JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 27, 107–117 (1995)

Area and Perimeter Computation of the Union of a Set of Iso-rectangles in Parallel

MOHAN S. KANKANHALLI*,[1] AND WM. RANDOLPH FRANKLIN†,[2]

*Institute of Systems Science, National University of Singapore, Kent Ridge, Singapore 0511; and †Electrical, Computer and Systems Engineering Department, Rensselaer Polytechnic Institute, Troy, New York 12180

Finding the area and perimeter of the union/intersection of a set of iso-rectangles is a very important part of circuit extraction in VLSI design. We combine two techniques, the uniform grid and the vertex neighborhoods, to develop a new parallel algorithm for the area and perimeter problems which has an average linear time performance but is not worst-case optimal. The uniform grid technique has been used to generate the candidate vertices of the union or intersection of the rectangles. An efficient point-in-rectangles inclusion test filters the candidate set to obtain the relevant vertices of the union or intersection. Finally, the vertex neighborhood technique is used to compute the mass properties from these vertices. This algorithm has an average time complexity of $O(((n + k)/p) + \log p)$ where n is the number of input rectangle edges with k intersections on p processors assuming a PRAM model of computation. The analysis of the algorithm on a SIMD architecture is also presented. This algorithm requires very simple data structures which makes the implementation easy. We have implemented the algorithm on a Sun 4/280 workstation and a Connection Machine. The sequential implementation performs better than the optimal algorithm for large datasets. The parallel implementation on a Connection Machine CM-2 with 32K processors also shows good results. © 1995 Academic Press, Inc.

INTRODUCTION

Iso-rectangles (rectangles with sides parallel to the axes) are extensively used in VLSI design [21, 19]. Various geometric problems involving iso-rectangles arise in VLSI since chip layouts are represented as sets of iso-rectangles. Mask information used in the fabrication of integrated circuits is expressed in terms of iso-rectangles. The chip layout is described by different layers each of which consists of a set of iso-rectangles. Each rectangle represents a segment of a metal wire, an ion implantation region or a via contact. An important problem in the area of VLSI design is that of circuit extraction from the mask geometry of the chip. This involves determining the nodes (set of rectangles which are electrically connected) and then finding the node resistance and capacitance (from the area, perimeter of the node and other electrical properties). This information is then utilized for timing verification of the integrated circuit. This problem requires that, given a set of iso-rectangles in a plane, the covered area and the perimeter of the contour of the *union* be determined. The union problem is also important in the field of digital picture processing when the picture is represented by the medial axis transform [24, 5, 25].

Another problem is to compute the area of the *intersection* of several polygons, each defined as a set of iso-rectangles. This is used in calculating the probability of a fatal defect in a chip. For example, suppose that a layer of insulator lies between two conducting layers. Then the area of intersection of the three layers is the probability that a random hole in the middle layer will cause a short between the upper and lower layers. In this paper, we present solutions to two important iso-rectangle problems, the *union* and *intersection* problems. We now give the formal definitions of these problems.

Union Problem

Given $R_1, R_2, ..., R_n$, a list of n iso-rectangles in a plane, the union problem computes the area and perimeter of the region R where

$$R = R_1 \cup R_2 \cup \cdots \cup R_n. \tag{1}$$

Intersection Problem

Given $R_1, R_2, ..., R_n$, a list of n iso-rectangles in a plane, the intersection problem computes the area and perimeter of the union of all pairwise intersections, i.e., the region R, where

$$R = \bigcup \{R_i \cap R_j \mid i, j = 1, ..., n; i \neq j\}. \tag{2}$$

Previous Work

Sequential Algorithms. The union problem is also known as the *measure* problem. It was first proposed by Klee [15]. This two-dimensional problem can be generalized for d dimensions. Klee proposed an $O(n \log(n))$ solution to the one-dimensional problem using sorting. For the two-dimensional problem, Bentley developed an optimal $O(n \log(n))$ solution based on the line-sweep paradigm [4]. This was extended to three dimensions by van Leeuwen

[1] E-mail: mohan@iss.nus.edu.
[2] E-mail: wrf@ecse.rpi.edu.

0743-7315/95 $6.00

and Wood, who used the quad-tree to solve the problem [22]. Güting also has an optimal $O(n \log(n))$ algorithm for two dimensions using the divide-and-conquer strategy [10]. His algorithm computes the contour of the union (set of boundary edges) as well. Widmayer and Wood have a time and space optimal algorithm for computing boolean operations on two masks described by a set of iso-rectangles [23]. They also use the line-sweep paradigm and their algorithm generalizes easily for more than two layers. However, they do not compute the area of the result of the boolean operations. Recently, Overmars and Yap have obtained an $O(n^{d/2} \log(n), n)$ time-space upper bound for the d-dimensional measure problem [18]. They have alongside presented a plane-sweep algorithm for the three-dimensional problem. There are no known published sequential algorithms for explicitly solving the intersection problem, but some of the algorithms for Boolean operations on rectilinear polygons could be used for this problem.

Parallel Algorithms. Kane and Sahni have designed systolic algorithms for rectilinear polygons which are the union of a set of iso-rectangles [13]. They have developed algorithms for OR, AND, Oversizing, and Undersizing operations. The polygons are represented by their edges which are fed to the left end of a systolic chain of processors. As the edges float to the right, they compare themselves with edges that are resident in the processors and generate the result polygons. The output polygons float to the left and are output from the left end of the chain. These algorithms are useful if a dedicated parallel processor is to be built.

Chandran and Mount have presented a parallel algorithm for the union problem which takes $O(\log(n) \log(\log(n)))$ time using $O(n)$ processors for n rectangles [5]. They define horizontal slabs by passing horizontal lines through the endpoints of the vertical edges of all rectangles. Vertical strips are similarly defined. The area and the perimeter of the union of the iso-rectangles is computed by recursively merging the slabs and the strips. This approach uses the plane-sweep paradigm along with Cole's sorting algorithm [6]. Their algorithm can also compute the contour of the union. The algorithm, however, uses complex data structures and seems difficult to implement. The authors have made no mention of implementation.

Lu and Varman have presented optimal algorithms for many iso-rectangle problems on a mesh-connected computer (MCC) [16]. A two-dimensional MCC consists of a number of identical processors arranged in a two-dimensional array. It is a SIMD (single-instruction multiple-data) machine which executes the same instruction in parallel on different data items. They present algorithms for the area of the union and the intersection of a set of iso-rectangles. Both algorithms take $O(\sqrt{n})$ time on a $2\sqrt{n} \times 2\sqrt{n}$ MCC. For the area of the union, the plane is divided into horizontal strips by a set of horizontal lines passing through the points at the top and bottom of the rectangles. The total area is the sum of the covered area of each of the strips. Subsequently, divide-and-conquer is used to divide the strips successively into two equal subsets (slabs) by a vertical line until each slab contains exactly one vertical edge. Then the adjacent slabs are merged together in a binary tree fashion. The slab construction and merge are carried out in parallel. The area-of-the-intersection algorithm is a slight modification of the area-of-the-union algorithm with the extra information of overlap stored in the slabs. The area for a strip is considered only if there is an overlap. They have given detailed algorithms which can be easily implemented on a MCC. However, these algorithms cannot be used for a general purpose parallel computer.

