Open access • Journal Article • DOI:10.1145/116825.116827

An optimal parallel algorithm for the visibility of a simple polygon from a point

Mikhail J. Atallah, Hubert Wagener, Danny Z. Chen

Institutions: Purdue University, Technical University of Berlin

Published on: 01 Jul 1991 - Journal of the ACM (ACM)

Topics: Polygonal chain, Visibility polygon, Star-shaped polygon, Simple polygon and Parallel algorithm

1988

An Optimal Parallel Algorithm for the Visibility of a Simple Polygon from a Point

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

Danny Z. Chen

Hubert Wagner

Report Number:
88-759

AN OPTIMAL PARALLEL ALGORITHM FOR THE VISIBILITY OF A SIMPLE POLYGON FROM A POINT

Mikhail J. Atallah
Danny Z. Chen
Hubert Wagener

CSD-TR-759
April 1988
(Revised March 1989)

An Optimal Parallel Algorithm for the Visibility of a Simple Polygon from a Point

Mikhail J. Atallah* Purdue University

Danny Z. Chen† Purdue University

Hubert Wagener‡ T.U. Berlin

Abstract

We present a parallel algorithm for computing the visible portion of a simple polygonal chain with n vertices from a point in the plane. The algorithm runs in $O(\log n)$ time using $O(n/\log n)$ processors in the CREW-PRAM computational model, and hence is asymptotically optimal.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems–*computations on discrete structures, geometrical problems and computations*; I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling–*geometrical algorithms*.

General Terms: Algorithms, Theory

Additional Key Words and Phrases: computational geometry, simple polygons, visible regions, intersections of polygonal chains, parallel computational complexity

*Dept. of Computer Science, Purdue University, West Lafayette, IN 47907. This author's research was supported by the Office of Naval Research under Grants N00014-84-K-0502 and N00014-86-K-0689, and the National Science Foundation under Grant DCR-8451393, with matching funds from AT&T.

†Dept. of Computer Science, Purdue University, West Lafayette, IN 47907.

‡Fachbereich Informatik, T.U. Berlin FR 6-2, Franklinstr. 28/29, 1000 Berlin 10, W-Germany.

1 Introduction

Visibility is one of the most fundamental topics in computational geometry. Visibility problems find applications in many areas, such as graphics and robotics. Also, visibility problems often appear as subproblems of many other problems in computational geometry (like shortest-path with obstacles). In this paper, we consider one important visibility problem: given a point q and a simple n-vertex polygonal chain P in the plane, find all the points of P that are visible from q if P is opaque. Our goal is to provide an efficient parallel algorithm for this problem in the CREW-PRAM computational model. Recall that CREW-PRAM is the synchronous shared-memory model where concurrent reads are allowed but no two processors can simultaneously attempt to write in the same memory location (even if they are trying to write the same thing).

One of the major tasks of parallel algorithm design for PRAM models is to come up with parallel algorithms that are *optimal*, i.e., that run as fast as theoretically possible for the problem they solve and simultaneously have a *time* × *processors* bound that is within a constant factor of the time complexity of the best sequential algorithm for the problem they solve. This goal has been elusive for many simple problems that are trivially in the class NC. Until recently, the convex hull problem was one of the few geometric problems for which an optimal algorithm was known. Recently, the "cascading divide-and-conquer" technique [C,ACG] has yielded a long list of optimal algorithms for geometric problems, in particular for the visibility problem when the opaque objects are nonintersecting line segments. The algorithm in [ACG] runs in $O(\log n)$ time using $O(n)$ processors, which is optimal for arbitrary non-intersecting line segments, but is suboptimal when the n line segments form the boundary of a simple (possibly closed) polygonal chain. No modification of [ACG] seems to yield an optimal algorithm for that problem. Indeed, in order to obtain an optimal algorithm for that problem, this paper follows a very different approach, and a CREW-PRAM algorithm that takes $O(\log n)$ time and uses $O(n/\log n)$ processors is provided. The contribution of this paper is actually twofold: first, it provides the first optimal parallel algorithm for the problem of visibility of a simple polygonal chain from a point, which also gives efficient parallel algorithms for other geometric problems on a simple polygonal chain (some of them are mentioned in the concluding section); second, it presents geometric insights that allow efficient detection of intersections between the visibility chains of different portions of the polygonal chain. These insights are likely to be useful in solving other problems about simple polygonal chains.

This algorithm is optimal to within a constant factor because (i) there is an obvious $\Omega(n)$ sequential lower bound for the problem, and (ii) an $\Omega(\log n)$ lower bound on its CREW-PRAM

time complexity is easily obtained by reducing to it the problem of computing the maximum of n entries. Several sequential algorithms [EA,L,DS] have solved the problem within a linear time bound.

In the next section, we give the notations and definitions used in this paper. An overview of the algorithm is sketched in Section 3. Section 4 presents the crucial geometric insights, and the algorithm based on them. In Section 5, we mention some applications of the algorithm.

Throughout, all logarithms are to the base 2 unless otherwise specified.

2 Terminology

The input consists of a point q and a simple polygonal chain $P = (v_1, v_2, \ldots, v_n)$ in the plane (possibly $v_1 = v_n$), where the given sequence of vertices is such that when we visit them in the order v_1, v_2, ..., v_n, we are traveling along chain P and encounter each point on P exactly once (except at the starting point v_1 if $v_1 = v_n$). Let s_i denote the segment joining v_i to v_{i+1}. The order in which a walk along P from v_1 to v_n encounters the v_i's is called the *chain order* and is denoted by $<_p$. We say v_i has *rank* i in the chain order, and denote it by $rank(v_i)$. For example, $v_3 <_p v_9$ since $rank(v_3) = 3 < 9 = rank(v_9)$. We extend the notion of rank to all points on P as follows: for a point p in the interior of a segment s_i, $rank(p) = rank(v_i)$.

