Algorithmica (1996) 15: 126–153

Algorithmica

Sweep Methods for Parallel Computational Geometry[1]

M. T. Goodrich,[2] M. R. Ghouse,[2] and J. Bright[2]

Abstract. In this paper we give efficient parallel algorithms for a number of problems from computational geometry by using versions of parallel plane sweeping. We illustrate our approach with a number of applications, which include:

- General hidden-surface elimination (even if the overlap relation contains cycles).
- CSG boundary evaluation.
- Computing the contour of a collection of rectangles.
- Hidden-surface elimination for rectangles.

There are interesting subproblems that we solve as a part of each parallelization. For example, we give an optimal parallel method for building a data structure for line-stabbing queries (which, incidentally, improves the sequential complexity of this problem). Our algorithms are for the CREW PRAM, unless otherwise noted.

Key Words. Parallel algorithms, Computational geometry, Constructive solid geometry, Hidden-line elimination, Plane sweeping.

1. Introduction. There are a number of algorithms in computational geometry that rely on the "sweeping" paradigm (e.g., see [20], [34], and [42]). The generic framework in this paradigm is for one to traverse a collection of geometric objects in some uniform way while maintaining a number of data structures for the objects that belong to a "current" set. For example, the current set of objects could be defined by all those that intersect a given vertical line as it sweeps across the plane, those that intersect a line through a point p as the line rotates around p, or those that intersect a point p as it moves through the plane. The problem is solved by updating and querying the data structures at certain stopping points, which are usually called "events." We are interested in the problem of parallelizing sweeping algorithms.

Most previous approaches to parallelizing sweeping algorithms have been to abandon the sweeping approach altogether and to solve the problem using a completely different paradigm. Examples include the line-segment intersection methods of Rüb [46] and Goodrich [23], the trapezoidal decomposition algorithm of Atallah *et al.* [4], the method of Aggarwal *et al.* [2] for constructing Voronoi diagrams, and the method of Chow [16] for computing rectangle intersections. A notable exception, which kept with the plane sweeping approach, were the methods of Atallah *et al.* [4] for two-set dominance counting, visibility from a point, and computing three-dimensional maxima points. In each of these algorithms, Atallah *et al.* adapted the cascading technique used in Cole's merge

[1] This research appeared in preliminary form in *Proc. 2nd ACM Symp. on Parallel Algorithms and Architectures*, 1990, pp. 280–289. The research of M. T. Goodrich was supported by the National Science Foundation under Grants CCR-8810568, CCR-9003299, CCR-9300079, and IRI-9116843. The research of M. J. Ghouse and J. Bright was supported by the NSF and DARPA under Grant CCR-8908092.

[2] Department of Computer Science, The Johns Hopkins University, 34th and Charles Street, Baltimore, MD 21218, USA.

Received May 25, 1993; revised March 12, 1994. Communicated by M. Snir.

sort [17], to achieve a full parallelization of the sequential fractional cascading method of Chazelle and Guibas [15]. This allowed them to parallelize sweeping methods that sweep the objects with a vertical line, maintaining the set of objects cut by the line, and computing an associative function (such as "plus" or "min") on the current set of objects for each event.

In this paper we give methods for parallelizing other types of sweeping algorithms. Specifically, we address problems where the sweep can either be described as a single sequence of data operations or a related collection of operation sequences. The techniques do not depend on the sweep being defined by moving a vertical line across the plane, nor any other specific geometric object for that matter. We study cases where the sweep involves moving a point around a planar subdivision and cases where the sweep can be viewed as involving a number of coordinated line sweeps. We motivate our approach by giving efficient parallel algorithms for a number of computational geometry problems. In these cases the approach of Atallah *et al.* is not in itself sufficient. In particular, we derive the following results:

- *Hidden-surface elimination.* One is given a collection of opaque polygons in $\Re^3$ and asked to determine the portion of each polygon that is visible from $(0, 0, +\infty)$ [22], [47], [50]. We show that this problem can be solved in $O(\log n)$ time using $O(n \log n + I)$ processors) in the CREW PRAM model, where n is the number of edges and I is the number of edge intersections in the projections of the polygons to the xy-plane.
- *CSG evaluation.* One is given a collection of primitive objects, which are either polygons (in the two-dimensional case) or polytopes (in the three-dimensional case), and a tree T such that each leaf of T has an object associated with it and each internal node of T is labeled with a boolean operation (such as union, intersection, exclusive-union, or subtraction) [45], [52], [53]. The problem is to construct a boundary representation for the object described by the root of T. We show that the two-dimensional version of this problem can be solved in $O(\log n)$ time using $O(n \log n + I)$ processors, and we also show how to extend this method to three-dimensional CSG evaluation.
- *Constructing rectangle contours.* One is given a collection of iso-oriented rectangles in the plane and asked to determine the edges of the contour of their union [12], [35], [57], [58]. We show that this problem can be solved in $O(\log n)$ time using $O(n \log n + k)$ work (which is optimal), where k is the size of the output.
- *Rectilinear hidden-surface elimination.* One is given a collection of opaque iso-oriented rectangles in $\Re^3$ and asked to determine the portion of each rectangle that is visible from $(0, 0, +\infty)$ [9], [22], [25], [28], [37], [43]. We show that this problem can be solved in $O(\log^2 n)$ time using $O((n + k) \log n)$ work, where k is the size of the output.

One of the main ingredients in each of our solutions is the use of a parallel data structure of Atallah *et al.* [6] called the *array-of-trees*. We apply this data structure in a variety of ways in order to solve each of the above problems. Interestingly, for each problem, there is some additional difficulty to be overcome in order to apply our general framework, which was not an issue in the original sequential algorithm. In the case of hidden-surface elimination the difficulty is the definition of a comparison rule for polygons that is consistent even if the overlap relation contains cycles. For CSG evaluation the difficulty

involves solving an off-line expression evaluation problem (which is of independent interest). Also, in the three-dimensional case, our method uses a parallel construction of a line-stabbing data structure of Chazelle and Guibas [15], which, incidentally, improves the sequential preprocessing time for constructing this structure. In the case of rectangle contour construction the difficulty is to construct a version of the array-of-trees that allows for optimally reporting all pieces of the output (this modification, which is perhaps the most significant contribution of this paper, involves an interesting "pruning" technique applied to the array of trees). In the rectilinear hidden-surface elimination problem the difficulty involves describing a search procedure so that it only reports one copy of each piece of the output, even though a single piece may be stored in the array-of-trees as $O(\log n)$ separate subpieces.

The computational model we use for our algorithms is the CREW PRAM. Recall that processors in this model act in a synchronous fashion and use a shared memory space, where many processors may simultaneously access the same memory location only if they are all reading that location. Many of our results use the paradigm that the pool of virtual processors can grow as the computation progresses, provided the allocation occurs *globally* [23], [46]. In this scheme r new processors are allowed to be allocated in time t only if an r-element array that stores pointers to the r tasks these processors are to begin performing in step $t + 1$ has already been constructed. This is essentially the same as the traditional CREW PRAM model, except that in the traditional model only one request is preformed, at the beginning of the computation (to allocate a number of processors that usually depends on the input size, e.g., n or n^2). Many of our results can be viewed as showing that when it is wished to run a parallel algorithm on a machine with a fixed number of processors, by simulating an algorithm that uses a dynamically expandable pool of virtual processors, we are able to achieve superior speedups over simulations using processor-static algorithms.

2. Parallel Persistence.

We begin our discussion by reviewing a parallel data structure called the array-of-trees. We then present two extensions to this structure. All of the structures we describe here can be viewed as parallel examples of the persistence paradigm of Driscoll *et al.* [19]. In our framework a linked data structure D [19], an initial assignment of values to the nodes of D, and a sequence σ of m update operations that operate on the nodes of D, but do not add new links[3] to D, are given. The interpretation of the sequence σ is that operation i updates the structure resulting from performing operations $1, 2, \ldots, i-1$. The problem is to produce an auxiliary structure, A, such that A allows a single processor to perform a "query in the past" on D, i.e., a query on the instance of D as it appeared after some update operation i.

As a simple example of this approach, consider the parallel prefix problem [32], [33], where a sequence of numbers $(a_1, a_2, \ldots, a_n)$ is given for which one wishes to compute all the prefix sums $s_k = \sum_{i=1}^{k} a_i$. This can be viewed as a sequence σ of n operations of the form $\sigma_i =$ "$s := s + a_i$," and the computational problem that of building a data structure (i.e., an array) so that the value of s can be quickly determined after executing

[3] In the sequential setting one is also allowed to change links [19].

the sequence $\sigma_1\sigma_2 \cdots \sigma_i$ (assuming s is initially 0). We refer to the resulting array of values for such a variable s as the *event list* for s, and we refer to the ith entry in this array as the value s had at *time* i. Of course, such an array can be constructed in $O(\log n)$ time using $O(n/\log n)$ processors in the EREW PRAM model [32], [33], so that a query "in the past" can be answered in $O(1)$ time.

In the above example the underlying "skeleton" structure was a single variable, s. We show in the following subsections that, for a variety of other skeleton structures, D, if the entire sequence σ is given in advance, then an efficient data structure can be constructed in parallel to allow a single processor to answer queries in the past for D.

2.1. *The Array-of-Trees.* The *array-of-trees* data structure, which we define below, was developed by Atallah *et al.* [6], who were the first to address this problem in a parallel setting. They gave a solution for the case where the underlying data structure is a complete n-node binary tree T and each operation in σ is either an *enable*(v), which "turns on" the leaf v and updates the nodes from v to the root to reflect this, or a *disable*(v), which turns off the leaf v and updates the nodes from v to the root to reflect this. The updating action here is allowed to include, for each node v involved in the update, the computation of a constant number of labeling functions on the children of v. Their method runs in $O(\log n)$ time and $O(m \log n)$ space,[4] using $O(n+m)$ processors in the CREW PRAM model, where $m = |\sigma|$.

DEFINITION. The array-of-trees, which we denote by $B(r)$, is a directed acyclic graph built on an underlying tree T, where T has a list $\sigma(v)$ at each node v. $\sigma(v)$ is the subsequence of σ consisting of all operations whose argument u occurs in the subtree rooted at v (the operations in $\sigma(v)$ occur in the order that they appear in σ). For each operation σ_t in $\sigma(v)$ there is a record (t, *val*, *left*, *right*), where t is the position of this operation in σ (i.e., its "time of execution"), *val* is a value for v, and *left* and *right* are pointers (which are **null** if v is a leaf.) The values stored in the *val* field are the results of the operation σ_t performed "bottom-up" on the values stored in the subtree rooted at v.

For example, if one is interested in counting the number of active leaves, then, for every leaf x, *val* could store "*count* = 0" if σ_t = *disable*(x) and "*count* = 1" if σ_t = *enable*(x). Also, we include a record in $B(x)$ for the initial assignment of x, giving it a time-value $t = 0$. Intuitively, $B(x)$ represents the history of σ when one restricts attention to the operations in $\sigma(x)$. That is, if we let $(t_1, t_2, \ldots, t_{|B(x)|})$ denote the list of t-values in $B(x)$, then each record (t_i, *vals*, **null**, **null**) in $B(x)$ can be thought of as representing a (trivial) binary tree representing the portion of T related to x from time t_i to time $t_{i+1} - 1$.

