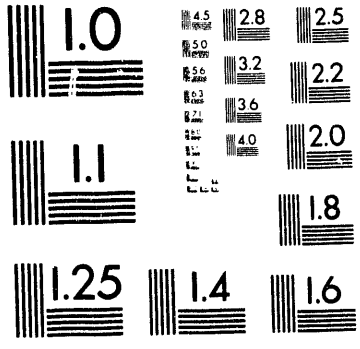

1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1

1 of 1

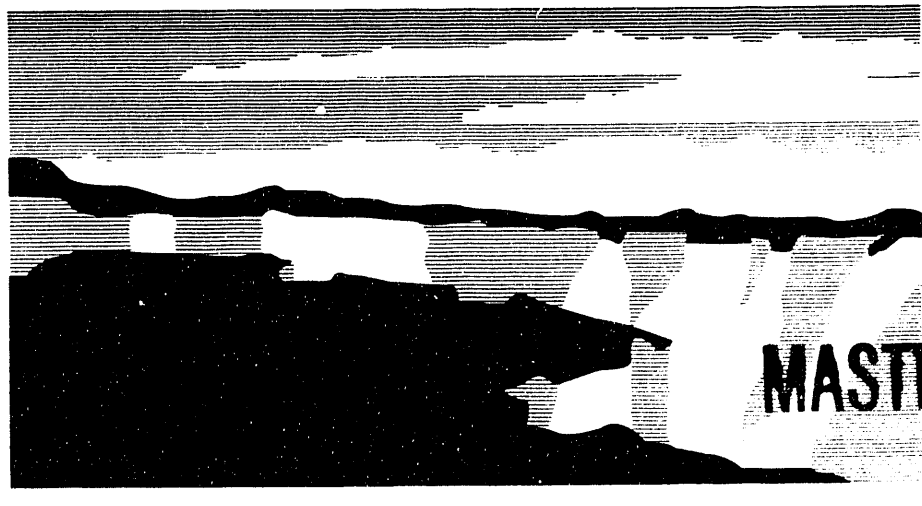
Conf-931115--11

LA-UR- 93-3173

Title: FAST DATA PARALLEL POLYGON RENDERING

Author(s): F. A. Ortega, and C. D. Hansen

Submitted to: Supercomputing '93
Portland, OR
November 18, 1993



Los Alamos
NATIONAL LABORATORY

Form No. 836 R5
ST 2629 10/91

Fast Data Parallel Polygon Rendering

Frank A. Ortega
X-Division Numerical Laboratory
Los Alamos National Laboratory
Los Alamos, New Mexico 87545
fao@lanl.gov

Charles D. Hansen
Advanced Computing Laboratory
Los Alamos National Laboratory
Los Alamos, New Mexico 87545
hansen@acl.lanl.gov

Abstract

This paper describes a parallel method for polygonal rendering on a massively parallel SIMD machine. This method, based on a simple shading model, is targeted for applications which require very fast polygon rendering for extremely large sets of polygons such as is found in many scientific visualization applications. The algorithms described in this paper are incorporated into a library of 3D graphics routines written for the Connection Machine. The routines are implemented on both the CM-200 and the CM-5. This library enables a scientist to display 3D shaded polygons directly from a parallel machine without the need to transmit huge amounts of data to a post-processing rendering system.

1 Introduction

In recent years, massively parallel processors (MPPs) have proven to be a valuable tool for performing scientific computation. The memory systems on this type of computer are far greater than those found on traditional vector supercomputers. As a result, scientists who utilize these MPPs can execute their three dimensional simulation models with a much finer grid resolution than previously possible. These extremely large grid sizes prove both to be a blessing and a curse. The finer grids allows for better simulation of the underlying physics. However, the finer grids also cause a data explosion when visualization and analysis are applied to them. A fully populated 64K CM-200 has 8 Gigabytes of available memory. A 1024 node CM-5 contains 32 Gigabytes of physical memory. While it is true that time-steps in current simulations don't utilize the entire memory systems of these machines, it is not uncommon for a dataset from a single time-step in a dynamic simulation to be in excess of several Gigabytes.

