Volume Rendering of Scientific Data on the T3D

Scott Whitman, Cray Research, Inc., c/o JPL, Pasadena, CA USA and *Peggy Li* and *James Tsiao*, Jet Propulsion Laboratory, Pasadena, CA USA

ABSTRACT: *We discuss a direct volume rendering system for regular structured data implemented on the T3D. This paper discusses how to efficiently deal with a large volume dataset from an integrated scientific application, in this case, a global climate modeling system. Because of the large size volume, workstations are inherently unsuitable for storing even one volume let alone a series of time steps. Using the T3D SHMEM library, we were able to make use of asynchronous data transfer and employ communications overlap with computation. In addition, using an object space parallel decomposition of the dataset allows the algorithm to reduce cache misses on access to the volume data. The system is currently being tested and implemented on the T3D at JPL and we report on the results obtained.*

Introduction

Three-dimensional rendering is an increasingly important tool for the visualization of scientific data sets. Applications include visualization of the output from medical or scientific instruments such as seismographs, radars and magnetometers. Rendering is also widely used to visualize the data from numerical simulation models, such as molecular models, ocean and atmosphere models, etc. With the emergence of parallel supercomputers, the data volume generated from the supercomputer simulation models will overwhelm any workstation visualization tool. Therefore, it is important to create an interactive visualization tool capable of visualizing very large data sets and interacting with the scientific model in real-time where the data is resident, that is, on the supercomputer.

In this paper, we discuss a direct volume rendering system for regular structured data implemented on the Cray T3D. The system consists of a parallel volume rendering Application Programmers Interface (API) and an X/Motif-based Graphic User Interface (GUI). The API is similar to the OpenGL API both syntactically and semantically. It provides a suite of routines for viewing transformation, lighting, shading, and classification. It also has additional support for parallel processing, such as input data decomposition and output image compositing. The core of the API is a parallel direct volume renderer using the splatting approach. The renderer adapts both object space decomposition and image space decomposition for more efficient, better load-balanced and more scalable computation. The goal of this project is to provide an interactive visualization system which is capable of rendering very large 3D, 4D, or even 5D datasets produced by supercomputer-based scientific modeling applications; here 4D means 3-D in space and 1-D in time and 5D means 4D data with multiple parameters per data point. The rendering API may be incorporated into application programs to produce image products or real-time display. The advantages of the API are two fold: 1) It helps debugging and verifying the correctness of the model thus reducing the model building and development time; 2) It converts large volumes of raw model data into condensed image format which saves both storage space and data transfer time.

The X/Motif-based GUI is designed to run on a user's local workstation with a network interface to the parallel renderer on a remote machine. Our eventual goal is to combine scalable parallel rendering, efficient network interface, and friendly user interface to provide a seamless environment for distributed interactive visualization at the scientist’s desktop. In this paper, we describe the design and the implementation of the parallel volume renderer on the T3D and some preliminary performance results. We also discuss our plans for further functional and performance enhancement.

Background

In recent years, there has been a fair amount of work in parallel volume rendering with the research equally divided among the three main sequential algorithms: shearing, splatting, and ray casting. Shearing is most unlike the other two in that it is an image warping technique so we will focus on previous work that is most relevant to the rendering technique employed herein, splatting. Westover, the principal architect of the splatting algorithm, originated a design for a parallel splatter [Westover90,Westover91]. He analyzed functional as well as data decompositions, although his target architecture was a small scale multiprocessor. This early work formed the basis for later improvements.

Elvins [Elvins92] was the first to use the approach like Westover for splatting on a general purpose multiprocessor, the NCube machine. The idea here was to partition individual slices across the processors in an interleaved fashion to facilitate load balancing. Still, depending on the number of sheets available, some processors might be too lightly or heavily loaded. A master processor would then collect the image and send it to the host for output. Elvins did not explore any load balancing techniques nor any other decomposition methods.