Wu *et al.* have developed algorithms for the union problem which take $O(n)$ time using $O(n)$ processors for a PRAM model and $O(n^2)$ processors for a MCC [25]. They use an *area tree* which is a generalization of the segment tree. The area tree differs from a quadtree in that the grid it is defined on is not necessarily a square and some of the nodes near the bottom of the tree may have two instead of four children. The area is computed by first constructing the area tree and then summing the area for the filled nodes. The perimeter is similarly computed by keeping track of the edges of the rectangle represented by a node and deleting the duplicate entries. The algorithm for the MCC uses a sweep-line, which marks the boundary points of the union, to obtain the area and perimeter.

Miller and Stout have presented a variety of $\Theta(\sqrt{n})$ time algorithms for computational geometry on a MCC [17]. In particular, they present an algorithm for the area of the union of a set of iso-rectangles. Their algorithm is similar to that of Lu and Varman [16]. They maintain slabs of areas by a divide-and-conquer strategy in which the sections counted more than once are discarded. The final area is obtained by merging the areas of the slabs. They claim that their algorithms can also be used on hypercubes. They have not mentioned any implementations—on MCC or otherwise.

Zubair has recently developed an optimal solution to the union problem [26]. He has developed a parallel algorithm based on the divide-and-conquer strategy which has a time complexity of $O(\log\ n\ \log\ \log\ n)$ using $O(n/\log\ \log\ n)$ processors on a CREW PRAM model. The abscissae of the rectangles are sorted and then the input set is divided into two almost equal sets. The solution then recurses on the two subsets till each set contains atmost one rectangle. Then the partial solutions are merged in parallel. The merging is complicated with many special cases and the algorithm assumes the availability of an optimal parallel sorting algorithm. No mention is made of implementation of the algorithm.

From the literature survey presented above, it can be seen that there are no good parallel algorithms for the union and intersection problems. Chandran and Mount [5] provide an algorithm which is complicated and difficult to implement. Wu *et al.* [25] have an algorithm which is of

quadratic time complexity. Lu and Varman [16] and Miller and Stout [17] present parallel algorithms for the specialized mesh connected computer. Zubair [26] has developed a parallel algorithm based on the divide-and-conquer strategy. The algorithm uses the Cole parallel merge-sort algorithm [6] which makes it complex and difficult to implement. It also has to handle a large number of special cases. For none of these algorithms has there been any mention of implementation. The aim here is to develop efficient parallel algorithms which are practical enough to be implemented on real parallel machines.

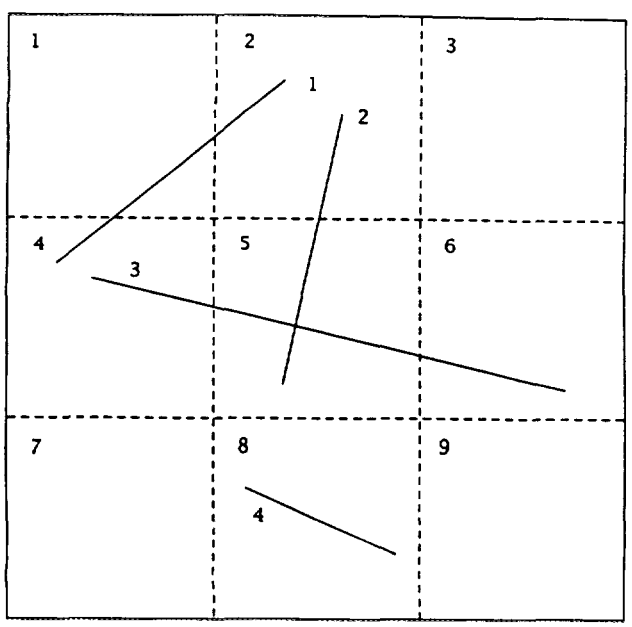
PRELIMINARIES

Our algorithms use the combination of the following techniques for achieving parallelism:

1. *uniform grid*;
2. *vertex neighborhoods*.

The Uniform Grid Technique

The *uniform grid* spatial subdivision technique divides the extant of a scene uniformly into many smaller subregions (cells) of identical shape and size. It is a flat, nonhierarchical grid which is superimposed on the data (Fig. 1). The grid adapts to the data since the number of grid cells, or resolution is a function of some statistic of the input data, such as the average length of the edges. Each geometric entity, like an edge, is entered into a list for each cell it passes through. The technique is like a *divide-and-conquer* mechanism because problems of the same type but of smaller size are generated for each cell. For example, this allows the edges in each cell to be tested against each other for intersections. It has been theoretically and experimentally shown [2] that the uniform grid is an efficient technique for computing edge intersections for random data. The technique takes an average time of $O(n + k)$ for a set of n edges having k intersections. We have also found the uniform grid to be an efficient technique for real world databases such as those from VLSI, cartography, and computer graphics. Other applications and algorithms developed with this technique are presented in [2, 9].



{(2,1),(1,1),(4,1),(2,2),(5,2),(4,3),(5,3),(6,3),(8,4)}

FIG. 1. Uniform grid example and (cell,edge) pairs.

Asano *et al.* [3] contend that the popularity of bucketing schemes similar to the uniform grid is because they often outperform theoretically better algorithms. Pullar [20] has experimentally shown that the sequential version of the uniform grid technique for determining the intersections between a set of segments lying in the plane is faster than the theoretically better plane-sweep technique for a variety of data sets. The overhead of maintaining the dynamic data structure for the plane-sweep technique is the reason for its inefficiency. In the area of parallel processing, the plane-sweep technique is even less attractive when compared to the uniform grid. This is because optimal geometric techniques such as the plane-sweep, impose a temporal ordering of computation on the objects, e.g., by scan line. The uniform grid technique does not impose one when none is strictly necessary. For example, computation of intersections within the grid cells can be done in any order. Therefore, the uniform grid technique is much simpler to parallelize. Its uniformity and regularity also makes the mapping of tasks on to the processors much simpler. When more sophisticated schemes such as quadtrees and octrees are used, distribution of tasks among the processors becomes more complicated with the increased overhead for maintaining the data structures. Also, the non-hierarchical nature of the partitioning exploits the large memories of modern machines and avoids tree data structures and indirect references through pointers to locate objects. This helps in avoiding log factors (due to tree traversals) in the complexity of the algorithm.

