Parallel Rendering of 3D AMR Data on the SGI/Cray T3E

Kwan-Liu Ma
Institute for Computer Applications in Science and Engineering
Mail Stop 403, NASA Langley Research Center
Hampton, Virginia 23681-2199
Email: kma@icase.edu

Abstract

This paper describes work-in-progress on developing parallel visualization strategies for 3D Adaptive Mesh Refinement (AMR) data. AMR is a simple and powerful tool for modeling many important scientific and engineering problems. However, visualization tools for 3D AMR data are not generally available. Converting AMR data onto a uniform mesh would result in high storage requirements, and rendering the uniform-mesh data on an average graphics workstation can be painfully slow if not impossible. The adaptive nature of the embedded mesh demands sophisticated visualization calculations. In this work, we compare the performance and storage requirements of a parallel volume renderer for regular-mesh data with a new parallel renderer based on adaptive sampling. While both renderers can achieve interactive visualization, the new approach offers significant performance gains, as indicated by our experiments on the SGI/Cray T3E.

1 Introduction

Increasingly, leading-edge scientific computations with demanding memory and processing requirements are being performed on massively parallel (MP) supercomputers. The size of the resulting solution data is often too large if not impossible for an average graphics workstation to process efficiently for data visualization and analysis. To support applications which use these MP supercomputers, visualization tools appropriate to the parallel architecture must be developed such that the solution data can be studied at the highest possible resolution in an efficient manner. The challenge, then, is to develop algorithms and methodologies which can exploit the available parallelism to perform visualization and rendering operations on the parallel systems, and to deliver the results to the researcher's desktop in a timely and efficient manner.

Although there is a growing body of work in parallel graphics and visualization, only a small subset of this has been directed toward data on irregular, non-uniform, or unstructured grids. Furthermore, the challenges of the new problem sizes and the state-of-the-art computer architectures extend into regimes which have not been thoroughly explored by parallel visualization researchers.

This paper reports our experience and results from developing visualization strategies making use of MP supercomputers for Adaptive Mesh Refinement (AMR) data, in particular, the type of data generated from applications using the PARAMESH package developed at NASA's Goddard Space Flight Center [19]. AMR is a simple and powerful tool for modeling many important scientific and engineering problems. However, visualization tools for 3D AMR data are not generally available because the associated mesh contains multiresolution components. In this research, we proceed in the following three phases:

1. Modify an existing parallel volume rendering algorithm for regular-mesh data and port it to the T3E for visualizing AMR data. The main goal is to study the performance of the parallel rendering algorithm on the T3E, and to demonstrate interactive 3D visualization of the data without sacrificing accuracy and image quality. Our experience with this renderer also helps us estimate our potential capability to render AMR data with improved methods.

2. Develop a new parallel renderer that can render the AMR data directly without converting the adaptive mesh to a uniform mesh. Our goal is to reduce runtime memory requirements, rendering time, and thus the number of processors needed to achieve interactive visualization.

3. Develop support for runtime visualization. We plan to demonstrate runtime monitoring of a parallel numerical simulation created with the PARAMESH package. That is, visualization is generated in place on the same MP supercomputer where the simulation runs. To perform runtime visualization, we need to provide the user

with an interface to interact with the simulation, and to further reduce memory and processing requirements of the visualization calculations to make this approach attractive.

In this paper, we present our results from the first two phases on the SGI/Cray T3E using up to 256 processors. In Section 2, we provide a brief description of PARAMESH and the kind of adaptive meshes it can generate. In Section 3, we introduce volume rendering and highlight research issues in parallel volume rendering. Section 4 describes the preprocessing tasks specific to the kind of AMR data to be visualized . The new parallel rendering algorithm is introduced in Section 5, and Test results are presented in Section 6. In the final section, we discuss the implication of this work and direction of future research.

2 PARAMESH - Parallel AMR Code

PARAMESH [19] is a package of Fortran 90 subroutines designed to provide an application developer with an easy route to extend an existing serial code which uses a logically Cartesian structured mesh into a parallel code with adaptive mesh refinement. The package builds a hierarchy of sub-grids to cover the computational domain, with spatial resolution varying to satisfy the demands of the application. These sub-blocks form the nodes of a tree data-structure (quadtree in 2D or octree in 3D). Each grid block has a logically Cartesian mesh, and the index ranges are the same for every block. Thus, in 2D, if we begin with a 10×20 grid on one black covering the entire domain, the first refinement step would produce 4 child blocks, each with its own 10×20 mesh, but now with mesh spacing one-half that of its parent. Any or all of these children can themselves be refined, in the same manner. This process continues, until the domain is covered with a quilt-like pattern of blocks with the desired spatial resolution everywhere. During the refinement process, the refinement level is not allowed to jump by more than one level at any location in the spatial domain.

