Boolean Combinations of Polygons in Parallel

Chandrasekhar Narayanaswami*
IBM Advanced Workstation Division
11400 Burnet Road, Austin, TX 78758

William Randolph Franklin
Electrical, Computer, and Systems Engineering Dept.
Rensselaer Polytechnic Institute
Troy, New York 12180, USA

Abstract

A parallel algorithm to determine the Boolean combinations, such as the union and intersection, of polygons is described. It uses partitioning of geometric space and data parallelism for parallelization. Intersections between the edges of the polygons are first determined to partition the edges into segments that are then classified as either completely inside, outside, or on the other polygon. Depending on the Boolean combination an appropriate subset of these segments is the set of edges in the result.

The algorithm has been implemented on a 16-processor Sequent Balance 21000 shared memory machine and tested with polygons containing several thousands of edges. Parallel efficiencies ($100 \times T_1/(P \times T_P)$) in excess of 70% were obtained consistently. In some cases, efficiencies were as high as 88% with 15 processors.

1 Introduction

The problem of Boolean combination of polygons is as follows: Given the *boundary* of two polygons P_α and P_β, we need to determine the boundaries of the various regularized (no dangling edges) Boolean combinations such as the intersection, union and difference of the two polygons. Figure 4 illustrates the problem.

Boolean combinations of polygons are used for clipping polygons with a view area and computing visible surfaces in computer graphics, for computing the areas of overlapping layers of circuit elements in VLSI, and in the evaluation of the boundary of objects defined in the Constructive Solid Geometry (CSG) representation scheme. We are not aware of any other existing parallel algorithms for this problem [4].

For the purpose of our algorithm the edges can be given in a random order. The edges are oriented such that the interior of the polygon lies to their right. The polygons can have holes and can have more than one disjoint component. The format of the result of our algorithm is the same as that of the input.

2 The Algorithm

The main steps in our CREW PRAM parallel algorithm are as follows:

Step 1:
Compute the points of intersection between the edges of the two polygons.

Step 2:
Use computed intersections to partition the edges of the polygons into sub-edges and classify them as either inside, outside, or on the boundary of the other polygon. Whenever possible, as shown in Figure 2 the sub-edges are classified by an analysis of the orientations of the edges causing the intersection. If the sub-edge lies on the boundary of the other polygon, its relative orientation with respect to the other polygon is also determined. Table 1 shows all the possible classifications for a sub-edge.

Step 3:
Some edges of one polygon may not intersect the other polygon and some of the sub-edges may not be classified in the above step because of collinearity and incidence conditions (see Figure 1). Such fragments are classified with respect to the other polygon by using a point-with-respect-to-polygon test as shown in Figure 3.

Step 4:
Determine which sub-edges or edges need to be included to obtain the boundary of the desired Boolean combination. A salient feature of our algorithm is that the additional cost to determine the result of a different Boolean combination is a pass through the list of sub-edges.

2.1 Computation of the Points of Intersection

The uniform grid technique [1, 5] is used for computing the points of intersection in parallel and is described below:

Step 1:
Partition the 2-D region of interest into $G \times G$ uniform square non-overlapping cells as shown in Figure 3. The fineness of the grid G, is a function of the length of the edges in the input polygons. Usually $G = cL^{-1}$ is a good heuristic, where L is the average length of the edges and c is a tuning constant.

Step 2:
Insert the edges of the polygon into the cells of the grid, i.e., determine the *(cell, edge)* tuples. The cells which an edge passes through are determined by using a variant of the Bresenham raster line drawing algorithm [2]. If an edge passes through a grid corner, it is entered in all four cells adjacent to the corner.

This step can be executed in parallel because the computation is mutually independent with respect to the edges. A round-robin partitioning scheme is used to distribute the edges among the processors. Contention for resources occur in this step when more than one processor wants to write into the same cell's edge list. Collisions are resolved by using *atomic locks* to lock the cell data structure whenever it is updated.