If u and w are two points in the plane, then $\overline{uw}$ $(= \overline{wu})$ denotes the line segment joining them. We assume that every chain we consider in this paper is *simple*, that is, no two segments in it intersect each other (except possibly at their endpoints), and each segment has at most two common endpoints with the other segments in the chain. If, in P, $v_1 \neq v_n$, then P is *open*, otherwise it is *closed*. From now on, all chains are assumed to be open because the closed case is reduced to the open case by first "opening it" by removing a segment s from it (any s will do), then solving the visibility problem for $P - s$ using the algorithm for the open case, and finally including the effect of s in $O(\log n)$ additional time using the $n/\log n$ processors available. Each chain C has a length, denoted by $|C|$, which is the number of line segments in it. Given a chain C, let Q be the star-shaped polygon consisting of the portion of the plane visible from q when C is the only opaque object. Then $VIS(C)$, the *visibility chain* of C from q, equals the boundary of Q minus the (at most two) edges on the boundary of Q that are incident to the point at infinity. Once we have $VIS(C)$, it is trivial to extract from it the portion of C visible from q (i.e., $VIS(C) \cap C$). Hence, our goal from now on is to compute $VIS(P)$ for the input polygonal chain P.

A point p is given by its x-coordinate and y-coordinate, denoted by $x(p)$ and $y(p)$, respectively.

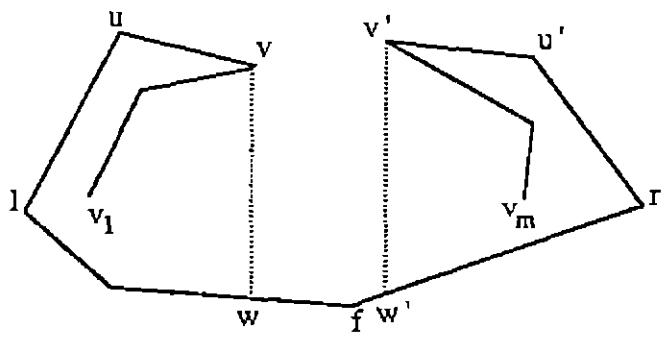

Figure 1. Illustrating the definitions.

Without loss of generality, we assume in the rest of this paper that the visibility is computed with respect to $q = (0, +\infty)$. Note that $VIS(C)$ is *monotone* with respect to the x-axis, in the sense that its intersection with any vertical line is either a single point of C, or a vertical line segment that connects two visible portions of C (Figure 1). Except for their endpoints, these vertical line segments of $VIS(C)$ do not belong to C, and we therefore call them the *extra vertical segments* of $VIS(C)$. In Figure 1, C is the chain from v_1 to v_m, and the segments of $VIS(C)$ are, from left to right, $\overline{lu}$, $\overline{uv}$, $\overline{vw}$, $\overline{wf}$, $\overline{fw'}$, $\overline{w'v'}$, $\overline{v'u'}$, $\overline{u'r}$ ($\overline{vw}$ and $\overline{w'v'}$ are the extra segments). Note that the two endpoints of $VIS(C)$ are the same as the leftmost and rightmost points of C, and need not coincide with the endpoints of C (in Figure 1, the endpoints of C are v_1 and v_m, while the endpoints of $VIS(C)$ are l and r).

The monotonicity of $VIS(C)$ enables us to store it in a binary search tree structure that allows one processor to search, in time proportional to its height, by x-coordinate or, alternatively, by leaf order (i.e., "find the k-th vertex of $VIS(C)$"). The tree structure also supports "split" operations in time proportional to its height (i.e., "remove from the tree all leaves whose x-coordinate is less than x_0"). Even if these splits are done very naively (i.e., if each of the two trees resulting from a split has the same height as the original one), we shall later show that the height of this binary search tree remains logarithmic in its number of leaves. To avoid introducing new terminology, we also use the same symbol (i.e., $VIS(C)$) to denote both the visibility chain of C and the balanced tree data structure describing it.

We say that point v is *below* point w if $y(v) < y(w)$ and $x(v) = x(w)$. We say v is *to the left* of w if $x(v) < x(w)$, in which case w is *to the right* of v. For two points v and w, $x(v) < x(w)$, we say that point u is *geometrically in between* v and w iff $x(v) \leq x(u) \leq x(w)$. If a point v of a chain C

is also on $VIS(C)$, then we say that v is an *open point* of C. We use $I(C)$ to denote the interval on the x-axis determined by the vertical projection of C on the x-axis. In Figure 1, $I(C)$ is $[x(l), x(r)]$. Suppose that chain C is partitioned into k subchains and the visibility chain of each subchain is available. Then when we talk about *combining* the k visibility chains, we mean computing $VIS(C)$ from these k visibility chains.

To simplify the exposition, we assume that no segment of P is vertical, and that no two consecutive segments of P are collinear, (the general case can be included in our solution without much difficulty).

3 An Overview of the Algorithm

We call VisChain the recursive procedure for computing the visibility chain of a simple polygonal chain. The procedure is outlined below. The initial call to the procedure is VisChain$(P, \log n)$, where P is a simple polygonal chain and $n = |P|$.

VisChain(C, d)

Input: A simple polygonal chain C of length m, and $\max(1, m/d)$ CREW-PRAM processors.

Output: The visibility chain $VIS(C)$ from the point $(0, \infty)$.

Step 1. If $m \leq d$, then compute $VIS(C)$ with one processor in $O(m)$ time, using any of the known sequential linear-time algorithms.

Step 2. If $d < m \leq d^2$, then divide C into two subchains C_1 and C_2 of equal length and recursively call VisChain(C_1, d) and VisChain(C_2, d) in parallel. Then compute $VIS(C)$ from $VIS(C_1)$ and $VIS(C_2)$, in $O((\log m)^2)$ time using one processor.

Step 3. If $m > d^2$, then partition C into $g = (m/d)^{1/4}$ subchains $C_1, C_2, \ldots, C_g$ of length $m^{3/4}d^{1/4}$ each. In parallel, call VisChain(C_1, d), VisChain(C_2, d), ..., VisChain(C_g, d). Then compute $VIS(C)$ from $VIS(C_1)$, $VIS(C_2)$, ..., $VIS(C_g)$, in $O(\log m)$ time and using the m/d processors available.

end.

The main difficulty lies in the "conquer" steps: two visibility chains $VIS(C_i)$ and $VIS(C_j)$ can have *two* intersections, and we have (cf. Step 3 above) only $g^2 = (m/d)^{1/2}$ processors to compute these two intersections between each pair $(VIS(C_i), VIS(C_j))$. Doing this in $O(\log m)$

time may appear impossible at first sight: the length of each of $VIS(C_i)$ and $VIS(C_j)$ can be $m^{3/4}d^{1/4}$, and there is a well-known *linear* lower bound [CD] on the work needed for computing the two intersections of two arbitrary polygonal chains that intersect twice (even if both chains are convex). This seems to imply that, since we have only $(m/d)^{1/2}$ processors assigned to the task, it will take $m^{3/4}d^{1/4}(m/d)^{-1/2} = m^{1/4}d^{3/4}$ time rather than the claimed $O(\log m)$. What saves us is the fact that C_i and C_j are subchains of a simple polygonal chain. How to exploit this fact is one of the main contributions of this paper.