Figure 1 shows a simple example of a 2D mesh generated by PARAMESH and the corresponding quadtree. Note that each block contains exactly 4×4 cell-centered data points. More PARAMESH's examples can be found at *http://outside.gsfc.nasa.gov/ESS/amr.html.*

This type of grid poses some challenges to 3D visualization calculations. First, there is no existing 3D visualization tool which can handle data on an adaptive mesh directly. Efficient, direct visualization of the AMR data requires the development of new rendering algorithms. Second, during visualization calculations, the transition from a coarser resolution block to a finer black block (or vice versa) can causes serious aliasing artifacts if care is not taken. Third, the resulting simulation data for real-world application problems

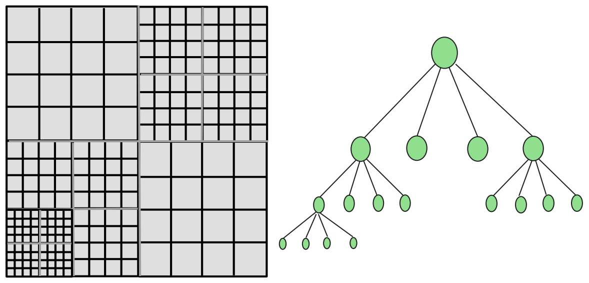

Figure 1. A 2D grid generated by PARAMESH and the corresponding quadtree.

are large, and to generate high quality visualization the runtime memory requirement can be so high that an average graphics workstation becomes useless.

3 Parallel Volume Rendering

Volume rendering is a very effective 3D visualization method because it can display more information in a single visualization than methods such as isosurface or slicing. In fact, we can also use volume rendering to produce isosurfaces or cut-planes, or a mixture of them. Most importantly, volume rendering is particularly effective for visualizing fine features and those features that cannot be defined analytically.

Volume rendering is computationally expensive due to the interpolation and shading calculations required for every sample point in the spatial domain of the data. The level of graphics hardware support presently available for volume rendering is still limited. The fastest software volume renderer takes at least tens of seconds to several minutes to produce a high resolution image of a regular volume data, say containing $256\times256\times256$ data points, on an average workstation. Many optimization techniques using preprocessing and special data structures have been proposed to speed up the rendering calculations [8, 11, 6] but for very large datasets interactive visualization is still unattainable.

Multiresolution representations, compression and feature extraction are plausible but they are not the complete solutions. As parallel computers become more accessible, parallel and distributed rendering seems to be the most promising solution which offers maximum flexibility without sacrificing accuracy and image quality. Parallel rendering incorporated with multiresolution representations, compression and/or, feature extraction techniques seems to be the ultimate solution for large data visualization.

Different parallel volume rendering algorithms have been developed for either shared memory [17, 2, 5, 18] or distributed memory parallel computers [22, 13, 15]; by using raycasting [13, 4, 15, 18], cell projection [24, 23, 12],

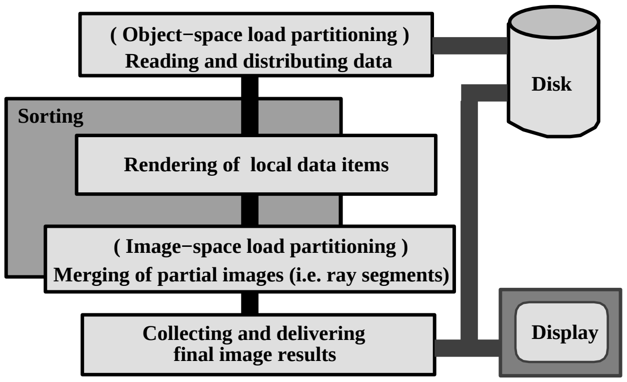
(Object−space load partitioning)
Reading and distributing data
Disk
Sorting
Rendering of local data items
(Image−space load partitioning)
Merging of partial images (i.e. ray segments)
Collecting and delivering final image results
Display

Figure 2. A basic parallel volume rendering algorithm.

shear-warp [1, 5, 21] or splatting algorithms [9, 16]; and for data on particular grid structures such as rectilinear [13, 3, 4], curvilinear [2, 23], unstructured [24, 10, 12] or hybrid grids [23]. Parallel rendering algorithms for adaptive-mesh data are absent.