Though the uniform grid has many advantages, it also has some important limitations. Uneven spatial densities of the objects reduce the efficiency of the uniform grid technique. However, we have observed that this is not as serious a problem as it appears from a theoretical point of view because in such cases the added complexity of constructing a finer subdivision for better object resolution using an optimal method loses some of the benefits of such a subdivision [14]. Another disadvantage is that in some applications where separate data structures are used for each grid cell, the memory requirements of the uniform grid technique may be greater than that of other methods. However, with the decreasing cost of memory, this may not be a serious disadvantage.

We use this technique in our algorithm to compute intersections of the edges constituting the sides of the input rectangles.

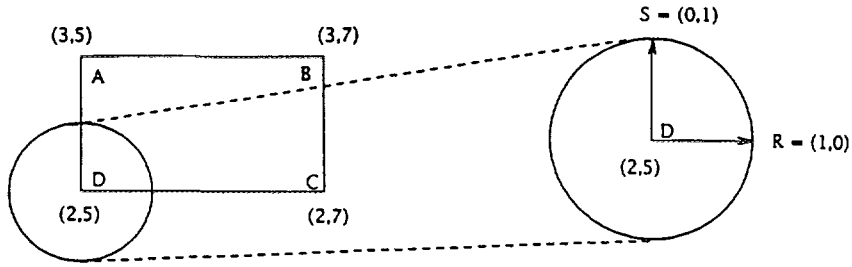


FIG. 2. The vertex neighborhoods for vertex D of a rectangle ABCD.

The Vertex Neighborhoods Technique

For a polygon in 2-D, the neighborhood of a vertex includes location and the directions of its edges, but not their lengths. Explicit global topology of the polygon need not be known. The *vertex neighborhood* of a point (x, y) is represented as a 4-tuple:

$$v = (x, y, R, S).$$

R and S are rays from v in the direction of the two edges incident on v. If we rotate from R to S in a positive direction we will sweep the inside of the polygon ABCD (Fig. 2).

For a polygon P, $X(p)$ is a *characteristic function* iff

$$X(p) = \begin{cases} 1, & \text{if } p \in P, \\ 0, & \text{otherwise.} \end{cases} \tag{3}$$

A *cross function*, $\chi_v(p)$ on each vertex for each of the four wedges is defined as shown in Fig. 3. It can be shown [8] that

$$\chi_v(p) = \begin{cases} \frac{1}{2} - \frac{\theta}{2\pi}, & \text{if } \phi < \pi \text{ and } p \text{ is in a wedge of } \theta, \\ -\frac{\theta}{2\pi}, & \text{if } \phi < \pi \text{ and } p \text{ is in a wedge of } \pi - \theta, \\ \frac{1}{4} - \frac{\theta}{2\pi}, & \text{if } \phi < \pi \text{ and } p \text{ is on the line between two wedges,} \\ \frac{\theta}{2\pi} - \frac{1}{2}, & \text{if } \phi > \pi \text{ and } p \text{ is in a wedge of } \theta, \\ \frac{\theta}{2\pi}, & \text{if } \phi > \pi \text{ and } p \text{ is in a wedge of } \pi - \theta, \\ \frac{\theta}{2\pi} - \frac{1}{4}, & \text{if } \phi > \pi \text{ and } p \text{ is on the line between two wedges.} \end{cases} \tag{4}$$

When p is on a line, its weight is the average of the weights of the neighboring regions. This is required when p is on an extended edge of P. It can be shown [8] that the characteristic function of the whole polygon is the sum of the χ_v functions of the vertices:

$$X(p) = \sum_{v \in V} \chi_v(p). \tag{5}$$

Furthermore, as derived in [8], the area of the polygon can be calculated as

$$\text{Area} = A = \int_{p \in U} \sum_{v \in V} \lim_{R \to \infty} w_R(p) \chi_v(p)\, dp \tag{6}$$

$$= \sum_{v \in V} \lim_{R \to \infty} \int_{p \in U} w_R(p) \chi_v(p)\, dp, \tag{7}$$

where w_R is a sequence of weight functions in the region R of the entire plane U which satisfies the following conditions:

1. $\int_{p \in U} w_R(p)\, dp$ exists,
2. $\lim_{|p| \to \infty} w_R(p) \to 0$, and
3. for all $\varepsilon > 0$ and for all r there exists R_0 such that

$$R > R_0 \quad \text{and} \quad |p| < r = 1 - \varepsilon < w_R(p) < 1 + \varepsilon.$$

This essentially means that we can locally compute a function for each vertex using the vertex neighborhoods and can obtain the global mass property, area, by summing these functions over all the vertices. This means it leads to $O(n)$ algorithms for n vertices. The interested reader is referred to [7, 8] for details, proofs, and extensions to three dimensions.

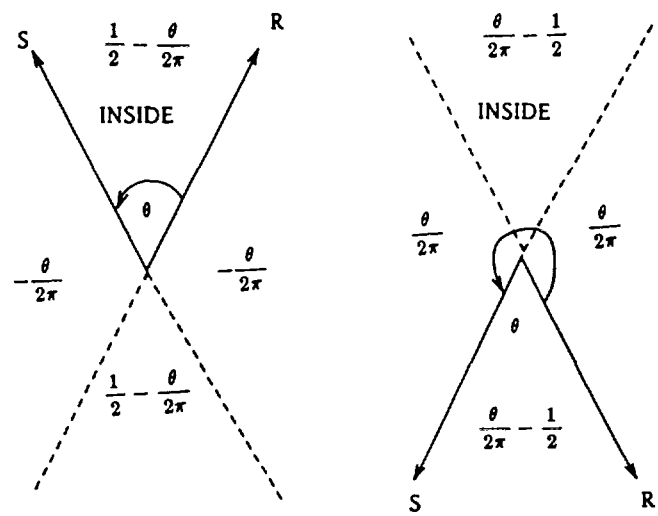


FIG. 3. Cross function on concave and convex vertices.

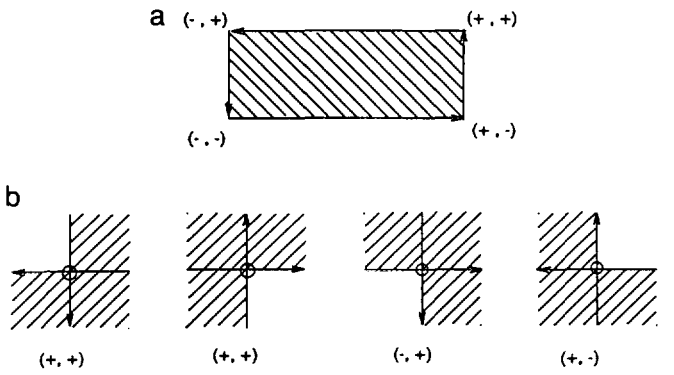


FIG. 4. (a) Signs of input vertices for area computation. (b) Signs of intersection vertices for area computation.