Observe that, if we could perform the various steps of the above algorithm within the claimed bounds, then it would indeed run in $O(d + \log m)$ time with $O(m/d)$ processors since its time and processor complexities would satisfy the following recurrences:

$$t(m,d) = \begin{cases} c_1 d & \text{if } m \le d \\ t(m/2,d) + c_2(\log m)^2 & \text{if } d < m \le d^2 \\ t(m^{3/4}d^{1/4},d) + c_3 \log m & \text{if } d^2 < m \end{cases}$$

$$p(m,d) = \begin{cases} \max\{1,\ m/d\} & \text{if } m \le d \\ \max\{m/d,\ 2p(m/2,d)\} & \text{if } d < m \le d^2 \\ \max\{m/d,\ (m/d)^{1/4}p(m^{3/4}d^{1/4},d)\} & \text{if } d^2 < m \end{cases}$$

where c_1, c_2, c_3 are constants. From the above recurrences, the following bounds for $t(m,d)$ and $p(m,d)$ are easy to prove by induction:

$$t(m,d) \le \begin{cases} \alpha_1 d & \text{if } m \le d \\ \alpha_2 d + \beta_2(\log m)^2 \log d & \text{if } d < m \le d^2 \\ \alpha_3 d + \beta_3 \log m & \text{if } d^2 < m \end{cases}$$

$$p(m,d) = m/d$$

where α_1, α_2, β_2, α_3, β_3 are constants.

Choosing $d = \log m$, the above implies that $t(m, \log m) = O(\log m)$ and $p(m, \log m) = O(m/\log m)$. Hence the call $\mathbf{VisChain}(P, \log n)$ would compute $VIS(P)$ in $O(\log n)$ time using $O(n/\log n)$ processors.

Thus, in the rest of this paper, it suffices for us to show how, with m/d processors, to do the "combine" part of Step 2 in $O((\log m)^2)$ time, and more importantly, how to implement the "combine" part of Step 3 in $O(\log m)$ time.

We use the terminology of the above algorithm in the rest of this paper, so that a C_i is one of the subchains from the partition of C, and $VIS(C_i)$ is already available from the recursive call that computed it (i.e., we are focusing on the "combine" part). We define B_i to be the subchain of C

which is *before* C_i along the chain order, that is, B_i consists of the concatenation of $C_1, C_2, \ldots, C_{i-1}$. The subchain A_i of C which is *after* C_i along the chain order is defined similarly, that is, A_i consists of the concatenation of $C_{i+1}, C_{i+2}, \ldots, C_g$.

4 Visibility Chains and Their Intersections

This section presents the geometric insights together with their algorithmic implications. The most crucial ones are Lemma 4.5 and Lemma 4.6.

4.1 Simple Geometric Facts

Let $s = \overline{ab}$ be any straight line segment in $VIS(C) \cap C$, where a is encountered before b by a v_1-to-v_n walk along P. Then s is *leftward* if $x(b) < x(a)$, and is *rightward* if $x(a) < x(b)$. For example, in Figure 1, segments $\overline{uv}$ and $\overline{u'v'}$ are leftward, while segments $\overline{fw}$ and $\overline{fw'}$ are rightward. Let p be a vertex of $VIS(C)$, and let s and s' be the two segments of $VIS(C)$ having p as their common endpoint. Observe that, if none of s or s' is vertical, then either both of these segments are leftward, or both are rightward. If both are leftward (resp., rightward) then p is said to have a *left (resp., right) arrow tag*. If one of $\{s, s'\}$ is vertical or does not exist (in case p is an endpoint of $VIS(C)$), say it is s', then p has a left (resp., right) arrow tag iff s is leftward (resp., rightward). In Figure 1, the arrow tags of v, u, l, r, u', and v' are left, while those of w, f, and w' are right.

Lemma 4.1 *Let C be a simple chain, C' be a subchain of C. Then $VIS(C) \cap VIS(C')$ has at most three connected components (i.e., at most three separate portions of $VIS(C')$ appear in $VIS(C)$). If $C - C'$ has a single connected component (i.e., C' is at the beginning or the end of C), then $VIS(C) \cap VIS(C')$ has at most two connected components.*

Proof. We begin with the proof of the part when C' is at the beginning or the end of C. By contradiction, suppose that $VIS(C) \cap VIS(C')$ has three connected components. Let a, b, c be arbitrary points on each of these three components, respectively, with $x(a) < x(b) < x(c)$. Let u and w be points of $VIS(C) - VIS(C')$ such that $x(a) < x(u) < x(b) < x(w) < x(c)$. Now, the path Q in $C - C'$ joining u to w cannot pass above any of $\{a, b, c\}$, and therefore Q must pass below b in a way that isolates b, making it impossible for C' to go through b without crossing Q. This contradicts the fact that C is simple. We now prove the part when C' is neither at the beginning nor at the end of C. By contradiction, assume that $VIS(C) \cap VIS(C')$ has four connected components, and let a, b, c, e be points on each of those four components, respectively, with $x(a) < x(b) < x(c) < x(e)$. Let u, v, w be points of $VIS(C) - VIS(C')$ such that $x(a) < x(u) < x(b) < x(v) < x(c) < x(w) < x(e)$. Let A

and B be the two connected components of $C - C'$. By the pigeonhole principle, at least two points in the set $\{u, v, w\}$ are both in A or both in B (say, both in A). But then, $VIS(A \cup C') \cap VIS(C')$ has three connected components, contradicting the already proved part of the lemma. □

The above lemma implies that, in order to obtain from $VIS(C_i)$ the portions of $VIS(C_i)$ that are visible in $VIS(C)$, we need only perform a constant number of "split" operations on the tree representing $VIS(C_i)$. Over all such i, the total of $O(g)$ such splits results in $g' \leq 3g$ trees that are then used to create the tree $VIS(C)$ by simply building a complete binary tree "on top" of the roots of the g' trees, resulting in the height of $VIS(C)$ being higher by $\log(3g)$ $(= O(\log m))$ than the highest of the $VIS(C_i)$ trees. We shall explain later how to obtain the correct ordering of the g' trees used to build $VIS(C)$. For now, we simply observe that this method of building $VIS(C)$ from the $VIS(C_i)$'s results in the height of $VIS(C)$ being logarithmic in the number of its leaves (the proof of this is by an easy induction).