Geometry provides an excellent representation of simulations in the visualization process. Some scientific simulations contain explicit geometry. For example, material interface boundaries may be explicitly represented. For simulations which do not contain explicit geometry, there are a plethora of visualization techniques which generate geometry as an intermediate representation: isosurfaces, particles, spheres, vectors, icons, etc. In some cases such as sparse data sets, geometry extraction proves to be a compression technique without information loss. However, it is more typical for geometric extraction techniques to generate larger amounts of data than is present in the original data set [1].

One proven analytical technique for scientific visualization is the generation of isosurfaces. Two common methods for visualizing isosurfaces are volumetric rendering with a specific opacity map and the extraction of 3D contours [2, 3]. Volumetric techniques directly render the dataset into an image.

The rendering process, especially for large non-uniform datasets, can be quite time consuming [4]. This poses a problem if the viewing angle is not known *a priori* and needs to be determined interactively. Contouring techniques, on the other hand, extract a geometric representation of the specified contour(s). Typically, these are in the form of polygons. An advantage of this class of techniques is that view direction can be changed interactively by rendering the polygons on a workstation with dedicated hardware such as a Silicon Graphics Incorporated Onyx-RE2.

Recent research results have shown that 3D contours can be efficiently extracted at interactive rates from large dynamic datasets on massively parallel processors [1]. These techniques can generate over 1.24 million polygons for each time-step of a dynamic simulation with a grid size of only 256^3.[1] These large polygon sets impede interactive visualization on machines with dedicated graphics hardware in two ways: the amount of local-area network traffic and the total number of polygons throughput for the dedicated graphics hardware. In the simplest case (disregarding color information), each vertex of each polygon consists of 3 floating point locations and 3 floating point normal positions or 24 bytes/vertex. With 250000 polygons (assuming triangles), this is 18Mbytes per time-step. On an ethernet, best-case is 14.4 seconds and average-case is much worst. Obviously, the amount of data overloads the network capacity. Additionally, 250000 polygons is at the limit of disjoint polygons/second which state of the art graphics engines can render.

One solution would be to move the entire dataset over to the workstation and generate the isosurfaces locally utilizing optimization techniques such as spatial decomposition. Another similar solution would be to analyze the raw data set with a commercially available visualization tool such as AVS, Explorer, etc. In addition to network transport issues, the problem with both of these solutions is that the size of the dataset overwhelms the workstation. The raw data can be over 128Mbytes per time step. In addition to the network problems previously mentioned, our experience is that data sets of this magnitude cause the workstation to page excessively with memory page faults when generating the isosurface. While it is true that environments such as AVS are being ported to MPPs, it has been our experience that the geometry component of these environments is not supported on the MPP but is still utilized on the workstation. The necessitates the transport of data, albeit filtered data, to the workstation.

A more appropriate solution would be to render the geometry on the MPP where the data already exists. This is completely compatible with the previously described MPP isosurfacing techniques and utilizes the power of the massively parallel computer to generate an image. As we will show, image generation time is much less than direct volume rendering and network traffic is limited to the image size rather than the data size. Additionally, this model extends the usefulness of visualization environments such as AVS or Explorer since images are transferred to a workstation rather than full, or reduced, datasets which still require further processing. Another benefit of this approach is the capability of directly calling rendering functions from the running computational model. This allows not only for simulation monitoring/steering but also has proven to be an extremely useful tool in the debugging process.

2 Related Work

An increasing number of parallel polygonal rendering algorithms have been developed. The main strategy has been to perform a parallelization in two stages: scan conversion and rasterization. This strategy follows the gross functionality which is implemented in most hardware graphics pipelines. The standard

[1] Currently, this is not considered a large dataset. Typically simulations on an MPP machine use grid resolution upwards of 512^3.

Figure 1: Standard Graphics Pipeline

graphics pipeline is shown in figure 1. Polygons are transformed from world space to screen space, clipping to the screen is performed, polygons are scan converted, hidden surface elimination is performed. Lighting can be applied either to the vertices before scan conversion, Gouraud shading, or at each pixel, Phong shading.

Scott Whitman approached the problem of polygonal rendering on a shared memory massively parallel processor; the BBN TC-2000 [5]. He splits the standard graphics pipeline into three stages: front-end, rasterization, and back-end. In the front-end stage, the polygons are read into the system, transformed, back-face culled, clipped to the screen, and stored in shared memory. When the polygons are stored in shared memory, a bucketization is performed to determine in which tiles the polygon lies. The bounding box is used as the determinate. The rasterization phase is similar to the standard graphics pipeline. Polygons are scan converted, shaded, z-buffered, and finished scan lines are stored in a virtual frame buffer. The back-end writes the virtual frame buffer to the actual frame buffer. Whitman explores many different tiling schemes but the basic parallel rendering follows the steps outlined.

