Efficient Plane Sweeping in Parallel *
(Preliminary Version)

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 visibility from a point, 3-dimensional maxima, multiple range-counting, and rectilinear segment intersection counting. We never use the AKS sorting network in any of our algorithms.

1 Introduction

The plane-sweeping technique has proven effective for developing efficient sequential algorithms for a variety of geometric problems. This technique, in 2-dimensions, involves sweeping a line through a set of geometric objects (such as line segments), updating global data structures at each critical point (such as an endpoint). It has been used to find efficient sequential algorithms for a host of computational geometry problems (see [16]). 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 [1,2,3,9,12], 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** [6]†: Given a simple n-vertex polygon P, determine the trapezoidal edge(s) for each 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.

*This work was supported by the National Science Foundation under Grant DCR-84-51393 and by the Office of Naval Research under Grant N00014-K-0502.

†See [16] for other references.

Problem	Previous Bounds	Our Bounds
Trapezoidal Decomposition	$(\log^2 n, n \log n)$ [1]	$(\log n \log\log n, n \log n)$ or $(\log^2 n, n)$
Triangulation	$(\log^2 n, n \log n)$ [1]	$(\log n \log\log n, n \log n)$ or $(\log^2 n, n)$
Planar Point Location	$(\log^2 n, n \log n)$ [1] $Q(n) = O(\log^2 n)$	$(\log n \log\log n, n \log n)$ $Q(n) = O(\log n)$
Intersection Detection	$(\log^2 n, n \log n)$ [1]	$(\log^2 n, n)$
Int. Detection (CRCW model)	not considered	$(\log n \log\log n, n \log n)$
Visibility	not considered	$(\log n \log\log n, n)$
3-D Maxima	"	$(\log n \log\log n, n)$
Multiple Range-Counting	"	$(\log n \log\log n, n)$
Rect. Segment Int. Counting	"	$(\log n \log\log n, n)$

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.

2. **Triangulation** [6]†: Given a simple n-vertex polygon P, augment P with diagonal edges so that each interior face is a triangle.

3. **Planar Point Location** [13]†: 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.

4. **Intersection Detection** [19]†: Given n line segments in the plane, determine if any two intersect.

5. **Visibility from a Point** [11]†: 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.

6. **3-Dimensional Maxima** [15]†: 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. **Multiple Range-Counting** [17]†: Given l points in the plane and m isothetic rectangles (ranges) determine the number of points interior to each rectangle. The problem size is $n = l + m$.

8. **Rectilinear Segment Intersection Counting** [16]: Given n horizontal and vertical line segments in the plane, determine for each segment the number of other segments which intersect it.

As in [1,2] our framework is one in which we have $O(n)$ processors with which we wish to achieve the best time and space performance possible. Unless otherwise stated, our algorithms will be for the CREW PRAM parallel model (as in [1,2]). Recall that this is the synchronous parallel model in which processors share a common memory where concurrent reads are allowed, but not concurrent writes.

In [1] Aggarwal et al. 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 in [1] include intersection detection, trapezoidal decomposition, 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 in [1], 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 [8]. 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 AKS sorting network [4], which introduces a large 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, visibility from a point, 3-dimensional maxima, multiple range-counting, and rectilinear segment intersection counting. 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, however, 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 selecting 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, we define the *cousin* of a in B to be the greatest element in B which is less than or equal to a. If there is no such b_j in B, then we say that the cousin 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 cousin in B of any such a. In the next lemma we show that if we have two sorted lists A and B whose elements are taken from the same total order, we can find the cousin 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 cousin 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 [20] (which is implementable in the CREW PRAM model [5]) first finds cousins and then does the merge. Thus, the lemma follows directly from the work of [5] and [20]. ∎

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 [14]. Stated in its simplest form, given an array 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 [14] for details). Parallel prefix is used as a building block 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, but essentially the same technique applies for other 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.

