1985

Efficient Plane Sweeping in Parallel

Mikhail J. Atallah
Purdue University, mja@cs.purdue.edu

Michael T. Goodrich

Report Number:
85-563

EFFICIENT PLANE SWEEPING IN PARALLEL

Mikhail J. Atallah
Michael T. Goodrich

CSD-TR-563
December 1985
Revised October 1986

Efficient Plane Sweeping in Parallel *

Mikhail J. Atallah
Michael T. Goodrich

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

Abstract

We present techniques which result in improved parallel algorithms for a number of problems whose efficient sequential algorithms use the plane-sweeping paradigm. The problems for which we give improved algorithms include intersection detection, trapezoidal decomposition, triangulation, and planar point location. Our technique can be used to improve on the previous time bound while keeping the space and processor bounds the same, or improve on the previous space bound while keeping the time and processor bounds the same. We also give efficient parallel algorithms for 3-dimensional maxima, multiple range-counting, rectilinear segment intersection counting. and visibility from a point. In addition to being asymptotically better than previous solutions, our algorithms do not use the AKS sorting network, thus avoiding the large multiplicative constant found in the time bounds of the previous solutions.

*This research was supported by the Office of Naval Research under Grants N00014-84-K-0502 and N00014-86-K-0689, and the National Science Foundation under Grant DCR-8451393, with matching funds from AT&T.

1 Introduction

The plane-sweeping technique has proven effective for developing efficient sequential algorithms for a variety of geometric problems. This technique, in two dimensions, involves sweeping a line through a set of geometric objects (such as line segments), updating global data structures at *critical* points (sometimes called *event* points, e.g. segment endpoints). It has been used to find efficient sequential algorithms for a host of computational geometry problems (see Lee and Preparata (1984)). It also seems to be a very sequential technique.

Most of the sequential algorithms which use plane-sweeping are already optimal to within a multiplicative constant. There is already a small but growing body of work on finding efficient parallel algorithms for computational geometry problems (e.g., Aggarwal *et al.* (1985), Atallah and Goodrich (1985, 1986), Chow (1980), and ElGindy (1986)), addressing the question of what kinds of speed-ups can be achieved through parallelism. In this paper we present efficient parallel algorithms for a number of problems whose efficient sequential algorithms use the plane-sweeping paradigm. We list the problems addressed in this paper below, and summarize our results in Table 1.

1. **Trapezoidal Decomposition**: Given a simple n-vertex polygon P, determine the trapezoidal edge(s) for each vertex. A *trapezoidal edge* for a vertex v_i is an edge s of P which is directly above or below v_i and such that the vertical line segment from v_i to s is interior to P.

2. **Polygon Triangulation**: Given a simple n-vertex polygon P, augment P with diagonal edges so that each interior face is a triangle.

3. **Arbitrary Triangulation**: Given a set S of n points in the plane, connect pairs of points by edges so that each interior face of the convex hull of S is a triangle.

4. **Planar Point Location**: Given a planar subdivision consisting of n edges, construct in parallel a data structure which, once built, enables one processor to quickly determine for any query point p the face containing p. We let $Q(n)$ denote the time for performing such a query.

5. **Intersection Detection**: Given n line segments in the plane, determine if any two intersect.

6. **3-Dimensional Maxima**: Given a set S of n points in 3-dimensional space, determine which points are maxima. A *maximum* in S is any point p such that no other point of S has x, y, and z coordinates that simultaneously exceed the corresponding coordinates of p.

7. **Two-Set Dominance Counting**: Given a set $V = \{p_1, p_2, \ldots, p_i\}$ and a set $U = \{q_1, q_2, \ldots, q_m\}$ of points in the plane, compute for each point q_i in U the number of points in V which are

2-dominated by q_i. The problem size is $n = l + m$.

8. **Multiple Range-Counting:** Given l points in the plane and m isothetic rectangles (ranges) determine the number of points interior to each rectangle. The problem size is $n = l + m$.

9. **Rectilinear Segment Intersection Counting:** Given n horizontal and vertical line segments in the plane, determine for each segment the number of other segments which intersect it.

10. **Visibility from a Point:** Given n line segments such that no two intersect (except possibly at endpoints) and a point p, determine that part of the plane visible from p.

As in references (Aggarwal *et al.*, 1985) and (Atallah and Goodrich, 1985), our framework is one in which we have a linear number of processors and wish to achieve the best time bound possible. It may also be desirable in this context to try to achieve the best space performance possible as well. Unless otherwise stated, our algorithms will be for the CREW PRAM parallel model. Recall that this is the synchronous parallel model in which processors share a common memory where concurrent reads are allowed, but not concurrent writes.

Aggarwal *et al.* (1985) show that several problems whose efficient sequential algorithms use the plane-sweeping paradigm can be solved in parallel in $O(\log^2 n)$ time and $O(n \log n)$ space using $O(n)$ processors in the CREW PRAM model. The problems addressed by Aggarwal *et al.* include intersection detection, trapezoidal decomposition, polygon triangulation, and planar point location, among others. We reduce the time bound from $O(\log^2 n)$ to $O(\log n \log\log n)$ for each of these problems (keeping the space bound at $O(n \log n)$) by using a special data structure, which we call the *plane-sweep tree*, which is similar to a data structure used by Aggarwal *et al.*, but differs from it in some important ways. We build this data structure by using parallel merging and a technique similar to the sequential "fractional cascading" technique of Chazelle and Guibas (1986). If space is important, then our technique can be modified to achieve $O(n)$ space and $O(\log^2 n)$ time. We manage to achieve $O(n)$ space performance, even though this data structure takes $\Theta(n \log n)$ space, by never completely building it. Instead, we use it as we are constructing parts of it and destroying other parts of it. Also, the previous algorithms use the sorting network of Ajtai, Komlós, and Szemerédi (1983) (sometimes refered to as the AKS sorting network), which introduces a large multiplicative constant into the time complexity. We never use the AKS network.

We also present a technique which we use to efficiently solve other problems as well: namely, 3-dimensional maxima, multiple range-counting, rectilinear segment intersection counting, and visibility from a point. This technique is based on the divide-and-conquer paradigm and for each of these problems it achieves $O(\log n \log\log n)$ time and $O(n)$ space bounds using $O(n)$ processors. Instead of dividing and merging in the usual way, we divide based on how sequential plane-sweeping

stores objects during the sweep, and we "marry" subproblem solutions by merging lists of critical points and computing labels associated with each critical point. The key to this technique is in finding critical-point labels which can be computed quickly in parallel and which can be used to solve the problem at hand once we have completed the divide-and-conquer procedure.

In the next section we give some preliminary definitions and observations. In Section 3 we present the plane-sweep tree technique, and in Section 4 we present our second technique.

2 Preliminaries

In this section we introduce some notation and review some known results which we will use later in the paper. For any point p in the plane we use $x(p)$ and $y(p)$ to denote, respectively, the x and y coordinates of p. If $p \in \Re^3$, then we use $z(p)$ to denote the z-coordinate of p. Given a set S of non-intersecting line segments in the plane, we define a partial order on the elements of S such that two segments in S are comparable iff there is a vertical line which intersects both segments. The segment with the lower intersection is said to be the *smaller* of the two. Note that if there is a vertical line which intersects all the segments in S, then this partial order is actually total.

Given a sorted (nondecreasing) list $B = (b_1, b_2, \ldots, b_m)$ and an element a taken from the same total order as the b_j's, the *predecessor* of a in B is the greatest element in B which is less than or equal to a. If $a < b_1$, then we say that the predecessor of a is ϕ (ϕ is a special symbol such that $\phi < b$ for every element b in the total order). Clearly, we can use binary search to locate the predecessor in B of any such a. The next easy lemma states that if we have two sorted lists A and B whose elements are taken from the same total order, we can find the predecessor in B of every element in A efficiently in parallel.

Lemma 2.1: *Given two sorted arrays A and B whose elements are taken from the same total order, the predecessor in B of each element in A can be determined in $O(\log\log n)$ time using $O(n)$ processors on a CREW PRAM, where $n = |A| + |B|$.*

Proof: The parallel merging algorithm of Valiant (1975) (which Borodin and Hoprocoft (1985) have shown to be implementable in the CREW PRAM model) first finds predecessors and then does the merge. Thus, the lemma follows directly from the work of Valiant and Borodin and Hopcroft. ∎

Parallel merging is a powerful tool in designing efficient parallel algorithms, and we make repeated use of it in this paper. Another powerful parallel technique is the *parallel prefix* technique. Stated in its simplest form, given a sequence of integers $A = (a_1, a_2, \ldots, a_n)$, it allows us to compute all the partial sums $c_k = \sum_{j=1}^{k} a_j$ in $O(\log n)$ time using $O(n/\log n)$ processors (see Kruskal,

Rudolph, and Snir (1985) for a more in-depth study of this technique). Parallel prefix is used as a subroutine in many of our algorithms.

3 The Plane-Sweep Tree Technique

In this section we present the plane-sweep tree technique. We present it for the case when the objects under consideration are line segments in the plane, but essentially the same technique applies for other planar objects as well. We describe the technique in a very general setting, and in the subsequent subsections we show how it can be applied to solve specific problems.

3.1 Definitions and Observations

Let $S = \{s_1, s_2, \ldots, s_n\}$ be a set of non-intersecting line segments in the plane. To simplify the exposition we assume that no two endpoints have the same x-coordinate.

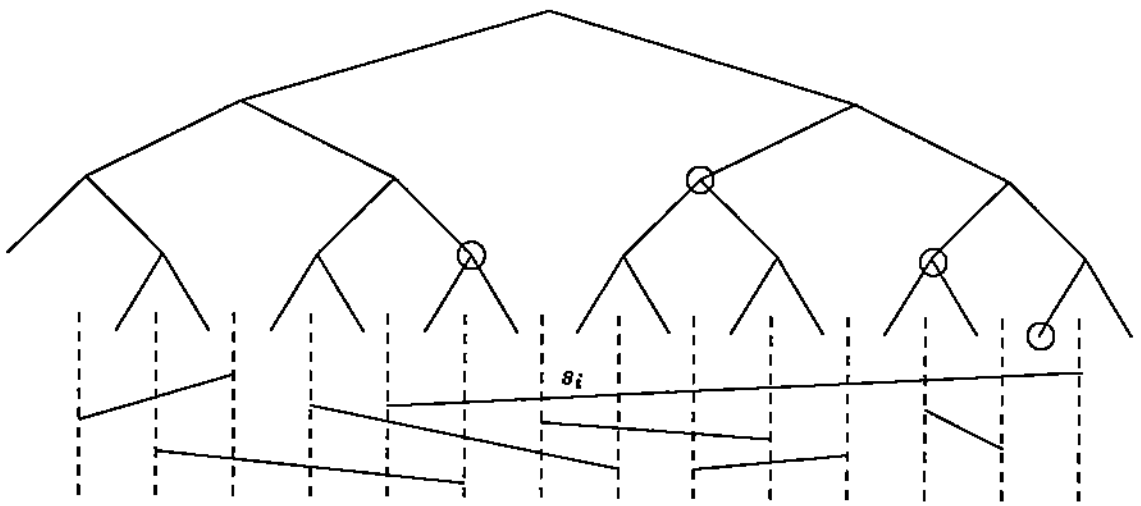


Figure 1: The skeleton of the plane sweep tree. The circled nodes are the nodes of T covered by s_i.

The idea of using a tree to parallelize plane-sweeping is due to Aggarwal *et al.* (1985). We review some of the definitions and observations from their work as it relates to ours. Let T be the complete binary tree with its leaves corresponding to the $2n+1$ intervals formed by projecting the segments' endpoints onto the x-axis. Associated with each node $v \in T$ is an interval $[a_v, b_v]$ on the x-axis which is the union of the intervals associated with the descendants of v. Let Π_v denote the vertical strip $[a_v, b_v] \times (-\infty, \infty)$. A segment s_i *covers* a node $v \in T$ if it spans Π_v but not Π_z, where z is the parent of v (See Figure 1). Clearly, no segment covers more than 2 nodes of any level of T; hence, every segment covers at most $O(\log n)$ nodes of T.

As in (Aggarwal *et al.*, 1985) we define $H(v)$ and $W(v)$ for each node $v \in T$ as follows:

$$\begin{aligned} H(v) &= \{s_i \mid s_i \text{ covers } v\}, \\ W(v) &= \{s_i \mid s_i \text{ has at least one endpoint in } \Pi_v\}. \end{aligned}$$

However, here we also define two other sets, $L(v)$ and $R(v)$. $L(v)$ (resp. $R(v)$) is the set of segments with one endpoint in Π_v and another left (right) of Π_v. More formally, if we let $left(\Pi_v)$ ($right(\Pi_v)$) denote the left (right) vertical boundary of Π_v, then

$$\begin{aligned} L(v) &= \{s_i \mid s_i \in W(v) \text{ and } s_i \cap left(\Pi_v) \neq \emptyset\}, \\ R(v) &= \{s_i \mid s_i \in W(v) \text{ and } s_i \cap right(\Pi_v) \neq \emptyset\}. \end{aligned}$$

We study the relationships between H, L, and R in the following lemma. The observations made in this lemma are needed in the construction presented in the next subsection.

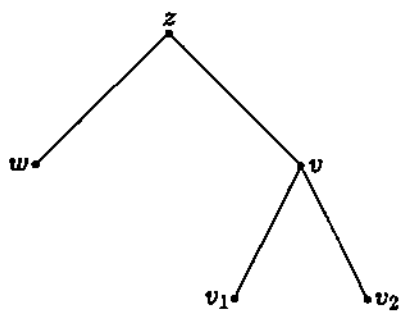


Figure 2: A configuration of nodes in T.

Lemma 3.1: *Let v be a node in T with children v_1 and v_2, sibling w, and parent z (Figure 2 illustrates the case when w is to the left of v). Let $A + B$ denote the union of two disjoint sets A and B, and let $A - B$ denote set difference where $B \subseteq A$. Then we have the following:*

(1) $L(v) = L(v_1) + H(v_1)$;

(2) $R(v) = R(v_2) + H(v_2)$;

(3) $H(v) = R(w) - (R(w) \cap L(v))$ if v is the right child of z (as is the case in Figure 2); $H(v) = L(w) - (L(w) \cap R(v))$, if v is the left child of z.

Proof: We first prove Equation (1). Since Π_{v_1} and Π_v have the same left vertical boundary, $L(v_1) \subseteq L(v)$. Also, from the definition of H we know that all segments in $H(v_1)$ intersect v's left vertical boundary; that is, $H(v_1) \subseteq L(v)$. Noting that $H(v_1) \cap L(v_1) = \emptyset$, we have that $L(v) = H(v_1) + L(v_1)$.

