Binary-Swap Volumetric Rendering on the T3D

Chuck Hansen, Michael Krogh, James Painter
Advanced Computing Laboratory
Los Alamos National Laboratory
Los Alamos, New Mexico 87545

Guillaume Colin de Verdière*
Centre d'Etudes de Limeil-Valenton
CEL-V/DMA/AIM
94195 Villeneuve-Saint-Georges, France

Roy Troutman
National Energy Research Supercomputer Center
Lawrence Livermore National Lab
Livermore, CA 94550

Abstract

This paper presents a data distributed parallel raytraced volume rendering algorithm and its implementation on the CRI T3D. This algorithm distributes the data and the computational load to individual processing units to achieve fast and high-quality rendering of high-resolution data. The volume data, once distributed, is left intact. The processing nodes perform local raytracing of their subvolume concurrently. No communication between processing units is needed during this local ray-tracing process. A subimage is generated by each processing unit and the final image is obtained by compositing subimages in the proper order by the Binary-Swap algorithm. Performance of this algorithm on the T3D is presented and compared to an implementation on the CM-5.

1 Introduction

The visualization of 3D scalar data sets has proven an enormous boon for scientists. 3D scalar data sets arise not only from scientific simulations but also from high resolution scanners used for medical imaging and non-invasive testing. As instrument technology improves, the resolution of these scanners increases. Likewise, as the memory capacity of massively parallel computers increases, we are seeing the resolution of scientific models also increase. These advances lead to a tremendous data visualization problem: how can one interactively explore these very large data sets?

There are many different methods for exploring 3D data sets. Among these are slicing and dicing to form 2D data sets and utilizing the plethora of existing 2D visualization tools. The use of 3D computer graphics techniques is another very useful way to convey the important information contained within data sets. Direct volume rendering has proven to be an effective method for examining such data sets. However, the direct volume rendering is very computationally intensive and therefore often fails to achieve interactive rendering rates. This is compounded by the large data set size often seen by modern simulations on massively parallel computers or from high-resolution scanning devices. For large datasets[1], the computation requirements are such that single processor workstations are incapable of real-time volumetric rendering. In addition to the intense computational requirements, the need for high memory bandwidth and fast I/O rates predominate with this large amount of data. One solution is to utilize massively parallel processors (MPP) where possible. For sites which already have MPPs, this provides a cost-effective solution for volume rendering.

In this paper, we describe the CRI-T3D implementation of a volume renderer which was initially developed for a Thinking Machines Corporation CM-5. This volume renderer performs a data-space decomposition by dividing the 3D scalar field among processing elements (PEs). Once distributed in a viewpoint independent fashion, the volume can be rendered by employing a local renderer at each PE and compositing the results. We use a method which we call *Binary-Swap Compositing* which keeps all PEs active during the compositing phase. We provide explanations of design choices made during the T3D algorithm devel-

*Author currently at the ACL through a grant from DGA/DRET

[1] we consider large to be upward from 256^3 floating point data: 67MBytes

opment and describe performance results while comparing these to an implementation on the CM-5.

2 Related Work

There are two predominate methods for direct volume rendering: ray-casting and projection methods such as splatting or shear/warp [8, 9, 15]. Ray casting refers to *casting* multiple rays, typically one from each pixel, through the data set and performing interpolation at sample points along the ray. These samples are summed to obtain the color/transparency at each pixel. Projection methods directly project the scalar elements onto the image. These can be a polygonal representation or the element convolved with a function. The projections are done in a front-to-back or back-to-front fashion to obtain the final image. Since volume rendering is an inherently parallel task, both methods can exploit parallelism for accelerated rendering. Indeed, we have seen an increasing number of parallel algorithms for fast volume rendering [1, 2, 5, 13]. Both methods for direct volume rendering can be parallelized through data space decomposition as well as image space schemes. Ray-casting methods typically parallelize through data space decomposition.

