Parallel Object-Space Hidden Surface Removal

Wm. Randolph Franklin* and Mohan S. Kankanhalli

Electrical, Computer, and Systems Engineering Dept.
Rensselaer Polytechnic Institute
Troy, NY 12180

Abstract

A parallel object-space hidden surface removal algorithm for polyhedral scenes is presented. The uniform grid technique is used to achieve parallelism for the hidden line removal. A conflict-detection and back-off strategy is then used to obtain parallelism for the visible region reconstruction from the visible segments. The algorithm has been implemented on a Sequent Balance 21000 shared-memory parallel computer. An average speedup of 10 has been obtained using 15 processors.

CR categories: I.3.3, I.3.5, I.3.7.

1 INTRODUCTION

Hidden surface removal has been well-researched by the computer graphics and computational geometry communities. There is a wealth of literature on sequential hidden surface removal algorithms. The classic paper on hidden surface removal algorithms is by Sutherland, Sproull and Schumacker [23]. They introduced the taxonomy of hidden surface algorithms: *image-space* algorithms which iterate over the pixels of the display screen and determine the intensity of each of them, *object-space* algorithms which determine visibility of the objects of the scene such as a face and *list-priority* algorithms which work in object-space initially but the final output is in image-space. Joy et. al. present an up-to-date review of the field in their tutorial on image synthesis [15]. Some recent hidden surface removal algorithms are [17], [18].

There have been several studies on parallel image-space hidden surface removal [4], [7], [10], [14], [16], [19], [24]. Hardware implementations of image-space algorithms have been considered extensively [1], [2], [6], [11], [20], [25], but there has not been much work in the area of parallel object-space hidden surface removal. Hornung has developed an object-space algorithm which reduces the computation time for a network of polygons [13]. He then considers issues of extending this approach for a parallel machine. He gives general guidelines for parallelizing the algorithm but no specific details are given. Rankin describes a hidden line removal algorithm which can be parallelized [21]. However, it cannot be extended for the hidden surface problem. Reif and Sen have presented an object space parallel hidden surface algorithm which runs in time $O(\log^4(n+k))$ using $O(\frac{(n+k)}{\log(n+k)})$ processors on a CREW PRAM model [22]. This method is valid only for surfaces of the form z = f(x,y). The algorithm is extremely complicated and the authors admit that their algorithm is not practical.

Our aim has been to develop an efficient, parallel, object space hidden surface removal algorithm which is simple enough to be implemented on real parallel computers. This algorithm is based on the Franklin algorithm [8].

*Email: wrf@ecse.rpi.edu, Phone: (518) 276-6077

2 PRELIMINARIES

The key behind the parallelism achieved is the Uniform Grid technique. The *uniform grid* is a flat, non-hierarchical grid which is superimposed on the data. 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. The grid is completely regular i.e. it is not finer in the denser regions of the data. The use of the uniform grid technique will become apparent from the algorithm presented in the next section. One objection against the uniform grid technique could be that it is not suitable for irregular scenes and that hierarchical methods such as quadtrees need to be used. But this has not been a problem in practice [3]. From the parallel processing viewpoint, implementation becomes a lot easier with a flat data structure since the overhead on scheduling can be reduced by using a static scheduling scheme. Readers are referred to [3] for further details on the uniform grid technique and its applications to geometric computing.

It is assumed that a perspective transformation has been applied on the scene so that the viewpoint is at infinity in the Z direction and the orthographic projection can be used. The scene consists of polyhedra with non-intersecting planar faces. The polyhedra are specified by their vertices, edges and faces. The vertices (edges) are ordered around the face. It is also assumed that the scene is scaled and projected to fit a 1×1 window. The output is a set of polygons where each polygon is an ordered list of vertices (edges).

Definition: A *blocking face* of a grid cell is the frontmost face, from the viewpoint, whose projection covers the cell completely (Figure 1). If it exists for a cell, then all faces of the cell behind the blocking face are invisible from the viewpoint. The blocking face helps eliminate a lot of unnecessary computation.

The algorithm for hidden surface removal without the

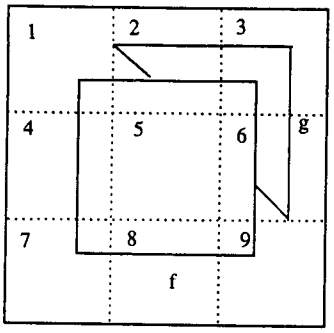


Figure 1: Face "f" is the blocking face for cell 5

visible region reconstruction is now presented. The algorithm for visible region reconstruction in parallel, which uses a conflict-detection and back-off strategy, is presented subsequently.