The proof of (2) is similar.

So we have yet to justify Equation (3). First note that by definition all segments in $H(v)$ must have an endpoint in Π_w. If v is the right child of z, then they must intersect Π_w's right vertical boundary (which is also Π_v's left vertical boundary). So, if we remove from $R(w)$ all those segments which have and endpoint in Π_v, then we will have all those segments which span Π_v but do not span Π_z, that is, the set $H(v)$. Therefore, $H(v) = R(w) - (R(w) \cap L(v))$. The argument that $H(v) = L(w) - (L(w) \cap R(v))$ for the case that v is the left child of z is analogous. ∎

Lemma 3.1 essentially states that the sets L, R, and H associated with a node in the tree T can be defined in terms of sets associated with nodes one level below it in T. An important property of the sets $L(v)$, $R(v)$, and $H(v)$ is that for any $v \in T$ the segments in $L(v) \cup H(v)$ (resp., $R(v) \cup H(v)$) can be linearly ordered. We use this fact, and Lemma 3.1, in the next subsection to show how to efficiently construct $H(v)$ for every node v in T.

3.2 Constructing the Plane-Sweep Tree

In this subsection we show how to efficiently construct and traverse the plane-sweep tree T. The next lemma states that the set operations $+$ and $-$ of Lemma 3.1 can both be performed in $O(\log\log n)$ time.

Lemma 3.2: *Let A and B be two sets represented as sorted arrays. If $A \cap B = \emptyset$, then $A + B$ can be computed in $O(\log\log n)$ time using $O(n)$ processors. If $B \subseteq A$, then $A - B$ can be computed in $O(\log\log n)$ time using $O(n)$ processors.*

Proof: If $A \cap B = \emptyset$, then the set $A + B$ can be constructed by simply merging A and B into one sorted list. This can clearly be done in $O(\log\log n)$ time and $O(n)$ processors (Borodin and Hopcroft, 1985; Valient, 1975). If $B \subseteq A$, we construct $A - B$ by first determining the predecessor in B of each $a_i \in A$ (which can be done in $O(\log\log n)$ time by Lemma 2.1). Then, by assigning a processor to each element in A, we compress A by moving each element in A and not in B over by the rank of its predecessor. Since this compressing operation can be done in constant time, the set $A - B$ can be constructed in $O(\log\log n)$ total time. ∎

From Lemma 3.1 we know that the sets L, R, and H for any level l of T can be defined in terms of sets on the level below l. We have yet to see how these sets can be constructed in $O(\log\log n)$ time using a linear number of processors. From Lemma 3.2 we know that the constructions implicit in Equations (1) and (2) of Lemma 3.1 can be performed in $O(\log\log n)$ time. Equation (3), however, also uses set intersection, so we cannot perform the construction implicit in Equation (3) by using Lemma 3.2. To get around this problem we exploit a regularity property of the segments in the intersection $(R(w) \cap L(v))$ of Equation (3) in order to compute all these intersections as a

preprocessing step, storing them away for future use. The details of this and other preprocessing steps follow.

Preprocessing steps:

Input: A set $S = \{s_1, s_2, \ldots, s_n\}$ of non-intersecting segments.

Output: The skeleton of T, the plane-sweep tree for S, with a set $I(v)$ constructed for each node $v \in T$, where $I(v)$ is the set of all segments with one endpoint in $\Pi_{lchild(v)}$ and the other in $\Pi_{rchild(v)}$. (We do not yet compute $L(v)$, $R(v)$, or $H(v)$ for any $v \in T$.)

Step 1. Sort the set of endpoints of $s_1, \ldots, s_n$ by increasing x-coordinate, and build the skeleton of the tree T on top the the $2n + 1$ intervals determined by these endpoints.

Comment: Since we only perform this step once, we can use parallel merging to sort the endpoints in $O(\log n \log\log n)$ time using $O(n)$ processors, instead of using the sorting network of Ajtai, Komlós, and Szemerédi (1983), which would introduce a large multiplicative constant. (Our algorithms take $O(\log n \log\log n)$ anyway, so there is no point in using the AKS network to perform this step in $O(\log n)$ time.)

Step 2. Let J be the set of all (v, s_i) pairs such that v is the lowest node in T such that $s_i \subset \Pi_v$ (that is, v is the least common ancestor of the Π's containing s_i's two endpoints). Clearly, J can be constructed in $O(\log n)$ time using $O(n)$ processors.

Step 3. Sort J lexicographically and use a straight-forward parallel prefix type of computation, to compute the set $I(v) = \{s_i \mid (v, s_i) \in J\}$ for each $v \in T$.

Comment: Observe that $\sum_{v \in T} |I(v)| = n$.

Step 4. Sort each $I(v)$ by the y-coordinates of the intersections of the s_i's in $I(v)$ with the vertical boundary separating the vertical strips $\Pi_{lchild(v)}$ and $\Pi_{rchild(v)}$.

End of Preprocessing Steps.

Observation 3.3: *The preprocessing steps take $O(\log n \log\log n)$ time and $O(n)$ space using $O(n)$ processors on a CREW PRAM. For each $v \in T$ the set $I(v)$ consists of all segments with one endpoint in $\Pi_{lchild(v)}$ and the other in $\Pi_{rchild(v)}$.*

Proof: Immediate. ■

Note that the set $R(w) \cap L(v)$, as well as $L(w) \cap R(v)$, of Equation (3) in Lemma 3.1 is exactly the set of all segments with one endpoint in Π_w and the other in Π_v. Thus, by Observation 3.3, we can rewrite Equation (3) of Lemma 3.1 as $H(v) = R(w) - I(z)$ if v is a right child, and $H(v) = L(w) - I(z)$ otherwise. Having observed this, we are now ready to describe how to construct the plane-sweep tree T.

The Build-Up Algorithm (BUILDUP):

Input: The skeleton of the plane-sweep tree T built in the preprocessing steps (including the sets $I(v)$ for each $v \in T$).

Output: The plane-sweep tree T with the set $H(v)$ constructed for every node $v \in T$. The contents of each $H(v)$ are sorted by the "above" relationship defined in Section 2.

Step 0. Initialize T by constructing $L(v)$, $R(v)$, and $H(v)$ for each leaf v in T. Note that each of these sets will have at most 1 entry.

Step 1. **For** $l =$ lowest level of T **until** $l = 0$ **repeat** Steps 2–4 below, in parallel for each vertex v at level l in T.

Step 2. Use equations (1) and (2) of Lemma 3.1 and Lemma 3.2 to build the sets $L(v)$ and $R(v)$ from the sets for v's children.

Step 3. Use the modified equation (3) of Lemma 3.1 (that is, $H(v) = R(w) - I(z)$ if v is a right child, and $H(v) = L(w) - I(z)$, otherwise) and Lemma 3.2 to build $H(v)$ from $I(z)$ (which was precomputed) and the appropriate $R(w)$ or $L(w)$ constructed in Step 2.

Step 4. Discard the sets L and R for the nodes on level $l+1$ (the level below l), as they are no longer needed.

End of Algorithm BUILDUP.

Theorem 3.4: *The BUILDUP algorithm correctly builds the set $H(v)$ for every node v in T in $O(\log n \log\log n)$ time and $O(n \log n)$ space using $O(n)$ processors on a CREW PRAM.*

Proof: The correctness of BUILDUP follows from Lemma 3.1, the fact that the segments in $L(v)$ (resp., $R(v)$ or $H(v)$) are linearly ordered, and the fact that the segments in $L(v) \cup H(v)$ (resp., $R(v) \cup H(v)$) are totally ordered. Steps 1 and 2 are performed by using Lemma 3.2 and therefore take $O(\log\log n)$ time. Also, Step 3 clearly takes $O(1)$ time. For any node v the number of processors necessary to perform Steps 1–3 for v is proportional to the number of descendants of v. Since Steps 1–3 are performed for nodes which are all on the same level of T in parallel, we use $O(n)$ processors. The fact that we use at most $O(n \log n)$ space follows from the fact that a segment can cover at most $2 \log n$ nodes of T. Thus, the BUILDUP algorithm runs in $O(\log n \log\log n)$ time and $O(n \log n)$ space using $O(n)$ processors. ■

We are now ready to show how to traverse the plane-sweep tree. In all the problems we solve using this technique, an essential computation done while traversing the plane-sweep tree is that we want to locate for each input point p the segment in $H(v)$ which is directly above (or below) p, for all $v \in T$ such that $p \in \Pi_v$. We call this computation the *multilocation* of p in T. The specific set of multilocations we will perform will vary from problem to problem, and will become apparent in the subsections on applications. We augment T with sets and pointers in a manner similar to the sequential "fractional cascading" technique of Chazelle and Guibas (1986) so that the multilocation of any query point p can be performed in $O(\log n)$ serial time. To perform the multilocation of a point p we first find the leaf $v \in T$ such that $x(p) \in [a_v, b_v]$. Then, for every node z on the path from v to the root, we search in $H(z)$ to find the segments in $H(z)$ which are

directly above or below p (note that this leaf-to-root path consists of all nodes $z \in T$ such that $p \in \Pi_v$). The main idea of the augmenting technique is that we want the search done at a node v to allow us to perform the search at $parent(v)$ in constant time (rather than in $O(\log n)$ time). As in (Chazelle and Guibas, 1986) we make the following definition: given a sorted sequence A the *k-sample* of A, denoted $SAMP_k(A)$, is a sequence consisting of every k-th element of A.

The Algorithm AUGMENT:

Input: A set S of non-intersecting line segments in the plane, and the plane-sweep tree T built for S, with the sets $H(v)$ constructed for every node $v \in T$ (as produced by the BUILDUP algorithm).

Output: An augmented plane-sweep tree T', which allows a multilocate of any query point p to be done in $O(\log n)$ serial time.

Method: The idea is to construct an augmented list $A(v)$ for every node $v \in T$ such that $H(v) \subseteq A(v)$, and associate pointers with the elements of $A(v)$ so that, given the position of an element in $A(v)$, we can locate that element in both $H(v)$ and $A(parent(v))$ in $O(1)$ additional time.

Step 1. Let $A(r) = H(r)$, where r is the root of the plane-sweep tree T.

Step 2. For $l = 1$ (the level just below the root) **until** $l =$ lowest level **repeat** Steps 3–5 below in parallel for each vertex $v \in T$ on level l.

Step 3. Merge $H(v)$ and $SAMP_4(A(z))$ into one sorted list and store this list as $A(v)$, where $z = parent(v)$.

Step 4. Use Lemma 2.1 to determine for each $s_i \in A(v)$ its predecessor in $A(z)$. For each $s_i \in A(v)$ let $up(s_i)$ be a pointer to the predecessor of s_i in $A(z)$.

Step 5. Use Lemma 2.1 to determine for each $s_i \in A(v)$ its predecessor in $H(v)$. For each $s_i \in A(v)$ let $over(s_i)$ be a pointer to the predecessor of s_i in $H(v)$.

End of AUGMENT.

Theorem 3.5: *AUGMENT runs in $O(\log n \log\log n)$ time and $O(n \log n)$ space using $O(n)$ processors on a CREW PRAM. The augmented tree T' it produces allows us to multilocate any query point p in $O(\log n)$ serial time.*

Proof: We first prove that the space complexity of T' is the same as that of T, namely, $O(n \log n)$. We prove this by examining the extent to which any set $H(v)$ contributes to the space of T'. For any $v \in T$, on level l, AUGMENT copies $|H(v)|/2$ elements to nodes on level $l+1$, $|H(v)|/4$ to level $l+2$, and so on. Thus, any set $H(v)$ contributes at most $|H(v)|$ extra space to T'. Therefore, the space required by T' is at most 2 times the space used by T. Hence, the space complexity of AUGMENT is $O(n \log n)$. That the number of processors used is $O(n)$ follows by a similar argument. In order to do the parallel merges we need to know ahead of time how many elements will be involved, for all $v \in T$. This is not a problem, however, because we can calculate the number of processors needed to compute $A(v)$ for each $v \in T$ as a preprocessing step. The time complexity

of AUGMENT is clearly $O(\log n \log\log n)$, since Steps 3–5 are all done using parallel merging or Lemma 2.1.

A multilocate of a point p proceeds as follows (WLOG, we describe the version which finds the segments directly below p in the appropriate $H(v)$'s, the version for finding segments above p being similar). Locate the leaf v in T corresponding to the interval $[a_v, b_v]$ such that $x(p) \in [a_v, b_v]$. We begin the sequence of searches by using binary search to locate the segment in $A(v)$ which is directly below p; this is the predecessor of p in $A(v)$. Let $c_v(p)$ denote this segment. We can then follow the pointer $over(c_v(p))$ to find the segment in $H(v)$ which is directly below p. Now, by following the pointer $up(c_v(p))$ to the list $A(z)$, where $z = parent(v)$, we can use a sequential search from $up(c_v(p))$ to locate the segment $c_z(p)$ in $A(z)$ which is directly below p in $O(1)$ time. This is because $c_z(p)$ can be no more than 4 storage locations away from $up(c_v(p))$ in the array $A(z)$. From this point on every search will take $O(1)$ time to complete. Since there are $O(\log n)$ nodes which must be searched, the sequence of searches can be performed in $O(\log n)$ total time. ∎

We show in the following subsections how to apply BUILDUP and AUGMENT to solve specific geometric problems. Before doing so, however, we describe how to perform a collection of m multilocations using only $O(n)$ space, at the expense of more time. Let $V = \{p_1, p_2, \ldots, p_m\}$ be a set of points we wish to multilocate in T, where $m = O(n)$. The method is similar to the BUILDUP procedure, but differs from it in two respects. First, after constructing the set $H(v)$ for all v on a level l (in Step 2), we perform a binary search in $H(v)$ for all points p_i such that $p_i \in \Pi_v$ to find the segments in $H(v)$ directly above and below p_i (this is one of the searches needed for the multilocation of p_i). Next, after we have completed the searches of nodes on level l for all points $p_i \in V$, we can discard the sets L, R, and H for all nodes on level $l+1$ (this of course means that we do not output any $H(v)$'s as BUILDUP does). Since we never construct sets for more than 2 levels in the tree at a time, we never use more than $O(n)$ space. Also, recall that the space used by all the $I(v)$'s is $O(n)$. The time taken for this is clearly $O(\log n)$ for each level of T, or $O(\log^2 n)$ overall. We summarize the above discussion in the following theorem.

Theorem 3.6: *Given a set S of n non-intersecting segments and a set V of $O(n)$ query points, we can perform the multilocation of all the points in V in $O(\log n \log\log n)$ time and $O(n \log n)$ space (or, alternatively, in $O(\log^2 n)$ time and $O(n)$ space) using $O(n)$ processors on a CREW PRAM.* ∎

We are now ready to show how the plane-sweep tree technique is used to solve a number of geometric problems. The first application we present is for trapezoidal decomposition.