Step 3:
For every pair of edges e_i of P_α, and e_j of P_β in C_i, that intersect at the point (x, y) in C_i, two tuples, $(e_i, e_j,$

*This work was done at Rensselaer Polytechnic Institute

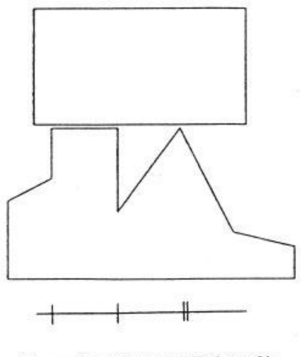


Figure 1: Special Cases Occurring in Polygon Combination. Edges incident on other edges create duplicate intersections.

x, *y*, *type_of_xsect*) and (e_j, e_i, *x*, *y*, *type_of_xsect*) are created. The type of intersection records whether it is due to a vertex incidence or a proper crossing of edges.

To handle cases of collinearity and incidence (see Figure 1), overlapping collinear edges and edges which touch each other at a vertex are not considered to intersect. However, if a vertex of one edge lies in the interior of the other edge, the two edges are considered to intersect. An intersection due to crossing edges is called *proper* and that due to incidence of a vert on an edge is called *improper*.

This step is executed in parallel by distributing the grid cells among the processors because the computation is mutually independent with respect to the cells. To achieve good load-balancing in cases where certain regions contain a significant number number of edges, adjacent cells are given to different processors.

Step 4:

The set of (e_i, e_j, *x*, *y*, *type_of_xsect*) tuples is now sorted by e_i so that all the tuples of intersection along each edge are in consecutive locations. A parallel sort [3] may be used for this purpose.

2.2 Formation and Classification of Sub-edges

This step is parallelized by distributing the (e_i, e_j, *x*, *y*, *type_of_xsect*) tuples among the processors so that each processor is responsible for a few edges. Each edge is divided into sub-edges in the following manner:

Step 1:

The elements of the set of (e_i, e_j, *x*, *y*, *type_of_xsect*) tuples corresponding to intersections along the edge, e_i, to be partitioned and classified, are converted to a set of (e_j, *x*, *y*, *type_of_xsect*) tuples. The fvertices of e_i are also entered into this set of tuples as they each form a vertex of a sub-edge. These tuples are sorted by either the x or the y coordinate of the point of intersection, depending on the slope of the edge. Intersection points that are within a distance of ϵ (an infinitesimal) are coalesced.

Step 2:

Consecutive elements from the sorted set of (e_j, *x*, *y*, *type_of_xsect*) tuples now define the vertices of the sub-edges into which the edge is partitioned. The sub-edge is oriented in the same direction as the original edge of which it is a part. If at least one vertex of the sub-edge was the

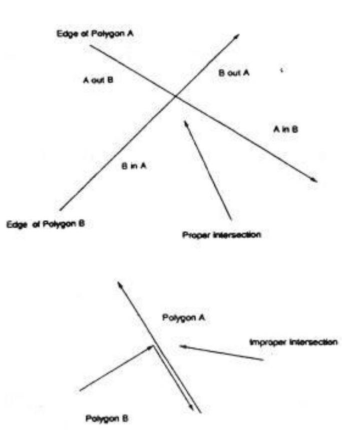


Figure 2: Classification of Sub-Edges at an Intersection Vertex. Sub-edges that are to the left of the edges causing them are outside the other polygon.

result of a *proper* intersection it is a *proper* sub-edge. Other sub-edges are *improper* sub-edges.

Step 3:

Proper sub-edges are classified at this stage by an analysis of the (e_j, *x*, *y*, *type_of_xsect*) tuples responsible for causing them. Figure 2 shows how the orientation of the edges is used for this purpose. The classification of improper sub-edges is described in the following section.

2.3 Residual Segment Classification

Unpartitioned edges and improper sub-edges of partitioned edges are classified by using a point-with-respect-to-polygon test. Note that we cannot propagate the classifications from one unpartitioned edge to another along the boundaries of the polygons as we do not know the contours of the polygons. The point-with-respect-to-polygon test (see Figure 3) is done by passing a horizontal ray from the midpoint of the sub-edge to be classified to the right extremity of the scene boundary, to determine the first edge of the other polygon it intersects. The ray-shooting procedure ends at the first cell in which the ray intersects an edge or edges of the other polygon. In this cell, determine the first edge of the other polygon to intersect the ray. Check whether the point to be classified lies to the right, left, or on this edge and classify the sub-edge appropriately. If the ray does not intersect or lie on any edge, the point to be classified lies outside the other polygon.