3 THE ALGORITHM

In this algorithm, the scene is organized into buckets (cells) using the uniform grid technique. The edges are then intersected to obtain the visible segments. The visible regions are obtained from the visible segments. Finally, a point in the visible region is used to find the shading value of that region.

1. Determine a grid size G using some statistic of the input scene, such as $G = c \min(\frac{1}{l}, \sqrt{n})$, where l is the average length of the edges, c is a constant and n is the number of edges.

2. Cast a $G \times G$ grid on the scene. For each of the projected faces in parallel, determine which cells it covers – either fully or partially. Let f = current face being considered and f_b be the current blocking face. If f covers the cell fully, compare it with the f_b of that cell. If f is in front of f_b, then f is the new blocking face, else it is to be discarded. If f does not cover the cell fully and it is in front of f_b, then add it to the list of faces for that cell. This whole step can be done in parallel.

3. For each cell in parallel, compare the partially covering faces with the cell's blocking face. If any face is behind the blocking face then discard it from the list of faces for that cell. This step is needed to remove some partially covering faces that were in front of a previous blocking face but are behind the final blocking face for that cell.

4. For each projected edge in parallel, determine the grid cells it belongs to. Check if the edge is behind the blocking face for each cell. If it is not, then add it to the list of edges for that cell.

5. For each cell in parallel, determine the intersection points of the projected edges in that cell. Consider the intersection only if it lies in that cell. Associate the intersection point with both the edges.

6. For each edge in parallel, sort the intersection points along it and partition the edge into segments. Each segment is either completely visible or completely hidden.

7. For each segment in parallel, check its visibility by comparing with the blocking face and the list of partially covering faces. If it is visible, add it to the list of visible segments.

8. In parallel, determine the regions formed by the visible segments. This is the visible region reconstruction problem. The parallel algorithm for this problem is presented in section 4.

9. For each polygon of the visible regions in parallel, take a point inside that polygon. In the cell containing the point, find the closest face in which the point lies. Since that polygon is a part of that face, assign the shading value of the face to the polygon. If the point does not lie in any face, assign the background shading value to that polygon.

4 VISIBLE REGION RECONSTRUCTION IN PARALLEL

Visible region reconstruction is a planar graph traversal problem. More specifically, let E be a set of edges in the XY-plane such that the members of E constitute a legal planar subdivision. This means that every face (region) is bounded except for the outer ones which are infinite in size. The planar graph traversal finds the regions of the planar subdivision given in terms of its edges. The regions are to be specified by their vertices (or edges) in a positive order. For hidden surface removal, the edges are the visible segments and the regions found are the visible parts of the projected faces of the input scene.

The algorithm we present for this problem is an extension of the Franklin-Akman algorithm [9]. A conflict-detection and back-off strategy is introduced to extend the algorithm for parallel processing. The input is assumed to be a set of visible edges which constitutes a legal connected planar graph. This condition can be relaxed to allow for disconnected components which are not geometrically nested. If this is not satisfied, then the depth ordering of the nested components is not known. So a preprocessing stage of finding the connected components would be necessary. We assume, without loss of generality, that the input has no nested components. Figure 2 shows an example of a legal input graph and an illegal input graph.

4.1 PARALLELISM STRATEGY

Intuitively, the algorithm proceeds in this manner. For every vertex, the edges incident to it are obtained. The edges are then sorted around each vertex in some order (clockwise). Two consecutive edges in this sorted list define a corner of a face. These corners could be merged in parallel to obtain the faces. Each corner is marked in order to be used only once. The merging can be started in parallel, but there is a problem. Ideally, each processor should work on a different face but there is no a priori way of ensuring this. Hence conflicts may arise if two processors are determining the same face. This conflict is resolved using a *conflict-detection and back-off* strategy. Each processor is given a priority number (which is based on the processor identification number).