Thomas Crockett and Tobias Orloff implemented a standard scan-line conversion algorithm on a distributed memory system, INTEL iPSC/860, using message passing [6]. They approached the problem by combining the first three stages of the standard graphics pipeline and then splitting the render process into two distinct steps: splitting polygons into trapezoids and rasterization of transformed trapezoids. They evenly distributed the polygons to all processors and also even divided the image space into equally sized horizontal strips. Thus, their algorithm achieves both object and pixel parallelism. The algorithm alternates between splitting triangles into trapezoids and rasterization.

Crow, Demos, Hardy, McLaughin and Sims utilized a connection machine, CM-2, to develop and implement a photorealistic renderer [7]. Their goal was to provide very high quality rendering for one of the film production houses in Hollywood. Since they were working on a SIMD machine, they chose to program their algorithm in a data parallel manner. They split the traditional polygonal rendering pipeline into fours stages: transforms, clipping, scan conversion and shading. Rather than following earlier approachs on the CM-2, they assigned each vertices to a processor for applying transforms. The transformation matricies were passed one after the other and applied to the vertexes. Clipping was done by assigning one polygon to each processor. Scan conversion was similarly handled. Shading was handled by assigning a pixel to each processor. Their algorithm was effective for hundreds of thousands of polygons with complex photo realistic rendering features.

3 Parallel Progamming Models

Architecturally, MPPs fall into two classes of machines: MIMD and SIMD/SPMD. Similarly, there are two programming models for parallelism which are exploited in massively parallel systems: execution parallelism and data parallelism.

The execution parallel model divides a task up into a number of subtasks that can run concurrently. For example, one could assign each object to be rendered to a processor. Each of these tasks can execute concurrently and independently. It is not necessary for the tasks to execute in lockstep. Typically, there is some synchronization barrier which synchronizes these independent tasks for the next step in the process such as hidden-surface removal. Since each of the tasks execute independently, this model maps well onto MIMD architectures such as the CM-5 from Thinking Machines, Inc. [8] and the Paragon XP/S from Intel Supercomputers, Inc. [9].

In the data parallel model the same operation is performed on all the selected data elements. For example given an array of numbers, a constant could be added to each number. In the data parallel model this operation would logically occur simultaneously on each element of the array. Actual hardware may or may not perform this operation simultaneously on all selected data elements. This would depend on whether or not enough physical processors exist for each data element. In the case where there is more data than processors, each physical processor is given a number of data elements upon which it operates. Each of these data elements is considered a virtual processor. The concept of a virtual processor allows one to treat the program as if there were a physical processor per data item. This model maps well onto SIMD/SPMD architectures such as the CM-200 and CM-5[2] from Thinking Machines, Inc.[10], the MP1 from MasPar Computer Corp. [11], and the Pixar Image Computer [12].

It has been recognized that for some problems the data parallel model is somewhat easier to program than the execution parallel model. Modern scientific compilers, such as Fortran 90 and C*, take advantage of the data parallel model in their language constructs. Furthermore, it has been shown that data parallel programs can achieve good speed up on MIMD architectures [13].

4 A Data Parallel Renderer

The choice to employ the data parallel programming model for the polygonal rendering algorithm was made for several reasons. Our computing facilities include several Connection Machines from Thinking Machines. While the CM-5 at our site is a MIMD MPP, the multiple CM-200s are strictly SIMD MPPs. It was our desire to develop an algorithm which would be portable to both machines. Secondly, the scientific applications which run on these MPPs utilize CMFortran as their primary language. Ease of use was a high priority so we wanted to develop an algorithm suite which would integrate well with existing applications. Lastly, we wanted a rendering environment which would take advantage of the CM/AVS visualization software on the CM-5 yet perform rendering in near real-time. Since the current version of CM/AVS relies on the workstation to perform rendering, performance suffers for large data sets due to the poor response time seen with network traffic.

The idea behind the data parallel renderer is to maximize the number of operations occurring in parallel while minimizing communication. To accomplish this, it is advantageous to be careful about data layout.