3.1 Basic Steps

A basic parallel volume rendering algorithm can be illustrated by the diagram shown in Figure 2. The first step is to determine how the data is partitioned to ensure a statically balanced load or to facilitate dynamic load redistribution. Then the partitioned data are distributed to each processor. Data distribution in particular utilizing parallel I/O can impact the overall rendering efficiency. However, I/O issues are not discussed in this paper.

3.1.1 Data Partitioning

Data partitioning is a difficult task since there are many factors contributing to the partitioning strategy selected. For volume visualization, these factors include:

- Opacity transfer function.
- The number of data items.
- The shape and size of each data item.
- View.

The opacity transfer function essentially specifies what in the data should be made visible. For some types of data, the mapping from data values to opacities is straightforward. For example, intensity values of medical scanned data are usually in direct proportion to the opacity values used. On the other hand, finding the appropriate opacity transfer function for flow field data, is often an exploring process. Opacity transfer function influences load balancing because an opaque surface blocks the volume region behind and therefore waives the rendering of that region. Second, there is certain overhead in processing each data item so the number of data items assigned to a processor can affect load. Next, data items of different shapes and sizes require different computational loads. Finally, it is clear that viewing position affects the impact of all the other three factors. Because of these factors and the high overhead of using dynamic load balancing, simple static load balancing techniques that can achieve load balancing in general cases are desirable.

3.1.2 Local rendering

After data are distributed, each processor renders its local data items independently of other processors. A data item can be either a point, a cell or a slice. We call a grid point a voxel (volume element). A cell is usually composed of three to eight voxels. For example, a three-voxel cell is a tetrahedron and an eight-voxel cell is a hexahedron. A slice is a collection of voxels or cells on the same plane. The raycasting algorithms work on several voxels in the same neighborhood at a time. The projection or splatting algorithms work on one voxel or one cell a time. The algorithms based on factorization of the viewing transformation (i.e. shear-warp) work on one slice a time. The AMR data is block-based so we choose a raycasting algorithm [13] that have been studied extensively on other parallel architectures.

The local rendering step is essentially a sampling process based on the viewing transformation. At each sample point, a value is computed using trilinear interpolation of nearest voxels values. The value is mapped to color and opacity. Careful selections of colors and transparencies can often bring out very important features in the data. To obtain the final image, all color and opacity values calculated are composited according to the (front-to-back or back-to-front) depth order. Consequently, some form of sorting must take place before, during or after the local rendering step to derive the correct compositing order of the sampling results [14]. Front-to-back compositing is based on the *over* operator [20] which is:

$$\alpha_{accumulated} + = \alpha_{current\_sample\_pt} \times (1 - \alpha_{accumulated})$$

where α is opacity. Figure 3 shows ray-casting resampling of the data on a regular grid distributed to multiple processors. The local rendering performed by each processor produces many ray segments. Ray segments corresponding to the same pixel are composited with a global process to derive the final pixel value. The basis of a typical parallel volume rendering algorithm is that each ray may be broken into segments which can be computed by different processors concurrently.

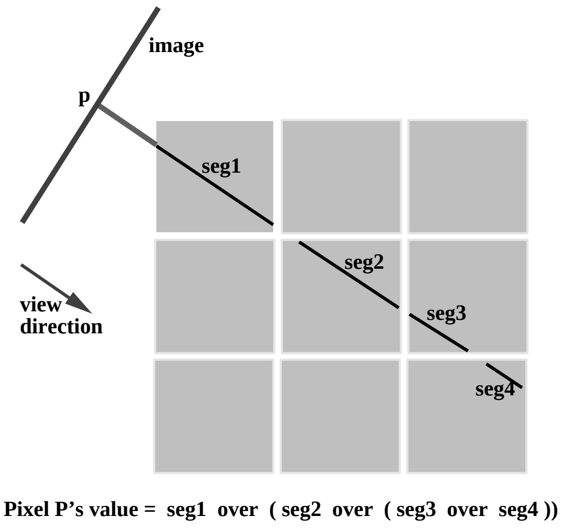

Figure 3. Parallel ray-casting resampling. Each subdomain is handled by a different processor. Ray segments corresponding to the same pixel are merged in a compositing step to derive the final pixel value.