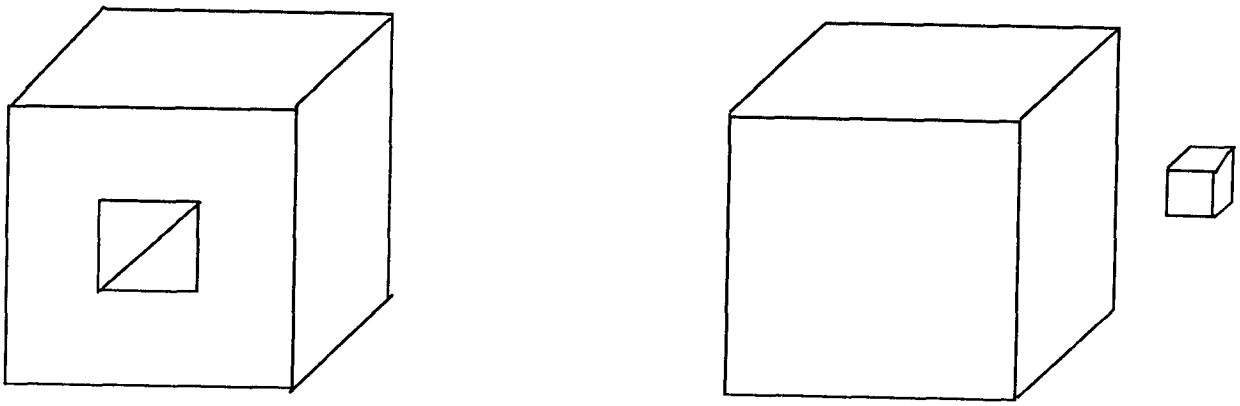


Figure 2: Illegal input on left & legal input on right

In this strategy, each of the corners are initially marked as 'unused'. As soon as a processor works on a corner, if it is marked as 'unused', it is marked with that processor's priority number. If it is not marked as unused, it means it has been processed by some other processor whose priority is the value of the mark and there is a conflict. If that priority is higher than the processor's priority, then the processors backs off and starts working on a new corner. If the priority is lower, then it overwrites the mark with its priority and continues the work. The other processor will ultimately find that this face is being worked on by a higher priority processor, so it will eventually back off. So, in this strategy, the higher priority processor continues the merging and obtains the face. Once a face is obtained, then its vertices are marked so that it is not found by some other processor again. If there are several processors working on the same face, then the processor with the highest priority among them will ultimately traverse the face. This may mean some loss of work and lower efficiency but asymptotically, with a large number of corners, the penalty will not be too much. In practice, the speedup obtained was roughly half of that of the other parts of the hidden surface removal algorithm.

4.2 REGION RECONSTRUCTION ALGORITHM

The input to the algorithm consists of a set of n edges in the XY-plane specified by the end-point coordinates:

$$E = \{((x_{i1}, y_{i1}), (x_{i2}, y_{i2})); i = 1, ..., n\}$$

No order is assumed in E. Now we want to find for each vertex, the edges around it. Therefore, all edges have to be considered twice – once by the first vertex and once by the second vertex. So, we construct in parallel:

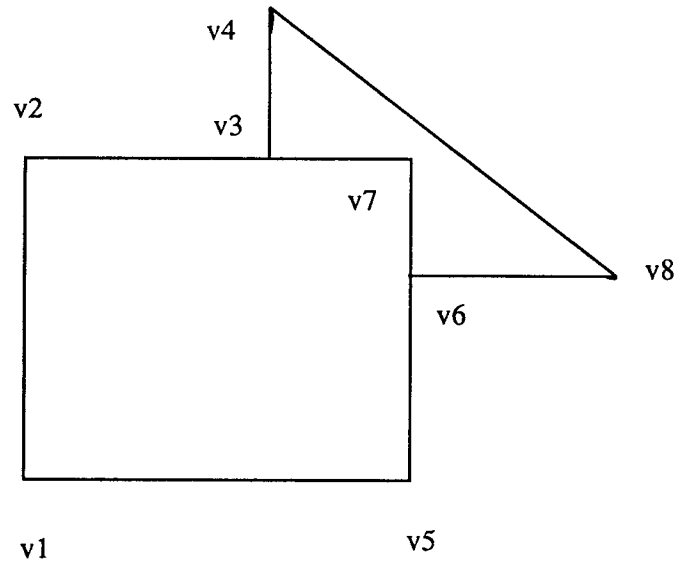
$$E^1 = \{((x_{i2}, y_{i2}), (x_{i1}, y_{i1})); ((x_{i1}, y_{i1}), (x_{i2}, y_{i2})) \in E\}$$

Then, hash each edge in $E \cup E^1$ using the X and Y coordinates of the first end-point as the key. This can be done for each edge in parallel. By this operation, we have all edges having the same first end-point in the same bucket. The elements which are in a bucket are the edges of the planar graph which have an end-point(vertex) equal to their key. We call the other end-points (vertices) of these edges a *row* and the common key vertex a *pivot*. The pivot is not included in the row. Thus the input planar graph can be visualized as a table made of a family of rows, each having a different pivot. Now, for each row in parallel, we sort the elements of the row about the pivot in an angular order. If the vertices are sorted in a clockwise order, then the final faces (except the infinite face) output by the algorithm will be in a counter-clockwise order and vice-versa. We call this final table of sorted elements the *Navigation Table*. The navigation is carried out on this table to obtain the regions (faces). The algorithm for navigating in parallel is presented in the next section. This procedure actually finds the faces. An example of an input graph, the navigation table and the faces is shown in figure 3.

