Deterministic Parallel Computational Geometry

Mikhail J. Atallah and Michael T. Goodrich

1 Introduction

Computational Geometry is concerned with the design and analysis of algorithms for solving geometric problems. These problems generally deal with collections of simple geometric objects, such as lines, points, planes, circles, etc., about which we are asked to answer some basic questions. Many of the problems in computational geometry come from applications in pattern recognition, computer graphics, statistics, operations research, computer-aided design, robotics, etc. The problems that arise from these areas are likely to come from real-time applications or situations in which the input consists of a large number of geometric objects. This implies that these problems need to be solved as fast as possible. For many of these problems, however, we already are at the limits of what can be achieved through sequential computation. Thus, it is natural to study what kinds of speed-ups can be achieved through parallel computing. As an indication of the importance of this research direction, we note that four of the eleven problems used as benchmark problems to evaluate parallel architectures for the DARPA Architecture Workshop Benchmark Study of 1986 were computational geometry problems.

Unfortunately, many of the techniques used to find efficient sequential algorithms for computational geometry problems do not translate well into a parallel setting. While providing elegant paradigms for designing sequential algorithms, these techniques use methods that seem to be inherently sequential. Therefore, one needs to develop new paradigms for computational geometry, ones better suited for finding efficient parallel algorithms. This chapter gives a number of algorithms for solving computational geometry problems efficiently in parallel. The algorithms all have linear or "almost" linear speed-ups over the best known sequential algorithms for these

problems.

The geometric problems that we address in this chapter all deal with planar objects. The only exception is in Section 5, where we consider a problem dealing with 3-dimensional points. In each section in this chapter we address an important computational geometry problem and show how one can solve this problem efficiently in parallel. In some cases the parallel efficiency comes primarily from new geometric insights, and in other cases it comes primarily from new algorithmic techniques, which is an interesting aspect of parallel computational geometry. The specific problems we address include computing convex hulls, constructing the intersection of half planes, computing the visible region from a point in the presence of opaque obstacles, finding the separation distance between two polygons, and computing three-dimensional maxima points. Each of these problems are considered fundamental in computational geometry; we have tried to motivate their significance where appropriate.

2 Convex Hull

This section deals with the problem of computing the convex hull of a set S of points in the plane. We begin with a few definitions. A *polygonal chain* is a sequence $(p_1, s_1, p_2, s_2, \ldots p_n, s_n, p_{n+1})$, where each s_i is a line segment with endpoints p_i and p_{i+1}. This notational convention implies that s_i and s_{i+1} share a common endpoint, p_{i+1}, since each s_i can alternately be denoted $\overline{p_i p_{i+1}}$. The s_i's are called the *edges* of the polygonal chain, and the p_i's are called its *vertices*. A polygonal chain is *simple* if for every segment s_i in it, the only other segments that intersect s_i are s_{i-1} and s_{i+1}, and their respective intersections with s_i are p_i and p_{i+1} (see Figure 1a). Note, to simplify our discussions we sometimes use the the term *n-gon* as a shorthand for "simple polygon with n edges." Unless otherwise stated, every polygonal chain we consider in this chapter is simple. A polygonal chain is the *boundary of a simple polygon* if $p_1 = p_{n+1}$, in which case its sequence of vertices and edges can be viewed as being *circular*, in the sense that one can write the sequence beginning at any vertex. In this case the polygonal chain partitions the plane into *two* regions—one finite region, the other infinite (this is due to a topological fact known as the Jordan Curve Theorem). By convention, the finite region is called the *interior* of the polygon, the

Figure 1: Examples of (a) a simple polygonal chain and (b) a polygon.

polygonal chain is called the *boundary* of the polygon, and the infinite region is called the *exterior* of the polygon (see Figure 1b).

Let $P = (p_1, s_1, p_2, \ldots, p_n, s_n, p_{n+1})$ be a polygon. A polygon $P' = (p_{i_1}, s'_{i_1}, p_{i_2}, \ldots, p_{i_k}, s'_{i_k}, p_{i_{k+1}})$ is a *subpolygon* of P if $1 \leq i_1 < i_2 < \cdots < i_{k+1} \leq n+1$, and each s'_{i_j} is the edge from p_{i_j} to $p_{i_{j+1}}$. This definition is analogous to the notion of a subsequence of a sequence of numbers.

A planar region R is *convex* if, given any two points p and q in R, the line segment $\overline{pq}$ is also (entirely) in R. If R is a simple polygon, then this is equivalent to requiring that the intersection of R with any line is either empty or a line segment. The following lemma establishes an interesting property of subpolygons of convex polygons.

Lemma 2.1 *Any subpolygon of a convex polygon is itself a convex polygon.*

Proof: See Exercise 5. ■

As we show in this section, this simple geometric observation has important consequences for the design of an efficient parallel algorithm for constructing the *convex hull* of a set S of n points in the plane. Given such a set of points, the convex hull of S is the smallest convex polygon that contains no points of S in its exterior, i.e., each point of S is either in its interior or on its boundary. The *convex hull problem* is to produce a polygonal representation of the convex hull of S, with the vertices listed in clockwise order (so the interior of the convex hull is always to the right of each convex hull edge). Intuitively, we can imagine the plane to be a large wooden board with a nail sticking up from the position of each point in S. If we were to stretch a rubber band so that it surrounds all the nails, and let the rubber band shrink to a resting state, then it would assume the contour of the boundary of the convex hull of S, with

Figure 2: Illustrating the upper and lower hulls.

a nail at each vertex. Convex hulls have applications in a host of problem domains, including computer vision, computer graphics, and statistics. They are often useful any time one wishes to capture the "shape" of a set of points.

The problem of constructing the convex hull of a set S of n points in the plane is known to have an $O(n \log n)$ time sequential solution (see Exercise 1), together with an $\Omega(n \log n)$ lower bound in the comparison model (see Exercise 2). In this section we give an n-processor CREW PRAM algorithm that runs in $O(\log n)$ time.

Let $S = \{p_1, p_2, \ldots, p_n\}$. The convex hull of S consists of two portions: the *upper hull* and the *lower hull*. The upper (resp., lower) hull consists of the edges of the hull such that the interior of the hull polygon is below (resp., above) them. See Figure 2. We shall focus on the problem of computing the upper hull $UH(S)$. The problem of computing the lower hull $LH(S)$ is symmetrical and therefore omitted. We assume that the recursive procedure returns an array $UH(S)$ containing the points of the upper hull in left to right order. It also returns the size of the upper hull (i.e., $|UH(S)|$).

To simplify the exposition, we assume that n is a power of 2, that no two points in S have same x (respectively, y) coordinate, and that no three points in S lie on the same line. We also assume that the points have already been sorted by x coordinate, so that $x(p_i) < x(p_{i+1})$ for all $i \in \{1, 2, \ldots, n-1\}$. This can be accomplished in $O(\log n)$ time using n processors in the CREW PRAM model by the parallel mergesort procedure.

The main idea is to recursively solve the problem for the two point sets $S_1 = \{p_1, p_2, \ldots, p_{(n/2)-1}\}$ and $S_2 = \{p_{(n/2)}, p_{(n/2)+1}, \ldots, p_n\}$. As we will show, when these two recursive calls return $UH(S_1)$ and (respectively) $UH(S_2)$, we can com-

Figure 3: Illustrating the definition of u and v.

bine their answers and obtain $UH(S)$ in constant time and using a linear number of processors on a CREW PRAM. Thus, the time and processor recurrences will be $T(n) = T(n/2) + c_1$ and $P(n) = \max\{n, 2P(n/2)\}$, respectively with boundary conditions $T(1) = c_2$ and $P(1) = 1$. Their solutions are $T(n) = O(\log n)$ and $P(n) = n$, respectively. We now show that the "combine" stage can indeed be done in constant time with a linear number of processors.

Consider the line T that is above and tangent to both $UH(S_1)$ and $UH(S_2)$. Let u (resp., v) be the point at which this tangent touches $UH(S_1)$ (resp., $UH(S_2)$). See Figure 3. The line segment $\overline{uv}$ is called the *upper common tangent* to $UH(S_1)$ and $UH(S_2)$. Observe that $UH(S)$ consists of the portion of $UH(S_1)$ to the left of u, followed by u, the edge $\overline{uv}$, v, and the portion of $UH(S_2)$ to the right of v. This observation implies that, if we somehow knew the points u and v, then we could obtain $UH(S)$ in constant time with $O(n)$ processors, as follows. Let α (resp., β) be the rank of u (resp., v) in the array $UH(S_1)$ (resp., $UH(S_2)$). We copy the first α entries of $UH(S_1)$ into the first α positions of $UH(S)$, add the edge $\overline{uv}$, and then fill the remainder of $UH(S)$ with the last $|UH(S_2)| - \beta + 1$ entries of $UH(S_2)$. Thus the problem of obtaining $UH(S)$ from $UH(S_1)$ and $US(S_2)$ is essentially that of identifying these two points u and v.

Let $UH(S_1) = (p_1, s_1, p_2, \ldots, p_l)$ and $UH(S_2) = (q_1, t_1, q_2, \ldots, q_m)$, where $l \leq n/2$ and $m \leq n/2$. The procedure we now give for locating u and v uses $l + m$ ($\leq n$) processors. Before we describe this method, however, note that if we had lm processors available, then it would be trivial to find the desired upper common tangent in constant time by the following "brute force" algorithm: assign a processor to each possible pair (p_i, q_j). This processor tests whether this pair is the one we seek

Figure 4: Counterexample.

(i.e., whether $p_i = u$ and $q_j = v$) by checking, in constant time, whether the segment $\overline{p_i q_j}$ is tangent to $UH(S_1)$ at p_i and to $UH(S_2)$ at q_j. This test is accomplished just by looking locally around p_i and q_j. Our goal, however, is to use only $l+m$ processors.

