Generalized Sweep Methods for Parallel Computational Geometry

(Preliminary Version)

Michael T. Goodrich*,† Mujtaba R. Ghouse† Jonathan Bright†

Dept. of Computer Science, The Johns Hopkins Univ., Baltimore, MD 21218

Summary of Results

In this paper we give efficient parallel algorithms for a number of problems from computational geometry by using generalized 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, and
- hidden-surface elimination for rectangles.

Our algorithms are for the CREW PRAM.

1 Introduction

There are a number of algorithms in computational geometry that rely on the "sweeping" paradigm (e.g., see [15, 24, 32]). 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 all together and solve the problem using a completely different paradigm. Examples include the line-segment intersection methods of Rüb [36] and Goodrich [18], the trapezoidal decomposition algorithm of Atallah, Cole, and Goodrich [4], the method of Aggarwal, Chazelle, Guibas, Ó Dúnlaing, and Yap [2] for constructing Voronoi diagrams, and the method of Chow [11] for computing rectangle intersections. A notable exception, which kept with the plane sweeping approach, were the methods of Atallah, Cole, and Goodrich [4] for 2-set dominance counting, visibility from a point, and computing 3-dimensional maxima points. In each of these algorithms, Atallah *et al.* parallelized 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 general methods for parallelizing various 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 technique does 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 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 $(0, 0, +\infty)$ [17, 37, 38]. We show that this problem can be solved in $O(\log n)$ time using $O((n + I)\log^2 n)$ work (i.e., $O((n + I)\log n)$ 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 2-D case) or polytopes (in the 3-D 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) [35, 40, 41]. The problem is to construct a boundary representation for the object described by the root of T. We show

*This research was supported by the National Science Foundation under Grant CCR-8810568.

†This research was supported by the NSF and DARPA under Grant CCR-8908092.

that the 2-D version of this problem can be solved in $O(\log n)$ time using $O((n+I)\log^2 n)$ work, and we also show how to extend this method to 3-D 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 [9, 25, 43, 44]. 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 $(0, 0, +\infty)$ [7, 17, 20, 22, 27, 33]. 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, Goodrich, and Kosaraju [5] called the *array-of-trees*. We apply this data structure in a variety of ways in order to solve each of the above problems.

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* [18, 36]. In this scheme one allows r new processors to be allocated in time t only if one has already constructed an r-element array that stores pointers to the r tasks these processors are to begin performing in step $t+1$. This is essentially the same as the traditional CREW PRAM model, except that in the traditional model one performs only one request, 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).

2 Parallel Persistence

We begin our discussion by reviewing a parallel data structure called the array-of-trees, and presenting 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.* [14]. In our framework one is given a linked data structure D, 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[1] to D. The interpretation of the sequence σ is that operation i updates the structure resulting from performing operations $1, 2, .., i-1$. 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. We show in this section that, for a variety of "skeleton" structures, D, if one is given the entire sequence σ in advance, then one can solve this problem quickly in parallel.

2.1 The Array-of-Trees

Atallah, Goodrich, and Kosaraju [5] were the first to address this problem in a parallel setting, and give 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[2], using $O(n+m)$ processors in the CREW PRAM model, where $m = |\sigma|$. The resulting data structure is called the *array-of-trees* [5].

The array-of-trees, which we will denote by $B(r)$, is defined recursively on the nodes in T (r is the root of T). In each leaf node x of T one stores $\sigma(x)$, the subsequence of σ consisting of all operations that have x as their argument (in the order that they appear in σ). With each operation σ_t in $\sigma(x)$ one associates a record (t, *vals*, *left*, *right*), where t is the position of this operation in σ (i.e., its "time of execution"), *vals* is a list of values for x, and *left* and *right* are pointers (which are null for leaf x). The values stored in the *vals* list depend only on x and σ_t. For example, if one is interested in counting active leaves, then *vals* could store "*count* = 0" if $\sigma_t = disable(x)$ and "*count* = 1" if $\sigma_t = enable(x)$. 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 $(t_1, t_2, ..., t_{|B(x)|})$ denote the list of t-values in $B(x)$, then each record $(t_i, vals, \text{null}, \text{null})$ 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 one defines $B(v)$ 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

[1] In the sequential setting one is also allowed to change links [14].

