Fast Data Parallel Polygon Rendering

Frank A. Ortega

X-Division Numerical Laboratory
Los Alamos National Laboratory
Los Alamos, New Mexico 87545

Charles D. Hansen James P. Ahrens*

Advanced Computing Laboratory
Los Alamos National Laboratory
Los Alamos, New Mexico 87545

Abstract

This paper describes a data parallel method for polygon rendering on a massively parallel machine. This method, based on a simple shading model, is targeted for applications which require very fast rendering for extremely large sets of polygons. Such sets are found in many scientific visualization applications. The renderer can handle arbitrarily complex polygons which need not be meshed. Issues involving load balancing are addressed and a data parallel load balancing algorithm is presented. The rendering and load balancing algorithms are implemented on both the CM-200 and the CM-5. Experimental results are presented. This rendering toolkit enables a scientist to display 3D shaded polygons directly from a parallel machine avoiding the transmission of 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 to be both a blessing and a curse. The finer grids allow 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 physical 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 data set 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[13].

*Author's current address is Department of Computer Science, FR-35, University of Washington, Seattle, WA 98195

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[17, 18]. Volumetric techniques directly render the data set into an image. The rendering process, especially for large non-uniform data sets, can be quite time consuming[7]. 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 the view direction can be changed interactively by rendering the polygons at high frame rates.

Recent research results have shown that 3D contours can be efficiently extracted at interactive rates from large dynamic data sets on massively parallel processors[13]. 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 250,000 polygons (assuming triangles), this is 18Mbytes per time-step. To transfer this size of data on an ethernet, the best-case transfer time is 14.4 seconds and the average-case transfer time is much worse. Obviously, the amount of data overloads the network capacity. Additionally, 250,000 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 data set 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 data set overwhelms the workstation. The size of 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. The goal of this research is to develop a rendering algorithm which is efficient for large numbers of polygons.[2] For such large polygon sets, we make the assumption that most of the polygons will cover a relatively small number of pixels since scientific data sets tend to produce such polygons. 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, data sets 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.

In the next section, previous work in this field will be reviewed followed by a description of the data

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

[2] We consider 250K and upwards a large polygon set.

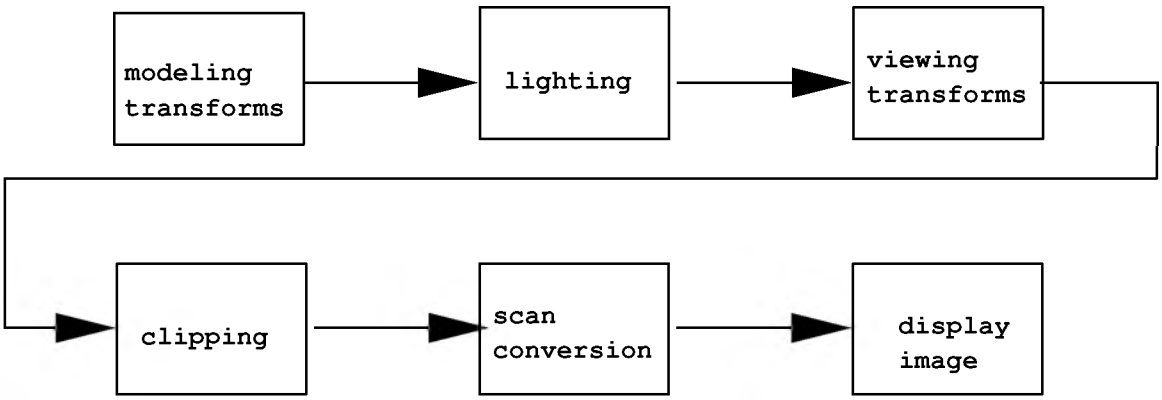

Figure 1: Standard Graphics Pipeline

parallel rendering algorithm. Load balancing is a critical issue which will be addressed. Results from experiments on both a CM-200 and a CM-5 will be presented.