The idea of using a tree to parallelize plane-sweeping is due to [1] and is based on a data structure of [7]. We review some of the definitions and observations from [1] and [7] as it relates to our work. 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.

Lemma 3.1 [1]: *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 [1] and [7], we define $H(v)$ and $W(v)$ for each node $v \in T$ as follows:

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

However, here we also define two other sets. Let $left(\Pi_v)$ ($right(\Pi_v)$) denote the left (right) vertical boundary of Π_v.

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

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.

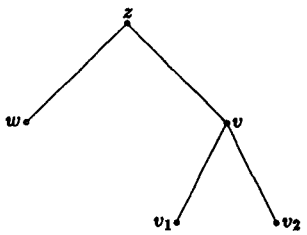

Figure 1: A configuration of nodes in T.

Lemma 3.2: *Let v be a node in T with children v_1 and v_2, sibling w, and parent z (Figure 1 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) $L(v) = H(v_1) + L(v_1)$;

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

(3) $H(v) = R(w) - (R(w) \cap L(v))$ if v is the right child of z (as is the case in Figure 1); $H(v) = L(w) - (L(w) \cap R(v))$, if v is the left child of z.

Proof: The proof is given in the technical report [3]. ∎

Lemma 3.2 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 3.2, 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 3.2 can both be performed in $O(\log \log n)$ time.

Lemma 3.3: *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 [5,20]. If $B \subseteq A$, we construct $A-B$ by first determining the cousin 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 cousin. 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 3.2 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 efficiently in parallel. From Lemma 3.3 we know that the constructions implicit in Equations (1) and (2) of Lemma 3.2 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 3.3. 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 $H(v)$.)

Step 1. Sort the set of endpoints of $s_1, \ldots, s_n$ by their x-coordinates, 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 [5,20] to sort in $O(\log n \log \log n)$ time using $O(n)$ processors, instead of using the AKS sorting network [4] 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 $s_i \subset \Pi_v$. 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 [14] 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 3.4: *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 3.2 is exactly the set of all segments with one endpoint in Π_w and the other in Π_v. Thus, by Observation 3.4, we can rewrite Equation (3) of Lemma 3.2 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. **For** l = lowest level **until** $l = 0$ **repeat** Steps 1–3 below, in parallel for each $v \in T$ at level l.

Step 1. Use equations (1) and (2) of Lemma 3.2 and Lemma 3.3 to build the sets $L(v)$ and $R(v)$ from the sets for v's children.

Step 2. Use the modified equation (3) of Lemma 3.2 (that is, $H(v) = R(w) - I(z)$ if v is a right child, and $H(v) = L(w) - I(z)$, otherwise) and Lemma 3.3 to build $H(v)$ from $I(z)$ (which was precomputed) and the appropriate $R(w)$ or $L(w)$ constructed in Step 1.

Step 3. 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 3.5: *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 3.2, 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 3.3 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 Lemma 3.1. 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 set of locations the *multilocation* of p in T. The specific 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 [8] 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 [8] 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 cousin in $A(z)$. For each $s_i \in A(v)$ let $up(s_i)$ be a pointer to the cousin of s_i in $A(z)$.

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

End of AUGMENT.

Theorem 3.6: *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 T, namely, $O(n \log n)$. We prove this by examining the extent that 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 cousin 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 3.7: *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 [6] is to find the trapezoidal edge(s) for each vertex of P (see Figure 2).

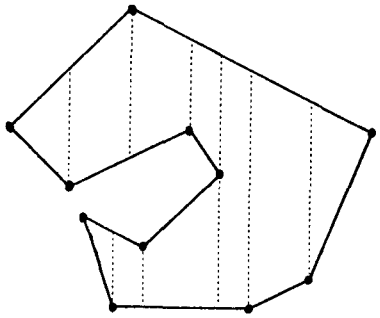

Figure 2: A trapezoidal decomposition of a simple polygon.