4.3 NAVIGATION PROCEDURE

Assume that each processor has a unique id and all the elements of the rows have been marked as unused (mark = -1). The processor priority number is its id and larger priority number implies higher priority. The navigation procedure is executed by each processor in parallel with each processor handling a different pivot (and its row).

```
for each row of navigation table in parallel {

  current_pivot = first element of the row;
  for each element of the row {

    current_row = current_pivot;
    current_vertex = this element;
    face = {current_pivot}; /* 1st vertex of the
    face is the pivot */
    do forever {

      lock( current_vertex mark); /* Lock mark to
                          ensure it is updated by
                          only one processor */
      if my_id < mark(current_vertex) {

        break; /* this face is being traversed by
               a higher id processor so back off.
             Move to next element of the row. */

       } /* end if */
       else  mark current_vertex as used with
             my_id;

       unlock( current_vertex mark);
       append current_vertex to face;
```



1 Navigation Table:

v1: v5, v2
v2: v3, v1
v3: v7, v4, v2
v4: v8, v3
v5: v1, v6
v6: v8, v7, v5
v7: v3, v6
v8: v6, v4

2 The Faces Obtained:

f1:(v1, v5, v6, v8, v4, v3, v2)
f2:(v2, v3, v7, v6, v5, v1)
f3:(v3, v4, v8, v6, v7)

Figure 3: Example of a Navigation Table

```
        if current_vertex = current_pivot {

          output face and mark all vertices of
          the face with infinity; /* we have
                          found the face */
          goto next element of the row;

        } /* end if */

        previous_row = current_row;
        current_row = current_vertex;
        current_vertex = the element of
        current_row following the previous_row
                          with wrap around;

      } /* end do forever */

    } /* end for each element */

} /* end for each row in parallel */
```

5 ANALYSIS OF THE ALGORITHM

5.1 CORRECTNESS

The correctness of the algorithm given in section 3 is easy to see because it is a refinement of the following naive algorithm: Intersect all pairs of edges by pairwise comparison and divide the edges into segments. Each segment is fully visible or fully hidden. Compare each segment with all the faces to determine its visibility. Construct the visible regions from the visible segments and compare each region against all faces to see which face it corresponds to and shade it accordingly.

The correctness of the region reconstruction algorithm is also easy to see. The navigation algorithm is correct because we cover all pivots and each element of a row is considered exactly once by only one of the processors. All elements of the rows are considered because the processor assigned to a pivot begins the navigation for each element of the row. So all elements are traversed which gives all the faces.

5.2 COMPLEXITY

For the analysis, we assume a Concurrent-Read Exclusive-Write (CREW) PRAM model of computation. Let p be the number of processors. For the sake of the analysis, the input scene is assumed to consist of isothetic squares independently and identically distributed (i.i.d.) Each edge is of length l.

For computing the grid size, we use:

$$G = c\min(\frac{1}{l}, \sqrt{n})$$

Assume that:

$$G = \frac{c}{l}$$

Since the cell size is a constant of the face size as n increases, the probability p that a given face completely covers a face is:

$$p = (\frac{c-1}{c+1})^2, c \geq 1.$$

As the number of faces in the cell increases to infinity, the expected number of the first blocking face q is:

$$q = \sum_{i=1}^{\infty} ip(1-p)^{i-1} = \frac{1}{p} = (\frac{c+1}{c-1})^2$$

Therefore, the expected number of faces in a cell after the faces behind the blocking face is deleted (step 3 of the algorithm) is bounded by q. Let r be the expected number of cells in which a face falls. We have:

$$r = (c+1)^2.$$

Let

$$s = \frac{\text{number of faces}}{n}.$$

So, the number of faces per cell before step 3 is $\frac{rsn}{G^2}$ of which q faces per cell are not deleted. So the fraction of faces left after step 3 is:

$$\frac{qG^2}{rsn} = (\frac{c}{c-1})^2 \frac{1}{sl^2 n}$$

This is also the fraction of the edges that are in a cell after the cell grid is cast because the edges are also similarly distributed.

Let u be the average number cells in which an edge falls. We have:

$$u = c + 1.$$

The average number of edges per cell is $\frac{un}{G^2}$. So, the average number of edges per cell left after deletions is:

$$\frac{qG^2}{rsn} \cdot \frac{un}{G^2} = \frac{qu}{rs} = \frac{c+1}{(c-1)^2 s}$$

which is a constant. Therefore, intersecting the edges in a cell takes a constant time. It also implies that an edge is partitioned into a bounded number of segments.

Since the number of faces in a cell, the edges in a cell and the number of intersections in a cell are bounded by constants as shown above, we can deduce the following time complexities. Steps 1 and 2 of the algorithm take a time of $O(\frac{n}{p})$. Steps 2 and 3 take a time of $O(\frac{G^2}{p}) = O(\frac{n}{p})$. Step 4 will again take a time of $O(\frac{n}{p})$. Steps 5, 6 and 7 will take a time of $O(\frac{G^2}{p}) = O(\frac{n}{p})$. Therefore, the parallel hidden line removal algorithm takes a time of $O(\frac{n}{p})$ for n faces using p processors.

