concatenated to produce a representation of $H(e,f)$. Note that the H_i's are already available (either by our preprocessing step or by the recursive call). The processors assigned to (e,f) can easily determine the ordering of these hourglasses along the path in D in $O(\log n_z)$ time (by a simple list-ranking procedure). Using the method of Lemma 3.3, these processors can then construct a representation of $H(e,f)$ in $O(\log n_z)$ time, since we have $O((n_z/r)^{1/2})$ processors available to concatenate $O((n_z/r)^{1/4})$ hourglasses. This completes the construction, as the other portions of our structure are easily constructed in $O(\log n)$ time using $O(n/\log n)$ processors, given T'' and $S(T'')$.

Before we explain how to answer a shortest-path query with our stratified decomposition tree data structure, let us analyze the complexity of this construction algorithm. By the polygon-cutting theorem [12], the subtree rooted at a child boundary node of z has at most $(2/3)^l n_z$ nodes, where $l = (\log(n_z/r))/4$. This implies that each such subtree has at most $(rn_z)^{5/4 - 1/\log_3 16}$ nodes (which is less than $(rn_z)^{0.854}$). Thus, the time complexity, $t(n_z)$, of the recursive part of this construction is characterized by the following recurrence relation:

$$(*) \qquad t(n_z) \leq t((rn_z)^{0.854}) + b \log n_z$$

for some constant b. This relation implies that $t(n_z)$ is $O(\log n_z + \log r \log\log n_z)$. The number of processors, $p(n_z)$, needed for this construction satisfies the recurrence

$$p(n_z) = \max\left\{ \sum_{\text{children } v \text{ of } z \text{ in } T''} p(n_v), \frac{cn_z}{r} \right\}$$

for some constant c. Since $n_v \le n_z^{0.854}$ for each child v of z, and $n_v \ge r\lceil \log^4 n \rceil$ for each leaf v, this implies that $p(n_z)$ is $O(n_z/r)$. Finally, the space complexity satisfies the recurrence

$$s(n_z) = \sum_{\text{children } v \text{ of } z} s(n_v) + d\left(\frac{n_z}{r}\right)^{3/4} \log n_z$$

for some constant d. Since $\log n_z \le (n_z/r)^{1/4}$, $n_v \ge r\lceil \log^4 n \rceil$ for each leaf v, $n_v \le n_z^{0.854}$ for each child v or z in T'', this implies that the space needed is $O(n_z \log n_z/r)$. We have already observed that the preprocessing step can be implemented in $O((\log r + \log\log n_z)^2)$ time and $O(n)$ space using $O(n)$ work (i.e., $O(n/(\log r + \log\log n_z)^2)$ processors). Taking $r = \lceil \log n \rceil$, and combining the above discussion with Lemma 2.3, gives us the following lemma.

LEMMA 3.6. *Given an n-node triangulated simply polygon P, we can construct $S(T'')$ and all its associated hourglass representations, as above, in $O(\log n)$ time and $O(n)$ space using $O(n/\log n)$ processors in the CREW PRAM model.*

3.4. Answering Shortest-Path Queries. Suppose we are given two points p and q inside P. We can use $S(T'')$ to construct a representation of the shortest path between p and q inside P. We first locate the polygons P_v and P_w, where v and w are leaves in $S(T'')$ with $p \in P_v$ and $q \in P_w$. This can be done in $O(\log n)$ time, assuming we have preprocessed P for planar point location [19], [45], which requires $O(n/\log n)$ processors given a triangulation of P. If $v = w$, then we apply Lemma 3.5 to construct the shortest path, and we are done. If $v \ne w$, then we determine the paths, π_v and π_w, from v and w, respectively, to their least common ancestor, z in $S(T'')$ (by "marching" up $S(T'')$ from v and w). We also determine the pseudoedge e (resp. h) that is on the boundary of P_v (resp. P_w) and is intersected by the shortest path from p to q. We do this by a simple binary search through the list *Pseudo*(v) (resp. *Pseudo*(w)). Specifically, let *sum*(v) (resp. *sum*(w)) denote the prefix summation value for the dual of some pseudoedge on P_v (resp. P_w). We can locate the pseudoedge e in P_v by searching for the position of *sum*(w) among the *sum* values for the pseudoedges in *Pseudo*(v), and a similar search locates the pseudoedge h in P_w.

We then apply Lemma 3.5 to construct a representation of the hourglass $H(p, e)$ and $H(h, q)$ in $O((\log\log n)^2)$ time. By two more binary searches we can locate the pseudoedges f and g on P_z intersected by the shortest path from p to q. Since (f, g) is, by definition, an edge in *Pairs*(z), we already have a representation of $H(f, g)$.