Let C' and C'' be two subchains of C such that C' and C'' are disjoint except possibly at a common endpoint. Since both $VIS(C')$ and $VIS(C'')$ can contain open points of C, and since they can intersect each other, we would like to compute exactly where the intersections occur in order to find which portions of $VIS(C')$ are hidden by $VIS(C'')$ (and vice versa). The next lemma ensures that the number of intersections between the two visibility chains $VIS(C')$ and $VIS(C'')$ is no more than *two*.

Lemma 4.2 *If C' and C'' are two subchains of C that are disjoint except that they may share one endpoint, then there are at most* two *intersections between $VIS(C')$ and $VIS(C'')$. If there are two intersections, then one of $I(C')$ or $I(C'')$ contains the other. If (say) it is $I(C')$ that contains $I(C'')$, then C' hides two disjoint portions of $VIS(C'')$ that are at the beginning and the end of $VIS(C'')$ (so that the portion of $VIS(C'')$ not hidden by C' is contiguous in $VIS(C'')$).*

Proof. The lemma would follow if we could show that there do not exist four points a, b, c, e such that $x(a) < x(b) < x(c) < x(e)$, a and c are in $VIS(C')$ and are not hidden by C'', while b and e are in $VIS(C'')$ and are not hidden by C'. Suppose to the contrary that four such points exist. The only way C' and C'' can link a to c and (respectively) b to e without hiding any of the four points $\{a, b, c, e\}$ would require an intersection between C' and C'', contradicting the fact that C is simple. □

Figure 2 gives an example in which $VIS(C')$ and $VIS(C'')$ have two intersections and $I(C')$ contains $I(C'')$.

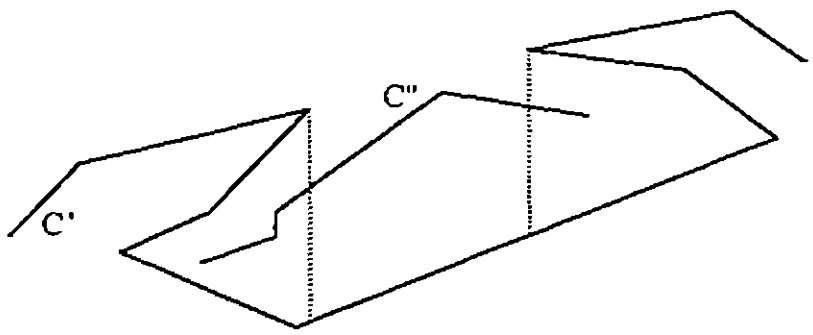

Figure 2. Illustrating the two-intersection case.

Although the above lemma limits to two the number of possible intersections between the two visibility chains of two subchains that are disjoint (except possibly at a common endpoint), the linear-work lower bound for detecting intersections between polygonal chains proved by Chazelle and Dobkin [CD] holds even for two chains that intersect each other no more than twice. We shall exploit the fact that the two chains are subchains of a simple chain in order to get around the lower bound. Specifically, the rest of the paper shows how to compute, for each C_i, the (by Lemma 4.1, at most two) portions of $VIS(C_i)$ that are hidden by A_i (computing the portions of $VIS(C_i)$ hidden by B_i is done in a symmetrical way and is therefore omitted). Note that there can be two intersections between $VIS(A_i)$ and $VIS(C_i)$, and we must compute these intersections in order to compute the (by Lemma 4.1, at most three) portions of $VIS(C_i)$ hidden by A_i and B_i. The computation of the portions of $VIS(C_i)$ hidden by A_i and B_i immediately gives us the (at most three) portions of $VIS(C_i)$ that belong to $VIS(C)$. Once we have done this (in parallel) for every $i \in \{1, \ldots, g\}$, it is easy to "stitch" the resulting $g' \leq 3g$ pieces of $VIS(C)$ and create $VIS(C)$: first split the trees representing $VIS(C_1)$, $VIS(C_2)$, ..., $VIS(C_g)$, in order to discard all portions of the $VIS(C_i)$'s that are invisible in $VIS(C)$; then the problem essentially becomes that of sorting the $O(g)$ endpoints of those portions of the $VIS(C_i)$'s that are visible in $VIS(C)$, which can be done in time $O(\log m)$ using g processors. We have g^4 processors available, more than enough to do this sorting. Thus we are justified in focusing, for the rest of the paper, on the problem of determining the portions of $VIS(C_i)$ that are hidden by A_i.

4.2 Simple Computational Observations

We next observe that, although $VIS(A_i)$ is not available after the recursive calls of Step 3 return the $VIS(C_j)$'s, we can still use the g^3 processors assigned to each C_i in order to answer meaningful queries about $VIS(A_i)$.

Lemma 4.3 *Let the $VIS(C_i)$'s be given. Let l be any vertical line, and let w be the highest intersection point between l and $VIS(A_i)$. Then g processors suffice for computing, in $O(\log m)$ time, the point w and the arrow tag of w on $VIS(A_i)$.*

Proof. Although we do not have $VIS(A_i)$ itself, we know that w is one of the $g-i$ points determined by the $g-i$ intersections of l with each of $VIS(C_{i+1})$, $VIS(C_{i+2})$, ..., $VIS(C_g)$, where $1 \leq i$. Thus w can be obtained in $O(\log m)$ time by (i) computing the intersection between l and each of $VIS(C_{i+1})$, $VIS(C_{i+2})$, ..., $VIS(C_g)$, then (ii) choosing the highest such intersection. As for the arrow tag computation, it too is done in $O(\log m)$ time by computing the immediate predecessor and successor of w on $VIS(A_i)$; these are easy to obtain, since they are determined by the set of $2(g-i)$ vertices that are adjacent to $x(w)$ on each of $VIS(C_{i+1})$, $VIS(C_{i+2})$, ..., $VIS(C_g)$. □

Corollary 4.4 *Let the $VIS(C_i)$'s be given. Given k vertical lines $\{l_1, \ldots, l_k\}$ in left-to-right order, let w_j, $1 \leq j \leq k$, be the highest intersection point between l_j and $VIS(A_i)$. With gk processors assigned to each C_i, each w_j and its arrow tag for $VIS(A_i)$ can be computed in $O(\log m)$ time.*