Theorem 3.8: *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 below each vertex as follows. First, use algorithms BUILDUP and AUGMENT to construct an augmented plane-sweep tree T' for S. As in [1], we solve the problem by performing a multilocation of each $v_i \in P$. In our case we use Theorem 3.6 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 3.7), we can construct a trapezoidal decomposition of P in these same bounds. ∎

In the next subsection we show how to use trapezoidal decomposition in solving the 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 [14] 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 3).

End of algorithm DECOMP.

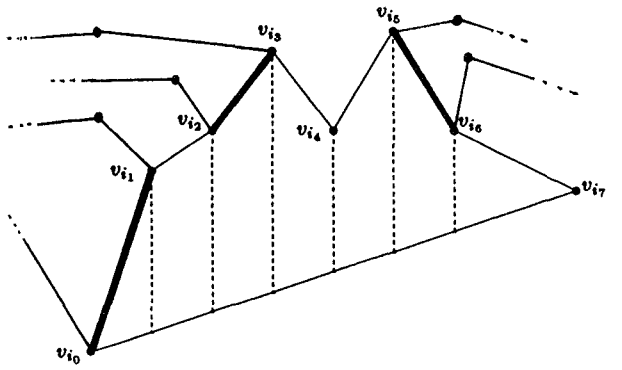

Figure 3: The polygon P_i for $s_i = (v_{i_0}, v_{i_7})$ and $V_i = \{v_{i_1}, v_{i_2}, \ldots, v_{i_6}\}$. The edges in P_i but not in P are shown in boldface. Note that the sequence of vertices $v_{i_1}, \ldots, v_{i_6}$ is monotone in the x-direction.

Theorem 3.9: *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, because 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. It is easy to show that the vertices of V_i are all on the same side of s_i; that is, that each polygon P_i is one-sided (we omit the proof). 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 already made in this paper. ∎

After decomposing P into polygons $P_1, P_2, \ldots, P_k$, we now triangulate each P_i in parallel. The algorithm which follows will triangulate a one-sided monotone polygon in $O(\log n)$ time and $O(n)$ space using $O(n)$ processors.

Algorithm OSM-TRIANGULATE:

Input: A one-sided monotone polygon P. Let s denote the distinguished edge. WLOG, P is monotone in the x-direction.
Output: A triangulation of P.
Method: Let $V = \{v_1, v_2, \ldots, v_n\}$ denote the set of vertices of P which are not endpoints of s, and $s = (v_0, v_{n+1})$. WLOG, all the vertices of V are above s. One of the ideas in our algorithm is the use of the $\sqrt{n}$ parallel divide-and-conquer technique [1,2]. We divide the vertices of V into $\sqrt{n}$ subsets of size $\sqrt{n}$ each, find the lower convex hull of each subset, and triangulate all the parts of P above the lower hull edges recursively in parallel (Steps 1 and 2). We then repeatedly merge adjacent pairs of lower hulls into single lower hulls, triangulating the portion of P between each pair. Unfortunately, doing this in a straightforward manner would result in an $O(\log^2 n)$ running time, because it takes $O(\log n)$ time to compute the common tangent line between two lower hulls. So we compute all the tangent lines which will merge pairs of hulls as a preprocessing step (Step 3) to the conquer step (Steps 4–6). This allows us to do the pair-wise hull-mergings in constant time. After we complete all the lower-hull mergings, the untriangulated portions of P are structured so as to be triangulatable in $O(\log n)$ time. The details 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, and compute the lower convex hull $LH(V_i)$ of the vertices of each subset V_i in parallel (see Figure 4). Add all hull edges to P (if they are not already edges of P).

Comment: The vertices of each $LH(V_i)$ are listed by increasing x-coordinate. The lower hull of m points in the plane sorted by x-coordinate can be constructed in $O(\log m)$ time and $O(m)$ space using $O(m)$ processors [1,2].