[2] If all future queries need go no deeper than the root of T, then the space can be reduced to $O(m)$ [5].

$B(u)$ 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 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 one could have 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 $(t_1, t_2, ..., t_{|B(v)|})$ 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 $t_{i+1} - 1$.

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, one solves the problem 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 1-dimensional range-query based on the y-coordinates of the endpoints of s.

One can also use more complicated types of combining rules. For example, Goodrich [18] 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 one is combining two records α_l and α_r 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, he adds the following test:

> 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. 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. In the full version of the paper we show that it is easy to extend the method of Atallah, Goodrich, and Kosaraju [5] to construct this version of the array-of-trees with the same performance as before, i.e., $O(\log n)$ time and $O(n \log n)$ space using $O(n)$ processors.

We also allow one to define a "pruned" version of the compressed array-of-trees [18]. 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 $B(u)$. Thus, in this case, each record α in $B(v)$ corresponds to the root of a binary tree containing the number of 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 generalize 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 pseudo-tree, for fixed k. A *pseudo-tree* is a directed acyclic graph $G = (V, E)$ such that the nodes in V have been partitioned into $V_1, V_2, ..., V_m$ with the V_i's forming the nodes of a 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 pseudo-tree is of *order* k if $|V_i| \leq k$ for each $i \in \{1, 2, ..., m\}$. Thus, if G is a tree, it is an order-1 pseudo-tree. 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 pseudo-tree with k being $O(1)$. In the final version of this paper we show that one can use the cascading merge scheme of Goodrich and Kosaraju [21], which is based on linked-lists instead of arrays, to construct this "array-of-pseudo-trees" data structure in $O(\log n)$ time and $O(n \log n)$ space using $O(n)$ processors. Note: the method of Atallah, Goodrich, and Kosaraju [5] 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 $f_v: \mathcal{U} \times \mathcal{U} \rightarrow \mathcal{U}$ 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, i.e., an algebraic structure that satisfies the composition, closure, and combination properties of Miller and Teng [29]. Intuitively, an algebraic structure $(\mathcal{U}, \mathcal{F})$ is contractable if one can apply the parallel tree-contraction schemes of Brent [8] or Miller and Reif [28] to evaluate T in $O(\log n)$ time using $O(n)$ processors (which can in fact be reduced to $O(n/\log n)$ [1, 23]). For example, any semi-ring is contractable [8, 28], and so is the algebraic structure defined by the boolean operations used in CSG evaluation (on the universe $\{0,1\}$) [19].

Suppose that, in addition to the expression tree T, one is given a sequence σ of m update operations for the leaves of T. That is, each t-th 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 $\mathcal{U}$. The *off-line expression-evaluation* problem is to determine for each $t \in \{1, 2, ..., m\}$ the value that would be defined by the root of T after sequentially performing the assignments $\sigma_1, \sigma_2, ..., \sigma_t$, given the initial values assigned to the leaves of T.

Using the methods of Abrahamson *et al.* [1] or Kosaraju and Delcher [23] one can convert T into an equivalent circuit $\mathcal{C}$, where $\mathcal{C}$ has $O(\log n)$ depth and is, in fact, an order-4 pseudo-tree. The time needed for this conversion is $O(\log n)$ using $O(n/\log n)$ processors [1, 23]. Given this circuit, and the initial values associated with its "leaves", we then apply the array-of-pseudo-trees 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 $\mathcal{C}$ for each time instance in σ). Moreover, since we are only interested in the value of the "root" of $\mathcal{C}$, we need not store all portions of the array-of-pseudo-trees, and can implement the construction using only $O(n + m)$ space [21]. 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, one can determine the value associated with the root of T 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

There are a number of sequential algorithms that follow an approach of constructing an arrangement [15] 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 planar subdivision, which the following lemma addresses:

Lemma 3.1: *Given a connected planar subdivision R, one can construct a spanning tree on R in $O(\log n)$ time using $O(n/\log n)$ processors.*