The algorithm employs the standard graphics pipeline previously described with some modifications. The clipping stage is postponed until later in the pipeline. The basic steps are as follows:

[2]The CM-5 can actually be programmed with either model. The Run Time System has support which causes the machine to run as a SPMD MPP.

1. transform the polygons according to interactive controls (rotations, translation and scaling)
2. transform the polygons from world space to screen space
3. shade the vertices
4. scan convert the polygons
5. clip against the viewport
6. perform hidden surface elimination

The first three steps operate on vertices while the fourth step operates on polygons and then scanlines. The last two steps operate on pixels. In order to maximize operations occurring in parallel, we want to layout the vertex and polygon information in a data parallel manner. Following that, the scanlines should be laid out similarly. Finally, we need to operate on the pixels in parallel.

To accomplish the first steps, the polygon vertex data is stored in three arrays: X, Y, Z components. This data, for all polygons, is spread across available processors such that the vertices for a single polygon lie within a processor. The color data for a polygon also lies within the same processor. When the transformations are performed, the vertex data for all the polygons is transformed simultaneously.

Next, shading is performed for each polygon. In this implementation, we are optimizing for speed. Therefore, we perform simple Gouraud shading. More advanced shading techniques would be easy to implement. Again, since the polygon vertices are data parallel, the shading for the polygons are performed simultaneously.

To save time and maximize the parallelism across polygons, a modified scan line conversion algorithm was used [14]. This algorithm assumes that the polygonal set consists of large numbers of small polygons. We have found this is a valid assumption since the target application of this renderer is scientific data particularly data derived from very large computational models. Typical polygonal set sizes can range from 100,000 to millions of polygons [1]. This algorithm takes advantage of the fact that each polygon has relatively few scan lines passing through it compared with the number of scan lines in the image.

This step iterates over the maximum number of scan lines through any polygon. Since scan conversion is concurrently executed for all polygons, the largest number of scan lines necessary to process the entire set of polygons is maximum number of scan lines spanning any polygon. This is, of course, the polygon with the maximum image-space height in Y. At the initiation of this step, the first scan line within every polygon is processed simultaneously. As the number of scan lines processed approaches the maximum, fewer polygons will be processed since some polygons will have fewer scan lines passing through them than others.

We start the scan-conversion process by finding any intersections that the scan line makes with each of the polygon sides. There must be at least two intersections but, depending on the polygon shape, there may be many intersections. In order to process a general polygon shape, the polygon scan line intersections are sorted in ascending x order and grouped into line end pairs. The even-odd rule is then used to select the segments that are inside the polygon[15]. At this stage, the end points and color data for the segments are gathered into a data structure such that the start and end points are spread across the processors in a data parallel manner. This utilizes generalized router communication. A buffer is formed of the segment end-points. The buffer holds the segments until a sufficiently large number is gathered. The buffering of the segments is performed in order to maximize the number of segments that will be broken down into individual pixels. Recall that as the algorithm progresses there are fewer polygons that contain a scan line thus it is possible that fewer segments will be generated each pass.

At this point, each processor contains a start and end point for a segment. The pixels will also be contained in a data parallel array which spreads the pixels across all processors. Processing the lines to pixels in parallel requires a loop over the number of pixels in the x direction of the longest line. The first pixels from all the lines are processed, then the second, etc. The x, y and z values for the pixels are interpolated from the line ends. Any pixels that lie outside the viewport are clipped. Since the polygon scan lines are processed in parallel, there is a good possibility that many of the segments will generate pixels with the same image location. The Connection Machine can not handle these "collisions" correctly with standard interprocessor communications. The generalized data router must be used in conjunction with a sendmax combiner [16]. This utility uses the router for fast communications but it presented another problem: the utility can only work with one sending array at a time, and the color value for the pixel needs to be stored in the image array for the pixel which has the maximum z value. This problem was solved by combining the z values and the color value into a double precision array before the zbuffer compare. These are the values saved into the zbuffer. Thus, after all the polygons have been processed, the image data must be extracted from the zbuffer data before displaying the image.

5 Graphics Library Features

The algorithm and its components described in this paper have been built into a 3D graphics library which can be linked with any program. The library can process arbitrarily complex polygons. These polygons can be rasterized into flat shaded images or Gouraud shaded images. More advanced lighting models would be simple to add to the rendering routines. In addition, the polygons can be processed into wireframe or wireframe with hidden line removal images. Unshaded color contour images of polygon vertex data can also be created. And finally, z-buffered points and lines can be generated.

