

Parallel Sphere Rendering

Michael Krogh, James Painter, Charles Hansen

Advanced Computing Laboratory

Los Alamos National Laboratory

Los Alamos, New Mexico 87545

{krogh,jamie,hansen}@acl.lanl.gov

Abstract. Sphere rendering is an important method for visualizing molecular dynamics data. This paper presents a parallel algorithm that is almost 90 times faster than current graphics workstations. To render extremely large data sets and large images, the algorithm uses the MIMD features of the supercomputers to divide up the data, render independent partial images, and then finally composite the multiple partial images using an optimal method. The algorithm and performance results are presented for the CM-5 and the T3D.

1 Introduction

In recent years, massively parallel processors (MPPs) have proven to be a valuable tool for performing scientific computation. Available memory on these types of computers is greater than that found on most traditional vector supercomputers. For example, a fully populated 256 node T3D has 16 gigabytes of physical memory. A 1024 node CM-5 contains 32 gigabytes of physical memory. As a result, scientists who utilize these MPPs can execute their three dimensional simulation models with much greater detail than previously possible. While current simulations don't typically 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. For example, molecular dynamics simulations of structural materials have reached 600 million atoms [1, 13]. While researchers don't usually perform simulations with 100 million atoms, 10 million to 40 million atom simulations are becoming routine. Figure 1 and Figure 2 show images of such data.

Figure 1 shows a molecular dynamics simulation composed of 104 million atoms (the atoms are generic and not of a particular element). The atoms are arranged in a thin plate, approximately 1500 x 1500 x 48, with outward forces applied to both the left and right edges. A tiny notch at the lower edge of the plate bifurcates with time to produce a brittle fracture. Colors indicate kinetic energy: grey to blue are low energies and yellow to red are high energies. Figure 2 is a close up of the bifurcation.

With such large data sets, visualization is an essential tool for analyzing the simulation output. Researchers wish to gain insight into both macroscopic phenomena as well as microscopic phenomena. Traditional methods, such as statistical analysis and

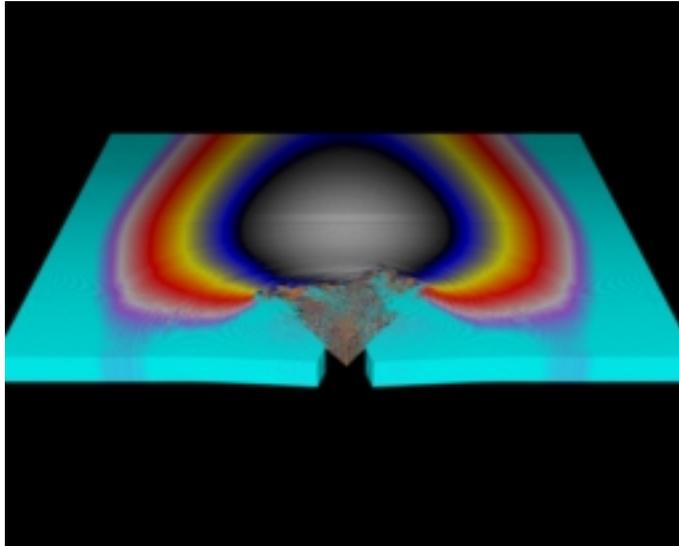


Figure 1: 104M Atoms

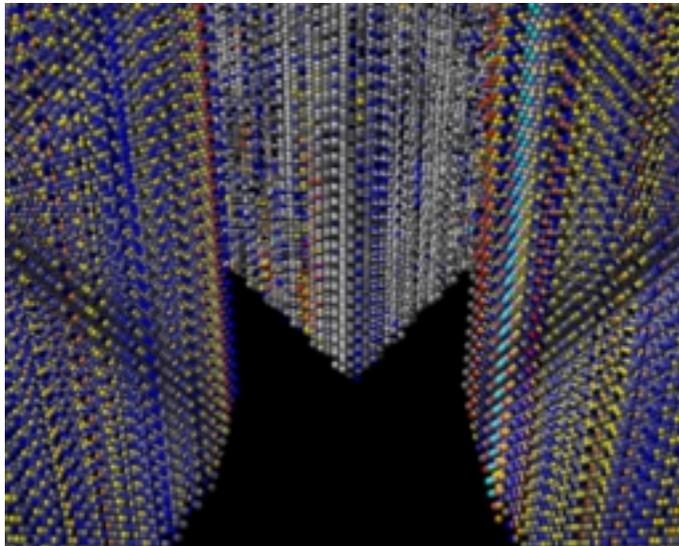


Figure 2: 104M Atoms (close up)

browsing through the raw simulation data, aren't adequate by themselves for analyzing data sets ranging in size from gigabytes to terabytes. Visualization, because of its high bandwidth, enables a researcher to explore his or her data sets in a more timely and fruitful manner [15].

