Algorithmica (2001) 31: 179–207
DOI: 10.1007/s00453-001-0042-6

Algorithmica
© 2001 Springer-Verlag New York Inc.

An Efficient Output-Size Sensitive Parallel Algorithm for Hidden-Surface Removal for Terrains

N. Gupta[1,2] and S. Sen[1]

Abstract. We describe an efficient parallel algorithm for hidden-surface removal for terrain maps. The algorithm runs in $O(\log^4 n)$ steps on the CREW PRAM model with a work bound of $O((n+k)\,\text{polylog}(n))$ where n and k are the input and output sizes, respectively. In order to achieve the work bound we use a number of techniques, among which our use of persistent data structures is somewhat novel in the context of parallel algorithms.

Key Words. Parallel algorithms, Hidden surface elimination, Output-sensitive, Data structure, Terrain.

1. Introduction

1.1. *The Problem.* The hidden-surface elimination problem (see [SSS] for the early history) has been a fundamental problem in computer graphics and can be stated as: given n polyhedral faces in $\mathbb{R}^3$ and a projection plane, we wish to determine which portions of the faces are visible when viewed in a given direction. We are interested in an object–space solution (independent of the display device) for this problem. That is, we are interested in a combinatorial description of the visible scene which can then be rendered on any display device. The image–space solutions compute the visibility information at every pixel which makes them device dependent. It has been shown that the worst-case output size for hidden-surface elimination can be $\Omega(n^2)$ for n segments, and, hence, the worst-case optimal algorithms for these problems will have a running time of $\Omega(n^2)$.

A slightly different version is the hidden-line elimination problem, where we are concerned only with the visibility of the edges (not regions). The algorithms for hidden-surface removal can be easily modified for the hidden-line elimination case. There are algorithms for hidden-line elimination in the literature whose running times are sensitive to the number of intersections, I, (of the projections of the segments) in the image plane. However, in practice, the size of the displayed image can be far less than the number of intersections in the image plane. By size, we mean the number of vertices and edges of the displayed image as a (planar) graph. This would happen when a large number of these intersections are occluded by the visible surfaces (see Figure 1). We study a special class of surfaces called polyhedral terrains which occur frequently in practice. A terrain is a three-dimensional polyhedral surface which can be represented as a function

[1] Department of Computer Science and Engineering, Indian Institute of Technology, New Delhi 110016, India. {neelima,ssen}@cse.iitd.ernet.in.
[2] Present address: Hansraj College, University of Delhi, Delhi 110007, India.

Received July 29, 1998; revised October 5, 1999. Communicated by F. P. Preparata.
Online publication July 19, 2001.

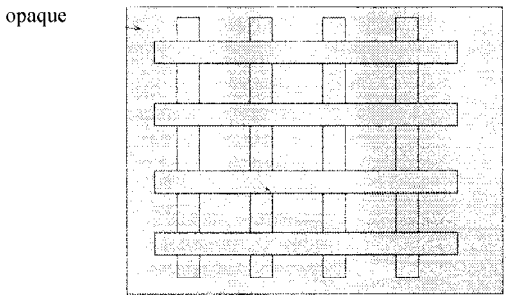

Fig. 1. The visible scene has only constant complexity whereas the number of intersections is $\Omega(n^2)$.

of two variables (see Figure 2). Most geographical features can be represented in this manner. A large number of scenes in graphics applications can be modelled efficiently and effectively by polyhedral terrains. The term (*upper*) *profile* refers to the piecewise linear function $Z(y)$, which is the pointwise maximum in the $+z$ direction of the projection of edges onto the z–y plane. Other commonly used terms for upper profile are *upper envelope* and *silhouette*. Therefore, a profile is a monotone polygon with respect to the y-axis. In fact, monotonicity turns out to be a very useful property in making the algorithm somewhat simpler than hidden-surface removal algorithm for general surfaces. However, even for terrains, it is known whether the maximum size of the visible image can be $\Omega(n^2)$. Our aim is to design a fast output-sensitive[3] parallel algorithm for terrains, which computes a description of the output in a device-independent manner.

1.2. *Sequential Algorithms*. McKenna [M] and Devai [D] proposed algorithms for the general problem that run in $O(n^2)$ time, and, hence, are worst-case optimal. There are algorithms for hidden-line elimination whose running times are sensitive to the number of intersections, I, (of the projections of the segments) in the projection plane, typically of the order of $O((n + I)\log n)$ (for example, see [N] and [S]). This was improved to $O(n \log n + I + t)$ by Goodrich [G] where t is the number of intersecting polygons in the image plane.

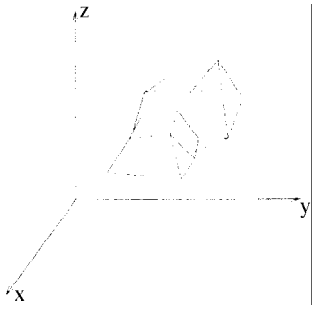

Fig. 2. A typical terrain map.

[3] We often use the shorter term output-sensitive instead of output-size sensitive.

The first known efficient output-sensitive algorithms were designed for the restricted input-class consisting of *iso-oriented* rectangles in $\mathbb{R}^3$ [GO], [Be], [AGO]. For the class of polyhedral terrains, Reif and Sen [RS1] designed the first efficient algorithm whose running time is $O((k+n)\log^2 n)$ where k is the output size. Preparata and Vitter [PV] presented an algorithm with the same running time and claimed that their algorithm was simpler. The algorithm in [OKS] improved the running time to $O((n\alpha(n)+k)\log n)$ where $\alpha(n)$ is the inverse Ackerman's function. For the case of nonintersecting objects there are algorithms which are somewhat output sensitive—for example, the algorithm of Overmars and Sharir [OS] takes about $O(n\sqrt{k}\log n)$ steps given an ordering of the objects. de Berg et al. [dBHO$^+$] and Agarwal and Matousek [AM] obtain improved bounds without an initial ordering for nonintersecting objects. However, these are still far away from the ideal bound of $((n+k)\,\text{polylog}(n))$.

1.3. *Parallel Algorithms.* The primary objective of designing parallel algorithms is to obtain very fast solutions to problems, keeping the total work (the processor–time product) close to the best sequential algorithms. For example, if $S(n)$ is the best known sequential time complexity for input size n, then we aim for a parallel algorithm with $P(n)$ processors and $T(n)$ running time to minimize $T(n)$ subject to keeping the product $P(n)\cdot T(n)$ close to $O(S(n))$. A parallel algorithm that actually does total work $O(S(n))$ is called a work optimal algorithm.

Relatively little work has been done in the context of parallel algorithms for hidden-surface removal. Reif and Sen [RS1] had proposed a parallelization of their algorithm with $O(\log^4 n)$ running time in a model that is stronger than PRAM. The more challenging theoretical goal was to keep the work bound close to the output-sensitive sequential algorithm. The resulting algorithm was quite complex and required parallel (dynamic) updates on a shared nested data structure that were not only hard to implement but also difficult to analyze. Here, we exploit some of their ideas but adopt a different strategy to build the parallel data structure. The resulting algorithm is relatively simpler and also easier to analyze. The main reason for this is that the underlying data structure is static although it has to be rebuilt a (small) number of times. Our bounds are also superior in the sense that we are able to match the bounds of [RS1] in a standard PRAM model (processor allocation was assumed to be free in the model used by [RS1]).

Goodrich et al. [GGB] presented parallel algorithms for hidden-surface elimination. For the general scenes, their algorithm computes all the I pairwise intersections on the projection plane. For the case of iso-oriented rectangles in $\mathbb{R}^3$, their algorithm is output-sensitive and runs in $O(\log^2 n)$ time using $O((n+k)\log n)$ total parallel operations. The crux of their method is a parallel data structure called *array-of-trees* introduced by Kosaraju et al. [KAG], that has some flavor of persistent data structures. In this paper we make more direct use of persistent data structures in our parallel algorithm.

We are not aware of any other published work in the context of provably efficient output-sensitive parallel algorithms for more general surfaces. The importance of output-size sensitivity for parallel algorithms cannot be overemphasized for the following simple reason. The advantage of using extra processors will be lost otherwise (for small output size) compared with an efficient output-sensitive sequential algorithm. The rest of the paper is organized as follows. In Section 2 we give a brief overview of our approach. In Section 3 we describe some of the basic parallel routines used frequently in the main

algorithm. Section 4 forms the crux of the paper. Since the algorithm is somewhat involved, we give a top-down description of the algorithm and the data structures accompanied by analysis.

2. An Overview of Our Algorithm. Recall from the Introduction that terrains in this paper refer to piecewise linear surfaces which meet a vertical line in exactly one point. Assume that the surface is a function $z = f(x, y)$, it is being viewed from $x = \infty$, and the viewing plane is the z–y plane. We are viewing the scene in a direction perpendicular to the projection plane, however, the algorithm works for the perspective projection as well. A characteristic of these surfaces is that the upper boundary of the projection of the line segments on the z–y plane is monotone with respect to the y-axis. We assume that the terrain is available as a graph G whose vertices are 3-tuples (x, y, z) of coordinates such that $z = f(x, y)$ and whose edges correspond to the segments of the polyhedral surface. The terms edges and segments have therefore been used interchangeably. We also assume that only the top part of the surface is visible, i.e., the faces closest to the observer rise from the ground level. A key property that allows one to solve the visibility problem efficiently is that the edges can be ordered from "front" to "back" using the following observation. Project G onto the x–y plane (call it G_{xy}) and now the ordering of the segments in the scene in the increasing distance from the viewer corresponds to the ordering of the edges of G_{xy} along x. That is, we can define a partial order on the edges as follows: edge $e_i \prec e_j$ if there is a ray in the viewing direction that intersects e_i before e_j. The projection of the edges on the x–y plane preserves this ordering.

2.1. *A Sequential Approach.* In the sequential algorithm the edges are ordered in the increasing distance from the viewer by decomposing G_{xy} into monotone chains of edges.

DEFINITION. A chain $C = (u_1, u_2, \ldots, u_p)$ is a planar straight line graph (PSLG) with vertex set $\{u_1, \ldots, u_p\}$ and edge set $\{(u_i, u_{i+1}), i = 1, \ldots, p - 1\}$. A chain is called monotone with respect to a straight line l if a line orthogonal to l intersects C in at most one point.

The edges are processed one by one sequentially in order. The algorithm maintains an upper profile of the edges processed so far and tests the visibility of the next edge being processed by determining its intersection with the current profile. Since the edges are processed in the order of increasing distance from the viewer, the profile lies in front of the next edge and therefore occludes the portion of the edge which lies behind it. Thus the portion of the projected edge lying below the profile (which is a simple monotone polygon) is not visible and hence is discarded (see Figure 3(a), the dotted portion of the edge is not visible). The upper profile is updated with the visible portions of the edge (see Figure 3(b)). Clearly, the portion of an edge declared visible is visible in the final image (i.e., it cannot be occluded by edges processed later). Some vertices and edges of the profile may be deleted at this point which only means that they are no longer part of the "upper boundary" of the final image but they are very much visible in the final image and therefore are remembered. Finally, we have all the vertices and edges of the final image which can be used by the rendering procedure to draw it on the screen.