3.3 Trapezoidal Decomposition

Let $P = \{v_1, v_2, \ldots, v_n\}$ be a simple polygon, where the v_i's denote the vertices of P and are listed so that the interior of P is to the left of the walk $v_1 v_2 \ldots v_n$. For any vertex v_i of P a *trapezoidal*

edge for v_i is an edge of P which is directly above or below v_i and such that the vertical line segment from v_i to this edge is interior to P. Note that a vertex can have 0, 1 or 2 trapezoidal edges. The trapezoidal decomposition problem is to find the trapezoidal edge(s) for each vertex of P (see Figure 3).

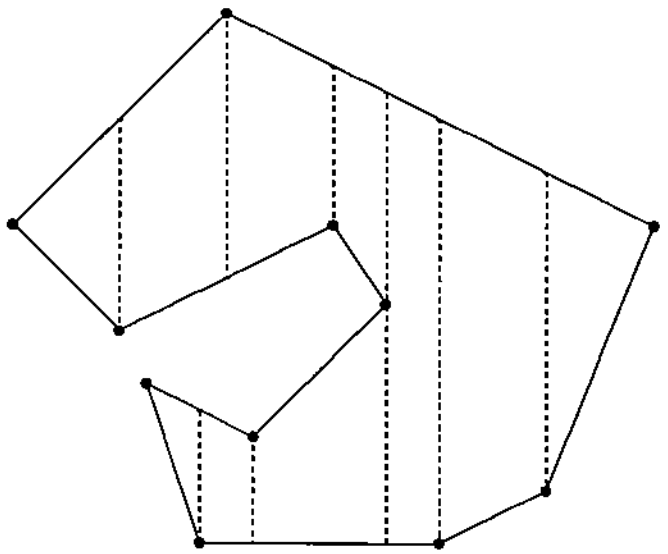


Figure 3: A trapezoidal decomposition of a simple polygon.

Theorem 3.7: *A trapezoidal decomposition of P can be constructed in $O(\log n \log\log n)$ time and $O(n \log n)$ space (or, alternatively, in $O(\log^2 n)$ time and $O(n)$ space) using $O(n)$ processors on a CREW PRAM.*

Proof: We first prove the $O(\log n \log\log n)$ time result. Let $S = \{s_1, s_2, \ldots, s_n\}$ be the set of edges of P, i.e., $s_i = (v_i, v_{i+1})$, for $i = 1, 2, \ldots, n-1$, and $s_n = (v_n, v_1)$. We find the trapezoidal edge (if any) below each vertex as follows. First, use algorithms BUILDUP and AUGMENT to construct an augmented plane-sweep tree T' for S. As in (Aggarwal *et al.*, 1985), we solve the problem by performing a multilocation of each $v_i \in P$. In our case we use Theorem 3.5 to perform all $O(n)$ multilocates in $O(\log n)$ time using $O(n)$ processors. During the multilocation, for each vertex v_i, we keep track of the segment below v_i and with minimum vertical distance from v_i (call this segment $trap(v_i)$). When we complete all the multilocations, for each v_i, $trap(v_i)$ will store the segment which is directly below v_i in the totally ordered set of segments that are cut by the

vertical line through v_i (i.e., the union of all $H(v)$ such that $v_i \in \Pi_v$). By a similar procedure we can find for each v_i the segment in S which is directly above v_i. We can then test in constant time if these segments are trapezoidal edges or not by checking if the line segment from v_i to the segment $trap(v_i)$ is interior to P or not.

Since the necessary multilocations can alternatively be performed in $O(\log^2 n)$ time and $O(n)$ space using $O(n)$ processors (by Theorem 3.6), we can construct a trapezoidal decomposition of P in these same bounds. ■

It is easy to see that roughly the same technique can be used to find the trapezoidal decomposition of a arbitrary set of non-intersecting line segments, with the same complexity bounds. We chose to describe the solution for simple polygons because in the next subsection we show how to use trapezoidal decomposition in solving the polygon triangulation problem.

3.4 Triangulation

Let $P = \{v_1, v_2, \ldots, v_n\}$ be a simple polygon, where the v_i's denote the vertices of P and are listed so that the interior of P is to the left of the walk $v_1 v_2 \ldots v_n$. We wish to augment P with diagonal edges so that each interior face of the resulting planar subdivision is a triangle. Our method consists of two phases. The first is to use trapezoidal decomposition to decompose P into one-sided monotone polygons $P_1, P_2, \ldots, P_k$. We say that a polygon P is *one-sided* if there is a distinguished edge on P such that the vertices of P are all above (or all below) that edge (except for the endpoints of the edge). In the second phase we triangulate each P_i in $O(\log n)$ time and $O(n)$ space using $O(n)$ processors. The algorithm DECOMP which follows is the first phase in our triangulation procedure.

Algorithm DECOMP:

Input: A simple polygon $P = \{v_1, v_2, \ldots, v_n\}$.

Output: A decomposition of P into one-sided monotone polygons.

Step 1. Construct a trapezoidal decomposition for P.

Step 2. For every s_i construct V_i, the set of vertices of P for which s_i is a trapezoidal edge. This can be done by sorting lexicographically the set of (s_i, v_j) pairs such that s_i is a trapezoidal edge for v_j, and then using a parallel prefix computation to construct the set V_i for each s_i.

Step 3. Sort the vertices in every V_i by x-coordinate, in parallel.

Step 4. For each edge $s_i = (v_{i_0}, v_{n_i+1})$, suppose $V_i = \{v_{i_1}, v_{i_2}, \ldots, v_{i_{n_i}}\}$. Augment P by adding edges $(v_{i_j}, v_{i_{j+1}})$ for $j = 0, 1, 2, \ldots, n_i$, if they are not already in P. Let P_i be the polygon consisting of s_i and of the edges $(v_{i_j}, v_{i_{j+1}})$, for $j = 0, 1, \ldots, n_i$ (see Figure 4).

End of algorithm DECOMP.

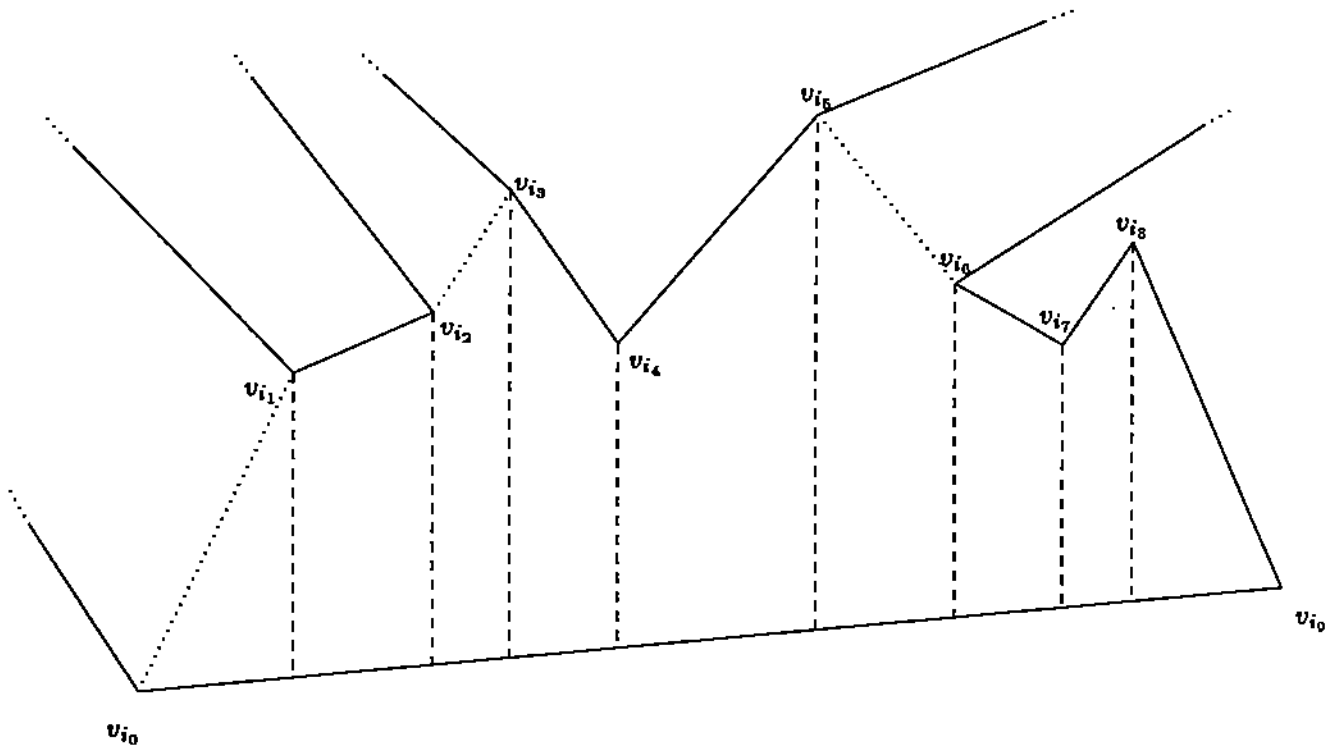


Figure 4: The polygon P_i for $s_i = (v_{i_0}, v_{i_9})$ and $V_i = \{v_{i_1}, v_{i_2}, \ldots, v_{i_8}\}$. The edges in P_i but not in P are shown dotted. Note that the sequence $v_{i_0}, v_{i_1}, v_{i_2}, \ldots, v_{i_9}$ is monotone in the x-direction.

Theorem 3.8: *The algorithm DECOMP correctly decomposes a simple polygon P into one-sided monotone polygons in $O(\log n \log\log n)$ time and $O(n \log n)$ space (or, alternatively, in $O(\log^2 n)$ time and $O(n)$ space) using $O(n)$ processors on a CREW PRAM.*

Proof: First, note that the P_i's form a decomposition. That is, an edge added to construct some P_i may coincide with an edge added to construct some P_j, but it cannot cut across any other edge. This is because edges are only added between vertices which are both vertically visible from the same segment. Second, the vertices of V_i are all on the same side of s_i, because the vertical line from any point in V_i to the segment s_i must be interior to P, and the interior of P can only be on one side of s_i. Thus, each P_i is one-sided. Finally, each P_i is monotone because we sorted the points in V_i by x-coordinate in Step 3. The complexity bounds for DECOMP follow from observations made in Section 3.3. ∎

After decomposing P into polygons $P_1, P_2, \ldots, P_k$, we now triangulate each P_i in parallel. So let us concentrate on the problem of triangulating a one-sided monotone polygon. WLOG, we are given a one-sided monotone polygon which is monotone in the x-direction and the vertices not on the distinguished edge s are all above s. Let a one-sided monotone polygon $P = (v_1, v_2, \ldots, v_n, v_1)$ be given, where $v_1 v_n$ is the distinguished edge. We triangulate P by using the following algorithm

to triangulate the underside of the monotone polygonal chain $V = (v_1, v_2, \ldots, v_n)$ in $O(\log n)$ time and $O(n)$ space using $O(n)$ processors.

Algorithm CHAIN-TRIANGULATE(V):

Input: A monotone polygonal chain $V = \{v_1, v_2, \ldots, v_n\}$. WLOG, we assume that the indices of V are listed by increasing x-coordinate.

Output: A listing $LH(V)$ of the vertices belonging to the lower convex hull of V, sorted by increasing x-coordinate, and a triangulation of the under side of V (i.e., the region bounded from above by V and from below by $LH(V)$). (By symmetry, we could alternatively output a triangulation of the upper side of V, and a listing of the upper hull vertices.)

Method: One of the ideas in our algorithm is the use of the $\sqrt{n}$ parallel divide-and-conquer technique of Aggarwal *et al.* (1985) and Atallah and Goodrich (1985). We divide the vertices of V into $\sqrt{n}$ subsets $V_1, V_2, \ldots, V_{\sqrt{n}}$ containing $\sqrt{n}$ vertices each, and recursively solve the problem for each V_i in parallel. The main ideas behind our method for merging the subproblem solutions are the following. We first build a binary tree "on top" of the collection of subproblem solutions so that each leaf corresponds to a subproblem. (We use V_i to denote both the set and the leaf associated with it; the context will make clear which one is meant.) For any node w in this tree let $lchild(w)$ ($rchild(w)$) denote the left (right) child of w, let $parent(w)$ denote the parent node of w, and let $Desc(w) = \{v_i \mid v_i \in V_j$ and V_j is a descendent of $w\}$. In parallel for each node w in the tree, we construct the polygon P_w whose boundary consists of the following: the vertices in $LH(Desc(lchild(w)))$ not in $LH(Desc(w))$, followed by the edge of V between $Desc(lchild(w))$ and $Desc(rchild(w))$, followed by the vertices in $LH(Desc(rchild(w)))$ not in $LH(Desc(w))$, and finally followed by the supporting lower common tangent of $LH(Desc(lchild(w)))$ and $LH(Desc(rchild(w)))$ (See Figure 7). We then use the special structure of the P_w's to triangulate each P_w in $O(\log n)$ time using $O(|P_w|)$ processors. Since the P_w's form a decomposition of the untriangulated region below V and above $LH(V)$, this completes the triangulation of the underside of V. Incidentally, the lower convex hull $LH(V)$ is computed as a by-product of the step which constructs all the P_w's. The details of this algorithm follow.

Step 1. Divide V into $\sqrt{n}$ subsets $V_1, V_2, \ldots, V_{\sqrt{n}}$ of size $\sqrt{n}$ each using vertical dividing lines.
Time: $O(1)$; *Space:* $O(n)$; *Processors:* $O(n)$.

Step 2. Recursively call CHAIN-TRIANGULATE(V_i) in parallel for all V_i's.
Time: $T(\sqrt{n})$; *Space:* $\sqrt{n}S(\sqrt{n})$; *Processors:* $\sqrt{n}P(\sqrt{n})$, where $T(n)$ ($S(n)$) is the time (space) complexity of CHAIN-TRIANGULATE, and $P(n)$ is the number of processors used.