In structural molecular dynamics, researchers wish to see both the large scale structures in the data as well as microscopic phenomena, such as the onset of dislocations. This requires portrayal of the entire data set in a single image, where individual atoms may not be distinguishable, and the ability to zoom in on minute details, where only a fraction of the data can be seen. Of course, researchers also wish to see these features as they vary with time.

For a molecular dynamics simulation, the atoms are usually represented by spheres. One could argue that with several tens of millions of atoms, most would be subpixel in size and that a pixel representation would be adequate. This argument is certainly true when the viewpoint is sufficiently far away from all of the data. In this situation the atoms shrink to pixel or subpixel sizes. However, we tend to move the viewpoint in and around the data set thus causing the atoms to cover tens of pixels in diameter. This technique is very useful for showing minute details within the data sets as can be seen in Figure 2.

Empirically, we have found that a sphere tends to be the best representational form for an atom in our simulations. Spheres can vary cleanly from subpixel sizes to covering the entire image in extreme cases; they seem to introduce the fewest artifacts. Also, it is a long standing and familiar representational form for atomic data.

The problem with spheres is that they can be expensive to render. Most graphics workstations tend to approximate spheres with a collection of polygons which causes a data explosion. Every sphere is tessellated into many tens of polygons. Instead of rendering 40 million spheres for a single time step, one may need to render upwards of 400 million polygons. Clearly a different approach to rendering spheres is needed or vast improvements in hardware rendering.

Another question that can be asked is where to do the rendering. Given the availability of a MPP and a graphics workstation, one could render on either the MPP, the graphics workstation, or distributed across both. The graphics workstation is probably not the right choice for two reasons. First, 40 million atoms requires approximately 800 megabytes of storage and most workstations don't have sufficient memory to store many time steps of this size. Second, from our own experiments, we know it takes approximately 34 minutes to render 40 million atoms on a Silicon Graphics Inc. Onyx workstation equipped with RealityEngine2 graphics hardware. Additionally, it requires on the order of 30 minutes to move the data set from the supercomputer to the workstation. Two thousand time steps would take over 83 days to render! The distributed approach doesn't provide a solution either. The primary bottleneck is still the graphics hardware, and furthermore, moving hundreds of megabytes per time step can take a substantial amount of time even over high speed networks. Rendering on the MPP is an attractive solution given that the data resides on the MPP's disk farm and that the MPP has sufficient memory. The challenge becomes developing an algorithm that can exploit the multitudes of processors while minimizing overhead, typically communication costs.

2 Related Work

2.1 Sphere Rendering

Spheres can be rendered using several approaches. One approach, which is used by Silicon Graphics Inc., is to tessellate a sphere into a set of polygons which can then be rendered as any other polygons using the custom hardware [20]. Several tessellation methods exist, such as a bilinear, octahedral, icosahedral, cubic, and others. The problem with this approach is that each sphere is transformed into many polygons. The number of polygons varies depending upon the degree of smoothness desired. Spheres with lesser degrees of tessellation appear less spherical than those with higher degrees of tessellation. Others have used similar approaches. Staudhammer describes a method where spheres are tessellated but the shading is based on the sphere not the polygons, thus the rendered sphere has a more spherical appearance while perhaps using fewer polygons[21].

Silicon Graphics Inc. also provides alternative “bit-mapped sphere” rendering method that uses the texture memory of the reality engine to “splat” images of a phong-shaded sphere, with several restrictions [20]. Perspective projection is not allowed with bit-mapped spheres and clipping occurs on an entire sphere basis, meaning spheres pop in and out as they cross over the clip planes.

Porter’s approach treats a sphere as a primitive in his scan line based rendering algorithm [19]. For any spheres that intersect the current scan line, the intersection of the sphere silhouette and the image plane is determined and then a Bresenham algorithm is used to calculate z values for hidden surface removal. Shading is approximated with a cosine function. Additional features of his algorithm include sorting for transparency and pixel averaging at sphere intersections and border intersections. This is done for anti-aliasing.

Patterson proposes an algorithm that approximates a sphere with a parabola, thus replacing a square root operation with a division. His algorithm uses a modified Bresenham’s circle algorithm for iterating through the set of points in the framebuffer that make up the sphere [18].

Blinn developed an elegant incremental algorithm suitable for scan conversion of a single sphere, for example a planet model [2, 3, 4]. The technique requires a screen coordinate system where the near and far z planes are tangent to the sphere being rendered. As Blinn states, multiple spheres cannot be rendered using his technique in the same z depth coordinate system. This makes his technique unsuitable for z buffer based visible surface determination. He instead uses a painter’s algorithm for the visible surface problem. Halperin and Overmars also use a depth sort and painter’s algorithm to render spheres for molecular models. The $O(N \log N)$ time bound required for a depth sort, where N is the number of spheres, is unsuitable for rendering millions of spheres.