2 Related Work

An increasing number of parallel polygonal rendering algorithms have been developed. The main strategy has been to perform parallelization in two stages: scan conversion and rasterization. This strategy follows the gross functionality which is implemented in most hardware graphics pipelines. The standard 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, and 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 investigated the problem of polygonal rendering on a shared memory massively parallel processor, the BBN TC-2000[20]. He split 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 image tiles the polygon lies. The bounding box is used as the determinant. 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 message passing, the INTEL iPSC/860[6]. They approached the problem by combining the first three stages of the standard graphics pipeline and splitting the rendering process into two distinct steps: splitting polygons into trapezoids and rasterizing the transformed trapezoids. They evenly distributed the polygons to all processors and also divided the image space into equally sized horizontal strips. Thus, their algorithm achieves both object space and image space parallelism. The algorithm alternates between splitting triangles into trapezoids and rasterization.

Fiume, Fournier, and Rudolph presented a spanning scan line algorithm which ran on a simulator for an ultracomputer[8]. They proposed language extensions to implement a Z-buffer. The algorithm assigned each PE a span consisting of a trapezoidal or triangular regions. The PEs synchronized at the end of processing for each scan line. This led to a load balancing problem. The authors suggest

splitting the scan line if it is greater than some M number of pixels.

Crow, Demos, Hardy, McLaughin, and Sims utilized a Connection Machine, CM-2, to develop and implement a photorealistic renderer[9]. 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 four stages: transforms, clipping, scan conversion and shading. Rather than following earlier approaches on the CM-2, they assigned each of the vertices to a processor for transformation. The transformation matrices were passed one after the other and applied to the vertices. Clipping was done by assigning one polygon to each processor. Scan conversion was handled similarly. Shading was handled by assigning a processor to each pixel. Their algorithm was effective for hundreds of thousands of polygons with complex photo realistic rendering features although no timings were presented.

Schroder and Drucker describe a data parallel ray-tracing algorithm for heterogeneous databases[19]. Although ray-tracing is a different problem than polygon rendering by scan coversion, we include this related work since it provides an alternative to rendering on a massively parallel processor. Their algorithm addresses the load balancing problem by continually remapping available resources. They start by allocating one processor per ray. Each ray processor allocates the number of processors required to test insections with objects. As rays travel down through the ray-tree, processors sets are continually allocated. The time to render scenes containing small number of objects (7381 spheres and one polygon), varied from 98.3 to 118.7 sec on a 32K CM-2. Although the algorithm produces ray-traced images, the time is much slower than required. As pointed out, we are interested in rendering upwards from 250K polygons.

3 Parallel Progamming Models

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

The control parallel model divides a task up into a number of subtasks that can run concurrently. For example, one could assign each stage of the rendering pipeline 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. Since each of the tasks execute independently, this model maps well onto MIMD architectures such as the CM-5 from Thinking Machines, Inc. [5] and the Paragon XP/S from the Scientific Supercomputer Division of Intel, Inc. [1].

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 [3] from Thinking Machines, Inc.[4], the MP1 from MasPar Computer Corp. [2], and the Pixar Image Computer [16].

[3]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.

It has been recognized that for some problems the data parallel model is somewhat easier to program than the control 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 speedup on MIMD architectures [10].

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 Corporation. While the CM-5 at our site is a MIMD MPP, the 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 (the AVS kernel) to perform rendering, performance suffers for large data sets due to the poor response time seen with network traffic. Additionally, CM/AVS currently is restricted to the data parallel programming model on the CM-5.

The basic idea behind the data parallel renderer is to maximize the number of operations occurring in parallel while minimizing communication. While this trait is desirable in both data parallel and task parallel programming models, the SIMD/SPMD nature of data parallel programs imposes additional constraints. In data parallel programs, there is only one thread of control. For efficient programs, it is necessary to maximize the set of active processors at any given step in an algorithm. This is accomplished by judicious assignment of data to the processors, sometimes referred to as *layout*. To determine the optimal layout for the rendering process, let us examine the previously described standard graphics pipeline with respect to data operations. The basic steps are as follows:

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

The first step is to transform the polygons by applying desired rotations, translations and scalings. This is accomplished by building a 4 x 4 homogeneous affine transformation matrix constructed from the individual transformations. This transformation matrix is applied to each vertex resulting in transformed polygons as desired. The key point is that all operations are performed on the vertices.