Proof: (Sketch) For each face f in R one removes the edge on f that leads, in a counter-clockwise listing, into the vertex on f with smallest x-coordinate (and largest y-coordinate among those, if there are ties). This can easily be implemented in $O(\log n)$ time using $O(n/\log n)$ processors by performing what are essentially list-ranking procedures [3, 13]. In the final version of this paper we prove that this scheme does, in fact, produce a spanning tree for R. □

3.1 Hidden-Surface Elimination

The first application we address is the hidden-surface elimination problem. Suppose one is given a collection of polygonal faces in $\Re^3$ that do not intersect (except possibly at boundaries). The problem is to determine the portions of each polygon that are visible from $(0, 0, +\infty)$ assuming each polygonal face is opaque. 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). One can easily modify our method for the more general case by using parallel prefix computations where appropriate. Our method for solving this problem follows the general approach of Goodrich [17] and Schmitt [37], and consists of the following six steps:

Step 1. In this step we construct the polygon arrangement R of S projected to the xy-plane. This is the connected graph whose nodes are the polygon vertices and the intersection points between pairs of polygonal edges (in the xy-plane), as well as the "shadows" formed by projecting the vertex on each polygon with smallest x-coordinate in the negative y-direction until it hits

some other polygonal edge. Using the parallel segment-intersection algorithm of Goodrich [18], this arrangement can be constructed in $O(\log n)$ time using $O((n+I)\log n)$ processors.

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 generalization of the Euler-tour technique of Tarjan and Vishkin [39] 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 be an edge leaving the (unique) vertex with 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 will be using these points in Step 4). 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 $\sigma = (\sigma_0, \sigma_1, ..., \sigma_m)$ 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 e one would enter (resp., leave) 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 returns 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 (we do not use q in the 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 [13] or Anderson and Miller [3], this requires $O(\log n)$ time using $O((n+I)\log n)$ processors.

Comment: A vertex with 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 edge e_i in their interior, 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 depends crucially on the consistency of the relation $\mathcal{R}$, even in the face of possible cycles and gaps in the overlap relationship. The next two lemmas establish this consistency.

Lemma 3.2: *For any $i \in \{0,1,...,m\}$, if P and Q are both active at time i, then $(P,p) > (Q,q)$ implies $(Q,q) < (P,q)$, where p and q are the respective representatives for P and Q at time i.*

Proof: Suppose, for the sake of contradiction, that there are two polygon-representative pairs (P,q) and (Q,q) active at time i, with $(P,q) > (Q,q)$ and $(Q,q) > (P,p)$. That is, P is above Q at p and Q is above P at q. Since P and Q do not intersect, this implies that in the path from p to q in U we must either (i) leave Q and re-enter Q along an edge not containing q or (ii) leave P and re-enter P along an edge not containing p. But this implies that, in the activations of P and Q at time i, either Q is not paired with q or P is not paired with p, which is a contradiction. □

Thus, for any two active polygon-representative pairs (P,p) and (Q,q), the result of their comparison is the same regardless of whether one uses the point p or the point q to determine it. The next lemma shows that our comparison rule satisfies the transitive rule.

Lemma 3.3: *For any $i \in \{0,1,...,m\}$, 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: Similar to the proof of Lemma 3.2. We omit the details here. □

Thus, the comparison procedure used in Step 5 is consistent; 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 3.4: *Given a collection S of non-intersecting polygons in $\Re^3$, one can solve the hidden-surface elimination problem for S 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.

Let us illustrate this with an example. Suppose one is given a collection of line segments in the plane and one wishes to construct a data structure that allows one to quickly count or report the segments that are intersected by a query line ℓ, or return a line that intersects the most number of line segments. Using a well-known point-line duality [16, 30], 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). Thus, this 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 easily construct the arrangement formed by all the double-wedges, compute a spanning tree of this arrangement, and build an Euler tour of this tree in $O(\log n)$ time using $O(n^2)$ processors. We then can 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 both counting and reporting queries to be performed in $O(\log n)$ time (where in the reporting case, we first determine the number, k, of answers, and then allocate $\lceil k/\log n \rceil$ processors to the task of enumeration). To find a maximum stabbing line one could then perform $O(n^2)$ queries on the array-of-trees (one for each face in the arrangement), followed by a maximum computation to find the line that stabs the most line segments.