2.2 Parallel Rendering Classification

Many researchers have studied parallel algorithms for polygon and volume rendering in recent years [9, 10]. Molnar *et al.*, provide a useful taxonomy for parallel rendering which classifies parallel rendering methods as sort-first, sort-middle, or sort-last [16]. This proves useful when deciding on how to structure a rendering algorithm. Basically,

the rendering process requires a mapping from 3D object space to 2D image space which typically requires sorting the data. This sorting can occur at three different locations: at the beginning of the rendering process, during the middle, or at the end. Each method has its advantages and disadvantages.

Sort-first assigns each renderer to specific screen region. The renderers are responsible for the complete rendering process, both geometry processing and rasterization, for all primitives that map into their region of responsibility. The mapping of primitives from object space to image space is decided initially by a relatively simple pre-transformation operation. This approach is advantageous if a primitive has a high degree of tessellation, because less data is communicated, or when there is high frame-to-frame coherence, because fewer primitives need to be communicated on successive frames.

With sort-middle, primitives are evenly divided among the processors where tessellation and geometry processing are performed. Then the transformed primitives are sent to the rasterizer processors. These processors, assigned to disjoint regions of the image plane, are responsible for rasterizing primitives that lie within their region of the image. The advantage of this approach is that geometry processing tends to be fairly well load balanced, however disadvantages include potentially poor load balancing for the rasterizers and high communication costs if tessellation is high.

With the sort-last method, also known as image compositing, each processor is responsible for the complete rendering process, as with the sort-first approach. Instead of mapping primitives up front to their correct image space processor, primitives are equally distributed among the processors at the beginning, rendered into a complete image, and then the images are composited. Two advantages of this method are: potentially easier load balancing, and image compositing times depend only on image size and not model complexity. The disadvantage is that each processor must have enough memory for the image buffer and that communication involves transferring the image buffer during compositing. This can be rather significant for very large images.

2.3 Parallel Sphere Rendering

Thinking Machines offered parallel sphere rendering with their ***Render** library on the CM-200 [23]. This used the data parallel model to render spheres. The algorithm worked well for small spheres of equal radii achieving rendering rates of 20k spheres per second on the CM-2. However, the algorithm exhibited extremely poor performance when the spheres were of unequal size due to well known load balance problems in the data parallel regime.

Fuchs *et al.* describe a parallel sphere rendering algorithm, credited to Fred Brooks, for the per-pixel SIMD linear expression evaluators of Pixel Planes 3 [6]. A single sphere is rendered in a constant time, independent of its screen space coverage, achieving rendering rates of approximately 35,000 spheres per second. This algorithm was also implemented on Pixel Planes 5, achieving more than one million spheres per second [7, 12].

3 Massively Parallel Processors at the ACL

The Thinking Machines Corporation CM-5 is a production quality massively parallel supercomputer [22]. The CM-5 can consist of 32 to 16384 Sparc processors. The

CM-5 at the Advanced Computing Lab (ACL) at Los Alamos National Laboratory consists of 1024 processors, each with 32MB of local RAM for a total of 32GB. Each Sparc processor also has four 64-bit wide vector units which can assist in floating math operations. With four vector units up to 128 MFlops per node can be performed. This yields a theoretical speed of 128 GFlops for the ACL CM-5. In practice roughly 60 GFlops has been sustained. It should also be noted that not all vendor supplied languages use the vector units and may, in fact, rely on the Sparc processors to do all floating point math. In this case a user can expect to see only 5 MFlops per processor.

The T3D from Cray Research Inc. is a MPP consisting of 32 to 2048 DEC Alpha processors [5]. The ACL T3D has 256 processors, each with 64MB of RAM for a total of 16GB. Each Alpha chip is rated at 150 MFlops, thus the machine is theoretically capable of 38.4 GFlops total performance. However, we have seen speeds in the 2 MFlop to 20 MFlop per processor range on average. Like the CM-5 the nodes are arranged into partitions which must contain a power of 2 processors. The nodes are hosted by one of three CPUs of the YMP front-end. Unlike the CM-5, the partitions are dynamically determined and are not tied to a specific front-end host processor.

The T3D nodes (2 Alpha processors comprise a node) are organized into a torus network via 300 MB/s links. The torus is then connected to the YMP through two 200 MB/s high speed links. All I/O (including HIPPI, FDDI, and disk) from the Alpha processors is done through the YMP.

3.1 Programming Models

Both the CM-5 and the T3D support SIMD and MIMD programming models. The SIMD (Single Instruction, Multiple Data) model performs the same operation on all selected data elements. For example, given an array of numbers, a constant could be added to each number. When using the SIMD 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 element in the array.