Neumann [Neumann93] analyzed and implemented all three algorithms and concluded that splatting was consistently the fastest of the three. He derived a taxonomy and analyzed approaches as to where it was appropriate to decompose the domain object space vs. image space as well as whether it was suitable to group the data contiguously vs. interleaved or use task assignment to processors statically vs. dynamically. He concluded that an object space partition using block data regions (rather than slabs) uses the least amount of communication. This is the method we have chosen to implement here.

State et al. [State95] use a ray casting renderer combined with parallel compositing engines for implementation on the UNC Pixel Planes 5 system. While our design is similar to theirs, we do not have the luxury or limitations of this graphics hardware and we do not need to functionally partition the algorithm into different sets of processors as they do.

In the previous Spring CUG proceedings, two parallel algorithms were presented. Johnson and Genetti [Johnson95] distribute slabs of data to each processor where each slab is a contiguous set of slices. Each processor renders its local data and parallel compositing is done by constructing an image manipulation pipeline where portions of the image (contiguous sets of scan lines) are passed round robin amongst the processors. Our algorithm is similar in concept to this one in the data decomposition phase but we improve on the decomposition of the data to allow better load balancing and our final image composition is a more sophisticated technique with less communication and work. Hansen et al. [Hansen95] describe a ray caster with binary swap compositing. As in the previous approach, all processors perform compositing but less work is done than in Johnson's algorithm because the compositing is built up in stages over a tree and smaller regions are done at the lower stages. This algorithm works well but requires more communication and synchronization than our approach.

Interface and Data Input

The input files are in Network Common Data Form (netCDF) [Rew93] format. NetCDF is developed and maintained by Unidata, a National Science Foundation sponsored program. NetCDF functions as an I/O library callable from C or Fortran. It is designed to be machine independent and self describing for storing and retrieving scientific data. We chose netCDF as our data format because of the machine independent nature and that netCDF is well known in the scientific computing community.

The GUI to the renderer is designed to run on a remote workstation capable of X. The GUI provides a user-friendly, interactive environment to view and control the renderer. The GUI running on the workstation communicates with the renderer on the T3D via a TCP socket. A networking layer on the T3D manages the communications between the GUI and the renderer. The GUI issues simple ASCII commands to the T3D, and the networking layer calls the renderer using the renderer's API. This layer also retrieves image data from the renderer and sends it to the GUI. The GUI supports both 8-bit and 24-bit displays and the network interface supports real-time display via either a low-speed or a high-speed network. It is equipped with multiple control panels for interactive control of various rendering parameters. It is not only used as a GUI to the renderer, it will also control the interaction and execution of the model program when the model and the renderer are running together.

The ASCII commands used by the GUI are simple one word commands with optional parameters. The networking layer on the T3D parses each command and issues the renderer API calls necessary to accomplish the requested task. The networking interface also reads the netCDF data files and passes the data to the renderer. The GUI can also request information from the T3D, such as the dimensions of the data and the types of variables available. The networking layer sends this "meta" data back to the GUI on request.

When the renderer completes an image, it has the option of saving to file, passing the image back to the GUI, or send it to a HiPPI frame buffer. The user can select the different options from the GUI. The image output file format currently is the Portable Pixmap (PPM) format although support for others may be added in time. If the user requests the image to be sent back to the GUI, the network layer will send a compressed image back to the GUI, and the GUI will convert the image to an X image. Finally, the user can specify a HiPPI frame buffer, and the network layer will send the image to the specified buffer.

Algorithm Description

In this section we describe the volume renderer and in particular the splatting algorithm. We then illustrate the enhancements to the renderer to support parallelism.

Sequential Algorithm

Splatting, as defined by Westover, is a feed-forward voxel projection algorithm in which a filter is applied to each voxel and the resultant filtered voxel is projected onto the screen. The overview of the sequential volume renderer is provided in Figure 1. The reader is referred to Westover's paper for an excellent overview of the actual splatting process. Basically, the voxels are processed in sheets, with a sheet being the set of voxels in a 2-d grid of the rectilinear volume which is most orthogonal to the viewing direction at the current time. The sheets can be