Step 3. Build a complete binary tree B "on top" of the subsets V_i such that each leaf corresponds to a V_i (see Figure 5). For each $w \in B$ find the common tangent t_w supporting

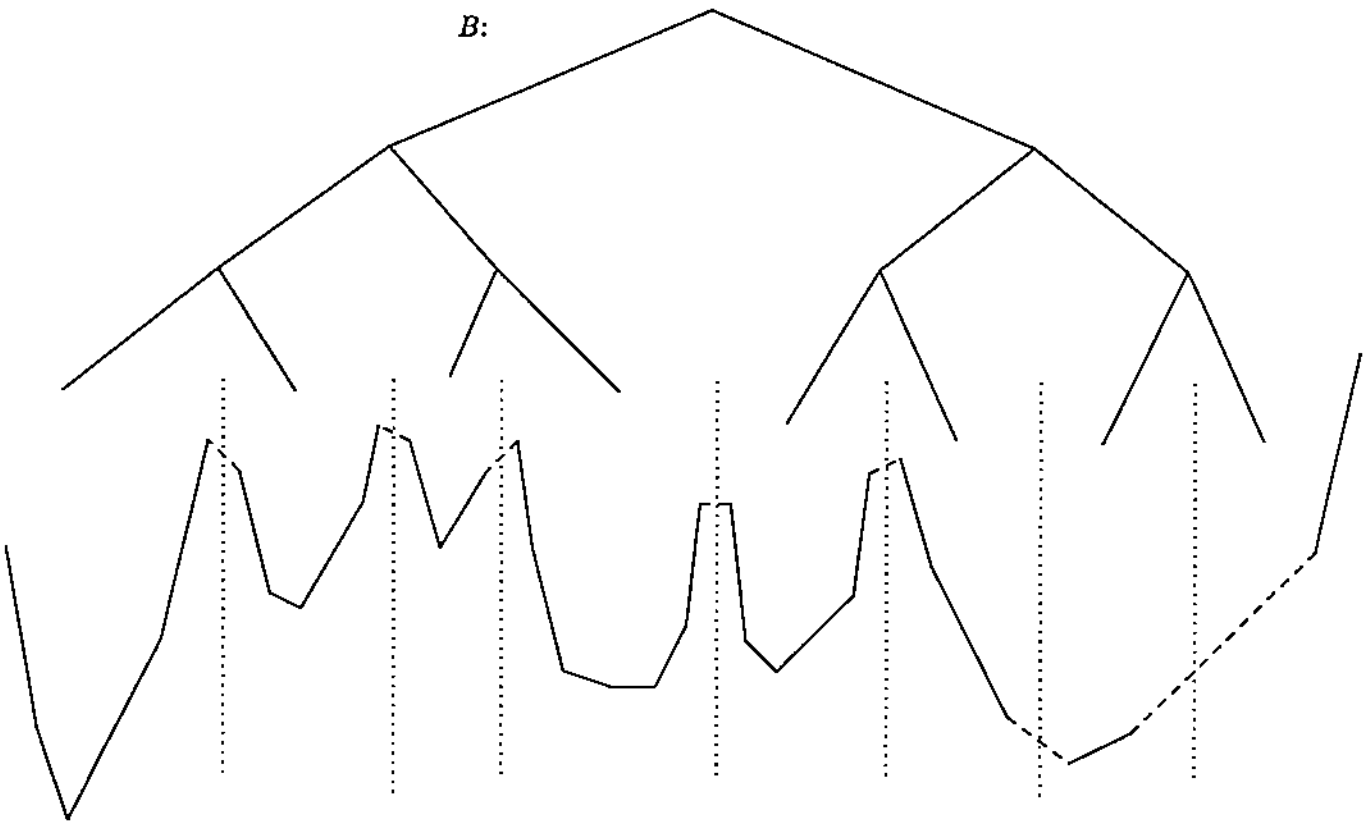


Figure 5: The binary tree B and the $\sqrt{n}$ lower hulls $LH(V_i)$ associated with the leaves of B.

$LH(Desc(lchild(w)))$ and $LH(Desc(rchild(w)))$ by performing Steps 3.1–3.2 below, where $Desc(w) = \{v_i \mid v_i \in V_j$ and V_j is a descendent of $w\}$. Note: we don't actually compute $LH(Desc(w))$, just the common tangent t_w.

Time: $O(\log n)$; *Space:* $O(n)$; *Processors:* $O(n)$.

Step 3.1. For each pair (i,j), $i,j = 1,2,\ldots,\sqrt{n}$, compute the common tangent line $t_{i,j}$ supporting $LH(V_i)$ and $LH(V_j)$ in parallel.

Comment: The common tangent line supporting two lower hulls can be computed in $O(\log n)$ time by a single processor using a binary search technique developed by Overmars and Van Leeuwen (1981). Thus, this step can be done in $O(\log n)$ time by assigning one processor to each of the $O(n)$ pairs of lower hulls.

Step 3.2. For each $w \in B$ let T_w be the set of tangent lines $t_{i,j}$ such that V_i is a descendant of $lchild(w)$ and V_j is a descendant of $rchild(w)$. In parallel for each T_w, find the minimum tangent line t_w, where comparisons are based on the y-coordinate of the intersection of each tangent line with a vertical line L_w separating the descendants of $lchild(w)$ and $rchild(w)$, respectively (see Figure 6). This can be done in $O(\log n)$ time using $|T_w|$ processors per T_w, or n processors overall.

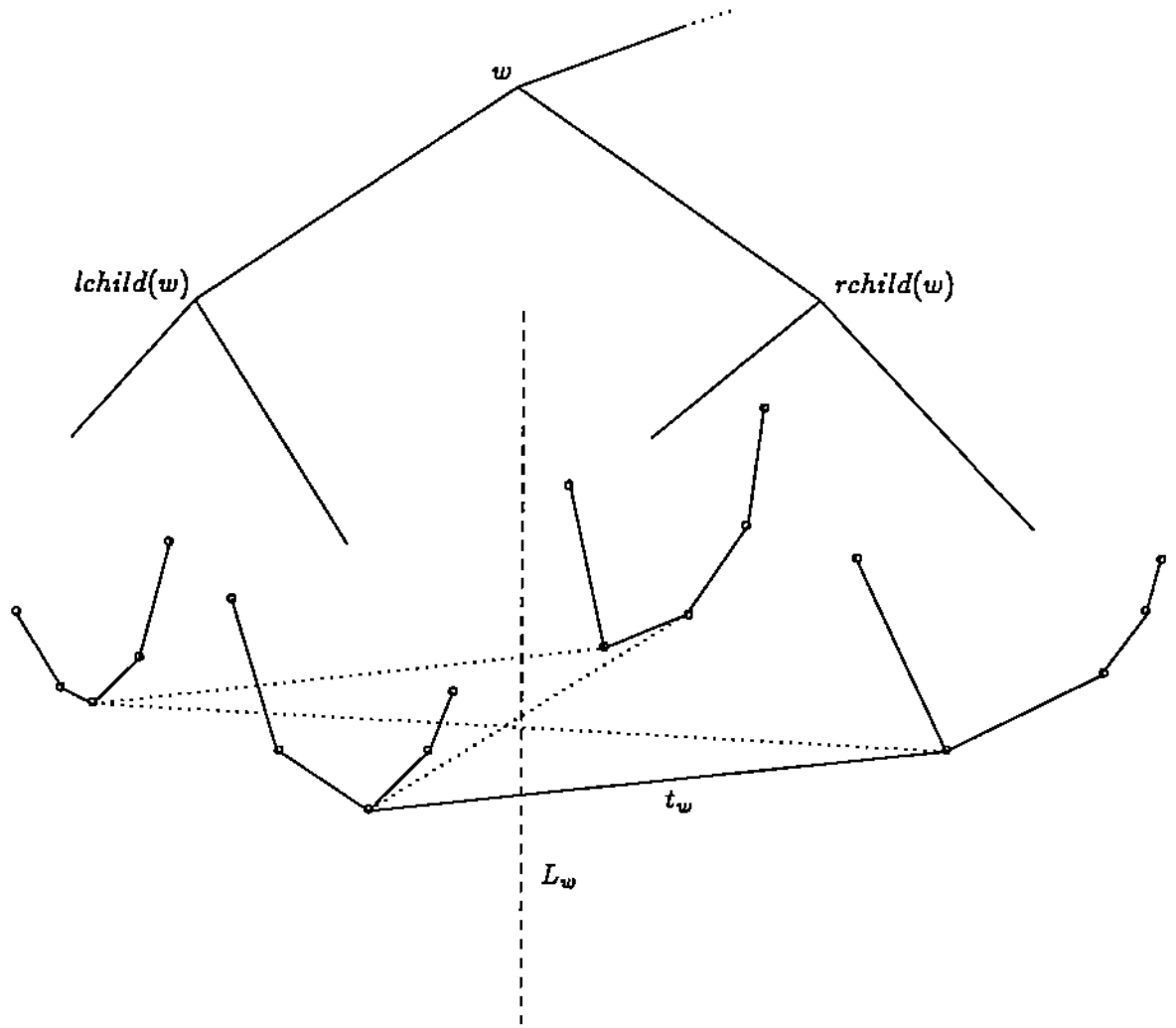


Figure 6: The tangent lines in T_w, supporting the descendents of *lchild*(w) and *rchild*(w). The tangent t_w is shown as a solid line, and the others are shown dotted.

Comment: The common tangent supporting $LH(Desc(lchild(w)))$ and $LH(Desc(rchild(w)))$ must be a member of T_w, and it cannot intersect L_w above t_w's intersection with L_w. Thus, t_w is, in fact, the lower common tangent supporting the vertices of $LH(Desc(lchild(w)))$ and $LH(Desc(rchild(w)))$.

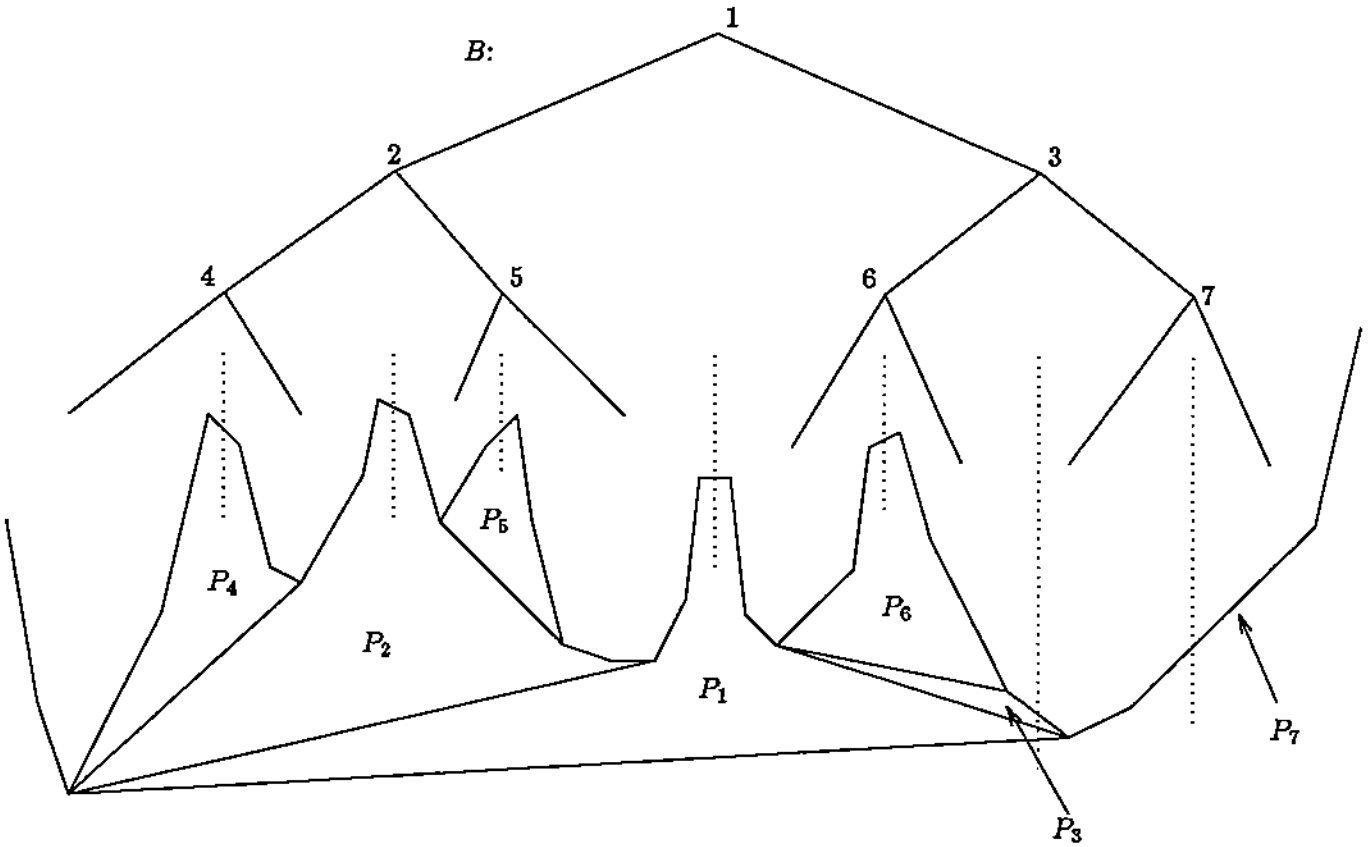


Figure 7: The polygons P_w for each internal node $w \in B$. Note that the polygon P_3 is simply a triangle and the polygon P_7 is just a line segment.

Step 4. In this step we construct $LH(V)$ and for each $w \in B$ we construct the polgyon P_w, where $P(w)$ is the polygon whose boundary consists of the vertices of $LH(Desc(lchild(w)))$ not in $LH(Desc(w))$, the edge of V between $Desc(lchild(w))$ and $Desc(rchild(w))$, the vertices of $LH(Desc(rchild(w)))$ not in $LH(Desc(w))$, and t_w (see Figure 7). This is done by performing the computation below.

Time: $O(\log n)$; *Space:* $O(n)$; *Processors:* $O(n)$.