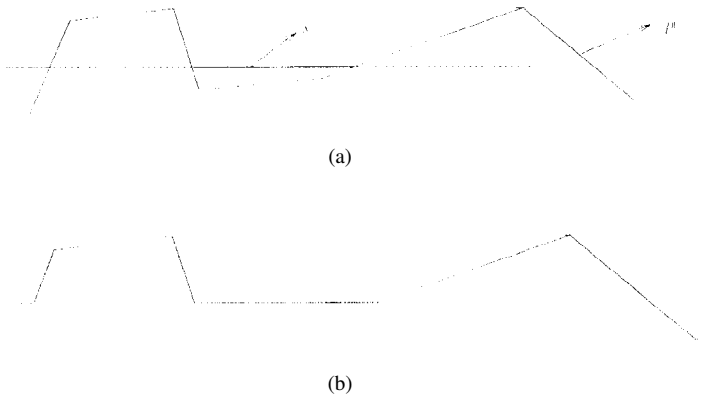

Fig. 3. The dotted part of the segment s lies below the profile p and hence is not visible, therefore it is discarded.

2.2. *An Overview of the Parallel Algorithm.* In the parallel scenario the above sequential algorithm has two major stumbling blocks. First, the edges are processed sequentially and the upper profiles are computed incrementally. We overcome this problem with the help of a separator tree and computing profiles using an approach similar to systolic implementation of parallel prefix computation. A separator tree provides a way to order the edges in the increasing distance from the viewer in parallel and also allows one to process them concurrently. Second, the intersections of an edge with a profile are computed sequentially. We use the divide-and-conquer approach to detect the intersections efficiently in parallel. We order the edges using a separator tree (described later). Let $e_1, e_2, \ldots, e_n$ be the ordered set of input edges. Let P_i denote the ith profile, i.e., the upper profile of the edges $e_1, e_2, \ldots, e_i$. Our aim is to compute P_i, $\forall i = 1, \ldots, n$. We call them *actual* profiles (however we omit "actual" most of the time and mention it only if it is not clear from the context).

We compute these profiles in two phases. In phase 1 we compute in parallel, for all the nodes of the separator tree, the upper profile of the edges in the leaves of the subtree rooted at the node (the edges in the leaves of the separator tree are sorted in the increasing distance from the viewer). Call the resulting tree the *profile computation tree* (henceforth referred to as the PCT). Notice that these profiles are not the actual profiles we are looking for. These are only intermediate profiles which are used to compute the actual profiles (think of the internal nodes in the prefix computation tree) in phase 2. In phase 2 we compute the actual profiles using an approach similar to the systolic implementation of parallel prefix computation [LF] (see Figure 4). Starting from the root of the profile computation tree the computation proceeds toward the leaves level by level. In this phase, at any node, the computation involves "merging" two profiles—an (actual) profile inherited from its parent and an (intermediate) profile computed in the previous phase by one of its children.

Merging is done by finding the intersections of the segments of the intermediate profile with the other profile and updating the other profile. However, as we will see later, our visibility structure (i.e., the vertices of the profiles) may be shared among different nodes at the same level of the PCT. We cannot afford to keep these profiles totally independent

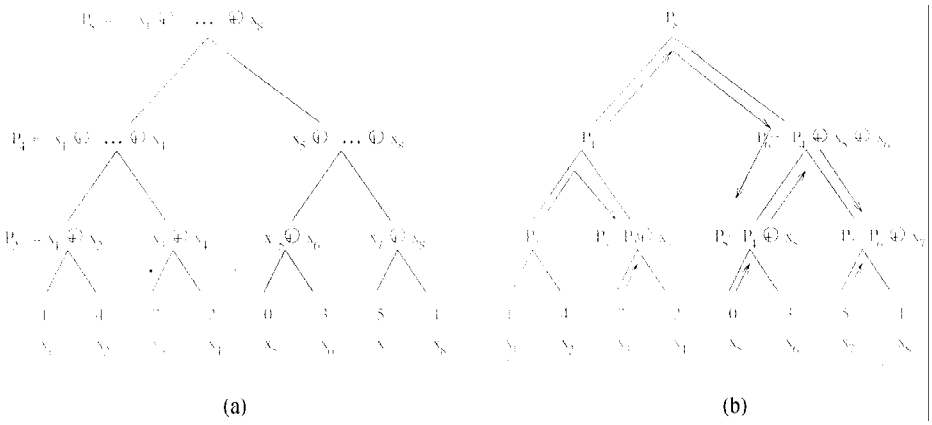

Fig. 4. An illustration of systolic implementation of parallel prefix sum computation. (a) Phase 1: computation proceeds bottom-up. (b) Phase 2: computation proceeds top-down.

of each other because that will jeopardize our main objective of designing an output-sensitive algorithm. Instead of keeping a visibility structure for each profile at a fixed level of the PCT we keep just one structure maintaining information about all the intersections computed so far and also provide a search structure to detect the intersections at the next level of the PCT.

REMARK. Our algorithm follows the basic approach of Reif and Sen [RS1]; however, our implementation of the merging phase is considerably simpler. One of the (sequential) algorithms of Overmars et al. [OKS] also follows a very similar approach. However, they avoid dynamic ray-shooting and implement merging in terms of various set operations (like intersection and differences) that leads to savings of a logarithmic factor in running time.

3. Some Basic Parallel Routines.

Before we describe the parallel algorithm in details, we briefly review some of the basic parallel routines that are used frequently in our algorithm.

LEMMA 3.1 [SV]. *Two sorted lists of sizes M and N can be sorted in $O(\log(M+N))$ time using $(M+N)/\log(M+N)$ processors.*

DEFINITION. Suppose there are n tasks and the ith task requests x_i processors such that $\sum_{i=1}^{n} x_i = O(n)$. Then the **processor allocation problem** of size n is to allocate a given set of processors among these tasks so that the ith task is alloted x_i processors (for all i).

LEMMA 3.2 [LF], [R]. *The processor allocation problem of size n can be solved in $O(\max\{n/p, \log n\})$ time with p processors on an EREW PRAM.*

DEFINITION. Given a bit vector of size n, the problem of **compaction** involves deleting the 0's and putting the 1's in contiguous locations starting from the first location.

LEMMA 3.3 [LF], [R]. *The compaction problem of size n can be solved in* $O(\max\{n/p, \log n\})$ *time with p processors on the EREW PRAM.*

LEMMA 3.4. *Given a convex polygon P of size m and a ray r, one can detect the intersection of r and P (or report that there are none) in* $O(\log m)$ *time sequentially.*

We shall also need frequent applications of the slow-down lemma which (in our context) can be formally stated as follows. Let $t_{p,r}$ denote the time to allocate p processors to a number of tasks whose total processor requirement is $O(r)$. That is $t_{p,r}$ is the time to solve the problem of processor allocation of size r with p processors.

LEMMA 3.5. *Let A be a parallel algorithm that executes in* Π *phases and performs a total number of N tasks (each task is not necessarily unit time but is performed by a single processor). Then the algorithm can be executed in time* $O(\Pi(t_{p,N} + t) + Nt/p)$ *using p processors in a PRAM where t is the time taken for each task.*

PROOF. This is a straightforward generalization of Brent's slow-down lemma [Br] in the context of PRAM algorithms. Let N_i be the number of tasks in phase i. Then the tasks can be distributed equally among p processors so that no processor gets more than $\lceil N_i/p \rceil$ tasks. Thus the total time is $\sum_i^{\Pi} O(N_i t/p + t_{p,N_i} + t)$, giving us the required result. □

LEMMA 3.6. *Let A be a parallel algorithm that executes in* Π *phases. Let* N_i *be the number of tasks in phase i, each executes in* $O(t_i)$ *time with* p_i *processors. Let* $t = \sum_{i=1}^{\Pi} t_i$ *and* $N = \max_i\{N_i p_i\}$. *Then the algorithm can be executed in* $O(\Pi t_{p,N} + t + Nt/p)$ *time with p processors.*

PROOF. This is a further generalization of the above lemma. Consider phase i of the algorithm A. Let $p < N_i p_i$. Let $p_i' = p/p_i$. Divide the N_i tasks in p_i' groups each of size $O(N_i/p_i')$. Distribute p processors equally among these groups so that each group gets p_i processors. Execute the tasks in each group one by one. The time in phase i is thus $O(N_i t_i/p_i' + t_{p,N_i p_i}) = O(N_i p_i t_i/p + t_{p,N}) = O(Nt_i/p + t_{p,N})$. However, if $p \geq N_i p_i$, then the time in phase i is $O(t_i)$. Thus the total time in phase i is $O(\max\{Nt_i/p + t_{p,N}, t_i\})$. The total time over all the phases is $O(\max\{N\sum_{i=1}^{\Pi} t_i/p + \Pi t_{p,N}, \sum_{i=1}^{\Pi} t_i\})$ giving us the required result. □

4. The Parallel Algorithm. We describe the algorithm in a top-down manner, treating the important steps in individual subsections accompanied by detailed analysis. Given a two-dimensional surface as a straight line graph in three dimensions, we project the line segments on the x–y plane. By the property of terrain maps, no two projected segments will intersect. If the graph is not triangulated, we triangulate the graph using the algorithm of Atallah et al. [ACG] for parallel triangulation. Since it is a planar graph, the number

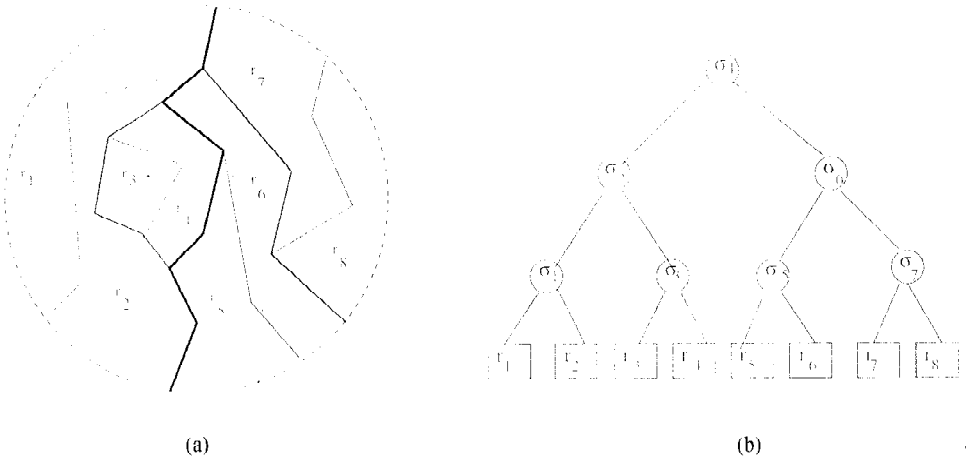

Fig. 5. Construction of a separator tree for a monotone subdivision: (a) Monotone subdivision S with the separator chains visualized. (b) Separator tree for S.

of edges and faces is still $O(n)$. Henceforth our discussions will be with respect to the triangulated graph. The main steps of the algorithm are:

1. **Order the edges** of the triangulated graph as follows: The triangulated graph is partitioned into roughly equal parts by a separator—a chain of edges monotone with respect to the y-axis. This is repeated recursively on each part until each part has a constant number of edges. This is done with the help of a separator tree (see Figure 5). Separator chains are ordered from front to back. Each edge e belongs to a range of separators $\{\sigma_j\colon\ i \leq j \leq k-1\}$. Order the edges based on the smaller index of the interval of chains it belongs to. That is, if edges e_1 and e_2 belong to $\sigma_i, \sigma_{i+1}, \ldots, \sigma_s$ and $\sigma_j, \sigma_{j+1}, \ldots, \sigma_t$, respectively, then $e_1 \prec e_2$ if $i \leq j$.
2. **Profile computation**
   (a) **Phase 1: Compute the intermediate profiles**—to compute the profiles we take the projection of the line segments on the z–y plane. For each node v in the separator tree do in parallel: compute the profile of the edges in the leaves of the subtree rooted at v. We call these profiles *intermediate* profiles. Note that these are not necessarily part of the final visible scene. As mentioned earlier, we call the resulting tree the profile computation tree (PCT). We use the term *layer* to imply a level of the PCT. Observe that the segments of the profiles may be shared among the layers of the PCT. For example, a profile at node v of the separator tree may share segments with a profile at the left (or the right) child l of v since the profile at l is the profile of the subset of edges of the set whose profile is computed at v (see Figure 6).
   (b) **Phase 2: Compute the actual profiles**—compute the actual (visible) profiles starting from the root of the PCT, proceeding layer by layer toward the leaves using an approach similar to the systolic implementation of parallel prefix computation. This step constitutes the crux of our algorithm.