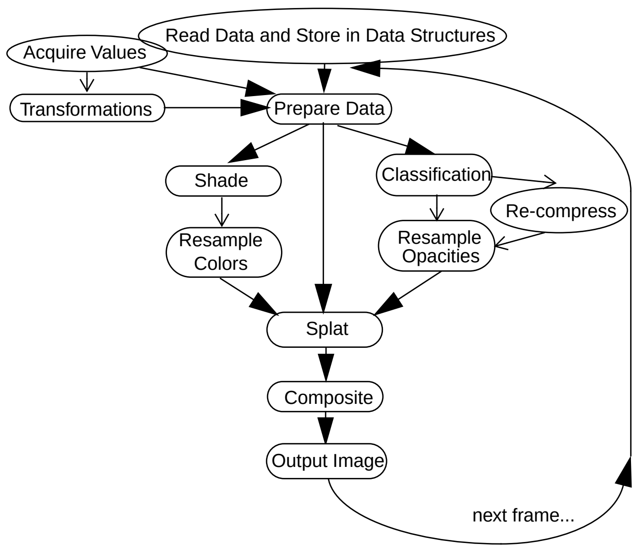


Figure 1: Overview of Sequential Algorithm

processed either front to back or back to front but the order must be consistent. Each voxel in a sheet is projected to the screen and its contribution to the underlying pixels is summed according to the applied filter. Iso-surface extraction can be accomplished using Marc Levoy's algorithm [Levoy88].

Figure 2 illustrates the splatting process. After each sheet is processed, it is composited with an accumulation buffer which retains the information of the sheets behind (or in front if we are going front to back). The compositing process is associative so the order (front-to-back or back-to-front) is unimportant as long as proper depth is maintained. Figure 3 illustrates the compositing process.