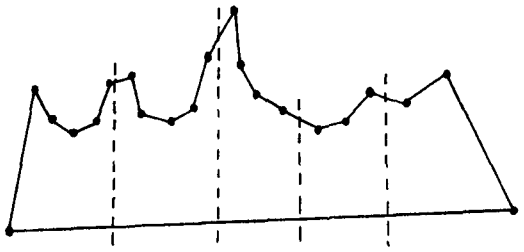

Figure 4: The $\sqrt{n}$ lower hulls associated with V.

Step 2. For each edge s' added to form $LH(V_i)$ there is a subset of vertices in V_i which are monotone in the x-direction and directly above s'. Thus, each such s' determines a one-sided monotone polygon, with at most $O(\sqrt{n})$ vertices. Recursively triangulate the polygons determined by each s' in parallel for all such edges s'.

Step 3. Build a complete binary tree B "on top" of the subsets V_i such that each leaf corresponds to a single V_i. For each $w \in B$ find the tangent t_w between $LH(lchild(w))$ and $LH(rchild(w))$, where $LH(w)$ denotes the lower hull of the descendants of w, by doing the following (note: in Step 3 we don't actually compute $LH(w)$, just the common tangent t_w):

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

Comment: The common tangent line between 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 [18]. 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 $rchild(w)$. Find the minimum tangent line t_w for each T_w in parallel, where comparisons are based on the intersection of the tangent lines with the vertical line separating the descendants of $lchild(w)$ and $rchild(w)$, respectively (see Figure 5).

Comment: t_w is the tangent line between $LH(lchild(w))$ and $LH(rchild(w))$.

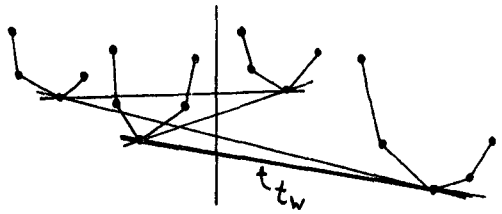

Figure 5: The tangent lines in T_w: between descendants of $lchild(w)$ and descendants of $rchild(w)$. The tangent t_w is shown in boldface.

Step 4. For each $w \in B$, construct P_w, the polygon which consists of t_w together with the portions of $LH(lchild(w))$ and $LH(rchild(w))$ that are above t_w (and hence do not appear in $LH(w)$; see Figure 6), by doing the following:

Step 4.1. **For** l = lowest level **until** $l = 0$ **repeat** Steps 4.2–4.3 below for each $w \in B$ on level l in parallel:

Comment: Let $w_1 = lchild(w)$ and $w_2 = rchild(w)$. Assume that $LH(w_1)$ and $LH(w_2)$ were constructed in the previous iteration, and that the vertices in $LH(w_1)$ and $LH(w_2)$ are sorted by x-coordinates.

Step 4.2. WLOG, the descendants of w_1 have smaller x-coordinates than the descendants of w_2. Let $t_w = (v_1, v_2)$, where $v_1 \in LH(w_1)$ and $v_2 \in LH(w_2)$.

Construct $LH(w)$ by concatenating the portion of $LH(w_1)$ left of v_1 (inclusive) with the portion of $LH(w_2)$ right of v_2 (inclusive).

Step 4.3. Concatenate the portion of $LH(w_1)$ right of v_1 (inclusive) with the portion of $LH(w_2)$ left of v_2 (inclusive). Let P_w denote the polygon consisting of this list and the edge t_w.

Comment: Steps 4.2 and 4.3 can both be done in constant time using $O(n_w)$ processors, where $n_w = |LH(w_1)| + |LH(w_2)|$.

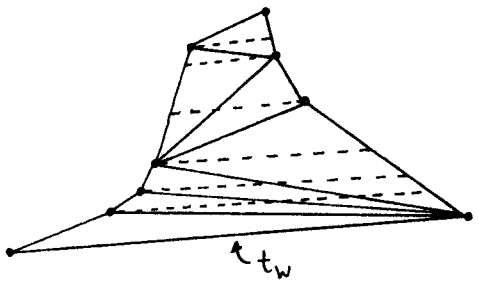

Figure 6: The triangulated polygon P_w. The left convex chain is from $LH(lchild(w))$ and the right convex chain is from $LH(rchild(w))$.