The MIMD (Multiple Instruction/Multiple Data) model, divides a task up into a number of subtasks that can run concurrently and independently. Some subtasks can occur in parallel while others might occur serially. For example a set of processors might be used to render frames for an animation. Each processor would have a data set to render along with the viewing and/or motion transformations for its given frame within the animation. Each renderer can execute in parallel and independently from the other processors to produce a finished image. In this example, no synchronization is needed between individual processors.

MIMD programs can be either host/node programs or hostless programs. A host/node program consists of a host program, frequently referred to as a master program, and some number of node programs, also known as slaves. When a host/node program commences execution, the host contains the thread of execution and is responsible for invoking the node programs. The host typically communicates some initialization parameters to the nodes and then instructs them to perform some task. The node programs do not need to stay in synchronization after this point, although programs typically involve some synchronization points, usually where communication between nodes or between nodes and the host occurs. On the CM-5 communication is per-

formed via the CMMD library whereas the T3D uses the PVM message passing library [8]. Hostless programs differ from host/node programs in that they do not have a user supplied host program as the name implies, although CMMD supplies a standard host program to provide initialization and I/O serving. On the T3D hostless programs can use either PVM, Cray's explicit shared memory library, or some other library such as FM [11] or ACLMPL [17].

4 Parallel Sphere Algorithm

Our sphere rendering algorithm is based on a sort-last approach. This approach has two essential advantages for our problem: better load balancing and reduced communications. Better load balancing is achieved because the spheres are evenly distributed across all processors for both geometry processing and rasterizing.

Reduced communication cost can easily be seen. Instead of having to transfer 40 million atoms, each of which consists of 20 bytes of data, between geometry processors and rasterizers, we only need to transfer images or partial images. The cost of transferring spheres can even be greater if the spheres were tessellated. Communication time costs for image compositing grow by $O(N)$, where N is the number of pixels in the final image, and depend neither on the number of spheres rendered nor on the number of processors used. This is discussed later in the paper.

The sphere renderer was originally designed as a MIMD program using a host/node model, written in C with CMMD, for the CM-5. The choice of MIMD over SIMD was made because of the desire to allow processors to compute asynchronously as much as possible. Even though each processor has the same number of spheres to render (plus or minus one), each sphere can be a different diameter in pixels. Additionally, spheres can be clipped if they fall outside of the image. This dictates varying amounts of computation. The host/node model was chosen over the hostless model because of the desire to display the resultant images through an X11 window. At this time the hostless model on the CM-5 does not support X11. Otherwise, the program can be easily structured using either model.

Molecular dynamics simulations are frequently time dependent and typically, the researcher wishes to see an animation of the time series. Thus, we included support for batch processing of sequences of data sets within the renderer.

Two additional features we required were the capability to render any number of atoms and images of any size while working within the memory constraints of any given partition size. The amount of memory available to the renderer is at most 32MB per node on the CM-5 (which might be shared with other running jobs) and 64MB per processor on the T3D (which is not shared); virtual memory does not exist. Because of this, the renderer must determine how much of the data set it can fit into memory as well as how much of the image will fit. If sufficient memory is not available, then either the data set, image, or both may be processed in subsets or tiles. Figure 3 shows the algorithmic steps for the main part of the renderer.

The only function of the host is to interact with the user and to initialize the nodes. The host will also be responsible for image display, via X11 or HIPPI frame buffers.

The main body of the renderer is a loop that iterates over the sequence of data sets. A data set, or some portion of it, is read in from disk: either an NFS (Network File System) mounted disk or from parallel disk (SDA on the CM-5 or RAID-3 on the T3D).

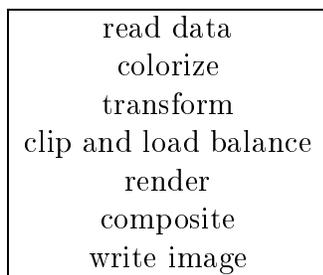


Figure 3: Steps in the Algorithm

The read operation on the CM-5, which looks like a normal UNIX read, has the option of automatically dividing up the data set among the processors in the current partition. For example, the first processor can read the first n th of the data, the second processor can read the second n th of the data, and so on.

Next each processor colorizes its atoms based on a scalar quantity associated with each atom. Atoms are then transformed from object space to screen space. Part of the transformation operation includes transforming the atom radius into an image space radius with a perspective transformation (*i.e.* spheres closer to the viewpoint will appear to be larger than those further away). This is accomplished with a perspective scaling factor and linear interpolation along the z depth. While this is not entirely accurate ¹, the results are indistinguishable from the exact result obtained with a “real” perspective transformation for our data sets and typical viewing parameters.

4.1 Load Balancing

After transformation certain spheres may fall outside of the viewing frustum, these spheres can be clipped. This has led to significant load balance problems. To compensate for this, we employ a simple heuristic. A list consisting of the number of visible atoms per processor is sorted. From this list the processors are paired up. The processor with the most atoms is paired with the processor with the least atoms; the processor with the second most atoms is paired with the second least, and so on. For a given pair of processors, the processor with the most atoms sends a portion of its atoms to the other so that both end up with the same number atoms plus or minus one.