Hsu described a data distributed volume renderer which was designed for the SIMD massively parallel MP-1 [5]. He distributes the volume based upon a block decomposition. Each PE gets a portion of the data and renders a local subimage. The locally rendered subimage pixels, which he calls segments, are routed to the PE which is responsible for that pixel in the final image. There, the segment is stored until all segments making up a ray are gathered after which the final compositing can be accomplished.

Camahort uses a similar method with block decomposition of the volume with a volume renderer designed for the CM-5 [1]. However, the segments are passed from PE to PE, in a systolic fashion, as they cross sub-volume boundaries. As the segments exit the data volume, they form a final pixel in the resulting image since the compositing is done on the fly.

Johnson and Genetti describe a volume renderer implemented on a CRI T3D [6]. They distributed the data in slices over the PEs. The slices are then individually rendered and then composited in parallel resulting in a final image. Their algorithm differs from ours in both the data distribution and compositing phase. As Johnson points out, the local rendering algorithm employed is independent of the data distribution and the compositing. Thus, this software framework provides an excellent platform for exploring different rendering techniques.

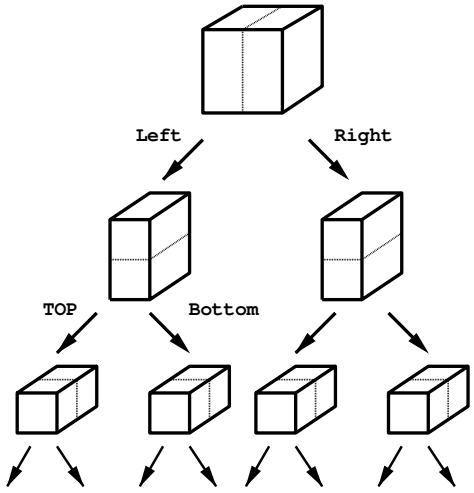

Figure 1: k-D Tree Subdivision of a Data Volume

3 Binary-Swap Volume Rendering

The algorithm described here has previously been published in two different papers [10, 11]. Here we describe a brief overview of the algorithm.

The basic idea behind our algorithm, like other similar methods, is very simple: divide the data up into smaller subvolumes distributed to multiple processors, render them separately and *locally*, and combine the resulting images in an incremental fashion. The memory demands on each processor are modest since each processor need only hold a subset of the total data set.

3.1 Data Subdivision

There are many ways to partition the data over the PEs with the only requirement being the ability to determine an unambiguous front-to-back ordering for the subvolumes. The distribution of the subvolumes determines a static load balancing of the data set. Ideally, we would like each subvolume to require about the same amount of computation. In practice, this is difficult to obtain since the amount of computation depends on the selected opacity transfer function. A common method is to partition the data into slices [6] or into blocks [5]. We use a k-D tree subdivision of the data. This performs alternating binary subdivision of the coordinate axes at each level in the tree as indicated in Figure 1. When the number of processors is

a power of two, the volume is divided equally among all three dimensions like the block subdivision. The advantage of the k-D tree is the hierarchical structure which is advantageous for image composition.

Once the volume is subdivided, it is useful to replicate the boundary of points along each face of the subvolume. This avoids message traffic for computing the gradients and interpolation operations.

3.2 Parallel Rendering

The rendering phase of our algorithm is based upon volume ray-casting as described by Levoy [9]. An image is constructed by casting rays from the eye through the image plane and into the volume of data. One ray per pixel is generally sufficient though we allow casting multiple rays per pixel. The 3D scalar field is sampled at evenly spaced points along the ray, usually at a rate of one or two samples per voxel. The volume data is interpolated to these sample points typically using a trilinear interpolant. Color and opacity are determined by applying a transfer function to the values. Shading can be accomplished by approximating the surface normal with the gradient of the data volume.