For each internal node v, $B(v)$ is defined in terms of $B(u)$ and $B(w)$, where u and w are the children of v. There is a record in $B(v)$ for each record in $B(u) \cup B(w)$, and these are sorted by t-values (i.e., a sorted merging of $B(u)$ and $B(w)$), removing the duplicate for $t = 0$. For a record $\alpha = (t, vals, left, right)$ in $B(v)$, the pointers *left* and *right* point to the records α_l and α_r in $B(u)$ and $B(w)$, respectively, with the largest t-value less than or equal to t (one of these records will have the same t-value as α). The values in

[4] If all future queries need go no deeper than the root of T, then the space can be reduced to $O(m)$ [6].

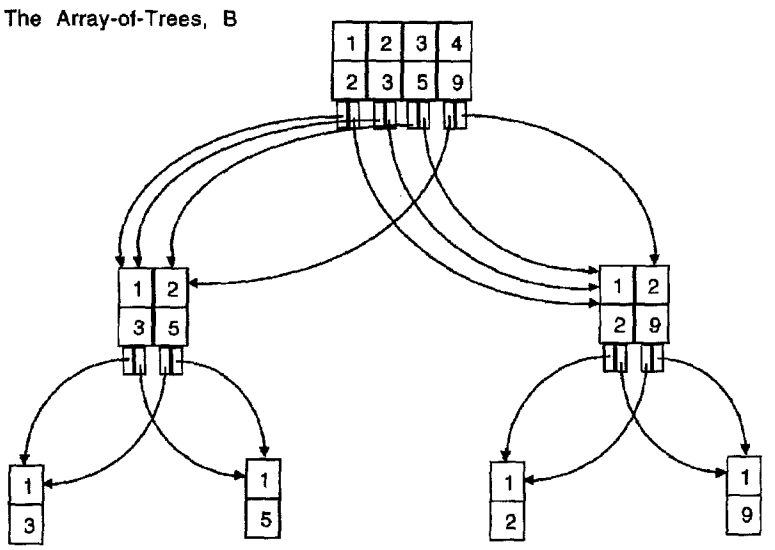

Fig. 1. An array-of-trees.

the *vals* list for α is defined by a combination rule (specified by the application) applied to α_l and α_r. For example, if one is interested in counting active leaves, then there could be a *count* field in *vals* that is computed by taking the sum of *count* fields in α_l and α_r. By a simple inductive argument, if we let $(t_1, t_2, \ldots, t_{|B(v)|})$ denote the list of t-values in $B(v)$, then each record in $B(v)$ represents the root of the subtree of T rooted at v from time t_i to time $t_{i+1} - 1$. (See Figure 1.)

Implicit in the definition of a combining rule is that, for each node v, the rule must specify a combined value for all values that might be stored in α_l and α_r. If each combining rule is defined over all values in the underlying universe, then this requirement presents no problems. However, if it is wished to apply this theorem to solve the hidden-surface elimination problem (as we do), where the natural elements at the leaves of T are the names of polygons in $\Re^3$ and the natural function is "highest polygon," then care must be taken to satisfy this implicit requirement. The main difficulties are that the overlap relation may contain many cycles [41] and some pairs of polygons do not overlap (hence, are incomparable by the "highest polygon" relation). We address these concerns in Section 3.1, where we show how to apply the array-of-trees to the hidden-surface elimination problem.

As a simple example of a use of the array-of-trees, consider the problem of counting the number of intersections between a set of vertical line segments and a set of horizontal line segments. In this case the skeleton tree T is a complete binary tree built on top of the y-coordinates of the horizontal segments, and the x-coordinates of the segment endpoints define the actions, left endpoints corresponding to enable operations and right endpoints corresponding to disable operations. The only information that needs to be stored in *vals* list for a node v is the count of the number of active leaf descendants of v. Given this data structure, the problem is solved by assigning a processor to each vertical segment and using that processor to search in the "copy" of T for the x-coordinate of s. The

search for s is a simple one-dimensional range-query based on the y-coordinates of the endpoints of s.

More complicated types of combining rules can also be used. For example, Goodrich [23] employs a *compressed* version of the array-of-trees where the combining rule affects both the values stored in the *vals* list and the *left* and *right* pointers. In particular, if two records α_l and α_r are combined to define a new record α (where α is for a node v and α_l and α_r are for v's left and right children, respectively), then, in addition to computing a *count* label of the number of active leaf descendants for v, the following test is added:

> If $\alpha_l.\,count = 0$, then $\alpha.\,left = \alpha_r.\,left$ and $\alpha.\,right = \alpha_r.\,right$, else, if $\alpha_r.\,count = 0$, then $\alpha.\,left = \alpha_l.\,left$ and $\alpha.\,right = \alpha_l.\,right$. Also, if $\alpha.\,count = 0$, then $\alpha.\,left = \alpha.\,right =$ **null**.

Note that by adding this simple rule, each record α in a $B(v)$ list represents the root of a tree with $\alpha.\,count$ leaves, i.e., a compressed binary tree built upon the active leaves that are descendants of v in T. Goodrich uses this approach to derive an optimal parallel algorithm for enumerating all intersections between a set of vertical segments and a set of horizontal segments.

2.2. *Extending the Array-of-Trees.* In this paper we make applications of a number of further extensions of the array-of-trees data structure. Here we present an overview of these extensions, which are described in detail, together with their applications, in Section 3. The first extension we add is that we allow each internal node v in the skeleton tree, T, to store data elements as well as the values of combining rules applied to v's children. Thus, we allow the operations in σ to enable and disable internal nodes of T as well as leaves. Using a modified version of the method of Atallah *et al.* [6] this version of the array-of-trees can be constructed with the same performance as before, i.e., $O(\log n)$ time and $O(n \log n)$ space using $O(n)$ processors. In particular, since their method is based upon merging lists in a binary tree, we can easily transform our extension to their framework by viewing the merge at each internal node as a three-way merge and transforming this back to the binary tree framework of Atallah *et al.* For each node v of T, construct the subsequence $\sigma(v)$ of σ consisting of those operations affecting v (as we did previously just for the leaves of T), and construct a simple array-of-leaves list $B'(v)$ for $\sigma(v)$. Then the merge defined for v is that of first merging $B(u)$ and $B(w)$, as before, where u and w are the children of v, followed by the merge of this list and $B'(v)$, to form $B(v)$. Since this modification at most doubles the depth of the tree T, and does not increase its size by more than a constant factor, the running time of this method is still $O(\log n)$ and the number of processors needed is still $O(n)$.

Allowing for internal nodes of T to be enabled and disabled is not the only extension we make, however. We also allow a "pruned" version of the compressed array-of-trees [23]. In particular, we assume the existence of a 0/1-valued *prune function*, $\pi(\alpha, v)$, and modify Goodrich's combining rule so that we use $\pi(\alpha_l, u) * \alpha_l.count$ and $\pi(\alpha_r, w) * \alpha_r.count$ instead of $\alpha_l.count$ and $\alpha_r.count$, respectively. Intuitively, if, say, $\pi(\alpha_l, u) = 0$, then we are "pruning" away the subtree rooted at α_l, and not passing it up to be a part of the subtree rooted at α. Note, however, that we do not destroy the tree rooted at α_l; it is still accessible from $B(u)$. (See Figure 2.) Thus, in this case, each record α in $B(v)$ corresponds to the root of a binary tree containing the number of

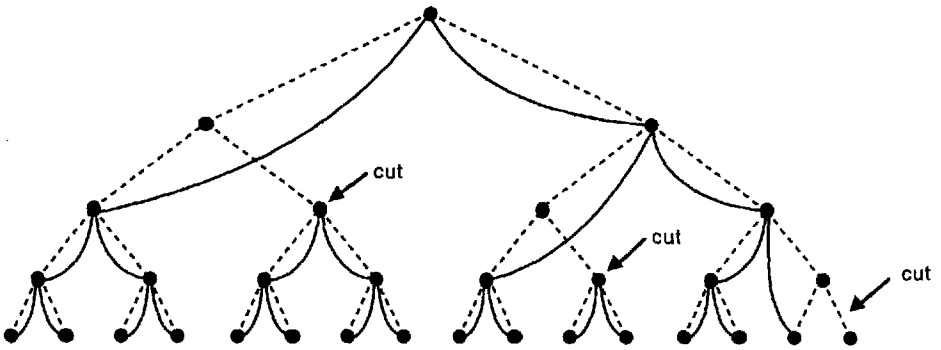

Fig. 2. A "pruned" binary tree. Broken lines indicate the skeletar tree and solid lines indicate the pruned (compressed) tree.

active descendent nodes of v in T that "survived" the pruning function at least as far up T as v.

The final extension we make to the array-of-trees is to develop the skeleton data structure upon which it is defined, so as to be something other than a complete binary tree. In particular, we allow the skeleton structure to be an order-k pseudotree, for fixed k. A *pseudotree* is a directed acyclic graph $G = (V, E)$ such that the nodes in V have been partitioned into $V_1, V_2, \ldots, V_m$ with the V_i's forming the nodes of a binary tree T. For each edge $(v, w) \in E$, either $v, w \in V_i$ for some i or (V_i, V_j) is an edge in T and $v \in V_i$ and $w \in V_j$. A pseudotree is of *order* k if $|V_i| \leq k$ for each $i \in \{1, 2, \ldots, m\}$. Thus, if G is a tree, it is an order-1 pseudotree. For our applications, we assume that the underlying tree, T, is a binary tree with height $O(\log n)$, and that G is an order-k pseudotree with k being $O(1)$. Our approach to constructing $B(r)$, where r is the "root" of G, is as above, except that now the merge at each node is possibly a $(2k + 1)$-way merge and the underlying graph is now a pseudotree, not a tree. Still, using the cascade merging scheme of Goodrich and Kosaraju [26], which is based on linked lists instead of arrays, this "array-of-pseudotrees" data structure can be easily constructed in $O(\log n)$ time and $O(n \log n)$ space using $O(n)$ processors. Note: the method of Atallah *et al.* [6] cannot be applied here, because their method is based on a cascade merging with arrays that would introduce a potentially large number of duplicate entries.

2.3. *Off-Line Expression Evaluation.* As an application of our extensions to the array-of-trees data structure, consider the following problem. Suppose one is given an n-node binary tree T such that each leaf represents a value taken from some universe $\mathcal{U}$ and each internal node v is labeled with a binary function f_v: $\mathcal{U} \times \mathcal{U} \to \mathcal{U}$ taken from a family of functions $\mathcal{F}$. The height of T is allowed to be as large as $O(n)$. The *expression evaluation* problem is to determine the value represented by the root of T based on a bottom-up evaluation. To make the problem tractable in a parallel setting, we assume the functions in $\mathcal{F}$ and the universe $\mathcal{U}$ form a *contractable* algebraic structure, that is, an algebraic structure that satisfies the composition, closure, and combination properties of Miller and Teng [39]. Intuitively, an algebraic structure $(\mathcal{U}, \mathcal{F})$ is contractable if the parallel tree-contraction schemes of Brent [11] or Miller and Reif [38] can be applied

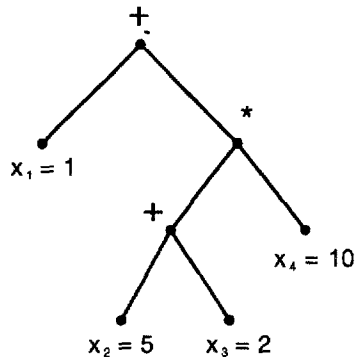

Fig. 3. The off-line expression-evaluation problem.

to evaluate T in $O(\log n)$ time using $O(n)$ processors (which can in fact be reduced to $O(n/\log n)$ [1], [30]). For example, any semiring is contractable [11], [38], as is the algebraic structure defined by the boolean operations used in CSG evaluation (on the universe $\{0, 1\}$) [24].