For the visible region reconstruction, assume that we have k visible segments. Obtaining the navigation table with the elements of a row unsorted will take a time of $O(\frac{k}{p})$ since hashing is used. Sorting the rows of the navigation table can at worst be $O(k \log k)$ since the maximum degree of the star vertex of a star graph (which is one of the worst-case inputs to the algorithm) can be $O(k)$. But, if such a case is detected and parallel sorting is used, this can be reduced to $O(\log k)$ using $O(k)$ processors. But, we assume a worse complexity of $O(k \log k)$ for this step. For the navigation algorithm, consider the (hypothetical) worst-case in which the conflicts are not detected and all the processors find all the faces around its assigned vertices. The complexity of the navigation step for a row is then $O(d_i f_{av})$, where f_{av} is the average size of a face. The worst-case for this step is a graph which is either a simple cycle or a star graph and for both cases the time complexity is $O(k)$. But in actual practice, we will do better than that because conflicts are detected in the algorithm and the average input is not likely to be the worst-case input. Then, step 9 of the algorithm takes $O(\frac{G^2}{p}) = O(\frac{n}{p})$.

So for the parallel hidden surface removal algorithm, the time complexity is of $O(\frac{n}{p} + k \log k)$ where n is the number of input edges, k is the number of visible segments and p is the number of processors assuming a CREW PRAM model of computation. Note that though the $k \log k$ term in the complexity suggests that the algorithm may not perform well, our implementation suggests otherwise. The reason for this is that the average data is not worst-case and secondly, the region reconstruction takes a small fraction (less than 10%) of the total time when p is equal to one. So, the algorithm is basically linear in the number of faces and the speedup is linear with the number of processors. The results are presented in the following section.

6 IMPLEMENTATION AND RESULTS

Practical implementation of parallel algorithms is extremely important since the theoretical performance does not take into account the limitations of resource contention and communication costs of real parallel machines. These factors can severely limit the speedups obtained. So, we implemented the algorithm on a Sequent Balance 21000 computer, which contains 16 National Semiconductor 32000 processors and compared the elapsed time when up to 15 processors were used to the time for only one processor. Since the Sequent Balance 21000 is a shared memory parallel computer, shared data structures are the communication mechanism for the processors. The synchronization of the processors is achieved by using atomic locks. The programming was done in C using the Sequent's parallel processing library routines for multitasking and synchronization.

For testing the implementation, we used the *tetrahedral pyramid* (figure 4) and the *meshed gears* (figure 5) databases from the "Standard Procedural Database" proposed by Eric Haines [12]. The results from processing the tetrahedral pyramid having 4096 faces (each of which is a triangle) is shown in figure 6. It is seen that an almost linear speedup is obtained by adding more processors. For 15 processors (which is the maximum that can be used), a speedup of 11 is obtained. The results for the meshed gears database which had 16 faces with 144 vertices and 1152 faces with 4 vertices is shown in figure 7. Here also, an almost linear speedup has been obtained and a speedup of 10 has been obtained for 15 processors. Note that though the speedup is linear, it is not the ideal speedup because of the overhead for forking the processes and locking of the data structures. The speedup is linear for the hidden line removal part but for the visible region reconstruction part, the speedup increases much slower. For 15 processors, the speedup obtained is about 6. But the time for region reconstruction is a small fraction, typically 3% to 10%, of the total time for the hidden surface removal. So, the overall speedup obtained is almost linear. This means that we should achieve an even bigger speedup on a machine with more processors. We are currently investigating the viability of the techniques used in this algorithm on SIMD (Connection Machine) and message-passing (Hypercube) architectures.

7 CONCLUSIONS

We have presented a new parallel object-space hidden surface removal algorithm. This algorithm is practical and has been successfully implemented on a commercial parallel machine. The uniform grid technique has been used to exploit parallelism in the geometric aspects of hidden surface removal. The uniform grid technique could be successfully used for parallelizing other geometric problems [3], [5]. A conflict-detection and back-off strategy has been introduced to achieve parallelism in the visible region reconstruction which is related to the topological aspects of the problem. This technique could be useful in parallelizing the topological aspects of other problems like polyhedron intersection and map overlay in cartography. These techniques lead to