3.1.3 Parallel image compositing

Following local rendering, all processors participate in a global process to composite the partial images generated into the final complete images for different image areas. Image-space load partitioning is as important as the object-space load partitioning to ensure overall rendering efficiency. Possible partitioning methods include pixel interleaving, block/strip interleaving, scanline interleaving and adaptive decomposition. Each method has different computational overhead and flexibility. In essence, each processor is responsible for a set of pixels and thus must allocate memory space to store color, opacity and depth values for each pixel. Therefore, the image space assignment should be done in such a way that overloading a few particular processors will never happen. This overloading problem is worst for highly adaptive mesh in which some very fine mesh structures occupies a relatively very small spatial region. We have found that in this situation a finer-level partitioning like pixel interleaving scales well with a large number of processors.

Parallel image compositing requires communication between processors. The order in which the compositing of ray segments is conducted is important. The simplest approach, which we call direct-send, is to have each processor send ray segments directly to the processor responsible for compositing them. This approach has been used by several researchers [4, 15, 10, 12] but it has trouble with link contention when many processors try to send ray segments to the same destination. Lee, et al. [7] proposed a parallel compositing algorithm to avoid link contention on a mesh network. For an $m \times n$ mesh, compositing is first performed along the column direction and then along the row direction (or vice versa). A total of $(m-1) \times (n-1)$ steps are carried out. In each step, every processor forwards a set of ray segments generated locally to the processor responsible for the corresponding image area. This compositing algorithm is essentially an optimized version of the direct-send method. The send is ordered to avoid link contention. As a result, it is a completely synchronous algorithm.

A binary method which pairs up processors in order of compositing can remove the link contention problem but at each phase of compositing, half of the processors would become idle. Then, at the top of the compositing tree, only one processor is active, doing the final composite for the entire image. In this way, when a massively parallel computer with a large number of processors is used, compositing would become a serious bottleneck. Ma et al. [13] developed a parallel image compositing algorithm called *binary-swap* to keep all processors busy during the whole course of the compositing. The key idea is that, 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. Consequently, as the compositing proceeds, the image each processor handles becomes smaller. We can therefore take advantage of nearest neighbor communication for some particular types of network (i.e. tree or hypercube) by swapping large images during earlier phases of the compositing between processors that are physically next to each other. For example, efficient binary swap compositing using a k-d tree structure was demonstrated on the CM5 [13]. Nevertheless, binary swap is also a completely synchronized process so it is not applicable to an asynchronous algorithm.

3.1.4 Image output

The end of the parallel compositing process typically leaves a subimage on each processor. Even though many parallel rendering algorithms have demonstrated impressive rendering rates, to support interactive visualization, an efficient mechanism is needed to assemble the subimages and deliver the final image to a display device or disk to keep up with the rendering rates. In addition to using high speed links and dedicated frame buffers, software techniques such as pipelining and compression should be used whenever appropriate.

4 Preprocessing AMR Data

In the AMR data sets we obtained for testing, blocks of data are organized hierarchically according to their spa-

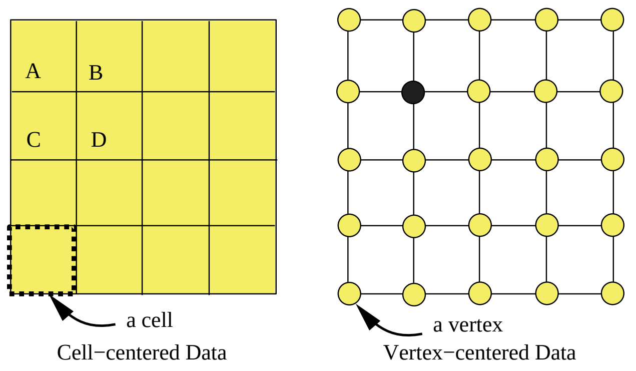

Figure 4. The original cell-centered data and the resulting vertex-centered data.

tial relationships. The data are floating point values and cell-centered. For each cell, eight variables are stored. To volume render such data, some preprocessing must be performed. First, we convert the cell-centered data to vertex-centered data by using linear interpolation. Figure 4 shows a 2D example of converting a 4×4 block of cell-centered data into a 5×5 block of vertex-centered data. The data value at the dark colored vertex (or node) is calculated by using the four cells (i.e. Cell A, B, C and D) sharing that vertex. For vertices at the block boundary, interpolation must be done between blocks. Note that in 3D cases, a vertex may be shared by as many as eight blocks, possibly with different resolutions. For a 3D dataset with N_b blocks of $n_x \times n_y \times n_z$ cells, this conversion generates $N_b \times (n_x + 1) \times (n_y + 1) \times (n_z + 1)$ vertices.