Sampling continues until the data volume is exhausted or until the accumulated opacity reaches a threshold cut-off value. The final image value corresponding to each ray is formed by compositing, front-to-back, the colors and opacities of the sample points along the ray. The color/opacity compositing is based on Porter and Duff's **over** operator [14]. It is easy to verify that the **over** is *associative*; that is,

$$a \textbf{ over } (b \textbf{ over } c) = (a \textbf{ over } b) \textbf{ over } c.$$

The associativity of the **over** operator allows us to break a ray up into segments, process the sampling and compositing of each segment independently, and combine the results from each segment via a final compositing step. This is the basis for our parallel volume rendering algorithm as well as recent methods by other authors [1, 5, 12, 13].

Local rendering is performed on each processor independently; that is, there is no data communication required during the subvolume rendering. All subvolumes are rendered using an identical view position and only rays within the image region covering the corresponding subvolume are cast and sampled.

In principle, any volume rendering algorithm could be used for local rendering. We have implemented several different ray-casting algorithms for local rendering. The slowest, yet most accurate, is based upon Phong shading. The gradient is tri-linearly interpolated along with the data value. These are used for

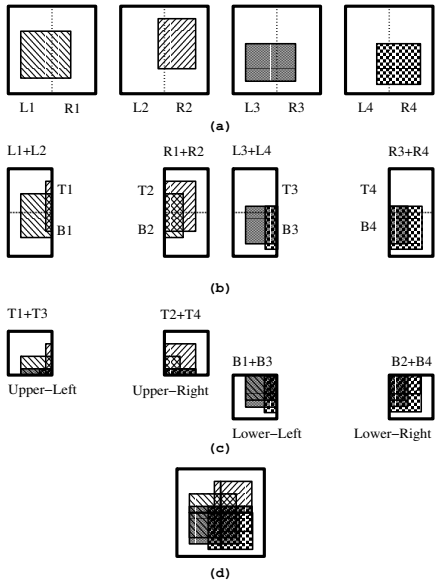

Figure 2: Parallel Compositing Process

shading at each sample point. The fastest method preshades the volume and interpolates the shaded intensity and data value. This saves two tri-linear interpolations per sample with minimal image degradation.

3.3 Image Composition

The final step in our algorithm is to merge the local images into a final image. As described earlier, the rule for merging is based on the **over** compositing operator. When all subimages are ready, they are composited in a front-to-back order. For a straightforward one-dimensional data partition, this order is also straightforward. When using the k-D tree structure, this front-to-back image compositing order can then be determined hierarchically by a recursive traversal of the k-D tree structure, visiting the "front" child before the "back" child. This is similar to well known front-to-back traversals of BSP-trees [4]. In addition, the hierarchical structure provides a natural way to accomplish the compositing in parallel: sibling nodes in the tree may be processed concurrently.

The compositing scheme, which we call *Binary-Swap*, fully parallelizes the compositing phase. At each compositing stage, the two processors involved in a composite operation split the image plane into two pieces and each processor takes responsibility for one of the two pieces. In the early phases, each processor is responsible for a large portion of the image area. In later phases as we move up the compositing tree, the processors are responsible for a smaller and

smaller portion of the image.

Figure 2 illustrates the *Binary-Swap* compositing algorithm graphically for four processors. When all four processors finish rendering locally, each processor holds a partial image, as depicted in (a). Each partial image is subdivided into two half-images by splitting along the X axis. In our example, as shown in (b), Processor 1 keeps only the left half-image and sends its right half-image to its immediate-right sibling, which is Processor 2. Conversely, Processor 2 keeps its right half-image, and sends its left half-image to Processor 1. Both processors then composite the half image they keep with the half image they receive. A similar exchange and compositing of partial images is done between Processor 3 and 4.

The key thing to note is that the Binary-Swap algorithm sends more data than other parallel compositing algorithms but can exploit the fast interconnection network of MPPs [11].

4 T3D Implementation

The focus of this research was to implement a fast, hopefully interactive, volume renderer on the CRI T3D. We will not describe the T3D architecture in this paper. The reader is referred to the wealth of documentation from CRI and other CUG papers[6, 3].