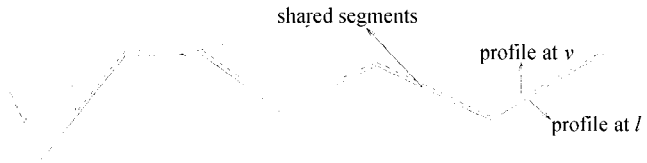

Fig. 6. l is the left child of the node v in the PCT; the profile at l is the profile of the subset of edges of the set whose profile is computed at v.

4.1. *Constructing the Separator Tree.* Let S be a planar subdivision whose regions are monotone polygons. A separator tree is a balanced binary tree T on S as described below. A separator σ of S is a monotone chain from $-\infty$ to $+\infty$. Let $r_1, r_2, \ldots, r_m$ be the regions of S, numbered such that $i < j$ whenever region r_i shares an edge with r_j and is to the left of r_j. The common boundary of the regions with index $\leq i$ and of the regions with index $> i$ is a separator of S, which we denote by σ_i. Each leaf of T represents a region of S, and each internal node represents a separator, such that the inorder sequence of the nodes of T is given by $r_1, \sigma_1, r_2, \sigma_2, \ldots, \sigma_{m-1}, r_m$. Each edge e of S belongs to a range of separators $\{\sigma_j\colon\ i \leq j \leq k-1\}$, where r_i and r_k are the regions to the left and right of e, respectively. For space efficiency, e is stored only once at the least common ancestor σ_l of r_i and r_k in T, and is called a proper edge of σ_l. See Figure 5. Step 1 of the main algorithm can be implemented in $O(\log n)$ time using a linear number of processors in an EREW PRAM using a procedure due to Tamassia and Vitter [TV]. Their result can be summarized as follows:

FACT 1. *Let S be a planar triangulated subdivision with n vertices. Then the separator tree consisting of monotone chains that decompose S can be constructed by an EREW PRAM in $O(\log n)$ time using n processors.*

4.2. *Computing the Intermediate Profiles.*

LEMMA 4.1. *The profile of a set of m segments can be constructed in $O(\log^2 m)$ time using $O(m\alpha(m)/\log m)$ processors in a CREW PRAM.*

PROOF. This is done by dividing the set of segments in halves, computing the profiles recursively for each half and merging the profiles as follows. Since the profile of $m/2$ segments can have size at most $O(m\alpha(m))$ [CS] we merge the vertices of the profiles in $O(\log m)$ time using $O(m\alpha(m)/\log m)$ processors (Lemma 3.1). We label (for this proof) a vertex "visible" if it is a part of the resultant profile and "invisible" otherwise. Find the predecessor of a vertex of one profile in the other profile. This can be computed while merging. From this we determine if the vertex is "visible" by checking if it lies above the segment (in the other profile) whose left endpoint is its predecessor (see Figure 7). The intersections can then be easily determined from the predecessor and the visibility information within the required bounds (see Appendix A for details). The points of intersections can be merged with the already merged set of vertices and the vertices of the new profile can be compacted by discarding the "invisible" vertices. The

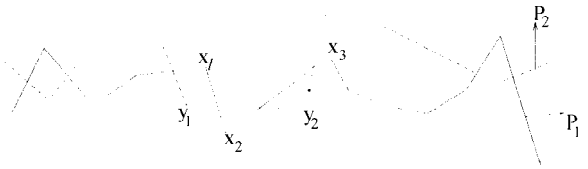

Fig. 7. The predecessor of x_1 (in p_2) is y_1 and it is visible because it lies above y_1y_2. The predecessor of y_2 (in p_1) is x_2 and it is invisible because it lies below x_2x_3.

required bounds follow from Lemmas 3.1 and 3.3. The total time bound follows from the recursive application of the procedure. □

Thus the profiles at all the nodes of the separator tree at a fixed layer can be computed in $O(\log^2 n)$ time using $O(n\alpha(n)/\log n)$ processors in a CREW PRAM or step 2(a) can be done in $O(\log^2 n)$ time using $O(n\alpha(n))$ processors in a CREW PRAM.

4.3. *Computing the Actual Profiles.* This step constitutes the crux of our algorithm. The actual profiles are computed layer by layer of the PCT starting from the root down to the leaves. Given the profiles and the data structure for intersection detection at a given layer, say L, of the PCT, the profiles at the next layer are computed by computing the intersections of the segments of some intermediate profiles with the actual profiles at the layer L. Since this is a very long section, we have organized it as follows. In Section 4.3.1 we explain how to compute the first intersection of a segment (ray) with a profile. We develop shared data structures based on the basic data structure of Chazelle and Guibas. In Section 4.3.2 we explain how to compute all the intersections at the next layer of the PCT by applying the following lemma.

LEMMA 4.2. *Given the data structure for intersection detection of a profile P of size m and a line segment s, we can find all the k_s intersections of s with P in time $O(\max\{(T_I + t_{p,k_s}) \log m, k_s T_I/p\})$ with p processors on a CREW PRAM, where T_I is the sequential time to detect the first intersection of a segment with P and t_{p,k_s} is the time for processor allocation of k_s tasks using p processors.*

PROOF. See Figure 8. Find the diagonal d of the profile such that the segment spans roughly an equal number of diagonals on either side. This can be done by a simple binary search for the endpoints of the segment and taking the middle diagonal of all the diagonals spanned by the segment. We then divide the line segment into two subsegments—one bounded by the left endpoint, say l, of s and the point of intersection, say t, of d and s and the other bounded by the right endpoint, say r, of s and t, and apply the sequential search (for an intersection) algorithm on both simultaneously (it is as if we have divided the line in two rays in opposite directions). If an intersection is detected, say q, in the right subsegment tr, we allocate an extra processor to the part of it between the intersection point q and the original endpoint r of s and repeat the procedure recursively on qr. Clearly, all the intersections will be detected in $O(\log m)$ stages. We call a subsegment "alive" if an intersection is detected in it, else call it "dead." An application

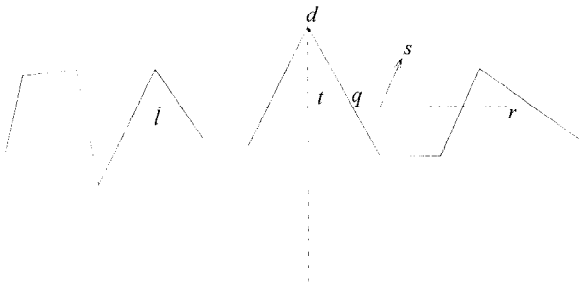

Fig. 8

of compaction can be used to delete the dead subsegments at each stage. Let the number of alive subsegments at the ith stage be s_i. Then, using s_i processors at the next stage, the first intersection of these subsegments can be computed in $O(T_I)$ time and with $p' < s_i$ processors they can be computed in $O(s_i T_I/p')$ time by Brent's slow-down lemma. The time at stage i is therefore $O(\max\{s_i T_I/p, T_I\})$ plus the time for processor allocation and compaction with p processors. The number of alive subsegments at stage i is at most twice the number of intersections detected at stage $i-1$, i.e., $s_i \leq 2k_{i-1}$ for $i \geq 2$, where k_i is the number of intersections detected at the ith stage, $s_1 = 2$. The time taken at stage i is therefore $O(\max\{T_I, t_{p,k_s}, k_{i-1}T_I/p\})$ or the total time over all the stages is $O(\max\{(T_I + t_{p,k_s}) \log m, (T_I/p) \sum_i k_i\}) = O(\max\{(T_I + t_{p,k_s}) \log m, k_s T_I/p\})$. □

REMARK. Since the total number of tasks is $O(k_s)$ the above result also follows by a direct application of Lemma 3.5.

4.3.1. *Computing the first intersection of a segment with a profile.* We organize this section as follows. In Section 4.3.1.1, we review the data structure due to Chazelle and Guibas (CG) which is the basic data structure used in our algorithm and also explain how to compute the first intersection of a segment with a profile given appropriate data structures with the CG data structure. In Section 4.3.1.2 we explain how to augment the above data structure with some shared data structures to make it complete for intersection detection. There we explain how to handle the shared visible portions of the profiles. In Section 4.3.1.3 we combine the results of the above two subsections and complete the construction of the data structure. This data structure stores the output vertices and also provides a way to search the intersections at the next layer of the PCT.

4.3.1.1. *Review of the data structure of Chazelle and Guibas.* To detect the first intersection between a segment and a profile (if an intersection exists), we use the data structure of Chazelle and Guibas [CG] to represent the profile. We review it briefly as required for our purpose. The sequential algorithm of Reif and Sen revolved around making this data structure dynamic. Given a simple polygon P (which is monotone in our case), we construct a binary tree where each node represents a portion of the polygon. The size (number of vertices) of the polygons associated with each node decreases geometrically with depth so that the tree has a logarithmic depth. The polygon is partitioned into roughly equal-sized polygons (in our case) by dropping the vertical attachments

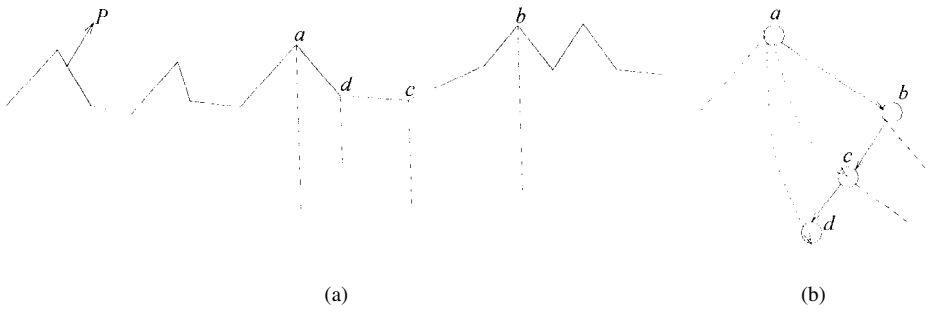

Fig. 9. (a) Profile P. (b) CG data structure for P.

called *diagonals* in the negative z-direction from all the vertices and choosing the middle one as a separator. Each node of the tree corresponds to a diagonal and the polygon it partitions. This is repeated until constant-sized polygons are obtained. See Figure 9. To detect an intersection between a line segment s and a polygon (upper profile) P the following procedure is used: Suppose that we know a node v such that s intersects the diagonal v (we use the same name for a node and the associated diagonal without any ambiguity). From the point of the intersection of s with the diagonal v we move along s in one direction, say toward the right. We would like to know the furthest diagonal d which s can cross without intersecting (any edge of) P in between. Clearly, such a diagonal lies in the right subtree of v. So we search for it in the right subtree of v going down the subtree level by level. The search procedure is recursive and can be outlined as follows:

Search_d (v, right child r of v) if s does not intersect P between v and r, then (move toward the right of r) Search_d (r, right child of r) else (d must be in between v and r) Search_d (v, left child of r).