Step 5. Triangulate each P_w by doing the following for each P_w in parallel.

Step 5.1. For each $v_i \in P_w$ find the edge $e_i = (v_j, v_{j+1})$ in P_w which is intersected by the line containing v_i and parallel to t_w.

Comment: This can be done in $O(\log \log n)$ time using parallel merging. 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 v_i to the lower of the two endpoints of e_i, for each v_i in parallel (see Figure 6).

Comment: After completing Step 5 we have triangulated everything but the portion of P between $LH(V)$ and $s = (v_0, v_{n+1})$. Note that each point on the lower hull of V is visible from either v_0 or v_{n+1} (possibly from both).

Step 6. Let $LH(V) = \{v_{i_1}, v_{i_2}, \ldots, v_{i_l}\}$, $i_1 < i_2 < \cdots < i_l$. Let $v_0 v_{i_j}$ be tangent to $LH(V)$. Complete the triangulation of P by adding the edges $(v_0, v_{i_1}), \ldots, (v_0, v_{i_j})$ and $(v_{i_j}, v_{n+1}), \ldots, (v_{i_l}, v_{n+1})$ (see Figure 7).

End of algorithm OSM-TRIANGULATE.

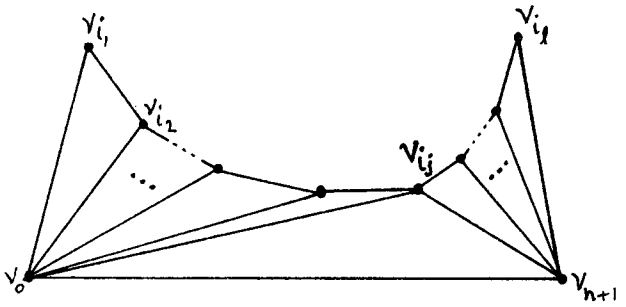

Figure 7: Triangulating the remaining portion of P.

Theorem 3.10: *The algorithm OSM-TRIANGULATE correctly triangulates a one-sided monotone polygon P in $O(\log n)$ time and $O(n)$ space using $O(n)$ processors.*

Proof: The correctness of OSM-TRIANGULATE follows by induction and the comments made in Step 5. We have already observed that Steps 1, 3, 4, 5, and 6 can all be done in $O(\log n)$ time and $O(n)$ space using $O(n)$ processors. Thus, the time complexity of OSM-TRIANGULATE, $T(n)$, is determined by the recurrence relation $T(n) = T(\sqrt{n}) + O(\log n)$, which has solution $T(n) = O(\log n)$. Also, the number of processors used, $P(n)$, is determined by the recurrence $P(n) = \max\{\sqrt{n}P(\sqrt{n}), cn\}$ for some constant c, which has solution $P(n) = O(n)$. ■

Theorems 3.9 and 3.10 imply that we can triangulate a simple polygon 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 next show 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 3.11: *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 3.8. ■

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 3.12 [1]: *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 3.13: *Given n line segments in the plane we can 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. 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 3.6 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 3.7 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 $p_i \in \Pi_v$. ∎

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 problem sequentially, we scan a set of objects by sliding a vertical line along the x-axis, storing the objects in some kind of binary search tree as we go. At various points (*critical* points) during the plane-sweeping we perform 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: visibility from a point, 3-dimensional maxima, multiple range-counting, and rectilinear segment intersection counting.

4.1 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. 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 $S = \{s_1, s_2, \ldots, s_n\}$.

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 $(x(p_i), x(p_{i+1}))$.

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.

Comment: 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 [5,20] 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 $p_i \in X$ if p_i "came from" X_1, then define $VIS(p_i) = \min\{VIS_1(p_i), VIS_2(c(p_i))\}$, where $c(p_i)$ denotes the cousin of p_i in X_2. If p_i came from X_2, then define $VIS(p_i) = \min\{VIS_1(c(p_i)), VIS_2(p_i)\}$, where $c(p_i)$ is the cousin of p_i in X_1. If $c(p_i) = \phi$ (i.e., p_i has no cousin), then we take $VIS_1(\phi) = VIS_2(\phi) = +\infty$.