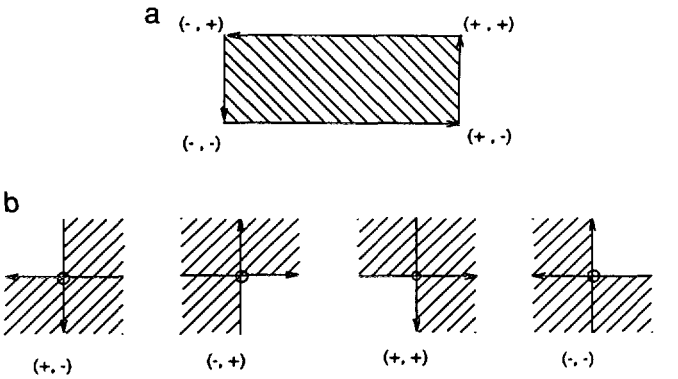


FIG. 5. (a) Signs of input vertices for perimeter computation. (b) Signs of intersection vertices for perimeter computation.

For our case of iso-rectangles, the w_R sequence of weight functions is chosen to be unity in the first quadrant of the plane and zero elsewhere. So, for the special case of iso-rectangles, the positive X and Y axes can be used as cutting lines for the w_R weighting function. In this case, the weighted area for a vertex is the area of the wedge bounded by the positive X and Y axes, taking proper signs into account. So the above relations simplify considerably [14] and we obtain

$$A = \sum_{v \in V} \sigma(x)x\sigma(y)y \quad \text{where } v = (x, y). \tag{8}$$

$\sigma(x)$ and $\sigma(y)$ are the signs assigned as shown in Fig. 4a for vertices in the union polygon which are the input vertices. The corresponding signs for the vertices created by intersection of two rectangles are shown in Fig. 4b. Note that the ordered pair $(+, -)$ in the figure means that $\sigma(x) = +1$ and $\sigma(y) = -1$. We also assume that the interior of the polygon is by convention to the left of the oriented edge. For the perimeter of the union polygon, we have

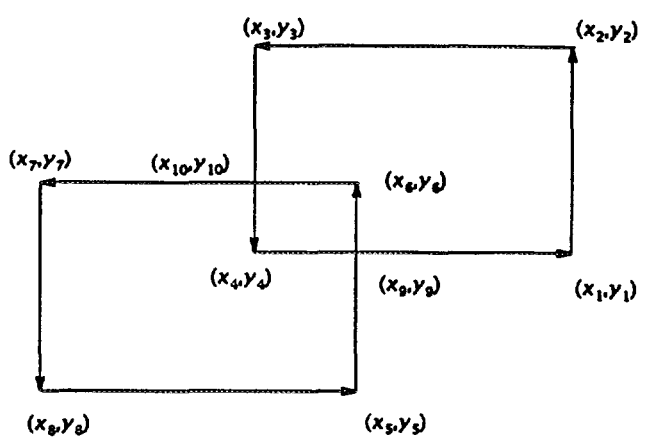
$$P = \sum_{v \in V} (\sigma(x)x + \sigma(y)y). \tag{9}$$

Again $\sigma(x)$ and $\sigma(y)$ are as given in Fig. 5a for input vertices and in Fig. 5b for the intersection vertices. Figure 6 illustrates an example of the use of these relations.

Obviously, these relations can be derived without using the vertex neighborhood technique. The advantage of the vertex neighborhoods technique lies in that these relations can be generalized for any kind of polygons. Furthermore, the relations can be also generalized to any kind of polyhedra in three dimensions [8].

Overview of the Approach

We present a new algorithm for the union and intersection of a set of iso-rectangles in the next section. We use a combination of the uniform grid and vertex neighborhood



$$\text{Area} = -x_1y_1 + x_2y_2 - x_3y_3 - x_5y_5 - x_7y_7 + x_8y_8 + x_9y_9 + x_{10}y_{10}$$

$$\text{Perimeter} = x_1 - y_1 + x_2 + y_2 - x_3 + y_3 + x_5 - y_5 - x_7 + y_7 - x_8 - y_8 - x_9 + y_9 + x_{10} - y_{10}$$

FIG. 6. Example of area and perimeter computation.

techniques in the algorithm. We have also developed an efficient point-inclusion-in-rectangles test. The algorithm has an expected linear time performance, although it is not worst-case optimal. The algorithm time complexity is linear in the size of input and the speedup is also linear with the number of processors.

The uniform grid technique is applied to obtain all the intersections of the edges which constitute the sides of the input rectangles. The union/intersection of the rectangles has two kinds of vertices—points which are vertices of the input rectangles and points which are intersections of the sides of two input rectangles. Thus, the set of intersections of the edges and the input vertices form a superset of the vertices of the union/intersection. Our algorithm efficiently filters out the union/intersection vertices from this superset by using a point-in-rectangles inclusion test. Once we obtain the set of vertices of the union/intersection of the iso-rectangles, the vertex neighborhood technique can be applied to compute the area and perimeter. The input to the algorithm is a set of iso-rectangles and the output is the area and perimeter of the union/intersection of the input set of iso-rectangles.

THE ALGORITHM

We now present the details of the algorithm. The edges of each rectangle are directed such that the interior of the rectangle is to the left of the edge.

1. From the values of N (the number of input edges) and L (the average edge length) determine a grid resolution G. A good heuristic is

$$G = c \min\left(\sqrt{N}, \frac{1}{L}\right), \tag{10}$$

where c is a constant, which can be used for fine tuning [2, 14]. The average edge length can be computed in parallel for each rectangle. The global average can then be computed by a reduction operation.

2. Cast a $G \times G$ uniform grid over the data, in parallel.

3. For each edge of every rectangle, determine the cells it crosses. Store this information in an appropriate data structure which allows retrieval by cell. This can also be done in parallel for each edge.

4. For each rectangle in parallel, determine which cells it completely overlaps. Mark these cells as *covered*. This is analogous to the blocking face concept used in the parallel visible surface determination algorithm in [9]. Here, we wish to record whether any grid cell is completely covered by a rectangle. This information is used to make the point-in-rectangles test efficient.

5. For each cell in parallel, retrieve the edges passing through it and test them against each other for intersections. Store these intersections. The intersection points and the input vertices of the rectangles together constitute the superset of the vertices for the union of the rectangles.