One may be tempted to give the following constant time, n processor "solution" (**which doesn't work**) :

1. Consider the subpolygon P of $UH(S_1)$ obtained by choosing every $(k\sqrt{l})$-th vertex of $UH(S_1)$, $1 \leq k \leq \sqrt{l}$. That is, P is the $\sqrt{l}$-gon $P = (p_{\sqrt{l}}, s'_{\sqrt{l}}, p_{2\sqrt{l}}, \ldots, p_l)$.

2. Consider the subpolygon Q of $UH(S_2)$ obtained by choosing every $(k\sqrt{m})$-th vertex of $UH(S_2)$, $1 \leq k \leq \sqrt{m}$. That is, Q is the $\sqrt{m}$-gon $Q = (q_{\sqrt{m}}, t'_{\sqrt{m}}, q_{2\sqrt{m}}, \ldots, q_m)$.

3. Use the above-mentioned brute force approach to find the common tangent to P and Q in constant time. (It requires $\sqrt{lm} \leq l + m$ processors.) Say it is the line joining $p_{i\sqrt{l}} \in P$ to $q_{j\sqrt{m}} \in Q$.

4. The vertices of P divide $UH(S_1)$ (resp., $UH(S_2)$) into $\sqrt{l}$ portions, call them P_1, $P_2, \ldots$, $P_{\sqrt{l}}$. Similarly, The vertices of Q divide $UH(S_2)$ into $\sqrt{m}$ portions, call them Q_1, $Q_2, \ldots$, $Q_{\sqrt{m}}$. Use the brute force algorithm between the $2\sqrt{l}$ points in $P_i \cup P_{i+1}$ and the $2\sqrt{m}$ points in $Q_j \cup Q_{j+1}$ (i.e., between the two portions of P adjacent to $p_{i\sqrt{l}}$ and the two portions of Q adjacent to $q_{j\sqrt{m}}$).

The reason the above approach fails is that the "locality" property needed for Step 4 need not hold. Namely, we do not necessarily have $u \in P_i \cup P_{i+1}$ and $v \in Q_j \cup Q_{j+1}$. Indeed, the portion of $UH(S_1)$ containing u might be quite far from $p_{i\sqrt{l}}$, as might the portion of $UH(S_2)$ containing v be quite far from $q_{j\sqrt{m}}$. Figure 4 gives an example of how this might happen. The correct solution to the common tangent problem makes a more judicious use of the basic idea of the above (erroneous) steps 1–4. It also makes use of the next two propositions.

Proposition 1 *Let p be any point of $UH(S_1)$. Then the upper tangent to $UH(S_2)$ passing through p can be computed in time $O(k)$ by an $m^{1/k}$ processor CREW PRAM, where k is any integer such that $2 \leq k \leq \log m$. Similarly, if p is any point of $UH(S_2)$, then the upper tangent to $UH(S_1)$ passing through p can be computed in time $O(k)$ by an $l^{1/k}$ processor CREW PRAM, where k is any integer such that $2 \leq k \leq \log l$.*

Proof. We give the proof for $p \in UH(S_1)$ (the proof for the $p \in UH(S_2)$ case is similar). Let $m' = m^{1-1/k}$. Let Q' be the subpolygon of $UH(S_2)$ consisting of every m'-th vertex of $UH(S_2)$, i.e. $Q' = (q_{m'}, q_{2m'}, \ldots, q_m)$. Since Q' has $m^{1/k}$ vertices and we have $m^{1/k}$ processors, it is easy to find in constant time the upper tangent to Q' passing through p, say this tangent touches Q' at $q_{im'}$ (the tangent is found in constant time by the same method as for searching for x in an array of n elements using n CREW PRAM processors). Let q_j be the vertex of $UH(S_2)$ at which the desired tangent touches $UH(S_2)$. We can easily test whether $q_j = q_{im'}$ in constant time and one processor, just by looking in the vicinity of $q_{im'}$ in $UH(S_2)$. If the test is positive, and $q_j = q_{im'}$, then we are done. So suppose $q_j \neq q_{im'}$. We then test whether q_j is to the left of $q_{im'}$ or to the right of $q_{im'}$, as follows. Let L be the line through p and $q_{im'}$. Also, let L_{left} the portion of L strictly to the left of $q_{im'}$, and let L_{right} be the portion of L strictly to the right of $q_{im'}$ (so neither L_{left} nor L_{right} contain $q_{im'}$). If L_{left} is above segment $q_{im'-1}q_{im'}$ and L_{right} is below segment $q_{im'}q_{im'+1}$ (see Figure 5a), then q_j is to the right of $q_{im'}$. Otherwise, if L_{left} is below segment $q_{im'-1}q_{im'}$ and L_{right} is above segment $q_{im'}q_{im'+1}$, then q_j is to the right of $q_{im'}$ (see Figure 5b). Without loss of generality, assume the test reveals that q_j is to the left of $q_{im'}$, i.e., $j < im'$ (the case $im' < j$ is symmetrical). Then it is not hard to prove that we have $(i-1)m' \leq j$ (see Exercise 8). Therefore, it suffices to find the upper tangent to the subpolygon

Figure 5: Illustrating the proof of Proposition 1.

$(q_{im'-m'}, s'_{im'-m'}, q_{im'-m'+1}, \ldots, q_{im'-1})$ passing through p. Thus, by doing a constant amount of work, we have reduced the polygon size by a factor of $m^{1/k}$. Doing this at most k times finds the desired point of tangency. ∎

Proposition 2 *Given a vertex, p, on $UH(S_1)$, one can determine if u is to the left of p, to the right of p, or at p in $O(k)$ time using $m^{1/k}$ processors, where k is any integer such that $2 \leq k \leq \log m$. Similarly, Given a vertex, q, on $UH(S_2)$, one can determine if v is to the left of q, to the right of q, or at q in $O(k)$ time using $l^{1/k}$ processors, where k is any integer such that $2 \leq k \leq \log l$.*

Proof. We only consider the case $p \in UH(S_1)$, since the case $q \in UH(S_2)$ is similar. Use the previous proposition to find the tangent to $UH(S_2)$ passing through point p; let T be this tangent. If T is tangent to P then $u = p$. Otherwise, let γ be the vertex of P just to the left of p. Observe that u is to the left of p on $UH(S_1)$ if and only if γ is above line T. ∎

We can now give the algorithm **TANGENTS** for finding the upper common tangent to $UH(S_1)$ and $UH(S_2)$ (and hence u and v). Recall that both $UH(S_1)$ and $UH(S_2)$ are monotone in the x direction, i.e., the x-coordinate of p_i is smaller than that of p_{i+1} and the x-coordinate of q_i is smaller than that of q_{i+1}.

1. Set $P' := UH(S_1)$, $Q' := UH(S_2)$.
   Comment. The algorithm will iteratively decrease the size of P' and/or Q', maintaining the property that the upper common tangent between $UH(S_1)$ and $UH(S_2)$ is the same as the one between P' and Q'.

2. Repeat the following steps 3–7 until either P' is a single point or Q' is a single point. Without loss of generality, assume that it is P' that ends up becoming

Figure 6: Illustrating algorithm **TANGENTS**.

a single point (call it p_u); now use Proposition 1 to find, in constant time, the tangent to Q' passing through p_u, and output the tangent thus found (this is the desired upper common tangent between $UH(S_1)$ and $UH(S_2)$).

3. Let $P'' = (a_1, s'_1, a_2, \ldots, a_{\sqrt{l}})$ be the subpolygon obtained by considering every $\sqrt{l}$-th vertex of P', i.e., the $\sqrt{l}$ vertices of P'' divide P' into $\sqrt{l}$ equal portions. Call these portions A_1, $A_2, \ldots$, $A_{\sqrt{l}}$, so that a_i is adjacent in P' to portions A_i and A_{i+1}. By convention, a_i belongs to A_i but not to A_{i+1}. Let $Q'' = (b_1, t'_1, b_2, \ldots, b_{\sqrt{m}})$ be analogously defined for Q', and let the resulting portions of Q' be B_1, $B_2, \ldots$, $B_{\sqrt{m}}$. Use the already mentioned brute force method for finding the common tangent between P'' and Q'' (this is possible and takes constant time because we have $l + m \geq \sqrt{lm}$ processors). Say the tangent thus found joins $a_i \in P''$ to $b_j \in Q''$. (See Figure 6.)

4. Test whether the common tangent to P' and Q' touches P' in A_i. (This is done in constant time by using Proposition 2 twice, once at vertex p_{i-1} and once at vertex p_i.) If the answer is "yes" then do $P' := A_i$, otherwise P' remains unchanged.

   Implementation Note. The assignment $P' := A_i$ is done in constant time simply by remembering the new first and last vertex of P'.

5. Test whether the common tangent to P' and Q' touches P' in A_{i+1}. If it does then do $P' := A_{i+1}$, otherwise P' remains unchanged.

6. Test whether the common tangent to P' and Q' touches Q' in B_j. If it does then do $Q' := B_j$, otherwise Q' remains unchanged.

7. Test whether the common tangent to P' and Q' touches Q' in B_{j+1}. If it does then do $Q' := B_{j+1}$, otherwise Q' remains unchanged.

Since every usage of Proposition 2 takes constant time, the time complexity of the algorithm is equal to the number of times that steps 3–7 get executed. We now bound the number of times steps 3–7 are executed.

Lemma 2.2 *Let a_i, b_j, P', Q', P'', and Q'' be as in Step 3 of the algorithm* **TANGENTS**. *Also, let $p_u q_v$ be the common tangent to P' and Q' ($p_u \in P'$, $q_v \in Q'$). Then at least one of the following statements (1), (2), (3), or (4) is true:*

1. $p_u \in A_i$;
2. $p_u \in A_{i+1}$;
3. $q_v \in B_j$;
4. $q_v \in B_{j+1}$.