Comment: Taking the minimum of $VIS_1(p_i)$ and $VIS_2(c(p_i))$ (or $VIS_1(c(p_i))$ 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. 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 4.1: *The algorithm VISIBILITY solves the visibility from a point problem in $O(\log n \log\log n)$ time and $O(n)$ space using $O(n)$ processors on a CREW PRAM.*

Proof: The correctness proof of VISIBILITY is by induction and is omitted. The main observation is that in the conquer step (3) when computing $VIS(p_i)$ we need only compare the two segments which span the 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 cousin in the other set.

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 $O(n)$. ∎

The next application we look at is 3-dimensional maxima.

4.2 3-Dimensional Maxima

Let $V = \{p_1, p_2, \ldots, p_n\}$ be a set of points in $\Re^3$. 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 optimal sequential algorithm for this problem [15]. For simplicity, we assume that no two input points have the same x (resp., y, z) coordinate.

Algorithm 3-D MAXIMA:

Input: A list of points $V = \{p_1, p_2, \ldots, p_n\}$ in $\Re^3$.

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

Step 0. (Preprocessing) Sort the points of V by y-coordinate. (This preprocessing step is performed only once.)

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.

Comment: 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 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 $q_i \in X$ if q_i came from X_1 then let $ZO(q_i) = \max\{ZO_1(q_i), ZO_2(c(q_i))\}$ and $ZT(q_i) = \max\{ZT_1(q_i), ZO_2(c(q_i))\}$, where $c(q_i)$ is the cousin of q_i in X_2. If q_i came from X_2, then $ZO(q_i) = \max\{ZO_1(c(q_i)), ZO_2(q_i)\}$ and $ZT(q_i) = ZT_2(q_i)$, where $c(q_i)$ is the cousin of q_i in X_1. (By convention, $ZO_j(\phi) = ZT_j(\phi) = z_j$, where z_j is the maximum z-coordinate in X_j, $j = 1, 2$. Note that we can easily compute z_j, since it is the maximum of $ZO_j(q)$ and $z(q)$, where q is the first element in X_j.)

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.

Theorem 4.2: *The algorithm 3-D MAXIMA solves the 3-dimensional maxima problem in $O(\log n \log\log n)$ time and $O(n)$ space using $O(n)$ processors on a CREW PRAM.*

Proof: The proof of correctness is by induction, and is given in detail in the technical report [3]. By the same argument as in the proof for Theorem 4.1 the algorithm 3-D MAXIMA runs in $O(\log n \log\log n)$ time and $O(n)$ space using $O(n)$ processors. ∎

It is worth noting that we can use the algorithm 3-D MAXIMA as the bottom 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 [4] and parallel prefix [14]).

Theorem 4.3: *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 parallization of the algorithm by Kung, Luccio, and Preparata [15], using 3-D MAXIMA as the basis for the recursion. We omit the details. ∎

Next, we address the multiple range-counting problem.

4.3 Multiple Range-Counting

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. We know from [10] 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 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 dominance counting problem. The next algorithm does this.

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 one list W, and sort the points in W by y-coordinate. Also, we mark each point in W which came from V. Initially, the CO and CT label for each point is 0.

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

Comment: 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 $v_i \in X$ if v_i came from X_1, then define $CO(v_i) = CO_1(v_i) + CO_2(c(v_i)) + \chi_V(c(v_i))$ and $CT(v_i) = CT_1(v_i)$, where $c(v_i)$ is the cousin of v_i in X_2. If v_i came from X_2, then define $CO(v_i) = CO_1(c(v_i)) + CO_2(v_i) + \chi_V(c(v_i))$ and $CT(v_i) = CT_1(c(v_i)) + CT_2(v_i) + \chi_V(c(v_i))$, where $c(v_i)$ is the cousin of v_i in X_1. ($CO_j(\phi) = CT_j(\phi) = \chi_V(\phi) = 0$, $j = 1, 2$.)