As noted, the initial algorithm was developed for a Thinking Machines CM-5. As one might imagine, the hardware/software environment of the T3D is different enough from the CM-5 to require substantial algorithmic changes. In this section, we describe what was required for the CRI T3D implementation.

The original publication of the Binary-Swap algorithm described not only the CM-5 implementation but also an implementation on a cluster of workstations under PVM [10]. This was the starting point for the T3D implementation since the T3D has a PVM environment. Like the CM-5, the T3D can be viewed as a host-node message passing machine with the YMP serving as the host and the T3D serving as the nodes. The steps of the algorithm are as shown in Figure 3.

The main functions, as described in the previous section, are steps 3 through 9. Several of these steps needed to be modified to obtain high performance on the T3D.

One of the initial algorithmic modifications involved how the image was transferred from the nodes to the host. With the CM-5, each individual PE sent the sub-image to the host for display. This was ideal for the CM-5 yet the worst possible scheme for the T3D. When any node on the T3D communicates with

	HOST	NODE
1	send data	
2		receive data
3	send view and transfer function	
4		receive view and transfer function
5		render sub image
6		composite sub images
7		send result to host
8	receive image	
9	display image	
10	repeat steps 3 to 9	

Figure 3: Host and Node Algorithm Steps

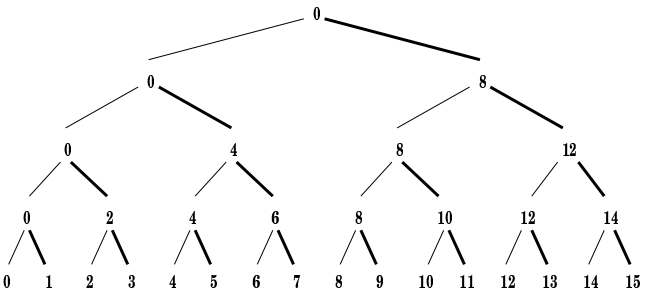

Figure 4: 16 Node Broadcast Tree Localizing Communication

the YMP, agents on the YMP-side field the request. Therefore, as each PE sent its subimage, the load on the YMP went from quiescent to overloaded as the agents fought for YMP resources to field the numerous simultaneous requests. This lead to a tremendous degradation of performance of the renderer. A better method is for a single PE to communicate with the YMP.

To accomplish this, we perform a tree based gather of the sub images to PE0. A broadcast tree, shown in Figure 4, is traversed in the reverse order copying the sub-image to the parent at each level. Data copies are performed only on the links indicated by the bold lines. Once we have the final image in the PE0 memory, we can either send it to the YMP through PVM, use the /proc method for transferring the image from a PE to the YMP memory[2], or display directly with HIPPI by writing to the proper device. By utilizing

[2]This involves having a process on the PE to write into the memory of a process on the YMP by writing to the /proc filesystem. Details can be found in the CRI documentation.

/proc, or an asynchronous write to HIPPI, and sending the next viewing transformation and transfer function before displaying the image, we can overlap the rendering with the image display. In the host-node diagram, step 9 is moved to step 4 on the host-side.

Sending the viewing transformation and transfer function to each PE requires a broadcast. On the CM-5, this broadcast was efficiently handled by the native message passing library. On the T3D, once again, we found that using the YMP as the host leads to performance problems when doing PVM style broadcasts. With the X11-based GUI, we are required to employ the YMP as the host since the T3D nodes do not support X11. In the naive approach, the YMP must communicate to each PE causing a bottleneck at the host. A better approach is to send the data to a single PE and let that PE broadcast to the others. A PVM broadcast from a single PE to the other PEs with a pvm_recv(-1) leads to an inefficiency. Since the YMP is part of the PVM group it still impacts the performance even though it does not participate in the broadcast and all communication takes place on the T3D. The solution is to specify the sending PE in the receive call. As seen in Table 1, this leads to a 7 fold performance increase when scaling to a large number of nodes. Additionally, we have found that for large messages the broadcast tree previously defined gives much better performance than the pvm_bcast command even if one uses a pvm_recv(PE0) which would seem to indicate that pvm_bcast is inefficiently implemented.