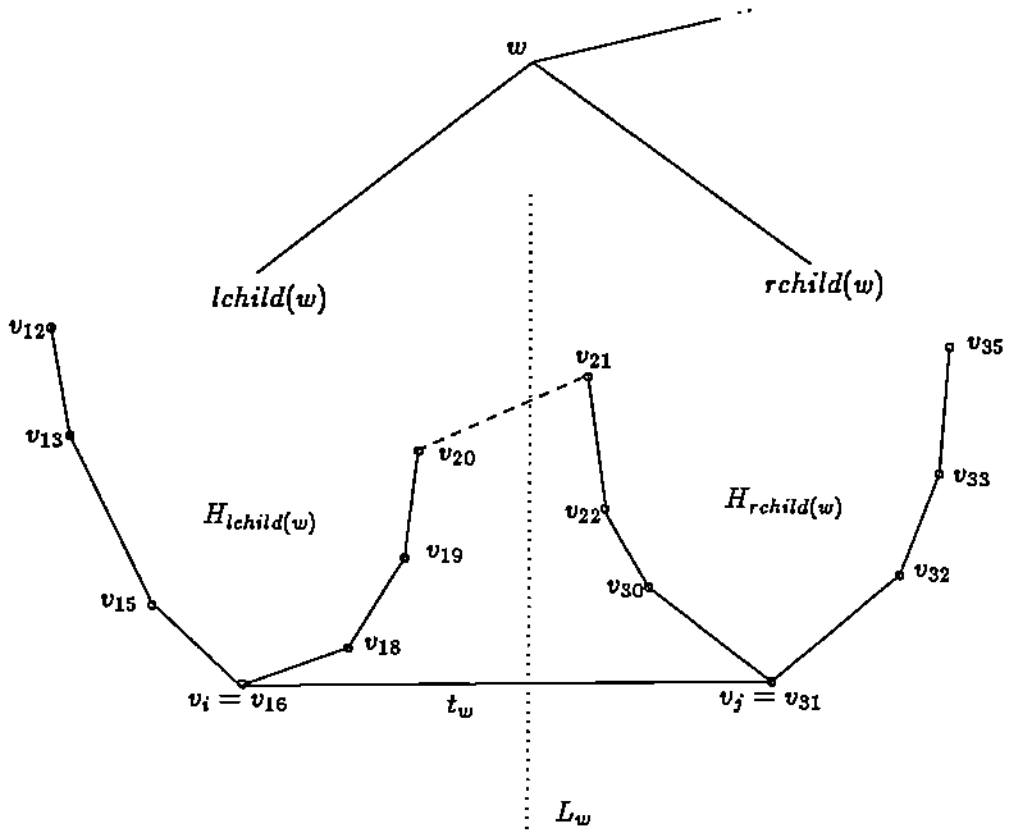


Figure 8: Constructing P_w and H_w from $H_{lchild(w)}$, $H_{rchild(w)}$, and t_w. In this case $H_w = (v_{12}, v_{13}, v_{15}, v_{16}, v_{31}, v_{32}, v_{33}, v_{35})$ and $P_w = (v_{16}, v_{18}, v_{19}, v_{20}, v_{21}, v_{22}, v_{30}, v_{31})$.

For each leaf node w let $H_w = LH(V_i)$, where V_i is the monotone
    chain associated with w in B;
For $l =$ the penultimate level of B to the top level of B **do**
    For each node w on level l **pardo**
        Let $t_w = v_i v_j$ be the common tangent
            joining $LH(Desc(lchild(w)))$ and $LH(Desc(rchild(w)))$;
        Split $H_{lchild(w)}$ at v_i, so that $H_{lchild(w)} = H_{1,lchild(w)} + (v_i) + H_{2,lchild(w)}$;
            {We use + to denote the concatenation of vertex lists}
        Split $H_{rchild(w)}$ at v_j, so that $H_{rchild(w)} = H_{1,rchild(w)} + (v_j) + H_{2,rchild(w)}$;
        Set $P_w := (v_i) + H_{2,lchild(w)} + H_{1,rchild(w)} + (v_j)$;
        Set $H_w := H_{1,lchild(w)} + (v_i, v_j) + H_{2,rchild(w)}$;
        Discard $H_{lchild(w)}$, $H_{rchild(w)}$, $H_{1,lchild(w)}$, $H_{2,lchild(w)}$, $H_{1,rchild(w)}$, and $H_{2,rchild(w)}$;
        { P_w is the polygon associated with w, and $L(w)$ is the lower
        convex hull of $Desc(w)$ (see Figure 8) }
    EndParFor
EndFor $\{H_{root} = LH(V)\}$

Comment: For any $w \in B$ each of the splitting and concatenating steps of the inner (parallel) for-loop can be performed in $O(1)$ time using $O(|Desc(w)|)$ processors. Thus, Step 4 runs in $O(\log n)$ time using $O(n)$ processors. That the space complexity is $O(n)$ follows from Euler's theorem (the P_w's, t_w's, and $LH(Desc(w))$'s from a planar graph having vertex set V).

Step 5. Triangulate each P_w in parallel by performing Steps 5.1–5.2 below.
Time: $O(\log n)$, *Space:* $O(n)$, *Processors:* $O(n)$.

Step 5.1. For each $v_i \in P_w$ find the edge in P_w which is intersected by the line containing v_i and parallel to t_w. Let e_i denote the edge chosen for v_i (e.g., in Figure 9 e_{11} is the edge v_8v_9). This can be done in $O(\log n)$ time and $O(|P_w|)$ processors by doing a binary search for each $v_i \in P_w$ in parallel.

Comment: Note that this implies that the vertex v_i is visible from the lower of the two endpoints of e_i.

Step 5.2. Augment P_w by adding an edge from each v_i to the lower of the two endpoints of e_i (see Figure 9). This can clearly be done in $O(1)$ time using $O(|P_w|)$ processors.

Comment: We show below that in adding an edge from each v_i to the lower endpoint of the corresponding e_i we triangulate P_w. Since the P_w's form a partition of the region between V and $LH(V)$, this completes the triangulation of the underside of V.

End of CHAIN-TRIANGULATE.

Theorem 3.9: *Given a monotone polygonal chain V the algorithm CHAIN-TRIANGULATE correctly triangulates the under side of V, as well as computing the lower convex hull $LH(V)$, in $O(\log n)$ time and $O(n)$ space using $O(n)$ processors on a CREW PRAM.*

Proof: We first show that CHAIN-TRIANGULATE is correct. The correctness of the algorithm depends on the following facts: (1) t_w is the tangent line between $LH(Desc(lchild(w)))$ and $LH(Desc(rchild(w)))$ for all $w \in B$, (2) the P_w's form a decomposition of the region bounded from above by V and bounded from below by $LH(V)$, and (3) each P_w is triangulated correctly. Facts (1) and (2) follow immediately from comments made above, so we have yet to show that each P_w is triangulated correctly. That is, we need to show that, for any P_w, in adding an edge from each $v_i \in P_w$ to the lower endpoint of the edge e_i we form a triangulation of P_w. Consider any edge e in P_w, other than t_w, with endpoints v_i and v_j. Let v_k (v_l) be the lower endpoint of e_i (e_j). It is enough to show that the slice of P_w between v_iv_k and v_jv_l is triangulated correctly. WLOG, t_w is parallel to the x-axis, $y(v_i) < y(v_j)$, and the edge e has positive slope. If $v_k = v_l$, then in adding the edges v_iv_k and v_jv_l we construct the triangle $v_iv_jv_k$ (See Figure 10.a). If $v_k \neq v_l$, then there must be a chain of vertices $(v_k = v_{k_1}, v_{k_2}, \ldots, v_{k_i} = v_l) \subset P_w$ such that $y(v_{k_1}) < y(v_i) < y(v_{k_2}) < y(v_{k_3}) < \cdots < y(v_{k_i}) < y(v_j)$ (see Figure 10.b). Thus, in Step 5.2 we

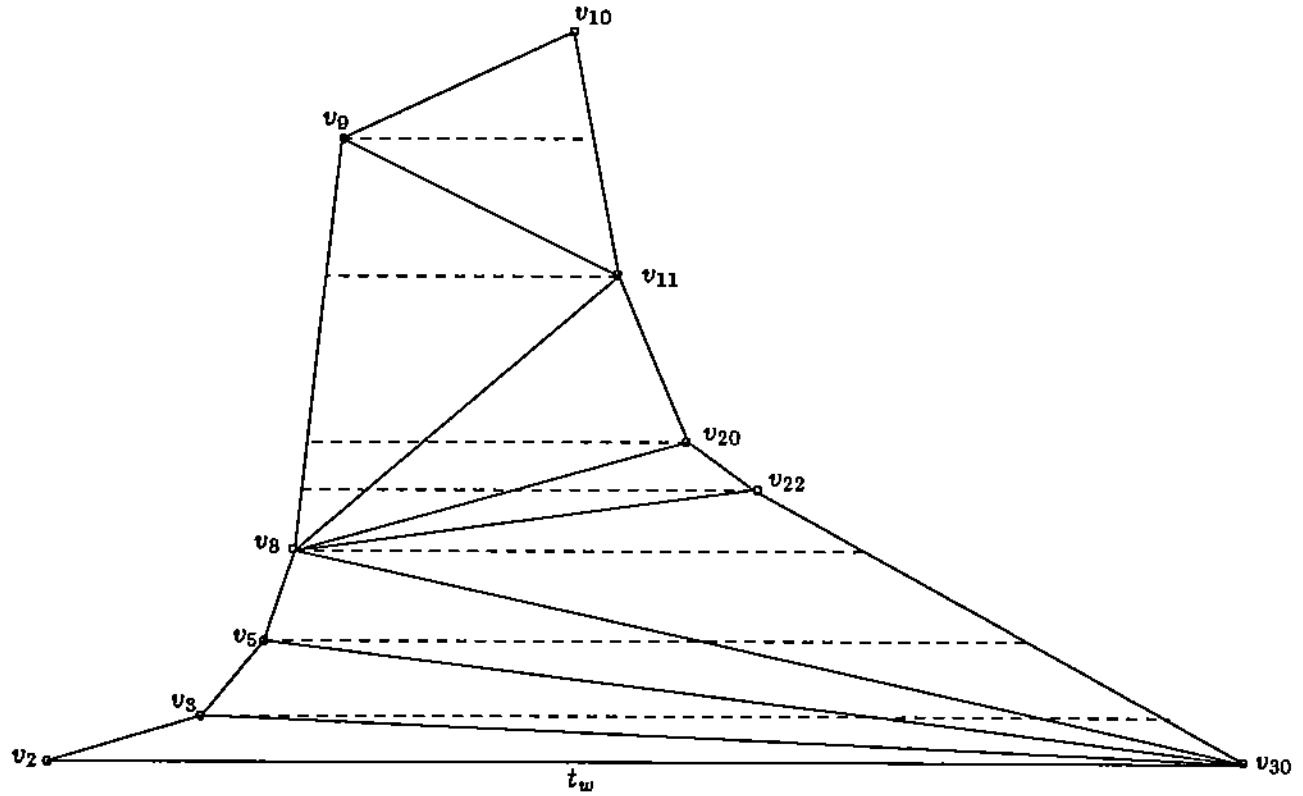


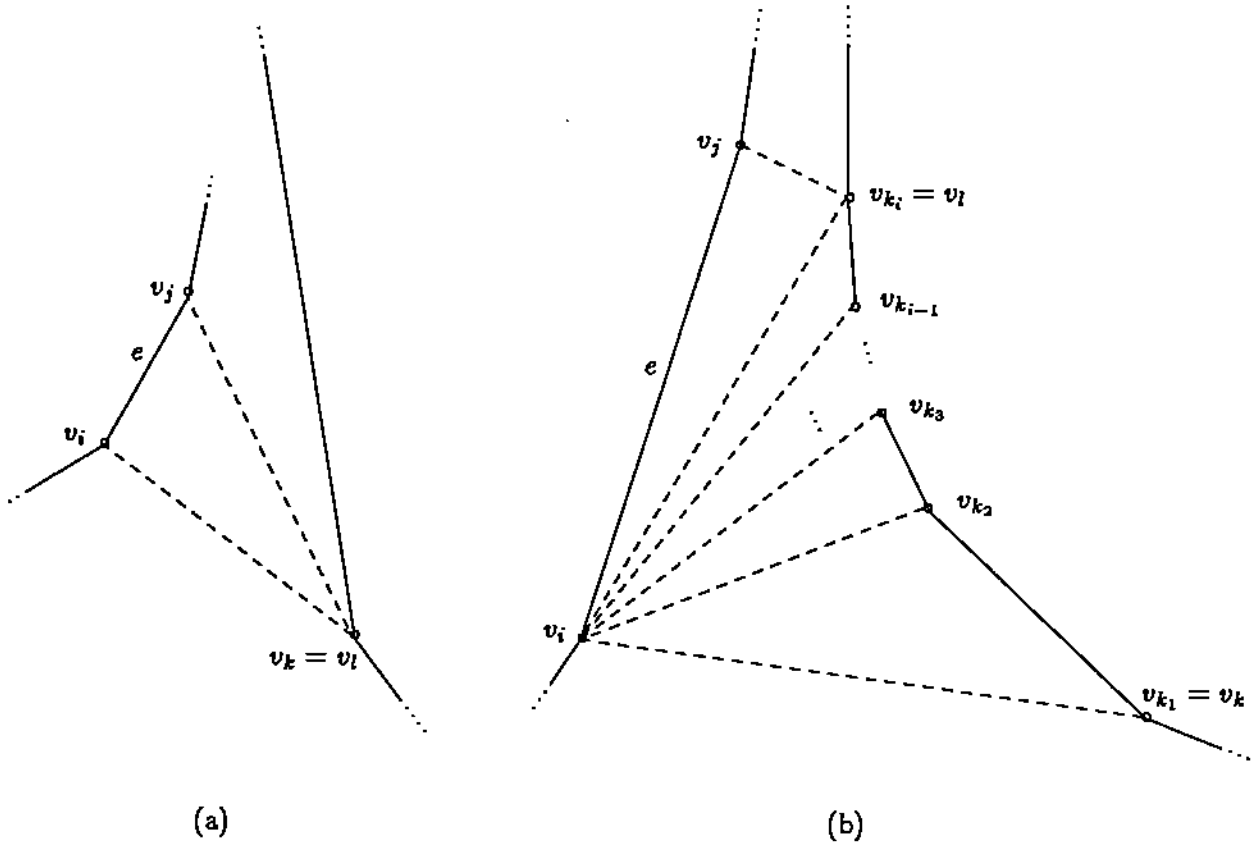
Figure 9: An example of a triangulated polygon P_w. The left convex chain is from $LH(Desc(lchild(w)))$ and the right convex chain is from $LH(Desc(rchild(w)))$.

add an edge from each vertex $v_{k_2}, \ldots, v_{k_i}$ to v_i. Therefore, the portion of P_w between v_iv_k and v_jv_l consists of the triangle $v_iv_jv_l$ and a series of triangles $v_iv_{m+1}v_m$, for $m \in \{k_1, \ldots, k_{i-1}\}$ (See Figure 10.b). This completes the proof of the correctness of CHAIN-TRIANGULATE. We now move on to the complexity bounds of the algorithm.