Next, we have the option of mapping the data onto a uniform mesh so we can utilize an existing volume renderer. Mapping the data onto a uniform mesh results in a significant increase in storage requirements. For a dataset with N_b blocks of $n_x \times n_y \times n_z$ vertices organized as an N_l-level octree, it takes $(2^{N_l+1})^3 \times sizeof(float)$ bytes rather than $n_x \times n_y \times n_z \times N_b \times sizeof(float)$ bytes to store one variable. In this work, we actually generated such uniformly sampled data so that a comparison can be made with direct rendering of the AMR data.

Finally, the floating-point data values are quantized to 8-bit values because the renderer typically uses a lookup table of 256 entries for color and opacity mapping in the rendering step. Consequently, this quantization step reduces the storage requirement by at least a factor of four.

5 Rendering AMR Data

To visualize the AMR data, we have taken two different approaches. The first approach is the simplest but also the most expensive one: we map the data onto a uniform mesh and then apply a parallel volume renderer to the uniform data [13]. Data are distributed among processors by evenly partitioning the spatial domain such that each processor handles a subdomain containing about the same number of voxels. The projected area of each subdomain is also approximately the same because of the uniform mesh. The renderer uses the binary-swap compositing algorithm for merging the ray segments [13]. The opacity function selected will determine the level of load imbalance.

The second approach is to render the AMR data directly. This requires the development of a new parallel rendering algorithm. While the development of the new renderer is still under way, its prototype version allows us to demonstrate the potential improvement we can achieve.

The new algorithm distributes the blocks of data in a round robin fashion to achieve static load balancing. Local rendering is performed at each processor by ray-casting each block independently of other blocks and processors. Note that an immediate consequence of rendering at the block level is that many small ray segments are generated which would impose more communication between processors if we send them to the destination processors once they are generated. We therefore buffer ray segments into larger messages which are not delivered until the buffer becomes full [12]. In this way, we avoid sending many small messages which are costly due to the high message overhead.

Image space partitioning uses pixel interleaving to achieve maximum independence of viewing direction. Unlike the renderer we used for the uniform-mesh data, this renderer overlaps the local rendering and image compositing steps, and operates in a highly asynchronous fashion.

When rendering a block, sampling can be done in two different ways. One way which is straightforward to implement but quite expensive computationally is to sample at a fixed interval along the ray regardless of the resolution of the block. The other way is to sample adaptively according to the resolution of the block. Adaptive rendering is the natural thing to do for adaptive meshes. While adaptive rendering can reduce computational cost in a significant way, it must be implemented carefully to achieve exactly the same image quality. Each sample value must be weighted appropriately to ensure accurate image results. We are currently working on this problem.

6 Test Results

We have tested the two different rendering algorithms described in previous sections on the SGI/Cray T3E computer operated at the NASA Goddard Space Flight Center. The T3E is a distributed-memory massively parallel computer. Normal configurations of the machine may consist of 32 to 2048 Processor Elements (PEs) each with 128 megabytes to 2 gigabytes of DRAM memory. Although this memory is attached directly to each PE (physically distributed), it is globally addressable. We do not exploit this feature since we use MPI for explicit message passing to access non-local data. In the Goddard T3E, each PE is 300MHz DEC Alpha 21164 microprocessor with peak performance of 600 million floating point operations per second. All PEs are connected by a high-bandwidth, low-latency bidirectional three-dimensional torus system interconnect network. This topology ensures short connection paths and high bisection bandwidth (the maximum rate at which one half of the system can exchange data with the other half). Adaptive routing allows messages on the interconnect network to be rerouted around temporary “hot spots”. Interprocessor data payload communication rates are 480 megabytes per second in every direction through the torus; in a 512-PE CRAY T3E system, bisection bandwidth exceeds 122 gigabytes per second.

Because we planned to conduct a large number of tests using a large number of processors, with a limited allocation of computer time, we selected a relatively small AMR dataset for our performance study. In fact, large data demanding more computational load would mask the parallelization penalties associated with our algorithms because the communication cost would be negligible compared to the computational cost. This test dataset was generated from a magnetohydrodynamics (MHD) simulation of solar wind. It contains eight variables in 361 $8\times8\times8$ blocks. Figure 5 and 6 show volume visualization of the first variable which is density. Both images display a bow shock, with highest density in the shock itself. In Figure refvis1, the block structure is superimposed into the visualization while mesh within each block is not shown. In Figure refvis2, both the exterior surface and interior of the shock are revealed by enhancing low and hight values with the opacity transfer function.