Proof. If $p_u = a_i$ or $q_v = b_j$ then the lemma holds, so suppose that $p_u \neq a_i$ and $q_v \neq b_j$. By its definition, the line through p_u and q_v is above both a_i and b_j. Therefore at least one of p_u or q_v is above the line through a_i and b_j. If p_u is above the line through a_i and b_j, then we prove that (1) or (2) must hold by the following case analysis:

Case 1: In P', p_u is to the left of a_i. Then we claim that $p_u \in A_i$ (and hence (1) holds). Suppose to the contrary that $p_u \notin A_w$ where $w < i$. By the definition of a_i and b_j, the vertex $a_w \in P''$ must lie on or below the line $a_i b_j$. The three vertices p_u, a_w, a_i occur in that order on P' (see Figure 7). Consider the positions of these three vertices relative to the line $a_i b_j$: The first vertex is (by hypothesis) above that line, the second is (as we have just argued) on or below it, and the third is (by definition) on it. This contradicts the convexity of P'. Thus, (1) holds.

Case 2: In P', p_u is to the right of a_i. An argument similar to that for Case 1 shows that $p_u \in A_{i+1}$; hence, (2) holds.

If q_v is above line $a_i b_j$, then an argument similar to that above shows that one of (3) or (4) must hold. ■

Figure 7: Illustrating Case 1.

Corollary 2.3 *Steps 3–7 of algorithm* **TANGENTS** *are executed at most three times.*

Proof. Lemma 2.2 implies that, every time we execute steps 3–7, at least one of the statements $P' := A_i$, $P' := A_{i+1}$, $Q' := B_j$, $Q' := B_{j+1}$ is executed. This implies that either P' decreases in size by a factor of $\sqrt{l}$, or Q' decreases in size by a factor of $\sqrt{m}$. This proves the corollary. ∎

We have thus established the following:

Theorem 1 *Algorithm* **TANGENTS** *correctly computes the upper common tangent between $UH(S_1)$ and $UH(S_2)$, in constant time with $l + m$ processors.*

Corollary 2.4 *The convex hull of a planar set of n points can be computed in $O(\log n)$ time using n processors on a CREW PRAM.*

The next section uses the convex hull algorithm to solve the problem of computing the intersection of n half-planes. That a convex hull algorithm can be used in this way may seem surprising, since this problem is defined on a set of half-planes and any convex hull algorithm assumes its given a set of points.

3 Intersections of Half-Planes

Let $L = \{l_1, l_2, \ldots, l_n\}$ be a set of n planar lines. We let $H(l_i)$ denote the half-plane that is to the left of l_i, assuming a given orientation of l_i. One could, for example, specify each line l_i by two points p_i and q_i that determine it, with the convention that the orientation of l_i is from p_i to q_i. In this section we address the problem of computing the intersection of the n half-planes, i.e., in computing the region $F = \cap_{i=1}^{n} H(l_i)$. For the reader familiar with the terminology of operations

research, this problem is equivalent to computing the feasible region defined by a two-variable linear program. It has a number of applications, one of which we will explore (in Section 3.1).

Without loss of generality, we assume that no l_i is parallel to the y-axis. If this is not the case, then one can easily change the coordinate axes so that it is true. In this section we show how to compute F in $O(\log n)$ time and using n processors. This region is convex (see Exercise 4) and is described by a polygonal chain that may be closed or may begin and end at semi-infinite edges.

Rather than give a new algorithm for solving this problem, we shall show how it can actually be solved by using the convex hull algorithm of the previous section. The main idea is based on a geometric transform f that maps a point into a line and a line into a point: a point $p = (a, b)$ is mapped into the line $f(p)$ whose equation is $y = ax + b$, and the line l whose equation is $y = ax + b$ is mapped into the point $f(l) = (-a, b)$. It is easy to verify (Exercise 14) that the transform f maintains the relative positions of lines and points: if one of $\{\alpha, \beta\}$ is a line and one is a point, then α is below β if and only if $f(\alpha)$ is below $f(\beta)$. If A is a set of lines or points, then we use $f(A)$ to denote the set $\{f(a) : \ a \in A\}$.

To simplify the description of our method, we partition L into two sets, L^+ and L^-, where L^+ is the subset of L containing the lines l_i such that $H(l_i)$ is the half-plane above l_i, and L^- is the subset of L containing the lines l_i such that $H(l_i)$ is the half-plane below l_i. Let $F^+ = \cap_{l \in L^+} H(l)$ and $F^- = \cap_{l \in L^-} H(l)$. Then, clearly $F = F^+ \cap F^-$. Therefore, since computing the intersection of two convex polygonal chains can easily be done in $O(\log n)$ time with $O(n/\log n)$ processors (see Exercise 4), we can restrict our attention to the computation of F^+ and F^- separately. (See Figure 8.)

Before we describe our algorithm, however, we must generalize our definition of polygonal chains to allow for chains that begin and end with a ray (or "semi-infinite" line), rather than requiring that chains begin and end with a point. Specifically, we allow a polygonal chain P to be given as an alternating sequence of points and edges, where each point is given by its coordinates and each edge is specified by the line that contains it. This is essentially the same as the definition of the previous section, except we now allow a polygonal chain to have an edge at each end, as well as allowing

Figure 8: Examples of (a) C^+, (b) C^-, and (c) their intersection.

it have a point at each end, as before. An "edge" that is at the end of polygonal chain has only one endpoint, hence, is a ray. The beauty of using this convention is that, for any polygonal chain P, $f(P)$ is also a polygonal chain (see Exercise 15).

Since the computations of F^+ and F^- are symmetric, let us concentrate on the construction of F^+. Let $C^+ = (s_1, p_1, s_2, p_2 \ldots, s_{m-1}, p_{m-1}, s_m)$ denote the polygonal chain defining F^+, which, of course, starts and edges with a ray, one emanating from p_1 along the line specified by s_1 and one emanating from p_{m-1} along the line specified by s_m. Our algorithm for constructing C^+ is simply the following:

- Construct $f(C^+)$ and apply f^{-1} to get C^+.

But what is $f(C^+)$?

Lemma 3.1 *$f(C^+)$ is the upper hull of the points in $f(L^+)$, i.e., $f(C^+) = UH(f(L^+))$.*

Proof: As we have already observed, since C^+ is a polygonal chain having a ray at each end, $f(C^+)$ is a polygonal chain having a point at each end. Let I^+ be the set of intersections between lines in L^+, and observe that $f(I^+)$ is the set of lines determined all pairs of points of $f(L^+)$. Note that the vertices of C^+ are exactly those points in I^+ that are on or above all the lines in L^+, listed from left to right. Thus, the edges of $f(C^+)$ are determined by the *lines* of $f(I^+)$ that on are above all the *points* of $f(L^+)$, listed by increasing slopes. In other words, $f(C^+)$ is the list of lines that contain edges of the upper hull of $f(L^+)$, listed by increase slopes. Therefore, $f(C^+)$ is, in fact, the upper hull of $f(L^+)$, listed right to left. ∎

Since $f(L^+)$ and its upper hull can be computed in $O(\log n)$ time by n CREW PRAM processors, it follows that $f(C^+)$, and hence C^+, can be computed within the same bounds. A symmetrical argument shows that C^- can be computed within the desired complexity bounds. Combining this with the observation that computing F is easy given F^+ and F^-, gives us the following theorem:

Theorem 2 *Given a set $L = \{l_1, l_2, \ldots, l_n\}$ of oriented lines in the plane, one can compute $F = \cap_{i=1}^{n} H(l_i)$ in $O(\log n)$ time using n processors in the CREW PRAM model.*

We consider an application of this theorem in the next subsection.

3.1 The Kernel of a Simple Polygon

Let $P = (p_1, s_1, p_2, s_2, \ldots, s_{n-1}, p_n)$ be a simple polygon P (so $p_1 = p_n$). We consider each edge s_i of P to be *opaque*, i.e., an observer in the plane cannot see through it. We define the *kernel of* P, denoted $K(P)$, to be the set of points from each of which all of P's boundary is visible; that is, point p is in $K(P)$ if for every point q on the boundary of P, the segment joining p to q does not intersect any edge e of P except possibly at e's endpoints. Note that this implies that $K(P)$ is a subset of the interior of P. Also note that the kernel may be empty. If the kernel of a polygon P is not empty, then P is said to be *star-shaped*. The kernel of a convex polygon is, of course, the polygon itself. Figure 9 shows a star-shaped polygon and its kernel (shaded in the figure). Intuitively, if we imagine the polygon P to be the floor plan of an art gallery, then the kernel of P can be viewed as the region where a guard could stand and be able to see every painting on the walls of P.

It turns out that, as a consequence of the algorithm for computing the intersection of n half-planes, the kernel of a polygon P can be computed in $O(\log n)$ time with n processors. We consider each edge s_i of P to be an oriented line segment such that the interior of P is on its right. We let $H(s_i)$ denote the half-plane to the right of the oriented line containing the edge s_i. It is not hard to show (see Exercise 6) that the kernel of P is the intersection of the n half-planes determined by the edges in P, i.e., it is $\cap_{i=1}^{n} H(s_i)$. This implies that the kernel can be computed in $O(\log n)$ time by n processors on a CREW PRAM.

Figure 9: A polygon and its kernel.

The next section deals with the problem of computing the distance between two convex polygons. There too, success will hinge on a careful exploitation of convexity.

4 The Distance Between Two Convex Polygons

In the problem we address in this section the input consists of two convex polygons $P = (p_1, s_1, p_2, s_2, \ldots, s_{n-1}, p_n)$ and $Q = (q_1, t_1, q_2, t_2, \ldots, t_{n-1}, q_n)$, where the p_i's (resp., q_i's) are given in clockwise cyclic order, that is, the interior of P is to the right of each oriented segment $\overline{p_i p_{i+1}}$, and the similar property holds for Q. We are interested in computing the shortest distance between P and Q. This distance is formally defined as follows:

$$d(P, Q) \;=\; \min_{u \in P, w \in Q} \; d(u, w)$$

where $d(u, w)$ denotes the Euclidean distance between points u and w, and the notation "$u \in P$" means that u is a point of P (either on the boundary of P or interior to P). This problem often arises in machine learning problems where one wishes to determine if two sets of points A and B can be separated by a line, and if they can be so separated, then one wishes to determine how "close" A is to B. Constructing the convex hulls of A and B immediately reduces this learning problem to that of computing the distance between two convex polygons, or determining if they intersect.