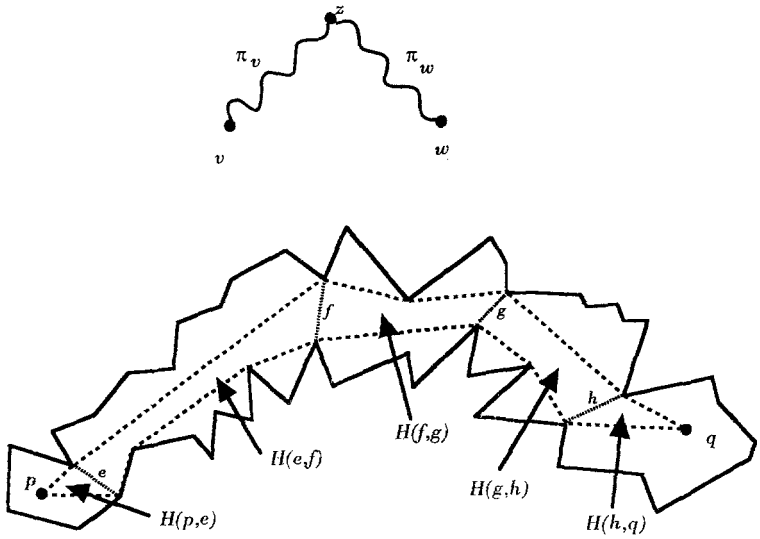


Fig. 6. The hourglasses needed to construct a representation of $H(p, q)$.

To implement the remainder of our search, then, we must construct a representation of $H(e,f)$ and $H(g, h)$, and then concatenate the representations $H(p, e)$, $H(e,f)$, $H(f, g)$, $H(g, h)$, and $H(h, q)$. (See Figure 6.) Let us concentrate on the construction of $H(e,f)$, as our method for constructing $H(g, h)$ is similar. Let $e = e_1, e_2, \ldots, e_a = f$ be the pseudoedges that are on a P_u such that u is on π_ω and are intersected by the shortest path from p to q, listed in order as they are intersected by this path. Our method for constructing $H(e,f)$ is a very simple recursive procedure: to construct $H(e_1, e_i)$, we construct $H(e_1, e_{i-1})$ recursively and concatenate this to $H(e_{i-1}, e_i)$ (which is already available in $S(T'')$), using the method of Lemma 3.2. Given the edge e_i, we can determine the edge e_i by a simple binary search in the list *Pseudo*(u), where P_u is the "current" polygon (associated with node u on π_v). Since each e_i is a pseudoedge belonging to a polygon P_u for u on π_ω, the time complexity of this simple procedure is the same as (*). Therefore, the construction of $H(e,f)$ (hence, $H(g, h)$) can be implemented in $O(\log n)$ time. By Lemma 3.2 this immediately implies we can compute a representation of $H(p, q)$ in $O(\log n)$ time. Combining this with Lemma 3.6, we have the following theorem:

THEOREM 3.7. *Given a triangulated simple n-node polygon P, we can construct a data structure in $O(\log n)$ time using $O(n/\log n)$ processors, in the CREW PRAM model, that allows for a single processor to build a binary tree representation of the shortest path between two query points in $O(\log n)$ time.*

Note that our construction does more than simply determine the length of the shortest path between two query points p and q: it constructs a representation of the hourglass between the triangles P_a and P_b containing p and q, respectively. This representation can also be used to determine, for any i, the ith edge of the

shortest path between p and q in $O(\log n)$ time, or can be used to enumerate all the edges of the shortest path in $O(\log n)$ time using $O(\lceil k/\log n \rceil)$ processors, where k is the number of edges in this path. In the next section we show how to apply this theorem to solve an important visibility problem.

4. Visibility from an Edge. In this section we show how to use the result of the previous section to compute the portion of P that is visible from a distinguished edge e. Moreover, we show that it can in fact be used to compute a data structure similar to that of Chazelle and Guibas [16] for determining the first point on the boundary of P hit by a ray of emanating from a point on e. Our method runs in $O(\log n)$ time using $O(n)$ processors, and, for the query problem, allows ray-shooting from e to be answered in $O(\log n)$ time by a single processor.

Before we describe our method, let us review some properties observed by Chazelle and Guibas [16] for the geometry of hourglasses in the context of visibility queries. We note that if we are only interested in visibility queries a typical hourglass contains more information than we need. For example, if an hourglass $H(e,f)$ is closed, we do not need any of the edges of $H(e,f)$ to answer a visibility query for any pair of points lying on e and f, respectively, since no visibility is possible between these two diagonals. Even if an hourglass $H(e,f)$ is open, there may be some edges that are not needed. In particular, the only edges of an open $H(e,f)$ that will ever have any influence on a line-of-sight are those edges that lie between the cross tangents between the two convex chains of $H(e,f)$.

Given two diagonals e and f on P, we define the *visibility polygon* of between e and f in the dual plane, as follows: If $H(e,f)$ is closed, then $V(e,f)$ is a *null* polygon. If $H(e,f)$ is open, then we define $V(e,f)$ as the dual of the polygon formed by removing the edges of $H(e,f)$ that do not lie between the tangent points for the cross tangents between the two chains for $H(e,f)$, and then extending the cross tangents to infinity. The polygon $V(e,f)$ can be viewed as a convex polygon in the two-sided plane, for we can place the convex chain encountered in a counter-clockwise traversal from e to f on the top half of a two-sided plane and the other chain on the bottom. Note that a point p is inside $V(e,f)$ if and only if p corresponds to a line in the primal plane that intersects e and f, but does not cross any of the edges of $H(e,f)$. Also note that if we take a simple polygon P, fix an edge e on P, and construct all the visibility polygons $V(e,f)$ such that f is another edge on P, then we construct a subdivision in the two-sided plane. As observed by Chazelle and Guibas [16], this subdivision constitutes a solution to the visibility from an edge problem, for each face $V(e,f)$ corresponds to the set of lines that intersect e and f while missing the boundary of P. Moreover, this subdivision, which we denote $Vis(e, P)$, has $O(n)$ size [16]. (See Figure 7.)