Suppose that, in addition to the expression tree T, a sequence σ of m update operations defined on the leaves of T are given. That is, each tth operation, σ_t, in σ is an assignment of the form $x_j := u$, where x_j is a leaf of T and u is value taken from U. The *off-line expression-evaluation* problem is to determine for each $t \in \{1, 2, \ldots, m\}$ the value that would be defined by the root of T after sequentially performing the assignments $\sigma_1, \sigma_2, \ldots, \sigma_t$, given the initial values assigned to the leaves of T. (See Figure 3.)

Using the methods of Abrahamson *et al.* [1] or Kosaraju and Delcher [30] T can be converted into an equivalent circuit C, where C has $O(\log n)$ depth and is an order-4 pseudotree. The time needed for this conversion is $O(\log n)$ using $O(n/\log n)$ processors [1], [30] (see also [11] and [24]). Given this circuit, and the initial values associated with its "leaves," we then apply the array-of-pseudotrees construction described above. This requires an additional $O(\log n)$ time using $O(n/\log n + m)$ processors, and gives us a solution to the off-line expression evaluation problem (by simply reading off the values stored at the "root" of C for each time instance in σ). Moreover, since we are only interested in the value of the "root" of C, we need not store all portions of the array-of-pseudotrees, and can implement the construction using only $O(n + m)$ space [26]. Thus, we have the following lemma:

LEMMA 2.1. *Given an n-node binary expression tree T whose operations are taken from a contractable algebraic structure, and an m-operation sequence σ of leaf-update operations, the value associated with the root of T can be determined after performing each operation in σ (as in a sequential evaluation) in $O(\log n)$ time using $O(n/\log n + m)$ processors.*

In the next two sections we address a number of applications of our extensions to the array-of-trees.

3. Sweeping Arrangements. Given a collection C of line segments $\Re^2$, the *arrangement* of C is defined to be the embedded planar graph G whose vertices correspond to the intersection points determined by pairs of segments in C, and such that (v, w) is an edge if there is a segment s in C containing v and w (and there is no other vertex in G between v and w on s). It is also common to add an edge to G from each segment endpoint v to the first vertex hit by a vertical ray emanating upward (resp. downward) from v. Such a graph is a special case of a larger class of graphs, planar subdivisions, defined by a subdivision of the plane into a collection of simple polygons (see [42] and [20]). There are a number of sequential algorithms that follow an approach of constructing an arrangement [20] and traversing that arrangement to solve the problem at hand. We address this approach from a parallel perspective.

One of the main subproblems that we must solve in each application is the construction of a spanning tree in a connected planar subdivision, which the following lemma addresses:

LEMMA 3.1. *Given a connected planar subdivision R, a spanning tree for R can be constructed in $O(\log n)$ time using $O(n/\log n)$ processors in the CREW PRAM model.*

PROOF. The method is quite simple: for each face f in R (other than the external face), remove the edge preceding the leftmost vertex of f in a counterclockwise traversal of f (see Figure 4). This is easily accomplished in $O(\log n)$ time using $O(n/\log n)$ processors via list ranking [3], [18] and parallel prefix computations [32], [33]. We have yet to show that this produces a spanning tree for R, of course. Let T denote the subgraph resulting from this computation.

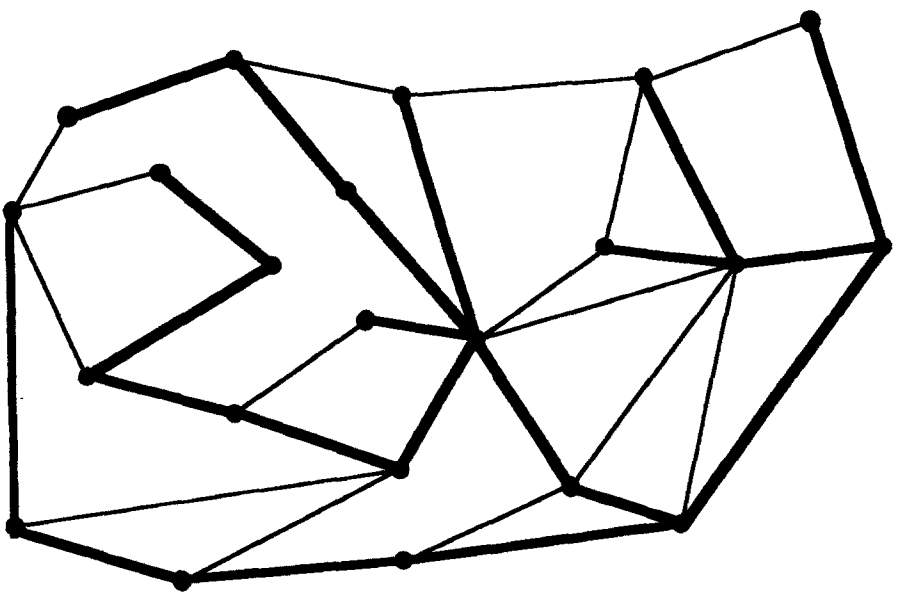

Fig. 4. The spanning tree for a connected planar subdivision. Light lines indicate the subdivision, heavy lines indicate the spanning tree.

CLAIM 1. *T is acyclic.*

PROOF OF THE CLAIM. Suppose, for the sake of contradiction, that there is a cycle $C = (v_0, v_1, \ldots, v_k, v_0)$ in T. Without loss of generality, C is a simple cycle. Let v_i be the leftmost vertex in C. Since C is a simple cycle in a planar subdivision, v_i must also be the leftmost vertex on a face of R other than the external face. However, this implies that (v_{i-1}, v_i) is not an edge in T; hence, C cannot be a cycle. Thus, T is acyclic. □

CLAIM 2. *T is connected.*

PROOF OF THE CLAIM. Suppose, for the sake of contradiction, that T is not connected. Let C_1 and C_2 be two connected components of T such that there is a vertex $v \in C_1$ and a vertex $w \in C_2$ with v and w being adjacent in R. Note that C_1 and C_2 must exist, since R is connected. Since (v, w) is not an edge of T, it must be an edge of a face f of T other than the external face of T. Thus, there must be another edge (v', w') on f with $v' \in C_1$ and $w' \in C_2$. Since (v', w') is not an edge of T, this in turn implies that there is a face $f' \neq f$ containing (v', w') such that f' is not the external face. We can continue this argument, defining a sequence of faces $f_1, f_2, \ldots$ that are adjacent in the planar dual of R. Since R is finite, these faces must form a cycle in the planar dual of R. However, we only remove an edge if it precedes the leftmost vertex on a face. Thus, the leftmost vertices on each of these faces must all have the same x-coordinate (for, otherwise, we have removed an edge preceding a vertex that is not leftmost in some face in this cycle). Moreover, there can be no edge on any face f_i that is incident to a vertex with smaller x-coordinate than this. However, this contradicts the observation that each f_i is not the external face. Therefore, T must be connected. □

Since T is by definition a spanning subgraph of R, these two claims immediately imply that T is a spanning tree for R. □

This lemma can be easily extended to construct a spanning forest of a disconnected subdivision. We leave the details to the interested reader. Having presented this lemma, we now turn to some applications of our parallel plane-sweeping approach.

3.1. *Hidden-Surface Elimination.* The first application we address is the hidden-surface elimination problem. Suppose a collection of polygonal faces in $\Re^3$ that do not intersect (except possibly at boundaries) is given. The problem is to determine the portions of each polygon that are visible from $(0, 0, +\infty)$ assuming each polygonal face is opaque. (See Figure 5.) For simplicity of expression we assume that no two polygon edges (resp. vertices) project to the same edge (resp. vertex) in the projection plane (the xy-plane). Our method can be easily modified for the more general case by using parallel prefix computations where appropriate.

Our method for solving this problem follows the general approach of Goodrich [22] and Schmitt [47]. This approach is based on the construction of the arrangement of polygons determined by projecting the polygons in S to the xy-plane and then traversing this arrangement to solve the hidden-surface elimination problem. This arrangement is

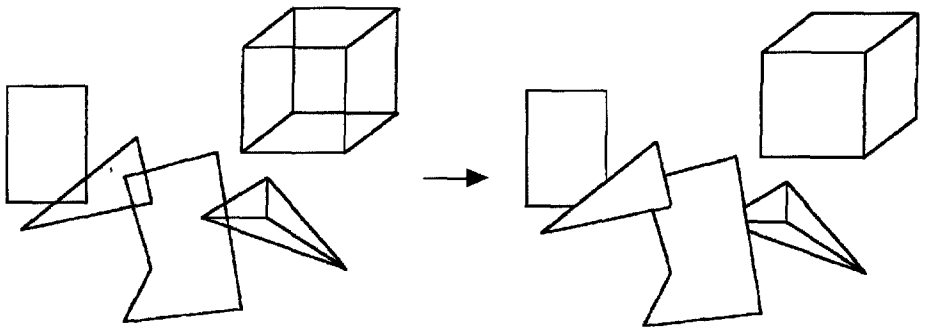

Fig. 5. The hidden-surface elimination problem.

the connected graph whose nodes are the polygon vertices and the intersection points between pairs of (projected) polygonal edges. In addition, for each edge e in the projection plane, we store the name of the polygon P that has an edge projecting to e, and to which side of e the interior of P projects. We say such an arrangement is *polygon-connected* provided two polygons P and Q intersect if and only if the vertices of P and Q are in the same connected component of the arrangement. Of course, this will always be true if the boundaries of P and Q intersect, but may not be the case if, say, Q is properly contained in the interior of P. We can easily force a polygon arrangement to be polygon-connected, however, by drawing an edge from each vertex v to the first point(s) in the arrangement that are hit by vertical rays emanating upward and downward (in the y-direction) from v. Of course, some vertices will have edges to shadows "at infinity," but this presents no difficulties. Our method, then, consists of the following six steps:

Step 1. In this step we construct the polygon arrangement R of S projected to the xy-plane. Using the parallel segment-intersection algorithm of Goodrich [23], along with the shadow-finding algorithm of Atallah *et al.* [4], this arrangement can be constructed in $O(\log n)$ time using $O(n \log n + I)$ processors in the CREW PRAM model.

Step 2. In this step we construct a spanning forest F of R using the method of Lemma 3.1, which implies that this step can be implemented in $O(\log n)$ time using $O((n + I)/\log n)$ processors.

Step 3. In this step we prepare for an application (in Step 4) of a variation on the Euler-tour technique of Tarjan and Vishkin [51] to operation sequences, by constructing an Euler tour of each tree of F. For each connected component of F we make the first edge in the tour of that component an edge leaving the (unique) vertex with the smallest x-coordinate. In addition, with each edge e_i in a tour we associate a point p_i on e_i in the xy-plane (we use these points in Step 4). p_i can be anywhere on the projection of e_i onto the xy-plane. Let U denote the union of these tours. We can easily perform this step in $O(\log n)$ time using $O((n + I)/\log n)$ processors.