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

3.3 CSG Boundary Evaluation

Suppose one is given a collection of "primitive" polygonal shapes and an expression tree T 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 [35]). The problem we address in this subsection is that of constructing a boundary representation for the object defined by the root of T.

Let us first address the 2-dimensional version of the problem, where the primitive objects are simple polygons. Using the approach of Goodrich [18], we can solve this problem in parallel as follows. We construct the arrangement of the polygons that define the primitives, find a spanning forest for this arrangement, and build an Euler tour of each tree in this forest in $O(\log n)$ time using $O((n+I)\log n)$ processors, where I is the number of edge intersections. We then build an instance of the off-line expression evaluation problem by assigning a "0" to each leaf of T, where "0" at leaf i represents "outside of primitive i" and a "1" represents "inside primitive i". The update operations for each edge in a tour are also based on this convention. By then applying the off-line expression evaluation theorem of the previous section, we can evaluate T for each cell of the polygon arrangement. Constructing a boundary representation of the defined region is then a simple matter, since a cell of the arrangement is inside the defined region iff it is labeled with a "1". This gives us the following theorem:

Theorem 3.5: *One can perform 2-D CSG evaluation in $O(\log n)$ time using $O((n+I)\log n)$ 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).*

We can also extend this approach to 3-dimensional CSG evaluation. The method is similar to that in the 2-dimensional case, except that the arrangement to be formed is a polytope arrangement, and the Euler tour traversal now occurs in a spanning forest of this 3-dimensional arrangement. We omit the details here, deriving the following theorem in the final version:

Theorem 3.6: *Given a CSG representation built upon polyhedral primitives, one can perform the CSG boundary evaluation in $O(\log^2 n)$ time using $O(n^2+I)$ processors, where n is the total number of edges and I is the number intersection points in the polyhedral arrangement (which is $O(n^3)$ in the worst case).*

4 Sweeping Through a Set of Rectangles

In this section we address the situation when one wishes 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, and performing hidden-surface elimination on a collection of rectangles in $\Re^3$.

In keeping with our notion of parallel persistence, one of the paradigms we use in our algorithms is that of an *event list*. An event list E is a list representing the history of some variable e. Each record in 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 time refers to the position in the operation sequence of the operation that caused the change). Both of our methods 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 for performing fractional cascading [10] in parallel [4]. Given a directed 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 $(v_1, v_2, ..., v_m)$ and an arbitrary element x, one processor can locate x in all of the $C(v)$'s in $O(\log n + m \log d)$ time where d is the degree of G [10]. Atallah, Cole and Goodrich [4] show how to perform the construction in $O(\log N)$ time with $O(N)$ space and $N/\log N$ processors on a CREW PRAM, where N is $|V|+|E|+\sum_{v \in V} |C(v)|$. Given the $C(v)$ lists at a node and at the neighbors of that node, an auxiliary list $A(v)$ is constructed that contains $C(v)$ and some fraction e of the elements from all the A lists at the neighbors of v. Each element of $A(v)$ stores its rank in $C(v)$ and in the A lists at the neighbors of v. Thus, given the rank of an element c in some $A(v)$ list, the rank of c in $A(w)$ can be found in constant time (for a neighbor w of v), by checking at most $1/e$ of the elements to one side of c in $A(v)$, to find one that occurs in $A(w)$ (which implicitly gives the rank of c in $C(w)$).

This technique is quite useful for reducing the time complexity for performing a sequence of similar searches. In some instances, it is convenient to allow the collection of similar searches to grow as the computation progresses. The next lemma shows that the cost of allowing for this extension is essentially free (in terms of the work needed to simulate it).

Lemma 4.1 (Goodrich [18]): *Given an algorithm A designed for a PRAM model that uses local processor allocation*[3]*, one can simulate A on an analogous PRAM with global allocation in a work-preserving fashion with a $\log p$-factor slow-down, where p is the final number of processors.*

[3] In a local allocation scheme, in any step t, each processor can, if desired, spawn another processor to begin, in step $t+1$, performing a specified task [34].