The next step transforms the polygons from world space to screen space by applying a perspective or orthogonal transformation to each of the vertices. This step can be combined with the previous step by correctly concatenating the perspective (or orthogonal) transformation matrix with the transformation matrix representing the desired rotations, translations and scalings. All operations are performed on the vertices.

We could perform back-face culling at this point. However, this introduces a conditional into the data parallel execution stream and as long as there are many more polygons than processors, it is no less efficient to render the back-facing polygons even though they'll most likely be occluded by some other object. The implementation of the renderer includes back-facing culling.

Next, shading is performed for each polygon. In this implementation, we are optimizing for speed. Therefore, we perform simple Gouraud shading. In Gouraud shading, the shading is computed at each vertex and then linearly interpolated across an edge when forming a scan line segment and linearly interpolated across the scan line segment during rasterization, resulting in a smoothly shaded object. More advanced shading techniques would be easy to implement. Like the first two steps, all the operations are performed on the vertices.

The fourth step scan converts the polygons by determining which polygon edges intersect a particular scanline and interpolating the X, Z and shaded color information along the polygon edge to determine the value for a particular Y scan-line. We start the scan-conversion process by finding, for each polygon, any intersections that a scan line makes with each of the polygon sides. Since polygons completely outside the viewport are ignored, there must be at least two intersections but, depending on the polygon shape, there may be many intersections. In order to process a general polygonal 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]. This operation is performed on polygon edges.

Next, the line segments which are completely outside the viewing window are discarded. This operation is performed on all line segments generated from the scan conversion step.

Hidden surface elimination is accomplished by employing a parallel Z-buffer algorithm[11]. This is done by rasterizing the line segments produced from the scan conversion process, clipping the resulting pixels against the viewport and then Z-buffering the non-clipped pixels. This operation is performed on pixels.

The first three steps operate on vertices while the fourth step operates on polygons. The fifth step operates on line segments and the final step operates on pixels. This analysis by data operations provides an excellent method for data layout.

If we strictly followed this, we would remap the virtual processors from vertices to polygons to line segments to pixels. The remapping of virtual processors involves general communication which is costly. If we map each polygon to a virtual processor and then iterate over the vertices within each polygon, we can eliminate one of the costly communications. The trade off is that this might result in a load balancing problem. When the polygon set contains polygons with varying numbers of vertices, iterating over the vertices within a polygon will result in idle virtual processors. The occurs when the iteration reachs the last vertex for some polygons while the polygons with more vertices still have remaining vertices to be processed. This can be mitigated by re-triangulating the polygons resulting in equal number of vertices for each polygon[14]. When this is the case, there is no load balancing problem since the steps involving vertices are perfectly balanced.

The data layout starts by assigning each polygon to a virtual processor. This layout is utilized by steps one through four. The scan conversion step generates line segments. These are each assigned a virtual processor. The fifth step marks as inactive, the virtual processors whose line segments fall outside the viewport. The last step uses the remaining line segments to generate pixels which are assigned to virtual processors used in the Z-buffering. Thus, there are only two remappings. The first four steps operate on objects thereby performing object space parallelism where as the last two steps operate on line segments and pixels thereby performing image space parallelism.

The most interesting parts of the algorithm are the scan conversion and Z-buffering. Let us look

more closely at those steps.

To save time and maximize the parallelism across polygons, a modified scan line conversion algorithm was used [12]. This algorithm is not restricted to convex polygons and scan converts arbitrarily complex polygons including those with holes. For this algorithm, we make the assumption 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 [13]. 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.