The algorithm we present in this section actually returns a pair of points u, w such that $d(P, Q) = d(u, w)$. It runs in $O(c^2)$ time with $O(n^{1/c})$ processors on a CREW PRAM. We give the algorithm assuming that P and Q are disjoint, so that there is a line separating them. Exercise 17 deals with the case of possibly zero distance, i.e.,

when P and Q intersect, in which case there is no separating line. Of course, once we have these points, u and w, any perpendicular to the line segment joining u and w is a line separating P from Q. Therefore the algorithm given below for the closest distance can also be used to give us a separating line.

To simplify the exposition, we assume that no three successive vertices of either polygon are collinear, and that no edge of P is parallel to an edge of Q. The assumption that no edge of P is parallel to an edge of Q implies that the desired points u and w are unique. The algorithm can easily be modified for the case when edges of P might be parallel to edges of Q, e.g., by adopting a suitable convention for returning a unique u, w pair in case $d(P, Q)$ is the distance between two parallel edges of P and Q respectively. In this latter case there is an infinite number of choices for u and w, and this is the only case where u and w are not unique.

Let p be a point, which is not necessarily a vertex, on the boundary of P, and define q similarly for Q. Let T_p (resp., T_q) be the line perpendicular to the line segment $\overline{pq}$ at point p (resp. q). It is easy to show (see Exercise 16) that $d(P, Q) = d(p, q)$ if and only if (i) T_p and T_q are tangent to P and Q respectively, and (ii) P and Q are on opposite sides of the region between T_p and T_q. Note, conditions (i) and (ii) are "local" and can thus be tested by one processor in constant time for a given pair of points p and q.

The above simple observation implies that, with $O(n^2)$ processors and in constant time, it is possible to compute the closest distance between P and Q and a pair of points achieving it. The brute force procedure for doing this assigns a processor to each pair (a, b), where a is a vertex or edge of P, and b is a vertex or edge of Q (but not when both a and b are edges—that case need not be considered, as it is subsumed by one of the cases in which one of $\{a, b\}$ is a vertex). Such a processor then computes, in constant time, the points $a' \in a$ and $b' \in b$ such that $d(a', b') = d(a, b)$. It then tests, also in constant time, whether (i) $T_{a'}$ and $T_{b'}$ are tangent to (respectively) P and Q, and (ii) P and Q are on opposite sides of the region between $T_{a'}$ and $T_{b'}$ (i.e., this region separates them). If so then the processor decides that $u = a'$ and $w = b'$, and furthermore no other processor reaches such a decision about the identity of u and w (by the uniqueness of u and w). Thus there are no "write conflicts" when that processor writes the names of u and w in the specified registers for them.

The algorithm we shall give still takes constant time, but it uses far fewer processors than n^2: in fact it uses $O(n^{1/k})$ processors for any constant k of our choice. It relies on the above simple observations as well as on the next two propositions.

Proposition 3 *Let p be a point external to Q. Then the point $q \in Q$ such that $d(p,q) = d(p,Q)$ can be computed in time $O(k)$ by an $n^{1/k}$ processor CREW PRAM, where k is any integer of our choice.*

Proof. Let $l = n^{1-1/k}$. Let Q' consist of every l-th vertex of Q, i.e., Q' is the subpolygon $(q_l, s'_l, q_{2l}, s'_{2l}, \ldots, q_n)$. Since Q' has $n^{1/k}$ vertices and we have $n^{1/k}$ processors, it is trivial to find in constant time the point $q' \in Q'$ such that $d(p,q') = d(p,Q')$. (Note that q' need not be a vertex of Q'.) If the perpendicular to the line segment $\overline{pq'}$ at point q' is tangent to Q, then we can stop and declare point q' as the desired point q. Otherwise, let α (resp. β) be the vertex of Q' that immediately precedes (resp. follows) point q' when the boundary of Q' is traced in a clockwise manner (see Figure 10). Note that in Q, there are $2l+1$ vertices between α and β (inclusive) if q' is a vertex, and there are $l+1$ vertices between α and β otherwise. We leave it to the reader to prove that, in Q, the desired point q occurs between α and β (inclusive). Let γ be the median of the (at most $2l+1$) vertices between α and β (inclusive): Test whether the desired point q is at γ, between α and γ, or between γ and β. This test trivially takes constant time with one processor. If $q = \gamma$ then we're done, so suppose (without loss of generality) that the test reveals that q is between α and γ. If this occurs, then we can focus our search for q to the section of Q between α and γ (excluding γ), which contains at most l vertices. Therefore, by doing a constant amount of work, we have reduced the polygon size by a factor of at least $n^{1/k}$. Doing this at most k times finds the desired point q. ∎

Proposition 4 *Let p_i and p_j be any two vertices of P, $i < j$, and let p_u be the vertex of P such that $d(p_u, Q) = d(P, Q)$. Then for any integer k of our choice, an $n^{1/k}$ processor CREW PRAM can, in $O(k)$ time, locate where p_u occurs with respect to p_i and p_j in the sequence $p_1, p_2, \ldots, p_n$ (i.e., it can determine whether $u = i$, $u = j$, $i < u < j$, or none of these).*

Proof. For any two indices $1 \le a, b \le n$, let $\sigma_{a,b}$ denote the sequence

$$\sigma_{a,b} = d(p_a, Q), d(p_{a+1}, Q), \ldots, d(p_b, Q)$$

Figure 10: Illustrating the proof of Proposition 3.

(assuming index $n+1$ equals 1). For example,

$$\sigma_{9,2} = d(p_9, Q), \ldots, d(p_n, Q), d(p_1, Q), \ldots, d(p_2, Q).$$

Observe that, because of convexity, there exist two indices a and b, $1 \leq a \leq b \leq n$, such that $\sigma_{a,b}$ and $\sigma_{b,a}$ are both sorted, one in increasing order and the other in decreasing order. This implies that we can locate where p_u occurs with respect to any pair p_i, p_j in the sequence $p_1, p_2, \ldots, p_n$ by performing a constant number of distance computations of the type $d(p_l, Q)$. By Proposition 3, each such distance computation can be done within the desired time and processor bounds. ∎

The following algorithm shows that, for any integer c of our choice, an $n^{1/c}$ processor CREW PRAM can find, in $O(c^2)$ time, the points $u \in P$ and $w \in Q$ such that $d(u, w) = d(P, Q)$.

Algorithm D for computing distance:

Input. Two disjoint convex polygons $P = (p_1, s_1, p_2, s_2, \ldots, p_n)$ and $Q = (q_1, t_1, q_2, t_2, \ldots, q_n)$. The p_i's (resp., q_j's) are given in clockwise cyclic order.

Output. Points u, w such that $d(u, w) = d(P, Q)$.

1. Set $\hat{P} := P$, $\hat{Q} := Q$, $s := n^{1/2c}$.

2. Repeat the following steps 3–5 until either $\hat{P}$ is a single point or $\hat{Q}$ is a single point. Without loss of generality, assume it is $\hat{P}$ that ends up becoming a single point (call it x): Use Proposition 3 to find, in $O(c)$ time, the point $y \in Q$ such that $d(x, y) = d(x, Q)$. Output the points x and y (these are the desired points u, w).

Figure 11: Illustrating the algorithm for the distance.

3. Let $P' = (a_1, s'_1, a_2, s'_2, \ldots, a_s)$ be the subpolygon obtained by considering every $(|\hat{P}|/l)$-th vertex of $\hat{P}$, i.e. the l vertices of P' divide $\hat{P}$ into l equal portions. Call these portions A_1, A_2, ..., A_l, so that a_i is adjacent in $\hat{P}$ to portions A_i and A_{i+1} By definition, a_i belongs to A_i but not to A_{i+1} Let $Q' = (b_1, t'_1, b_2, t'_2, \ldots, b_l)$ be analogously defined for $\hat{Q}$, and let the resulting portions of $\hat{Q}$ be B_1, B_2, ..., B_l. Use the already mentioned brute force method for finding the points $a \in P'$ and $b \in Q'$ such that $d(a, b) = d(P', Q')$. Since we have l^2 processors, this takes constant time.

   Let α_P be the vertex of P' that immediately precedes a on the boundary of P', and let β_P be the vertex of P' that immediately follows a on the boundary of P'. (Figure 11 illustrates the case when a is not a vertex of P'.) If a is a vertex of P' then α_P and β_P are (respectively) its predecessor and successor vertices on P', and hence there are then $(2|\hat{P}|/l) + 1$ vertices of $\hat{P}$ between α_P and β_P (inclusive). If a is not a vertex of P' then α_P and β_P are consecutive vertices of P', point a is on the segment of P' that joins α_P to β_P, and there are $(|\hat{P}|/l)+1$ vertices of $\hat{P}$ between α_P and β_P (inclusive). Let γ_P be the median of the (at most $(2|\hat{P}|/l) + 1$) vertices of $\hat{P}$ that are between α_P and β_P (inclusive). (Note that, if a is a vertex of P', then $\gamma_P = a$.) We use $P^{\alpha\beta}$ to denote the portion of P that is between α_P and β_P (excluding α_P and β_P). $P^{\alpha\gamma}$ and $P^{\gamma\beta}$ are analogously defined.

   Let α_Q, β_Q, γ_Q, $Q^{\alpha\beta}$, $Q^{\alpha\gamma}$ and $Q^{\gamma\beta}$ be similarly defined for b, Q' and $\hat{Q}$. (Figure 11 illustrates the case when b is a vertex of Q'.)