In the end either we conclude that there is no intersection (of s with P on the right side of v) or a constant-sized polygon is obtained in which s intersects P. The intersection can then be obtained in constant time. The intersections toward the left of v can be obtained similarly.

The above procedure requires $O(\log|P|)$ phases, where $|P|$ is the number of vertices in P and in each phase it involves checking whether a ray intersects P between two diagonals. For this we compute the lower convex chain of all the vertices of the polygon between the two diagonals. Clearly, a ray intersects the polygon between the two diagonals if and only if it intersects this convex chain. Whether a ray intersects a convex chain can be determined by a simple binary search and hence it takes $O(\log|P|)$ time (Lemma 3.4). The original data structure of CG was somewhat more complex based on dual transforms. The above procedure is along the lines of Preparata and Vitter [PV] and takes $O(\log^2|P|)$ time. To facilitate the above search procedure Chazelle and Guibas augment the above data structure with additional pointers that Reif and Sen referred to as *shooting pointers*. A shooting pointer is added between a node v and its descendant w if the diagonal v is in the boundary of the polygon associated with w, see Figure 9(b)

where the shooting pointers are shown as dotted arcs. The following properties can be easily verified for a monotone polygon:

PROPERTY 4.1. There can be at most two shooting pointers between a node v and its descendants at a fixed level.

PROPERTY 4.2. There can be at most one shooting pointer between a node w and its ancestors.

We refer to this structure of Chazelle and Guibas as the CG data structure in future. By an *edge* of CG we imply either a tree edge or a shooting pointer unless explicitly stated otherwise. We augment each edge ab of the CG data structure with the lower convex chain of the vertices of the profile between a and b. We call the resulting structure an *augmented* CG and refer to it as ACG hereafter.

LEMMA 4.3. *For a profile with m vertices, we can construct the CG data structure in $O(\max\{\log m, t_{p,m} + m \log m/p\})$ time using p processors on the CREW PRAM.*

PROOF. A profile (which is monotonic) can be divided recursively into halves, quarters, etc. Hence constructing the underlying CG tree is easy. Shooting pointers can be determined as follows: a node v has a shooting pointer to every node in the rightmost (respectively leftmost) branch of its left (respectively right) subtree. Hence at a fixed level there are at most two nodes to which v has a shooting pointer. Moreover, any node has at most one shooting pointer to its ancestors (mentioned above). Thus the CG data structure can be constructed in $O(\log^2 m)$ time using $O(m/\log m)$ processors or in the required time using p processors from Lemma 3.5. □

We defer the discussions on computing the convex chains for a while.

4.3.1.2. *Representing shared visible portions.* As mentioned earlier, we compute the actual profiles using an approach similar to the systolic implementation of the parallel prefix computation. The main operation at every node being the "merging" of two profiles—one actual profile inherited from its parent and the other an intermediate profile precomputed in phase 1 of profile computation by one of its children. A crucial factor here is the sharing of common visible segments between the profiles being computed at different nodes of the same layer of the PCT. If we keep one ACG structure for each profile this redundancy may multiply, leading to a very inefficient algorithm since we have to build the data structure repeatedly on the same parts of the profile again and again. The total number of computations during the course of the algorithm may turn out to be several times larger than the output size, thus jeopardizing our initial objective of designing an output sensitive algorithm. We tackle this problem as follows.

In phase 2 of the profile computation, at a fixed layer of the PCT, instead of keeping an ACG structure for every profile, we keep a single ACG structure for all the profiles. In other words, we keep all the intersections found up to a certain layer of the PCT in one ACG, which provides a search structure to detect the intersections at the next layer. That is, we construct CG on all the intersections found up to a certain layer, say L, of the

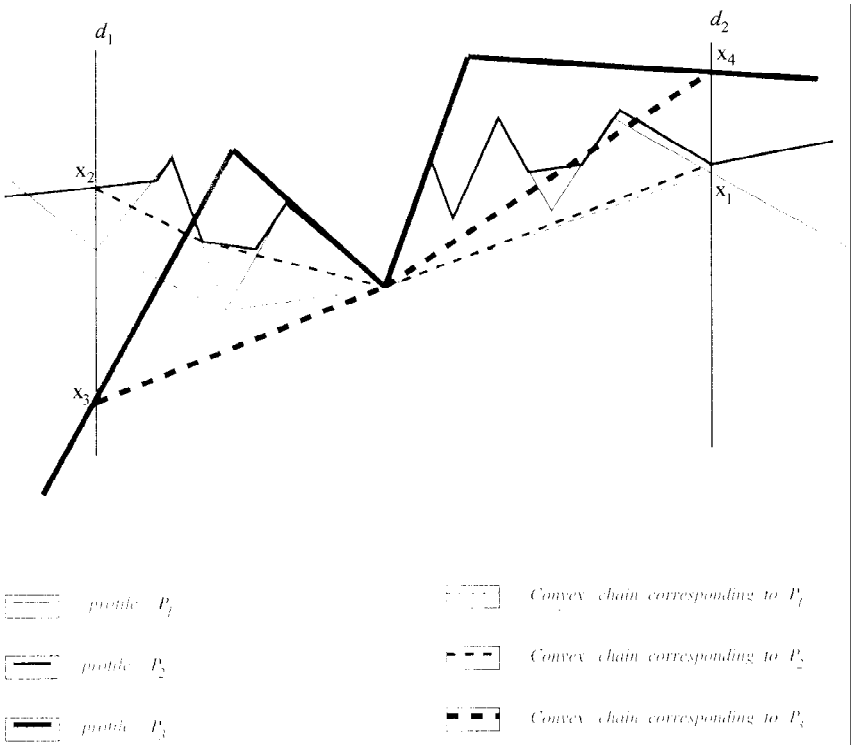

Fig. 10

PCT using the procedure outlined in Lemma 4.3. To find the intersection(s) of a segment s with a profile P_i we store the lower convex chain of the vertices of P_i between d_1 and d_2 for every edge d_1d_2 of CG. Since all the profiles computed up to a fixed layer of the PCT participate in detecting the intersections at the next layer, we must therefore keep a lower convex chain corresponding to each profile computed so far with every edge of CG, so that the proper chain is searched for intersection at the next layer (see Figure 10).

Convex chains are computed using divide-and-conquer. To compute the convex chain of the set $V_i(d_1, d_2)$ of the vertices of P_i between d_1 and d_2, divide $V_i(d_1, d_2)$ into halves, compute the convex chains recursively for each half and merge them as follows: let C_1 and C_2 be the convex chains of the halves. Define a lower common tangent between them as the common tangent of C_1 and C_2 such that both C_1 and C_2 lie above it. We omit "lower" in the following description. Let pq be the common tangent between C_1 and C_2. Then the convex chain of $V_i(d_1, d_2)$ is obtained by deleting the parts of C_1 and C_2 lying above pq and concatenating the remaining parts together with pq.

To construct the convex chain for P_i, we first construct a binary tree, denoted by $BT(P_i)$, which provides a skeleton to compute the convex chain recursively using divide-and-conquer. $BT(P_i)$ is thus a binary tree on $V_i(d_1, d_2)$. We want to compute $BT(P_i)$ for each P_i. Here again we confront the problem of storing the common visible vertices that

may be shared among the profiles. We cannot afford to keep multiple copies of a vertex. Here, we use a shared data structure along the lines of a *persistent binary tree* structure [DSST] to store $BT(P_i)$ for all P_i. We denote this structure by $PBT(d_1, d_2)$. Each node of $PBT(d_1, d_2)$ is labelled with an interval (called the *time stamp*) $[a, b]$ if it belongs to $BT(P_i)$ for all $i \in [a, b]$ such that P_i has been computed.

We start by labelling all the vertices between d_1 and d_2. To compute $PBT(d_1, d_2)$ for all the edges d_1d_2 of ACG, we must label all the vertices computed up to layer L of the PCT. At a fixed layer of the PCT, a vertex is labelled with an interval $[a, b]$ if it belongs to P_i which has been computed for all $i \in [a, b]$. As mentioned earlier, the segments may be shared between the intermediate profiles among the layers of the PCT (step 2(a), Figure 6), therefore a vertex may be detected repeatedly as we go down the PCT. Thus, we may have to update the labels of the vertices as we go down the PCT.

Below we explain how to **label the vertices** at a fixed layer of the PCT and how to update them if required, as we go down the PCT.

4.3.1.2.A. *Labelling the vertices.* We label a vertex with an interval $[i, j]$ if it was detected in the ith profile P_i and deleted in P_{j+1}. If the vertex has not been deleted, then j is a very big number, say M ($> n^2$). When a vertex is created (detected for the first time) in P_i it is labelled $[i, M]$. We update the labels of the vertices as we go down the PCT as follows: Suppose at some node u of the PCT we compute P_j from P_i. Let x be a vertex (created earlier) with label $[l, r]$:

1. If x is detected again in P_j (clearly, x is not a vertex in P_i for this case) and $j < l$, then update the label to $[j, r]$.
2. If x is a vertex of P_i and it is deleted by P_j, and $j < r + 1$, then update the label to $[l, j - 1]$.

Let π_{ij} denote the intermediate profile which is merged with P_i to compute P_j. In Figure 11 there may be a segment s belonging to π_{ir}, π_{is}, and π_{it}.

EXAMPLE 1. Let v be an intersection of s with P_{24}. It is labelled

(a) $[28, M]$ at layer l^2, assuming it has not been created earlier,
(b) updated to $[26, M]$ at layer l^3 if it is not deleted by P_{30} or to $[26, 29]$ if it is deleted by P_{30}, and
(c) further updated to $[25, M]$ or to $[25, 29]$ respectively at layer l^4.

EXAMPLE 2. Let w be a vertex with label $[l, r]$ ($l \leq 24$ and $r > 27$) in P_{24} which is deleted due to s. Its label is updated to

(a) $[l, 27]$ at l^2,
(b) further updated to $[l, 25]$ at l^3, and
(c) then to $[l, 24]$ at l^4.

Thus, at a fixed layer of the PCT, the label $[i, j]$ of a vertex just means that the vertex is visible in all the profiles between and including P_i and P_j and not visible in other profiles only at that layer of the PCT. Nothing can be said about a profile that has not been computed so far (see Example 3 below). Notice that these are precisely the profiles which are required for detecting the intersections at the next layer.