6. In this step, we filter out points from the superset so that we are left with the vertices of the union of the input set of rectangles. Note that points which lie inside any rectangle clearly do not belong to the set of vertices of the union. These are the points which do not lie on the boundary of the union of the iso-rectangles. They must be eliminated to obtain the desired set of vertices. Testing every intersection and input vertex to check if it lies in any rectangle would lead to a quadratic time algorithm. A more efficient method for this point inclusion test is described below. This method takes constant time for each point and therefore is linear for the whole set.

For each of the intersections found in step 5 and the vertices of each input rectangle, determine those points which are *not* inside any of the input rectangles. Note that if a point is *on* a rectangle then it is *not* inside that rectangle. The following steps are performed on each point of the superset in parallel:

(a) Find the cell to which the point belongs. If this cell is marked *covered*, then the point is internal and hence discarded.

(b) If the cell is not *covered*, shoot four rays from the point, in the directions (1,0), (0,1), (−1, 0), and (0, −1) up-to the current cell boundary. Initialize a count variable for each ray to zero.

(i) Find the number of intersections between each ray and the rectangle edges in the cell. If an edge is so oriented that the ray intersects it from outside, add −1 to the count. Add +1 if the ray intersects it from inside.

(ii) If the count is positive for any of the four rays, then the point is inside a rectangle and thus is discarded. Otherwise, the point is included in the output set. Note that checking the intersections in the current cell is sufficient to know whether the point is internal or not. This is due to the fact that unless a cell is completely covered, exactly one boundary of a rectangle containing the point must lie between the point and cell wall. A grid cell can be partially covered by a rectangle in only five ways (Fig. 7). All of these cases can be detected by shooting the four rays. On the other hand, if a cell is fully covered, it would have been marked so in step 4 itself.

7. For all the points found in step 6, assign the sign functions $\sigma(x)$ and $\sigma(y)$ as given in Figs. 4a and 4b for the area and Figs. 5a and 5b for the perimeter. This can be performed in parallel.

8. Find the area and perimeter from Eqs. (8) and (9) given in the previous section, summed over the points obtained in step 6. This is done in parallel for each point. Every processor maintains a partial sum for points in its domain.

9. Compute the area and perimeter of the union by adding up the partial results from each processor. The sums can be computed by forming a binary tree of the processors.

The above algorithm is a parallel solution to the union problem. In step 6 of the algorithm, we classified the inter-

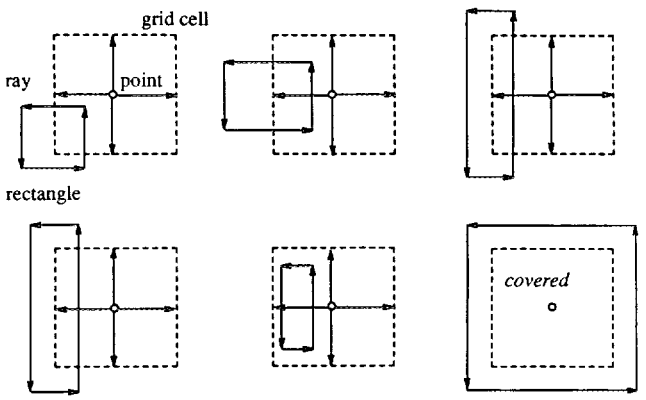


FIG. 7. Various cases of point-in-rectangle test.

section vertices as being either *inside* or *outside*. For the union problem, we discarded the inside vertices and were left with the vertices defining the boundary of the union. To the contrary, if we had discarded the outside vertices, the remaining set would constitute the vertices defining the boundary of the intersection of the rectangles. Therefore, if we modify step 6 of the above algorithm to discard the outside vertices, the resulting algorithm is the solution to the intersection problem.

ANALYSIS OF THE ALGORITHM

We present the analysis of the algorithm on two different computation models. The first analysis is done on the standard CREW PRAM model. This assumes a common shared memory and therefore, any communication step takes $O(1)$ time. While this is a good model for the purpose of theoretical analysis and for shared-memory machines, it ignores the important cost of communication of many real parallel machines.

We, therefore, also present the analysis on a "SIMD machine model" which explicitly takes the communication cost into account. Another reason for presenting this analysis is that we have implemented this algorithm on a Connection Machine 2 (CM-2) and this model is a more accurate representation of the CM-2 than the CREW PRAM model.

CREW PRAM Model Analysis

First, we analyze the algorithm assuming the concurrent-read exclusive-write PRAM model of computation. The edges of the input rectangles are assumed to be independently and identically distributed (IID). For steps 1 through 5 of the algorithm, the analysis of the parallel segment intersection algorithm can be done as shown in Akman *et al.* [2]. Assume that we have n input iso-rectangles with $O(n)$ edges. Assume that a grid size of $G = \lfloor c/l \rfloor$ is chosen, where c is a fine-tuning constant and l is the average edge-length. Finding the cells in which an edge falls takes time proportional to a constant plus the actual number of cells. The expected number of cells covered by the bounding box of an edge is $O(l^2G^2)$. Therefore, the total time required to place the edges in the cells is $O(nl^2G^2/p)$, or simply $O(n/p)$. Therefore, step 1 through 4 of the algorithm will take a time of $O(n/p)$. For the intersection part, there are $O(n)$ (cell,edge) pairs distributed among G^2 cells, for an average of $O(n/G^2)$ edges per cell, or equivalently, an average of $O(n^2/G^4)$ pairs to test. This is true because the edges are IID, their number in any cell is Poisson distributed. For a Poisson distribution, the mean of the square is the square of the mean. The time required to process all the cells becomes $O((n^2/G^4)\ (G^2/p))$ which is equal to $O(n^2/pG^2)$. Now, using $G = \lfloor c/l \rfloor$, this is equal to $O(n^2l^2/p)$. Now, $O(n^2l^2)$ is the expected number of intersections of the edges which is equal to $O(k)$. Therefore, step 5 will take a time of $O(k/p)$. The point-in-rectangle test of step 6 takes $O(1)$ time per point. This leads to a time of $O((n + k)/p)$ for step 6 since we perform the test for every input vertex and every intersection. Steps 7 and 8 also take $O(1)$ time per point resulting in a time complexity of $O((n + k)/p)$ for both these steps. Step 9 is performed in $O(\log_2 p)$ time since the sum can be computed using a binary tree of processors. Therefore, the algorithm for the union and intersection problems has a time complexity of $O(((n + k)/p) + \log_2 p)$ for an input set of n rectangles with k edge intersections on a p-processor CREW PRAM. Thus, the algorithm is linear in the sum of input and output (ignoring the $\log_2 p$ factor which is very small). The speedup of the algorithm is linear in the number of processors.