Summing up all the time complexity bounds from each of the above steps, we get that the time complexity, $T(n)$, of CHAIN-TRIANGULATE is characterized by the recurrence $T(n) = T(\sqrt{n}) + O(\log n)$, which has solution $T(n) = O(\log n)$. The space bounds, $S(n)$, can be characterized by the recurrence $S(n) = \max\{\sqrt{n}P(\sqrt{n}), bn\}$, for some constant b. Thus, $S(n) = O(n)$. The processor bounds, $P(n)$, can be characterized by the recurrence $P(n) = \max\{\sqrt{n}P(\sqrt{n}), cn\}$, for some constant c, which implies that $P(n) = O(n)$. ∎

Corollary 3.10: *Given a simple polygon P with n vertices, P can be triangulated in $O(\log n \log\log n)$ time and $O(n \log n)$ space (or, alternatively, in $O(\log^2 n)$ time and $O(n)$ space) using $O(n)$ processors on a CREW PRAM.*

Proof: Since we can triangulate the underside of a monotone polygonal chain in $O(\log n)$ time and $O(n)$ space using $O(n)$ processors, by Theorem 3.9, the bottle-neck computation in triangulating a



(a) (b)

Figure 10: The two cases for proving that the portion of P_w between v_iv_k and v_jv_l is triangulated.

simple polygon is in computing the trapezoidal decomposition used in partitioning P into one-sided monotone polygons. The complexity bounds follow, then, from Theorems 3.7 and 3.8. ■

We also note, in the corollary which follows, that that the algorithm CHAIN-TRIANGULATE can also be used in triangulating an arbitrary point set in $O(\log n)$ time and $O(n)$ space using $O(n)$ processors, which is optimal.

Corollary 3.11: *Given a set S of n points in the plane a triangulation of S can be computed in $O(\log n)$ time using $O(n)$ processors on a CREW PRAM, and this is optimal.*

Proof: The method is to first sort the points of S by x-coordinate. This can be done in $O(\log n)$ time using $O(n)$ processors by using the sorting network of Ajtai, Komlós, and Szemerédi (1983). After this is done, we can easily form a monotone polygonal chain V with each of the input points as vertices by joining consecutive vertices in the sorted order with an edge. We then use the algorithm CHAIN-TRIANGULATE twice: once to triangulate the region between V and its lower hull, and once to triangulate the region between V and its upper hull. By Theorem 3.9 we can see that the sorting step clearly dominates the complexity bounds, so we can triangulate an arbitrary point set in $O(\log n)$ time using $O(n)$ processors on a CREW PRAM. Since this problem has an $\Omega(n \log n)$

sequential lower bound (by a trivial reduction from sorting) we can do no better than $O(\log n)$ time using $O(n)$ processors on a CREW PRAM. ■

Merks (1986) recently established the above corollary using a substantially different method than the one we describe above. His method is also based on the $\sqrt{n}$-divide-and-conquer method, but it does not generalize to the problem of triangulating a simple polygon.

We next point out that the plane-sweep tree technique can be used to efficiently solve the planar point location problem.

3.5 Planar Point Location

Given a planar subdivision S consisting of n edges, construct a data structure which, once constructed, enables one processor to determine for a query point p the face in S containing p.

Theorem 3.12: *Given a planar subdivision S consisting of n edges, we can construct in parallel a data structure which, once constructed, enables one processor to determine for any query point p the face in S containing p in $O(\log n)$ time. The construction takes $O(\log n \log\log n)$ time and $O(n \log n)$ space using $O(n)$ processors on a CREW PRAM.*

Proof: The solution to this problem is to build the augmented plane-sweep tree for S and associate with each edge s_i the name of the face above and below s_i. A planar point location query can then be solved in $O(\log n)$ serial time by performing a multilocate like that used in the proof to Theorem 3.7. ■

It may seem that the usefulness of planar point location data structures is limited to sequential computational geometry, but this is not the case. For example, doing many planar point locations in parallel is the bottleneck computation in the parallel Voronoi diagram algorithm of Aggarwal *et al.* (1985). An easy consequence of Theorem 3.12 is that the running time of their algorithm can be improved from $O(\log^3 n)$ to $O(\log^2 n \log\log n)$, still using only $O(n)$ processors.

In the previous algorithms we assumed that segments did not intersect. In the next subsection we show that we can use the plane-sweep tree technique to detect if any two of n line segments intersect.

3.6 Intersection Detection

Given a set S of n line segments in the plane, determine if any two segments in S intersect. We begin by stating the conditions which we use to test for an intersection.

Lemma 3.13: *(Aggarwal et al., 1985) The segments in S are non-intersecting iff we have the following for the plane-sweep tree T of S:*

(1) For every $v \in T$ all the segments in $H(v)$ intersect the left vertical boundary of Π_v in the same order as they intersect Π_v's right vertical boundary.

(2) For every $v \in T$ no segment in $W(v)$ intersects any segment in $H(v)$. ∎

We use this lemma by testing for each condition at the appropriate point during the construction or traversal of the plane-sweep tree for S. We use these observations in the proof of the following theorem. We note that one result in the theorem is stated for the CRCW PRAM parallel model in which we allow for concurrent writes so long as all processors attempting to simultaneously write in the same memory cell are writing the same value. This is the only point in this paper in which we use the CRCW model; all other algorithms are for the (weaker) CREW PRAM model.

Theorem 3.14: *Given n line segments in the plane it is possible to detect if any two intersect in $O(\log n \log\log n)$ time and $O(n \log n)$ space using $O(n)$ processors on a CRCW PRAM (alternatively, in $O(\log^2 n)$ time and $O(n)$ space using $O(n)$ processors on a CREW PRAM).*

Proof: We begin with the proof of the $O(\log n \log\log n)$ time result. We can test for Condition (1) during the BUILDUP procedure. After building a set $H(v)$ in Step 2 of the BUILDUP procedure we can test Condition (1) by constructing two other sets $LB(v)$ and $RB(v)$, where $LB(v)$ ($RB(v)$) is the list of the intersection points of the segments in $H(v)$ with the left (right) vertical boundary of Π_v, listed in the same order as they appear in $H(v)$. If either of these lists is out of order, then there is an intersection. We can test whether either is out of order by comparing each element in $LB(v)$ (and $RB(v)$) with its two neighbors. If a processor detects an inconsistency then it writes a 1 to a global "intersection detected" flag (this is where we need the power of the CRCW model). Only if this flag is 0 do we proceed to the next level in T and repeat the above test. This will multiply the amount of work done by the BUILDUP algorithm by a factor of $O(1)$, so by Theorem 3.5 we can check Condition (1) in $O(\log n \log\log n)$ time and $O(n \log n)$ space using $O(n)$ processors.

If we complete the BUILDUP procedure and do not detect an intersection, then we can test for Condition (2) as follows. First, we execute the AUGMENT algorithm on T. Let $V = \{p_1, p_2, \ldots, p_{2n}\}$ be the set of endpoints of segments in S, and let $s(p_i)$ denote the segment in S with endpoint p_i. If $p_i \in \Pi_v$ for some $v \in T$, then clearly $s(p_i) \in W(v)$. If a segment $s(p_i) \in W(v)$ intersects a segment in $H(v)$, then it must intersect the segment in $H(v)$ directly below p_i or the segment in $H(v)$ directly above p_i (this is because we already know that no two segments of $H(v)$ intersect each other). We can then perform a multilocation of each p_i, and each time we find a segment in $H(v)$ directly above or below p_i we check if $s(p_i)$ intersects it. Thus, we can test Condition (2) in $O(\log n)$ additional time.

To prove the $O(n)$ space result, we use the alternative method of Theorem 3.6 to perform the necessary multilocations. We test Condition (1) each time a set $H(v)$ is constructed, $v \in T$. We

also test Condition (2) at this point, after performing the binary search in $H(v)$ for each point p_i such that $p_i \in \Pi_v$. The CRCW power is not needed in this case, because we are spending $O(\log n)$ time per level of the tree anyway, so we can afford to spend an additional $O(\log n)$ time performing an OR operation for each level of the tree. ■

We now move on to the critical-point merging technique and how to use it in conjunction with parallel divide-and-conquer to efficiently solve problems whose efficient sequential algorithms use the plane-sweeping technique.

4 Divide-and-Conquer with Critical-Point Merging

Often times when using the plane-sweeping paradigm to solve geometric problems sequentially, one scans a set of objects by sliding a vertical line along the x-axis, storing the objects in some kind of binary search tree as one goes. At various points (*critical* points) during the plane-sweeping one performs updates and queries on this tree. Intuitively, the method described in this section is to turn plane-sweeping on its side and use divide-and-conquer to compute all the critical-point queries. We begin by dividing the problem into two equally sized subproblems by splitting the set of objects as they would be split into subtrees in the binary search tree. After solving each subproblem in parallel we take the set of critical points for each subproblem and merge them into one list. The key to solving a problem in this manner is in defining labels to be associated with each critical point such that the labels of the merged list can be computed quickly in parallel, and, more importantly, such that when we have completed the construction we can use these labels to solve the problem at hand. Instead of describing the technique in a generic fashion, as we did with the plane-sweep tree, we describe it by presenting the solutions to four specific problems: 3-dimensional maxima, multiple range-counting, rectilinear segment intersection counting, and visibility from a point.

4.1 3-Dimensional Maxima

Let $V = \{p_1, p_2, \ldots, p_n\}$ be a set of points in $\Re^3$. For simplicity, we assume that no two input points have the same x (resp., y, z) coordinate. We say that a point p_i *1-dominates* another point p_j if $x(p_i) > x(p_j)$, *2-dominates* p_j if $x(p_i) > x(p_j)$ and $y(p_i) > y(p_j)$, and *3-dominates* p_j if $x(p_i) > x(p_j)$, $y(p_i) > y(p_j)$, and $z(p_i) > z(p_j)$. A point $p_i \in V$ is said to be a *maximum* if it is not 3-dominated by any other point in V. The 3-dimensional maxima problem, then, is to compute the set, M, of maxima in V. We show how to solve the 3-dimensional maxima problem efficiently in parallel in the following algorithm. The labels we use are motivated by the labels used in the binary search tree used in the plane sweep used in the optimal sequential algorithm for this problem (Kung, Luccio, and Preparata, 1975). We assume that the input points are given in sorted order,

since sorting can be done in $O(\log n \log\log n))$ time and $O(n)$ space using $O(n)$ processors by using parallel merging.

Algorithm 3-D MAXIMA:

Input: A list of points $V = \{p_1, p_2, \ldots, p_n\}$ in $\Re^3$. We assume that the p_i's are given sorted by y-coordinate (i.e., $y(p_i) < y(p_{i+1})$).

Output: A list $X = \{q_1, q_2, \ldots, q_n\}$ of the points in V sorted by increasing x-coordinate. We also have two labels ZO and ZT associated with each $q_i \in X$, such that $ZO(q_i)$ is the largest z-coordinate in the set of points which 1-dominate q_i, and $ZT(q_i)$ is the largest z-coordinate in the set of points which 2-dominate q_i.

Step 1. Divide V into two equally sized subsets V_1 and V_2 such that all the points in V_1 have smaller y-coordinate than points in V_2. Recursively solve the problem for V_1 and V_2 in parallel. After the parallel recursive call returns we will have lists X_1 and X_2 of the points in V_1 and V_2, respectively, sorted by increasing x-coordinate. We also have labels ZO_1 (ZO_2) and ZT_1 (ZT_2) defined correctly for the points in X_1 (X_2) (when dominance is restricted to X_1 (X_2)).

Step 2. Merge X_1 and X_2 into a single list X, basing all comparisons on the x-coordinates of the points involved. Let $X = \{q_1, q_2, \ldots, q_n\}$ (X is the set of points in V listed by increasing x-coordinate).

Step 3. **For each** $q_i \in X$ **pardo**

{ Let $pred_j(q_i)$ denote the predecessor of q_i in X_j, $j \in \{1, 2\}$.
If q_i has no predecessor in X_j, then $pred_j(q_i) = \phi$. Also, let
$first(X_j)$ denote the first element in X_j. }

$ZO_1(\phi) := ZT_1(\phi) := \max\{ZO_1(first(X_1)), z(first(X_1))\}$;

{ this is the maximum z-coordinate in X_1 }

$ZO_2(\phi) := ZT_2(\phi) := \max\{ZO_2(first(X_2)), z(first(X_2))\}$;

If $q_i \in X_1$ **then** { q_i "came from" X_1 }

$ZO(q_i) := \max\{ZO_1(q_i), ZO_2(pred_2(q_i))\}$;

$ZT(q_i) := \max\{ZT_1(q_i), ZO_2(pred_2(q_i))\}$

Else { q_i "came from" X_2 }

$ZO(q_i) := \max\{ZO_1(pred_1(q_i)), ZO_2(q_i)\}$;

$ZT(q_i) := ZT_2(q_i)$

EndIf

EndFor

Step 4. (Postprocessing) After we have computed the labels ZO and ZT for all points q_i, we know that q_i is a maximum iff $z(q_i) > ZT(q_i)$.

End of Algorithm 3-D MAXIMA.

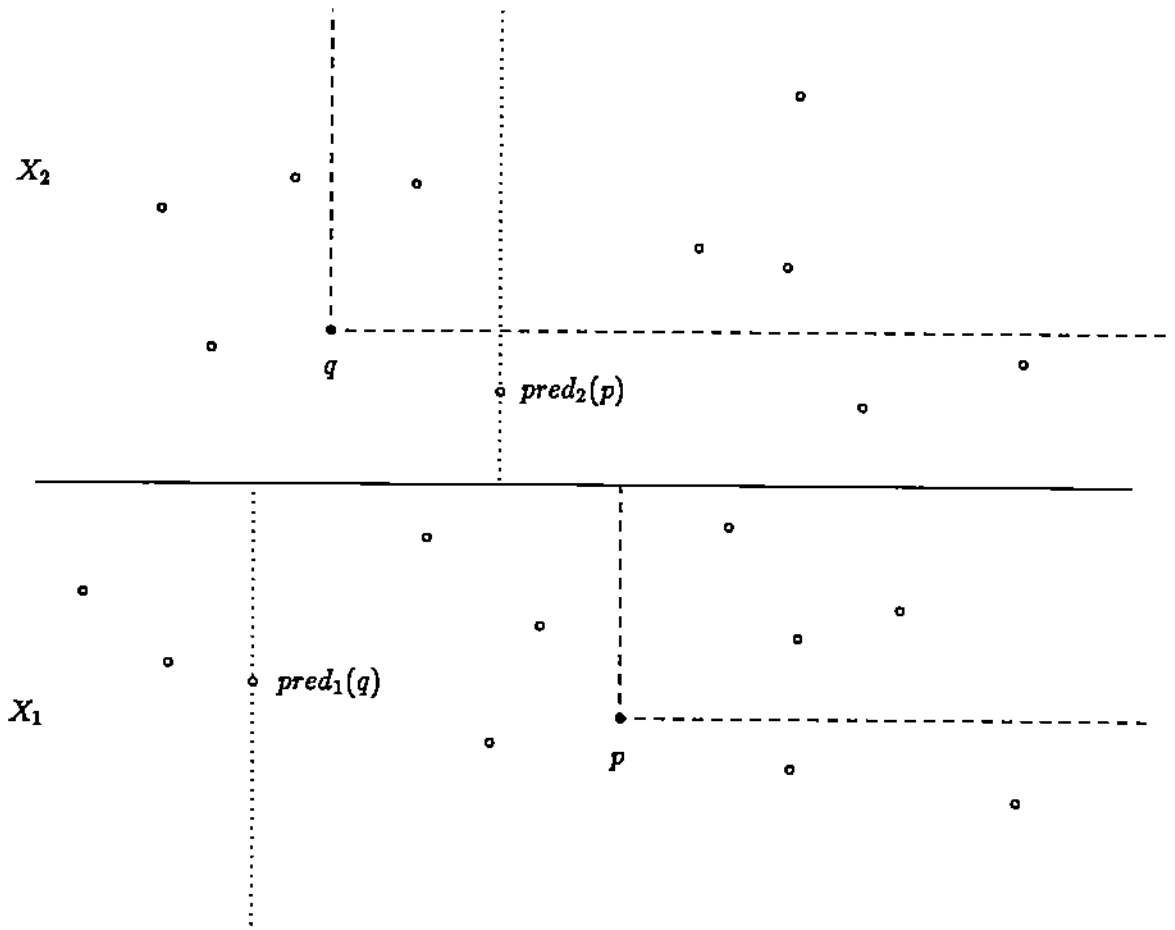


Figure 11: The different predecessor cases for 3-dimensional maxima. The figure is a projection of the points of V onto the xy-plane (x being the horizontal axis). Points enclosed in the dashed lines are the points which affect p (or q's) ZT label, while points to the right of the dotted lines affect p (or q's) ZO label.