We also made some rendering enhancements to the original algorithm. The addition of supersampling allows for denser sampling along a ray which results in smoother images. As mentioned earlier since we were striving for interactive frame rates, we added a faster rendering algorithm which only shades at the data points rather than the sample points. Thus, the rendering algorithm then only needs to tri-linearly interpolate the data value and the shading information rather than the data volume and the three components of the gradient resulting in two rather than four trilinear interpolations per sample. Additionally, the illumination equation is modified to drop the specular term which saves a power-function call. We refer to this as Gouraud shading since the shaded value is interpolated. Conversely, by interpolating the normal and using that for local shading at the sample point, a superior image is generated at a greater computational expense.

The addition of an X11-interface for controlling the parameters and transfer functions assists in the interactivity of render. Since the PEs already have the data set in memory, the X-11 GUI impacts only step 3 in the diagram above, leading to interactive modification of the rendering parameters.

In addition to the X11-based interface, an AVS interface was created for the renderer. The user has the ability to modify parameters, such as the number of processors and the resolution through a set of AVS widgets. When the user connects the rendering module to the colormap generator and display tracker modules, the transfer function and the transformation matrix can be easily and intuitively manipulated with a mouse. The output from the rendering module is simply an AVS image. The output can be passed through additional modules to magnify the image through bilinear interpolation to create an image which is larger and easier to view without requiring extensive computation.

Although AVS is available for the YMP front end of the T3D, it was felt that the highly interactive nature of AVS made this a poor place to run the interface. As a result the slave processes of the renderer were started up remotely through PVM from an HP 9000 via a HIPPI connection. Since the HP and the Alpha both use IEEE floating point representation, in some cases reading in floating point numbers on the HP and transmitting them through PVM to the back end of the T3D is faster than reading in YMP floating point numbers, converting them to IEEE and then transmitting the results.

The AVS interface provides a local module, within the AVS framework, which uses PVM to interact with the T3D. Thus, the T3D volume rendering module looks just like any other AVS module. The disadvantage to this approach is that should another AVS module on the T3D interact with the volume rendering module, such as data preprocessing, the volume data would get passed from the first AVS module to the AVS-kernel on the HP9000 and back to the AVS volume rendering module since the T3D lacks the necessary AVS system libraries for remote module execution.

Remotely controlling the renderer through even a high speed network slows down the rendering process. Rendering data interactively was found to be advantageous, however. Interesting sections of the dataset can be found in only a few iterations. AVS also made interface design trivial.

method	2	4	8	16	32	64	128
pvm_bcast recv(-1)	0.0020	0.0035	0.0083	0.0170	0.0330	0.0901	0.2264
pvm_bcast recv(PE0)	0.0016	0.0023	0.0026	0.0066	0.0070	0.0164	0.0309

Table 1: Broadcast Times for on the T3D Without the YMP Sending/Receiving

128^3 256^2	2	4	8	16	32	64	128
update	0.0016	0.0023	0.0026	0.0066	0.0070	0.0164	0.0309
render	2.8678	1.4657	0.8556	0.4929	0.2490	0.1343	0.0747
composite	0.0463	0.0520	0.0607	0.0649	0.0673	0.0689	0.0705
128^3 512^2							
update	0.0016	0.0019	0.0028	0.0065	0.0124	0.0438	0.0464
render	11.4777	5.8508	3.4155	1.9905	1.0121	0.5293	0.2903
composite	0.1821	0.2213	0.2296	0.2546	0.2498	0.2500	0.2565

Table 2: Rendering Times in Secs. for 128^3 Data Set Rendered with the Phong Renderer and the Opaque Transfer Function