The scan conversion process iterates over the maximum number of scan lines through any polygon. Since scan conversion is concurrently executed for all polygons in parallel, it is bounded by the maximum number of scan lines within any polygon. The number of iterations necessary to process the entire set of polygons is the 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, the ones with a smaller number of scan lines passing through them, will have completed the scan conversion process. We address this load balancing issue in the next section.

The line segments are sorted into ascending X order as previously described. 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 assigned to virtual processors in a data parallel manner. This utilizes generalized router communication to map the line segments to virtual processors.

In the Z-buffering step, line segments from the previous steps are converted to pixels. Processing the lines to pixels in parallel requires iterating 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. For the shorter line segments, the virtual processors are marked as inactive when the segment completes this pixelization process. The Z and color values for the pixels are interpolated from the line end points. 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 inter-processor communications. The generalized data router must be used in conjunction with a sendmax combiner [3]. This utility uses the router for fast communications but it presents 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 Z-buffer compare. These are the values saved into the Z-buffer. When displaying the image, the image data is extracted from the Z-buffer.

In the renderer, there are two key loops, one for scan converting polygons into line segments and one for Z-buffering the line segments. The bulk of the renderer's computation time is spent processing these loops. Both of these loops have the following general form:

```
max_iteration = MAXIMUM_VALUE(iter)
do i=1,max_iteration
   WHERE (i .le. iter)
      ...loop code...
   ENDWHERE
enddo
```

The parallel array *iter* contains the number of loop iterations for each data element processed in the loop. The number of loop iterations is data-dependent. In the scan conversion code, *iter* contains the number of scan lines in each polygon. During each iteration, a different scan line from each polygon is processed in parallel.

In the data parallel programming model, the loop executes for the maximum number of iterations. In the best case, the iteration values are the same. Such a loop is said to be balanced. If the iteration values vary over a wide range the loop is said to be unbalanced. Unbalanced loops are inefficient. As the loop progresses, processors become idle because they have finished their iterations. A second source of inefficiency results from any serial code in the loop being repeated for the maximum number of iterations.

A load balancing algorithm was developed which balances this type of data parallel loop. The algorithm works only for data parallel loops with independent loop iterations. That is, each iteration does not depend on any values computed in previous iterations. The algorithm has the following properties:

- Speeds up the computation of unbalanced data sets
- Has a low added cost to the computation of balanced data sets
- Utilizes the existing memory space
- Utilizes the existing code with simple modifications

Clearly, the algorithm should speed up the computation of unbalanced data sets. Any extra work to balance the data adds a cost to the computation of balanced data sets. If it is unknown whether the input data sets are balanced or unbalanced, a tradeoff can be made: non-optimal speed up of the unbalanced computations for a lower added cost to balanced ones. The renderer's input data sets will be balanced or unbalanced depending on the algorithm and the data used to generate them. For example, an isosurface algorithm will generate a fairly balanced data set if the input data is dense and on a uniform grid since the polygon sizes will be bounded by the cell size of the grid. An isosurface algorithm which performs a coplanar merge on its resulting polygons, however, can create unbalanced data sets since some polygons might be substantially larger than others. Unbalanced data sets can also result from the application of viewing transforms in the renderer. For example, applying a wide-angle perspective view transform to a balanced data set can result in an unbalanced data set, since, in the transformed data set, polygons closer to the viewer are much larger than polygons far from the viewer.

The load balancing algorithm utilizes existing memory space so that the full memory of the machine can be devoted to rendering the data sets. This is especially important for the massive data set sizes the renderer must handle. Since developers can utilize their existing code, the algorithm can easily be used as a mechanism for load balancing other loops of this form.

The load balancing algorithm is implemented in the data parallel programming model. In this model, virtual processors provide the abstraction of having one processor per data element of each parallel array. The data elements from each parallel array are stored in each virtual processor's data space. For example, in the scan conversion loop, each virtual processor stores information about one polygon: the x,y,z coordinates of each vertex, color and normal information. Each virtual processor's data space also contains the number of loop iterations needed to process the loop data.