The corresponding octree of this data has seven levels so after it is resampled into a uniform mesh, it contains $512\times512\times512$ voxels. The storage requirement is thus increased by a factor of 800. For a larger AMR data set, a parallel computer with sufficient memory space must be used just to make possible the rendering of the uniformly sampled data.

Figure 7 presents the total rendering time using up to 256 processors. Here, total rendering time only includes the local rendering time and image compositing time. Time for I/O and initialization of the renderer is not included. Logarithmic scale is used for both x and y axes to make the plots easy to interpret. Three different cases are considered:

- Uniform sampling on uniform mesh.
- Uniform sampling on adaptive mesh.
- Adaptive sampling on adaptive mesh.

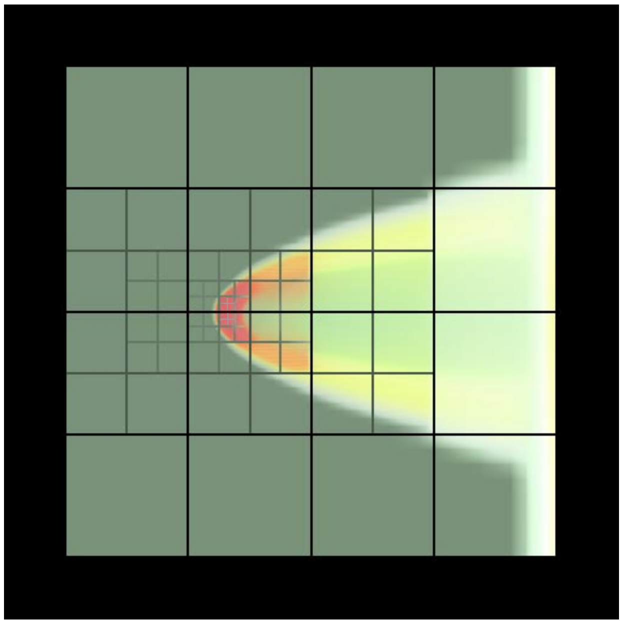

Figure 5. Visualization of the solution data from a MHD simulation using PARAMESH with block structure superimposed.

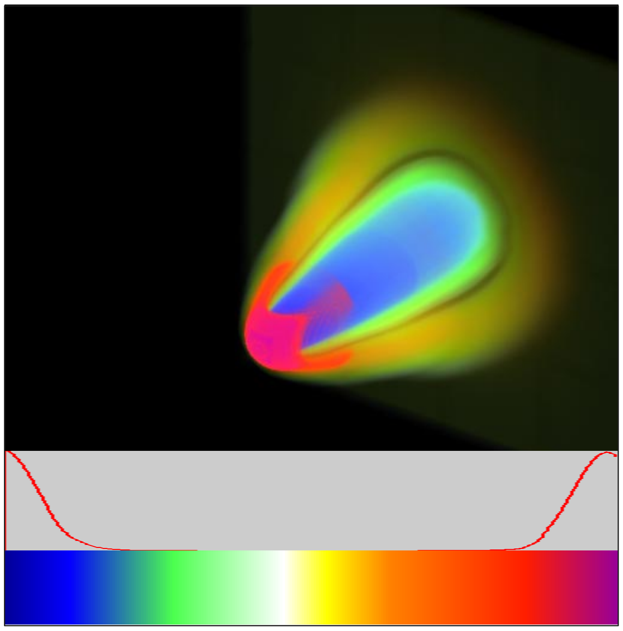

Figure 6. Visualization of solution data from a MHD simulation using PARAMESH, in which high and low density values are enhanced.

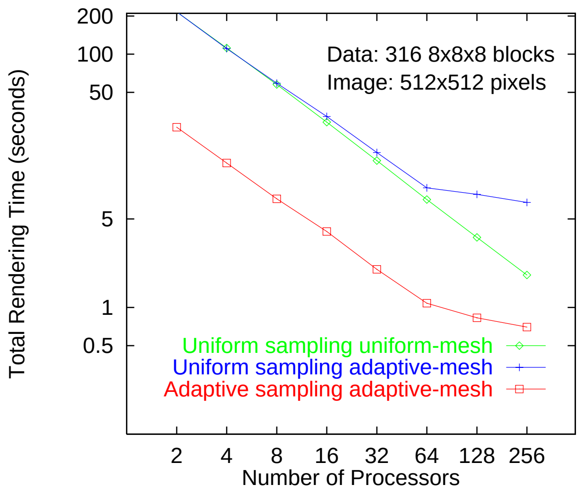

Figure 7. Total rendering time on T3E using up to 256 processors.