Proof. Assign g of the gk available processors to each vertical line, and use Lemma 4.3 for each such line. □

4.3 The Relative Positions of A_i and $VIS(C_i)$

This subsection gives a classification of the various possible relative positions of A_i and $VIS(C_i)$. We also point out how to identify each case. We do not yet compute the actual intersections of $VIS(A_i)$ and $VIS(C_i)$ (if any): this is postponed until the next subsection, when we will have developed more machinery for the computation of intersections (the most difficult cases will turn out to be Subcases 3.2 and 4.2 below, where two intersections might occur). Each of the subcases below can easily be seen to be identifiable in $O(\log m)$ time by using Lemma 4.3, where by "identifying a subcase", we mean just ascertaining that the subcase holds, not actually computing the portions of $VIS(C_i)$ hidden by A_i in that subcase.

Recall that we use $I(C)$ to denote the interval on the x-axis defined by the vertical projection of C on the x-axis. In the case analysis that follows, let $[x_l, x_r] = I(A_i) \cap I(C_i)$; a_{i1} and a_{i2} denote

the two highest points on $VIS(A_i)$ such that $x(a_{i1}) = x_l$ and $x(a_{i2}) = x_r$, while c_{i1} and c_{i2} denote the two highest points on $VIS(C_i)$ such that $x(c_{i1}) = x_l$ and $x(c_{i2}) = x_r$. $[x_l, x_r]$ is easily computed from $I(C_i), I(C_{i+1}), \ldots, I(C_g)$ in $O(\log m)$ time using $O(g)$ processors. The points a_{i1} and a_{i2} are obtained in $O(\log m)$ time using Lemma 4.3, while the points c_{i1} and c_{i2} are easy to obtained from $VIS(C_i)$ in $O(\log m)$ time by a simple one-processor search.

Case 1. The intervals $I(A_i)$ and $I(C_i)$ are disjoint except at a common endpoint (i.e., $x_l = x_r$). In this case, it is clear that no portion of $VIS(C_i)$ is hidden by A_i.

Case 2. The intervals $I(A_i)$ and $I(C_i)$ overlap without either of them containing the other. There are three subcases.

Subcase 2.1. Both c_{i1} and c_{i2} are above $VIS(A_i)$. Then there is no intersection between $VIS(A_i)$ and $VIS(C_i)$, and no portion of $VIS(C_i)$ is hidden by A_i.

Subcase 2.2. Both c_{i1} and c_{i2} are below $VIS(A_i)$. Then there is no intersection between $VIS(A_i)$ and $VIS(C_i)$, and the portion of $VIS(C_i)$ geometrically in between c_{i1} and c_{i2} is hidden by A_i (one of c_{i1} or c_{i2} is also hidden).

Subcase 2.3. One of c_{i1} or c_{i2} is above $VIS(A_i)$ and the other is below $VIS(A_i)$. Then $VIS(A_i)$ and $VIS(C_i)$ have exactly one intersection. To know which portion of $VIS(C_i)$ is hidden by A_i, we must later on compute the intersection between $VIS(A_i)$ and $VIS(C_i)$ (how to do this will be explained in the next subsection).

Case 3. $I(C_i)$ is contained in $I(A_i)$. There are three subcases.

Subcase 3.1. Both c_{i1} and c_{i2} are above $VIS(A_i)$. Then $VIS(A_i)$ and $VIS(C_i)$ do not intersect, and no portion of $VIS(C_i)$ is hidden by A_i.

Subcase 3.2. Both c_{i1} and c_{i2} are below $VIS(A_i)$. Then either $VIS(C_i)$ is completely hidden by A_i, or $VIS(A_i)$ and $VIS(C_i)$ have *two* intersections (cf. Lemma 4.2). Lemma 4.5 results in a method to distinguish these two situations, and to compute the portions of $VIS(C_i)$ that are hidden by A_i (the details are in the next subsection).

Subcase 3.3. One of c_{i1} or c_{i2} is above $VIS(A_i)$ and the other is below $VIS(A_i)$. Then there is exactly one intersection between $VIS(A_i)$ and $VIS(C_i)$. The portion of $VIS(C_i)$ hidden by A_i is not known until the intersection between $VIS(A_i)$ and $VIS(C_i)$ is found (how to find it will be explained in the next subsection).

Case 4. $I(A_i)$ is contained in $I(C_i)$. There are three subcases.

Subcase 4.1. Both a_{i1} and a_{i2} are above $VIS(C_i)$. Then the portion of $VIS(C_i)$ hidden by A_i is in geometrically between a_{i1} and a_{i2} (except both c_{i1} and c_{i2}), and $VIS(A_i)$ and $VIS(C_i)$ do not intersect.

Subcase 4.2. Both a_{i1} and a_{i2} are below $VIS(C_i)$. Then either A_i is completely hidden by $VIS(C_i)$, or $VIS(A_i)$ and $VIS(C_i)$ have *two* intersections. Lemma 4.6 results in a method to distinguish these two situations, and to compute, if there is one, the portion of $VIS(C_i)$ that is hidden by A_i (the details are in the next subsection).

Subcase 4.3. One of a_{i1} or a_{i2} is above $VIS(C_i)$ and the other is below $VIS(C_i)$. Then there is exactly one intersection between $VIS(A_i)$ and $VIS(C_i)$. The portion of $VIS(C_i)$ hidden by A_i is not known until the intersection between $VIS(A_i)$ and $VIS(C_i)$ is found (we will explain how to find it in the next subsection).

The above discussion considered the four possible cases and their subcases, and pointed out that each of them can easily be identified.

4.4 Computing the Portions of $VIS(C_i)$ Hidden by A_i

Now that we have identified which case and subcase holds for A_i and $VIS(C_i)$, we turn our attention to the problem of actually computing, for each subcase, the portions of $VIS(C_i)$ that are hidden by A_i. Note that, from Lemma 4.1, there are at most two such portions. Doing this for Case 1, Subcase 2.1, and Subcase 3.1 is trivial, since no portion of $VIS(C_i)$ is then hidden by A_i.

For Subcase 2.2 and Subcase 4.1, in which there is no intersection between $VIS(A_i)$ and $VIS(C_i)$, a simple one-processor binary search in $VIS(C_i)$ computes, in $O(\log m)$ time, the portion of $VIS(C_i)$ hidden by A_i.