A virtual processor is *freed* if it will stay idle during all remaining loop iterations. Virtual processors are freed before and during the execution of the loop. Virtual processors are freed before loop execution if they are not assigned work initially. In the scan conversion loop, virtual processors are not assigned

work initially if the polygons in their data space are clipped or back-face culled. Polygons are clipped when they lie outside the user's viewport window. Zooming in on the objects being viewed can cause polygons to be clipped. Polygons are back-face culled when their surface normals point away from the viewing direction. Virtual processors are freed during loop execution when they complete their specified number of iterations.

Any free virtual processors can take on new work. A virtual processor takes on new work by copying the relevant data from an active heavily loaded virtual processor's data space into its own. The remaining iterations of the active virtual processor are then split between all processors which share the data. For example, if the work is split between two virtual processors, one processor will compute the lower half of the iterations and the other the upper half. By splitting the iterations between virtual processors, the loop iterations are balanced across the virtual processors and the maximum number of iterations needed to complete the loop is reduced.

A distribution function is used to compute how free virtual processors are assigned new work. The inputs to the function are the number of free virtual processors and the iteration array. The output of the function is a distribution. A distribution is a parallel array which contains the number of free virtual processors assigned to each active data space.

Computing an optimal sequence of distributions is np-complete. So, the load balancing algorithm uses a heuristic function to calculate a distribution. A new distribution is computed at the beginning of each loop iteration using the current number of free virtual processors and iteration array values.

The distribution function is computed as follows: first, the average of the current number of iterations is computed. The average is computed by summing the current active iteration array values and dividing this sum by the number of data elements in the iteration array. The average is used to decide how free virtual processors are assigned to active data spaces. If there are more free virtual processors than active virtual processors, the free processors are distributed equally to all active processor's data spaces whose iterations are greater than the average. The remainder of the free processors are left idle. If there are less free virtual processors than active processors then the previous distribution is returned, resulting in no new load balancing.

The average is used because it is a lower bound on the new maximum number of iterations that can be achieved by any distribution of free processors. Since the average is the lower bound, there is no reason to assign free processors to data spaces with iterations less than or equal to the average.

To distribute active data spaces to free virtual processors, a sequence of communication steps is executed. These steps should only be executed when the iterations saved outweigh the communication costs. In order to assess when it is beneficial to distribute free processors, information about the new distribution is computed and timing statistics are taken.

The new maximum number of loop iterations with the current distribution is computed to decide whether the communication steps should occur. This value is computed by adding one to all distribution array elements where their corresponding iteration array elements are active. This addition incorporates all the currently active processors in the iteration array into the distribution array. The iteration array is then divided by the distribution array. This result calculates what the value of the iterations will be after has splitting occurs. The maximum of this result is the new maximum number of iterations.

Timing statistics are also computed to assess whether the communication steps should occur. Each time the communication steps are executed, they are timed and an average of the execution times is updated. The body of the loop is also timed, and an average of the execution times is updated each iteration. The ratio of the average communication step time and the average loop body time gives a measure of the time for the communication steps in terms of loop iterations. An initial estimate of this ratio is supplied by the developer.

Using the new maximum iteration and ratio value, a test is computed: if the new maximum number of iterations plus the ratio is less than the current maximum iteration than it is profitable to distribute the free virtual processors.

At the beginning each loop iteration, the free virtual processors are counted and the distribution function is called. The test is executed. If the test is successful, active data spaces are distributed to the free virtual processors. The following pseudo-code provides an overview of the load balancing algorithm:

```
max_iteration = MAXIMUM_VALUE(iter)

loop until max_iteration

   compute the distribution and the
     new maximum number of iterations
     based on current number of free
     virtual processors

   ratio = INT(average communication
     time / average loop time)

   if (new_max_iteration + ratio .lt.
     max_iteration) then
      do load balancing communication
        steps

      max_iteration = new_max_iteration
   endif

   ...loop code...
endloop
```