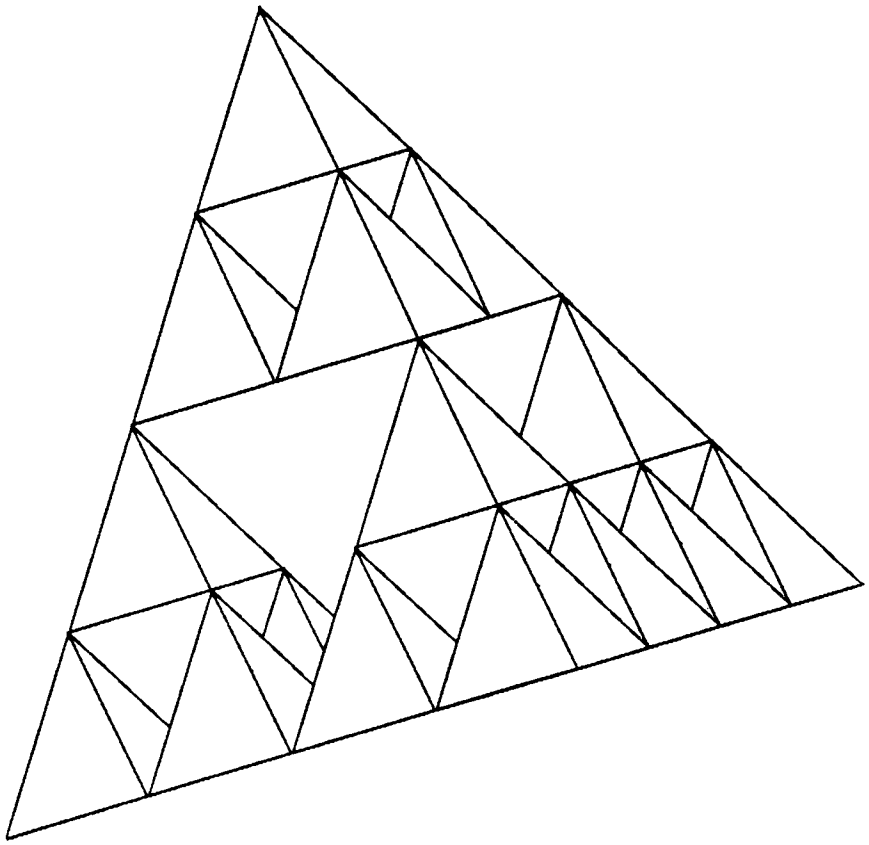


Figure 4: Tetrahedral Pyramid

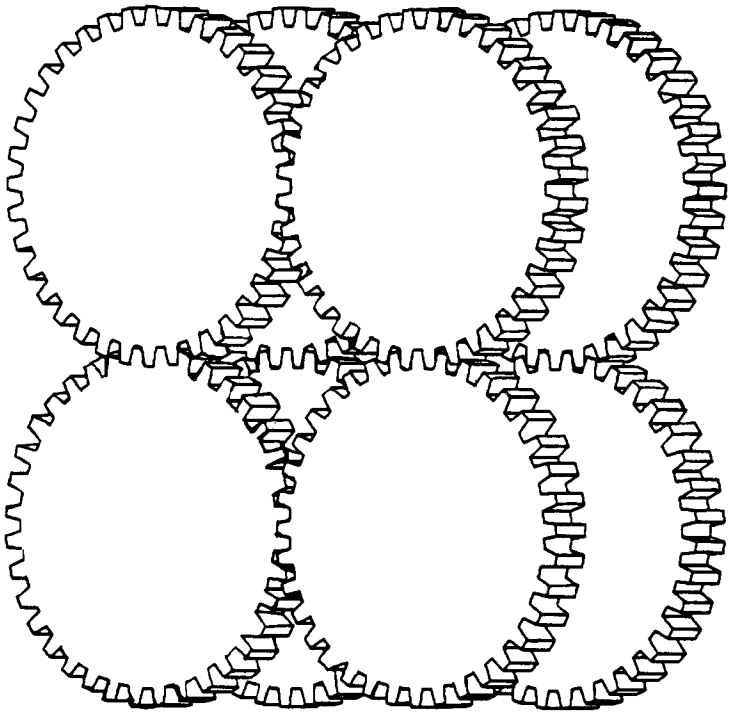


Figure 5: Meshed Gears

practical parallel algorithms which has been demonstrated by the implementation of the object-space hidden surface removal algorithm.

8 ACKNOWLEDGEMENTS

This work was supported by NSF Presidential Young Investigator grant CCR-8351942. Partial support for this work was provided by the Directorate for Computer and Information Science and Engineering, NSF Grant No. CDA-8805910. We also used equipment at the Computer Science Department and Rensselaer Design Research Center at RPI. 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 No. F306002-88-C-0031. Part of the research reported here was made possible through the support of the New Jersey Commission on Science and Technology and the Rutgers University CAIP Center's Industrial Members.