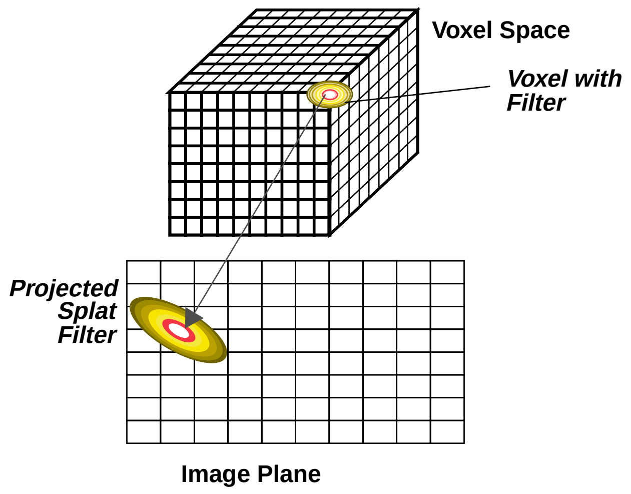


Figure 2: Overview of splatting process

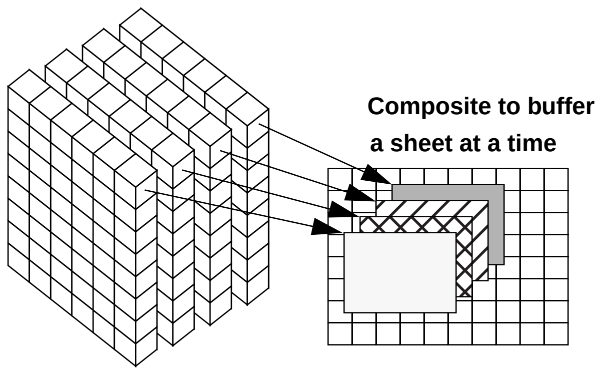


Figure 3: Compositing into sheet buffers

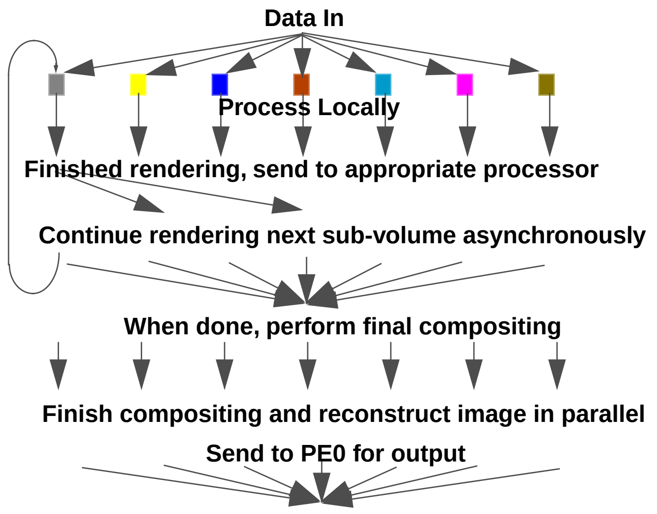
Parallel Algorithm

The parallel algorithm is very similar to the sequential algorithm with the following changes. First, each processor receives a block portion of the volume and renders that part locally, splatting and compositing the image. This creates an image of a portion of the volume. This image is then sent to the appropriate processor for the final compositing where all images are collected that project to that portion of the screen. That is, the rendered images in the first part will overlap each other in screen space based on the average depth complexity of the sub-volumes. These rendered images are then composited and reconstructed into a final image for display. In figure "Overview of parallel algorithm" on page 167, we illustrate this process.

In the next section, we describe the individual data decomposition schemes outlined here.

Object Space Decomposition

Each processor has 1/P portion of the input data volume, where P is the number of processors. However, to afford load balancing, we allow each processor to have multiple (R) sub-volumes per processor, interleaved over the entire volume.



Data In
Process Locally
Finished rendering, send to appropriate processor
Continue rendering next sub-volume asynchronously
When done, perform final compositing
Finish compositing and reconstruct image in parallel
Send to PE0 for output

Figure 4: Overview of parallel algorithm

Since the work in rendering is not evenly divided among all the voxels (because of classification), this allows minimal, albeit static, load balancing. Figure 5 illustrates the object space decomposition for the UNC head data set. We expect to work on more sophisticated techniques in the future. Karia [Karia94] implemented a very similar data decomposition on the Fujitsu AP1000 in a parallel ray casting modification of the Hansen et al. approach. He claims almost a 50% improvement over a decomposition that does not support multiple interleaved sub-volumes per processor. We have not specifically analyzed the performance improvement in our algorithm. Our expectations are high that it will provide enhanced performance which we will report on in detail in the future.

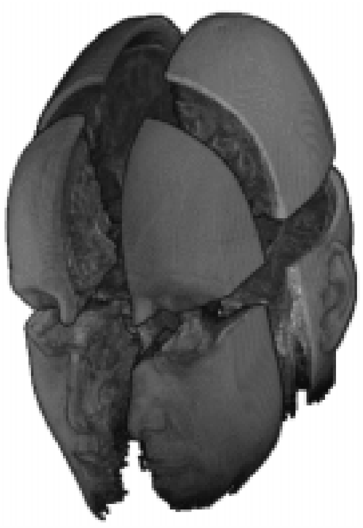


Figure 5: Object space decomposition

Neumann's [Neumann93] approach to load balancing involves copying additional data to each processor and then adjusting the partition boundaries between processors. The adjustment of the boundaries is based on the rendering times computed in the previous frame for all processors. If the difference between a given processor's time is lower then the average, it's boundary is adjusted to give it a larger sub-volume while the opposite is true if it took greater than average in the previous frame. This algorithm is an alternative to the approach we have chosen for object space load balancing. It could be applied to our approach and we may do so in the future.

Image Space Decomposition

After each processor has rendered a sub-volume, it has a projected image which may overlap other sub-volume images in depth. As such, a final compositing of these images needs to take place. In order to evenly distribute the work load in this phase, the image space is partitioned among the processors and is multiply interleaved so that each processor receives some number of screen space areas. Algorithms developed in the past required that communication for final compositing be performed after all rendering was completed. The reasons were: a) no buffering is needed of the images on the remote processor, b) it is suitable for message passing and c) with binary swap or image partition compositing, all processors are involved in the compositing operation.