4. Use Proposition 4 to detect whether $u = \alpha_P$, $u = \beta_P$, $u = \gamma_P$, $u \in P^{\alpha\gamma}$, $u \in P^{\gamma\beta}$, or none of these. If u equals α_P (resp. γ_P, β_P) then set $\hat{P}$ equal to α_P (resp., γ_P, β_P) and go to Step 5. Otherwise, if $u \in P^{\alpha\gamma}$ then do $\hat{P} := P^{\alpha\gamma}$ and go to Step 5. Otherwise, if $u \in P^{\gamma\beta}$ then do $\hat{P} := P^{\gamma\beta}$ and go to Step 5. Otherwise leave $\hat{P}$ unchanged. (An assignment like $\hat{P} := P^{\alpha\gamma}$ is done in constant time simply by remembering the new first and last vertex of $\hat{P}$.)

5. Use Proposition 4 to detect whether $w = \alpha_Q$, $w = \beta_Q$, $w = \gamma_Q$, $w \in Q^{\alpha\gamma}$, $w \in Q^{\gamma\beta}$, or none of these. If w equals α_Q (resp. γ_Q, β_Q) then set $\hat{Q}$ equal to α_Q (resp. γ_Q, β_Q) and go to Step 3. Otherwise, if $w \in Q^{\alpha\gamma}$ then do $\hat{Q} := Q^{\alpha\gamma}$ and go to Step 3. Otherwise, if $w \in Q^{\gamma\beta}$ then do $\hat{Q} := Q^{\gamma\beta}$ and go to Step 3. Otherwise leave $\hat{Q}$ unchanged.

Since every usage of Proposition 4 takes $O(c)$ time, the time complexity of the algorithm is equal to c multiplied by the number of times that steps 2-4 get executed. We now bound the number of times steps 2-4 are executed.

Lemma 4.1 *Let a, b, $P^{\alpha\beta}$, $Q^{\alpha\beta}$, u, and w be as in algorithm* **D**. *Assume that $u \notin \{\alpha_P, \beta_P\}$ and $w \notin \{\alpha_Q, \beta_Q\}$. Then at least one of the following statements (1) or (2) is true:*

1. $u \in P^{\alpha\beta}$,

2. $w \in Q^{\alpha\beta}$.

Proof. Let T_a be the line perpendicular at a to the segment ab, and let T_b be the line perpendicular at b to the segment ab (see Figure 11). By definition, T_a is tangent to P', and T_b is tangent to Q'. Without loss of generality, T_a and T_b are vertical, P' is to the left of T_a, and Q' is to the right of T_b. If $u = a$ or $w = b$ then the lemma holds, so suppose that $u \neq a$ and $w \neq b$. By the definition of u and w, we must have $d(u, w) \leq d(a, b)$. This implies that u is to the right of T_a or w is to the left of T_b (possibly both). Hence, it suffices to show that if u is to the right of T_a, then (1) holds, and if w is to the left of T_b, then (2) holds. We prove this by contradiction. Suppose that u is to the right of T_a and (1) does not hold, i.e. $u \notin P^{\alpha\beta}$. Without loss of generality, assume that u is below γ_P. Now, by walking from vertex γ_P clockwise

along the boundary of $\hat{P}$, one encounters vertex β_P before reaching u. Since γ_P is on or to the right of T_a, β_P on or to the left of T_a, and u to the right of T_a, this contradicts the convexity of P. A similar argument shows that if w is to the left of T_b, then (2) holds. ∎

Corollary 4.2 *Steps 3–5 of algorithm* **D** *are executed a total of at most* $4c-1$ *times.*

Proof. Lemma 2 implies that, every time we execute Steps 3–5, at least one of $\hat{P}$ or $\hat{Q}$ decreases in size by a factor of at least $s = n^{1/2c}$, thus proving the corollary. ∎

We have thus established the following:

Theorem 3 *Algorithm* **D** *correctly finds points* $u \in P$ *and* $w \in Q$ *such that* $d(u,w) = d(P,Q)$. *It uses* $n^{1/c}$ *processors and it runs in time* $O(c^2)$, *where* c *is any integer such that* $2 \leq c \leq \log n$.

Before proceeding to our next problem (3-dimensional maxima), we note that there is another version of the definition of the distance: the *boundary-to-boundary* distance, in which "$u \in P$" means that u is a point of the *boundary* of P. Interestingly, if we use this notion of distance, then determining the distance between two convex polygons has an $\Omega(\log n)$ *lower bound*, as the following shows.

Theorem 4 *The problem of computing the shortest boundary-to-boundary distance between two convex polygons* P *and* Q *has an* $\Omega(\log n)$ *lower bound on a CREW PRAM having a polynomial number of processors.*

Proof. We use the fact that there is a known $\Omega(\log n)$ lower bound for the problem of computing the logical OR of n bits $x_1, x_2, \ldots, x_n$. Therefore it suffices to show that a CREW PRAM can compute the logical OR of n bits in time equal to a constant plus the time for computing the distance between the boundaries of P and Q. We do this by converting the problem of computing the OR of $x_1, x_2, \ldots, x_n$ to that of computing the distance between the boundaries of the following two convex n-gons P and Q. Let P be any regular convex n-gon, so that all its edges have the same length. Let Q_0 consist of the regular convex n-gon whose vertices are the midpoints of the n boundary edges of P. The convex n-gon Q is obtained from Q_0 by "deforming" Q_0 very slightly so as to move some of its vertices slightly into the interior of P,

and others just outside of P. This deformation is governed by the Boolean values $x_1, x_2, \ldots, x_n$: we associate the i-th vertex, q_i, of Q_0 with x_i, and move q_i slightly inside of P if $x_i = 0$ and move q_i slightly outside of P if $x_i = 1$. The polygon Q so obtained has the property that its boundary is at a distance of zero from the boundary of P if and only if the logical OR of the x_i's is 1. The construction of P and Q from $x_1, x_2, \ldots, x_n$ can clearly be done in constant time with n processors. This establishes the lower bound. ∎

The rest of this chapter illustrates how the technique presented in the earlier chapter on parallel mergesorting, can be used to solve geometric problems.

5 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 denote the x, y, and z coordinates of a point p by $x(p)$, $y(p)$, and $z(p)$, respectively. 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.

5.1 Maximal Elements

The method is based on a divide-and-conquer strategy in which the subproblem merging step involves the computation of *two* labeling functions for each point, but is otherwise similar to the parallel mergesort algorithm described earlier in this book. We call such a divide-and-conquer scheme a *cascading divide-and-conquer*, since it involves a cascading merge. Specifically, for each point p_i we compute the maximum z-coordinate from among all points that 1-dominate p_i and use that label to also compute the maximum z-coordinate from among all points that 2-dominate p_i. We can then test if p_i is a maximum point by comparing $z(p_i)$ to this latter label. The details follow.

Without loss of generality, we assume the input points are given sorted by increasing y-coordinates, i.e., $y(p_i) < y(p_{i+1})$, since if they are not given in this order we can sort them in $O(\log n)$ time using n processors. Let T be a complete binary tree with leaf nodes $v_1, v_2, \ldots, v_n$ (in this order). In each leaf node v_i we store the list $B(v_i) = (-\infty, p_i)$, where $-\infty$ is a special symbol such that $x(-\infty) < x(p_j)$ and $y(-\infty) < y(p_j)$ for all points p_j in V. Initializing T in this way can be done in $O(\log n)$ time using n processors. We then perform a cascading merge from the leaves of T upwards, basing comparisons on increasing x-coordinates of the points (not their y-coordinates). Let $U(v)$ denote the sorted array of the points stored in the descendants of $v \in T$ sorted by increasing x-coordinates. For each point p_i in $U(v)$ we store two labels: $zod(p_i, v)$ and $ztd(p_i, v)$, where $zod(p_i, v)$ is the largest z-coordinate of the points in $U(v)$ that 1-dominate p_i, and $ztd(p_i, v)$ is the largest z-coordinate of the points in $U(v)$ that 2-dominate p_i. Initially, *zod* and *ztd* labels are only defined for the leaf nodes of T. That is, $zod(p_i, v_i) = ztd(p_i, v_i) = -\infty$ and $zod(-\infty, v_i) = ztd(-\infty, v_i) = z(p_i)$ for all leaf nodes v_i in T (where $U(v_i) = (-\infty, p_i)$). In order to be more explicit in how we refer to various ranks, we let $\text{pred}(p_i, v)$ denote the predecessor of p_i in $U(v)$ (which would be $-\infty$ if the x-coordinates of the input points are all larger than $x(p_i)$). (See Figure 12.) As we are performing the merge, we update the labels *zod* and *ztd* based on the equations in the following lemma:

Lemma 5.1 *Let p_i be an element of $U(v)$ and let $u = lchild(v)$ and $w = rchild(v)$. Then we have the following:*

$$zod(p_i, v) = \begin{cases} \max\{zod(p_i, u),\ zod(\text{pred}(p_i, w), w)\} & \text{if } p_i \in U(u) \\ \max\{zod(\text{pred}(p_i, u), u),\ zod(p_i, w)\} & \text{if } p_i \in U(w) \end{cases} \quad (1)$$

$$ztd(p_i, v) = \begin{cases} \max\{ztd(p_i, u),\ zod(\text{pred}(p_i, w), w)\} & \text{if } p_i \in U(u) \\ ztd(p_i, w) & \text{if } p_i \in U(w) \end{cases} \quad (2)$$

Proof: Consider Equation 1. If $p_i \in U(u)$, then every point that 1-dominates p_i's predecessor in $U(w)$ also 1-dominates p_i, since p_i's predecessor in $U(w)$ is the point with largest x-coordinate less than $x(p_i)$ (or $-\infty$ if every point in $U(w)$ has larger x-coordinate than p_i). Thus $zod(p_i, v)$ is the maximum of $zod(p_i, u)$ and

Figure 12: The combining step for 3-dimensional maxima Points to the right of the thin dotted line 1-dominate p_i (resp. p_j), and points enclosed in the thick dotted lines 2-dominate p_i (p_j).