Theorem 4.1: *The algorithm 3-D MAXIMA solves the three-dimensional maxima problem in $O(\log n \log\log n)$ time and $O(n)$ space using $O(n)$ processors on a CREW PRAM.*

Proof: We prove the correctness of the algorithm 3-D MAXIMA by induction. Suppose the ZO and ZT labels for each $q_i \in X_j$, $j \in \{1,2\}$, are computed correctly. Case 1: $q_i \in X$ comes from X_1 (i.e., $q_i \in X_1$). Notice that every point which 1-dominates q_i's predecessor in X_2 also 1-dominates q_i, since q_i's predecessor in X_2 is the point with largest x-coordinate less than $x(q_i)$. Also, since every point in X_2 has y-coordinate greater than $y(q_i)$, in this case, every point which 1-dominates q_i's predecessor will in fact 2-dominate q_i. Thus, in constructing $ZO(q_i)$ we need only take the maximum of the old ZO label for q_i and the ZO label for the predecessor of q_i in X_2, and q_i's new ZT label should be the maximum of its old ZT label and the ZO label of its predecessor in X_2. Case 2: $q_i \in X$ comes from X_2. Clearly, in this case, no point in X_1 can 2-dominate q_i, so the new ZT label for q_i should be the same as the old ZT label of q_i. Still, any point which 1-dominates q_i's predecessor in X_1 also 1-dominates q_i, so in order to update the ZO label for q_i we still need to take the maximum of the old value of ZO and the ZO label of the predecessor of q_i in X_1. (See Figure 11.) Thus, after we have done the updates of Step 3, each q_i's ZT label stores the maximum z-coordinate of all the points which 2-dominate q_i. Therefore, using these ZT labels we can use the test of Step 4 (and a parallel prefix computation) to construct the set M of maxima in X (in $O(\log n)$ additional time).

Lemma 2.1 implies that the algorithm's time complexity, $T(n)$, is determined by the recurrence $T(n) = T(n/2) + O(\log\log n)$, whose solution is $T(n) = O(\log n \log\log n)$. The space and number of processors used are clearly $O(n)$. ■

It is worth noting that we can use the method of algorithm 3-D MAXIMA as the basis step of a recursive procedure for solving the general k-dimensional maxima problem. The resulting time and space complexities are given in the following theorem. We state the theorem for $k \geq 3$ (the 2-dimensional maxima problem can easily be solved in $O(\log n)$ time and $O(n)$ space using the AKS sorting network and a parallel prefix computation.

Theorem 4.2: *For $k \geq 3$ the k-dimensional maxima problem can be solved in $O((\log n)^{k-2} \log\log n)$ time and $O(n)$ space using $O(n)$ processors on a CREW PRAM.*

Proof: The method is a straightforward parallelization of the algorithm by Kung, Luccio, and Preparata (1975), using a procedure very similar to 3-D MAXIMA as the basis for the recursion. We omit the details. ■

Next, we address the two-set dominance counting problem. We also show how the multiple range-counting problem and the rectilinear segment intersection counting problem can be reduced to two-set dominance problems efficiently in parallel.

4.2 Two-Set Dominance Counting and Related Problems

In the two-set dominance counting problem we are given a set $V = \{p_1, p_2, \ldots, p_l\}$ and a set $U = \{q_1, q_2, \ldots, q_m\}$ of points in the plane, and wish to know for each point q_i in U the number of points in V which are 2-dominated by q. In the algorithm which follows we show how to solve this problem efficiently in parallel.

Algorithm DOM-COUNT:

Input: A set $V = \{p_1, p_2, \ldots, p_l\}$ and a set $U = \{q_1, q_2, \ldots, q_m\}$ of points in the plane. For simplicity, we assume that the points in V and U are all distinct.

Output: A list $X = \{v_1, v_2, \ldots, v_{l+m}\}$ of the points defining this problem (v_i is either a p_j or a q_j) sorted by increasing lexicographical order. We also have labels CO and CT defined for each $v_i \in X$, where $CO(v_i)$ is the number of points in V 1-dominated by the point v_i, and $CT(v_i)$ is the number of points in V 2-dominated by v_i.

Step 0. (Preprocessing) Combine the points in V and U into one list W, and sort the points in W by y-coordinate. Also, we mark each point in W if it came from V. Initially, the CO and CT label for each point is 0.

Comment: For each $v_i \in W$ define the function X_V as follows: $X_V(v_i) = 1$ if $v_i \in V$; $X_V(v_i) = 0$ otherwise.

Step 1. Divide W into two equally sized subsets W_1 and W_2 such that all the points in W_1 have smaller y-coordinate than points in W_2. Recursively solve the problem for W_1 and W_2 in parallel. After the parallel recursive call returns we will have lists X_1 and X_2 of the points in W_1 and W_2, respectively, sorted by increasing lexicographical order. We also have labels CO_1 (CO_2) and CT_1 (CT_2) defined correctly for the points in X_1 (X_2) (when dominance is restricted to X_1 (X_2)).

Step 2. Merge X_1 and X_2 into a single list X, where all comparisons are done lexicographically. Let $X = \{v_1, v_2, \ldots, v_{n+m}\}$.

Step 3. For each $v_i \in X$ pardo

{ Let $pred_j(v_i)$ denote the predecessor of v_i in X_j. If v_i has no predecessor in X_j, $j = 1, 2$, then $pred_j(v_i) = \phi$. }

$CO_1(\phi) := CO_2(\phi) := CT_1(\phi) := CT_2(\phi) := X_V(\phi) := 0$;

If $v_i \in X_1$ **then** { v_i "came from" X_1 }

$CO(v_i) := CO_1(v_i) + CO_2(pred_2(v_i)) + X_V(pred_2(v_i))$;

$CT(v_i) := CT_1(v_i)$

Else { v_i "came from" X_2 }

$CO(v_i) := CO_1(pred_1(v_i)) + CO_2(v_i) + X_V(pred_1(v_i))$;

$CT(v_i) := CT_1(pred_1(v_i)) + CT_2(v_i) + X_V(pred_1(v_i))$

EndIf

EndFor

End of Algorithm DOM-COUNT.

Theorem 4.3: *Given a set V of l points in the plane and a set Q of m points in the plane, the algorithm DOM-COUNT computes for each $q_i \in Q$ the number of points in V 2-dominated by q_i in $O(\log n \log\log n)$ time and $O(n)$ space using $O(n)$ processors on a CREW PRAM, where $n = l+m$.*

Proof: The proof of correctness is by induction. For any point $v_i \in X$ the number of points 1-dominated by v_i, $CO(v_i)$, is equal to the old CO label for v_i plus the CO label for the predecessor of v_i plus 1 if the predecessor of v_i is in V, since the predecessor of v_i is 1-dominated by v_i. If v_i came from X_1 then the number of points 2-dominated by v_i, $CT(v_i)$ is simply the old CT label for v_i, since v_i cannot 2-dominate any points in X_2. If v_i came from X_2 then the number of points 2-dominated by v_i is the old CT label for v_i plus the CT value for the predecessor of v_i plus 1 if v_i is in V, since v_i 2-dominates its predecessor in this case. (See Figure 12). By an argument similar to the one used in the proof of Theorem 4.1 the algorithm DOM-COUNT runs in $O(\log n \log\log n)$ time and $O(n)$ space using $O(n)$ processors, where $n = l + m$. ■

There are a number of other problems which can be reduced to two-set dominance counting. We mention two here. We begin with the multiple range-counting problem. Given a set V of l points in the plane and a set R of m isothetic rectangles (ranges) the multiple range-counting problem is to compute the number of points interior to each rectangle.

Corollary 4.4: *Given a set V of l points in the plane and a set R of m isothetic rectangles, we can solve the multiple range-counting problem for V and R in $O(\log n \log\log n)$ time and $O(n)$ space using $O(n)$ processors, where $n = l + m$.*

Proof: We know from (Edelsbrunner and Overmars, 1982) that counting the number of points interior to a rectangle can be reduced to dominance counting. That is, if $d(p)$ is the number of

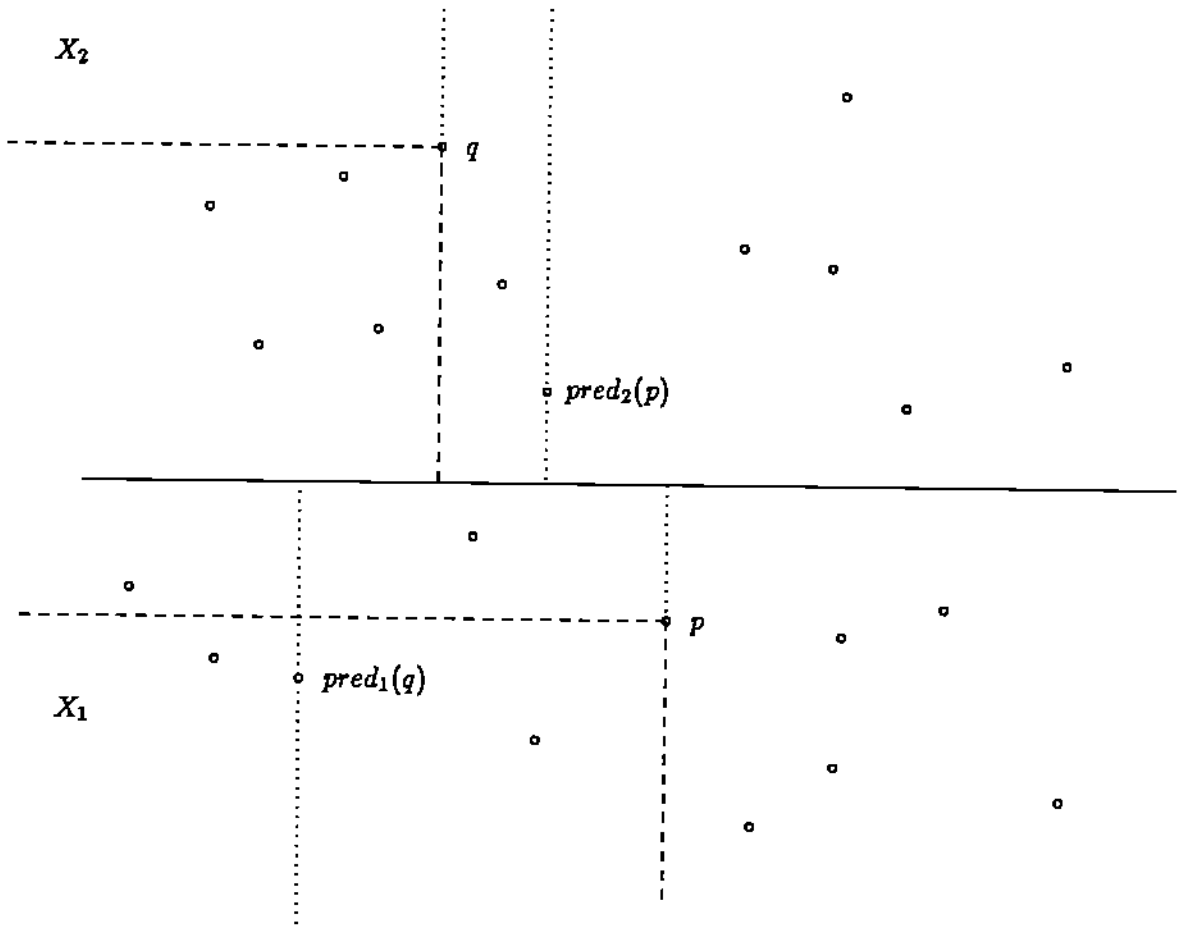


Figure 12: The different predecessor cases for 2-set dominance counting. Points enclosed in the dashed lines are the points which affect p (or q's) CT label, while points to the left of the dotted lines affect p (or q's) CO label.

points in V 2-dominated by a point p, given a rectangle $r = (p_1, p_2, p_3, p_4)$ (where vertices are listed in counter-clockwise order starting with the upper-righthand corner), then the number of points in V interior to r is $d(p_1) - d(p_2) + d(p_3) - d(p_4)$. Therefore, it suffices to solve the two-set dominance counting problem. ■

Another problem which reduces to two-set dominance counting is rectilinear segment intersection counting: given a set S of n rectilinear line segments in the plane, determine for each segment the number of other segments in S which intersect it.

Corollary 4.5: *Given a set S of n rectilinear line segments in the plane, we can determine for each segment the number of other segments in S which intersect it in $O(\log n \log\log n)$ time and $O(n)$ space using $O(n)$ processors on a CREW PRAM.*

Proof: Let U_1 (U_2) be the set of left (right) endpoints of horizontal segments, and let $d_1(p)$ ($d_2(p)$) denote the number of points in U_1 (U_2) 2-dominated by p. For any vertical segment s, with upper endpoint p and lower endpoint q, the number of horizontal segments which intersect s is $d_1(p) - d_1(q) + d_2(q) - d_2(p)$. This is because $d_1(p) - d_1(q)$ ($d_2(p) - d_2(q)$) counts the number of horizontal segments with a left (right) endpoint to the left of s and y-coordinate in the interval $[y(q), y(p)]$. Thus, $d_1(p) - d_1(q) - (d_2(p) - d_2(q))$ counts the number of horizontal segments with left endpoint to the left of s, right endpoint to the right of s, and y-coordinate in the interval $[y(q), y(p)]$ (i.e., the set of horizontal segments which intersect s). ■

The final problem we look at is visibility from a point.