The routines in the 3D graphics library are written in CMFortran and only contain the rendering process up until display of the resulting frame. Thus, both the CM-200 and the CM-5 can compile the same source code. At the time the library was written, C* was not yet available on the CM-5. On the CM-200, C* uses a transposed layout, called fieldwise. Since most of the scientific simulations that use the 3D graphics library routines are written in CMFortran, the routines enable the data to stay in the slicewise data model on the CM-200. For the CM-200, this is a nice performance feature since transposing from slicewise to fieldwise, or visa-versa, can be time consuming and adversely effects performance. This is not a problem on the CM-5.

There is a utility routine in the library that extracts an image array from the zbuffer array. The array is a CM integer array with the first element representing a color value for the top left pixel in the window. This format is consistent with X11R4 images and can then be displayed with available display software such as *Raster or CMX11.

The shading algorithm in the library supports 15 shades of up to 15 colors. The color model is designed for eight color planes or the PseudoColor video class[17]. The library expects indices 30 through 255 of the color table to provide the correct shading for 15 colors. There is a utility routine that will generate the correct color table from 15 user defined colors. The lighting model is quite simple using parallel white rays normal to the window.

6 Experiments

Several experiments were run on both the CM-200 and the CM-5. All timings were for an image size of 512×512, a smaller image size would speed up the rendering and a larger size would slow down the

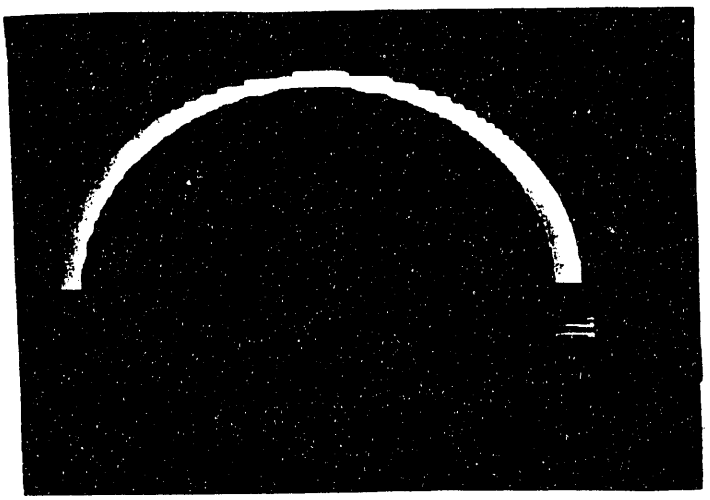


Figure 2: Image of Oil Well Perforator with 355,948 Small Polygons

rendering. Table 1 shows the times for rendering a data set with small polygons on the CM-200. Table 4 shows the times for rendering the same data set on the CM-5. Table 2 shows the times for rendering a data set with large polygons while Table 3 shows the times for rendering the same data set on the CM-5. The data set with small polygons fits our assumption that geometry generated from scientific data on massively parallel computers will typically be composed of many small polygons (polygons which have few scanlines passing through them). The data set with large polygons violates this assumption and the times are given to show the effect. On the CM-200, experiments were run with partition sizes of 16K, 32K and 64K. On the CM-5, experiments were run with partition sizes of 32, 64, 128, 256, and 512 nodes. Figure 2 shows the results of rendering a data set generated from a hydro-dynamics code running on the CM-200 and the CM-5.

As can be seen, the times shown for the data set containing large polygons are an order of magnitude slower on both the CM-200 and the CM-5. This is because there were polygons which covered over 3/4 of the image thereby lengthening the serial portion of the algorithm. As the size of the polygons increases, the polygon rasterizing speed decreases due to the iterative loops over the maximum polygon height and the maximum scan line size. However, since this was shown as a degenerate case, we will focus on the times for the small polygons. On a 64K CM-200 partition, rendering speeds of over 600,000 polygons per second were achieved. These are disjoint polygons and we have found this to be three times the speed of our SGI 380/VGX whose published rendering times are over one million meshed polygons per second. However, the rendering speed is dramatically reduced when the polygons to be rendered are not in cache and the polygons are disjoint rather than meshed. On a 512 partition of the CM-5, rendering speeds of close to one million polygons per second were recorded. This exceeds the speed of current state of the art dedicated graphics hardware.