We show that $Vis(e, P)$ for any fixed edge e on P can be constructed in $O(\log n)$ time using $O(n)$ processors. We describe a procedure with essentially the same recursive structure as that used to construct the stratified decomposition tree. In our algorithm description whenever we refer to a node v in the stratified decomposition tree $S(T'')$, we use f_v to denote the pseudoedge on P_v that is closest

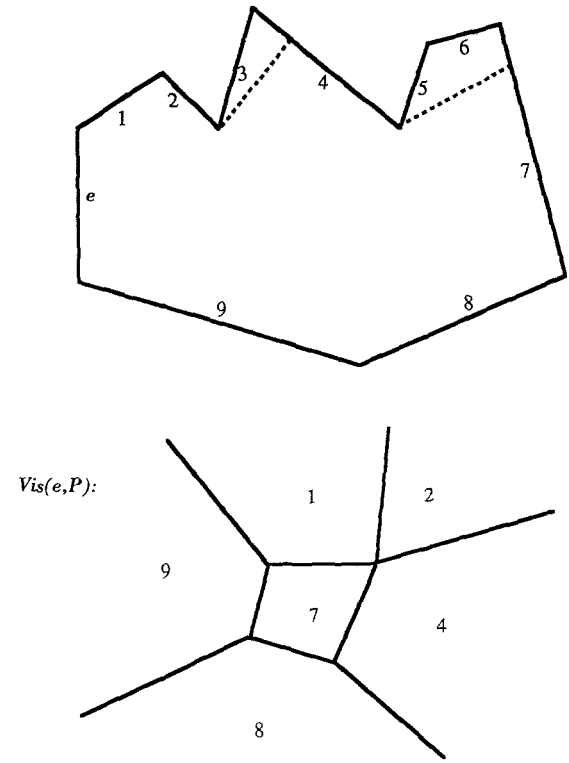


Fig. 7. The subdivision *Vis*(*e*, *P*).

to the distinguished edge e (which can be determined by the relative position of their dual edges in D).

In order to compute $Vis(e, P)$ we recursively compute $Vis(f_z, P_z)$ for each node z in $S(T'')$ (where we view e to be the root of the polygon-cutting theorem decomposition). As a preprocessing step, for each leaf z in T'', we use a straightforward parallelization of the divide-and-conquer method of Chazelle and Guibas [16] to construct $Vis(f_z, P_z)$ in $O(\log^2 n_z)$ time using $O(n_z)$ processors. Given a nonleaf boundary node z, we recursively compute $Vis(f_v, P_v)$ for each child boundary node v of z. Using the method of the previous section we then compute a representation of $H(f_z, f_v)$ for each child boundary node v of z, which can be done in $O(\log n_z)$ time using $O((n_z/r)^{1/2})$ processors, where, in this case, we take $r = 1$ (since we need $O(n)$ processors anyway). Intuitively, the hourglass $H(f_z, f_v)$ is a "tunnel" through which we must pass any line-of-sight from f_z to f_v. (See Figure 8.) Thus, our next step is to construct a representation of the visibility polygon $V(f_z, f_v)$ defined by $H(f_z, f_v)$. This is easily accomplished in $O(\log n_z)$ time using $O((n_z/r)^{1/2})$ processors, by determining the two cross-tangents if $H(f_z, f_v)$ is open (if $H(f_z, f_v)$ is closed, the computation is trivial). For each segment of s in the subdivision $Vis(f_v, P_v)$, we then assign a single processor to the task of computing the intersection of s with the interior of $V(f_v, f_z)$. This can also be done in $O(\log n_z)$

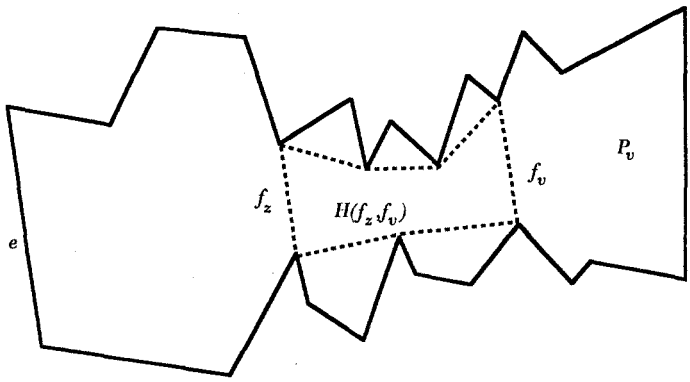


Fig. 8. The visibility through $H(f_z, f_v)$.

time, by a binary search type of computation, using a total of $O(n_v)$ processors for each child boundary node v of z. The surviving portion of each segment s (if there is any) defines a segment in $Vis(f_z, P_z)$, by an argument similar to that given by Chazelle and Guibas [16].

THEOREM 4.1. *Given a polygon P and a distinguished edge e on P, we can compute the portion of P visible from e in $O(\log n)$ time using $O(n)$ processors in the CREW PRAM model. Moreover, in these same bounds we can construct a data structure that allows a single processor to determine the edge of P hit by any query ray emanating from e in $O(\log n)$ time.*

PROOF. We have already discussed how to construct $Vis(e, P)$, given a triangulation of P. By Lemma 2.1, we can triangulate P in $O(\log n)$ time using $O(n)$ processors, as needed. For the query problem, we note that we can use any efficient parallel planar-point location data structure that runs in $O(\log n)$ time using at most $O(n)$ processors and results in an $O(\log n)$ query time (e.g., [7], [19], and [45]) to locate the point that is dual to the query ray in $Vis(e, P)$. □

5. The Visibility Graph. The problem we address in this section is the computation of the visibility graph G, of P, which is defined on the vertices of P such that (v, w) is an edge in G if and only if v is visible from w (see [30] for references). Our method for constructing G is based on Theorem 4.1 and two important lemmas, one geometric and one algorithmic. We begin by describing our algorithmic lemma.