SIMD Machine Model Analysis

Since the algorithm has been implemented on a CM-2, it is worthwhile to analyze its behavior on a SIMD machine model. The CM-2 works on the principle of data parallelism [11, 12]. Data parallel computing associates one processor with each data element and performs the same operation on the data at each processor. The CM-2 at the Northeast Parallel Architectures Center (NPAC) has 32,768 processors. Each data processor features 64K bits (8 kilobytes) of bit-addressable local memory and an arithmetic-logic unit that can operate on variable length operands. A data processor can access its local memory at a rate of at least 5 megabits per second. The processors communicate among themselves through a router which is configured as a hypercube. Each CM-2 chip contains 16 processors and one router node. These nodes are connected as a hypercube. The throughput of the router depends on the message length and the pattern of accesses; typical throughput values for the CM-2 router are 80 million to 250 million 32-bit accesses per second. We did some timing experiments and found that integer addition takes 10 μs, floating point addition takes 25 μs, move inside a processor takes 15 μs, and a send-to-another-processor operation takes 500 μs. Each pair of CM-2 chips share a floating point accelerator which can operate in single and double precision formats. The language for programming the CM-2 is C* which is an extension of the C language. It

also supports extensions of Lisp and Fortran. The program development is done on the front-end host (a Vax 8000 machine).

For the SIMD machine model, the analysis of the previous section will hold true for all steps except the casting of the grid (step 2). This is because there is no shared memory and thus the processors having edges need to communicate with the processors having the grid cells. This communication cost has to be calculated explicitly for the analysis. We first present the grid-casting algorithm.

Assume that we have $p = 32{,}000$ processors and $n = 1{,}000{,}000$ edges of the input rectangles. Further, assume that the uniform grid is scaled to make the average edge pass through $c = 3$ cells. The algorithm for casting the grid is as follows:

1. Distribute the edges randomly among the processors. Each processor will have on an average $n/p \approx 30$ edges.
2. Determine the cells through which each edge passes. Each processor will find information about $cn/p \approx 100$ cell–edge pairs. At the end of step 1 each processor has been allocated its equal share of edges and cells. In a data parallel computer like the CM-2, each data element is operated on exclusively. Thus, when the processor is working on an edge or a cell, it is unaware of the other edges and cells in its memory (as if they were residing on other processors). This concept of virtual processors occurs when the size of the data is larger than the number of processors. Each processor first operates on the edges in its memory and then starts to process the cells. A processor is an edge-processor or a cell-processor depending on its current function. After each edge-processor has determined the cells its edge passes through, it sends this information to the corresponding cell-processors. Also assume that the cells are randomly assigned to the p processors.
3. In one write step, each edge-processor sends a message to a cell-processor. A cell-processor randomly receives information from the edge-processors at an expected rate of one (cell,edge) pair per step. However, it is quite likely that several edge processors may attempt to send information simultaneously to the same cell-processor. Such a case could occur when the edges of many edge-processors belong to the same cell. Hence, the processor assigned to that cell may receive several messages at the same time.
4. If several edge-processors try to write to the same cell-processor randomly, any one of them will win, i.e., will succeed in writing. In fact this is exactly how the *send with overwrite* instruction is implemented on the Connection Machine.
5. Since, an edge-processor cannot know whether it has won, it must read back the information from the cell-processor after it sends the information to it. If some other processor has won, it must keep on trying to write until it succeeds.

The central issue here is that each edge-processor has some number (a random variable) of messages (edges) to send to a cell-processor. At every communication round, some unknown number of these (ranging from 1 through p) are delivered to the destination processor. The actual number delivered in each round is equal to the number of distinct destinations chosen by the edge-processors. The problem is then to find out how many communication rounds are needed on average to deliver all messages.

The average number of communication rounds is the same as the average number of time steps it takes for an edge processor to send a (cell,edge) pair message successfully. For an estimate of the number of time steps assume that the number of edges x sent to each cell-processor is Poisson distributed with probability density function $f_\lambda(x) = \lambda^x e^{-\lambda}/x!$. Let its probability distribution function be called F. We want to find the mean of $\max_i(f_\lambda(x_i))$ which is the expected number of time steps until the last cell-processor has received its first message (which signifies the number of communication rounds required). In our case, since the number of edge-processors is equal to the number of cell-processors, the mean λ is equal to 1.

Let $g(x) = \max_i(f_\lambda(x_i))$ be a probability density function and G be the corresponding probability distribution function. We want to compute the mean of $g(x)$.

Then, if $1 \le i \le p$ then $G = F^p$. This is because each of the $f_\lambda(x^i)$ is an independent random variable and the probability distribution function of the maximum of independent random variables is the product of the individual probability distribution functions. We will compute the median rather than the mean of $g(x)$ since it is easier to compute.

Therefore, we want to determine the median m which means that $G(m) = 0.5$. This implies that $F(m) = 2^{-1/p} = 0.99998$ for $p = 32{,}000$. From the tables for probability distribution function of a Poisson distribution [1], we find that $m = 8$, satisfies the equation.

Thus, we expect eight tries for the last edge-processor to send its first pair to the appropriate cell-processor. In other words, we expect on the average that eight communication rounds will be required to send all messages. Our implementation results showed that the maximum number of rounds was 11, and the typical number ranged from 1 to 4.

IMPLEMENTATION AND RESULTS

A sequential version of the algorithm was implemented in C and run on a Sun 4/280 machine. The program consisted of 400 lines of code. Several steps of the algorithm were combined for efficient implementation. For example, casting of the grid was combined with the marking of cells as covered; i.e., steps 2, 3, and 4 were combined. Similarly, the point-in-rectangle test was performed for an intersection as soon as it was obtained; i.e., steps 5 and 6 were done together. Finally, steps 7 and 8 were combined to compute the area and perimeter at the same time. Degeneracies of a ray being collinear with an edge were handled by perturbing the ray by a small amount ε. The sequential

TABLE I
Summary of Timings for Sequential Implementation on a Sun4

Database	Rectangles	Intersections of rectangle edges	Grid size	Total time (s)
VLSI data, part of *WaRPI*	2,500	25,232	400	5.32
VLSI data, part of *WaRPI*	25,000	262,058	750	34.28
VLSI data, part of *WaRPI*	50,000	538,776	1,000	82.76
VLSI data, part of *WaRPI*	100,000	1,457,593	1,500	199.36
WaRPI chip	454,766	6,941,110	1,000	875.23

algorithm was tested for several database fragments of the *WaRPI* chip layout (Fig. 8). The correctness of the implementation was validated by testing for various special cases like rectangles with a coincident edge or a vertex. These tests were used for all implementations to check whether they correctly compute the area and perimeter. They also check whether the implementations handle the special cases and degeneracies correctly or not. The timings obtained for computing area and perimeter are summarized in Table I. The actual area and perimeter for the databases are shown only in Table III below, since they were observed to be identical (as they should be) for each implementation.