$zod(\text{pred}(p_i, w), w)$ in this case. The case when $p_i \in U(w)$ is similar. Next, consider Equation 2. We know that every point in $U(w)$ has y-coordinate greater than every point in $U(u)$, by our construction of T. Therefore, if $p_i \in U(u)$, then every point in $U(w)$ that 1-dominates p_i's predecessor in $U(w)$ must 2-dominate p_i. Thus, $ztd(p_i, v)$ is the maximum of $ztd(p_i, u)$ and $zod(\text{pred}(p_i, w), w)$. On the other hand, if $p_i \in U(w)$ then no point in $U(u)$ can 2-dominate p_i; thus, $ztd(p_i, v) = ztd(p_i, w)$. ∎

We use these equations during the cascading merge to maintain the labels for each point. By Lemma 5.1, when v becomes full (and we have $U(u)$, $U(w)$, and $U(v) = U(u) \cup U(w)$ available), we can determine the labels for all the points in $U(v)$ in $O(1)$ additional time using $|U(v)|$ processors. Thus, the running time of the cascading merge algorithm, even with these additional label computations, is still $O(\log n)$ using n processors. Moreover, after v's parent becomes full we no longer need $U(v)$ any more, and can deallocate the space it occupies, resulting in an $O(n)$ space algorithm. After we complete the merge, and have computed $U(root(T))$, along with all the labels for the points in $U(root(T))$, note that a point $p_i \in U(root(T))$ is a maximum if and only if $ztd(p_i, root(T)) \leq z(p_i)$ (there is no point that 2-dominates p_i and has z-coordinate greater than $z(p_i)$). Thus, after completing the cascading merge we can construct the set of maxima by compressing all the maximum points into one contiguous list using a simple parallel prefix computation. We summarize in

Figure 13: The shaded region is visible from p.

the following theorem:

Theorem 5 *Given a set V of n points in $\Re^3$, one can construct the set M of maxima points in V in $O(\log n)$ time and $O(n)$ space using n processors in the CREW PRAM model.*

6 Visibility from a Point

Given a set of line segments $S = \{s_1, s_2, \ldots, s_n\}$ in the plane that do not intersect, except possibly at endpoints, and a point p, the visibility from a point problem is to determine the part of the plane that is visible from p assuming every s_i is opaque. Intuitively, one can think of the point p as a specular light source, the segments as walls, and the problem to determine all the parts of the plane that are illuminated (see Figure 13).

We can use the cascading divide-and-conquer technique to solve this problem in $O(\log n)$ time and $O(n)$ space using n processors. Without loss of generality, we assume that the point p is at negative infinity below all the segments. The algorithm is essentially the same if p is a finite point, except that the notion of segment endpoints being ordered by x-coordinate is replaced by the notion that they are ordered radially around p. In other words, it suffices to compute the *lower envelope* of the n segments to give a method for computing the visibility from a point. For simplicity of expression, we also assume that the x-coordinates of the endpoints are distinct.

In the previous section the set of objects consisted of points, but in the visibility

Figure 14: An example of visibility merging. The dashed segments correspond to the visible region for $x(u)$ and the solid segments correspond to the visible region for $x(w)$. For simplicity, we store the pointers $pred(p_i, u)$, and $pred(p_i, w)$ in arrays and 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).

problem we are dealing with line segments. The method is slightly different in this case. In this case we store the segments in the leaves of a binary tree and perform a cascading merge of the x-coordinates of intervals of the x-axis determined by segment endpoints. We maintain a single label for each interval that represents the segment that is visible from $-\infty$ on that interval. The details follow.

Let T be a complete binary tree with leaf nodes $v_1, v_2, \ldots, v_n$ ordered from left to right. We associate the segment s_i with the leaf v_i and at v_i store the list $U(v_i) = (-\infty, p_1, p_2)$, where p_1 and p_2 are the two endpoints of s_i, with $x(p_1) < x(p_2)$, and $-\infty$ is defined such that $x(-\infty) < x(p)$ and $y(-\infty) < y(p)$ for all points p. We then perform a generalized cascading-merge from the leaves of T, basing comparisons on increasing x-coordinates of the points. For each internal node v we let $U(v)$ denote an array of the points stored in the descendants of $v \in T$ sorted by increasing x-coordinates. For each point p_i in $U(v)$ we store a label $vis(p_i, v)$, which stores the segment with endpoints in $U(v)$ that is visible in the interval $(x(p_i), x(\text{succ}(p_i, v)))$, where $\text{succ}(p_i, v)$ denotes the successor of p_i in $U(v)$ (based on x-coordinates). Initially, the vis labels are only defined for the leaf nodes of T. That is, if $U(v) = (-\infty, p_1, p_2)$, where $s_i = p_1p_2$, then $vis(-\infty) = +\infty$, $vis(p_1) = s_i$, and $vis(p_2) = +\infty$. We use $\text{pred}(p_i, v)$ to denote the predecessor of p_i in $U(v)$. As we are performing the cascading-merge, we update the vis labels based on the equation in the following lemma (see Figure 14):

Lemma 6.1 *Let p_i be an element of $U(v)$ and let $u = lchild(v)$ and $w = rchild(v)$. Then we have the following (if two segments s_i and s_j are comparable by the "above" relation, then we let $\min\{s_i, s_j\}$ denote the lower of the two):*

$$vis(p_i, v) \;=\; \begin{cases} \min\{vis(p_i, u),\ vis(\text{pred}(p_i, w), w)\} & \text{if } p_i \in U(u) \\ \min\{vis(\text{pred}(p_i, u), u),\ vis(p_i, w)\} & \text{if } p_i \in U(w) \end{cases}$$

Proof: If we restrict our attention to the segments with an endpoint in $U(u)$, then for any point $p_i \in U(u)$ the segment visible (from $-\infty$) on the interval $(x(p_i), x(\text{succ}(p_i, v)))$ is the minimum of the segment visible on the interval $(x(p_i), x(\text{succ}(p_i, u)))$ and the segment that is visible on the interval $(x(\text{pred}(p_i, w)), x(\text{succ}(\text{pred}(p_i, w), w)))$. This is because the interval $(x(p_i), x(succ(p_i, v)))$ is exactly the intersection of the interval $(x(p_i), x(succ(p_i, u)))$

and the interval $(x(\text{pred}(p_i, w)), x(\text{succ}(\text{pred}(p_i, w), w)))$, and there is no segment in $U(v)$ with an endpoint interior to the interval $(x(p_i), x(succ(p_i, v)))$. Thus, $vis(p_i, v)$ is equal to the minimum of $vis(p_i, u)$ and $vis(\text{pred}(p_i, w), w)$. The case when $p_i \in U(v)$ is similar. ∎

By Lemma 6.1, after merging the lists $U(lchild(v))$ and $U(rchild(v))$ we can determine the labels for all the points in $U(v)$ in $O(1)$ additional time using $|U(v)|$ processors. Thus, the running time of this generalized cascading-merge algorithm is still $O(\log n)$ using n processors. After we complete the merge, and have computed $U(root(T))$, along with all the vis labels for the points in $U(root(T))$, then we can compress out duplicate entries in the list $(vis(p_1, root(T)), vis(p_2, root(T)), \ldots, vis(p_{2n}, root(T)))$ using a parallel prefix computation to construct a compact representation of the visible portion of the plane. We summarize in the following theorem:

Theorem 6 *Given a set S of n non-intersecting segments in the plane, one can find the lower envelope of S in $O(\log n)$ time and $O(n)$ space using n processors in the CREW PRAM model, and this is optimal.*

Proof: The correctness and complexity bounds follow from the discussion above. Since we require that the points in the description of the lower envelope be given by increasing x-coordinates, we can reduce sorting to this problem, and thus can do no better than $O(\log n)$ time using n processors (in the comparison model). ∎

7 Exercises

Exercise 1 *Prove an $O(n \log n)$ time sequential upper bound for the convex hull problem.*

Exercise 2 *Prove an $\Omega(n \log n)$ time sequential lower bound for the convex hull problem in the comparison model. Hint: find a way to reduce the sorting problem to the convex hull problem.*

Exercise 3 *Prove an $\Omega(\log n)$ time lower bound for the convex hull problem and a CREW PRAM with a polynomial number of processors.*

Exercise 4 *Prove that the intersection of two convex regions is also convex. Also, give an $O(\log n)$ time algorithm, using $O(n/\log n)$ processors in the CREW PRAM model, for computing the intersection of two convex regions. Your method should work even if either of the regions is unbounded. (Hint: use the fact that one can merge two n-element sorted lists in $O(\log n)$ time using $O(n/\log n)$ processors in the CREW PRAM model.)*

Exercise 5 *Let $P = (p_1, s_1, p_2, \ldots, p_n)$ be a simple polygon, and let $P' = (p_{i_1}, s'_{i_1}, p_{i_2}, \ldots, p_{i_k})$ be a subpolygon of P, i.e., $1 \leq i_1 < i_2 < \cdots < i_k \leq n$. Prove or disprove the following statements:*

1. *If P is simple and convex, then P' is simple and convex.*
2. *If P is simple, then P' is simple.*
3. *If P is simple and star-shaped, then P' is simple and star-shaped.*

Exercise 6 *Prove that the two definitions of "kernel" given in Section 3.1 are equivalent.*

Exercise 7 *Let P be a star-shaped polygon, P' be a subpolygon of P. Prove or disprove the following statement: "the kernel of P' is inside the kernel of P."*

Exercise 8 *Let p, Q', q_j and q_{im} be as in the proof of Proposition 1, with $j < im$. Prove that $(i-1)m \leq j$.*

Exercise 9 *Generalize Proposition 2 to arbitrary convex polygons P and Q, rather than just upper hulls.*

Exercise 10 *Generalize the algorithm for finding the upper common tangent between $UH(S_1)$ and $UH(S_2)$ so that it runs in $O(k^2)$ time and uses $O(n^{1/k})$ processors, where k is a constant.*