5.1. Two-Set Intersection Reporting. The algorithm we describe in this subsection is for a seemingly unrelated problem of computing the intersections between two collections of nonintersecting line segments. Our method improves a similar lemma of Rüb [40], and may be of independent interest.

LEMMA 5.1. *Suppose we are given two sets of line segments, A and B, such that no two segments in A (resp. B) intersect, except possibly at endpoints. The we can compute all the pairwise intersections between segments in A and B in $O(\log n)$ time using $O(n + k/\log n)$ processors in the CREW PRAM model, where $n = |A| + |B|$ and k is the number of answers.*

PROOF. The method is based on the approach of Rüb, but improves the number of processors needed for reporting the answers by a logarithmic factor. The main idea is to build a (single) segment tree T on the x-coordinates defined by the segments in A and B as in [7]. The leaves of T correspond to slabs defined by placing vertical lines between consecutive endpoint x-coordinates. Internal nodes in T correspond to the union of the slabs associated with their descendants. For each v in T, we construct a list $C_A(v)$, where $C_A(v)$ is the list of all segments in A that span the slab, Π_v, for v, but do not span the slab, Π_z, for v's parent z. These lists are sorted by the "above" relationship. We define $C_B(v)$ lists similarly for the segments in B. We then take a d-contraction of T, and let T' denote this new tree, where $d = \lceil \log n \rceil$. Using the method of Atallah *et al.* [7], we can construct all the C_A and C_B lists for T' in $O(\log n)$ time and $O(n \log n)$ space using $O(n)$ processors. By a lemma similar to one given by Chazelle [13], it is easy to show that if two segments $s \in A$ and $t \in B$ intersect, then there must be a node v such that

(1) $s \in C_A(v)$ and $t \in C_B(v)$,
(2) $s \in C_A(v)$ and t has an endpoint in Π_v,
(3) $t \in C_B(v)$ and s has an endpoint in Π_v, or
(4) v is a leaf and both s and t have an endpoint in Π_v.

(See Figure 9.) We can determine all intersections of type 1 by merging the sorted lists $C_A(v)$ and $C_B(v)$ twice: the first time with comparisons based on segment intersections with the left vertical boundary of Π_v (call this the "leftist" rule) and the second time with comparisons based on segment intersections with the right

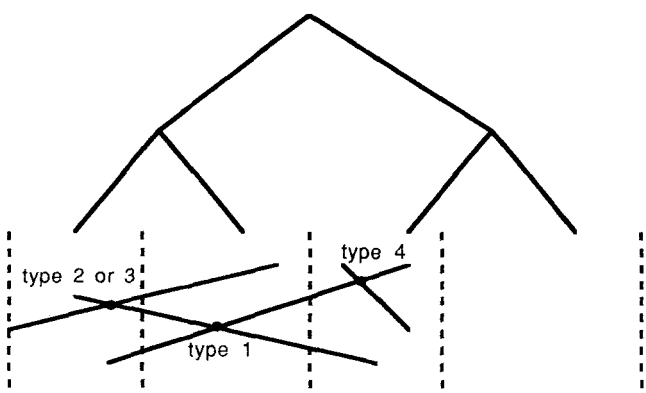


Fig. 9. The characterization of intersections.

vertical boundary of Π_v (call this the "rightist" rule). For a segment s in $C_A(v)$ the segments in $C_B(v$ between s's rank in $C_B(v)$ based on the leftist rule verses s's rank based on the rightist rule are exactly the segments in $C_B(v)$ that intersect s. This computation can be performed in $O(\log n)$ time using $O((|C_A(v)| + |C_B(v)|)/\log n)$ processors, by the merging of [11] and [41], for each v, or $O(n)$ processors overall. Type 4 intersections are even easier to detect, for we can assign a processor to each segment s and use this processor to compare s with all the other segments that have an endpoint in the same leaf slab(s) as s, since there are $O(\log n)$ such segments.