In the final version of this paper we give an example use of both of these techniques for the problem of building a parallel data structure to compute all the intersections between a line and a polygon in time $O(\log^2 n)$ using $O(1 + k/\log n)$ processors, where k is the size of the output. The method involves building essentially the same data structure as that of Chazelle and Guibas [10] for this problem, but by implementing our method in parallel, using the 2-level array structure of Wagener [42], our constructing requires $O(\log n)$ time using only $O(n/\log n)$ processors. This actually improves the sequential complexity of building this structure, as the method of Chazelle and Guibas runs in $O(n \log n)$ time [10].

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 (non-decreasing) sorted list of the y-coordinates of the endpoints of the segments in S. Let T be the complete binary tree whose $2n+1$ leaves, in left to right order, correspond to the intervals $(-\infty, y_1]$, $[y_1, y_2]$, $[y_2, y_3]$, ..., $[y_{m-1}, y_m]$, $[y_m, +\infty)$, 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. 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 2 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 [6].

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. Sequentially, this problem was first studied by Lipski and Preparata [25], and time optimality was achieved by Wood [44]. Time and space optimality was subsequently achieved by Widmayer and Wood [43]. In the parallel domain, Chandran and Mount [9] 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.

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 one desires the contour cycles, 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[4]. We use the following procedure to determine all the vertical line segments of the contour, then repeat it, exchanging the roles of the x- and y-axes, to obtain the horizontal segments.

Step 1. In this step we build the segment tree, complete with all the *Cover*(v) and *End*(v) lists constructed for each node, where the vertical segments come from the rectangle vertical boundaries and are sorted by the x-coordinates. We view each *Cover*(v) as an event list where x coordinates act as the "time" field. Using the method of Atallah, Cole and Goodrich [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 determined by 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, Cole and Goodrich [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 (maxima) 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). The x-coordinates that determine the "times" in $\mathbf{H}(v)$ correspond to the x-coordinates of the vertical boundaries of the rectangles in *Cover*$(v) \cup$ *End*(v). We also define two flags, *top* and *bottom*, for each entry in an $\mathbf{H}$ list, to mark whether one of the segments in Π_v continues into the respective slab-neighbor of 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 i-th value in $\mathbf{C}(v)$, then (x_i, h_i) is the i-th 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, 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 that 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 $c = 1$, then $h_i = 0$. If $c = 0$, then h_i is the sum of h_u and h_w (taking *top* and *bottom* flags into account), where $\beta = (x_u, h_u)$ and $\gamma = (x_w, h_w)$. We give the details in the final version, showing that this step can be implemented 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 1 iff both of the following hold: (i) the record in $\mathbf{C}(v)$ with largest x-value less than or equal to x_i has c-value equal to 0, and (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.

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.

4.4 Hidden-Surface Removal

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 [22], with more efficient algorithms being recently reported by Goodrich, Atallah, and Overmars [20], Bern [7], Goodrich [17], Mehlhorn [27], and Preparata, Vitter, and Yvinec [33]. 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 [20, 7, 27]. 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

[4] Matias and Vishkin [26] 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.

apply Lemma 4.1 to derive the claimed bounds.

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 e's intersection with $\Pi(v)$ 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 $(0, 0, +\infty)$.

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

2. We construct $Top(v)$ for all nodes v, in $O(\log n)$ time with $O(n)$ processors by a parallel implementation of a method of Goodrich *et al.* [20]; we give the details in the final version.

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 $low_v = \max\{\min\{low_u, low_w\}, top\}$ [7]. 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.

4. In this step we determine all the visible vertical edges. We assign a processor P to every vertical edge e of a input rectangle. P visits every node in the segment tree which e covers, searching down from the root of the segment tree, and spawns as many extra processors as needed to output the pieces of e that are visible.

Comment: Let v be a node such that e spans Π_v. Note that if $z(e) < low$, where low is the value associated with the entry in $Low(v)$ active at $x(e)$, then e is completely obscured by rectangles with edges in the restricted subscene for v. Also, if $z(e) > high$, where $high$ is the active value in $High(v)$, then no rectangle with an edge in the restricted subscene for v is higher than e. This doesn't imply that e is visible in Π_v, however. For e to be visible in Π_v we must also have that $z(e)$ is larger than all the active top values in $Top(v')$ lists, where v' is an ancestor of v.