5 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 rendering. Table 1 shows the times for rendering a data set consisting of 228,288 small polygons on the CM-200 with partition sizes of 16K, 32K and 64K. Table 3 shows the times for rendering the same data set on the CM-5 with partition sizes of 32, 64, 128, 256, and 512 nodes. Table 2 shows the times for rendering a data set with large polygons while Table 4 shows the times for rendering the same data set on the CM-5. The rows of each table contain data for three different viewing angles. The first row is the data for a viewing angle of (0,0), the second row is the data for a viewing angle of (45,45), and the third row is the data for a viewing angle of (90,90). The columns are partition sizes for each of the MPPs. All times are reported in seconds. 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. Figure 2 shows the

Times (sec)			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	365,260	681,456

Table 1: Rendering of Small Polygons on CM-200

Times (sec)			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

results of rendering a data set generated from a hydro-dynamics code running on the CM-200 and the CM-5[4].

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 some polygons covered over 3/4 of the image. Recall that 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. The addition of the load balancing algorithm to the renderer helps alleviate this problem.

On a 64K CM-200 partition, rendering speeds of over 600,000 polygons per second were achieved on the small polygon data set. 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 workstation 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 node partition of the CM-5, rendering speeds of close to one million non-meshed polygons per second were recorded. This exceeds the speed of current state of the art dedicated graphics hardware.

Figure 3 shows the speedup of the algorithm run on the CM-200 while Figure 4 shows the speedup of the algorithm run on the CM-5. The speedup curves are relative to the implementation on the smallest partition available for each of the MPPs. While the demonstrated speedup is not linear, the graphs show that near linear speedup for the data set with small polygons was achieved. Rendering the data set with large polygons exhibited worse speedup due to the large number of iterations being performed.

The renderer's scan conversion loop has been modified to use the load balancing algorithm. Load balancing the scan conversion loop improves the renderer's performance on unbalanced data sets. These improvements are detailed in Tables 5 and 6. On balanced data sets, performance is slightly degraded, by 2 to 7 percent of the render's performance without the load balancing modifications. Performance is especially improved on data sets which have been zoomed and clipped because of the many free virtual processors available before the loop executes. These improvements are shown in Tables 7 and 8. The

[4]See the color plates for a better picture of Figure 2.

Times (sec)				
32	64	128	256	512
5.756	4.754	1.482	0.796	0.463
3.402	2.877	0.966	0.498	0.302
2.476	1.048	0.550	0.401	0.236

Polygons/second				
32	64	128	256	512
39,660	48,020	154,040	286,794	493,062
67,104	79,349	236,322	458,409	755,920
92,200	217,832	415,069	569,296	967,322

Table 3: Rendering of Small Polygons on CM-5

Times (sec)				
32	64	128	256	512
37.326	20.169	12.488	8.466	6.220
30.684	16.034	9.758	6.393	4.717
17.261	9.353	5.543	3.594	2.590

Polygons/second				
32	64	128	256	512
5,939	10,807	17,455	25,746	35,045
7,104	13,595	22,338	34,097	46,212
12,628	23,306	39,325	60,651	84,163