For type 2 and type 3 intersections, we must use a more involved procedure than this, however. Since the segments in A (resp. B) do not intersect, we can apply the fractional cascading technique[6] of Chazelle and Guibas [15], as implemented in parallel by Atallah *et al.* [7]. This allows a single processor to locate a single point p in the $C_A(v)$ list (resp. $C_B(v)$ list) for each v such that p is in Π_v in $O(\log n)$ time, for these v's define a leaf-to-root path and each look-up of p in a list is based on the "above-below" relation. Thus, by assigning a processor to each segment s in A we can locate the endpoints p and q of s in each $C_B(v)$ such that p and q are in Π_v in $O(\log n)$ time. We can then "read off" all the intersections of s with the segments in $C_B(v)$—they are exactly the segments between the positions of p and q in the list $C_B(v)$. If, on the other hand, only one of the endpoints for s (say, p) is in the slab Π_v, then we must locate, in the list $C_B(v)$, the point q' where s intersects one of the vertical boundaries for Π_v. If the other endpoint, q, on s is in Π_w, where w is v's sibling, then we can afford to perform a binary search to locate the position of q' in $C_B(v)$, for this case can occur in at most two nodes in the tree. We cannot afford to perform a binary search for q', however, if q is not in Π_w. Fortunately, if q is not in Π_w, then s is in $C_A(w)$ by definition. Therefore, we can find the position of q' in $C_B(v)$ by merging $C_B(v)$ with $C_A(w)$, basing comparisons on the y-coordinates of segment intersections with the vertical line separating Π_v and Π_w. For example, if w is the right child of v's parent, then we use the rightist ordering for $C_B(v)$ and the leftist ordering for $C_A(w)$. (See Figure 10.) This provides the position of q' in $C_B(v)$ for all segments s with $p \in \Pi_v$ but $q \notin \Pi_w$, where w is v's sibling, and can be implemented in $O(\log n)$ time using $O(n)$ processors [11], [41]. We can then "read off" all the intersections of s with the segments in $C_B(v)$—they are exactly the segments between the positions of p and q' in $C_B(v)$. We can perform a similar computation for each segment s from B.

Note that this method gives us, in $O(\log n)$ time for each segment s, an implicit representation of all the intersections s has with segments in the other set. Given this information for all segments, we may, after a parallel prefix computation,[7]

---

[6] Recall that in this technique we build a data structure upon a graph that has lists stored at its nodes. This structure allows a single processor to locate a key x in each list on a path π in $O(\log n + |\pi| \log \Delta)$ time, where Δ is the degree of the graph. This construction can be done sequentially in $O(n)$ time [15] and in parallel in $O(\log n)$ time using $O(n/\log n)$ processors [7].

[7] In a parallel prefix computation we reduce the problem at hand to that of computing all prefix sums $s_k = \sum_{i=1}^{k} a_i$ for n values $a_1, a_2, \ldots, a_n$, which all can be found in $O(\log n)$ time using $O(n/\log n)$ processors [32], [33].

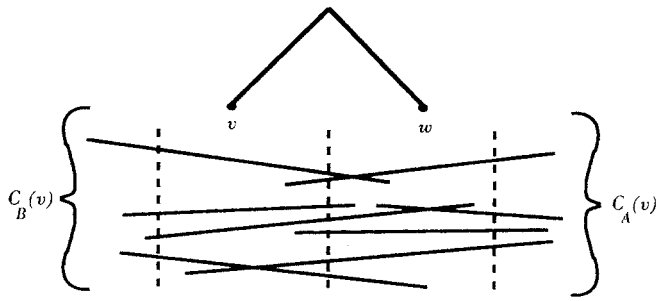


Fig. 10. The relationship between $C_B(v)$ and $C_A(w)$.

globally allocate enough processors to report all of these intersections optimally in $O(\log n)$ time, i.e., $O(k/\log n + 1)$ processors. This established the lemma. □

We use the lemma to help solve the seemingly unrelated problem of constructing the visibility graph, using a point-line duality to drive the reduction.

We have already given our algorithmic lemma in Lemma 5.1, thus we have only to give our geometric lemma.

5.2. A Dual Characterization. As observed by Edelsbrunner *et al.* [21], the set of lines that intersect a line segment (p, q) corresponds to the set of points in the dual plane between the dual lines $\mathscr{D}_p$ and $\mathscr{D}_q$. This structure is called the *double wedge* for the segment (p, q). As we observe in the following lemma, this structure can also be used to characterize visibility-graph edges (see Figure 11).

LEMMA 5.2. *Let P_v and P_w be two disjoint subpolygons of P, separated by a common diagonal e. There is a visibility-graph edge from a vertex p in P_v to a vertex q in P_w if and only if all of the following conditions hold:*

1. *p is visible from e; hence, there is a segment s_p corresponding to p in $Vis(e, P_v)$.*
2. *q is visible from e; hence, there is a segment s_q corresponding to q in $Vis(e, P_w)$.*
3. *s_p intersects s_q and this intersection point is inside the double-wedge defined by e.*

PROOF. Suppose each of the above conditions hold. We must show that p is visible from q. Each point on s_p is the dual of a line that passes through p. Moreover, by the definition of $Vis(e, P_v)$, each point on s_p is the dual of a line that intersects e (and no other points on the boundary of P between p and e). A similar property holds for q, s_q, and e. Also, the intersection of s_p and s_q corresponds to the line ℓ (in the primal plane) determined by p and q. Since the dual of ℓ lies inside the double-wedge for e (a shared edge of P_v and P_w), ℓ cannot intersect any points of P between p and q. The arguments for the "only if" part of the proof follow by a similar argument (in reverse order, of course). □

This establishes our geometric lemma.