128^3 512^2	2	4	8	16	32	64	128
Phong	59.0891	29.5250	16.4777	11.0566	7.0924	4.0365	1.9779
Gouraud	33.2535	16.5976	8.6021	4.5614	2.3388	1.2653	0.6935
Speedup	1.777	1.779	1.915	2.424	3.032	3.19	2.85

Table 3: Difference Between Gouraud and Phong Shading for the Transparent Transfer Function

5 Experiments

In this section, we present performance results of the T3D Binary-Swap volume renderer and compare the performance to the Thinking Machines CM-5. All the times were gathered with optimization level **O3**. We ran several tests with different sized data volumes, 64^3, 128^3, and 256^3, of a MRI scanned head. Figure 5 shows the rendered data set with the transfer function set to isolate the skin. We also timed the rendering of different image sizes: 256x256, 512x512, 1280x1024. All tests were performed on a variety of partition sizes to determine scalability. We used two different transfer functions in these tests: opaque, as seen in Figure 5 and transparent. The transparent transfer function was chosen as a worst-case example since the rays never terminate early whereas the opaque transfer function terminates most rays early.

Figure 5: Human Head Data Set Rendered with Opaque Skin

Table 2 gives the time in seconds for the various rendering phases for a 128^3 volume at image resolutions of 256x256 and 512x512. Figure 6 shows a graph[3] of the sum of the rendering phases at various image sizes. The times are named with the convention: [machine].[volumesize].[imagesize].[transferfunction]. As described above, the opaque transfer function took advantage of early ray termination thus was much faster for a given image size. As one would expect, the larger pixel coverage slows down the renderer since more rays

[3] All graphs are given in loglog scales.

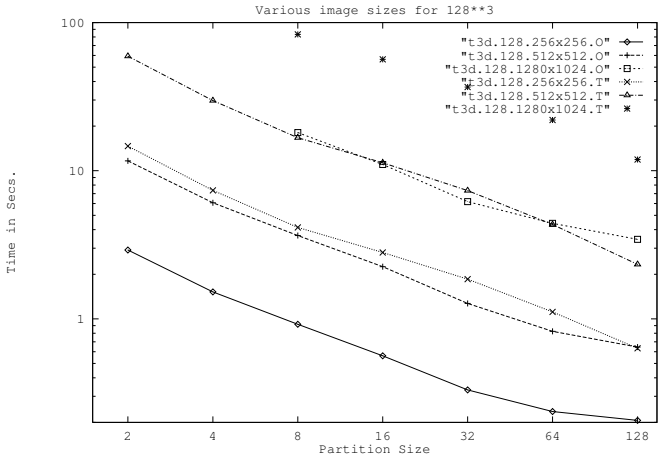

Figure 6: 128^3 Volume at Different Image Resolutions

are cast. The slight fluctuation in speedup is due to differing loads on the YMP.

Table 3 gives the rendering times of Phong shading and Gouraud shading for the transparent transfer function at an image resolution of 512x512. As one would expect, Gouraud shading is much faster; about 3 times faster for large numbers of nodes.

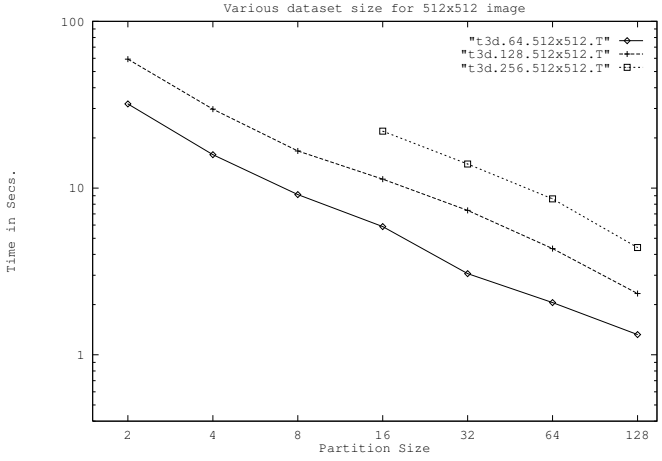

Figure 7: Effects of Scaling the Volume

The algorithm scales extremely well with the volume size as shown by Figure 7. The three curves are for volumes sizes of: 64^3, 128^3, and 256^3. The 256^3 volume would not fit on a partition smaller than 16 nodes. The scaling is roughly a factor of 2 for a factor of 8 increase in the volume size. The times are the sum of the rendering phases (update + render + composite) for the Phong renderer.