4.3 Visibility from a Point

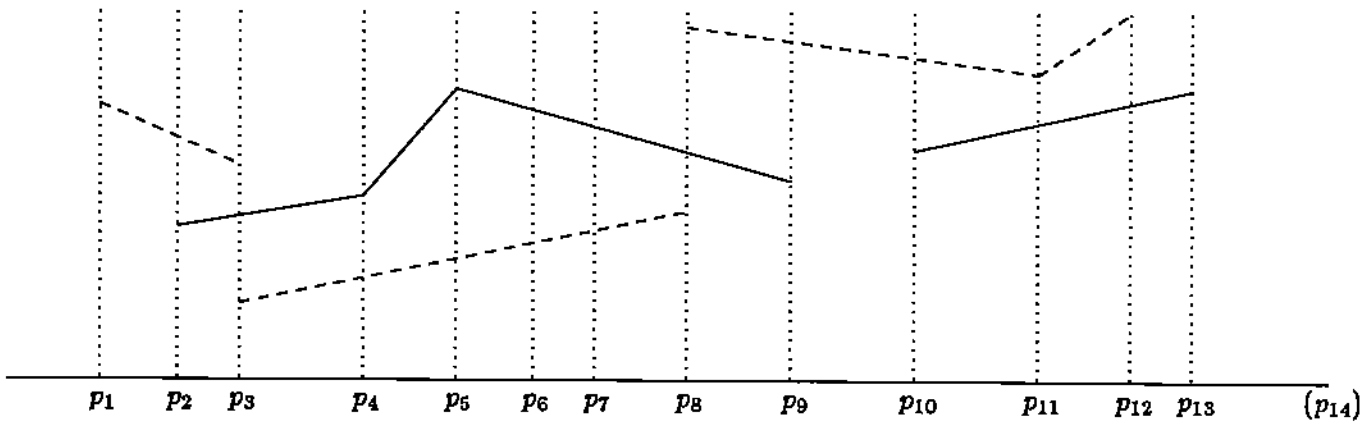
Given a set of line segments $S = \{s_1, s_2, \ldots, s_n\}$ which do not intersect, except possibly at endpoints, and a point p, determine the part of the plane which is visible from p when every s_i is opaque. We can use divide-and-conquer with critical-point merging to solve this problem in $O(\log n \log\log n)$ time and $O(n)$ space using $O(n)$ processors. WLOG, the point p is at negative infinity below all the segments. For simplicity, we assume that the x-coordinates of the endpoints are distinct.

Algorithm VISIBILITY:

Input: A set of non-intersecting line segments $S = \{s_1, s_2, \ldots, s_n\}$. The s_i's are not given in any particular order.

Output: A set $X = \{p_1, p_2, \ldots, p_{2n}\}$ consisting of the endpoints of the segments in S sorted by x-coordinates ($x(p_i) < x(p_{i+1})$). We also have a label VIS associated with each $p_i \in X$, such that $VIS(p_i)$ is the segment in S visible on the interval $(x(p_i), x(p_{i+1}))$, for $i = 1, 2, \ldots, 2n-1$, and $VIS(p_{2n}) = +\infty$; by convention, $VIS(p_i) = +\infty$ if no segment is visible on the interval $(x(p_i), x(p_{i+1}))$.

Before merge:



$X_1 =$ | 1 | 3 | 6 | 7 | 8 | 11 | 12 |

$X_2 =$ | 2 | 4 | 5 | 9 | 10 | 13 |

$pred_2 =$ | ϕ | 2 | 5 | 5 | 5 | 10 | 10 |

$pred_1 =$ | 1 | 3 | 3 | 8 | 8 | 12 |

After merge:

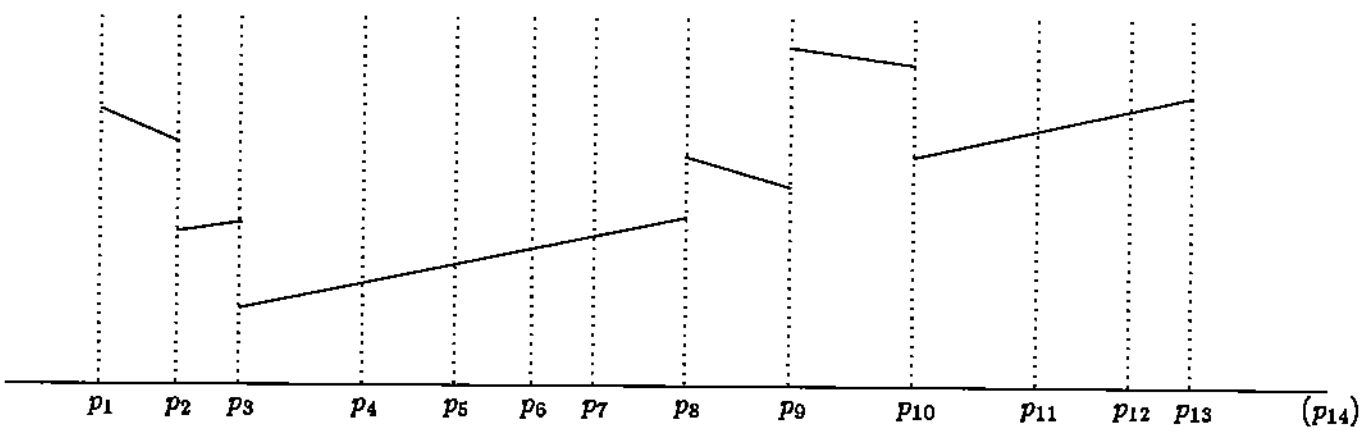


Figure 13: An example of visibility merging. The dashed segments correspond to the visible region for X_1 and the solid segments correspond to the visible region for X_2. For simplicity in describing the vectors X_1, X_2, $pred_2$, and $pred_1$ we denote each point p_i by its index i. Note that points are never removed, even if the same segment defines the visible region for many consecutive intervals (e.g., p_3 through p_7).

Step 1. Partition S into subsets $S_1 = \{s_1, \ldots, s_{n/2}\}$ and $S_2 = \{s_{n/2+1}, \ldots, s_n\}$, and recursively solve the problem for S_1 and S_2 in parallel. After the parallel recursive call returns we will have a list X_1 of the endpoints of segments in S_1 sorted by x-coordinates, and a similarly defined list X_2 for S_2. We also have labels VIS_1 (VIS_2) labels correctly defined for each point in X_1 (X_2) when visibility is restricted to segments in S_1 (S_2).

Step 2. Use parallel merging to merge the two sorted lists X_1 and X_2 into a single list X, where comparisons are based on the x-coordinates of points. Let $X = \{p_1, p_2, \ldots, p_{2n}\}$.

Step 3. **For each** $p_i \in X$ **pardo**

{ Let $pred_j(p_i)$ denote the predecessor of p_i in X_j, $j \in \{1, 2\}$.
If p_i has no predecessor in X_j, then $pred_j(p_i) = \phi$. }
$VIS_1(\phi) := VIS_2(\phi) := +\infty$;
If $p_i \in X_1$ **then** { p_i "came from" X_1 }
$VIS(p_i) := \min\{VIS_1(p_i), VIS_2(pred_2(p_i))\}$
Else { p_i "came from" X_2 }
$VIS(p_i) := \min\{VIS_1(pred_1(p_i)), VIS_2(p_i)\}$
EndIf

EndFor

Comment: Taking the minimum of $VIS_1(p_i)$ and $VIS_2(pred_2(p_i))$ (or of $VIS_1(pred_1(p_i))$ and $VIS_2(p_i)$) is well defined, since the segments being compared span the interval $(x(p_i), x(p_{i+1}))$ and do not intersect (see Figure 13). Having observed this, note that Step 3 completes the construction, since the list of labels $VIS(p_i)$ is a description of the visible part of the plane.

End of Algorithm VISIBILITY.

Theorem 4.6: *The algorithm VISIBILITY solves the problem of computing the visibility from a point in $O(\log n \log\log n)$ time and $O(n)$ space using $O(n)$ processors on a CREW PRAM.*

Proof: The correctness of VISIBILITY follows from the observation that in the conquer step (3) when computing $VIS(p_i)$ we need only compare the two segments which span the vertical strip $(x(p_i), x(p_{i+1})) \times (-\infty, \infty)$. This is precisely what is happening in Step 3 when we compare the old VIS label of a critical point with the VIS label of its predecessor in the other set.

By the same argument as in the proof for Theorem 4.1 the algorithm VISIBILITY runs in $O(\log n \log\log n)$ time and $O(n)$ space using $O(n)$ processors. ∎

5 Conclusion

In this paper we have given general techniques for solving a number of geometric problems whose efficient sequential algorithms use the plane-sweep paradigm. These techniques can be viewed

as efficient parallel analogues to the plane-sweeping paradigm. We applied the plane-sweep tree technique to intersection detection, trapezoidal decomposition, polygon triangulation, and planar point location. We were able to achieve an $O(\log n \log\log n)$ time bound for each problem, using $O(n)$ processors. For the problem of triangulating an arbitrary point set we were able to achieve an $O(\log n)$ time bound using $O(n)$ processors, which is optimal. We were able to achieve a faster running time for arbitrary triangulation than for polygon triangulation because we could avoid the bottle-neck computation in our polygon triangulation algorithm, that is, constructing a trapezoidal decomposition (which is not even defined for an arbitrary point set). A consequence of our planar point location algorithm is that the time bound of the Voronoi diagram construction algorithm of Aggarwal *et al.* (1985) can be improved from $O(\log^3 n)$ to $O(\log^2 n \log\log n)$, still using only $O(n)$ processors.

We applied divide-and-conquer with critical-point merging technique to visibility from a point, 3-dimensional maxima, two-set dominance counting, multiple range-counting, and rectilinear segment intersection counting. We were able to achieve an $O(\log n \log\log n)$ time bound for each problem, using $O(n)$ processors.

Acknowledgment

We would like to thank Greg Frederickson for his valuable comments that considerably improved the presentation of Section 4.

References

Aggarwal, A., Chazelle, B., Guibas, L., Ó'Dúnlaing, C., and Yap, C. (1985), "Parallel Computational Geometry," *Proc. 25th IEEE Symp. Found. of Comp. Sci.*, pp. 468–477.

Ajtai, M., Komlós, J., and Szemerédi, E. (1983), "Sorting in $c \log n$ parallel steps," *Combinatorica*, 3, pp. 1–19.

Atallah, M. J., and Goodrich, M. T. (1985), "Efficient Parallel Solutions to Some Geometric Problems," to appear in *Jour. of Parallel and Dist. Comp.* A preliminary version appeared in *Proc. 1985 IEEE Int. Conf. on Parallel Proc.*, pp. 411–417.

Atallah, M. J., and Goodrich, M. T. (1986), "Parallel Algorithms for Some Functions of Two Convex Polygons," to appear in 24th Allerton Conference on Communication, Control, and Computing.

Borodin, A., and Hopcroft, J. (1985), "Routing, Merging, and Sorting on Parallel Models of Computation," *Jour. of Comp. and Sys. Sci.*, **30**(1), pp. 130–145.

Chazelle, B., and Guibas, L. J. (1986), "Fractional Cascading: I. A Data Structuring Technique," *Algorithmica*, **1**(2), pp. 133–162.

Chow, A. (1980), "Parallel Algorithms for Geometric Problems," Ph.D. dissertation, Comp. Sci. Dept., Univ. of Illinois at Urbana-Champaign.

Edelsbrunner, H., and Overmars, M. H. (1982), "On the Equivalence of Some Rectangle Problems," *Info. Proc. Letters*, 14(3), pp. 124–127.

El Gindy, H. (1986), "A Parallel Algorithm for Triangulating Simplical Point Sets in Space with Optimal Speed-up," to appear in 24th Allerton Conference on Communication, Control, and Computing.

Kruskal, C., Rudolph, L., and Snir, M. (1985), "The Power of Parallel Prefix," *Proc. 1985 IEEE Int. Conf. on Parallel Proc.*, pp. 180–185.

Kung, H. T., Luccio, F., Preparata, F. P. (1975), "On Finding the Maxima of a Set of Vectors," *Jour. of ACM*, 22(4), pp. 469–476.

Lee, D. T., and Preparata, F. P. (1984), "Computational Geometry—A Survey," *IEEE Trans. on Computers*, C-33(12), pp. 872–1101.

Merks, E. (1986), "An Optimal Parallel Algorithm for Triangulating a Set of Points in the Plane," Tech. Report TR 86-9, Simon Fraser University, Burnaby, British Columbia, Canada.

Overmars, M. H., and Van Leeuwen, J. (1981), "Maintenance of Configurations in the Plane," *Jour. of Comp. and Sys. Sci.*, 23, pp. 166–204.

Valiant, L. (1975), "Parallelism in Comparison Problems," *SIAM Jour. on Comp.*, 4(3), pp. 348–355.

Problem	Previous Bounds	Our Bounds
Trapezoidal Decomposition	$(\log^2 n,\ n \log n)$	$(\log n \log\log n,\ n \log n)$ or $(\log^2 n,\ n)$
Polygon Triangulation	$(\log^2 n,\ n \log n)$	$(\log n \log\log n,\ n \log n)$ or $(\log^2 n,\ n)$
Arbitrary Triangulation	not considered	$(\log n,\ n)$
Planar Point Location	$(\log^2 n,\ n \log n)$ $Q(n) = O(\log^2 n)$	$(\log n \log\log n,\ n \log n)$ $Q(n) = O(\log n)$
Intersection Detection	$(\log^2 n,\ n \log n)$	$(\log^2 n,\ n)$
Int. Detection (CRCW model)	not considered	$(\log n \log\log n,\ n \log n)$
3-D Maxima	"	$(\log n \log\log n,\ n)$
Two-Set Dom. Counting	"	$(\log n \log\log n,\ n)$
Multiple Range-Counting	"	$(\log n \log\log n,\ n)$
Rect. Segment Int. Counting	"	$(\log n \log\log n,\ n)$
Visibility	"	$(\log n \log\log n,\ n)$

Table 1: **Summary of Results.** The pair $(t(n), s(n))$ denotes that the parallel algorithm runs in $O(t(n))$ time and $O(s(n))$ space, using $O(n)$ processors. All previous bounds are due to Aggarwal *et al.* (1985).