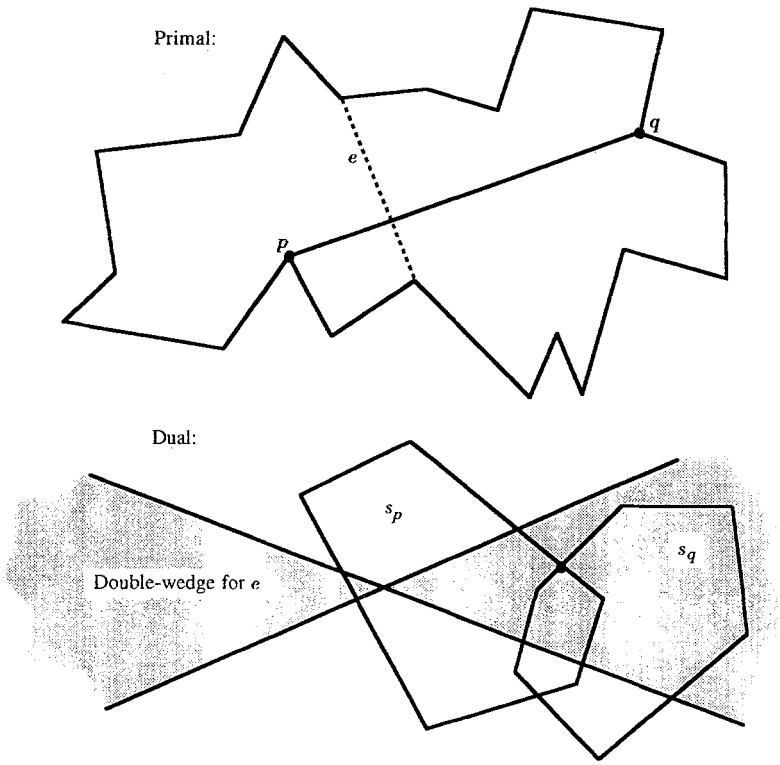


Fig. 11. Visibility subdivisions and the double-wedge for e.

5.3. Computing the Edges of G. Having presented our two important lemmas, we are now ready to give our method for constructing G. Our method is to apply Theorem 4.1 to construct the subdivisions $Vis(e_v, P_u)$ and $Vis(e_v, P_w)$ for each node v in the centroid decomposition tree T for P, where u and w are the children of v. Moreover, we perform this computation for each v in parallel. This can be done for all v in parallel in $O(\log n)$ time using $O(n \log n)$ processors by Theorem 4.1, since the total size of all the subdivisions can be $O(n \log n)$. We then use the method of Lemma 5.1 to compute for each node v in T the intersection of the segments in $Vis(e_v, P_u)$ with the segments in $Vis(e_v, P_w)$. This requires $O(\log n)$ time using $O(n \log n + k/\log n)$ processors, where k is the number of answers. By Lemma 5.2, each intersection point corresponds to an edge in the visibility graph. Thus, we have the following:

THEOREM 5.3. *Given an n-node polygon P, we can construct the visibility graph for P in $O(\log n)$ time using $O(n \log n + k)$ processors in the CREW PRAM model, where k is the number of edges.*

6. Applications. There are also a number of other problems that can be solved efficiently in parallel using the stratified decomposition tree approach. We mention

a few of them here, giving a brief description of how each of them can be solved in parallel.

6.1. Shortest-Path Tree from a Vertex. An important problem dealing with shortest paths in a simple polygon is that of computing the shortest-path tree from a vertex v of P, i.e., the tree that is defined by the union of all the shortest paths from v to every other vertex of P. Using the shortest-paths data structure of Section 3, it is straightforward to solve this problem in $O(\log n)$ time using $O(n)$ processors. The method is to assign a processor to each vertex w of P and have that processor compute the first edge on the shortest path from w to v.

6.2. Convex Ropes. Another important problem in this domain is that of computing the convex ropes of a simple polygon [28]. This problem is defined as follows. Let v be a vertex on the boundary of the convex hull of P, and let w be any other vertex of P. The *clockwise convex rope from v to w* is defined to be the shortest path from v to w that is clockwise convex (in that it always makes "right turns") and never enters P, if such a path exists. The counterclockwise convex rope from v to w is defined similarly. The problem is to compute all the nodes that have both clockwise and counterclockwise ropes from a vertex on the convex hull of P. Intuitively, if we imagine the convex hull of P is being a rubber band, then for each such point p we can push the rubber band to p without having the band becoming "pinched." In other words, p is visible from some point outside the convex hull of P.

We can solve this problem in $O(\log n)$ time using $O(n)$ processors as follows. We first construct the convex hull H of P. This can be done in $O(\log n)$ time using $O(n/\log n)$ processors [49] (or with $O(n)$ processors by the more simple method of [3] and [8]). For each edge e of H that is not an edge of P, we can then determine the "bay" between the exterior of P and e, which is itself a simply polygon. Let B be the bay determined by such an edge e. We construct the shortest-path trees in B from both of e's endpoints, v and w. One of these trees can be viewed as the clockwise tree (say, the one from v) and the other can be viewed as the counterclockwise tree (say, the one from w). We remove any edge from the clockwise tree from v if it forms a left turn with the previous edge, and remove any edge from the counterclockwise tree from w if it forms a right turn with the previous edge. The vertices that remain in both trees are the vertices that are visible from a point outside the convex hull of P. Note that all the above operations can be performed in $O(\log n)$ time using $O(n)$ processors.

6.3. All-Farthest Neighbors. Suppose we are given a simply polygon P. Another interesting distance problem is to compute the farthest neighbor for each vertex of P, where distance is measured by the shortest path inside P. This is known as the all-farthest neighbors problem for a simple polygon [2], [27]. Guibas and Hershberger [27], exploiting the fact that the matrix of distances defined by the vertices of P satisfies an important monotonicity property [4], give an $O(n \log n)$-time algorithm to this problem sequentially. Atallah and Kosaraju [9] show how to find the all row-maxima for such arrays in $O(f(n) \log n)$ time using $O(n)$

processors, where $f(n)$ is the time needed to determine the value of an (i, j) entry in this matrix. Therefore, since we show how to preprocess P to allow for $f(n)$ to be $O(\log n)$, this immediately implies that we can solve the all-farthest neighbors problem for a simple polygon in $O(\log^2 n)$ time using $O(n)$ processors.