128^3 512^2	32	64	128
T3D			
Update	0.0124	0.0438	0.0464
Render	1.0121	0.5293	0.2903
Composite	0.2498	0.2500	0.2565
CM-5			
Update	0.0109	0.0106	0.0321
Render	13.4987	7.0119	3.8126
Composite	0.2523	0.2855	0.2091

Table 4: Phases of the Volume Renderer on the CM-5 and T3D

When running the volume renderer in interactive mode with a 128^3 data set rendering into a 256x256 image, we can achieve about 4 fps to the HIPPI frame buffer from a 128 node partition. The compositing and rendering take about the same amount of time.

Figure 8 shows the rendering time, for Phong shading, of the T3D compared to the CM-5. As can be seen in Table 4, the update and compositing phase were similar on the two machines while the rendering phase was an order of magnitude faster on the T3D. The primary reason for this is due to the problems of trying to access the vector units on the CM-5. A large portion of the CM-5 rendering kernel still runs on the Sparc-2 scalar processor which is much slower than the DEC Alpha processor.

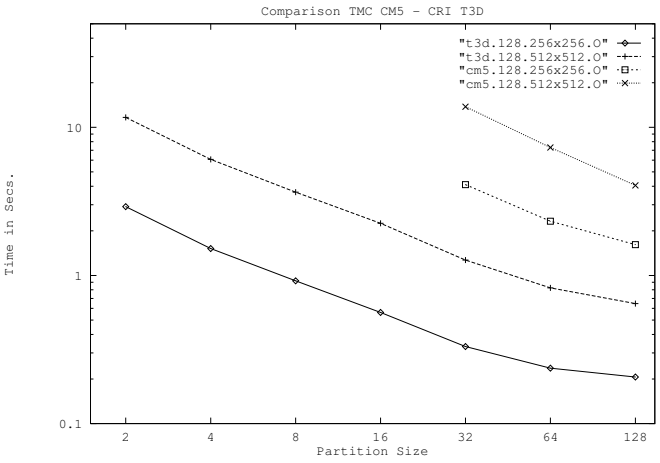

Figure 8: Rendering Times for the CM-5 and T3D

6 Conclusions

This paper presented an algorithm for fast volume rendering on a CRI T3D. The rendering framework allows for experimentation with different rendering techniques.

We believe that the performance of the algorithm can be improved in several ways. The message passing environment is definitely slow. If we move to shmem put/get then the communication should be dramatically improved. A different rendering algorithm would improve the interactivity of the volume renderer. Moving to a model which only uses the YMP for displaying the image would definitely help with performance. We anticipate exploring these as well as merging the volume renderer with a polygon renderer in the near future.

Perhaps because we are used to the MPP environment of the CM-5, we found that the programming environment on the T3D to be lacking in several aspects. The tools provided on the T3D were inadequate for large code development and did not scale well. In fact, we found that by debugging the PVM code on an SGI and moving the debugged version to the T3D, the development cycle was greatly enhanced. We were greatly disappointed to find that MPP tools, such as Apprentice, do not work with the C++ programming environment. The lack of robustness of the debugger, Totalview, meant that debugging on the T3D was a very painful experience[4].

The implementation of the PVM library on the T3D is less than optimal. For example, the PVM reduce instruction scales linearly rather than logarithmically as one would expect. The performance of PVM on the T3D nodes requires careful programming so as not to involve the YMP. Also, the black-art of the MPP environment variables is less than satisfying. In some cases, one must *tune* these variables just to get the program to run which we find unacceptable. We would like to see CRI support a *fast* native message passing environment such as **FM**[7].