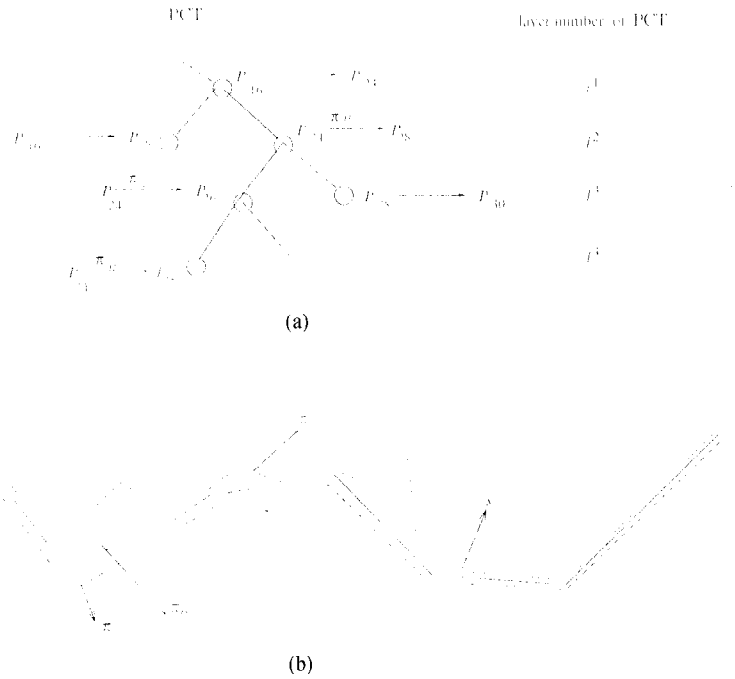

Fig. 11. (a) The PCT. (b) Segment s is shared among the profiles π_{ir}, π_{is}, and π_{it}.

EXAMPLE 3. In Figure 11 a vertex with label [16, 27] at layer l^2 is visible in P_{16}, P_{20}, and P_{24} but not in P_{28}. For other profiles (specifically those between P_{24} and P_{28} and those before P_{16}) we cannot say anything. Similarly, a vertex with label [26, 29] at l^3 is visible in P_{26} and P_{28} but not in P_{24} and P_{30}. We cannot say anything for P_{25} and P_{29}.

4.3.1.2.B. *Building the data structure for intersection detection.* At a fixed layer of the PCT we do the following:

1. Construct CG on the intersections computed so far.
2. For every edge d_1d_2 of CG compute
   (a) $PBT(d_1, d_2)$, and
   (b) the convex chains.
3. Compute the intersections for the next layer and label them. Also, update the labels of the old vertices.

Notice that although computing the intersections should be the first step at any layer it can also be thought of as the last step at the previous layer. We are doing so just for ease of presentation. In this section we mainly discuss step 2. The details follow.

1. **Constructing the CG.** For m vertices, we can construct the CG data structure in $O(\max\{\log m, t_{p,m} + m \log m/p\})$ time using p processors on a CREW PRAM by the method of Lemma 4.3.

2. (a) **For an edge d_1d_2 of CG compute $PBT(d_1, d_2)$.** We compute $BT(P_i)$ for all the profiles P_i computed so far, i.e., $PBT(d_1, d_2)$ using divide-and-conquer on the set $V(d_1, d_2)$ of all the vertices between d_1 and d_2. Let the diagonal d divide $V(d_1, d_2)$ in halves, compute $PBT(d_1, d)$ and $PBT(d, d_2)$ recursively and merge them as explained below. Thus $PBT(d_1, d_2)$ can be computed in $O(\log k')$ stages where k' is the number of vertices between d_1 and d_2. **Invariant:** The root nodes of $PBT(a, b)$ are labelled with disjoint and consecutive intervals of the form $[i_1, i_2]$, $[i_2+1, i_3]$, $\ldots$, $[i_{r-1}+1, i_r]$ for all pairs of diagonals a, b at all stages. These intervals correspond to the profile stamps, i.e., an interval $[a, b]$ correspond to all the profiles P_i, $i \in [a, b]$. This is obviously true at stage 1 where we have k' persistent trees, each with a single root node (and consisting of just a singleton vertex) with an interval label. $PBT(d_1, d)$ and $PBT(d, d_2)$ are merged as follows: Let the root nodes of $PBT(d_1, d)$ and $PBT(d, d_2)$ each be labelled with intervals of the form $[i_1, i_2]$, $[i_2 + 1, i_3]$, $\ldots$, $[i_{r-1} + 1, i_r]$. Let $[l_1, r_1]$ and $[l_2, r_2]$ be the union of intervals labelling the roots of $PBT(d_1, d)$ and $PBT(d, d_2)$, respectively. Introduce two hypothetical left endpoints $r_1 + 1$ and $r_2 + 1$ (in case $r_1 = r_2$, only one such point is introduced) and merge the left endpoints of all the intervals together with $r_1 + 1$ and $r_2 + 1$. Let $i'_1, i'_2, \ldots, i'_s$ be the merged sequence. Create $s - 1$ nodes $v_1, v_2, \ldots, v_{s-1}$ with labels $[i'_1, i'_2 - 1]$, $[i'_2, i'_3 - 1]$, $\ldots$, $[i'_{s-1}, i'_s - 1]$, respectively. Let the left (right) child of a node v_r be the root of $PBT(d_1, d)$ ($PBT(d, d_2)$) containing the label $[i'_r, i'_{r+1} - 1]$ of v_r. However, if such a node (whose label contains the label of v_r) does not exist in $PBT(d_1, d)$ ($PBT(d, d_2)$), then the left (right) child of v_r is nil. This can happen with the nodes in the left end or the right end of the sequence $v_1, v_2, \ldots, v_{s-1}$. For example, if i'_1, i'_2, i'_3 are left endpoints in say $PBT(d_1, d)$, then the nodes v_1, v_2 have their right children nil. The hypothetical points have been introduced to take care of the label of the last node(s) (see Appendix B for details).

CLAIM 4.1. *Let v_r be a root node of $P = PBT(d_1, d_2)$ with label $[a, b] = [i'_r, i'_{r+1} - 1]$. Then the binary tree rooted at v_r, denoted by T_{v_r}, represents $BT(P_i)$ for all $i \in [a, b]$.*

PROOF. We prove our claim by proving the following: Let $i \in [a, b]$.

(a) A vertex (of profiles) which is stored at a leaf of T_{v_r} has a label that includes i (i.e., a vertex in T_{v_r} is also in $BT(P_i)$).
(b) A vertex (of profiles) whose label includes i belongs to T_{v_r}, i.e., a vertex in $BT(P_i)$ is also in T_{v_r}.

Part (a) follows by induction by observing that the label of v_r includes i implies that the labels of the left and the right subtrees of v_r also include i. We prove (b) by contradiction. It is easy to see that if x_j is a node created at stage j and whose label contains i but x_j does not belong to T_{v_r}, then there there exists a node x_{j+1} created at stage $j + 1$ satisfying the same. Thus, if x is a vertex whose label includes i and x does not belong to T_{v_r}, then by induction there exists a root node v of $PBT(d_1, d_2)$ satisfying the same. This contradicts the fact that the root nodes of $PBT(d_1, d_2)$ are labelled by disjoint intervals. □

CLAIM 4.2. *The number of nodes created at stage i is $O(k')$.*

Recall that k' is the number of vertices between d_1 and d_2.

PROOF. Consider $PBT(d_1, d_2)$ as a binary tree where each node u is split into at most $N_l + N_r + 1$ nodes where N_l and N_r are the number of nodes the left and the right children (respectively) of u are split into. Let this number denote the size of u. At the leaf level or at stage 1 we have k' nodes, each of constant size, where k' is the total number of vertices between d_1 and d_2. At stage i we have $k'/2^i$ nodes, each of size $O(2^{i-1} - 1)$. Therefore total number of nodes at stage i is $O(k')$. □

It follows that the total size of $PBT(d_1, d_2)$ is $O(k' \log k')$.

LEMMA 4.4. *$PBT(d_1, d_2)$ can be computed in $O(\max\{\log^2 k', k' \log^2 k'/p + t_{p,k'} \log k'\})$ time with p processors on a CREW PRAM, where k' is the number of vertices between d_1 and d_2.*

PROOF. Consider $PBT(d_1, d_2)$ as a binary tree as explained in the above proof. Let N_l and N_r be the sizes of the left and the right children of the node u at stage i. Lower limits of the intervals labelling the $N_l + N_r$ nodes can be merged in $O(\log(N_l + N_r))$ time using $O(N_l + N_r)$ processors. $N_l = N_r = O(2^{i-1} - 1)$ for all the $k'/2^i$ nodes. Thus at stage i, we have $k'/2^i$ merging tasks, each of which performs merging in $O(i)$ time using $O(2^i)$ processors. Thus by Lemma 3.6 the total time of the construction of $PBT(d_1, d_2)$ is $O(\max\{\log^2 k', k' \log^2 k'/p + t_{p,k'} \log k'\})$ using p processors. □

(b) **Computing the convex chains.** Every node of $PBT(d_1, d_2)$ stores a convex chain of some vertices (that correspond to the leaves of the subtree rooted at the node) between d_1 and d_2. We store this chain in a binary tree and we use the terms "root of the tree," the "tree," and the "convex chain" it represents interchangeably. Each node of the tree represents a lower convex hull[4] of a set of vertices in three parts: two parts are stored in the left and the right subtrees and the third part is an edge connecting the first two parts. We call this edge the "connector." Each node of the tree is labelled with $[p_1, p_2]$ if it represents the part of the chain between p_1 and p_2, i.e., the lower convex hull of the vertices whose y-coordinates are in the interval $[y(p_1), y(p_2)]$ and also store in it the connector pq between the convex subchains stored in its left and right subtrees (see Figure 12). These are used for the binary search on the convex chains. This structure is similar to the dynamic convex-hull data structure of Overmars–Leeuwen [OL].

For a node u of $PBT(d_1, d_2)$ let

- S_u denote the convex chain at u, i.e., the convex chain of the vertices which are at the leaves of the subtree rooted at u,
- l_u and r_u be the left and the right children of u in $PBT(d_1, d_2)$,
- $p_u q_u$ be the common tangent between S_{l_u} and S_{r_u}, and
- L_u, R_u be the parts of S_u to the left of p_u and the right of q_u, respectively. That is, L_u, R_u are the left and the right subtrees of the root of S_u.

[4] A lower convex hull is the part of C extending from the point with the minimum y-coordinate to the point with the maximum y-coordinate in counterclockwise direction.

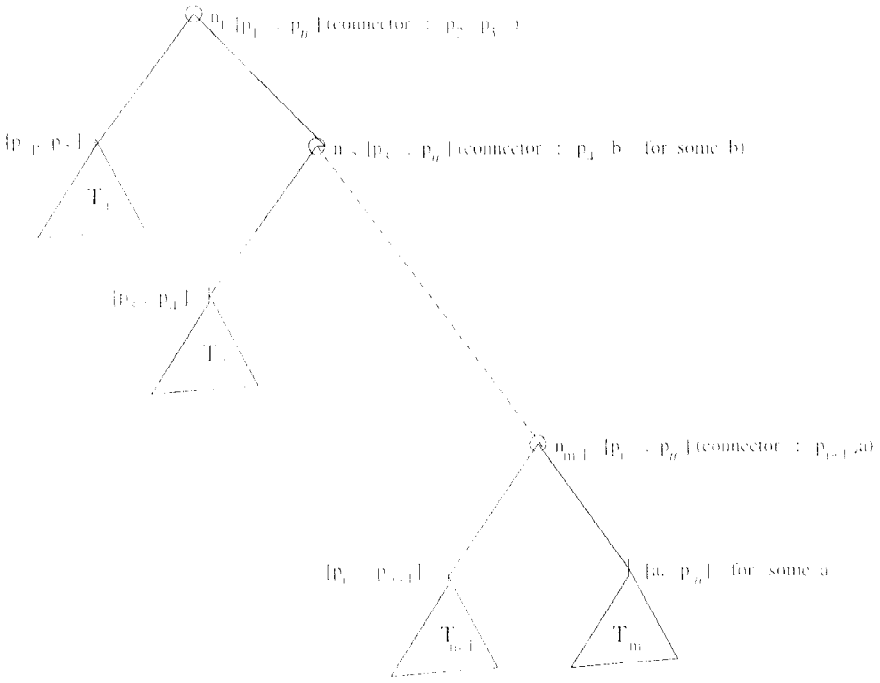

Fig. 12. L_u: part of the convex chain at u toward the left of the common tangent stored at u.