To get an idea of how our sequential algorithm compares with the worst-case optimal sequential algorithm, we also implemented the optimal algorithm for the union problem on a Sun 4/280 machine. This algorithm was coded in C, as well. The worst-case optimal algorithm for the union problem was suggested by Bentley [4] and is fully described in Preparata and Shamos [19]. This algorithm uses the plane-sweep technique. The vertical edges of the iso-rectangles are sorted by their X coordinate. This sorted list is used to maintain the event point schedule. The sweep-line status is maintained in a segment tree. This algorithm has a time complexity of $O(n \log n)$ which is optimal. The timing of the algorithm applied on the same data sets and running on the same machine is given in Table II.

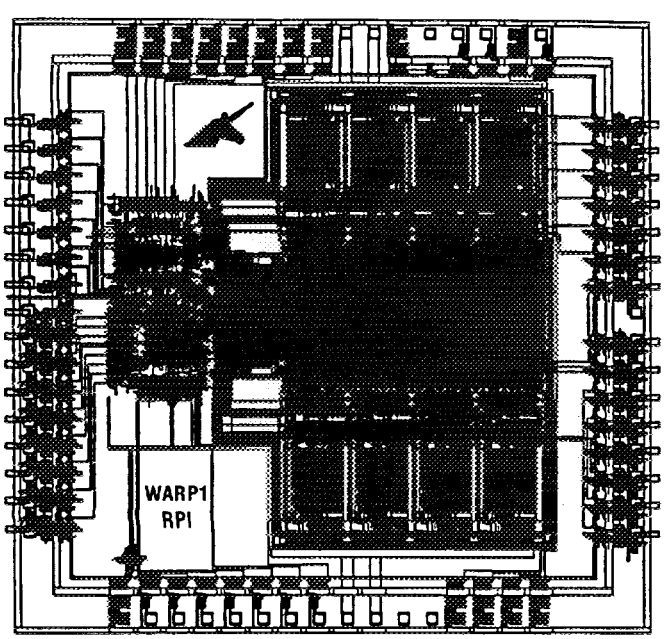


FIG. 8. The *WaRPI* chip.

The results of Table I show that for our algorithm, the growth in time is almost linear. For the optimal algorithm, the time growth is faster (as can be expected from the $O(n \log n)$ time complexity). It is experimentally observed that the optimal algorithm performs better on datasets with less than 100,000 rectangles. However, our algorithm is superior for larger datasets. Comparing the time complexities of the two algorithms, we see that the optimal algorithm is always of $O(n \log n)$ time complexity while our algorithm is of $O(n)$ in the average case and $O(n^2)$ in the worst case.

The parallel algorithm was implemented on the SIMD Connection machine. For the grid casting operation, we used the algorithm presented in the previous section. The algorithm was implemented in C* which has an object-oriented flavor like C++. The major data structure was the *cell* domain. This was the domain definition for edges as well as cells. Domains in C* correspond to the objects in an object-oriented language. These are analogous to classes in C++. On the Connection Machine, each instance of the domain resides on a different processor. Data parallel computation is obtained when the same instruction is executed in parallel on all instances of the domain. The pointer to a domain instance is actually the address of the processor on which that domain instance resides. Therefore, pointers provide the processor address for communication. If the number of domain instances is more than the

TABLE II
Summary of Timings for the Optimal Algorithm on a Sun4

Database	Rectangles	Total time (s)
VLSI data, part of *WaRPI*	2,500	3.28
VLSI data, part of *WaRPI*	25,000	33.74
VLSI data, part of *WaRPI*	50,000	72.54
VLSI data, part of *WaRPI*	100,000	213.02
WaRPI chip	454,766	1,066.32

TABLE III
Summary of Timings for the CM Implementation

Database	Rectangles	Area	Perimeter	Grid size	No. of procs	No. of V procs	Distr. time (s)	Exec. time (s)
WaRPI part	250	0.023	0.81	90	8K	8K	0.11	1.42
WaRPI part	2,500	0.091	2.01	90	8K	8K	1.3	1.41
WaRPI part	25,000	0.144	4.98	512	32K	32K	12.25	1.59
WaRPI part	50,000	0.173	6.36	512	32K	64K	24.3	3.42
WaRPI part	100,000	0.321	10.91	512	32K	128K	47.7	7.48
WaRPI chip	454,766	0.762	19.33	512	32K	512K	216.31	36.21

actual number of processors, the same physical processor is shared by several domain instances. Each of these domain instances represents a *virtual processor*. The Connection Machine features powerful reduction operations. These allow it to compute an associative operator on a variable on all processors in logarithmic time. The result is stored in a front-end host variable. For example, the reduction-sum operation computes the sum of all the instances of a domain variable and writes the sum to a front-end variable. This operation was used for step 9 of the algorithm. The results for the data-sets used in the sequential implementations is given in Table III.

Note that the execution times presented in Table III do not include the time taken to distribute the edge data to the processors. The distribution time is listed in a separate column. It took 216 s to distribute the whole chip data to the CM-2 processors, while it took 36 s to compute the area and perimeter. The overhead in distributing the data becomes much less significant if this algorithm is used as a part of a VLSI design package which would include the complete circuit verification and logic simulation. Also, a faster (and parallel) disk system like the CM-2 Data-Vault would also significantly reduce the I/O time. The total execution time for the first two data sets is almost the same. This is because the size of the data in both cases is less than 8k (the minimum number of processors that can be used). In fact, the total time taken is not very different for two datasets of any size less than the number of processors. If the number of virtual processors exceeds the number of physical processors, the time taken is roughly proportional to the ratio of the number of virtual processors to the number of physical processors. The time grows slightly worse than linear with respect to the virtual processor ratio because of the overhead in managing the context switching of the virtual processors. The maximum number of retries in writing to a common processor ranged from 1 to 11, while the average varied from 1 to 4. For each dataset, the area and perimeter computed by all of the three programs was compared and found to be identical.

CONCLUSIONS

We have presented a new method for computing the area and perimeter of the union/intersection of a set of iso-rectangles. The algorithm uses the uniform grid technique for computing the intersections and the vertex neighborhood technique for computing the area and perimeter. We have also developed a new efficient point-in-rectangles test which is used to determine the vertices of the union/intersection of the iso-rectangles. The method is efficient and very easy to implement. The sequential implementation of the method compares well with the optimal algorithm for the problem. The parallel implementation on the SIMD Connection Machine also exhibits very good performance. This algorithm could be easily extended to find the contour cycles. This would require the use of the planar graph traversal algorithm presented in [9, 14]. Other mass properties such as the moment of inertia can also be computed easily. The ideas presented in this paper can be generalized for other types of polygons and polyhedra. This algorithm also illustrates how the different parallel algorithm design techniques can be combined to develop parallel algorithms for geometric problems.