We currently use a sequential sort on a single processor in $O(P \log P)$ time, where P is the number of processors. This has proven to be adequate for the machine sizes we have available (≤ 512 PEs). A parallel merge sort algorithm could be used to reduce the sorting time to $O(P)$.

Empirically, we’ve found that performing this simple load balancing step twice can yield good results while incurring only a small amount overhead (the third and fourth times cost more than they save). The choice of this simple method over more complex, but potentially more optimal, methods is due to the desire to avoid more complicated communication between nodes which can be very slow.

¹A sphere viewed in perspective will yield an elliptical projection in screen space in the general case. We approximate this ellipse with a circular projection.

4.2 Scan Conversion

After all spheres have been transformed, each is scan converted, one at a time, into the image and z buffers. The image buffer consists of 4 bytes per pixel: red, green, blue, and alpha (which is not used at this time). The z buffer is a single floating point value per pixel. The scan conversion is done by evaluating a distance equation for each pixel within the bounding box for the sphere's silhouette. If the current pixel is within the sphere, then a z buffer comparison is made to determine visibility. If this checks, then the color is determined. Instead of computing the true perimeter of the sphere, which involves a square root operation, we simply leave off the square root operation for both the distance check and the z buffer comparison. This yields significant savings.

We have additional savings in determining pixel colors. Instead of using photo-realistic lighting calculations (*e.g.* Phong), we use a simple formula which approximates diffuse and ambient shading with a single light source with very good results. We compute the distance from the projected center of the sphere's silhouette to the pixel. This value is used to linearly interpolate an intensity that varies from "white" at the center to "almost black" at the perimeter. This value is multiplied by the atom's color to achieve a "shaded" RGB color. This is similar to precomputing the intensity term in diffuse shading. To simulate a light source located above and to the right of the viewpoint, we add in an offset in the distance calculation.

4.3 Compositing

After all P processors have rendered their spheres, they synchronize and the P images are composited. Since transparency is not supported, we do not have to worry about compositing order. The CM-5 has a CMMD function, **CMMD_reduce_v**, that implements a global reduction on an array using a specified operator. We use **CMMD_reduce_v** twice, once to composite the z buffer and then a second time to composite the image buffer. When compositing the z buffer, we specify a minimum operator and our z buffer as the array to reduce. After this call, each processor has a new z buffer array that contains the minimal z value across all processors for each pixel location. This will select the values for each pixel which are closest to the viewpoint. Next the processors loop over the composite z buffer and compare its value at each pixel location with that of its original z buffer. If the values match, then it knows that its pixel is closest to the viewpoint, otherwise it is obscured by some other pixel on another processor. In this case, the processor would set its pixel at that location to black, a minimal value. Following this, the processors would call **CMMD_reduce_v** again but this time specifying the image buffer along with the maximum operator. Only pixels still having a color will contribute to the resultant image. After this step each processor contains an identical image for the current image chunk. On the CM-5 this operation takes $O(N \log P)$.

If the renderer was not able to read in the entire data set, it then reads in the next portion of the data set and renders those atoms for the same partial image. But instead of starting with an empty image and z buffer, it uses the ones which were previously calculated. After the entire data set has been processed, the first processor writes out the final image, or current partial image, to the image file. If the image is being computed in tiles, then the next tile is rendered. This continues until a full image is

rendered.

5 T3D Issues

ACL acquired its T3D after we completed the CM-5 version of the renderer. We felt that an implementation for the T3D would be trivial since Cray had implemented PVM on the MPP. Due to performance problems and various restrictions and bugs, Cray's PVM implementation proved to be unusable for our application.

The routines for doing I/O had to be rewritten for the T3D. If all processors try to read their own portion of the data file, the YMP host becomes overloaded since it performs the actual I/O and would spawn a task for each Alpha processor. To avoid this, we had the host read the data file and parcel out the data to the Alpha processors. This did not yield good performance due to PVM. Thus, the YMP was removed as the host and processor 0 inherited the host's tasks and all file I/O.

We were left with the choice of using either Cray's explicit shared memory library or some other message passing library such as FM [11]. We did not want to use FM for this particular program for two reasons: it uses an active message model which would require extensive rewriting; and it does not support messages lengths as large as were required. The CRAY shared memory library provides a fast, but non portable one sided communications application program interface. In addition, the user must manage all synchronization between processors and cache coherency. We felt that direct use of the shared memory library would restrict the portability of our code. Our decision was to develop a lightweight library, which looks quite similar to CMMD, on top of the explicit shared memory library. Our timings indicate that our additional layer incurs very little overhead beyond direct shared memory calls [17].