Let u be a node created at stage l of $PBT(d_1, d_2)$. Recall that $PBT(d_1, d_2)$ is constructed bottom-up, therefore stage l represents the distance from the leaves. Given S_v for all the nodes v created at stages $\leq l$ compute S_u as follows: If l_u (or r_u) is nil, then S_u is the same as S_{r_u} (or S_{l_u}, respectively), then stop, else we compute S_u in three parts:

(a) Compute the common tangent $p_u q_u$ between S_{l_u} and S_{r_u}.
(b) Compute L_u.
(c) Compute R_u.

We store the common tangent and the pointers to L_u and R_u in S_u, and a pointer to S_u in u. To save space one can store S_u in u itself. However, storing it separately makes the analysis easier. The common tangent can be computed by a binary search on S_{l_u} and S_{r_u}. We show how to compute L_u. R_u can be computed analogously. Initially L_u is empty. To compute L_u, split S_{l_u} on p_u as follows: Let $v = l_u$,

If $p_v = p_u$
then *Paste* (append) L_v to L_u and stop (Paste is explained later)
else
if the common tangent at v is to the left of p_u (i.e., L_u contains the common tangent at S_v) then

- Paste L_v to L_u.
- SPLIT_1(R_v, p_u).
- else SPLIT_1(L_v, p_u).

SPLIT_1(root,p) is a recursive procedure as given below:

SPLIT_1(root,p)

If root has label $[a, p]$ for some a (for R_u we may be looking for a label $[p, b]$ for some b), then Paste root to L_u and stop, else if $p \in$ right child r of the root, then

- Paste the left child l of root to L_u.
- SPLIT_1(r, p).
- else SPLIT_1(l, p).

"root" above is the root of a tree representing some convex subchain of S_v.

Notice that in the above procedure we are pasting roots of some trees (subtrees of S_v) representing the convex subchains of S_v, to L_u. Also notice that we paste at most one convex subchain (or subtree) from each level of S_v except possibly at the last level, i.e., the last two subchains may come from the same level of S_v. Let $T_1, T_2, \ldots, T_m$, $m = O(\textit{height of } S_v)$, be the trees pasted to L_u in this order. Let $ht(T_i)$ denote the height of T_i, $\forall i$. Then,

$ht(T_i) \leq ht(T_{i-1}) - 1$.

Paste T_i to L_u as follows (see Figure 12):

(a) If $i \leq m - 1$, create a node n_i with right child nil and left child T_i, label n_i with $[p_i, p_u]$ where $[p_i, c]$ is the label of the root of T_i (for some c). n_1 is the root of the tree representing L_u.
(b) If $2 \leq i \leq m - 1$, make n_i the right child of n_{i-1}. Let $[a_1, b_1]$ and $[a_2, b_2]$ be the labels of T_{i-1} and the label of n_i, respectively, then store the connector b_1a_2 in n_{i-1}.
(c) If $i = m$, make T_m (the one with label $[a, p_u]$ for some a) the right child of n_{m-1}. Let $[a_1, b_1]$ and $[a_2, b_2]$ be the label of T_{m-1} and the label of T_m, respectively, then store the connector b_1a_2 in n_{m-1}.

To paste to R_u, interchange left and right in the above steps.

CLAIM 4.3. *The height of the tree rooted at n_1 is $ht(T_1) + 1$.*

PROOF. By induction one can show that the height of the tree rooted at n_i is $ht(T_i)+1$. See Appendix C for details. □

Also, the height of S_u is clearly O(height of $S_v + 1$).

Thus, L_u can be computed in $O(m) = O(l)$ time sequentially. Recall that u is a node of $PBT(d_1, d_2)$ created at stage l. Similarly, R_u (and hence S_u) can be computed in $O(l)$ time sequentially.

REMARK. We had observed earlier that a ray intersects a profile P_i between diagonals d_1 and d_2 if and only if it intersects the lower convex chain of the vertices of P_i between d_1 and d_2. Notice that d_1 and d_2 were the vertices of P_i, so that the convex chain extends from d_1 to d_2. However, now d_1 and d_2 are not necessarily the vertices of P_i. Therefore we introduce the following points for the purpose of computing the convex chains (see Figure 10): Let p_i (q_i) be the leftmost (rightmost) point of P_i

between d_1 and d_2. Let e_l (e_r) be the left (right) edge incident on p_i (q_i). Find out the intersection of e_l (e_r) with d_1 (d_2) and introduce these points with the same label as that of p_i (q_i). This is easy to do. Follow the left (right) splines of the roots of $PBT(d_1, d_2)$ and introduce these points as the left (right) children of the last nodes. The total number of the vertices become at most thrice. In Figure 10, x_1 (respectively x_2) is an intersection of the rightmost (respectively leftmost) edge of P_1 (respectively P_2) between d_1 and d_2 with the right diagonal d_2 (respectively left diagonal d_1). Similarly, x_3 and x_4 are the intersections of the leftmost and the rightmost edges, respectively, of P_3 with d_1 and d_2, respectively. Hence we have the following lemma:

LEMMA 4.5. *Let u be a node created at stage l of $PBT(d_1, d_2)$, then given S_{l_u} and S_{r_u}, S_u can be computed in $O(l)$ time sequentially. Also, the height of the tree representing S_u is $O(l)$.*

Since the convex chains S_v (for the nodes v created earlier) that are used for the construction of S_u are not destroyed at any step of the computation of S_u, then S_u can be computed independently for all nodes u created at stage l of $PBT(d_1, d_2)$ in parallel.

4.3.1.3. *Putting the pieces together.* Suppose we have computed all the intersections up to layer L of the PCT—the number of intersections computed so far is $O(k)$. Let $t_{p,r}$ be the time to solve the processor allocation problem of size r with p processors. Then the ACG at a fixed layer of the PCT is constructed as follows:

1. **Compute CG:** $O(\max\{\log k, t_{p,k} + k \log k/p\})$ time using p processors by Lemma 4.3.
   For each edge d_1d_2 of CG:
   (a) **Compute $PBT(d_1, d_2)$:** $O(\max\{\log^2 k', t_{p,k'} \log k' + k' \log^2 k'/p\})$ time with p processors, where k' is the total number of vertices between d_1 and d_2 (Lemma 4.4).
   (b) **Compute the convex chains:** consider stage i of $PBT(d_1, d_2)$. Let n_i be the total number of nodes created at stage i of $PBT(d_1, d_2)$. Then for each u created at stage i, S_u can be computed in $O(i) = O(\log k')$ time sequentially (Lemma 4.5). Since the size of $PBT(d_1, d_2)$ is $O(k' \log k')$, by Lemma 3.5 the convex chains at all the nodes of $PBT(d_1, d_2)$ can be computed in $O(\max\{\log^2 k', t_{p,k'} \log k' + k' \log^2 k'/p\})$ time using p processors in a CREW PRAM. The term $t_{p,k'}$ follows from the term t_{p,N_i} for phase i in the proof of Lemma 3.5. Also, the maximum height of any of the trees representing these convex chains is $O(\log k')$ from Lemma 4.5.
2. **Compute the convex chains for all the edges of CG** by proceeding level by level from the leaf up to the root using the following observation:
   Let a, b, c be the nodes of CG as shown in Figure 13.

$$PBT(a, b) = \text{Merge}(PBT(b, c), PBT(a, c)),$$

   where Merge($PBT(b, c)$, $PBT(a, c)$) is:
   Split the intervals labelling the roots of $PBT(b, c)$ and $PBT(a, c)$ and create new nodes for $PBT(a, b)$ with the newly created intervals as their labels (as explained earlier in the beginning of Section 4.3.1.2.B). Suppose we have computed $PBT(d_1, d_2)$ (together with the convex chains) for all nodes d_2 at levels $\geq j$ (levels closer to leaves)

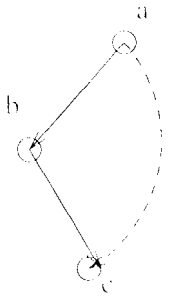

Fig. 13

of CG. Let c be a node at level j of ACG and let a, b, c be as shown in Figure 13. Then, by inductive hypothesis, we have $PBT(b, c)$ and $PBT(a, c)$. We denote them by P^1 and P^2 and the convex chain by S_u for all the nodes u belonging to P^1 or P^2. We denote $PBT(a, b)$ by P. We merge P^1 and P^2 and compute the new convex chains as follows:

(a) Split-up the intervals labelling the roots of P^1 and P^2 and create new nodes for P with the newly created intervals as their labels (as explained earlier).
(b) For each newly created node u above, compute the common tangent between S_{l_u} and S_{r_u} ($l_u \in P^1, r_u \in P^2$).
(c) For each newly created node u, compute the convex chain S_u from S_{l_u} and S_{r_u} (as explained earlier).

Time to merge P^1 and P^2

Let s_l, s_r, s be the sizes of the left and right subtrees of c and the tree rooted at c, respectively. Then $s_l = s_r = O(k/2^j)$ and $s = O(k/2^{j-1})$.

(a) P^1 and P^2 each have $O(k/2^j)$ roots. Recall that the total number of nodes created at any stage of $PBT(d_1, d_2)$ is $O(k')$ where k' is the number of vertices between d_1 and d_2. Hence step (a) above can be done in $O(\log k)$ time using $k/2^{j-1}$ processors.
(b) The number of roots of P is $O(k/2^{j-1})$. The common tangents can therefore be computed with $k/2^j$ processors in $O(\log k)$ time by a binary search on the convex chains associated with P^1 and P^2.
(c) Computing the convex chains also takes $O(j) = O(\log k)$ time with $k/2^{j-1}$ processors by Lemma 4.5.

Therefore P along with the convex chains can be computed in $O(\log k)$ time with $k/2^{j-1}$ processors. The total number of nodes at level j of CG is 2^j and the number of edges incident on each node is two (one tree edge and one superpointer). Therefore by Lemma 3.6, we can compute ACG, i.e., $PBT(d_1, d_2)$ together with the convex chains for all edges d_1d_2 of CG, in $O(\max\{\log^2 k, t_{p,k} \log k + k \log^2 k/p\})$ time using p processors. Hence we have the following lemma:

LEMMA 4.6. *At a fixed layer of the PCT, the ACG structure can be constructed in* $O(\max\{\log^2 k, t_{p,k} \log k + k \log^2 k/p\})$ *time using* p *processors on a CREW PRAM.*

To search whether a ray intersects a profile and detect the first intersection in case it does, proceed level by level of ACG starting from the root, according to the recursive search procedure laid down earlier in Section 4.3.1. At each level, it involves searching whether a ray intersects a convex chain between two diagonals, say d_1 and d_2. To search the convex chain corresponding to a profile P_i, a binary search is done on S_u where u is the root of $PBT(d_1, d_2)$ labelled with an interval containing i. This can be done in $O(\log k)$ sequential time at each level of ACG, i.e., a total of $O(\log^2 k)$ sequential time. Thus, detecting whether a segment intersects a profile and computing its first intersection if it does, takes $O(\log^2 k)$ sequential time. From Lemma 4.2, all k_s intersections of a segment s with a profile can be detected in $O(\max\{\log^2 k + t_{p,k_s}) \log k, k_s \log^2 k/p\})$ time with p processors.