On the positive side, the data network is much faster than on other MPPs; most notably the CM-5. Also, floating point performance was easier to obtain with the DEC Alpha processors than with the CM-5 vector units when programming in MIMD model. In general, vector unit memory management problems are traded for cacheing problems on the T3D. However for the volume rendering code the cacheing problems impact performance far less than the vector-unit

[4] We have been informed that the next release of Totalview should address the stability problems.

management on the CM-5.

The T3D is still an early massively parallel processor. As such, we expect the software environment to greatly improve over the next year.

7 Acknowledgements

We would like the thank the Advanced Computing Laboratory for providing an outstanding environment for performing this research. We also thank DGA/DRET for providing the grant which has allowed Guillaume Colin de Verdière to spend a year with our group. David Rich, as always, provided good response to our T3D problems as well as critical comments on this paper. Mark Dalton, CRI Los Alamos, listened to many complaints with good humor and provided much needed assistance time and again.

References

[1] Emilio Camahort and Indranil Chakravarty. Integrating Volume Data Analysis and Rendering on Distributed Memory Architectures. In *Proceedings 1993 Parallel Rendering Symposium*, pages 89–96, 1993.

[2] Brian Corrie and Paul Mackerras. Parallel Volume Rendering and Data Coherence. In *Proceedings 1993 Parallel Rendering Symposium*, pages 23–26, 1993.

[3] Jerry Delapp David Rich and Stephen Pope. Accepting the T3D. In *Proceedings of '94 Fall CUG*, pages 225–233, 1994.

[4] H. Fuchs, G.D. Abram, and E. D. Grant. Near Real-Time Shade Display of Rigid Objects. In *Proceedings of SIGGRAPH 1983*, pages 65–72, 1983.

[5] William M. Hsu. Segmented Ray Casting for Data Parallel Volume Rendering. In *Proceedings 1993 Parallel Rendering Symposium*, pages 7–14, 1993.

[6] Greg Johnson and Jon Genetti. High Resolution Interactive Volume Rendering on the Cray T3D. In *Proceedings of '94 Fall CUG*, pages 119–126, 1994.

[7] Vijay Karamcheti and Andrew Chien. A Comparison of Architectural Support for Messaging on the TMC CM-5 and the Cray T3D. In *Proceedings of ISCA*, 1995.

[8] Philippe Lacroute and Mark Levoy. Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation. In *Proceedings of SIGGRAPH 1994*, pages 451–458, 1994.

[9] Marc Levoy. Display of Surfaces from Volume Data. *IEEE Computer Graphics and Applications*, pages 29–37, May 1988.

[10] K.L. Ma, J.S. Painter, C.D. Hansen, and M.F. Krogh. A Data Distributed, Parallel Algorithm for Ray-Traced Volume Rendering. In *Proceedings 1993 Parallel Rendering Symposium*, pages 15–22, 1993.

[11] K.L. Ma, J.S. Painter, C.D. Hansen, and M.F. Krogh. Parallel Volume Rendering using Binary-Swap Compositing. *IEEE Comput. Graphics and Appl.*, 14(4):59–68, July 1993.

[12] Kwan-Liu Ma and Jamie S Painter. Parallel Volume Visualization on Workstations. *Computers and Graphics*, 17(1), 1993.

[13] Ulrich Neumann. Parallel Volume-Rendering Algorithm Performance on Mesh-Connected Multicomputers. In *Proceedings 1993 Parallel Rendering Symposium*, pages 97–104, 1993.

[14] T. Porter and T. Duff. Compositing Digital Images. *Computer Graphics (Proceedings of SIGGRAPH 1984)*, 18(3):253–259, July 1984.

[15] Lee Westover. Footprint Evaluation for Volume Rendering. In *Proceedings of SIGGRAPH 1990*, pages 367–376, 1990.