This step is done in parallel by distributing among the processors, the edges and sub-edges that are yet to be classified.

2.4 Output Determination

Table 1 shows the elements to be included to obtain the boundary of common Boolean operations. To use it, select the operation desired, and then read down that column. For each row below with a "+" or "-", read to the left to find the type of element to include. A "-" means that the element is to be used with its direction reversed and a "+" means that the element is to be used in the same direction

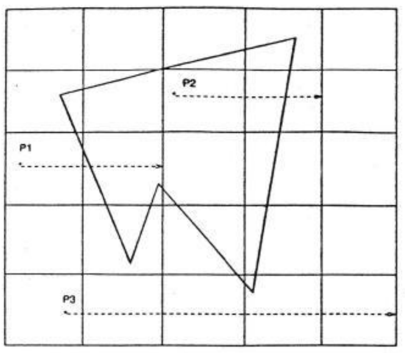


Figure 3: Point Classification w.r.t. Polygons using the Uniform Grid. Points P_1 and P_3 are outside the polygon. P_2 is inside. Note that the ray stops at the appropriate cell.

Table 1: Segments to be Included in Various Polygon Combinations

Elements to include	$P_\alpha \cup P_\beta$	$P_\alpha \cap P_\beta$	$P_\alpha - P_\beta$
Part of P_α inside P_β		+	
Part of P_α outside P_β	+		+
Part of P_β inside P_α		+	−
Part of P_β outside P_α	+		
Part of P_α on P_β, in same dir.	+	+	
Part of P_β on P_α, in same dir.			
Part of P_α on P_β, in opp. dir.			+
Part of P_β on P_α, in opp. dir.			

as reported. For example, if the elements for $P_\alpha \cup P_\beta$ are needed, all elements of P_α which are outside P_β, all elements of P_β which are outside P_α, and all elements of P_α which are on P_β and have the same orientation, i.e., the interiors of both polygons lie to the same side of this element, are included in the result.

This step is done in parallel by distributing the sub-edges among the processors. Since the edges in the result can be specified in any order the processors can write their results in local lists. The local lists are concatenated at the end to have a single list of edges for the output polygon.

3 Complexity of Algorithm

Let n_α and n_β be the number of edges in P_α and P_β, respectively. Let k be the number of intersections.

The cost of determining the cells through which an edge n_i of length l_i passes through is approximately $l_i \times G$. Now $LG = c$ and $L = \sum_{i=1}^{i=n_\alpha+n_\beta} l_i/(n_\alpha + n_\beta)$. In the average case where L/l_i is bounded and thus $l_i \times G$ is bounded, the cost for this step is constant per edge as observed in extensive experiments [5]. Thus the number of *(cell, edge)* tuples is $O(n_\alpha + n_\beta)$ and the sequential complexity of this step is $O(n_\alpha + n_\beta)$.

As shown in [5] the k intersections can be determined sequentially in $O(n_\alpha + n_\beta + k)$ time in the average case.

Every intersection chops an existing edge into two parts. Therefore after the intersection process there are $n_\alpha + n_\beta + k$ sub-edges. The cost to classify the sub-edges determine which sub-edges to use in the output is $O(n_\alpha + n_\beta + k)$ in the average sequential case due to the use of the uniform grid.

From the discussion above, the sequential complexity of the algorithm is linear in the sizes of the tuple-sets created, i.e., it is $O(n_\alpha + n_\beta + k)$.

The following section shows that the parallel implementation yields close to linear speedup and hence the parallelization of the sequential algorithm is quite efficient. Pullar [7] has experimentally shown that in the sequential case the uniform grid technique to compute intersections performs significantly better on many commonly used data sets than more sophisticated theoretical methods such as plane-sweep technique [6], use of quad-trees [8], etc. Therefore, with the help of the above two observations, the parallel algorithm presented does have merit.