4.3.2. *Detecting intersections at the next layer of the PCT.* At the next layer of the PCT several (actual) profiles are being computed in parallel. Suppose at a node u of the PCT, we compute P_j by merging P_i ($i < j$, P_i inherited from the parent of u) with an intermediate profile π_{ij} precomputed (by the left child of u) in phase 1. For each segment s of π_{ij} we compute the intersection of s with P_i. Some of the vertices of P_i may be deleted as they lie below s and hence do not contribute to P_j. Some new intersections may also be detected. Suppose we have sufficient processors initially to assign two processors to each segment of all the intermediate profiles. More processors are allocated to segments as more intersections are detected as explained in Lemma 4.2. However, now we divide a segment into two subsegments by taking the middle diagonal of the entire set of diagonals (intersections computed so far) spanned by s rather than taking the middle diagonal of the set of vertices of just one profile P_i with which its intersection is to be computed. So the total number of stages is $\log k$. The total number of segments (whose intersections are required to be computed) is $O(n\alpha(n))$ and the total number of intersections is $O(k)$. All the intersections of all the segments can be computed in $O(\log k)$ stages. As in the proof of Lemma 4.2, at each phase we have a number of subsegments—some alive and some dead. Dead subsegments are deleted by an application of compaction. The first intersections can be computed in $O(\log^2 k)$ sequential time and the total number of alive subsegments over all the $O(\log k)$ stages is $O(k + n\alpha(n))$. Thus by Lemma 3.5 (or as done in the proof of Lemma 4.2) all the intersections of all the segments can be computed in $O(\max\{\log^3 k, t_{p,k+n\alpha(n)} \log k + (k + n\alpha(n)) \log^2 k/p\})$ time. Finally the processor allocation problem of size r can be done in $O(r \log r/p)$ time using p processors on a CREW PRAM. The term $t_{p,k+n\alpha(n)} \log k$ is thus subsumed in $(k + n\alpha(n)) \log^2 k/p$. Hence all the intersections at the next layer of the PCT can be computed in $O(\max\{\log^3 n, (k + n\alpha(n)) \log^2 n/p\})$ time using p processors, or all intersections can be detected in $O(\max\{\log^4 n, (k + n\alpha(n)) \log^3 n/p\})$ time over all layers of the PCT using p processors. The new intersections computed at a fixed layer of the PCT can be sorted and merged with the already existing vertices in the required bounds. Update the labels of the repeated vertices and discard the ones with the old labels using compaction.

Detecting the deleted vertices and updating their labels: Observe that at a fixed layer of the PCT a vertex is deleted (and also detected) at most at one node of the PCT, i.e., no two processors attempt to update the label of a vertex at the same time. Label each new intersection point with its segment and the profile. Sort these points on y-

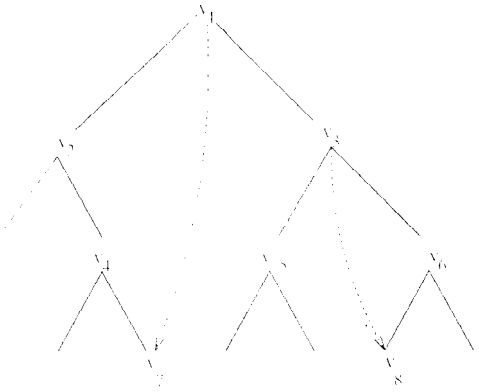

Fig. 14

coordinates and then use a stable sorting algorithm to sort them by their segments. To each consecutive pair of points I_1, I_2 (I_1 to the left of I_2) belonging to the same segment assign one processor. Let $d_1, d_2, \ldots, d_r$ be the diagonals between I_1 and I_2. Let d be the least common ancestor of d_1 and d_r in CG. Then there exists a path from d to d_1 and a path from d to d_r in CG of length $O(\log k)$ consisting only of these diagonals (possibly including shooting pointers)—see Figure 14. Let I_1 lie between the diagonals v_4 and v_7 and let I_2 lie between v_8 and v_6. Then $d_1 = v_7$, $d_r = v_8$, and $d = v_1$ and the required paths are v_1, v_7 and v_1, v_3, v_8). The vertices of P_i lying between I_1 and I_2 are the union of the vertices of P_i lying between these diagonals. The vertices of P_i lying between d'_j and d'_{j+1} are obtained by following the tree rooted at a root node of $PBT(d'_j, d'_{j+1})$ labelled with an interval containing i. Hence the label of a deleted vertex can be updated in $O(\log^2 k)$ time. See Appendix D for details. This completes our description of all the algorithmic steps and we summarize Algorithm **Parhidsurf** below.

Algorithm Parhidsurf

1. Given a two-dimensional surface as a straight line graph in three dimensions, project the line segments on the x–y plane and triangulate the graph.
   Time $= O(\log n)$, processors $= O(n)$ using any efficient algorithm like that in [ACG].
2. Order the edges of the (planar) triangulated graph in the increasing distance from the viewer using the method of [TV]. The ordered set of edges is stored in a *separator tree*.
   Time $= (\log n)$, processors $= O(n)$.
3. For all the nodes of the separator tree do in parallel
   Compute the upper profile of the edges in the leaves of the subtree rooted at the node (the edges in the leaves of the separator tree are sorted in the increasing distance from the viewer).
   Time $= O(\log^2 n)$, processors $= O(n\alpha(n))$ using Lemma 4.1.

4. For each layer L of the PCT do
   (a) Compute the CG structure on all the k intersections computed so far. Time $= O(\max\{\log k, t_{p,k} + k \log k/p\})$ using p processors by Lemma 4.3.
   (b) For each level l of CG build a shared data structure ACG for detecting intersections of a line segment with a monotone polygon:
      (i) For all edges d_1d_2 of CG with d_2 at level l (for all profiles computed so far represented by a common shared data structure) do in parallel
         - Compute $PBT(d_1, d_2)$ (Section 4.3.1.2.B, step 2(a)).
         - Compute all the convex chains (Section 4.3.1.2.B, step 2(b)).
      (ii) Decrement l.

      Time $= O(\max\{\log^2 k, t_{p,k} \log k + k \log^2 k/p\})$ time using p processors by Lemma 4.6.
   (c) Detect the intersections for the next layer, i.e., $L + 1$ of the PCT as follows:

      For all the nodes u at layer $L + 1$ of the PCT do in parallel
      Suppose at node u the profile P_j is computed by merging the intermediate profile π_{ij} with P_i. For all the segments s of π_{ij} do in parallel

         Compute all the intersections of s with P_i as explained in Section 4.3.2 or in the proof of Lemma 4.2. To search the convex chain corresponding to a profile P_i, a binary search is done on the labels of the roots of the PCT corresponding to the interval containing i. The binary tree rooted at that node gives us the corresponding convex chain.

      Time $= O(\max\{\log^3 n, (k + n\alpha(n)) \log^2 n/p\})$ using p processors.
   (d) Increment L.

   All the intersections can be detected in $O(\max\{\log^4 n, (k + n\alpha(n)) \log^3 n/p\})$ time over all layers of the PCT using p processors.

Therefore we arrive at the main result of this paper.

THEOREM 4.1. *The hidden-line elimination problem for terrains can be solved in* $O(\max\{\log^4 n, (k + n\alpha(n)) \log^3 n/p\})$ *time using* p *CREW processors where* n *and* k *are the input and the output sizes, respectively.*

REMARK. For $p = n\alpha(n)/\log n$, the work bound is $O((k + n\alpha(n)) \log^3 n)$ which is within an $O(\log n)$ factor of the sequential bound of Reif and Sen [RS2].

5. Concluding Remarks and Open Problems. We presented an output-size sensitive parallel algorithm for hidden-line elimination for terrain maps. The algorithm provides a solution in a device-independent manner. Our algorithm achieves a work bound of $O((k + n\alpha(n)) \log^3 n$ which is only about an $O(\log n)$ factor away from the sequential running time of Reif and Sen [RS2] and an $O(\log^2 n)$ factor away from that of Overmars

et al. Our algorithm runs in $O(\max\{\log^4 n, k \log^3 n/p\})$ time for $k > n\alpha(n)$ using $p = n\alpha(n)/\log n$ processors in a CREW PRAM.

Our algorithm can be simplified by using an idea of Overmars et al. [OKS]. By "clipping" the intermediate profiles with respect to the actual profiles in the downward phase of our computation, we can avoid the sharing of data structures. This can lead to an improvement in the running time by a logarithmic factor since we may be able to modify the basic algorithm in a way that eliminates the ray-shooting data structure. However, we feel that our approach of using a persistence data structure in a parallel setting is of independent interest and has potential applications to more general scenes.

A natural direction for further work is to generalize the algorithm for any arbitrary three-dimensional scene. However, we will need efficient algorithms for partitioning the scene into disjoint parts such that an ordering of edges is feasible. Moreover, such a partitioning scheme will also have to output-size sensitive.

Appendix A. Compute the Intersections of Profiles. Let $x_1, x_2, \ldots, x_r$ and $y_1, y_2, \ldots, y_s$ be the vertices of profiles p_1 and p_2, respectively. Let $z_1, z_2, \ldots, z_m (m = r + s)$ be the merged sequence. If for some i, z_i and z_{i+1} are vertices of the same profile, say p_2, and their visibilities are same, then the segment $z_i z_{i+1}$ does not intersect the other profile. If their visibilities are different, then let x_r be the predecessor of both z_i and z_{i+1} in p_1, then the segment $z_i z_{i+1}$ intersects the segment $x_r x_{r+1}$ of p_1.

Next, consider the subsequence $x_i, y_j, x_{i+1}, y_{j+1}, x_{i+2}$ of $z_1, z_2, \ldots, z_m$ for some i and j (see Figure 15). Let $Vis(z_i)$ denote the visibility of z_i.

Case 1. If $Vis(y_j) = Vis(y_{j+1}) =$ visible, then $y_j y_{j+1}$ does not intersect p_1.
Case 2. If $Vis(y_j) = Vis(y_{j+1}) =$ invisible, then
  (a) if $Vis(x_{i+1}) =$ invisible, then $y_j y_{j+1}$ intersects both $x_i x_{i+1}$ and $x_{i+1} x_{i+2}$,
  (b) if $Vis(x_{i+1}) =$ visible, then $y_j y_{j+1}$ does not intersect p_1.
Case 3. If $Vis(y_j) \neq Vis(y_{j+1})$, then
  (a) if $Vis(x_{i+1}) =$ invisible, then $y_j y_{j+1}$ intersects $x_{i+1} x_{i+2}$,
  (b) if $Vis(x_{i+1}) =$ visible, then $y_j y_{j+1}$ intersects $x_i x_{i+1}$.

Appendix B. Merge $PBT(d_1, d)$ and $PBT(d_1, d_2)$: Introducing Hypothetical Points $r_1 + 1$ and $r_2 + 1$. Let $r_{\min} = \min\{r_1, r_2\}$ and $r_{\max} = \max\{r_1, r_2\}$. Let $i''_1, i''_2, \ldots, i''_t$ be the merged sequence of left endpoints not including $r_1 + 1$ and $r_2 + 1$.