Table 4: Rendering of Large Polygons on CM-5

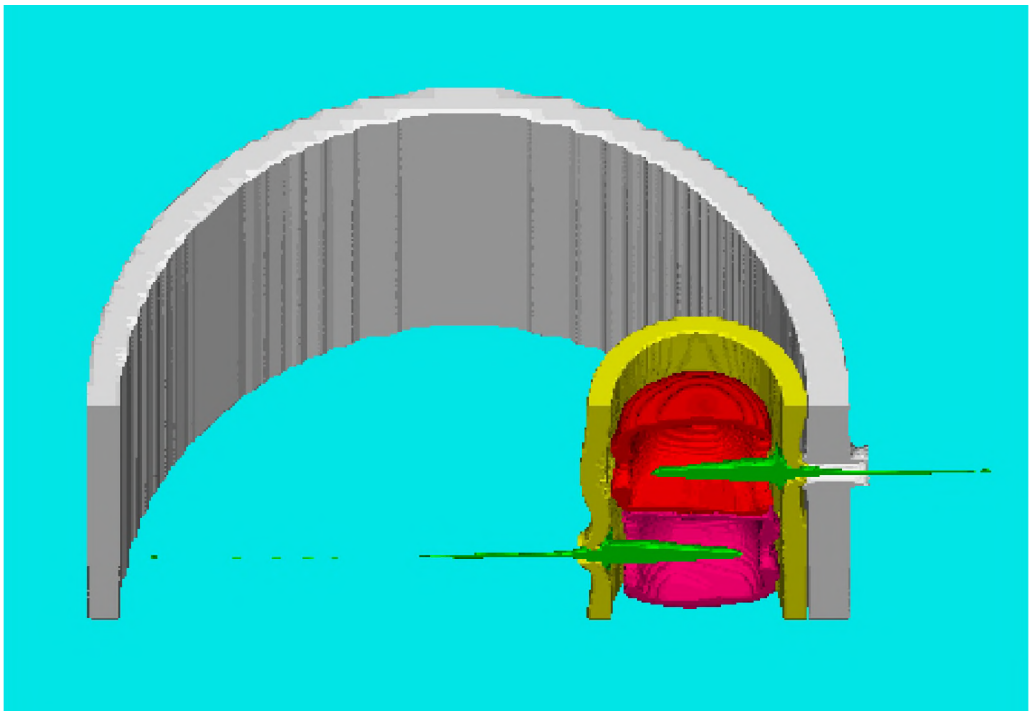

Figure 2: Image of Oil Well Perforator with 355,948 Small Polygons

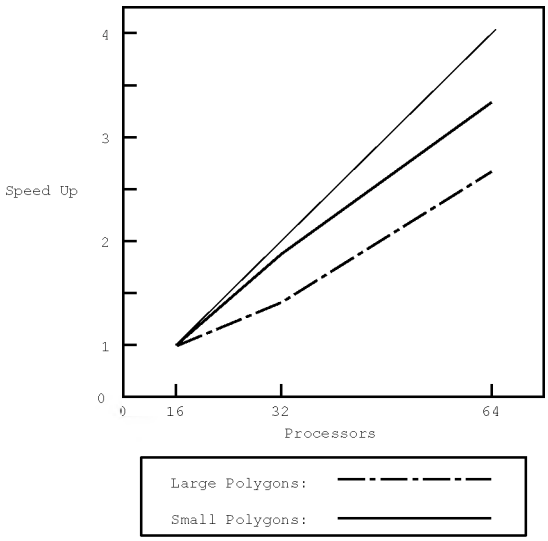

Figure 3: CM-200 speed up

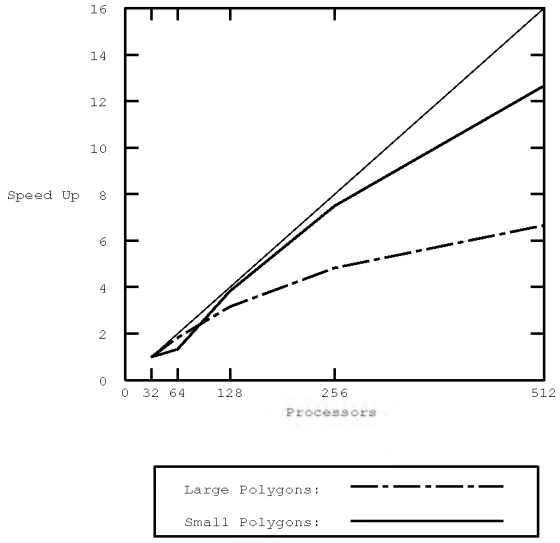

Figure 4: CM-5 speed up

Times (sec)				
32	64	128	256	512
29.387	17.580	11.029	7.544	5.712
22.585	13.288	8.235	5.624	4.201
16.034	8.985	5.465	3.650	2.832

Table 5: Rendering of Unbalanced Polygons without Load Balancing on CM-5

times were obtained by appling a scale factor of 3 to the unbalanced data set. With a viewing angle of (0,0) on this zoomed and clipped unbalanced data set, performance of the render with load balancing is 3 to 6 times faster than the performance of the render without the load balancing modifications.