Step 4. In this step, from U, we construct a sequence of operations $\sigma = (\sigma_0, \sigma_1, \ldots, \sigma_m)$ that operate on a binary tree T such that each leaf of T is associated with a polygon P. For each edge e_i in U we associate an operation σ_i, where σ_i is **enable**(P) (resp.

disable(P)) if in traversing the projection of e we would enter (resp. leave) the projection of the interior of P. In addition to enabling the polygon P, the **enable**(P) operation assigns the name of a point p on e_i to a label *rep* in the *vals* list for P (this is the *representative* for P for as long as P is active). We also maintain a *max* label for each node v in T (stored in the *vals* list for a record in $B(v)$), which stores the polygon-representative pair (P, p), where P is the "highest" polygon when comparisons are based on the following rule $\mathcal{R}$: Given the query "$(P, p) > (Q, q)$?", return "yes" if and only if the projection of point p onto the plane containing face P is above the projection of p onto the plane containing face Q, where P is the more recently inserted polygon (otherwise, we would use q in this comparison). This step can easily be implemented by performing a list-ranking procedure within the individual tours in U. Using the methods of Cole and Vishkin [18] or Anderson and Miller [3], this requires $O(\log n)$ time using $O((n + I) \log n)$ processors.

Comment. A vertex with the smallest x-coordinate in its component is not contained inside the interior of any polygon projection in the xy-plane. Thus, for each σ_i, the set of active polygons at "time" i consists of all the polygons that contain the projection of edge e_i in the interior of their projection onto the xy-plane, since we start with $\emptyset$ for each such tour.

Step 5. In this step we perform an array-of-trees construction on σ using the labels listed above (with comparison rule $\mathcal{R}$). This step requires $O(\log n)$ time and $O(n + I)$ space using $O(n + I)$ processors. We show below that even if the overlap relationship contains cycles, the computation of the *max* labels still proceeds correctly.

Step 6. For each edge e_i in F, with associated operation σ_i, the **max** label associated with the record for time i stored at the root of T stores the name of the polygon visible along e_i (i.e., the "highest" polygon). If e_i is on or above this polygon, then e_i is visible; otherwise, e_i is invisible. In this step we remove from R all the edges that are invisible, and indicate for each visible edge e_i the polygons of S that are visible on each side of e_i. Given the information computed in previous steps, this step can easily be implemented in $O(\log n)$ time using $O((n + I)/\log n)$ processors.

End of Algorithm.

The correctness of the above algorithm crucially depends on $\mathcal{R}$ being a consistent relation even in the face of possible cycles and gaps in the overlap relationship. That $\mathcal{R}$ is symmetric follows immediately from its definition. The next lemma establishes that $\mathcal{R}$ is also transitive.

LEMMA 3.2. *For any $i \in \{0, 1, \ldots, m\}$, if three polygons, P, Q, and R, are active at time i, then $(P, p) > (Q, q)$ and $(Q, q) > (R, r)$ imply that $(P, p) > (R, r)$, where p, q, and r are the respective representatives for P, Q, and R at time i.*

PROOF. Suppose not. Then P is above Q at p or q, Q is above R at q or r, but P is below R at p or r. Since the polygons in S do not intersect (although their intersections in $\Re^2$ do in this case), this implies that the path in the Euler tour that contains p, q, and r must leave and re-enter P, Q, or R along an edge not containing p, q, or r, respectively. However, this implies that one of (P, p), (Q, q), and (R, r) is not an active polygon-

representative pair at time i, which is a contradiction. For example, suppose the relative order of polygon-point pairs in the tour is $\cdots (P, p) \cdots (Q, q) \cdots (R, r) \cdots$. That is, P is above Q at q, Q is above R at r, while, by assumption, P is below R at r. Then P is above Q at q but P is below Q at r. Thus, in going from q to r we must have left and re-entered P or Q; hence, (P, p) or (Q, q) cannot be active at time i (contradiction). The arguments for the other cases are similar, and are left to the interested reader. □

Thus, the comparison procedure used in Step 5 is a consistent relation; hence, the **max** label associated with each record in the B list for the root in T stores the name of the polygon visible along the edge e_i, where i is the time value associated with that record. Thus, we have the following theorem:

THEOREM 3.3. *Given a collection S of nonintersecting polygons in $\Re^3$, the hidden-surface elimination problem for S can be solved in $O(\log n)$ time using $O((n + I) \log n)$ processors in the CREW PRAM model, where n is the total number of edges and I is the number of edge intersections in the projection plane.*

3.2. *Arrangement Queries.* The arrangement sweeping technique can also be used to build various geometric data structures in parallel. The main idea is to build the arrangement, an operation sequence for that arrangement, use the array-of-trees data structure to evaluate the sequence, and then perform queries for this sequence to solve the problem.

We illustrate this with an example. Suppose one is given a collection of line segments in the plane and it is wished to construct a data structure that allows the segments that are intersected by a query line l to be quickly counted or reported, or to return a line that intersects the most number of line segments. Using a well-known point-line duality [21], [40], the set of all lines intersecting a line segment dualizes to the set of all points lying in a certain double-wedge (a region defined by all points between two intersecting lines). In particular, using a duality that preserves "above" relations (so a point p above a line l dualizes to a line $\mathcal{D}_p$ above the point $\mathcal{D}_l$), all the lines intersecting a line segment s dualize to all the points contained between the duals of s's endpoints. Thus, answering segment-intersection queries is equivalent to the problem where one is given a collection of double-wedges and asked to build a data structure that counts or reports all double-wedges containing a query point l.

We can solve this problem as follows. First, we can construct the arrangement formed by all the double-wedges using the line-arrangement algorithm of Goodrich [24], compute a spanning tree of this arrangement (as above), and build an Euler tour of this tree. This can all be done in $O(\log n)$ time using $O(n^2/\log n)$ processors. We can then construct a skeleton binary tree T, whose leaves correspond to double-wedges, and an operation sequence σ for the Euler tour, where the operations are **enable**(s), which corresponds to entering the double-wedge for s, and **disable**(s), which corresponds to leaving the double-wedge for s. Building the compressed array-of-trees data structure for this T and σ allows us to label each face f with the number of double-wedges containing it, or, alternately, with a pointer to the root of the array-of-trees corresponding to the position in the tour where we entered f. If we are only interested in counting queries, then we can evaluate σ by a simple parallel prefix computation, as described in Section 2, which

requires $O(\log n)$ time using $O(n^2/\log n)$ processors, and allows counting queries to be answered in $O(1)$ time given the position of f's visitation in the tour. This immediately implies that we can find a maximum stabbing line in $O(\log n)$ time using $O(n^2/\log n)$ processors. If we wish to answer reporting queries, then we can construct an array-of-trees data structure for σ, which requires $O(\log n)$ time using $O(n^2)$ processors. A reporting query can then be answered in $O(\log n)$ time, where we first determine the number, k, of answers, with a single processor, and then allocate $\lceil k/\log n \rceil$ processors to the task of enumeration.

In the next subsection we give an application of arrangement sweeping via off-line expression evaluation.

3.3. *CSG Boundary Evaluation.* Suppose a collection of "primitive" polygonal shapes and an expression tree T are given such that each leaf of T has a primitive object associated with it and each internal node of T is labeled with a boolean operation, such as union, intersection, exclusive-union, or subtraction (a CSG representation [45]). Note that while T is defined in terms of primitive shapes, the definition is such that T defines the boundary of the overall object when considered as a boolean expression. The problem we address in this subsection is that of constructing a boundary representation for the object defined by the root of T. (See Figure 6.)

We first address the two-dimensional version of the problem, where the primitive objects are simple polygons. Using the approach of Goodrich [23], we can solve this problem in parallel as follows. We construct the arrangement of the polygons that define the primitives (including the vertical shadows of each vertex), find a spanning forest for this arrangement, and build an Euler tour of each tree in this forest (similar to the first three steps in our hidden-surface elimination algorithm). This takes $O(\log n)$ time using $O(n \log n + I)$ processors, where I is the number of pairwise edge intersections.

We then build an instance of the off-line expression-evaluation problem, which will allow us to label each vertex, edge, and face of the arrangement as being either "inside" or "outside" of the region defined by T. In particular, we construct an operation sequence σ for each Euler tour just constructed. We begin each such σ with a vertex in the tour with a vertical shadow extending to $-\infty$. Each operation in σ is an assignment of the form $x_i = b$, where x_i is some leaf in T and b is a boolean constant with the

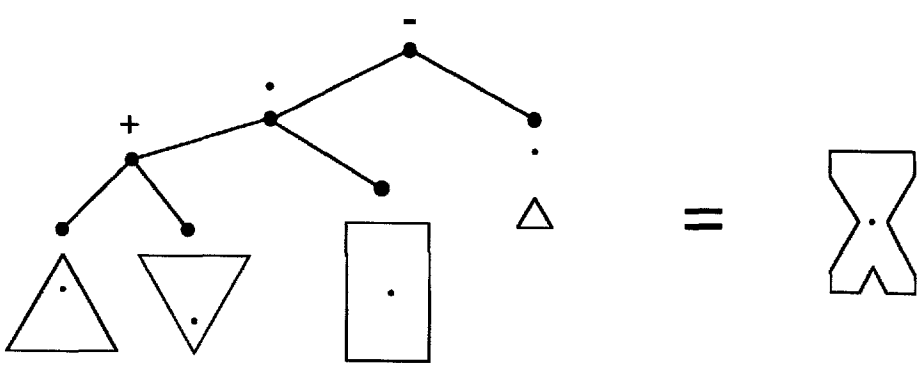

Fig. 6. The CSG boundary-evaluation problem.

following interpretation: a "0" represents the proposition "outside of primitive i" and a "1" represents "inside primitive i." We assume that each leaf of T begins with a value of "0," since we begin σ with a vertex with a vertical shadow extending to $-\infty$. Note that while T is defined in terms of regions, its definition is such that, given any point p in the plane, we can evaluate T as a boolean expression, where each leaf value contains the appropriate value relative to p, and this evaluation will tell us whether or not p is inside ("1") or outside ("0") the object described by T. Thus, a solution to the off-line expression-evaluation problem for σ will allow us to label each vertex, edge, and face of the arrangement as either being inside or outside the object defined by T. Therefore, by applying the off-line expression-evaluation theorem of the previous section, we can evaluate T for each cell of the polygon arrangement in $O(\log n)$ time using $O(n + I)$ processors. Constructing a boundary representation of the defined region is then a simple matter of removing edges and vertices that are not on the boundary (which can now be determined by a simple local test). This gives us the following theorem:

THEOREM 3.4. *Two-dimensional CSG evaluation can be performed in $O(\log n)$ time using $O(n \log n + I)$ processors, where n is the total number of primitive edges and I is the number of edge intersections in the polygon arrangement (which is $O(n^2)$ in the worst case).*

4. Sweeping Through a Set of Rectangles. In this section we address the situation when it is desired to perform a number of coordinated sweeps in parallel, which together define a sweep through a set of rectangles. We motivate our approach with two important applications: computing the contour of a collection of rectangles in the plane [35], [58], [57], [12], and performing hidden-surface elimination on a collection of rectangles in $\Re^3$ [28], [25], [9], [37], [43].

In keeping with our notion of parallel persistence, one of the paradigms we use in our algorithms is that of an *event list*. Recall that an event list $\mathbf{E}$ is an array representing the history of some variable e. Each record in an event list $\mathbf{E}$ corresponds to a change in the value of e, and stores both the new value of e and the "time" at which the change occurred (where, in the context of this section, we let the time be a real number that corresponds to the position in the operation sequence of the operation that caused the change to e). Both of our methods also depend on the use of a number of additional parallel techniques, which we discuss in the following two subsections.