One improvement the T3D version has over the CM-5 version is in compositing. By writing our own global reduction operations, we were able to merge the two separate global reduction operations and intervening comparison into a single operation. The z depth and RGB color of a pixel can be combined into a single 64 bit integer, with depth in the most significant 40 bits. This allows a normal 64 bit integer maximum reduction to accomplish the z buffer composition.

This new z buffer reduction uses our Binary Swap compositing scheme which runs in $O(N/2 + N/4 + N/8 + \dots + N/P)$ time where N is the number of pixels per processor and P is the number of processors² [14]. In the limit as P goes to infinity, this reduces to $O(N)$ time. Since the total problem size grows by the product $N \times P$, assuming a dense image distribution on each processor, an $O(N)$ algorithm on P processors is optimal.

6 Results

To gauge our algorithm's performance we timed several experiments on various CM-5 partitions and T3D partitions. The test case is a data set consisting of 5 million atoms and the image size is 1024 x 1024. For comparison we also ran a test case on a Silicon

²This analysis is correct for network topologies such as the hypercube where each path of communication is a single hop for this particular operation. Other topologies will have greater costs for some communication paths due to the possibility of link contention. In practice, link contention is rarely an issue on the T3D because of the speed of the interconnection network.

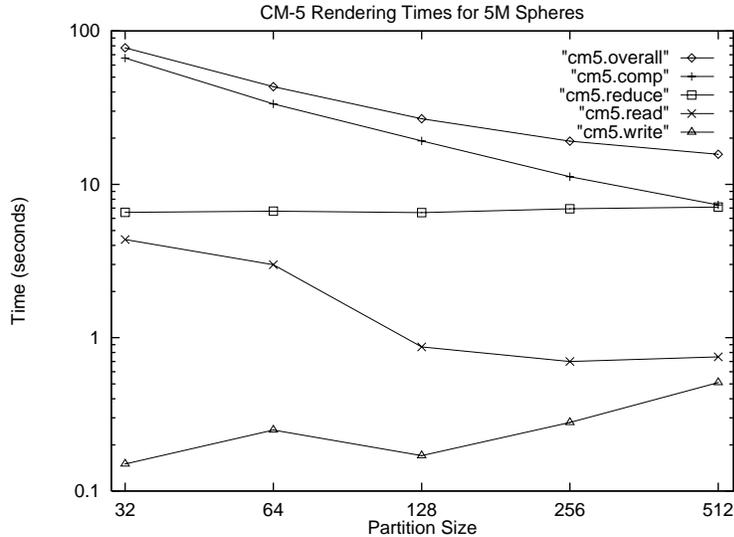


Figure 4: CM-5 Rendering Times

Graphics Inc. Onyx with a RealityEngine2 graphics board to get a sense of how long a typical high-end graphics workstation would take to complete the task.

The SGI workstation uses a simple, but optimized, program that invokes the SGI sphere drawing routine. Their sphere drawing routine tessellates a sphere into a set of triangle or quadrangle meshes (depending on user selected tessellation method). The number of polygons generated depends upon the user selected tessellation factor. Using a tessellation factor of 3 and the octahedral tessellation method, the SGI renders our test data set in 261 seconds, including 5 seconds for reading the data from a local disk. Using the bit-mapped sphere primitives reduced the run time to 63 seconds. We were forced to use orthogonal projection with the bit mapped sphere library as noted earlier.

One factor not included in the SGI times is the time it takes to transfer the data from the SDA to a local disk on the SGI. Over Ethernet it takes several minutes for a 5 million atom data set. This can be a significant cost that we would incur while generating a time series animation.

Figure 4 and Table 1 show rendering times for various partition sizes for the CM-5. Our initial results are quite promising. **Total** is the time for the program to read in the data, render, and write out the image. **Read** is the time to read in the data. **Write** is the time to write out the image. Both will vary from run to run due to other jobs using the SDA. **Composite** is the time the nodes spent doing compositing. **Computation** is the difference between the **Total** less the **Read**, **Write**, and **Composite**. As can be seen, the algorithm scales reasonably well across all partition sizes. The knee between 64 and 128 processors for the **Read** time corresponds to there being enough total memory to read the data set in a single pass. For partition sizes 32 and 64 two reads are required to process all of the data. The limiting factor for the CM-5 version of the renderer is the compositing operation.