ACKNOWLEDGMENTS

This work was supported by NSF Presidential Young Investigator Award Grant CCR-8351942 and NSF Grant CCR-9102553. Part of this work was conducted using the computational resources of the Northeast Parallel Architectures Center (NPAC) at Syracuse University, which is funded by and operates under contract to DARPA and the Air Force Systems Command, Rome Air Development Center (RADC), Griffiss Air Force Base, NY, under Contract F306002-88-C-0031. We are thankful to Jim Guilford and Edwin Rogers of the Computer Science Department, RPI for the *WaRPI* chip data. We are grateful to George Nagy of RPI and Jarek Rossignac of IBM T. J. Watson Research Center for their many useful comments. Finally, we thank the anonymous reviewers, whose suggestions have improved the quality of the paper.

REFERENCES

1. Abramowitz, M., and Stegun, I. A. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. U.S. Government Printing Office, Washington, DC, 1972.
2. Akman, V., Franklin, W. R., Kankanhalli, M., and Narayanaswami, C. Geometric computing and uniform grid technique. *Computer-Aided Design* **21,** 7 (Sep. 1989), 410–420.
3. Asano, T., Edahiro, M., Imai, H., Iri, M., and Murota, K. Practical use of bucketing techniques in computational geometry, In G. T. Toussaint (Ed.). *Computational Geometry*. Elsevier Science, Amsterdam, 1985, pp. 153–195.

4. Bentley, J. L. Algorithms for Klee's rectangle problems. Unpublished notes, Carnegie Mellon University, 1977.
5. Chandran, S., and Mount, D. Shared memory algorithms and the medial axis transform. *Proc. 1987 Workshop on Computer Architecture for Pattern Analysis and Machine Intelligence*, Seattle, Oct. 1987, pp. 44–50.
6. Cole, R. Parallel merge sort. *Proc. 26th Annual Symposium on Foundations of Computer Science*, 1986, pp. 511–516.
7. Franklin, W. R. Rays—New representation for polygons and polyhedra, *Comput. Graphics Image Process.* **22** (1983), 327–338.
8. Franklin, W. R. Polygon properties calculated from the vertex neighborhoods. *Proc. Third Annual Symposium on Computational Geometry*, Waterloo, June 1987, pp. 110–118.
9. Franklin, W. R., and Kankanhalli, M. S. Parallel object-space hidden surface removal. *Comput. Graphics* **24**, 4 (Aug. 1990), 87–94.
10. Güting, R. H. Optimal divide-and-conquer to compute measure and contour for a set of iso-rectangles. *Acta Inform.* **21** (1984), 271–291.
11. Hillis, W. D. *The Connection Machine*. MIT Press, Cambridge, MA, 1985.
12. Hillis, W. D., and Steele, G. L. Data parallel algorithms, *Comm. ACM*, **29**, 12 (Dec. 1986), 1170–1183.
13. Kane, R., and Sahni, S. Systolic algorithms for rectilinear polygons, *Computer-Aided Design* **19,** 1 (Jan./Feb. 1987), 15–24.
14. Kankanhalli, M. S. Techniques for parallel geometric computations. Ph.D. thesis, Electrical, Computer & Systems Engineering Department, Rensselaer Polytechnic Institute, Troy, NY, Oct. 1990.
15. Klee, V. Can the measure of $U[a_i, b_i]$ be computed in less than $O(n \log(n))$ steps? *Amer. Math. Monthly* **84,** 4 (Apr. 1977), 284–285.
16. Lu, M., and Varman, P. Optimal algorithms for rectangle problems on a mesh-connected computer, *J. Parallel Distribut. Comput.* **5** (1988), 154–171.
17. Miller, R., and Stout, Q. F. Mesh computer algorithms for computational geometry. *IEEE Trans. Comput.* **38,** 3 (Mar. 1989), 321–340.
18. Overmars, M. H., and Yap, C. New upper bounds in Klee's measure problem. *Proc. 29th Annual Symposium on Foundations of Computer Science*, White Plains, Oct. 1988.
19. Preparata, F. P., and Shamos, M. I. *Computational Geometry*. Springer-Verlag, New York, 1985.
20. Pullar, D. Comparative study of algorithms for reporting geometrical intersections. *Proc. Fourth International Symposium on Spatial Data Handling*, Zurich, July 1990, pp. 66–76.
21. Ullman, J. D. *Computational Aspects of VLSI*. Computer Science Press, Los Alamitos, CA, 1984.
22. van Leeuwen, J., and Wood, D. The measure problem for rectangular range in d-space, *J. Algorithms* **2** (1980), 282–300.
23. Widmayer, P., and Wood, D. A time- and space-optimal algorithm for Boolean mask operations for orthogonal polygons, *Comput. Vision Graphics Image Process.* **41** (1988), 14–27.
24. Wu, A. Y., Bhaskar, S. K., and Rosenfeld, A. Computation of geometric properties from the medial axis transform in $O(n \log n)$ time. *Comput. Vision Graphics Image Process.* **34** (1986), 76–92.
25. Wu, A. Y., Bhaskar, S. K., and Rosenfeld, A. Parallel computation of geometric properties from the medial axis transform. *Comput. Vision Graphics Image Process.* **41** (1988), 323–332.
26. Zubair, M. An optimal speedup algorithm for the measure problem, *Parallel Comput.* **13,** 1 (Jan. 1990), 61–71.

Received November 2, 1991; revised September 13, 1993; accepted June 14, 1994

MOHAN S. KANKANHALLI received the B.Tech. degree in electrical engineering from the Indian Institute of Technology, Kharagpur in 1986. He obtained the M.S. and Ph.D. degrees, both in computer and systems engineering, in 1988 and 1990 from the Rensselaer Polytechnic Institute, Troy, NY. His doctoral work involved developing techniques for designing parallel algorithms for geometric problems. These techniques have been applied to computer graphics, CAD, and solid modeling. Since then, he has been a researcher with the Institute of Systems Science, National University of Singapore. He initially worked with the visualization group on medical imaging. He is now with the multimedia group. He is currently working on projects in graphics modeling and multimedia information systems. His research interests include computer graphics and imaging, multimedia, geometric algorithms, and parallel algorithms.

WM. RANDOLPH FRANKLIN is an associate professor in both the Electrical, Computer, and Systems Engineering Department and Computer Science Department at Rensselaer Polytechnic Institute. He obtained the B.Sc. degree from the University of Toronto in computer science and his A.M. and Ph.D. degrees from Harvard University in applied math. He received an NSF Presidential Young Investigator Award in 1984 and an RPI Early Career Award in 1987. Efficiently processing large geometric databases using parallel computers is the goal of Dr. Franklin's research. Computational geometry, graphics, CAD algorithms, and data structures are the main themes.