4.1. *Some Algorithmic Tools.* The first algorithmic tool we review is the *fractional cascading* of Chazelle and Guibas [15], and how it can be implemented in parallel [4]. The general framework is one in which we are given a directed bounded-degree graph $G = (V, E)$, where each node v in G contains a sorted list $C(v)$. The problem is to construct a data structure so that given a walk $(v_1, v_2, \ldots, v_m)$ and an arbitrary element x, one processor can locate x in all of the $C(v_i)$'s quickly. An efficient solution involves constructing an "auxiliary" list $A(v)$ for each $C(v)$ list, such that

(i) $C(v) \subseteq A(v)$,
(ii) $\sum_{v \in V} |A(v)|$ is $O\left(\sum_{v \in V} |C(v)|\right)$, and

(iii) given the position of x in $A(v)$ one can locate x in $C(v)$ and in $A(w)$, for each neighbor w of v, in $O(1)$ time.

Atallah *et al.* [4] derive the following lemma for this problem:

LEMMA 4.1. *Given a directed bounded-degree graph $G = (V, E)$, a fractional cascading data structure for G, including all the auxiliary lists $A(v)$ for each $v \in V$, can be built in $O(\log N)$ time using $O(N)$ space with $O(N/\log N)$ processors on a CREW PRAM, where N is $|V| + |E| + \sum_{v \in V} |C(v)|$.*

As observed by previous researchers [15], [4], this technique is quite useful for reducing the time complexity for performing a sequence of similar searches, provided the sequence forms a single path in the graph G. In some instances, however, it is convenient to allow the collection of similar searches to grow as a tree, rather than a single path. Specifically, one can imagine such a collection of searches being implemented by a group of processors, where, in any step t, one may wish to allow for various processors to each "spawn" another processor, and have the new processor begin executing in step $t + 1$ [44]. The next lemma shows that the only real cost of allowing for this extension is in the parallel time. In terms of work, it is essentially free, in that it only requires an increase of at most a constant factor in the work needed to simulate it for our problems.

LEMMA 4.2 [23]. *Given an algorithm A designed for a PRAM model that uses a spawning processor allocation scheme and requires $O(w)$ work in t time, A can be simulated on an analogous PRAM with global allocation with $O(w)$ work and time $O(t \log p)$, where p is the final number of processors.*

For an example of an application that uses both of the above lemmas, consider the following problem. Suppose an n-node simple polygon P is given, and it is wished to build a data structure for P that allows the determination of all the intersections of a query line L with P efficiently in parallel. One possible solution is to build a parallel version of the data structure of Chazelle and Guibas [15], for performing such queries sequentially in $O(k \log n)$ time. The method is as follows:

1. Build a complete binary tree B on the edges of P, in the order that they appear in a clockwise traversal. So each node v in B corresponds to a chain, P_v, of P, consisting of all edges stored in descendants of v.
2. For each v in B construct the convex hull, H_v, of P_v. This can be done in $O(\log n)$ time using $O(n/\log n)$ processors [55], [56].
3. Construct a fractional cascading data structure on the upper and lower chains of the H_v's (which form lists at each node in B) using edge slopes as keys for the fractional cascading. The underlying graph is the tree B. This can also be done in $O(\log n)$ time using $O(n/\log n)$ processors, by Lemma 4.1, and completes the construction.

Thus, the entire construction can be implemented in $O(\log n)$ time using $O(n/\log n)$ processors.[5] Given the position of the slope of L in the upper and lower hulls of B_v (based on slope), it can be determined if L intersects B_v (and hence P_v) in constant time [15] (by a few simple calculations). Therefore, given a line L, by a top-down search (using the fractional cascading auxiliary lists) all the places L intersects P can be determined in $O(\log n)$ time using $O(1+k)$ processors, assuming a local processor allocation scheme, where k is the number of answers. By applying Lemma 4.2, this immediately implies that such a query can be performed in $O(\log^2 n)$ time using $O(1 + k/\log n)$ processors in the CREW PRAM model.

Returning to the problem of sweeping through a set of rectangles, in the following subsection we review an important data structure, which our algorithms use as the skeleton structure for an array-of-trees construction.

4.2. *The Segment Tree.* Let $S = \{s_1, s_2, \ldots, s_n\}$ be a set of vertical line segments in the plane, and let $Y = (y_1, y_2, \ldots, y_{2n})$ be the (nondecreasing) sorted list of the y-coordinates of the endpoints of the segments in S. To simplify the exposition we assume that no two endpoints in S have the same x-coordinate, i.e., $x_i < x_{i+1}$. (It is straightforward to modify our algorithm for the general case.) Let T be the complete binary tree whose $2n+1$ leaves, in left to right order, correspond to the intervals $(-\infty, y_1]$, $[y_1, y_2]$, $[y_2, y_3]$, $\ldots$, $[y_{m-1}, y_m]$, $[y_m, +\infty)$, respectively. Associated with each internal node $v \in T$ is a closed interval $I_v = [y_i, y_j]$ which is the union of the intervals associated with the descendants of v. (Of course, the leftmost and rightmost nodes at each level will have intervals $I_v = (-\infty, y_i]$ and $I_v = [y_j, +\infty)$.) Let Π_v denote the horizontal slab $I_v \times (-\infty, +\infty)$. We say a segment s_i *covers* a node $v \in T$ if it spans Π_v but not $\Pi_{parent(v)}$. Clearly, no segment covers more than two nodes of any level of T; hence, every segment covers at most $O(\log m)$ nodes of T. For each node $v \in T$ we define two sets, $Cover(v)$, $End(v)$:

- $Cover(v)$ is the set of segments in S that cover v.
- $End(v)$ is the set of all segments that do not span Π_v but have an endpoint in Π_v.

The tree T together with the above lists constructed for each node in T constitutes the *segment tree* for S [8]. (See Figure 7.)

4.3. *Computing the Contour.* Given a set of isothetic rectangles, we consider the problem of computing and reporting the contour of the union of the rectangles. (See Figure 8.) Sequentially, this problem was first studied by Lipski and Preparata [35], and time optimality was achieved by Wood [58]. Time and space optimality was subsequently achieved by Widmayer and Wood [57]. In the parallel domain Chandran and Mount [12] produced a CREW PRAM algorithm that runs with $O(n)$ processors in $O(k_{\max})$ time, where $k_{\max}$ is the largest number of output subsegments associated with any one line segment (which can be $\theta(n)$).

We achieve $O(\log n)$ time with $O(n + k/\log n)$ processors (which is optimal), where k is the size of the output. Our method reports the edges of the contour. If the contour

[5] This method actually improves the sequential time complexity of constructing the Chazelle–Guibas data structure [15], as their method runs in $O(n \log n)$ time.

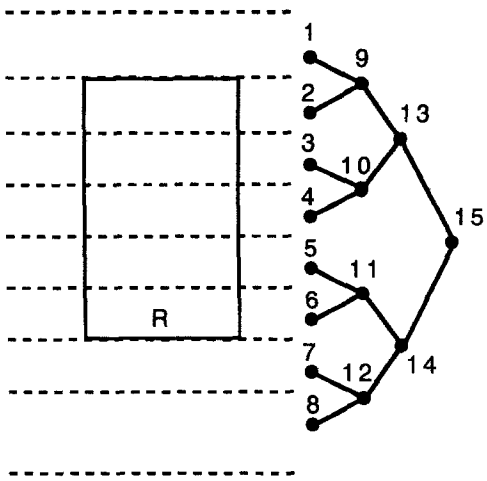

Fig. 7. A segment tree. R is in $Cover(2)$, $Cover(10)$, $Cover(11)$, $End(1)$, $End(7)$, $End(9)$, $End(12)$, $End(13)$, $End(14)$, and $End(15)$.

cycles are desired, then the work bound of our method becomes $O(n \log n + b(k))$, where $b(m)$ is the work for performing stable bucket sorting of m elements.[6] Our procedure, which we describe below, follows the general framework of Wood [58]. We determine all the vertical line segments of the contour, and then repeat our procedure, exchanging the roles of the x- and y-axes, to obtain the horizontal segments.

Step 1. In this step we build the segment tree on the set of vertical segments obtained from the rectangle vertical boundaries, complete with all the $Cover(v)$ and $End(v)$ lists constructed for each node, such that each leaf contains one such segment, and the segments are assigned to the leaves in order sorted by the x-coordinates (so the leftmost

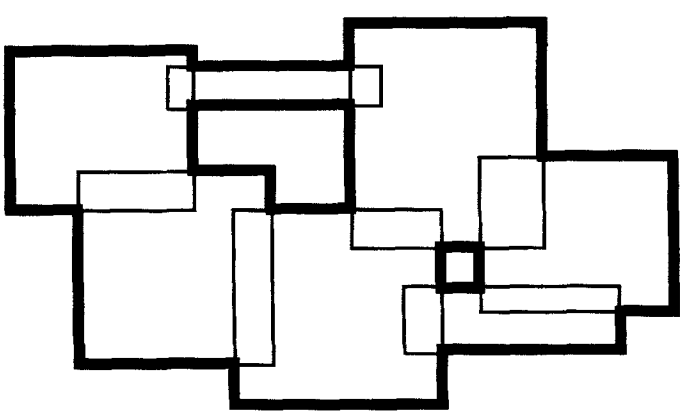

Fig. 8. The contour of a set of isothetic rectangles.

[6] Matias and Vishkin [36] give a randomized method running in $O(\log m \log\log m)$ expected time with $O(m \log\log m)$ work on an arbitrary-CRCW PRAM.

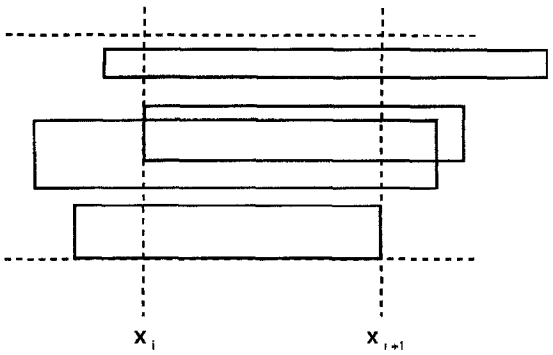

Fig. 9. The values of h_i, *top*, and *bottom* on an interval (x_i, x_{i+1}).

leaf has the lowest x-value and the rightmost has the highest.) We view each $Cover(v)$ as an event list where x-coordinates act as the "time" field. Using the method of Atallah, *et al.* [4], we can implement this step in $O(\log n)$ time with $O(n)$ processors.

Step 2. For each node v, we construct an event list, $\mathbf{C}(v)$, such that, for each entry $\alpha_i = (x_i, c_i)$ in $\mathbf{C}(v)$, c_i is the number of segments that *cover* v during the interval $[x_i, x_{i+1})$. The x-values (x_i, x_{i+1}) in this case are the x-coordinates of the segments in $Cover(v)$. Given the $Cover(v)$ lists, this construction is essentially just a collection of parallel prefix computations. We then construct the fractional cascading auxiliary lists for these $\mathbf{C}(v)$ lists, using the method of Atallah *et al.* [4], which runs in $O(\log n)$ time with $O(n)$ processors (since the total size is $O(n \log n)$).

Step 3. For each node v, we construct an event list, $\mathbf{H}(v)$, such that, for each entry $\beta_i = (x_i, h_i)$, h_i is the number of (maximal) rectangular regions in Π_v that extend horizontally from x_i to x_{i+1} and do not intersect the interior of any rectangle in $Cover(v) \cup End(v)$. Intuitively, h_i is the number of "holes" from x_i to x_{i+1} when we restrict our attention to the rectangles in $Cover(v) \cup End(v)$. (See Figure 9.) In the terminology of Section 2, the x-coordinates that correspond to the x-coordinates of the vertical boundaries of the rectangles in $Cover(v) \cup End(v)$ determine the "times" in $\mathbf{H}(v)$. We also define a flag *top* (resp. *bottom*), for each entry in an $\mathbf{H}$ list, such that $\mathbf{H}[i].top$ (resp. $\mathbf{H}[i].bottom$) is **true** if and only if the topmost (resp. bottommost) maximal rectangle from x_i to x_{i+1} does not extend into the slab adjacent to Π_v.