Figure 5 and Table 2 show rendering times for various partition sizes for the T3D. **Read** also includes the time spent moving the data from processor 0 to the other processors during the read operation. Again, **Read** and **Write** will vary due to other processes and YMP load. The knee between 16 and 32 nodes corresponds to there

Partition	Read	Write	Composite	Computation	Total
32	4.37	0.15	6.57	66.57	77.66
64	2.99	0.25	6.68	33.45	43.37
128	0.87	0.17	6.54	19.20	26.78
256	0.70	0.28	6.93	11.21	19.12
512	0.75	0.51	7.11	7.34	15.71

Table 1: CM-5 Rendering Times

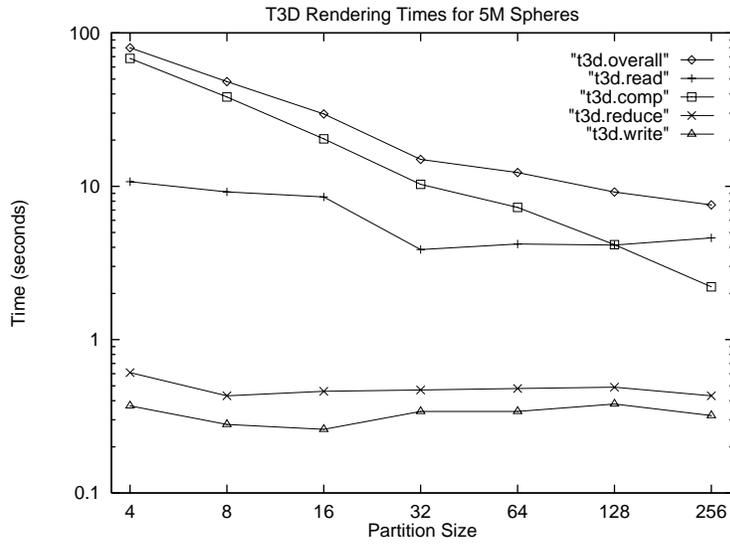


Figure 5: T3D Rendering Times

Partition	Read	Write	Composite	Computation	Total
4	10.71	.37	.61	68.25	79.94
8	9.20	.28	.43	38.25	48.16
16	8.52	.26	.46	20.39	29.63
32	3.87	.34	.47	10.30	14.98
64	4.20	.34	.48	7.28	12.30
128	4.14	.38	.48	4.16	9.17
256	4.61	.32	.43	2.21	7.57

Table 2: T3D Rendering Times

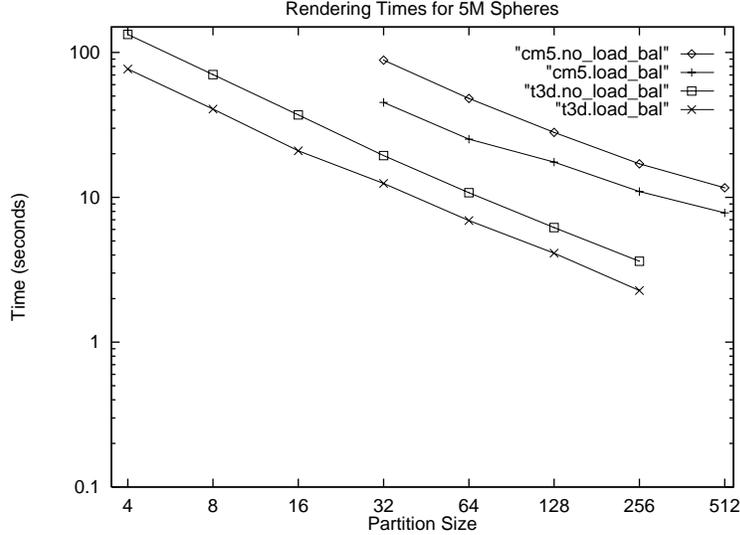


Figure 6: Computation Times

Partition	T3D LB	T3D no LB	CM-5 LB	CM-5 no LB
4	76.96	133.30	NA	NA
8	40.66	70.37	NA	NA
16	20.95	37.10	NA	NA
32	12.48	19.40	45.22	88.57
64	6.92	10.76	25.20	48.13
128	4.12	6.18	17.53	28.06
256	2.27	3.62	10.96	17.04
512	NA	NA	7.82	11.63

Table 3: Computation Times

being enough total memory for the entire data set to be processed in one pass. Here the algorithm exhibits better scaling than the CM-5 version as can be seen with the compositing and computation times. Although, the limiting factor with the T3D version is reading and distributing the data.

When comparing the T3D version to the CM-5 version, it is easy to see that the processors in the T3D are significantly faster. However, the T3D is hindered by not having parallel disk I/O available to the Alpha processors directly, thus requiring one processor to perform the read and data distribution.

Computational times with load balancing turned on and turned off are presented in Figure 6 and Table 3. Here **Computation Time** also includes any time spent load balancing with associated communications. It should be noted that the CM-5 does not support partition sizes smaller than 32 nodes and that time on the 1024 node configuration was not available for these experiments. The numbers differ from the previous results due to a different viewpoint which results in roughly 1/3 of the atoms being clipped. It is easy to see that load balancing is beneficial for situations in which it can be applied.