As shown in Figure 6, the final compositing phase receives data from processors after they render their local sub-volume. Here we describe how this is accomplished. The three processors on the left have created three images shown. Each portion of the image overlaps some portion of a processor's final compositing area, shown in outlined regions here. The idea is that the single sub-volume rendered image is then broken into components which are then sent to the appropriate processors for final compositing.

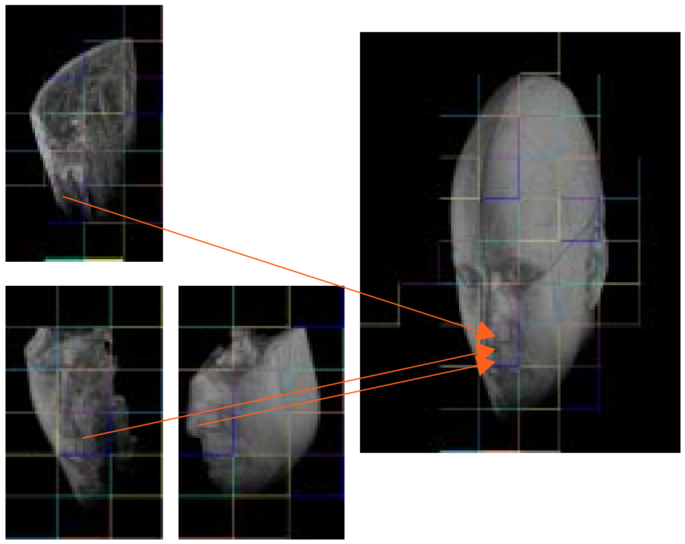


Figure 6: Final compositing process

Communication

As soon as a processor is finished rendering a sub-volume, it may send the individual final compositing regions off to the appropriate processors and then proceed to render the next sub-volume. We save memory and communication by having each processor wrap a bounding box around the sub-image and only send the portion of the image which is actually relevant (that is, no blank pixels if possible). This is accomplished using the Cray SHMEM library with the ***shmem_put*** call. Shmem_put allows the programmer to send data to a remote processor without the processor having to receive it. All that is necessary is that a memory location be set aside apriori. However, there is a complication if more than one processor needs to send data to a given PE simultaneously. This is remedied in the following manner. Each processor maintains a pointer to the buffer in which others will *put* into its memory. The pointer is atomically updated by a processor which is sending the data. This is accomplished by adding the exact amount of data to be sent to the address and putting the updated address back. The ***shmem_swap*** routine facilitates this atomically. In fact, this is the only instance where a message passing implementation would have to differ from our implementation. The data itself can be put anytime and even potentially out of order from another processor since the location to put the data is secured.

This one sided communication is very fast and since it involves no overhead on the other processor, communication can be overlapped with computation. In fact, because the image data is not even used until much later, no cache coherency need be maintained while the put is proceeding. It is certainly true that each processor will have to send to many others but since all-to-all communication will not be happening simultaneously, it will be spread out in time over the execution of the splatting renderer. The idea here was to make maximum use of load balancing by spreading out the compositing work and also spread out the communication in time and space so no single processor is a hot spot or bottleneck. Based on the performance analysis data in the next section, our current belief is that the overhead of doing many small communications (even spread out) is possibly too much and we expect to reduce this in a future version to a few larger communications.

Image Output

After all processors have done final compositing on their sub-images, we are now left with a disjoint set of images which must be reconstructed and sent out as a contiguous image. The contiguous image can either be sent to a HiPPI frame buffer, disk, or X window display. In any case, sending all the sub-images to PE0 would obviously result in a hot spot contention situation. Additionally, a pyramid reconstruction scheme requires synchronization at each level in the pyramid, which also reduces throughput. The scheme we devised is to have each processor responsible for a given contiguous group of scan lines, non-interleaved. As in the previous section, each processor will ***Put*** its appropriate data to the processors with the given set of scan lines for which the sub-image overlaps. This means all processors are communicating to all which can bottleneck the system. After all processors have sent their data, we end up with each processor having a contiguous set of scan lines. These are then ***Put*** to PE0 in the correct memory location and finally PE0 writes the image out.