9 REFERENCES

1. Ajjanagadde V.G. and Patnaik L.M., "Design and Performance Evaluation of a Systolic Architecture for Hidden Surface Removal", *Computers & Graphics*, 12, 1 (1988), 71-74.

2. Akeley K. and Jermoluk T., "High-Performance Polygon Rendering", *Computer Graphics*, 22, 4 (August 1988), 239-246.

3. Akman V., Franklin W.R., Kankanhalli M. and Narayanaswami C., "Geometric Computing and Uniform Grid Technique", *Computer-Aided Design*, 21, 7 (September 1989), 410-420.

4. Catmull E., "An Analytic Visible Surface Algorithm for Independent Pixel Processing", *Computer Graphics*, 18, 3 (July 1984), 109-115.

5. Chandrasekhar N. and Franklin W.R., "An Efficient Parallel Algorithm for Determining Boolean Combinations of Complex Polyhedra" (in preparation).

6. Dippe M. and Swensen J., "An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis", *Computer Graphics*, 18, 3 (July 1984), 149-158.

7. Fiume E., Fournier A. and Rudolph L., "A Parallel Scan Conversion Algorithm with Anti-Aliasing for a General-Purpose Ultracomputer", *Computer Graphics*, 17, 3 (July 1983), 141-150.

8. Franklin W.R., "A Linear Time Exact Hidden Surface Algorithm", *Computer Graphics*, 14, 3 (1980), 117-123.

9. Franklin W.R. and Akman V., "Reconstructing Visible Regions from Visible Segments", *BIT*, 26, 1986, 430-441.

10. Fuchs H., "Distributing a Visible Surface Algorithm over Multiple Processors", *Proceedings of the ACM Annual Conference* (1977), pp. 449-451.

11. Fuchs H., Poulton J., Eyles J., Greer T., Goldfeather J., Ellsworth D., Molnar S., Turk G., Tebbs B. and Israel L., " Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories", *Computer Graphics*, 23, 3 (July 1989), 79-88.

12. Haines E., "A Proposal for Standard Graphics Environments", *IEEE Computer Graphics & Applications*, 7, 11 (November 1987), 3-5.

13. Hornung C., "A Method for Solving the Visibility Problem", *IEEE Computer Graphics & Applications*, 4, 7 (July 1984), 26-33.

14. Hu M. and Foley J.D., "Parallel Processing Approaches to Hidden Surface Removal in Image Space", *Computers & Graphics*, 9, 3 (1985), 303-317.

15. Joy K.I., Grant C.W., Max N.L. and Hatfield L., *Tutorial: Computer Graphics: Image Synthesis*, IEEE Computer Society Press, Washington D.C., 1988.

16. Kaplan M. and Greenberg D.P., "Parallel Processing Techniques for Hidden Surface Removal", *Computer Graphics*, 13, 1979, 300-309.

17. Mulmuley K., "On Obstructions In Relation To A Fixed Viewpoint", *Proc. 30th Annual Symposium on Foundations of Computer Science* (Oct. 30 - Nov. 1, 1989), pp. 592-597.

18. Overmars M. and Sharir M., "Output-Sensitive Hidden Surface Removal", *Proc. 30th Annual Symposium on Foundations of Computer Science* (Oct. 30 - Nov. 1, 1989), pp. 598-603.

19. Parke F., "Simulation and Expected Performance Analysis of Multiple Processor Z-Buffer Systems", *Computer Graphics*, 14, 3 (July 1980), 48-56.

20. Potmesil M. and Hoffert E.M., "The Pixel Machine: A Parallel Image Computer", *Computer Graphics*, 23, 3 (July 1989), 69-78.

21. Rankin J.R., "A Geometric Hidden Line Processing Algorithm", *Computers & Graphics*, 11, 1 (1987), 11-19.

22. Reif J.H. and Sen S., "An Efficient Output-Sensitive Hidden Surface Removal Algorithm and its Parallelization", *Proc. fourth Annual Symposium on Computational Geometry* (June 1988), pp. 193-200.

23. Sutherland I.E., Sproull R.F. and Schumacker R.A., "A Characterization of Ten Hidden Surface Algorithms", *ACM Computing Surveys*, 6, 1 (March 1974), 1-55.

24. Theoharis T., *Algorithms for Parallel Polygon Rendering*, Lecture Notes in Computer Science, Vol. 373, Springer-Verlag, Berlin, 1989.