For Subcases 2.3, 3.3, and 4.3, in which there is exactly one intersection between $VIS(A_i)$ and $VIS(C_i)$, we must locate that intersection in order to find the portion of $VIS(C_i)$ hidden by A_i. In the search for that intersection, the arrow tags are not needed (however, they will play a crucial role in the two-intersection cases discussed later).

In Step 2 of the algorithm, the (one) intersection is found by applying a one-processor binary search method, and results in the intersection being computed in $O((\log m)^2)$ time (because there are $O(\log m)$ queries in the binary search, and each such query requires $O(\log m)$ time). Such an $O((\log m)^2)$ time one-processor search is fine in Step 2, since our goal there is to perform the "combine" within this time bound anyway.

In Step 3, however, we need to find the intersection in $O(\log m)$ time, and thus we cannot afford to use the one-processor search. However, since $g^3 = (m/d)^{3/4}$ processors are available for each C_i, we can use Corollary 4.4 (with $k = (g+1)^2$) to perform a search for the intersection, as follows. A query of this search involves (i) finding $g+1$ vertical lines for each $VIS(C_a)$, $i \leq a \leq g$, that would partition $VIS(C_a)$ into g equal pieces (the leftmost and rightmost such lines for $VIS(C_a)$ go

through the endpoints of $VIS(C_a)$), (ii) sorting the g' ($= O(g^2)$) vertical lines of (i) in left to right order, (iii) checking whether the intersection point we seek is on one of those g' vertical lines (if so we stop, if not we proceed to (iv)), (iv) computing the (left to right) sequence of points w_1, w_2, $\ldots$, $w_{g'}$ where w_t is the highest intersection of the t-th vertical line with $VIS(C_i) \cup VIS(A_i)$ (and may thus come from either $VIS(C_i)$ or $VIS(A_i)$). The query either finds the intersection, or allows us to restrict the next stage of the search to the region $[x(w_j), x(w_{j+1})] \times [-\infty, +\infty]$ where one of $\{w_j, w_{j+1}\}$ is on $VIS(C_i)$ while the other is on $VIS(A_i)$. Such a query is done in $O(\log m)$ time using Corollary 4.4 (where $k = (g+1)^2$). Clearly, a query either finds the intersection, or restricts the next stage of the search for it to the portion of every $VIS(C_a)$ in $[x(w_j), x(w_{j+1})] \times [-\infty, +\infty]$. The search terminates within $O(\log_g m)$ ($= O(1)$) such queries, because for each $a \in \{i, \ldots, g\}$, the portion of $VIS(C_a)$ that lies in $[x(w_j), x(w_{j+1})] \times [-\infty, +\infty]$ has a size that is smaller by a factor of g than the size of $VIS(C_a)$. At the "bottom" of the recursion, either the intersection has been found, or each $VIS(C_a)$ has been reduced to a size of $O(1)$ and hence it is a trivial matter to find the intersection among the $O(g)$ surviving segments.

For Subcases 3.2 and 4.2, in which there are either zero or two intersections between $VIS(A_i)$ and $VIS(C_i)$, we can no longer directly apply the above one-intersection search, because the outcome of a query no longer enables us to constrict the search range. The rest of this subsection develops the machinery needed to tackle these two tricky subcases.

First observe that, in Subcase 3.2, if $VIS(C_i)$ is not completely hidden by A_i, then there are exactly two intersections between $VIS(A_i)$ and $VIS(C_i)$, and the portion of $VIS(C_i)$ not hidden by A_i is contiguous in $VIS(C_i)$ (cf. Lemma 4.2). Thus, any point p of $VIS(C_i)$ not hidden by A_i must lie geometrically in between the two intersections. If we could find such a point p, then the two intersections would be found by using, for each of them, the one-intersection search procedure (one search would operate to the left of p, the other to the right of p). This reduces the problem of tackling Subcase 3.2 to that of locating such a point p. Lemma 4.5 (to be given below) will help us compute such a point p.

Similarly, in Subcase 4.2, if $VIS(A_i)$ is not completely hidden by C_i, then there are exactly two intersections between $VIS(A_i)$ and $VIS(C_i)$, and the portion of $VIS(C_i)$ hidden by A_i is contiguous in $VIS(C_i)$ (cf. Lemma 4.2). Thus, any point p of $VIS(A_i)$ not hidden by C_i must lie geometrically in between the two intersections. If we could find such a point p, then the two intersections would be found by using, for each of them, the one-intersection search procedure. Lemma 4.6 (to be given below) will help us compute such a point p.

In the next two lemmas, the rank of a point is always with respect to its chain order in the

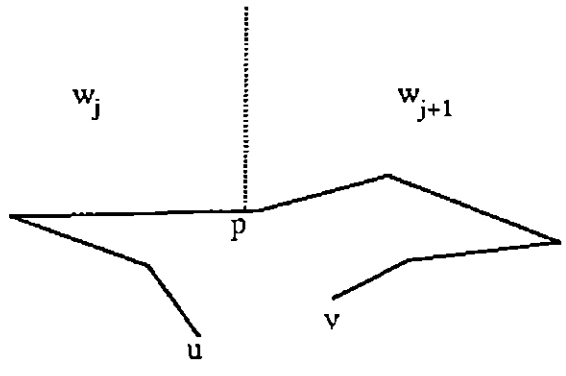

Figure 3. Illustrating the proofs of Lemmas 4.5 and 4.6.

original input chain P.