Performance

Figure 7 indicates the speedup we measured. This is in reference to ***4*** processors as the baseline. Performance tailed off more rapidly than we expected. We explain this tail off below.

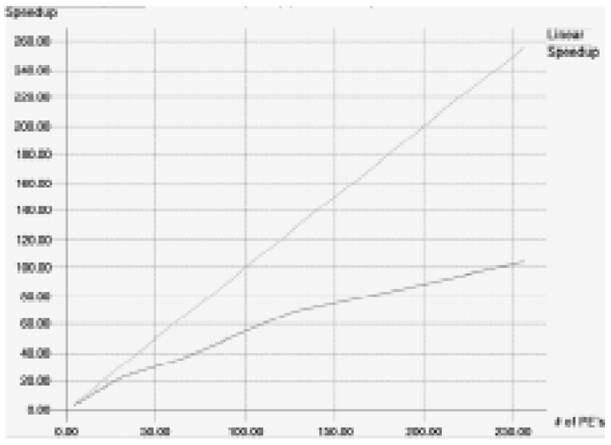


Figure 7: Speedup analysis

In table 1 we present a performance comparison for various overheads. These were measured as a function of 100%. It is interesting to note that at this point one can see that splatting dominates the computation at less than 64 processors. Our load imbalance for splatting added to the cost of splatting, particularly on the higher processor counts. The reason for this is the size of the regions with voxels is not necessarily constant across the processors. In addition, the classification scheme does not equally distribute the workload among all voxels. We expect to work more on the load balancing in the future. Setup cost is essentially constant for all processors which is why it becomes a higher percentage as the number of processors increases. This includes the time to construct the filter table and matrix for viewing. Shading and classifying are not included here since those are typically one time costs. This does give the effect, though, that the light source rotates with the object. For this reason, we have used two light sources on opposite sides of the object so that the effect is not objectionable.

# Processors	**4**	**16**	**32**	**64**	**128**	**256**
Setup %	0.4	1.4	2.6	4.1	8.2	13.6
Splat %	51	42	39.7	30	31	23.7
Splat Load Imbalance %	11.4	23.2	27.2	41.2	33.8	39.5
Local Compositing %	35	31	27.8	21.7	23.3	17.7
Final Compositing %	0.3	0.5	0.9	1.2	2.2	3.5
Final Compositing Load Imbalance %	0.05	0.05	0.06	0.05	0.04	0.02
Communication %	1.7	1.7	1.7	1.7	1.3	1.9

Table 1: Performance Analysis

Local compositing, that is, compositing done as part of the rendering of the local processor's sub-volume amounted to a fairly large portion of the time. The reason for this is that each sheet in each processor needs to be composited with every other sheet in that processor. While a front to back optimization does employ early termination if full opacity is reached, the overall work is not minimized significantly. Finally, the data here involved using an approach where the bounding box for the sheets are used for compositing. In our current enhancement, we extract only those pixels that are rendered and composite only these. There is some overhead with this approach and we have not yet determined if there is a significant benefit gained using this technique or not.

Final compositing was very small in proportion to the entire rendering. The reason here is that each processor only has to deal with several regions of the screen space and the depth complexity corresponding to those regions is very low (when compared to the number of sheets that are composited locally during the splatting phase). There is a synchronization point here which is included in the timings which explains why very little load imbalance is noticed. The synchronization point occurs after the image is finally composited before reconstruction and sendout to PE0. This is included since obviously an image distributed over the processors doesn't constitute a final rendering.

Communication was very small here as measured. This is an important point because the measurement of communication only included the time to actually make the call. We did not add any additional time to make sure the message was received since using the ***shmem*** mechanism did not require this nor did our algorithm. Thus, this is a perfect example where communication overlapped with computation can be a big win.

Conclusion

In this paper, we presented a parallel volume rendering algorithm using the splatting approach. In our implementation, we use both static object space decomposition and static image space decomposition to achieve load balancing. Our algorithm is a Sort-Last Sparse (SL-Sparse) algorithm based on Molnar's [Molnar94] classification given the fact that the sorting is done at the image space before the final compositing. We proposed an asynchronous image compositing scheme which can be fully overlapped with the splatting process thus hiding most of the communication cost. Although the early results showed poor load balance and increasing overhead as the number of processors increases, we believe the load imbalance will be improved by tuning the size of the input blocks and the way they are interleaved. Similarly, the communication overhead can be reduced by selecting a proper image region size for compositing.