25. Weinberg R., "Parallel Processing Image Synthesis and Anti-Aliasing", *Computer Graphics*, 15, 3 (Aug. 1981), 55-62.

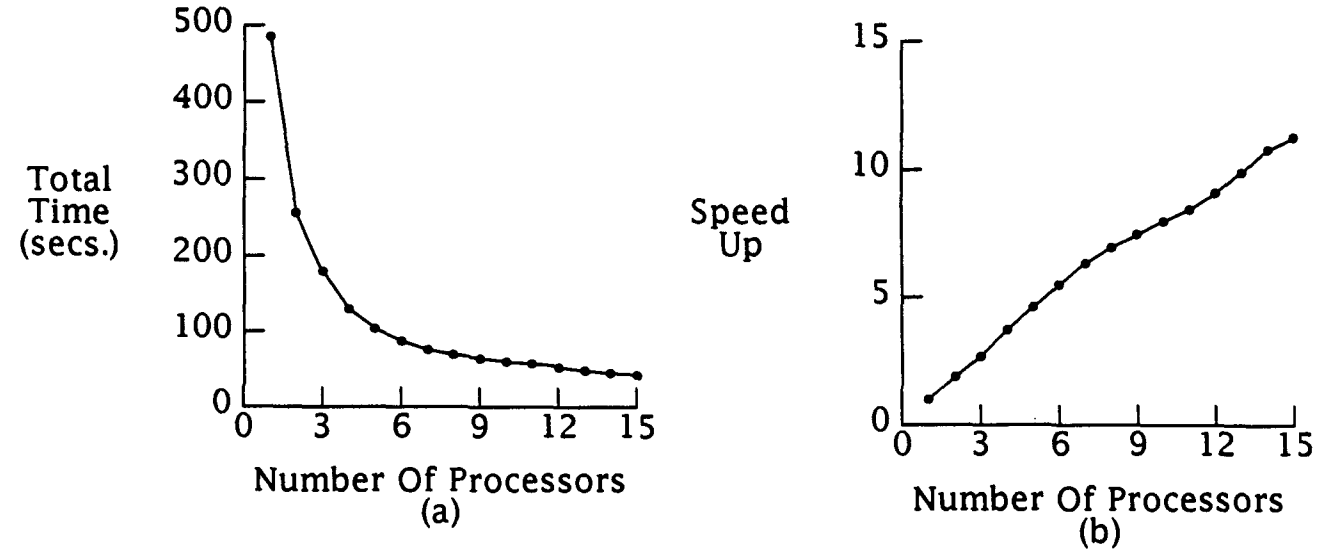


Figure 6: Timing for Tetrahedral Pyramid with grid size = 40.

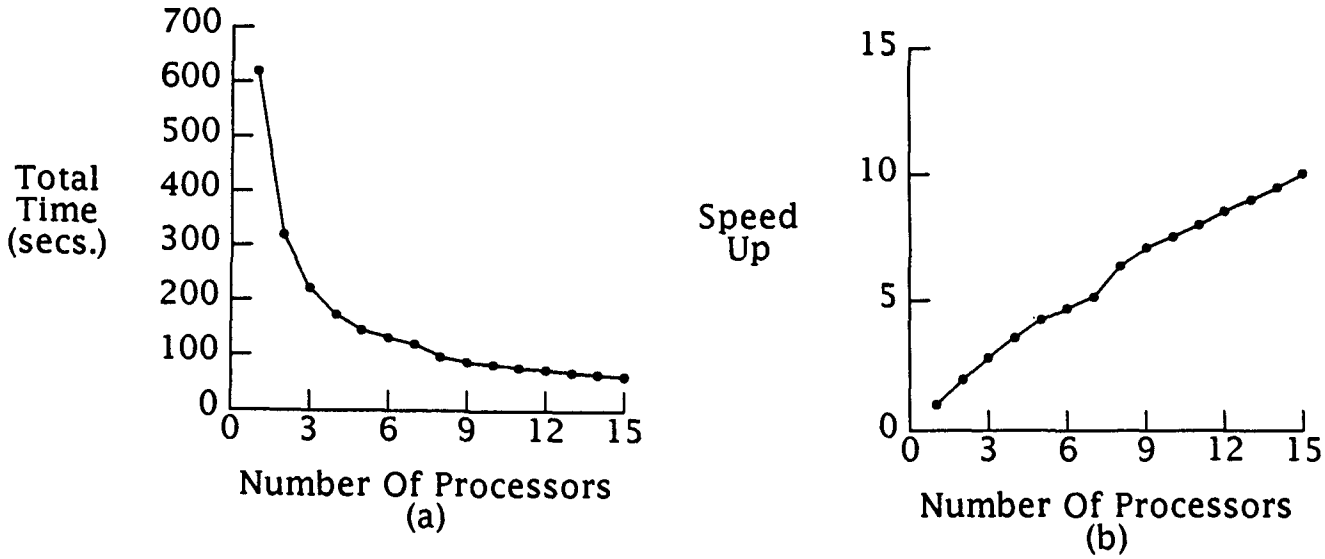


Figure 7: Timing for Meshed Gears with grid size = 32.