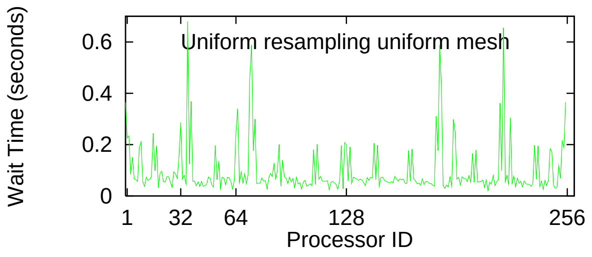

Figure 8. Wait time for the 256-processor case.

We see that uniform sampling on adaptive mesh takes slightly longer than on uniform mesh. This is due to the overhead of processing individual blocks and transferring the larger number of ray segments generated. Most importantly, adaptive sampling using 32 processors achieves roughly the same rendering rates as uniform-sampled rendering using 256 processors. This is a speedup factor of eight for the new rendering algorithm we are currently developing. The saving comes from taking fewer samples along each ray in the coarse mesh blocks, and can be characterized by

$$\frac{2^{N_l+1}}{(N_b \times n_x \times n_y \times n_z)^{\frac{1}{3}}}$$

relative to a similar ray-casting resampling approach.

We should point out a drawback of using a small data set for testing. For adaptive rendering using a large number of processors, each processor has only a very small number of blocks to render, like one or two. A large block projected to a large screen area always takes much longer time to render than a small block, even though they contain the same number of vertices. This results in load imbalance and loss of efficiency for the 128- and 256-processor cases as shown in Figure 7. Since our main goal here is to demonstrate that adaptive rendering requires fewer processors as well as a much smaller memory space to achieve interactive rendering rates, this small data set serves the purpose. However, further performance study using a large data set is needed to determine the true scalability of the new rendering algorithm.

Finally, we show in Figure 8 that partitioning the spatial domain evenly and coarsely as done by the uniform-sampled renderer leads to load imbalance because of the opacity transfer function used. This problem will likely be eliminated with the new rendering algorithm which uses finer partitioning and distributes data in a more random fashion. We are implementing a higher-resolution timing mechanism which will allow us to measure the wait time in the asynchronous rendering process. We expect this will be an order of magnitude smaller and quite even across processors.

7 Conclusions

Applications of advanced modeling techniques such as AMR, three-dimensional visualization techniques such as volume rendering, and state-of-the-art parallel computers such as the SGI/Cray T3E, together are among the frontiers of computational science. This paper reports our first attempt at combining these technologies and some experimental results. As we see an increasing use of the PARAMESH package in the modeling of many nationally-relevant scientific problems, the strategies we produce in this research will benefit users of PARAMESH (and other AMR technologies) in a significant way.

We have made possible three-dimensional visualization of AMR data generated from simulations using the PARAMESH package. Near-interactive rendering rates have been achieved. We have also demonstrated that it is much more efficient to render the AMR data directly rather than a finer, uniform resampling of the data. For the particular test case we show, the storage space is reduced by a factor of 800 and the rendering is speeded up by a factor 8. This allows us to use fewer processors to achieve desirable rendering rates. Our experience with AMR data and the T3E will help us develop an even more efficient and accurate algorithm for direct, adaptive rendering.

In addition, volume rendering can generate realistic visualization results if a sophisticated lighting model is used. To include lighting effects, we usually approximate the surface orientation at each voxel with the first gradient of voxel values. The orientation at each voxel is a vector and has three components. The extra storage space for saving these gradient values can be tremendous. However, we should include lighting calculations for final, high quality rendering. Other future work includes developing techniques to view the computational grid itself and display solution data superposed on a highly-magnified grid. These are of very strong interest to developers for debugging purposes.

8 Acknowledgements

This work was supported by the National Aeronautics and Space Administration under Contract No. NAS1-97046. The experiments described here were performed on the computer systems provided by NASA's HPCCP/CAS Project. We thank the system support personnel at NASA Goddard Space Flight Center for assistance in using their facilities. The author would also like to thank Michael Gross and Dan Spicer at Goddard for providing the datasets and discussion of the visualization requirements for AMR data, Ming-Yun Shih for writing the program to preprocess the datasets, and Tom Crockett for his comments on the manuscript.

References

[1] M. B. Amin, A. Grama, and V. Singh. Fast volume rendering using an efficient, scalable parallel formulation of the shear-warp algorithm. In *Proceedings of 1995 Symposium on Parallel Rendering*, pages 7–14, 1995.