The unbalanced data set was generated by merging the output polygons of an isosurface algorithm. This was done to reduce the number of polygons in the data set. The original isosurface polygons are shown in Figure 5[5]. The merged isosurface polygons are shown in Figure 6. Note the difference between the data sets, the polygons in the original data set are approximately the same size whereas the polygons in the merged data set have many different sizes, varying from extremely large polygons to extremely small polygons.

Load balancing the Z-buffering loop does not currently improve the renderer's performance. A small portion of the degradation results from the overhead of doing the load balancing (i.e. the initialization, distribution function calculation and testing). The larger portion results from timing fluctuations. The body of the Z-buffering loop is a single communication call. The execution time of this single communication call varies greatly. The communication ratio used to decide when to load balance is inaccurate because the average execution time of the single communication call varies so much over time. Unprofitable communication steps take place and the overall performance of the renderer degrades. This is a limitation of the load balancing algorithm: if the communication ratio is inaccurate, performance

[5]See the color plates for a better picture of Figure 5 and Figure 6.

Times (sec)				
32	64	128	256	512
11.482	7.047	4.180	2.368	1.502
12.381	7.284	4.128	2.481	1.556
13.191	7.194	4.371	2.561	1.765

Table 6: Rendering of Unbalanced Polygons with Load Balancing on CM-5

Times (sec)				
32	64	128	256	512
63.456	37.178	23.169	16.062	12.291
55.330	31.446	19.351	13.250	10.493
13.462	7.453	4.385	2.856	2.150

Table 7: Rendering of Zoomed and Clipped Unbalanced Polygons without Load Balancing on CM-5

Times (sec)				
32	64	128	256	512
18.244	10.063	5.666	3.205	2.023
36.318	19.679	10.778	5.882	3.733
13.811	7.711	4.453	1.767	1.234

Table 8: Rendering of Zoomed and Clipped Unbalanced Polygons with Load Balancing on CM-5

Figure 5: Closeup Image of Original Polygon Edges

Figure 6: Closeup Image of Merged Polygon Edges

can be degraded.

6 Conclusions

This paper described a data parallel method of polygon scan conversion allowing the visualization of large 3D simulations directly from a massively parallel processor. 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 processor to a graphics workstation. Issues involving load balancing were addressed and a data parallel load balancing algorithm was presented. The load balancing algorithm achieves the desired speed tradeoff between the computation of unbalanced data sets and balanced data sets by using a simple heuristic function to calculate the distribution of free virtual processors. The algorithm improves performance and only utilizes existing memory space. It does this by making use of virtual processors which are freed before and during loop execution. Performance data was provided that showed the rendering method can out-perform high-end commercially available graphics workstations.

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

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

[3] Thinking Machines Corporation. CM Fortran user's guide, 1991.

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

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

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

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

[8] E. Fiume et. al. A parallel scan conversion algorithm with anti-aliasing for a general purpose ultracomputer. *Computer Graphics*, 17(3), July 1983.

[9] 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.

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

[11] Foley, van Dam, Feiner, and Hughes. *Computer Graphics: Principles and Practice.* Addison Wesley, 1990.

[12] John D. Fowler. A reduced set scan line algorithm. Internal Document, Los Alamos National Laboratory.

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

[14] P. Hinker and C. Hansen. Geometric optimization. In *Proceedings of Visualization '93*, 1993.

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

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

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

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

[19] Peter Schroder and Steven Drucker. A data parallel algorithm for raytracing of heterogeneous databases. In *Proceedings of Computer Graphics Interface*, 1992.

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