Exercise 11 *Design an algorithm for determining whether a given line L and the boundary of a given convex polygon P intersect or not. If they do intersect, your algorithm should give both points of this intersection, and your algorithm should run in $O(k)$ time with $O(n^{1/k})$ processors, where $2 \leq k \leq n$ is any integer.*

Exercise 12 *Let S be a set of points in the plane. We wish to "mark" the points of S that belong to the convex hull of S. Give a constant-time CRCW algorithm for this problem, using a polynomial number of processors.*

Exercise 13 *Let S be a set of points in the plane, given in sorted order by their x-coordinates. We wish to "mark" each point of S that belongs to the upper hull of S as "on hull", and also mark each each point p that is not on the upper hull of S with the name of the upper hull edge that is directly above p. Derive an $O(\log \log n)$ time algorithm that uses only n processors. (Hint: use the fact that the maximum of n items can be found in constant time using n^2 processors in the CRCW PRAM model.)*

Exercise 14 *Let f be the transform defined in Section 3. Prove that f maintains the relative positions of lines and points: if one of $\{\alpha, \beta\}$ is a line and one is a point, then α is below β if and only if $f(\alpha)$ is below $f(\beta)$.*

Exercise 15 *Let f be the transform defined in Section 3, as in the previous exercise. Prove or disprove the following statement: "If P is a simple polygon, then $f(P)$ is a simple polygon."*

Exercise 16 *Let p be a point on the boundary of a convex polygon P, and define q similarly for a convex polygon Q. Note, p and q need not be vertices. Let T_p (resp. T_q) be the line perpendicular to the line segment $\overline{pq}$ at p and q, respectively. Prove that $d(P, Q) = d(p, q)$ if and only if (i) T_p and T_q are tangent to P and Q respectively, and (ii) P and Q are on opposite sides of the region between T_p and T_q.*

Exercise 17 *Modify the algorithm for computing the distance between two convex polygons so that it also works if they are not disjoint. That is, the distance between them may be zero (this happens either by one of them being contained inside the other, or by their two boundaries intersecting).*

Exercise 18 *Show that 2-dimensional maxima problem can be solved in $O(\log n)$ time and $O(n)$ space by a sorting step followed by a parallel prefix step.*

8 Bibliographic Notes

The convex hull problem is probably the most-studied of all computational geometry problems. See the book by Edelsbrunner [33] or the book by Preparata and Shamos [62] for further references on this and other sequential algorithms for computational geometry. A proof of the Jordan Curve Theorem can be found in the book by Munkres [57]. The convex hull algorithm presented in this chapter is due to Atallah and Goodrich [12], who also give the algorithm for computing the distance between convex polygons. The solutions to Exercises 10 and 17 can be found in [12]. Dadoun and Kirkpatrick [31] show how to achieve $O(k)$ time using $O(n^{1/k})$ processors for the polygon separation problem, which improves the running time for larger values of k. Other efficient parallel convex hull algorithms can be found in [11, 1, 56, 80], and a solution to Exercise 13 can be found in [17]. Although it is a good example of applying geometric duality in parallel computational geometry, the algorithm given in this chapter for computing the kernel of a simple polygon is not the best possible, for Cole and Goodrich [25] show how to solve this problem in $O(\log n)$ time algorithm for using only $O(n/\log n)$ processors in the EREW PRAM model. For more information about this problem and other "art gallery" problems, see the excellent book by O'Rourke [58]. The algorithm for the 3-dimensional maxima and visibility from a point is due to Atallah, Cole, and Goodrich [10]; they also give a number of other geometric applications of the cascading divide-and-conquer technique. Recently, Atallah and Chen [9] show how one can improve the processor bound for visibility from a point to $O(n/\log n)$ for the case when the segments form a simple polygon. Exercise 12 is from Akl [4].

The field of parallel computational geometry is relatively young; hence, the body of literature in this field is rather sparse. Still, the bibliography given below attempts to cover most of the papers covering this area to date. It is by no means exhaustive, however, and, if fact, omits a number of papers dealing with geometric algorithms for network models (these network algorithms are outside the scope of this chapter, but are covered in a book by Miller and Stout [55], which also contains an extensive bibliography of geometric algorithms for network models).

References

[1] A. Aggarwal, B. Chazelle, L. Guibas, C. Ó'Dúnlaing, and C. Yap, "Parallel Computational Geometry," *Algorithmica*, Vol. 3, No. 3, 1988, pp. 293–328.

[2] A. Aggarwal, D. Kravets, J. K. Park, and S. Sen, "Parallel Searching in Generalized Monge Arrays with Applications," *Proc. 2nd ACM Symp. on Parallel Algorithms and Architectures*, 1990, pp. 259–267.

[3] A. Aggarwal and J. Park, "Parallel Searching in Multidimensional Monotone Arrays," *J. Algorithms*, 1989.

[4] S.G. Akl, "A Constant-Time Parallel Algorithm for Computing Convex Hulls," *BIT*, Vol. 22, 1982, pp. 130–134.

[5] N. Alon and N. Megiddo, "Parallel Linear Programming in Fixed Dimension Almost Surely in Constant Time," *Proc. 31st IEEE Symp. on Foundations of Computer Science*, 1990, pp. 574–582.

[6] R. Anderson, P. Beame, and E. Brisson, "Parallel Algorithms for Arrangements," *Proc. 2nd ACM Symp. on Parallel Algorithms and Architectures*, 1990, pp. 298–306.

[7] A. Apostolico, M. J. Atallah, L. L. Larmore, and H. S. McFaddin, "Efficient Parallel Algorithms for String Editing and Related Problems," *Proc. 26th Allerton Conf. on Communication, Control, and Computing*, 1988, pp. 253–263.

[8] M. J. Atallah, P. Callahan, and M. T. Goodrich, "P-Complete Geometric Problems," *Proc. 2nd ACM Symp. on Parallel Algorithms and Architectures*, 1990, pp. 317–326.

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

[10] M. J. Atallah, R. Cole, and M. T. Goodrich, "Cascading Divide-and-Conquer: A Technique for Designing Parallel Algorithms," *SIAM Journal on Computing*, Vol. 18, No. 3, 1989, pp. 499–532.

[11] M. J. Atallah and M. T. Goodrich, "Efficient Parallel Solutions to Some Geometric Problems," *Journal of Parallel and Distributed Computing*, Vol. 3, No. 4, 1986, pp. 492–507.

[12] M. J. Atallah and M. T. Goodrich, "Parallel Algorithms for Some Functions of Two Convex Polygons," *Algorithmica*, Vol. 3, 1988, pp. 535–548.

[13] M. J. Atallah and J. J. Tsay, "On the Parallel Decomposability of Geometric Problems," *Proc. 5th ACM Symp. on Computational Geometry*, 1989, pp. 104–113.

[14] M. Ben-Or, "Lower Bounds for Algebraic Computation Trees," *Proc. 15th ACM Symp. on Theory of Computing*, 1983, pp. 80–86.

[15] J. L. Bentley and M. I. Shamos, "Divide-And-Conquer in Multidimensional Space," *Proc. 8th ACM Symp. on Theory of Computing*, 1976, pp. 220–230.

[16] J. L. Bentley and D. Wood, "An Optimal Worst Case Algorithm for Reporting Intersections of Rectangles," *IEEE Trans. on Computers*, Vol. C-29, No. 7, 1980, pp. 571–576.

[17] O. Berkman, D. Breslauer, Z. Galil, B. Schieber, U. Vishkin, "Highly Parallizable Problems," *Proc. 21st ACM Symp. on Theory of Computing*, 1989, pp. 309–319.

[18] G. Bilardi and A. Nicolau, "Adaptive Bitonic Sorting: An Optimal Parallel Algorithm for Shared Memory Machines," TR 86-769, Dept. of Computer Science, Cornell Univ., August 1986.

[19] A. Borodin and J. E. Hopcroft, "Routing, Merging, and Sorting on Parallel Models of Computation," *Journal of Computer and System Sciences*, Vol. 30, No. 1, 1985, pp. 130–145.

[20] R. P. Brent, "The Parallel Evaluation of General Arithmetic Expressions," *J. ACM*, Vol. 21, No. 2, 1974, pp. 201–206.

[21] B. M. Chazelle and D. P. Dobkin. Detection is Easier than Computation, *Proc. 12th ACM Annual Symp. on Theory of Computing,* 1980, pp. 146–153.

[22] B. Chazelle and L. J. Guibas, "Fractional Cascading: I. A Data Structuring Technique," *Algorithmica*, Vol. 1, No. 2, pp. 133–162.

[23] A. Chow, "Parallel Algorithms for Geometric Problems," Ph.D. thesis, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, 1980.

[24] R. Cole, "Parallel Merge Sort," *SIAM J. Comput.*, Vol. 17, No. 4, August 1988, pp. 770–785.

[25] R. Cole and M. T. Goodrich, "Optimal Parallel Algorithms for Point-Set and Polygon Problems", to appear in *Algorithmica* (appeared in preliminary form in *Proc. 4th ACM Symp. on Computational Geometry*, 1988, pp. 201–210).

[26] R. Cole, M. T. Goodrich, C. Ó Dúnlaing, "Merging Free Trees in Parallel for Efficient Voronoi Diagram Construction," *Proc. 17th International Conf. on Automata, Languages, and Programming*, 1990, 432–445.

[27] R. Cole and O. Zajicek, "An Optimal Parallel Algorithm for Building a Data Structure for Planar Point Location," *J. Parallel and Distributed Computing*, Vol. 8, 1990, 280–285.

[28] S. Cook and C. Dwork. Bounds on the Time for Parallel RAM's to Compute Simple Functions, *Proc. 14th ACM Annual Symp. on Theory of Computing,* 1982, pp. 231–233.

[29] N. Dadoun and D. Kirkpatrick, "Parallel Construction of Subdivision Hierarchies," *Journal of Computer and System Sciences*, Vol. 39, 1989, pp. 153–165.