[2] J. Challinger. Scalable Parallel Volume Raycasting for Nonrectilinear Computational Grids. In *Proceedings of Parallel Rendering Symposium*, pages 81–88, 1993. San Jose, October 25-26.

[3] B. Corrie and P. Mackerras. Parallel volume rendering and data coherence. In *Proceedings of Parallel Rendering Symposium*, pages 23–26, 1993. San Jose, October 25-26.

[4] W. M. Hsu. Segmented ray casting for data parallel volume rendering. In *Proceedings of Parallel Rendering Symposium*, pages 7–14, 1993. San Jose, October 25-26.

[5] P. Lacroute. Real-Time Volume Rendering on Shared Memory Multiprocessors Using the Shear-Warp Factorization. In *Proceedings of Parallel Rendering Symposium*, pages 15–22, 1995.

[6] P. Lacroute and M. Levoy. Volume rendering using a shear-warp factorization of the viewing transformation. In *SIGGRAPH 94 Conference Proceedings*, pages 451–457, 1994.

[7] T.-Y. Lee, C. S. Raghavendra, and J. N. Nicholas. Image composition schemes for sort-last polygon rendering on 2d mesh multicomputers. *IEEE Transactions on Visualization and Computer Graphics*, 2(3):202–217, September 1996.

[8] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3), July 1990.

[9] P. Li, S. Whitman, R. Mendoza, and J. Tsiao. ParVox - a parallel volume rendering system for distributed visualization. In *Proceedings of 1997 Symposium on Parallel Rendering*, pages 7–14, 1997.

[10] K.-L. Ma. Parallel volume ray-casting for unstructured-grid data on distributed-memory architectures. In *Proceedings of the Parallel Rendering '95 Symposium*, pages 23–30, 1995. Atlanta, Georgia, October 30-31.

[11] K.-L. Ma, M. Cohen, and J. Painter. Volume seeds: A volume exploration technique. *The Journal of Visualization and Computer Animation*, 2:135–140, 1991.

[12] K.-L. Ma and T. W. Crockett. A scalable parallel cell-projection volume rendering algorithm for three-dimensional unstructured data. In *Proceedings of 1997 Symposium on Parallel Rendering*, pages 95–104, 1997.

[13] K.-L. Ma, J. S. Painter, C. Hansen, and M. Krogh. Parallel Volume Rendering Using Binary-Swap Compositing. *IEEE Computer Graphics and Applications*, 14(4):59–68, July 1994.

[14] 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.

[15] U. Neumann. Parallel volume-rendering algorithm performance on mesh-connected multicomputers. In *Proceedings of Parallel Rendering Symposium*, pages 97–104, 1993. San Jose, October 25-26.

[16] U. Neumann. *Volume Reconstruction and Parallel Rendering Algorithms: A Comparative Analysis*. PhD thesis, UNC Chapel Hill, 1993.

[17] J. Nieh and M. Levoy. Volume rendering on scalable shared-memory MIMD architectures. In *1992 Workshop on Volume Visualization*, pages 17–24, 1992. Boston, October 19-20.

[18] M. E. Palmer, S. Taylor, and B. Totty. Exploiting deep parallel memory hierarchies for ray casting volume rendering. In *Proceedings of 1997 Symposium on Parallel Rendering*, pages 15–22, 1997.

[19] PARAMESH. URL:http://outside.gsfc.nasa.gov/ESS/eazydir/inhouse/macenice/paramesh/paramesh.html, 1998. NASA Goddard Space Flight Center.

[20] T. Porter and T. Duff. Compositing Digital Images. *Proceedings of SIGGRAPH '84*, 18(3), July 1984.

[21] K. Sano, H. Kitajima, H. Kobayashi, and T. Nakamura. Parallel processing of the shear-warp factorization with the binary-swap method on a distributed-memory multiprocessor system. In *Proceedings of 1997 Symposium on Parallel Rendering*, pages 87–94, 1997.

[22] P. Schröder and G. Stoll. Data parallel volume rendering as line drawing. In *1992 Workshop on volume Visualization*, pages 25–31, 1992. Boston, October 19-20.

[23] J. Wilhelms, A. V. Gelder, P. Tarantino, and J. Gibbs. Hierarchical and parallelizable diret volume rendering. In *Proceedings of the Visualization '96 Conference*, pages 57–64, October 1996.

[24] P. L. Williams. Parallel Volume Rendering Finite Element Data. In *Proceedings Computer Graphics International '93*, 1993. Lausanne, Switzerland, June.