7 Conclusions

We have shown that parallel sphere rendering can be performed at very high rates on massively parallel supercomputers and that MPPs can be significantly faster than a graphics workstation. Rendering rates on the three platforms are in the range of: 660K spheres/second for the T3D, 318K spheres/second for the CM-5, 19K spheres/second for the SGI using tessellated spheres, and 79K spheres/second for the SGI using bit mapped spheres. If reading the data was not included in these rates, such as for an interactive viewer, then the rates would be: 1689K spheres/second for the T3D, 334K spheres/second for the CM-5, 19K spheres/second for the SGI using tessellated spheres, and 87K spheres/second for the SGI using bit mapped spheres. Our algorithm makes use of several optimizations to enhance performance: direct sphere rendering, simplified rendering and lighting equations, trivial but efficient load balancing, and optimal compositing.

Our algorithm should be portable to other supercomputers or to workstation clusters due to its straight forward design. The algorithm only requires basic message passing primitives for communications, although an optimized global reduction operator can be of value.

Our algorithm has the advantage of not relying on a hardware specific rendering engine. This gives us the capability of integrating it in with other types of renderers, such as polygonal and volumetric. Writing a renderer that combines our polygon renderer, sphere renderer, and volume renderer is the focus of our next effort. We also plan to look at additional load balancing techniques that deal with spheres of varying radii.

References

- [1] D. M. Beazley, P. S. Lomdahl, N. Gronbech-Jensen, and P. Tamayo. High performance communication and memory caching scheme for molecular dynamics on the CM-5. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 800–809. IEEE Computer Society Press, 1994.
- [2] J. Blinn. How to draw a sphere — part 1. *IEEE Computer Graphics and Applications*, 15(1):78–83, Jan. 1995.
- [3] J. Blinn. How to draw a sphere — part 2. *IEEE Computer Graphics and Applications*, 15(1):70–76, Mar. 1995.
- [4] J. Blinn. How to draw a sphere — part 3. *IEEE Computer Graphics and Applications*, 15(5):87–93, Sept. 1995.
- [5] Cray Research Inc. CRAY T3D hardware reference manual, October 1993.
- [6] H. Fuchs et al. Fast spheres, shadows, textures, transparencies, and image enhancements in Pixel-Planes. In *SIGGRAPH 85 Conference Proceedings*, pages 111–120. Association for Computing Machinery, Inc., July 1985.
- [7] H. Fuchs et al. Pixel-Planes 5: A heterogeneous multiprocessor graphics system using processor-enhanced memories. In *SIGGRAPH 89 Conference Proceedings*, pages 79–88. Association for Computing Machinery, Inc., July 1989.

- [8] A. Geist et al. PVM3 user's guide and reference manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, Oak Ridge, TN, 1993.
- [9] IEEE Computer Society. *1993 Parallel Rendering Symposium Proceedings*. ACM SIGGRAPH, Oct. 1993.
- [10] IEEE Computer Society. *1995 Parallel Rendering Symposium Proceedings*. ACM SIGGRAPH, Oct. 1995.
- [11] V. Karamcheti and A. Chien. A comparison of architectural support for the TMC CM-5 and the Cray T3D. In *Proceedings of ISCA '95*, 1995.
- [12] A. Lastra, July 1996. Private Communication, University of North Carolina, Chapel Hill.
- [13] P. S. Lomdahl, P. Tamayo, N. Gronbech-Jensen, and D. M. Beazley. 50 GFlops molecular dynamics on the Connection Machine 5. In *Proceedings of Supercomputing 93*, pages 520–527. IEEE Computer Society Press, 1993.
- [14] K. Ma, J. Painter, C. Hansen, and M. Krogh. Parallel volume rendering using binary-swap compositing. *IEEE Computer Graphics and Applications*, 14(4):59–68, July 1994.
- [15] B. McCormick, T. DeFanti, and M. Brown. Visualization in scientific computing. *Computer Graphics*, 21(6), November 1987.
- [16] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications*, 14(4):23–32, July 1994.
- [17] J. Painter, P. McCormick, M. Krogh, C. Hansen, and G. C. de Verdière. The ACL message passing library. *EPFL Supercomputing Review*, 7, November 1995.
- [18] J. W. Patterson. Fast spheres. In R. J. Hubbard and R. Juan, editors, *Eurographics '93*, pages 61–72, Oxford, UK, 1993. Eurographics, Blackwell Publishers.
- [19] T. Porter. Spherical shading. In *SIGGRAPH 78 Conference Proceedings*, pages 282–285. Association for Computing Machinery, Inc., August 1978.
- [20] Silicon Graphics Inc. libsphere. online man page.
- [21] J. Staudhammer. On display of space filling atomic models in real-time. In *SIGGRAPH 78 Conference Proceedings*, pages 167–172. Association for Computing Machinery, Inc., August 1978.
- [22] Thinking Machines Corporation. The Connection Machine CM-5 technical summary, 1991.
- [23] Thinking Machines Corporation. *Render reference manual for Paris, 1991.