Interactive frame rate is an essential requirement for our rendering system. We are still a distance away from achieving this goal. We are investigating various alternatives for better rendering speed: 1) performance tuning -- besides the load balancing and communication overhead problems addressed above, we will also look into ways to increase cache coherence and improve cache performance; 2) other filters -- with a little degradation of the image quality, we can replace the Gaussian filter with a less-expensive one for splatting, such as a box filter; 3) data compression -- data pyramiding may be used to represent a high resolution input volume and rendering may take place at a lower resolution which matches the resolution of the image space. On the other hand, data compression may be used to represent a sparse natured data set and unnecessary computation can be avoided by skipping through the invalid data points.

Time-varying datasets are often too large to fit into the memory of existing supercomputers. Therefore, off-the-core rendering is the key function in order to support visualization of time-varying datasets. It is an even bigger challenge to achieve interactive frame rate for time-varying datasets because data input and data classification are also accounted for in the rendering time. We will either optimize the input and data preprocessing time or hide the overhead by overlapping the operations in the rendering pipeline. This is an open research area that requires careful study and design. Other capabilities we are also looking into include the representation and visualization of vector fields, arbitrary clipping planes, multiple iso-surfaces and multiple parameters rendering.

Acknowledgment

The work presented in this paper is sponsored jointly by Cray Research Inc. PATP program and NASA High Performance Computing and Communication (HPCC) Program, Earth and Space Science (ESS) Project.

References

[Elvins92] T. Todd Elvins "Volume Rendering on a Distributed Memory Parallel Computer" *Proceedings of Visualization '92*, 1992.

[Hansen95] Chuck Hansen, Michael Krogh, James Painter, Guillaume Colin de Verdiere, and Roy Troutman "Binary-Swap Volumetric Rendering on the T3D", *CUG 1995 Spring Proceedings*, Denver, CO, pp. 61-69, March 1995.

[Johnson95] Greg Johnson and Jon Genetti "Medical Diagnosis using the Cray T3D," *CUG 1995 Spring Proceedings*, Denver, CO, pp. 70-77, March 1995.

[Karia94] Raju J. Karia "Load Balancing of Parallel Volume Rendering with Scattered Decomposition," *Proceedings of the 1994 Scalable High Performance Computing Conference*, Knoxville, TN, May 1994.

[Levoy88] Marc Levoy "Display of Surfaces from Volume Data," *IEEE Computer Graphics & Applications*, Vol. 8, No. 5, May 1988, pp. 29-37.

[Molnar94] Steven Molnar, Michael Cox, David Ellsworth and Henry Fuchs "A Sorting Classification of Parallel Rendering," *IEEE Computer Graphics & Applications*, Vol. 14, No. 4, July 1994, pp. 23-32.

[Neumann93] Ulrich Neumann "Volume Reconstruction and Parallel Rendering Algorithms: A Comparative Analysis," Ph.D. Dissertation, UNC Chapel Hill, 1993.

[Rew93] Rew, Davis, and Emmerson, NetCDF User's Guide: An Interface for Data Access, University Corporation for Atmospheric Research, 1993.

[State95] Andrie State, Jonathan McAllister, Ulrich Neumann, Hong Chen, Tim J. Cullip, David T. Chen, and Henry Fuchs "Interactive Volume Visualization on a Heterogeneous Message-Passing Multicomputer," *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, ACM Press, April 1995, pp. 69-74.

[Westover90] Lee Westover "Footprint Evaluation for Volume Rendering," *Computer Graphics, Proceedings of Siggraph*, Vol. 24, No. 4, August 1990, pp. 367-376.

[Westover91] Lee Alan Westover "Splatting: A Parallel, Feed-Forward Volume Rendering Algorithm," Ph.D. Dissertation, UNC Chapel Hill, 1991.