Implementation. We can construct the $\mathbf{H}(v)$ lists at the leaves immediately from the entries in $\mathbf{C}(v)$. Specifically, if (x_i, c_i) is the ith value in $\mathbf{C}(v)$, then (x_i, h_i) is the ith value in $\mathbf{H}(v)$, where $h_i = 1$ if $c_i = 0$, and $h_i = 0$ otherwise. We construct the other $\mathbf{H}(v)$ lists by a bottom-up procedure. Assume, for some node v, we have already constructed the respective $\mathbf{H}$ lists for v's children, u and w (with Π_u being above Π_w), and we have a list of sorted x-coordinates, each of which is determined by the vertical boundary of a rectangle in $Cover(v) \cup End(v)$. Also assume (by the fractional cascading auxiliary pointers) that, for each element s in this list, we have a pointer to the elements, α, β, and γ, in $\mathbf{C}(v)$, $\mathbf{H}(u)$, and $\mathbf{H}(w)$, respectively, that have the largest x-coordinate less than or equal to s's x-coordinate. We define the (x_i, h_i) pair in $H(v)$ for s so that x_i is the x-coordinate of s, and h_i is defined as follows: Let $\alpha = (x, c)$. If $c \geq 1$, then

$h_i = 0$. If $c = 0$, then we set h_i equal to the sum of h_u and h_w, where $\beta = (x_u, h_u)$ and $\gamma = (x_w, h_w)$. This is not quite right, of course, since we have not taken the influence of the *top* and *bottom* flags into account. We can easily update h_i to reflect their influence, however, by decrementing h_i by one if neither $\beta.bottom$ nor $\gamma.top$ is **true**. This can easily be implemented (for all v in parallel) in $O(\log n)$ time with $O(n)$ processors.

Step 4. In this step we construct the pruned array-of-trees. Given an entry $\alpha_i = (x_i, h_i)$ in $\mathbf{H}(u)$, the pruning function $\pi(\alpha_i, u)$, which determines whether a pointer to α_i occurs in $B(v)$, where v is the parent of u, is equal to one if and only if both of the following hold:

(i) The record in $\mathbf{C}(v)$ with the largest x-value less than or equal to x_i has a c-value equal to zero.
(ii) $h_i > 0$.

We calculate this, and, in so doing, construct the pruned array-of-trees, level-by-level, starting at the leaves, as in Section 2.2. This step runs in $O(\log n)$ time using $O(n)$ processors.

Comment. The intuition behind this definition of π is that it "jumps over" linear chains of pointers within the tree, and so makes possible an optimal search of the tree in Step 5 below. These linear chains are caused by the pruning process, which may prune one child of a node and not the other.

Step 5. In this step we determine, for each vertical line segment L in S, all the subsegments of L that are part of the contour. Starting at the root, we search down the tree for the subsegments of L, checking the $\mathbf{H}$ list at each node, searching with the x-coordinate of L for uncovered intervals through which L might be seen. If the active h-value in the $\mathbf{H}$ list is zero, then we stop searching down this branch as no output can result. Once each node, v, covered by L is located, for each such v, we determine the total number k_v of uncovered subsegments of L in the subtree rooted at v, by examining the $\mathbf{H}$ lists at v's children. Then we request $\lceil k_v / \log n \rceil$ new processors, which start at the children of v, and search the compressed trees rooted there (in the pruned array-of-trees) for all pieces of the output that are on L in Π_v. This completes the algorithm, and gives us the following theorem.

THEOREM 4.3. *Given a collection of n isothetic rectangles in the plane, the edges of their contour can be determined in $O(\log n)$ time using $O(n + k/\log n)$ processors in the CREW PRAM model, which is optimal.*

In the next subsection we give another application of our coordinated parallel sweeping approach.

4.4. *Hidden-Surface Removal for Rectangles.* Given a set of n opaque iso-oriented rectangles parallel to the xy-plane, we wish to determine all of the portions of each rectangle that are visible from viewing at $(0, 0, +\infty)$. Sequentially, this problem was first studied by Güting and Ottmann [28], with more efficient algorithms being recently reported by Goodrich *et al.* [25], Bern [9], Mehlhorn *et al.* [37], and Preparata *et al.* [43]. The best sequential bound (optimizing for the term involving only n) is $O((n+k)\log n)$,

where k is the size of the output [25], [9], [37]. We show how to solve this problem in $O(\log^2 n)$ time using $O((n+k)\log n)$ work. Our algorithm description assumes a local-allocation scheme and runs in $O(\log n)$ time using $O(n+k)$ processors; we apply Lemma 4.2 to derive the claimed bounds. Our method is based on the approach of Bern [9] but avoids an inherently sequential step in his algorithm (which uses the union-find structure of Hopcroft and Ullman [29]) by use of the array-of-trees.

As in the previous algorithm, we use a segment tree to store the vertical edges of the rectangles. For each node v in this tree, we define the restricted subscene for v to consist of all vertical edges e such that e belongs to a rectangle in $Cover(v) \cup End(v)$, but the intersection of the projection of e, onto the xy-plane, with Π_v is more than a single point. Our method uses three event lists built for each node v: **Top**(v), **High**(v), and **Low**(v), where there is an entry in **Top**(v) for each vertical segment in $Cover(v)$, and an entry in **High**(v) and **Low**(v) for each vertical edge of the restricted subscene for v. Their meanings are as follows: If (x_i, top_i) is an element in **Top**(v), then top_i is the maximum z-coordinate of the rectangles in $Cover(v)$ that intersect the plane $x = x_i$. If $(x_i, high_i)$ is an element in **High**(v), then $high_i$ is the maximum z-coordinate of the rectangles in the restricted subscene for v that intersect the plane $x = x_i$. If (x_i, low_i) is an element in **Low**(v), then low_i is the minimum z-coordinate of the rectangles in the restricted subscene for v that intersect the plane $x = x_i$ and are visible from $(0, 0, +\infty)$. Since the "times" in each of these event lists are determined by x-coordinates, given some x-coordinate, x, we define the entry of one of these lists that has a largest x-coordinate less than or equal to x to be the entry *active* at x.

Our method for finding the vertical edges in the visibility map is as follows:

Step 1. We construct the segment tree, together with all the $Cover(v)$ and $End(v)$ lists, sorted by x-coordinates of the vertical segments.

Step 2. We construct **Top**(v) for all nodes v, in $O(\log n)$ time with $O(n)$ processors by a parallelization of the sequential method of Goodrich *et al.* [25] via the cascading divide-and-conquer paradigm of Atallah *et al.* [4]. We give the details in Section 5.

Step 3. From the $Cover(v)$ and $End(v)$ lists, we construct the lists of x-coordinates for the **High** and **Low** arrays, and apply fractional cascading to these arrays and the **Top** arrays constructed in the previous step. Also, given the **Top**(v) values previously constructed, we can construct the event lists for **Low** and **High** by a simple bottom-up procedure. Constructing these lists for the leaves is straightforward, so suppose we have already computed the **High** and **Low** lists for the children u and w of v. Consider an x-coordinate, x, for which we wish to compute its corresponding *high* and *low* values. Let $high_u$ and $high_w$ be the elements of **High**(u) and **High**(w) that are active at "time" x. Similarly, define low_u and low_w. Also, let top be the element of **Top**(v) active at x. Then $high_v = \max\{high_u, high_w, top\}$ and $low_v = \max\{\min\{low_u, low_w\}, top\}$ [9]. Thus, we can compute *high* and *low* in $O(1)$ time given these other values (which we can maintain during our bottom-up procedure). Therefore, this construction takes $O(\log n)$ time using $O(n)$ processors.

Step 4. In this step we determine all the visible vertical edges. We assign a processor P to every vertical edge e of an input rectangle. P visits every node in the segment tree which e covers, searching down from the root of the segment tree, and, for each such

node v, P determines if e is completely visible in Π_v, completely invisible in Π_v, or partially visible in Π_v. To help P make these determinations, as P traverses the tree it maintains a value *maxtop*, which is the largest *top* value active at $x(e)$ from all the **Top**(v') lists such that v' is an ancestor of v, where v is the current node in the traversal. Let *low* and *high* be the values in **Low**(v) and **High**(v), respectively, active at $x(e)$. The test for each of these possibilities is as follows:

If $z(e) <$ *maxtop* or $z(e) <$ *low*, then e is completely invisible in Π_v.
If $z(e) >$ *maxtop* and $z(e) >$ *high*, then e is completely visible in Π_v.
Otherwise, if $z(e) >$ *maxtop* and *low* $\leq z(e) \leq$ *high*, then e is partially visible in Π_v.

Comment. Having marked e relative to each slab Π_v that e covers as being completely visible, completely invisible, or partially visible, we must now determine the segments of e that are visible in each slab for which e is marked partially visible (we are done with e in the other slabs). Our method, which we describe in the next step, involves having the processor assigned to enumerating the visible segments of e in Π_v perform a search in the subtree T_v, rooted at v, spawning enough new processors to enumerate all these segments. The difficulty in this step comes from the need to output only $O(1)$ pieces of each visible segment even though each such segment can cover up to $O(\log n)$ nodes (in the segment tree sense of "cover") in T_v. Not fulfilling this requirement would mean that we would use more than $O(n + k)$ processors overall.

Step 5. We assign a processor P for e to each slab Π_v such that e is partially visible, and use this processor to enumerate all the visible segments of e in Π_v. In P's search down T_v we assume, inductively, that e is partially visible on Π_v and that P has already determined two segments, e_t and e_b, on e that are visible outside Π_v, with e_t being adjacent to the top boundary of Π_v and e_b being adjacent to the bottom boundary of Π_v (in the y-direction). (Initially, e_t and e_b are **nil**.) P uses the fractional cascading pointers from v to test in $O(1)$ time if e is completely visible, completely invisible, or partially visible in Π_u and Π_w, where u and w are the children of v, with Π_u being above Π_w (in the y-direction). There are several cases:

1. e is completely visible in Π_u and completely invisible in Π_w (e cannot be completely invisible or completely visible in both, by induction). Then P "grows" e_t to include $e \cap \Pi_u$, i.e., assigns $e_t := e_t \cup (e \cap \Pi_u)$, and outputs e_t and e_b (if e_b is not **nil**). P is done.
2. e is completely visible on Π_u and partially visible in Π_w. Then P grows e_t to include $e \cap \Pi_u$ (i.e., assigns $e_t := e_t \cup (e \cap \Pi_w)$) and continues its search in Π_w.
3. e is completely invisible in Π_u and completely visible in Π_w. Then P "grows" e_b to include $e \cap \Pi_w$, i.e., assigns $e_b := e_b \cup (e \cap \Pi_w)$, and outputs e_b and e_t (if e_t is not **nil**). P is done.
4. e is completely invisible in Π_u and partially visible in Π_w. Then P outputs e_t (if it is not **nil**) and continues its search in Π_w (with $e_t =$ **nil**).
5. e is partially visible on Π_u and completely visible in Π_w. Then P grows e_b to include $e \cap \Pi_w$ (i.e., assigns $e_b := e_b \cup (e \cap \Pi_w)$) and continues its search in Π_u.
6. e is partially visible in Π_u and completely invisible in Π_w. Then P outputs e_b (if it is not **nil**) and continues its search in Π_w (with $e_b =$ **nil**).
7. e is partially visible on Π_u and Π_v. Then P spawns a new processor P' to search in

Π_w with $e'_t = \textbf{nil}$ and $e'_b = e_b$ (P' runs the same program as P but uses e'_t and e'_b instead of e_t and e_b). P then sets $e_b := \textbf{nil}$ and continues its search in Π_u.