7. Open Problems. We have shown how to solve a number of shortest-path and visibility problems for simple polygons efficiently in parallel. Indeed, our method for preprocessing a simple polygon for shortest-path queries is optimal if P is triangulated. Nevertheless, there are a number of interesting open problems that follow from this work:

1. Can the visibility from an edge in a simple polygon be computed in $O(\log n)$ time with only $O(n/\log n)$ processors if P is triangulated?
2. Can the visibility graph for a simple polygon be constructed in $O(\log n)$ time using only $O(n + k/\log n)$ processors, where k is the size of the output? What about the visibility graph of an arbitrary set of segment in the plane?
3. Can a simple polygon be triangulated in $O(\log n)$ time using only $O(n/\log n)$ processors?
4. What is the fastest that the all-farthest neighbors problem for a simple polygon can be solved using only $O(n)$ processors?

Acknowledgment. We would like to thank S. Rao Kosaraju for several helpful conversations concerning the topics discussed in this paper.

References

[1] K. Abrahamson, N. Dadoun, D. A. Kirpatrick, and T. Przytycka, A Simple Parallel Tree Contraction Algorithm, Technical Report 87-30, Dept. of Computer Science, University of British Columbia, 1987.

[2] P. K. Agarwal, A. Aggarwal, B. Aronov, S. Rao Kosaraju, B. Scheiber, and S. Suri, Computing External-Farthest Neighbors in a Simple Polygon, *Discrete Appl. Math.*, **31**(2), 1991, 97–111.

[3] A. Aggarwal, B. Chazelle, L. Guibas, C. Ó'Dúnlaing, and C. Yap, Parallel Computational Geometry, *Algorithmica*, **3**(3), 1988, 293–328.

[4] A. Aggarwal and J. K. Park, Parallel Searching in Multidimensional Monotone Arrays, manuscript, 1989.

[5] R. J. Anderson and G. L. Miller, Deterministic Parallel List Ranking, in *Third Aegan Workshop on Computing* (AWOC 88) (J. H. Ref, ed.), Lecture Notes in Computer Science, Vol. 319, Springer-Verlag, Berlin, 1988, pp. 81–90.

[6] M. J. Atallah and D. Z. Chen, Optimal Parallel Algorithm for Visibility of a Simple Polygon from a Point, *Proc. 5th ACM Symp. on Computational Geometry*, 1989, pp. 114–123.

[7] M. J. Atallah, R. Cole, and M. T. Goodrich, Cascading Divide-and-Conquer: A Technique for Designing Parallel Algorithms, *SIAM J. Comput.*, **18**(3), 1989, 499–532.

[8] M. J. Atallah and M. T. Goodrich, Efficient Parallel Solutions to Some Geometric Problems, *J. Parallel Distrib. Comput.*, **3**(4), 1986, 492–507.

[9] M. J. Atallah and S. R. Kosaraju, An Efficient Parallel Algorithm for the Row Minima of a Totally Monotone Matrix, *Proc. 2nd SIAM–ACM Symp. on Discrete Algorithms*, 1991, pp. 394–403.