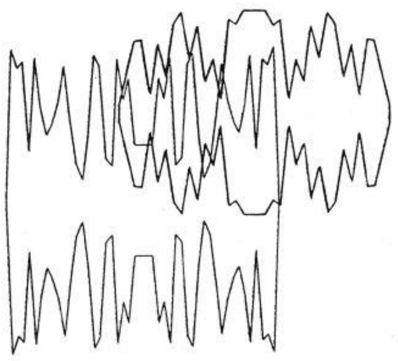
4 Implementation and Results

The algorithm was implemented on a 16-processor Sequent Balance 21000 parallel machine. Shared global arrays were used to store the edges, sub-edges, list of intersections, and the lists of *(cell, edge)* tuples.

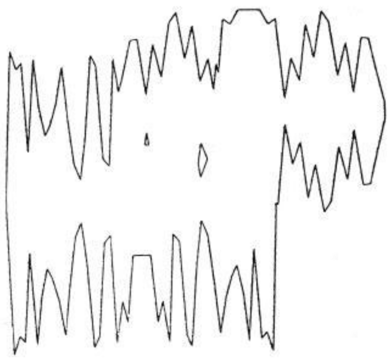
The algorithm was tested on a variety of data sets. Figure 4(a) shows an example where the edges of the two polygons were generated using a random number generator, with the restriction that a polygon should not intersect with itself. In a second example, each polygon consists of a set of squares. Test case polygons which differ drastically in shapes and standard deviation of the edge lengths have been chosen because they are expected to spend different fractions of time in the various phases of the algorithm. In the first example, the two polygons have 12,000 and 6,800 edges respectively, and in the second example, both polygons have 7,200 edges each. Due to the limited resolution of the hard copy devices, the figures show data sets which have fewer edges than the ones actually used. Figure 4(b) shows the results of our algorithm for the data set shown in Figure 4(a).

Figures 5 and 6 show the speedup of the algorithm as a function of the number of processors for both data sets discussed. They show that close to linear speedup was achieved in both cases. This indicates that the total time taken by the algorithm would continue to decrease as more processors are added, till other factors such as bus capacity and inherently sequential sections begin to dominate. A speedup of 13.50 and 10.31 with 15 processors was achieved for the first and second examples, respectively. Using 15 processors, computing all the Boolean combinations for the first example took 177.19 seconds, using a 97×97 grid. For the second example, the corresponding time was 18.03 seconds, using a 100×100 grid.

Table 2 shows the fraction of time taken and the corresponding speedups achieved for each phase of our algorithm for both the examples. Table 3 shows the time taken for the various phases of the algorithm as a function of the number of processors and the grid resolution G, for the first example. This study can be used in estimating the performance of the algorithm when more processors are used, and on machines with different architectures. The following observations can be made by examining Tables 2 and 3.



(a) P_α is the bigger polygon and P_β is the smaller polygon.



(b) Union of Randomly Generated Polygons

Figure 4: Data Set: Randomly Generated Edges

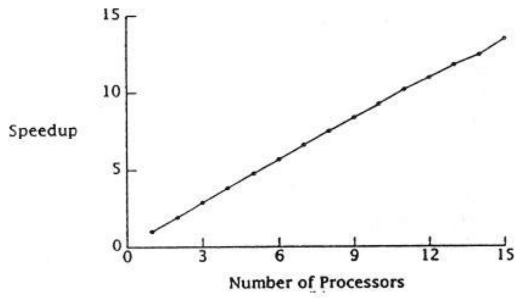


Figure 5: Speedup: Randomly Generated Polygons

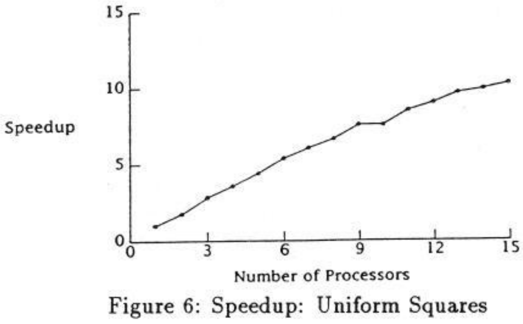


Figure 6: Speedup: Uniform Squares

Table 2: Complexity of Individual Phases of the Polygon Combination

Step 1	Putting edges in the grid
Step 2	Intersecting the edges
Step 3	Sorting the intersections
Step 4	Forming sub-edges
Step 5	Classifying uncut edges
Step 6	Complete polygon combination