Lemma 4.5 *Suppose that $I(A_i)$ contains $I(C_i)$. Let the left and right endpoints of $VIS(C_i)$ be l and r, respectively. Let $W = (w_1, w_2, \ldots, w_k)$ be a sequence of points on $VIS(A_i)$ that are all above $VIS(C_i)$, and $x(l) = x(w_1) < x(w_2) < \cdots < x(w_k) = x(r)$. If the portion of $VIS(C_i)$ that is in $[x(w_j), x(w_{j+1})] \times [-\infty, +\infty]$ contains a point p that is not hidden by A_i (i.e., that is visible if C_i and A_i are the only opaque objects), then exactly one of w_j or w_{j+1} has lowest chain rank among all of w_1, w_2, ..., w_k. If it is w_j then its arrow tag is left, and if it is w_{j+1} then its arrow tag is right.*

Proof. Let u and v be the endpoints of C_i (Figure 3). WLOG, assume $u <_p v$ (i.e., $C_i \cap A_i = v$). First observe that, if w_j and w_{j+1} have same chain rank, then they are on the same segment of $VIS(A_i)$, and that segment would hide p (otherwise one of the points $\{w_j, w_{j+1}\}$ would be below $VIS(C_i)$). Therefore they have distinct chain ranks, say $w_j <_p w_{j+1}$ (the case $w_{j+1} <_p w_j$ is symmetrical, with the roles of "left" and "right" being interchanged). The v-to-w_{j+1} walk (call it Q) along A_i goes through w_j, and we now show that this implies that (i) the first point among $\{w_1, \ldots, w_k\}$ encountered by this walk Q is point w_j, and that (ii) the arrow tag of w_j is left. Suppose (i) is not true, i.e., that Q encounters some w_t before encountering w_j. Then the w_j-to-w_{j+1} portion of Q would hide w_t, a contradiction. Suppose (ii) is not true (i.e., the arrow tag of w_j is right). Then w_j is "isolated" from w_{j+1} in the sense that the w_j-to-w_{j+1} portion of Q would have to intersect C_i in order to reach w_{j+1}, a contradiction. □

We now discuss the algorithmic implication of the above lemma for handling Subcase 3.2. Assume that we are in Step 3. The above lemma implies that in Subcase 3.2, the point p we seek

(if it exists) lies geometrically in between the unique pair (w_j, w_{j+1}) such that exactly one of w_j or w_{j+1} (say, w_j) has the lowest chain rank among all of $w_1, w_2, \ldots, w_k$, and w_{j+1} is on the side of w_j which is opposite to the direction of the arrow tag of w_j. Therefore the lemma implies that the point p we seek (if it exists) has $x(p)$ in one interval $[x(w_j), x(w_{j+1})]$ that is easy to identify in $O(\log m)$ time so long as we have kg processors. This suggests using a search procedure in which a query involves (i) finding $g+1$ vertical lines for each $VIS(C_a)$, $i \leq a \leq g$, that would partition $VIS(C_a)$ into g equal pieces (the leftmost and rightmost such lines for $VIS(C_a)$ go through the endpoints of $VIS(C_a)$), (ii) sorting the g' $(= O(g^2))$ vertical lines of (i) in left to right order, (iii) computing the (left to right) sequence of points $u_1, u_2, \ldots, u_{g''}$ defined by the intersection of $VIS(C_i)$ with the g' vertical lines ($g'' \leq g'$ as some of these intersections do not exist. If an intersection is a vertical segment then its upper endpoint is chosen), (iv) computing the (left to right) sequence of points $w_1, w_2, \ldots, w_{g'}$ defined by the intersection of $VIS(A_i)$ with the g' vertical lines (again, if an intersection is a vertical segment then its upper endpoint is chosen), as well as their arrow tags for $VIS(A_i)$, and (v) checking whether any of $u_1, u_2, \ldots, u_{g''}$ is not hidden by A_i (if some of them are not hidden by A_i then take arbitrarily one of them to be the point p and stop). Such a query is done in $O(\log m)$ time using Corollary 4.4 (where $k = (g+1)^2$). Lemma 4.5 implies that we can use the outcome of this query to restrict the next stage of the search for p to the region $[x(w_j), x(w_{j+1})]$ $\times [-\infty, +\infty]$. The search terminates within $O(\log_g m)$ $(= O(1))$ such queries, because the portion of each $VIS(C_a)$ that lies in $[x(w_j), x(w_{j+1})] \times [-\infty, +\infty]$ has a size that is smaller by a factor of g than the size of $VIS(C_a)$. At the "bottom" of the recursion, either p has been found, or each $VIS(C_a)$ has been reduced to a size of $O(1)$ and hence it is a trivial matter to find p among the $O(g)$ surviving segments (actually in that second case we get much more than p: we get the (possibly empty) portion of $VIS(C_i)$ which is not hidden by A_i). If the search terminates without finding such a point p, then we know that no intersections exist and all of $VIS(C_i)$ is hidden by A_i. If such a point p is found then we have already explained how the problem reduces to the one-intersection case.

As far as Step 2 is concerned, the problem is much easier since in that case we know explicitly $VIS(A_i)$ (because $i \in \{1,2\}$ and $A_i = C_2$ if $i = 1$, and is empty if $i = 2$). We then handle Subcase 3.2 by using only one processor to compute the two intersections and the portions of $VIS(C_i)$ hidden by A_i: the processor performs a binary search for the desired point p and spends $O(\log m)$ time on each query of the search. Since there are then $O(\log m)$ such queries (not $O(1)$), this one-processor search for p takes $O((\log m)^2)$ time, just as required for Step 2.

Lemma 4.6 *Suppose that $I(C_i)$ contains $I(A_i)$. Let the left and right endpoints of $VIS(C_i)$ be l and r, respectively. Let $W = (w_1, w_2, \ldots, w_k)$ be a sequence of points on $VIS(C_i)$ such that no point in W is below $VIS(A_i)$, and $x(l) = x(w_1) < x(w_2) < \cdots < x(w_k) = x(r)$. If the portion of $VIS(A_i)$ that is in $[x(w_j), x(w_{j+1})] \times [-\infty, +\infty]$ contains a point p that is not hidden by C_i (i.e., p is visible if C_i and A_i are the only opaque objects), then exactly one of w_j or w_{j+1} has highest chain rank among all of $w_1, w_2, \ldots, w_k$. If it is w_j then its arrow tag is right, and if it is w_{j+1} then its arrow tag is left.*

Proof. Let u and v be the endpoints of A_i (Figure 3). WLOG, assume $v <_p u$ (i.e., $C_i \cap A_i = v$). First observe that, if w_j and w_{j+1} have same chain rank, then they are on the same segment of $VIS(C_i)$, and that segment would hide p (otherwise one of the points $\{w_j, w_{j+1}\}$ would be below $VIS(A_i)$). Therefore they have distinct chain ranks, say $w_{j+1} <_p w_j$ (the case $w_j <_p w_{j+1}$ is symmetrical, with the roles of "left" and "right" being interchanged). The v-to-w_{j+1} walk (call it Q) along C_i goes through w_j, and we now show that this implies that (i) the first point among $\{w_1, \ldots, w_k\}$ encountered by this walk Q is point w_j, and that (ii) the arrow tag of w_j is right. Suppose (i) is not true, i.e., that Q encounters some w_t before encountering w_j. Then the w_j-to-w_{j+1} portion of Q would hide w_t, a contradiction. Suppose (ii) is not true (i.e., the arrow tag of w_j is left). Then w_j is "isolated" from w_{j+1} in the sense that the w_j-to-w_{j+1} portion of Q would have to intersect C_i in order to reach w_{j+1}, a contradiction. □