Case 1: $i''_j \leq r_{\min} < i''_{j+1}$ *for some* $j < t$. Then create nodes with the following labels: $[i''_1, i''_2 - 1], [i''_2, i''_3 - 1], \ldots, [i''_j, r_{\min}], [r_{\min} + 1, i''_{j+1} - 1], [i''_{j+1}, i''_{j+2} - 1], \ldots, [i''_{t-1}, i''_t - 1], [i''_t, r_{\max}]$.
Case 2: $r_{\min} \geq i''_t$. Then create nodes with the following labels: $[i''_1, i''_2 - 1], [i''_2, i''_3 - 1], \ldots, [i''_{t-1}, i''_t - 1], [i''_t, r_{\min}]$ and one more node with label $[r_{\min} + 1, r_{\max}]$ if $r_{\min} \neq r_{\max}$.

It is easy to see from above that the introduction of hypothetical points $r_1 + 1$ and $r_2 + 1$ takes care of both the conditions above.

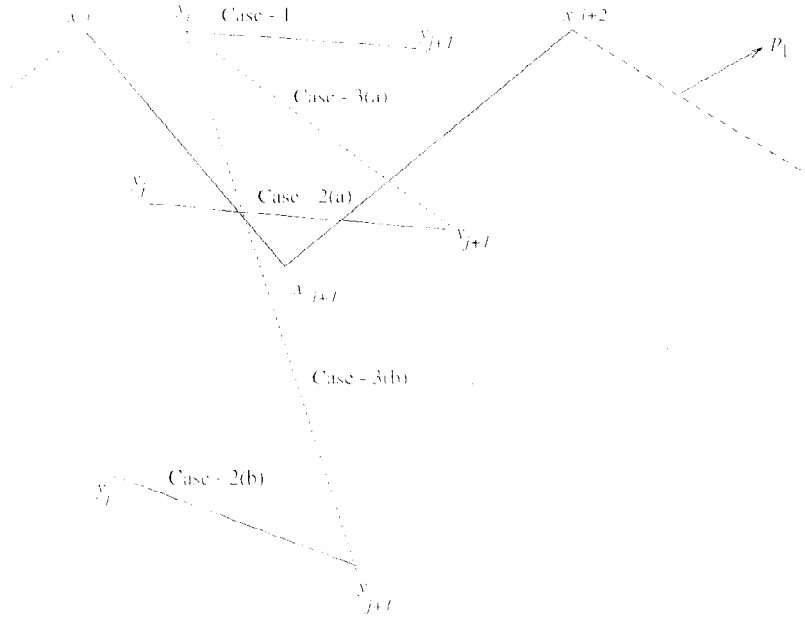

Fig. 15. Let $y_j y_{j+1}$ be one of the segments of profile p_2. Case 1: both y_j and y_{j+1} are visible and the segment $y_j y_{j+1}$ does not intersect the profile p_1. Case 2: both y_j and y_{j+1} are invisible. (a) x_{i+1} is invisible and the segment $y_j y_{j+1}$ intersects both the segments $x_i x_{i+1}$ and $x_{i+1} x_{i+2}$ of the profile p_1. (b) x_{i+1} is visible and the segment $y_j y_{j+1}$ does not intersect the profile p_1. Case 3: y_j is visible and y_{j+1} is invisible. (a) x_{i+1} is invisible and the segment $y_j y_{j+1}$ intersects the segment $x_{i+1} x_{i+2}$ of the profile p_1. (b) x_{i+1} is visible and the segment $y_j y_{j+1}$ intersects the segment $x_i x_{i+1}$ of the profile p_1.

Appendix C

CLAIM. *The height of the tree rooted at* $n_i = ht(T_i) + 1$.

PROOF. Let $ht(n_i)$ denote the height of the tree rooted at n_i. Since $ht(T_m) \leq ht(T_{m-1}) - 1$, $\max\{ht(T_m), ht(T_{m-1})\} = ht(T_{m-1})$. Thus $ht(n_{n-1}) = ht(T_{m-1}) + 1$. The claim is thus true for $i = m - 1$. Suppose it is true for all $i > j$, then

$$ht(n_j) = \max\{ht(T_j), ht(n_{j+1})\} + 1 = \max\{ht(T_j), ht(T_{j+1}) + 1\} + 1 = ht(T_j) + 1.$$

Hence proved. □

Appendix D. Computing the Vertices of a Profile Lying Below a Subsegment. To each consecutive pair of points belonging to the same segment, assign one processor. For the pair I_1, I_2, each labeled with (s, P_i), where I_1 is to the left of I_2, do a binary search on the tree edges of CG. Let d_1 be the leftmost diagonal to the right of I_1 and let d_2 be the rightmost diagonal to the left of I_2. Let d be the least common ancestor of d_1 and d_2. Let $d = v_1, v_2, \ldots, v_j = d_1$ be a path of tree edges in CG from d to d_1. Let $v'_1, v'_2, \ldots, v'_r$

be the nodes in order on this path to the right of I_1. These are the diagonals between d_1 and d. Then there exists a path $d = v'_1, v'_2, \ldots, v'_r = d_1$ in CG (possibly including superpointers). Similarly, there exists a path $d = w'_1, w'_2, \ldots, w'_s = d_2$ in CG of the diagonals between d and d_2. For all v'_l and for all w'_m, $PT(v'_l, v'_{l+1})$ and $PT(w'_m, w'_{m+1})$ contain all the vertices of P_i between v'_l, v'_{l+1} and between w'_m, w'_{m+1}, respectively. Let $PT(d'_1, d'_2)$ be under consideration. Let v be the root of $PT(d'_1, d'_2)$ with label $[a, b]$ containing i. By following a path from v down to a leaf one gets to a vertex of P_i between I_1 and I_2. If it lies above s, then no point of P_i is deleted by the subsegment $I_1 I_2$ of s, hence stop, else continue as follows. Let $n_{[a,b]}$ be the number of vertices between d'_1 and d'_2 whose labels contain $[a, b]$. This is precisely the number of vertices of P_i between d'_1 and d'_2. This number can be computed while building $PT(d'_1, d'_2)$ and stored at the root v. Thus with $n_{[a,b]}$ processors (or one processor for each deleted vertex), labels of all the deleted vertices between d'_1 and d'_2 can be updated in $O(\log k)$ time. Notice that a vertex may be counted more than once. However, since a vertex is deleted at most at one node of the PCT at a fixed layer, the total processor requirement is $O(k)$. Thus all the vertices of the profile P_i lying below the subsegment $I_1 I_2$ of s are deleted in $O(\log^2 k)$ time ($O(\log k)$ time for each pair $v'_l v'_{l+1}$ and $w'_m w'_{m+1}$ which are $O(\log k)$ in number) with one processor for each deleted vertex.

References

[ACG] M. J. Attalah, R. Cole, and M. T. Goodrich. Cascading divide-and-conquer: a technique for designing parallel algorithms. *Proceedings of the* 28*th IEEE Symposium on Foundations of Computer Science*, pages 151–160, 1987.

[AGO] M. Attalah, M. Goodrich, and M. Overmaars. An input-size/output-size trade-off in the time-complexity of rectilinear hidden-surface removal. *Proceedings of ICALP*, 1990.

[AM] P. K. Agarwal and J. Matoušek. Ray shooting and parametric search. *SIAM Journal on Computing*, 22(4):794–806, 1993.

[Be] M. Bern. Hidden surface removal for rectangles. *Proceedings of the* 4*th ACM Symposium on Computational Geometry*, pages 183–192, 1988.

[Br] R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 201–208, 1974.

[CG] B. Chazelle and L. Guibas. Visibility and intersection problems in plane geometry. *Proceedings of the ACM Symposium on Computational Geometry*, pages 135–146, 1985.

[CS] R. Cole and M. Sharir. Visibility problems for polyhedral terrains. Tech. Report No. 92, Courant Institute of Mathematical Sciences, 1986.

[D] F. Devai. Quadratic bounds for hidden-line elimination. *Proceedings of the* 2*nd Annual Symposium on Computational Geometry*, pages 269–275, 1986.

[dBHO$^+$] M. de Berg, D. Halperin, M. Overmars, J. Snoeyink, and M. van Kreveld. Efficient ray shooting and hidden surface removal. *Algorithmica*, 12:30–53, 1994.

[DSST] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarzan. Make the data-structures persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989.

[G] M. T. Goodrich. A polygonal approach to hidden-line elimination. *GVGIP*: *Graphical Models and Image Processing*, 54(1):1–12, 1992.

[GGB] M. Ghouse, M. Goodrich, and J. Bright. Generalized sweep methods for parallel computational geometry. *Proceedings of the* 2*nd ACM Symposium on Parallel Algorithms and Architectures*, pages 280–289, 1990.

[GO] R. Guting and T. Ottmann. New algorithms for special cases of the hidden-line elimination problem. *Proceedings of STACS*, pages 161–171, 1985.

[KAG] R. Kosaraju, M. Attalah, and M. Goodrich. Parallel algorithms for evaluating sequences of set-manipulation operations. *Proceedings of the Aegean Workshop on Computing*, pages 1–10. LNCS 319. Springer-Verlag, Berlin, 1988.

[LF] R. Ladner and M. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.

[M] M. McKenna. Worst-case optimal hidden-surface removal. *ACM Transactions on Graphics*, 19–28, 1987.

[N] O. Nurmi. A fast line-sweep algorithm for hidden-line elimination. *BIT*, 25:466–472, 1985.

[OKS] M. Overmars, M. Kartz, and M. Sharir. Efficient hidden surface removal for objects with small union size. *Computational Geometry*: *Theory and Application*, 2:223–234, 1992.

[OL] M. H. Overmars and J. van Leeuwen. Maintenance of configuration in the plane. *Journal of Computer and System Sciences*, 23:166–204, 1981.

[OS] M. Overmars and M. Sharir. Output-sensitive hidden-surface removal. *Proceedings of the* 30*th IEEE Symposium on Foundations of Computer Science*, pages 598–603, 1989.

[PV] F. Preparata and J. Vitter. A simplified technique for hidden-line elimination in terrains. *Proceedings of STACS*, 1992.

[R] J. H. Reif. *Synthesis of Parallel Algorithms*. Morgan Kaufmann, San Mateo, California, 1993.

[RS1] J. H. Reif and S. Sen. An efficient output-sensitive hidden-surface removal algorithm and its parallelization. *Proceedings of the* 4*th Annual Symposium on Computational Geometry*, pages 193–200, 1988.

[RS2] J. H. Reif and S. Sen. An efficient output-sensitive hidden-surface removal algorithm for polyhedral terrains. *Mathematical Computing Modelling*, 21(5):89–104.

[S] A. Schmitt. Time and space bounds for hidden-line and hidden-surface elimination algorithms. *Proceedings of EUROGRAPHICS*, pages 43–56, 1981.

[SSS] R. F. Sproull, I. E. Sutherland, and R. A. Schumacker. A characterization of ten hidden-surface algorithms. *Computing Surveys*, 6(1):1–25, 1974.

[SV] Y. Shiloach and U. Vishkin. Finding the maximum, merging and sorting in a parallel computation model. *Journal of Algorithms*, 2(1):88–102, 1981.

[TV] R. Tamassia and J. S. Vitter. Optimal parallel algorithms for transitive closure and point location in planar structures. *Proceedings of the ACM Symposium on Parallel Algorithm and Architectures*, pages 399–408, 1989.