Parallel Volume Rendering on the IBM Blue Gene/P

Tom Peterka[1], Hongfeng Yu[2], Robert Ross[1], Kwan-Liu Ma[2]
[1]Argonne National Laboratory
[2]University of California at Davis

Abstract

Parallel ray casting volume rendering is implemented and tested on an IBM Blue Gene distributed memory parallel architecture. Data are presented from experiments under a number of different conditions, including dataset size, number of processors, low and high quality rendering, offline storage of results, and streaming of images for remote display. Performance is divided into three main sections of the algorithm: disk I/O, rendering, and compositing. The dynamic balance between these tasks varies with the number of processors and other conditions. Lessons learned from the work include understanding the balance between parallel I/O, computation, and communication within the context of visualization on supercomputers, recommendations for tuning and optimization, and opportunities for scaling further in the future. Extrapolating these results to very large data and image sizes suggests that a distributed memory HPC architecture such as the Blue Gene is a viable platform for some types of visualization at very large scales.

Categories and Subject Descriptors (according to ACM CCS): I3.1 [Hardware Architecture]: Parallel processing, I3.2 [Graphics Systems]: Distributed / network graphics, I3.7 [Three-Dimensional Graphics and Realism]: Raytracing, I3.8 [Applications]

1. Introduction

As data sizes and supercomputer architectures grow towards the petascale and beyond, an attractive alternative to rendering on graphics clusters is to perform software-based visualization directly on parallel supercomputers. Benefits include the elimination of data movement between computation and visualization architectures, the economies of large scale, tightly coupled parallelism, and the possibility to perform in situ visualization. This paper examines the second contribution, large numbers of tightly connected processor nodes, within the context of a parallel ray casting volume rendering algorithm implemented on the IBM Blue Gene/P (BG/P) architecture at Argonne National Laboratory (ANL).

Volume rendering and parallel volume rendering on supercomputers has been published extensively in the literature, but this is the first such study conducted on BG/P, a modern supercomputer representative of others in its class. This research profiles and identifies bottlenecks in the rendering pipeline and suggests modifications to the parallel rendering algorithm to achieve scalability, and it offers a glimpse of the optimal balance between I/O, computation, communication, and interactivity requirements within the setting of parallel volume rendering on the BG/P.

The experiments include several different test conditions, including small to medium size data sets, real-time streaming of output images and offline storage of results, and both low and high quality renderings. From the results, one can draw several conclusions about how to best leverage the strengths of this architecture in visualization applications. Although the results are specific to a particular algorithm and architecture, the lessons learned can potentially apply more broadly to other supercomputer architectures that share some of the same characteristics as the Blue Gene, and to other parallel rendering algorithms as well.

Thus far, we have successfully scaled efficiently up to 512 cores, and tested out to 4096 cores. Remote streaming of a small, time-varying dataset at sub-second frame times was demonstrated. For the data sizes that we currently have available, performance is comparable to other methods and architectures, but we expect that the real benefits of this method will be apparent at still larger scales, for example,

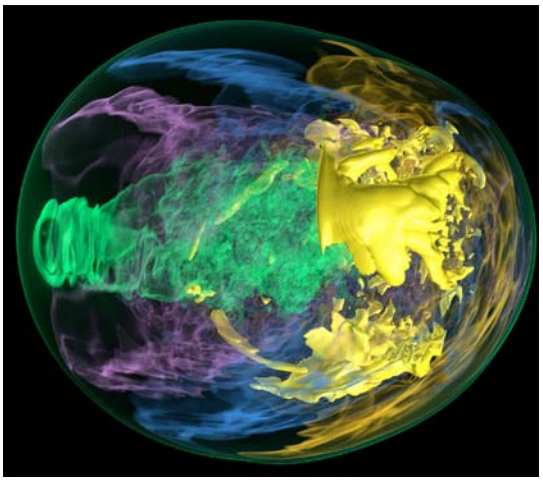

Figure 1: *Visualization of the early stages of supernova collapse.*

Direct correspondence to *tpeterka@anl.gov*

submitted to *Eurographics Symposium on Parallel Graphics and Visualization (2008)*

when the size of the data exceeds one billion voxels; image resolution is on the order of one million pixels, and the number of cores exceeds several thousand processors. Research is ongoing, and the goal of our future work is to measure performance at these scales.

2. Background

Dataset

The dataset shown in Figure 1 is one time step from a supernova simulation, made available by John Blondin at the North Carolina State University and Anthony Mezzacappa of Oak Ridge National Laboratory [1], through the US Department of Energy's SciDAC Institute for Ultrascale Visualization [2]. The model seeks to discover the mechanism behind the core-collapse supernova, which is the violent death of short-lived, massive stars. A spherical accretion shock instability, or SASI, is driven by the response of an initially spherical shock wave to global acoustic modes trapped in the interior.