Times			Polygons/second		
16K	32K	64K	16K	32K	64K
1.894	1.04	0.571	120,532	219,507	399,803
1.637	0.848	0.482	139,455	269,207	473,626
1.162	0.625	0.335	196,461	365260	681,456

Table 1: Rendering of Small Polygons on CM-200

Times			Polygons/second		
16K	32K	64K	16K	32K	64K
22.094	15.005	8.574	9,866	14,527	25,423
18.015	11.855	6.834	12,100	18,387	31,896
10.548	6.789	4.039	20,665	32,108	53,969

Table 2: Rendering of Large Polygons on CM-200

Times					Polygons/second				
32	64	128	256	512	32	64	128	256	512
5.756	4.754	1.482	0.796	0.463	39,660	48,020	154,040	286,794	493,062
3.402	2.877	0.966	0.498	0.302	67,104	79,349	236,322	458,409	755,920
2.476	1.048	0.550	0.401	0.236	92,200	217,832	415,069	569,296	967,322

Table 3: Rendering of Small Polygons on CM-5

Times					Polygons/second				
32	64	128	256	512	32	64	128	256	512
37,326	20,169	12,488	8,466	6,220	5,939	10,807	17,455	25,746	35,045
30,684	16,034	9,758	6,393	4,717	7,104	13,595	22,338	34,097	46,212
17,261	9,353	5,543	3,594	2,590	12,628	23,306	39,325	60,651	84,163