We now discuss the algorithmic implication of the above lemma for handling Subcase 4.2. Assume that we are in Step 3. In Subcase 4.2, the point p we seek (if it exists) lies geometrically in between the unique pair (w_j, w_{j+1}) such that exactly one of w_j or w_{j+1} (say, w_j) has the highest chain rank among all of $\{w_1, w_2, \ldots, w_k\}$, and w_{j+1} is on the side of w_j that is the same as the direction of the arrow tag of w_j. Therefore the lemma implies that the point p we seek (if it exists) is such that $x(p)$ is in one interval $[x(w_j), x(w_{j+1})]$ that is easy to identify in $O(\log m)$ time. Thus a search procedure somewhat similar to the one we described above for Subcase 3.2 can be used, in which a query involves (i) partitioning $VIS(C_i)$ into g equal pieces using $g+1$ vertical lines, (ii) computing the $g+1$ highest points $w_1, \ldots, w_{g+1}$ that are the intersections of the $g+1$ vertical lines with $VIS(C_i)$ as well as their arrow tags for $VIS(C_i)$, (iii) computing the $g+1$ highest points $u_1, \ldots, u_{g+1}$ that are the intersections of the $g+1$ vertical lines with $VIS(A_i)$, and (iv) checking whether any of $u_1, u_2, \ldots, u_{g+1}$ is not hidden by C_i (if some of them are not hidden then take arbitrarily one of them to be the point p and stop). Such a query is done in $O(\log m)$ time using Corollary 4.4. Lemma 4.6 implies that we can use the outcome of this query to restrict the next

stage of the search for p to the region of $VIS(C_i)$ geometrically in between the pair (w_j, w_{j+1}). At the "bottom" of the recursion, when the portion of $VIS(C_i)$ being searched has shrunk to a single segment, either one of this segments' endpoints is p, or we can conclude that no p exists. If the search terminates without finding such a point p, then we know that all of A_i is hidden by $VIS(C_i)$, and hence that no portion of $VIS(C_i)$ is hidden by A_i. If such a point p is found then we have already explained how the problem reduces to the one-intersection case.

As far as Step 2 is concerned, we handle Subcase 4.2 just like we handled Subcase 3.2 (i.e., an $O((\log m)^2)$ time one-processor search).

5 Conclusion

We have presented a parallel algorithm for computing the visible portion of a simple n-vertex polygonal chain from a point in the plane. This algorithm works for any polygonal chain that does not self-intersect. The algorithm runs in $O(\log n)$ time using $O(n/\log n)$ processors in the CREW-PRAM computational model, and thus is optimal. The techniques used in the algorithm are a combination of fourth-root divide-and-conquer and two-way divide-and-conquer [AHU], and include a method for logarithmic time computation of intersections of special polygonal chains that intersect twice. Notice that the points on the visibility chain of P for $q = (0, \infty)$ are obtained sorted by their x-coordinates. Once the visibility chain of P for $q = (0, \infty)$ is available, many problems on simple polygons can be solved optimally in $O(\log n)$ time using $O(n/\log n)$ processors. For example, we can optimally compute the convex hull of P in the above time and processor complexities in CREW-PRAM by first using our visibility algorithm, and then using the algorithm in [G]. A direct method for optimally computing the convex hull of a simple polygon in parallel was given by Wagener [Wa]. Also, we can find all the maxima [PS] of the vertices of P by using parallel prefix after the portion of P visible from $(0, \infty)$ has been computed. Another immediate consequence of our algorithm is that we can compute the visibility graph [We] of P in $O(\log n)$ time using $O(n^2/\log n)$ CREW-PRAM processors, which is worst case optimal. The algorithm is likely to find applications in other geometric problems involving polygonal chains.

References

[ACG] Atallah, M., Cole, R., and Goodrich, M. "Cascading divide-and-conquer: a technique for designing parallel algorithm." *Proc. 28th Annual IEEE Symposium on Foundations of Computer Science*, 1987, pp. 151-160. (In print, *SIAM J. Comput.*)

[AHU] Aho, A. V., Hopcroft, J. E., and Ullman, J. D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.

[C] Cole, R. "Parallel merge sort." *SIAM J. Comput.* 17 (1988), 770-785.

[CD] Chazelle, B., and Dobkin, D. P. "Intersection of convex objects in two and three dimensions." *Journal of ACM 34,1*, (1987), 1-27.

[DS] Dean, J. A., and Sack, J. R. "Efficient hidden-line elimination by capturing winding information." *Proc. 23rd Annual Allerton Conference on Communication, Control, and Computing*, 1985, pp. 496-505.

[EA] El-Gindy, H., and Avis, D. "A linear algorithm for computing the visibility polygon from a point." *J. of Algorithms 2* (1981), 186-197.

[G] Goodrich, M. "Finding the convex hull of a sorted point set in parallel." *Inform. Process. Lett. 26* (1987/88), 173-179.

[HS] Hall, D. W., and Spencer, G. *Elementary Topology.* Wiley, New York, 1955.

[KRS] Kruskal, C. P., Rudolph, L., and Snir, M. "The power of parallel prefix." *IEEE Trans. on Computers* C-34 (1985), 965-968.

[L] Lee, D. T. "Visibility of a simple polygon." *Computer Vision, Graphics, and Image Processing 22* (1983), 207-221.

[LF] Ladner, R. E., and Fischer, M. J. "Parallel prefix computation." *J. of ACM 27,4* (1980), 831-838.

[PS] Preparata, F. P., and Shamos, M. I. *Computational Geometry: An Introduction.* Springer-Verlag, New York, 1985, pp. 151.

[Wa] Wagener, H. "Optimally parallel algorithms for convex hull determination." Unpublished manuscript, Sept. 1985.

[We] Welzl, E. "Constructing the visibility graph for n line segments in $O(n^2)$ time." *Inform. Proc. Lett.* 20 (1985), 167-171.