[30] N. Dadoun and D. Kirkpatrick, "Cooperative Subdivision Search Algorithms with Applications," *Proc. 27th Allerton Conf. on Communication, Control, and Computing*, 1989.

[31] N. Dadoun and D. Kirkpatrick, "Optimal Parallel Algorithms for Convex Polygon Separation," Technical Report 89-21, Dept. of Computer Science, Univ. of British Columbia, 1989.

[32] H. Edelsbrunner. Computing the Extreme Distances between Two Convex Polygons, *J. Algorithms,* Vol. 6, 1985, pp. 213–224.

[33] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer-Verlag, NY, 1987.

[34] H. Edelsbrunner and M. H. Overmars, "On the Equivalence of Some Rectangle Problems," *Information Processing Letters*, Vol. 14, No. 3, 1982, pp. 124–127.

[35] H. ElGindy, "A Parallel Algorithm for Triangulating Simplical Point Sets in Space with Optimal Speed-up," *Proc. 24th Allerton Conf. on Communication, Control, and Computing*, 1986.

[36] H. ElGindy and M. T. Goodrich, "Parallel Algorithms for Shortest Path Problems in Polygons," *The Visual Computer: International Journal of Computer Graphics*, Vol. 3, No. 6, 1988, pp. 371–378.

[37] A. Fournier and Z. Kedem, "Comments on the All Nearest-Neighbor Problem for Convex Polygons," *Info. Proc. Letters*, Vol. 9, No. 3, 1979, pp. 105–107.

[38] M. T. Goodrich, "Efficient Parallel Techniques for Computational Geometry," Ph.D. thesis, Department of Computer Science, Purdue University, 1987.

[39] M. T. Goodrich, "Finding the Convex Hull of a Sorted Point Set in Parallel," *Information Processing Letters*, Vol. 26, 1987, pp. 173–179.

[40] M. T. Goodrich, "Intersecting Line Segments in Parallel with an Output-Sensitive Number of Processors," *SIAM Journal on Computing*, to appear (appeared in preliminary form in *Proc. 1989 ACM Symp. on Parallel Algorithms and Architectures*, 127–137).

[41] M. T. Goodrich, "Triangulating a Polygon in Parallel," *J. Algorithms*, *Journal of Algorithms*, Vol. 10, 1989, pp. 327–351.

[42] M. T. Goodrich, "Constructing Arrangements Optimally in Parallel," Technical Report 90/06, Dept. of Computer Science, Johns Hopkins Univ., 1990.

[43] M. T. Goodrich, M. Ghouse, and J. Bright, "Generalized Sweep Methods for Parallel Computational Geometry," *Proc. 2nd ACM Symp. on Parallel Algorithms and Architectures*, 1990, pp. 280–289.

[44] M. T. Goodrich, C. Ó'Dúnlaing, and C. Yap "Computing the Voronoi Diagram of a Set of Line Segments in Parallel," *Algorithmica*, to appear (appeared in preliminary form in *Lecture Notes in Computer Science: 382, Algorithms and Data Structures, WADS '89*, Springer-Verlag, 1989, pp. 12–23).

[45] M. T. Goodrich, S. Shauck, and S. Guha, "Parallel Methods for Visibility and Shortest Path Problems in Simple Polygons," *Proc. 6th ACM Symp. on Computational Geometry*, 1990, pp. 73–82.

[46] L. Guibas, L. Ramshaw, and J. Stolfi, "A Kinetic Framework for Computational Geometry," *Proc. 24th IEEE Symp. on Found. of Computer Science*, 1983, pp. 100–111.

[47] T. Hagerup, H. Jung, and E. Welzl, "Efficient Parallel Computation of Arrangements of Hyperplanes in d Dimensions," *Proc. 2nd ACM Symp. on Parallel Algorithms and Architectures*, 1990, pp. 290–297.

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

[49] C. P. Kruskal, L. Rudolph, and M. Snir, "The Power of Parallel Prefix," *Proc. 1985 IEEE Int. Conf. on Parallel Processing*, St. Charles, IL., pp. 180–185.

[50] R. E. Ladner and M. J. Fischer, "Parallel Prefix Computation," *J. ACM*, 1980, pp. 831–838.

[51] D. T. Lee and F. P. Preparata, "The All Nearest-Neighbor Problem for Convex Polygons," *Info. Proc. Letters*, Vol. 7, No. 4, 1978, pp. 189–192.

[52] D. T. Lee and F. P. Preparata, "An Optimal Algorithm for Finding the Kernel of a Polygon," *J. ACM*, Vol. 26, No. 3, 1979, pp. 415–421.

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

[54] E. Merks, "An Optimal Parallel Algorithm for Triangulating a Set of Points in the Plane," Technical Report TR 86-9, Simon Fraser Univ., Burnaby, British Columbia, 1986.

[55] R. Miller and Q.F. Stout, *Parallel Algorithms for Regular Architectures,* The MIT Press, Cambridge, Massachusetts, 1989.

[56] R. Miller and Q.F. Stout, "Efficient parallel convex hull algorithms," *IEEE Trans. Computers* **C-37** (1988), pp. 1605-1618.

[57] J. R. Munkres, *Topology: A First Course*, Prentice-Hall, Inc. (Englewood Cliffs, NJ: 1975).

[58] J. O'Rourke, *Art Gallery Theorems and Algorithms*, Oxford University Press (New York: 1987).

[59] M. H. Overmars and J. Van Leeuwen, "Maintenance of Configurations in the Plane," *Journal of Computer and Systems Sciences,* Vol. 23, 198l, pp. 166–204.

[60] F. P. Preparata and S. J. Hong, "Convex Hulls of Finite Sets of Points in Two and Three Dimensions," *CACM,* Vol. 20, No. 2, 1977, pp. 87–93.

[61] F. P. Preparata and D. E. Muller, "Finding the Intersection of n Half-spaces in Time $O(n \log n)$," *Theoretical Computer Science*, Vol. 8, 1979, pp. 45–55.

[62] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, 1985.

[63] J. H. Reif, "An Optimal Parallel Algorithm for Integer Sorting," *Proc. 26th IEEE Symp. on Foundations of Comp. Sci.*, 1985, pp. 496–504.

[64] J. H. Reif and S. Sen, "Optimal Parallel Algorithms for Computational Geometry," *Proc. 1987 IEEE Int. Conf. on Parallel Processing*, pp. 270–277.

[65] J. H. Reif and S. Sen, "An Efficient Output-Sensitive Hidden-Surface Removal Algorithm and Its Parallelization," *Proc. 4th ACM Symp. on Computational Geometry*, 1988, pp. 193–200.

[66] J. H. Reif and S. Sen, "Polling: A New Randomized Sampling Technique for Computational Geometry," *Proc. 21st ACM Symp. on Theory of Computing*, 1989, pp. 394–404.

[67] J. H. Reif and S. Sen, "Randomized Algorithms for Binary Search and Load Balancing with Geometric Applications," *Proc. 2nd ACM Symp. on Parallel Algorithms and Architectures*, 1990, pp. 327–337.

[68] C. Rüb, *Parallele Algorithmen zum Berechnen der Schnittpunkte von Liniensegmenten*, Doctoral Dissertation, Universität des Saarlandes, 1990.

[69] S. Sen, *Random Sampling Techniques for Efficient Parallel Algorithms in Computational Geometry*, Ph.D. thesis, Computer Science Dept., Duke Univ., 1989.

[70] M. I. Shamos, "Geometric Complexity," *Proc. 7th ACM Symp. on Theory of Computing,* 1975, pp. 224–233.

[71] M. I. Shamos and D. Hoey, "Closest-Point Problems," *Proc. 15th IEEE Symp. on Foundations of Computer Science*, 1975, pp. 151–162.

[72] Y. Shiloach and U. Vishkin, "Finding the Maximum, Merging, and Sorting in a Parallel Computation Model," *J. Algorithms*, Vol. 2, 1981, pp. 88–102.

[73] Q. F. Stout, "Constant-time geometry on PRAMs", *Proc. 1988 Int'l. Conf. on Parallel Computing*, vol. III, IEEE, pp. 104-107.

[74] R. Tamassia and J. S. Vitter, "Optimal Cooperative Search in Fractional Cascaded Data Structures," *Proc. 2nd ACM Symp. on Parallel Algorithms and Architectures*, 1990, pp. 307–316.

[75] R. Tamassia and J. S. Vitter, "Optimal Parallel Algorithms for Transitive Closure and Point Location in Planar Structures," *Proc. 1989 ACM Symp. on Parallel Algorithms and Architectures*, pp. 399–408.

[76] R. E. Tarjan and C. J. Van Wyk, "An $O(n \log \log n)$-Time Algorithm for Triangulating a Simple Polygon," *SIAM J. Comput.*, Vol. 17, 1988, pp. 143–178.

[77] G. T. Toussaint, "Solving Geometric Problems with Rotating Calipers," *Proc. IEEE MELECON '83*, Athens, Greece, May 1983.

[78] P. M. Vaidya, "Reducing the Parallel Complexity of Certain Linear Programming Problems," *31st IEEE Symp. on Foundations of Computer Science*, 1990, pp. 583–589.

[79] L. Valiant, "Parallelism in Comparison Problems," *SIAM Journal on Computing*, Vol. 4, No. 3, 1975, pp. 348–355.

[80] H. Wagener, "Optimally Parallel Algorithms for Convex Hull Determination," manuscript, 1985.

[81] C. A. Wang and Y. H. Tsin, "An $O(\log n)$ Time Parallel Algorithm for Triangulating a Set of Points in the Plane," *Information Processing Letters*, Vol. 25, 1987, pp. 55–60.

[82] C. C. Yang and D. T. Lee, "A Note on the All Nearest-Neighbor Problem for Convex Polygons," *Info. Proc. Letters*, Vol. 8, No. 4, 1979, pp. 193–194.

[83] C. K. Yap, "Parallel Triangulation of a Polygon in Two Calls to the Trapezoidal Map," *Algorithmica*, Vol. 3, 1988, pp. 279–288.