Table 4: Rendering of Large Polygons on CM-5

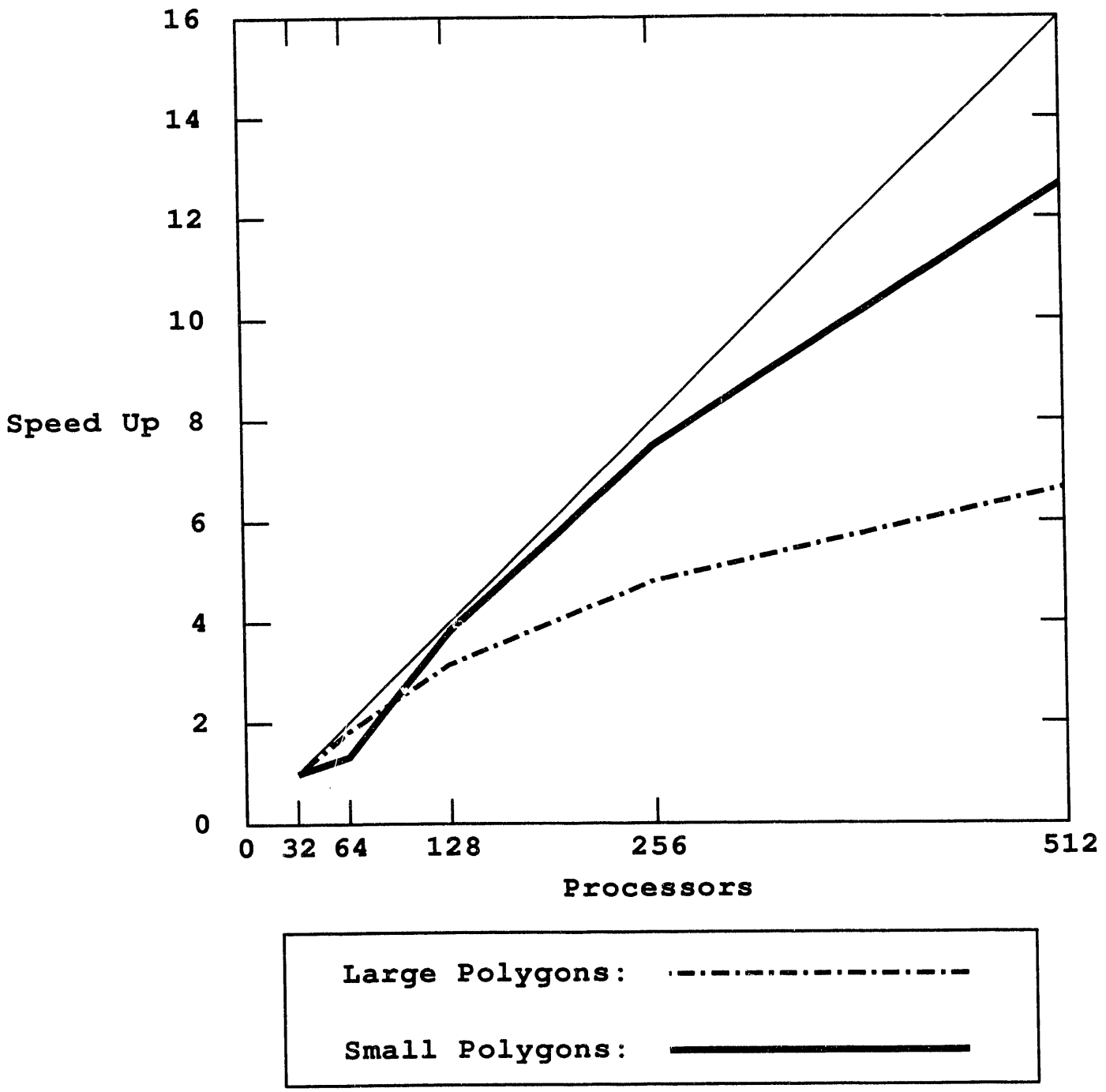


Figure 3: CM-5 speed up

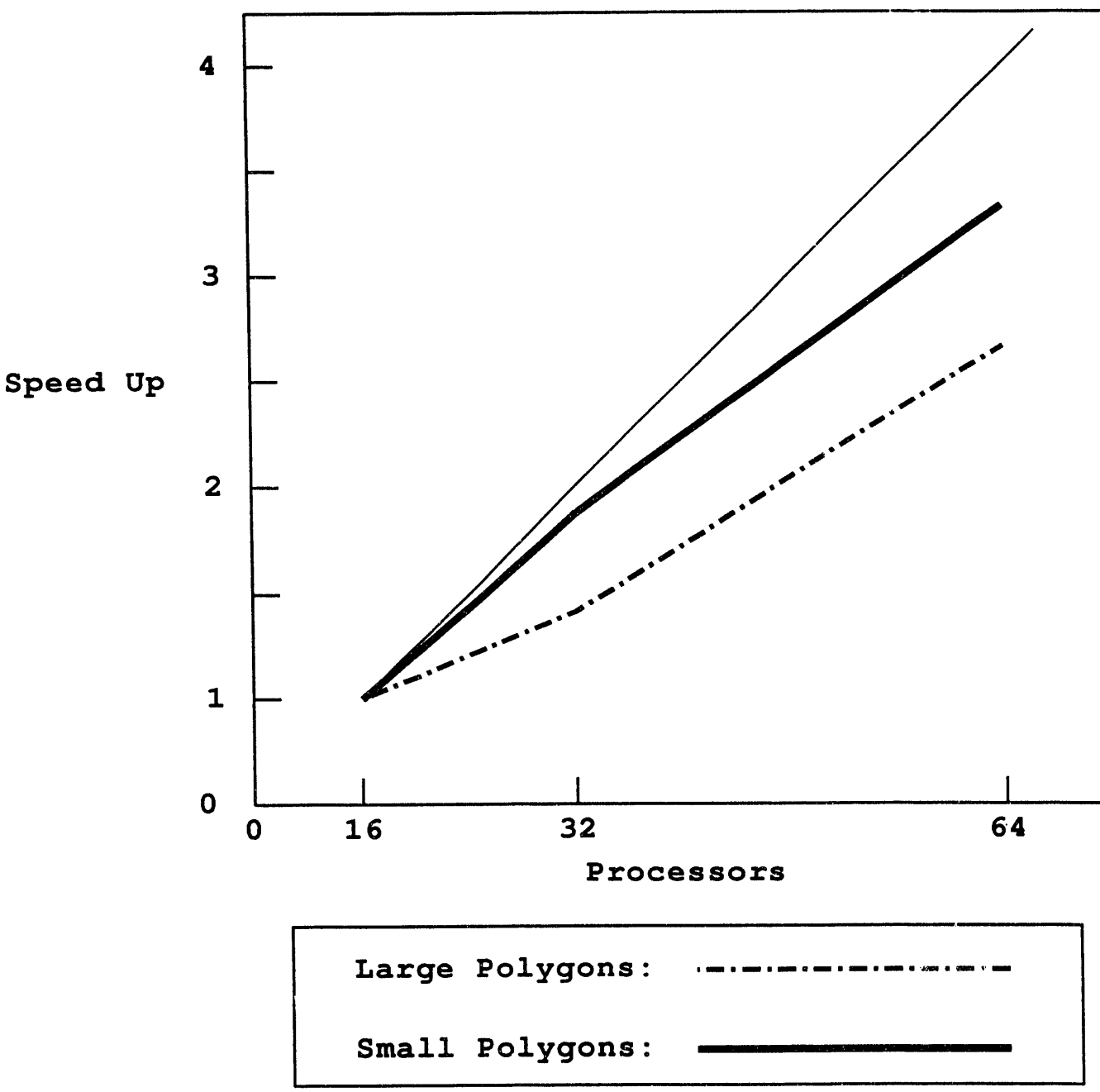


Figure 4: CM-200 speed up

Figure 3 shows the speedup of the algorithm run on the CM-5 while Figure 4 shows the speedup of the algorithm run on the CM-200. The speedup curves are relative to the implementation on the smallest partition available for each of the MPPs. While neither of the speedups were linear, the graphs show that near linear speedup for the data set with small polygons was achieved. The data set with large polygons exhibited worst speedup due to the iterations serially performed.

7 Conclusions

This paper described a parallel method of polygon scan conversion allows the visualization of large 3D simulations directly from the SIMD computer. This allows scientists to evaluate the simulation as it is running or shortly thereafter without the need to transfer huge amounts of data from a massively parallel computer to a graphics workstation. Performance data was provided that showed the method can out perform high-end commercially available graphics workstations although it requires tremendous resources.

8 Acknowledgements

We wish to thank John Fowler of Los Alamos National Laboratory for encouragement and assistance with this paper. We would also like to thank Harold Trease of Los Alamos National Laboratory for help with some CMFortran issues. The Advanced Computing Laboratory (ACL) at Los Alamos National Laboratory generously provided computing iron and tremendous assistance with timing runs. David Rich of the ACL was particularly stellar with the generosity of his time.

References

[1] C. Hansen and P. Hinker. Massively parallel isosurface extraction. In *Proceedings of Visualization '92*, pages 77–83, 1992.