This completes the algorithm.

The above procedure determines all the vertical segments in the visibility map. By running the algorithm once more, with the roles of the x-axis and y-axis reversed, we can find the visible horizontal edges. If we also wish to output the visible surfaces, then for every visible line segment we need to determine the visible rectangles that are immediately to the left and right of the segment. This can be easily be accomplished during the previous step, however, by noting the active values of *maxtop*, *low*, and *high* during the downward searches in the segment tree, labeling each visible segment s with the polygons visible on each side of s as soon as we have determined that $\dot{s}$ is visible. We leave the details to the interested reader. As we show in the following theorem, this algorithm runs in $O(\log n)$ time using $O(n + k)$ processors in the CREW PRAM with spawning processor allocation, hence, in $O(\log^2 n)$ time using $O((n + k)/\log n)$ processors in the CREW PRAM (with global processor allocation).

THEOREM 4.4. *Given n iso-oriented rectangles in $\Re^3$, the hidden-surface elimination problem for these rectangles can be solved in $O(\log^2 n)$ time using $O((n + k)/\log n)$ processors in the CREW PRAM model, where k is the size of the output.*

PROOF. By Lemma 4.2, it is sufficient to show that the algorithm runs in $O(\log n)$ time using $O(n + k)$ processors in the CREW PRAM with spawning processor allocation. Each step in our method can be implemented in $O(\log n)$ time, since the height of the segment tree is $O(\log n)$. Thus, we must show that the total number of processors is $O(n + k)$. Of course, Step 5 is the only step for which we spawn new processors, and every other step can be implemented with $O(n)$ processors. We concentrate, then, on Step 5. To show that the number of processors spawned in this step is $O(n+k)$, it suffices to show that $O(1)$ processors are spawned for each vertical segment in the visibility map. So, let s be a vertical segment in the visibility map and let e be the input edge containing s. Since the endpoints of s are determined by two y-coordinates from input rectangles, the endpoints of s fall on slab boundaries in the segment tree. Let v be the least-common ancestor of the nodes in the segment tree that s covers. Since s is visible, v must have been visited by a (single) processor P in Step 5. The only case for which a processor P spawns a new processor P' is for a node such that e is partially visible in the two slabs associated with that node's children. However, the only place where s forces a processor possibly to spawn a new processor to output eventually a piece of s is at v. In all other searches down the subtree rooted at v some piece of s will be stored in e_b (for the subtree of v's left child) or e_t (for the subtree of v's right child). Thus, we spawn at most two processors to output s. This completes the proof. □

5. Computing the Top Event Lists. In this section we show how to compute the visibility of the rectangles that cover v for each v in the segment tree of Section 4.4. Note that this was the operation that we had to perform in Step 2 of the algorithm in

that section. It is also the crucial difference between our approach and the sequential approach of Bern [9]. Our method is a nontrivial parallel implementation of the sequential algorithm of Goodrich *et al.* [25] for the same problem. We include our description here for completeness. Our method runs in $O(\log n)$ time using $O(n)$ processors.

5.1. *A Simple, but Inefficient, Method.* We begin by describing a simple method that takes $O(\log n)$ time using $O(n \log n)$ processors. The idea is to assign $|Cover(v)|$ processors to each node v in the segment tree T. These processors then perform a parallel mergesort procedure to sort the vertical boundaries of the rectangles in $Cover(v)$ by x-coordinates. In addition, with each merge of two lists A and B of rectangles we also compute the topmost rectangle, $top(x)$, between each x-coordinate in $A \cup B$. This extra computation can easily be implemented in $O(1)$ time by taking the **max** of the top values of the two overlapping intervals in A and B that determine each interval in $A \cup B$. Using Cole's parallel mergesort procedure [17] to implement each such procedure requires $O(\log n)$ time using $O(n \log n)$ processors (for all v in T).

5.2. *A Modest Improvement.* We can improve this approach by examining how various $Cover(v)$ lists in T relate to one another. In particular, we label each rectangle R with the depth d of the node in T that is the least-common ancestor of all the nodes that R covers. For each node v let $N_d(v)$ (resp. $S_d(v)$) denote the vertical boundaries of the rectangles in $End(v)$ that intersect the northern (resp. southern) boundary of Π_v and have depth label d, sorted by their x-coordinates. Atallah *et al.* observe the following [4]:

OBSERVATION 5.1. *Let v be a node in T at depth d, and let z denote v's sibling in T. Then $Cover(v) = S_1(z) \cup S_2(z) \cup \cdots \cup S_{d-1}(z)$ if v is a right child, and $Cover(v) = N_1(z) \cup N_2(z) \cup \cdots \cup N_{d-1}(z)$ if v is a left child.*

Also note that $N_d(v) = N_d(u) \cup N_d(w)$ and $S_d(v) = S_d(u) \cup S_d(w)$, where u and w are the children of v (provided v is at depth at least d). Thus, we can create $\lceil \log n \rceil$ copies, $T_1, T_2, \ldots, T_{\lceil \log n \rceil}$, of the tree T, and construct $N_d(v)$ and $S_d(v)$ for each v in T_d using the cascading divide-and-conquer technique of Atallah *et al.* [4]. This takes $O(\log n)$ time using $O(n)$ processors.[7] In addition, we can be computing the topmost rectangle on each x-interval in each $N_d(v)$ list (resp. $S_d(v)$ list) in these same bounds, as in the previous method.

Now to compute the visibility of the x-intervals determined by the rectangles in any $Cover(v)$, where, say, v is a right child at depth d, we simply need to merge $S_1(z), S_2(z), \ldots, S_{d-1}(v)$, applying the approach of the previous method during the merges. We can perform all of these merges in a bottom-up fashion in $O(\log d \log \log n)$ time using $O(|Cover(v)|/\log \log n)$ processors [10], [31], [54]. By Brent's theorem [11], this immediately implies that the visibility of the x-intervals determined by the

[7] This implies that some of the merges in a T_d may be vacuous, but this is easy to implement: simply have a processor assigned to the left (resp. right) child write a 1 to a *left* (resp. *right*) field at the parent to see if there needs to be a merge at this node. Thus, even though there are $O(n \log n)$ nodes overall, $O(n)$ processors suffice for all the merges.

rectangles in any $Cover(v)$ can be constructed in $O(\log n)$ time using $O(|Cover(v)| \log\log n/\log n)$ processors (i.e., $O(|Cover(v)| \log\log n)$ work).

5.3. *A Coordinated Attack.* Our approach to achieving $O(\log n)$ time using only $O(n)$ processors for the entire computation is to coordinate the construction of all the visibility lists using a "stratification" paradigm [13]. Let Vis_v denote the visibility list for the x-intervals of the rectangles in $Cover(v)$ (i.e., the upper envelope of these rectangles listed by increasing x-coordinates). Also, let $\mathcal{N}(\Pi_v)$ (resp. $\mathcal{S}(\Pi_v)$) denote the plane perpendicular to the xy-plane and containing the northern (resp. southern) horizontal boundary of Π_v. Our method for computing Vis_v for all v in T is as follows:

0. We begin by computing the trees $T_1, T_2, \ldots, T_{\lceil \log n \rceil}$ and the visibility of the x-intervals determined by the $L_d(v)$ and $R_d(v)$ in each T_d, as in the previous method. This takes $O(\log n)$ time using $O(n)$ processors.
1. We mark each node that is at a depth of T that is a multiple of $\lceil \log\log n \rceil$ as a *supernode*. For each supernode v, at depth d, we let $T(v)$ denote the subtree of T rooted at v and having the supernodes at depth $d + \lceil \log\log n \rceil$ as its leaves.
2. For each supernode v, let z be the nearest supernode ancestor of v (so v is a leaf in $T(z)$). We construct $Vis\_Northern\_Long(v)$ and $Vis\_Southern\_Long(v)$, where $Vis\_Northern\_Long(v)$ is a representation of the upper envelope in the $\mathcal{N}(\Pi_z)$ plane of the segments formed by intersecting $\mathcal{N}(\Pi_z)$ with the rectangles in $End(v)$, ignoring the rectangles in $End(v)$ that do not intersect $\mathcal{N}(\Pi_z)$. Intuitively, $Vis\_Northern\_Long(v)$ is the upper envelope of the "long" rectangles in $End(v)$. $Vis\_Southern\_Long(v)$ is defined similarly. These can be computed in $O(\log n)$ time using $O(n)$ processors by the method described in the previous subsection.
3. For each node v that is not a supernode we let z be the nearest supernode ancestor of v (so v is an internal node in $T(z)$). We construct $Vis\_Northern\_Long(v)$ and $Vis\_Southern\_Long(v)$, as defined in the previous step. We perform this computation for each z by applying the mergesort-like procedure of Section 5.1 to the solutions already at the leaves of $T(z)$ (combining solutions up the tree using a bottom-up merge). Since the height of each $T(z)$ is $O(\log\log n)$, this step takes $O((\log\log n)^2)$ time using $O(n_z/\log\log n)$ processors [10], [31], [54], where n_z is the number of rectangles which are stored in the leaves of T_z (in $Vis\_Northern\_Long(v)$ and $Vis\_Southern\_Long(v)$ lists) at the beginning of this step. Since a rectangle R can be contained in at most $\log n/\lceil \log\log n \rceil$ of these (leaf) supernode lists, $\sum_z n_z = n \log n/\lceil \log\log n \rceil$; hence, the total work needed for this step is $O(n \log n)$.
4. For each node v that is not a supernode (hence, has a nearest supernode ancestor z), we construct $Vis\_Cover\_Short(v)$, where $Vis\_Cover\_Short(v)$ is a representation of the upper envelope (in the $\mathcal{N}(\Pi_v)$ plane) of the segments formed by intersecting $\mathcal{N}(\Pi_v)$ with the rectangles in $Cover(v)$ that have both of their horizontal boundaries properly contained in Π_z. This can be done in $O(\log n)$ time using $O(m_v \log\log n)$ work by the method given above (in Section 5.2), where m_v is the number of rectangles involved for v. In particular, $Vis\_Cover\_Short(v)$ can be constructed by merging $S_{d-1}(w), S_{d-2}(w), \ldots, S_{d_z}(w)$, if, say, v is a right child (otherwise the N lists would be used for w), where w is v's sibling, d is the depth of v, and d_z is the depth of z.

Since any rectangle can cover at most $O(\log \log n)$ nodes in this way, this step can be implemented in $O(\log n)$ time using $O(n(\log \log n)^2)$ work.

5. For each node v we compute Vis_v, the upper envelope (in the $\mathcal{N}(\Pi_v)$ plane) of the segments formed by intersecting $\mathcal{N}(\Pi_v)$ with the rectangles in $Cover(v)$. We do this by merging $Vis\_Cover\_Short(v)$ with $Vis\_Northern\_Long(w)$ (resp. $Vis\_Southern\_Long(w)$) such that w is a sibling of v and w is to the right (resp. left) of v. Since any rectangle that covers v either has both its horizontal boundaries in Π_z or has one in a Π_w (where w is a sibling of v) and the other outside of Π_z, this gives us Vis_v for each v in T. Since we need perform only a single merge for each node v, this step can be implemented in $O(\log \log n)$ time using $O(n \log n)$ work [10], [31], [54]. This completes the construction.