Step	Time 1 proc	% time	Time 15 procs	% time	Speedup
Random Polygon Example, Grid Size = 97					
1	39.82	1.66	8.48	4.79	4.70
2	2101.27	87.83	141.34	79.77	14.87
3	4.38	0.18	4.04	2.28	Sequential
4	32.86	1.37	4.85	2.74	6.78
5	214.13	8.96	18.57	10.48	11.53
6.	2392.46	100.00	177.19	100.00	13.50
Square Example, Grid Size = 20					
1	7.96	2.25	4.64	13.52	1.72
2	143.14	40.44	10.93	31.84	13.10
3	2.48	0.70	2.48	7.22	Sequential
4	23.06	6.52	3.28	9.55	7.03
5	177.30	50.09	13.00	37.87	13.64
6.	353.94	100.00	34.33	100.00	10.31

Observation 1:
The overall parallel efficiency $(100 \times T_1/(P \times T_P))$ of the algorithm is more than 80% for the first example and about 70% for the second example. For the first example, the parallel efficiency corresponding to the case which takes the minimum time for the algorithm is about 85%.

Observation 2:
The intersection and the edge classification phases account for a large fraction of the total time of the algorithm. Both these phases show very good speedup (more than 10) and this is the reason for a respectable overall result.

Observation 3:
The performance of the phase which determines the *(cell, edge)* tuples tends to saturate as the number of processors is increased. The benefit accrued by using more processors is negated by the overhead of the locking routines used to resolve collisions in accessing the shared cell data structure. Better speedup for this phase is achievable by using temporary local buffers, which are later copied into the shared cell-entity list, and then sorting the cell-entity list by the cell number with a parallel bucket sort [4].

Observation 3:
When just one processor is used, sorting the $(e_j, x, y, type\_of\_xsect)$ tuples takes only a small fraction (less than 1%) of the total time. Hence our use of a sequential sorting algorithm for this phase does not degrade the performance significantly when 15 processors are used. However, as the number of processors increases, even small sequential sections (such as 1%) in the algorithm limit the maximum parallel efficiency achievable, and the use of a parallel sorting algorithm will be mandatory.

Observation 4:
The time taken by the grid, intersection, and classification phases varies with the grid size. The optimal grid size depends on the exact nature of the dependence of each of these phases on the grid resolution. In the sequential al-

Table 3: Variation of Time with Processors and Grid Resolution

Speedup = T_1(one proc.)/T_P(P procs.).
Parallel Efficiency = $100 \times T_1/(P \times T_P)$