[2] Mark Levoy. Efficient ray tracing of volume data. *ACM Transactions of Computer Graphics*, 9(3), July 1990.

[3] W. Lorensen and H Cline. A high resolution 3d surface contruction algorithm. In *Computer Graphics*, volume 21, pages 163–169, 1987.

[4] Todd Elvins. A survey of algorithms for volume visualization. *Computer Graphics Quarterly*, 26(3), August 1992.

[5] Scott Whitman. *Multiprocessor Methods for Computer Graphics Rendering.* Jones and Bartlett, Boston, 1992.

[6] T. Crockett and T. Orloff. A parallel rendering algorithm for mimd architectures. Technical Report 91-3, NASA Langley Research Center, 1991.

[7] F. Crow et al. 3d image synthesis on the connection machine. In *SIGGRAPH Course Notes: Parallel Processing and Advanced Architectures in Computer Graphics*, pages 107–128, 1989.

[8] Thinking Machines Corporation. The connection machine CM-5 technical summary, 1991.

[9] Intel Corporation. Paragon XP/S product overview, 1991.

[10] Thinking Machines Corporation. Connection machine CM-200 series technical summary, 1991.

[11] MasPar Computer Corporation. MP-1 family data-parallel computers, 1990.

[12] A. Levinthal and T. Porter. Chap - a SIMD graphics processor. *Computer Graphics*, 18(3), July 1984.

[13] P. Hatcher et al. Architecture independent scientific programming in dataparllel c: Three case studies. In *Proceedings of Supercomputing '91*, pages 208–217, 1991.

[14] John D. Fowler. A reduced set scan line algorithm. internal document, Los Alamos National Laboratroy.

[15] Oliver Jones. *Introduction to the X Window System.* Prentis Hall, Englewood Cliffs, New Jersey, 1989.

[16] Thinking Machines Corporation. Cm fortran user's guide, 1991.

[17] Adrian Nye. *Xlib Programming Manual.* O'Reilly and Associates, Inc., Sebastapol, California, 1990.

DATE
FILMED
10/20/93
END