Implementation of Step 4: Let P be the processor assigned to a vertical edge e on some rectangle. We associate with P a single piece of e, which we have found to be visible (although we haven't output it yet). Assume the processor is at a node v that e covers. If e is completely obscured in Π_v (which P can determine by the above tests[5]), then P outputs I and sets $I = \emptyset$. If e is completely visible in Π_v, then P "grows" I to include the part of e in Π_v. If, on the other hand, P determines that e is only partially visible (because $low < z(e) < high$), then P spawns a new processor P', which descends into the tree below to find the visible parts of e. P' inherits the interval I from P, and P continues it search for the other nodes that e covers (after re-setting its copy of I to $\emptyset$). The task for P' is to output the visible parts of e in Π_v, together with the piece of e (corresponding to I) which P had previously determined to be visible, but had not output. We omit the details here. This step runs in $O(\log n)$ time using $O(n + k)$ processors.

5. If one also wishes 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 step is similar to the previous one; we give its details in the final version.

6. All of the line segments corresponding to a vertical edge of a rectangle have been found. We then run the algorithm once more (with the roles of the x-axis and y-axis reversed) to find the visible horizontal edges.

References

[1] Abrahamson, K., Dadoun, N., Kirpatrick, D.A., and Przytycka, T., "A Simple Parallel Tree Contraction Algorithm," TR 87-30, Dept. of Comp. Sci., Univ. 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," *Lecture Notes 319: AWOC 88*, Springer-Verlag, 1988, 81–90.

[4] M.J. Atallah, R. Cole, and M.T. Goodrich, "Cascading Divide-and-Conquer: A Technique for Designing Parallel Algorithms," *SIAM J. on Comput.*, 18(3), 1989, 499–532.

[5] Note that P must maintain the maximum z-coordinate of active top values from v to the root, but this is easy to do.

[5] M.J. Atallah, M.T. Goodrich, and S.R. Kosaraju, "Parallel Algorithms for Evaluating Sequences of Set-Manipulation Operations," *Lecture Notes 319: AWOC 88*, Springer-Verlag, 1988, 1–10.

[6] J.L. Bentley and D. Wood, "An Optimal Worst Case Algorithm for Reporting Intersections of Rectangles," *IEEE Trans. on Computers*, **C-29**(7), 1980, 571–576.

[7] M. Bern, "Hidden Surface Removal for Rectangles," *4th ACM Symp. on Comp. Geom.*, 1988, 183–192.

[8] R.P. Brent, "The Parallel Evalutation of General Arithmetic Expressions," *J. ACM*, **21**(2), 1974, 201–206.

[9] Chandran, S., and Mount, D., "Shared memory algorithms and the medial axis transform", *1987 IEEE Workshop on Computer Arch. for PAMI.*

[10] B. Chazelle and L.J. Guibas, "Fractional Cascading: I. A Data Structuring Technique," *Algorithmica*, 1(2), 133–162.

[11] A. Chow, "Parallel Algorithms for Geometric Problems," Ph.D. thesis, Comp. Sci., Univ. of Illinois, 1980.

[12] R. Cole, "Parallel Merge Sort," *SIAM J. Comput.*, **17**(4), 1988, 770–785.

[13] R. Cole and U. Vishkin, "Approximate Scheduling, Exact Scheduling, and Applications to Parallel Algorithms," *27th FOCS*, 1986, 478–491.

[14] Driscoll, J.R., Sarnak, N., Sleator, D.D., Tarjan, R.E., "Making Data Structures Persistent," *18th STOC*, 1986, 109–121.

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

[16] H. Edelsbrunner, H.A. Maurer, F.P. Preparata, A.L. Rosenberg, E. Welzl, and D. Wood, "Stabbing Line Segments," *BIT*, **22**, 1982, 274–281.

[17] M.T. Goodrich, "A Polygonal Approach to Hidden-Line Elimination," *25th Allerton Conference on Comm., Control, and Comput.*, 1987, 849–858.

[18] M.T. Goodrich, "Intersecting Line Segments in Parallel with an Output-Sensitive Number of Processors," *1989 SPAA*, 127–137.