# Procs	Get Xsects	Get Sub Edges	Class. Sub Edges	Total Time	Speed up	Par. Eff.
			Grid Size = 60			
1	2199.73	37.70	340.94	2578.37	1.00	100.00
2	1108.22	22.70	222.79	1353.71	1.90	95.23
3	787.32	16.78	146.31	950.41	2.71	90.43
4	558.70	13.88	110.78	684.36	3.77	94.19
5	455.22	12.21	89.51	556.94	4.63	92.59
6	391.92	11.26	72.75	475.93	5.42	90.29
7	320.32	10.37	62.12	392.81	6.56	93.77
8	281.98	9.87	55.54	347.39	7.42	92.78
9	264.26	9.65	49.09	323.00	7.98	88.70
10	238.95	9.37	44.05	292.37	8.82	88.19
11	205.55	8.97	40.91	255.43	10.09	91.77
12	200.00	8.93	37.25	246.18	10.47	87.28
13	180.02	8.95	35.05	224.02	11.51	88.54
14	164.22	8.86	33.27	206.35	12.50	89.25
15	154.80	8.91	29.80	193.51	13.32	88.83
			Grid Size = 80			
1	2181.56	37.32	258.69	2477.57	1.00	100.00
2	1088.72	22.46	165.05	1276.23	1.94	97.07
3	730.29	16.52	109.78	856.59	2.89	96.41
4	577.21	13.70	82.55	673.46	3.68	91.97
5	487.18	12.02	67.42	566.62	4.37	87.45
6	368.53	11.16	55.73	435.42	5.69	94.83
7	315.48	10.37	47.26	373.11	6.64	94.86
8	302.28	9.88	41.00	353.16	7.02	87.69
9	246.92	9.61	37.42	293.95	8.43	93.65
10	254.21	9.31	33.00	296.52	8.36	83.55
11	203.65	9.28	30.36	243.29	10.18	92.58
12	195.70	9.30	27.57	232.57	10.65	88.78
13	174.08	9.28	25.45	208.81	11.87	91.27
14	162.21	9.03	24.11	195.35	12.68	90.59
15	166.63	9.11	22.27	198.01	12.51	83.42
			Grid Size = 100			
1	2162.23	37.19	214.60	2414.02	1.00	100.00
2	1087.36	22.40	133.46	1243.22	1.94	97.49
3	724.75	16.57	90.03	831.35	2.90	97.19
4	556.26	13.52	67.40	637.18	3.79	95.11
5	463.92	12.06	54.55	530.53	4.55	91.38
6	364.89	10.98	45.33	421.20	5.73	95.92
7	314.69	10.28	38.51	363.48	6.64	95.27
8	279.68	9.84	34.93	324.45	7.44	93.39
9	246.57	9.56	29.84	285.97	8.44	94.18
10	234.75	9.32	26.55	270.62	8.92	89.57
11	202.55	8.98	25.03	236.56	10.20	93.15
12	189.90	9.06	22.19	221.15	10.92	91.34
13	173.10	8.94	20.87	202.91	11.90	91.89
14	161.14	8.68	20.02	189.84	12.72	91.21
15	160.53	8.93	18.42	187.88	12.85	86.01

gorithm for segment intersection, the opposing nature of this dependence for the grid and intersection phases was the reason for the relative insensitivity of the total time as a function of the grid size [5]. A similar behavior is observed in the parallel polygon combination algorithm. In addition, since the various phases of the algorithm yield different speedup curves, the grid size which minimizes the total time can differ with the number of processors used. Table 3 shows that when either 5, 8, or 15 processors are used, the 60 × 60 grid is faster than the 80 × 80 grid. On the remaining occasions, the 80 × 80 grid is faster.

We expect the current implementation of our algorithm to show comparable results on any type of data set for which the sequential algorithm spends large fractions of time in the intersection and classification phases.

5 Conclusions

A parallel Boolean polygon combination algorithm that combined the use of the uniform grid technique, a sort, and data partitioning for parallelization was presented. Parallel efficiencies that were consistently above 70% and speedups between 10-13.5 with 15 processors were obtained on the Sequent Balance 21000 machine.

As shown in [4] these ideas can be extended to determine Boolean combinations of polyhedra and to distributed memory machines such as the Intel Hypercube.

References

[1] V. Akman, W. R. Franklin, M. Kankanhalli, and C. Narayanaswami. Geometric Computing and the Uniform Grid Data Structure. *Computer Aided Design*, 21(7):410–420, September 1989.

[2] J. E. Bresenham. Algorithm for Computer Control of Digital Plotter. *IBM Systems Journal*, 4(1):25–30, 1965.

[3] R. Cole. Parallel Merge Sort. *SIAM Journal of Computing*, 17(4):770–785, August 1988.

[4] C. Narayanaswami. *Parallel Processing for Geometric Applications.* PhD thesis, Rensselaer Polytechnic Institute, Troy, New York 12180, December 1990.

[5] C. Narayanaswami and M. Seshan. The Efficiency of the Uniform Grid for Computing Intersections. Master's thesis, Electrical, Computer, and Systems Engineering Dept., Rensselaer Polytechnic Institute, Troy, NY, December 1987.

[6] F. P. Preparata and M. I. Shamos. *Computational Geometry.* Texts and Monographs in Computer Science. Springer-Verlag, New York, 1985.

[7] D. Pullar. Comparative Study of Algorithms for Reporting Geometrical Intersections. In *Proceedings of the International Symposium on Spatial Data Handling*, pages 66–75, Zurich, July 1990.

[8] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS.* Addison-Wesley Series in Computer Science. Addison-Wesley, 1990.

76-5