Therefore, we have the following lemma:

LEMMA 5.2. *Given a collection of iso-oriented rectangles in $\Re^3$, and a segment tree T built upon their horizontal boundaries, the upper envelope of the rectangles in $Cover(v)$ for each v in T (Step 2 in our algorithm for hidden-surface elimination for rectangles) can be constructed in $O(\log n)$ time using $O(n)$ processors in the CREW PRAM model.*

6. Conclusion. We have shown how to design efficient parallel algorithms for a number of problems whose efficient sequential algorithms use various versions of the plane-sweeping paradigm. These problems included general hidden-surface elimination, CSG boundary-evaluation, rectangle contour computation, and hidden-surface elimination for rectangles. An interesting open problem that remains is to determine if *a priori* knowledge of all the "events" in a plane sweep is required in order to derive an optimal parallel algorithm for a problem solved sequentially by that plane sweep. Perhaps the most challenging such problem at the present time is to determine if the segment arrangement of n line segments in the plane can be computed in $O(\log n)$ time using $O(n + I/\log n)$ processors, where I is the number of intersections (recall that this was the bottle-neck computation in our hidden-surface elimination and CSG boundary evaluation algorithms). This problem can be solved sequentially in $O(n \log n + I)$ time using the beautiful, but rather involved, plane-sweeping algorithm of Chazelle and Edelsbrunner [14].

References

[1] K. Abrahamson, N. Dadoun, D. A. Kirpatrick, and T. Przytycka, A Simple Parallel Tree Contraction Algorithm, TR 87-30, Department of Computer Science, University of British Columbia, 1987.

[2] A. Aggarwal, B. Chazelle, L. Guibas, C. Ó'Dúnlaing, and C. Yap, Parallel computational geometry, *Algorithmica*, **3**(3) (1988), 293–328.

[3] R. J. Anderson and G. L. Miller, Deterministic parallel list ranking, AWOC 88, Lecture Notes in Computer Science, Vol. 319, Springer-Verlag, Berlin, 1988, pp. 81–90.

[4] M. J. Atallah, R. Cole, and M. T. Goodrich, Cascading divide-and-conquer: a technique for designing parallel algorithms, *SIAM J. Comput.*, **18**(3) (1989), 499–532.

[5] M. J. Atallah and M. T. Goodrich, Efficient parallel solutions to some geometric problems, *J. Parallel Distrib. Comput.*, **3**(4), 1986, 492–507.
[6] M. J. Atallah, M. T. Goodrich, and S. R. Kosaraju, Parallel algorithms for evaluating sequences of set-manipulation operations, AWOC 88, Lecture Notes in Computer Science, Vol. 319, Springer-Verlag, Berlin, 1988, pp. 1–10.
[7] B. G. Baumgart, A polyhedron representation for computer vision, *Proc. 1975 AFIPS National Computer Conf.*, AFIPS Press, 1975, pp. 589–596.
[8] J. L. Bentley and D. Wood, An optimal worst case algorithm for reporting intersections of rectangles, *IEEE Trans. Comput.*, **29**(7) (1980), 571–576.
[9] M. Bern, Hidden surface removal for rectangles, *Proc. 4th ACM Symp. on Computational Geometry*, 1988, pp. 183–192.
[10] A. Borodin and J. E. Hopcroft, Routing, merging, and sorting on parallel models of computation, *J. Comput. System Sci.*, **30**(1) (1985), 130–145.
[11] R. P. Brent, The parallel evaluation of general arithmetic expressions, *J. Assoc. Comput. Mach.*, **21**(2) (1974), 201–206.
[12] S. Chandran and D. Mount, Shared memory algorithms and the medial axis transform, *Proc.* 1987 *IEEE Workshop on Computer Architecture for PAMI*, 1987.
[13] B. Chazelle, Intersecting is easier than sorting, *Proc. 16th ACM Symp. on Theory of Computing*, 1984, pp. 125–134.
[14] B. Chazelle and H. Edelsbrunner, An optimal algorithm for intersecting line segments in the plane, *Proc. 29th IEEE Symp. on Foundations of Computer Science*, 1988, 590–600.
[15] B. Chazelle and L. J. Guibas, Fractional cascading: I. A data structuring technique, *Algorithmica*, **1**(2), 133–162.
[16] A. Chow, Parallel algorithms for geometric problems, Ph.D. thesis, Department of Computer Science, University of Illinois, 1980.
[17] R. Cole, Parallel merge sort, *SIAM J. Comput.*, **17**(4) (1988), 770–785.
[18] R. Cole and U. Vishkin, Approximate scheduling, exact scheduling, and applications to parallel algorithms, *Proc. 27th IEEE Symp. on Foundations of Computing*, 1986, pp. 478–491.
[19] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, Making data structures persistent, *Proc. 18th ACM Symp. on Theory of Computing*, 1986, pp. 109–121.
[20] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer-Verlag, New York, 1987.
[21] H. Edelsbrunner, H. A. Maurer, F. P. Preparata, A. L. Rosenberg, E. Welzl, and D. Wood, Stabbing line segments, *BIT*, **22** (1982), 274–281.
[22] M. T. Goodrich, A polygonal approach to hidden-line elimination, *Proc. 25th Allerton Conf. on Communication, Control, and Computing*, 1987, pp. 849–858.
[23] M. T. Goodrich, Intersecting line segments in parallel with an output-sensitive number of processors, *Proc.* 1989 *ACM Symp. on Parallel Algorithms and Architectures*, 1989, pp. 127–137.
[24] M. T. Goodrich, Applying parallel processing techniques to classification problems in constructive solid geometry, *Proc. 1st ACM–SIAM Symp. on Discrete Algebra*, 1990, pp. 118–128.
[25] M. T. Goodrich, M. J. Atallah, and M. Overmars, An input-size/output-size trade-off in the time-complexity of rectilinear hidden surface removal, *ICALP* '90, to appear.
[26] M. T. Goodrich and S. R. Kosaraju, Sorting on a parallel pointer machine with applications to set expression evaluation, *Proc. 29th IEEE Symp. on Foundations of Computing*, 1989, pp. 190–195.
[27] G. H. Güting, An optimal contour algorithm for iso-oriented rectangles, *J. Algorithms*, **5** (1984), 303–326.
[28] R. H. Güting and T. Ottmann, New algorithms for special cases of the hidden line elimination problem, *Proc. Symp. on Theoretical Aspects of Computer Science*, 1985, pp. 161–171.
[29] J. E. Hopcroft, and J. D. Ullman, Set merging algorithms, *SIAM J. Comput.*, **2**(4) (1973), 294–303.
[30] S. R. Kosaraju and A. L. Delcher, Optimal parallel evaluation of tree-structured computations by raking, *AWOC* 88, Lecture Notes in Computer Science, Vol. 319, Springer-Verlag, Berlin, 1988, pp. 101–110.
[31] C. P. Kruskal, Searching, merging, and sorting in parallel computation, *IEEE Trans. Comput.*, **32**(10) (1983), 942–946.
[32] C. P. Kruskal, L. Rudolph, and M. Snir, The power of parallel prefix, *Proc.* 1985 *Internat. Conf. on Particle Processing*, 1985, pp. 180–185.
[33] R. E. Ladner and M. J. Fischer, Parallel prefix computation, *J. Assoc. Comput. Mach.*, (1980), 831–838.

[34] D. T. Lee and F. P. Preparata, Computational geometry—a survey, *IEEE Trans. Comput.*, **33**(12) (1984), 872–1101.

[35] W. Lipski, Jr., and F. P. Preparata, Finding the contour of a union of iso-oriented rectangles, *J. Algorithms*, **1** (1980), 235–246.

[36] Y. Matias and U. Vishkin, On parallel hashing and integer sorting, Report UMIACS-TR-90-13, Institute for Advanced Computer Studies, University of Maryland, 1990.

[37] K. Mehlhorn, S. Näher, and C. Uhrig, Hidden line elimination for isooriented rectangles, ESPRIT Technical Report 90-25, 1990.

[38] G. L. Miller and J. H. Reif, Parallel tree contraction and its application, *Proc. 26th IEEE Symp. on Foundations of Computing*, 1985, pp. 478–489.

[39] G. L. Miller and S. H. Teng, Dynamic parallel complexity of computational circuits, *Proc. 19th ACM Symp. on Theory of Computing*, 1987, pp. 254–263.

[40] D. E. Muller and F. P. Preparata, Finding the intersection of two convex polyhedra, *Theoret. Comput. Sci.*, **7**(2) (1978), 217–236.

[41] M. S. Paterson and F. F. Yao, Binary partitions with applications to hidden surface removal and solid modeling, *Proc. 5th Symp. on Computational Geometry*, 1989, pp. 23–32.

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

[43] F. P. Preparata, J. S. Vitter, and M. Yvinec, Computation of the axial view of a set of isothetic parallelepipeds, Report LIENS-88-1, Dépt. de Math. et d'Info., Lab. d'Informatique de L'Ecole Normal Supérieure, 1988.

[44] J. H. Reif and S. Sen, An efficient output-sensitive hidden-surface removal algorithm and its parallelization, *Proc. 4th Symp. on Computational Geometry*, 1988, pp. 193–200.

[45] A. A. G. Requicha, Representations for rigid solids: theory, methods, and systems, *ACM Comput. Surveys*, **12**(4) (1980), 437–464.

[46] C. Rüb, Computing intersections and arrangements for red–blue curve segments in parallel, *Proc. 4th Canadian Conf. on Computational Geometry*, (1992), pp. 115–120.

[47] A. Schmitt, On the time and space complexity of certain exact hidden line algorithms, Report 24/81, Fakultät für Informatik, University of Karlsruhe, 1981.

[48] Y. Shiloach and U. Vishkin, An $O(\log n)$ parallel connectivity algorithm, *J. Algorithms*, **3** (1982), 57–63.

[49] E. Soisalon-Soininen and D. Wood, Optimal algorithms to compute the closure of a set of iso-rectangles, *J. Algorithms*, **5** (1984), 199–214.

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

[51] R. E. Tarjan and U. Vishkin, Finding biconnected components and computing tree functions in logarithmic parallel time, *SIAM J. Comput.*, **14** (1985), 862–874.

[52] R. B. Tilove, Set membership classification: a unified approach to geometric intersection problems, *IEEE Trans. Comput.*, **29**(10) (1980), 874–883.

[53] R. B. Tilove, A null-object detection algorithm for constructive solid geometry, *Comm. ACM*, **27**(7) (1984), 684–694.

[54] L. G. Valiant, Parallelism in comparison problems, *SIAM J. Comput.*, **4**(3) (1975), 348–355.

[55] H. Wagener, Optimally parallel algorithms for convex hull determination, Manuscript, 1985.

[56] H. Wagener, Optimally parallel hull construction for simple polygons in $O(\log \log n)$ time, *Proc. 33rd IEEE Symp. on Foundations of Computing*, 1992, pp. 513–599.

[57] P. Widmayer and D. Wood, Time and space-optimal contour computation for a set of rectangles, *Inform. Process Lett.*, **24** (1987), 335–338.

[58] D. Wood, The contour problem for rectilinear polygons, *Inform. Process Lett.*, **19** (1984), 229–236.