[19] M.T. Goodrich, "Applying Parallel Processing Techniques to Classification Problems in Constructive Solid Geometry," *1st ACM-SIAM Symp. Disc. Alg.*, 1990, 118–128.

[20] 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," to appear in *ICALP '90.*

[21] M.T. Goodrich and S.R. Kosaraju, "Sorting on a Parallel Pointer Machine with Applications to Set Expression Evaluation," *29th FOCS*, 1989, 190-195.

[22] R.H. Güting and T. Ottmann, "New Algorithms For Special Cases of the Hidden Line Elimination Problem," *Symp. on Theo. Aspects of Comp. Sci.*, 1985, 161–171.

[23] Kosaraju, S.R., and Delcher, A.L., "Optimal Parallel Evaluation of Tree-Structured Computations by Raking," *Lecture Notes 319: AWOC 88*, Springer-Verlag, 1988, 101–110.

[24] D.T. Lee and F.P. Preparata, "Computational Geometry—A Survey," *IEEE Trans. on Computers*, **C-33**(12), 1984, 872–1101.

[25] W. Lipski, Jr. and F.P. Preparata, "Finding the Contour of a Union of Iso-Oriented Rectangles," *J. Algorithms*, **1**, 1980, 235–246.

[26] Matias Y., Vishkin U., "On Parallel Hashing and Integer Sorting", Report UMIACS-TR-90-13, Inst. for Adv. Computer Studies, Univ. of Maryland, 1990.

[27] K. Mehlhorn, personal communication, October 1989.

[28] G.L. Miller and J.H. Reif, "Parallel Tree Contraction and its Application," *26th FOCS*, 1985, 478–489.

[29] G.L. Miller and S.H. Teng, "Dynamic Parallel Complexity of Computational Circuits," *19th STOC*, 1987, 254–263.

[30] D.E. Muller and F.P. Preparata, "Finding the Intersection of Two Convex Polyhedra," *Theo. Comp. Sci.*, 7(2), 1978, 217–236.

[31] M.S. Paterson and F.F. Yao, "Binary Partitions with Applications to Hidden Surface Removal and Solid Modeling," *5th Symp. on Comp. Geom.*, 1989, 23–32.

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

[33] F.P. Preparata, J.S. Vitter, and M. Yvinec, "Computation of the Axial View of a Set of Isothetic Parallelepipeds," Lab. d'Informatique de L'Ecole Normal Supérieure, Dépt. de Math. et d'Info., Report LIENS-88-1, 1988.

[34] J.H. Reif and S. Sen, "An Efficient Output-Sensitive Hidden-Surface Removal Algorithm and Its Parallelization," *4th Symp. on Comp. Geom.*, 193–200, 1988.

[35] A.A.G. Requicha, "Representations for Rigid Solids: Theory, Methods, and Systems," *ACM Computing Surveys*, **12**(4), 1980, 437–464.

[36] C. Rüb, "Parallel line segment intersection reporting," manuscript, 1989.

[37] A. Schmitt, "On the Time and Space Complexity of Certain Exact Hidden Line Algorithms," Univ. Karlsruhe, Fakultät für Informatik, Report 24/81, 1981.

[38] I.E. Sutherland, R.F. Sproull, and R.A. Schumacker, "A Characterization of Ten Hidden-Surface Algorithms," *Computing Surveys*, **6**(1), March 1974, 1–25.

[39] R.E. Tarjan and U. Vishkin, "Finding Biconnected Components and Computing Tree Functions in Logarithmic Parallel Time," *SIAM J. Comput.*, **14**, 1985, 862–874.

[40] R.B. Tilove, "Set Membership Classification: A Unified Approach to Geometric Intersection Problems," *IEEE Trans. on Computers*, **C-29**(10), 1980, 874–883.

[41] R.B. Tilove, "A Null-Object Detection Algorithm for Constructive Solid Geometry," *Comm. ACM*, **27**(7), 1984, 684–694.

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

[43] Widmayer, P., and Wood, D., "Time and Space-Optimal Contour Computation For a Set of Rectangles", *Info. Proc. Let.*, **24**, 1987, 335–338.

[44] Wood, D., "The Contour Problem For Rectilinear Polygons", *Info. Proc. Let.*, **19**, 1984, 1984, 229–236.