Comment: The dominance count of each v_i is stored in the label $CT(v_i)$.

End of Algorithm DOM-COUNT.

Theorem 4.4: *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, and is given in detail in the technical report [3]. 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 $n = l + m$. ■

Corollary 4.5: *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 $n = l + m$.* ■

4.4 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.

Theorem 4.6: *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: The method is similar to that used for multiple range counting. The details are given in the technical report [3]. ■

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 applied divide-and-conquer with critical-point merging to visibility from a point, 3-dimensional maxima, 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

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

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

[3] M.J. Atallah and M.T. Goodrich, "Efficient Plane Sweeping in Parallel," Purdue University Computer Science Tech. Report CSD-TR-563, March 1986.

[4] M. Ajtai, J. Komlós, and E. Szemerédi, "Sorting in $c \log n$ parallel steps," *Combinatorica*, Vol. 3, 1983, 1–19.

[5] A. Borodin and J.E. Hopcroft, "Routing, Merging, and Sorting on Parallel Models of Computation," *Jour. of Comp. and Sys. Sci.*, Vol. 30, No. 1, Feb. 1985, 130–145.

[6] B. Chazelle and J. Incerpi, "Triangulating a Polygon by Divide-and-Conquer," *Proc. of 21st Allerton Conf. on Comm. Control, and Comp.*, 1983, 447–456.

[7] B. Chazelle, "Intersecting is Easier Than Sorting," *Proc. 16th ACM Symp. Theory of Comp.*, 1984, 125–134.

[8] B. Chazelle and L.J. Guibas, "Fractional Cascading: I. A Data Structuring Technique," manuscript.

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

[10] H. Edelsbrunner and M.H. Overmars, "On the Equivalence of Some Rectangle Problems," *Info. Proc. Letters*, Vol. 14, No. 3, May 1982, 124–127.

[11] H. El Gindy and D. Avis, "A Linear Algorithm for Computing the Visibility Polygon from a Point," *J. of Algorithms*, Vol. 2, 1981, 186–197.

[12] M.T. Goodrich, "An Optimal Parallel Algorithm For the All Nearest-Neighbor Problem for a Convex Polygon," Purdue University Computer Science Tech. Report CSD-TR-533, August 1985.

[13] D. Kirkpatrick, "Optimal Search in Planar Subdivision," *SIAM Jour. on Comp.*, Vol. 12, No. 1, Feb. 1983, 28–35.

[14] C. Kruskal, L. Rudolph, and M. Snir, "The Power of Parallel Prefix," *Proc. 1985 Int. Conf. on Parallel Proc.*, 180–185.

[15] H.T. Kung, F. Luccio, F.P. Preparata, "On Finding the Maxima of a Set of Vectors," *Jour. of ACM*, Vol. 22, No. 4, October 1975, 469–476.

[16] D.T. Lee and F.P. Preparata, "Computational Geometry—A Survey," *IEEE Trans. on Computers*, Vol. C-33, No. 12, December 1984, 872–1101.

[17] G. Lueker and D. Willard, "A Data Structure for Dynamic Range Queries," *Info. Proc. Letters*, Vol. 15, No. 5, December 1982, 209–213.

[18] M.H. Overmars and J. Van Leeuwen, "Maintenance of Configurations in the Plane," *Jour. of Comp. and Sys. Sci.*, Vol. 23, 1981, 166–204.

[19] M. Shamos and D. Hoey, "Geometric Intersection Problems," *Proc. 17th IEEE Symp. Found. of Comp. Sci.*, 1976, 208–215.

[20] L. Valiant, "Parallelism in Comparison Problems," *SIAM Jour. on Comp.*, Vol. 4, No. 3, Sept. 1975, 348–355.