[10] D. Avis and G. T. Toussaint, An Optimal Algorithm for Determining the Visibility of a Polygon from an Edge, *IEEE Trans. Comput.*, **30**, 1981, 910–914.
[11] G. Bilardi and A. Nicolau, Adaptive Bitonic Sorting: An Optimal Parallel Algorithm for Shared Memory Machines, *SIAM J. Comput.*, **18**(2), 1989, 216–228.
[12] B. Chazelle, A Theorem on Polygon Cutting with Applications, *Proc. 23rd IEEE Symp. on Foundations of Computer Science*, 1982, pp. 339–349.
[13] B. Chazelle, Intersecting is Easier than Sorting, *Proc. 16th ACM Symp. on Theory of Computing*, 1984, pp. 125–134.
[14] B. Chazelle, Triangulating a Simple Polygon in Linear Time, Report CS-TR-264-90, Princeton University, May 1990.
[15] B. Chazelle and L. J. Guibas, Fractional Cascading: I. A Data Structuring Technique, *Algorithmica*, **1**(2), 1986, 133–162.
[16] B. Chazelle and L. J. Guibas, Visibility and Intersection Problems in Plane Geometry, Report CS-TR-167-88, Princeton University, 1988.
[17] R. Cole and U. Vishkin, Approximate Scheduling, Exact Scheduling, and Applications to Parallel Algorithms, *Proc. 27th IEEE Symp. on Foundations of Computer Science*, 1986, pp. 478–491.
[18] R. Cole and U. Vishkin, Optimal Parallel Algorithms for Expression Tree Evaluation and List Ranking, in *Third Aegean Workshop on Computing* (AWOC 88) (J. H. Reif, ed.), Lecture Notes in Computer Science, Vol. 319, Springer-Verlag, Berlin, 1988, pp. 91–100.
[19] R. Cole and O. Zajicek, An Optimal Parallel Algorithm for Building a Data Structure for Planar Point Location, *Parallel Distrib. Comput.*, **8**, 1990, 280–285.
[20] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer-Verlag, New York, 1987.
[21] H. Edelsbrunner, H. A. Maurer, F. P. Preparata, A. L. Rosenberg, E. Welzl, and D. Wood, Stabbing Line Segments, *BIT*, **22**, 1982, 274–281.
[22] H. ElGindy and D. Avis, A Linear Algorithm for Computing the Visibility Polygon from a Point, *J. Algorithms*, **2**, 1981, 186–197.
[23] H. ElGindy and M. T. Goodrich, Parallel Algorithms for Shortest Path Problems in Polygons, *The Visual Computer*, **3**(6), 1988, 371–378.
[24] M. R. Garey, D. S. Johnson, F. P. Preparata, and R. E. Tarjan, Triangulating a Simple Polygon, *Inform. Process Lett.*, **7**(4), 1978, 175–179.
[25] M. T. Goodrich, Triangulating a Polygon in Parallel, *J. Algorithms*, **10**, 1989, 327–351.
[26] M. T. Goodrich, Intersecting Line Segments in Parallel with an Output-Sensitive Number of Processors, *Proc. ACM Symp. on Parallel Algorithms and Architectures*, 1989, pp. 127–137.
[27] L. J. Guibas and J. Hershberger, Optimal Shortest Path Queries in a Simple Polygon, *J. Comput. System Sci.*, **39**, 1989, 126–152.
[28] L. J. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. E. Tarjan, Linear Time Algorithms for Visibility and Shortest Path Problems Inside Simple Polygons, *Proc. 2nd ACM Symp. on Computational Geometry*, 1986, pp. 1–13.
[29] L. J. Guibas, L. Ramshaw, and J. Stolfi, A Kinetic Framework for Computational Geometry, *Proc. 24th IEEE Symp. on Foundations of Computer Science*, 1983, pp. 100–111.
[30] J. Hershberger, Finding the Visibility Graph of a Simple Polygon in Time Proportional to Its Size, *Algorithmica*, **4**, 1989, 141–155.
[31] S. R. Kosaraju and A. L. Delcher, Optimal Parallel Evaluation of Tree-Structured Computations by Raking, in *Third Aegean Workshop on Computing* (AWOC 88) (J. H. Rief, ed.), Lecture Notes in Computer Science, Vol. 319, Springer-Verlag, Berlin, 1988, pp. 101–110.
[32] C. P. Kruskal, L. Rudolph, and M. Snir, The Power of Parallel Prefix, *Proc. Internat. Conf. on Parallel Processing*, 1985, pp. 180–185.
[33] R. E. Ladner and M. J. Fischer, Parallel Prefix Computation, *J. Assoc. Comput. Mach.*, **27**, 1980, 831–838.
[34] D. T. Lee and F. P. Preparata, Euclidean Shortest Paths in the Presence of Rectilinear Barriers, *Networks*, **14**, 1984, 393–410.
[35] U. Manber, *Introduction to Algorithms: A Creative Approach*, Addison-Wesley, Reading, MA, 1989.
[36] G. L. Miller and J. H. Reif, Parallel Tree Contraction and Its Applications, *Proc. 26th IEEE Symp. on Foundations of Computer Science*, 1985, pp. 478–489.

[37] D. E. Muller and F. P. Preparata, Finding the Intersection of Two Convex Polyhedra, *Theoret. Comput. Sci.*, 7(2), 1978, 217–236.
[38] M. H. Overmars and J. Van Leeuwen, Maintenance of Configurations in the Plane, *J. Comput. System Sci.*, **23**, 1981, 166–204.
[39] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, New York, 1985.
[40] C. Rüb, Parallel Line Segment Intersection Reporting, manuscript, 1989.
[41] Y. Shiloach and U. Vishkin, Finding the Maximum, Merging, and Sorting in a Parallel Computation Model, *Algorithms*, **2**, 1981, 88–102.
[42] D. D. Sleator and R. E. Tarjan, A Data Structure for Dynamic Trees, *J. Comput. System Sci.*, **26**, 1983, 362–391.
[43] J. Stolfi, Oriented Projective Geometry, *Proc. 3rd ACM Symp. on Computational Geometry*, 1987, pp. 76–85.
[44] S. Suri, Computing Geodesic Furthest Neighbours in Simple Polygons, *J. Comput. System Sci.*, **39**, 1989, 220–235.
[45] R. Tamassia and J. S. Vitter, Optimal Parallel Algorithms for Transitive Closure and Point Location in Planar Structures, *Proc. ACM Symp. on Parallel Algorithms and Architectures*, 1989, pp. 399–408.
[46] R. E. Tarjan, *Data Structures and Network Algorithms*, SIAM, Philadelphia, PA, 1983.
[47] R. E. Tarjan and U. Vishkin, Finding Biconnected Components and Computing Tree Functions in Logarithmic Parallel Time, *SIAM J. Comput.*, **14**, 1985, 862–874.
[48] R. E. Tarjan and C. J. Van Wyk, An $O(n \log \log n)$ Time Algorithm for Triangulating Simple Polygons, Report CS-TR-052-86, Princeton University, 1986.
[49] H. Wagner, Optimally Parallel Algorithms for Convex Hull Determination, unpublished manuscript, September 1985.
[50] C. K. Yap, Parallel Triangulation of a Polygon in Two Calls to the Trapezoidal Map, *Algorithmica*, **3**, 1988, 279–288.