Visualization plays a key role in understanding the origin of this instability of the supernova shock wave, which allows scientists to quickly visualize different combinations of variables or isolate features by manipulating the transparency of the rendered data. In this dataset, a single scalar variable, angular momentum, is stored at uniform, structured grid locations. Each of two hundred time steps of time-varying data is stored in a separate file. Files are stored in raw, binary format, in 32-bit floating-point format.

Algorithm

Parallel volume rendering algorithms have been well documented in the literature. Beginning with Levoy's classic ray casting in 1987 [3] and optimizations in 1990 [4], parallel versions began to appear in 1993 with [5] and [6]. More recently, Yu demonstrated that parallel volume rendering performance can be further improved by overlapping simulation with visualization [7]. Parallel volume rendering has also been studied within the context of cluster computing [8] and in standard visualization toolkits such as VTK [9], ParaView [10], and VisIt [11], [12].

Our implementation uses post classification after trilinear interpolation, includes lighting [13], [14], and is optimized for early ray termination based on maximum opacity and blank voxel regions. Sort-last parallelization occurs both in object space and in image space. The dataset is divided into n approximately equal size partitions, where n is the number of processes. Each process computes a completed sub-image corresponding to its local data, including local front-to-back compositing of samples along each ray of its local subimage using the "over" operator [15] and early ray termination.

Stompel et al. [16] provide an overview of various methods for sort-last compositing of the n sub-images, and Cavin et al. [8] analyze relative theoretical performance of these methods. These overviews show that compositing algorithms usually fall into one of the following categories: plain or optimized direct send, plain or optimized tree, and parallel pipeline. The direct send approach is easiest to understand; each process requests the sub-images from all of those processes that have something to contribute to it [17], [18], [19]. Since the possibility for network contention is high in direct send, the SLIC [16] optimization attempts to schedule communication.

Rather than sending compositing data monolithically, tree methods exchange data between pairs of processes, building larger completed subimages at each level of the compositing tree. To keep more processes busy at higher levels on the tree, Ma et al. introduced the binary swap optimization [20]. Lee et al. discuss a parallel pipeline compositing algorithm in [21] for polygon rendering, although this seldom appears in the context of parallel volume rendering.

For simplicity and a high degree of parallelism, we use the direct send compositing approach. After all n processes complete their subimages, the total image space is divided among n processes as well, so that each process now contains $1/n$ of the total data volume and is responsible for $1/n$ of the total image area. Finally, all of the finished, composited subimages are sent to a single root process that tiles them into a final image, which can either be stored to disk or streamed to a remote display location.

The time to write the final image is not significant for the image sizes tested, so we choose to ignore the time to write the final output image to disk, and define the time that a frame takes to complete as the time from the start of reading the time step from disk to the time that the final image is ready at the root process. This frame time has three distinct components, and for a given data size, the relative contribution of each component to the total time depends on the number of processes:

$$t_{frame} = t_{io} + t_{render} + t_{composite} \qquad (1)$$

The I/O time, t_{io}, is the length of time required by a collective reading of the time step data file by all processes simultaneously. The rendering time, t_{render}, is the time that it takes for all processes to complete their local subimage rendering. The compositing time, $t_{composite}$, is the time to composite all subimages into a single image on a single process. The following section describes the implementation of each component in more detail.

Before the execution of the first frame, a one-time initialization step allocates data structures and determines partitioning parameters; static load balancing is used. The time for this setup is on the order of tens of seconds, and because it occurs only once, we omit it from the frame time.

Blue Gene architecture

The Blue Gene/L and Blue Gene/P systems at ANL provide ample opportunities to experiment with parallel rendering. This work began with 2048 cores of the BG/L system and has scaled so far up to 4096 cores on the BG/P system. The current single rack of BG/P is for testing and development, but in the near future, ANL's BG/P system will contain 128K cores. Online documentation from IBM can be found at [22]; the reader is directed there for specifications and configuration diagrams. For our purposes, the key differences between the older BG/L and the new BG/P are

that BG/P provides twice as many cores, twice the memory footprint, approximately a 2X faster interconnect network, and a 1.2X faster clock speed per core.

Processor cores are grouped together into nodes; the BG/P has 4 cores per node. Within a node, the cores can operate together to execute one user process, in pairs for two processes, or independently for four user processes, depending on the selected mode. Application processes execute on top of a micro-kernel that provides basic OS services. The Blue Gene architecture has two separate interconnection networks – a 3D torus for inter-process point-to-point communication, and a tree network for collective operations as well as for communicating with I/O nodes. BG/P there has one I/O node for every 64 compute nodes. At the front end, the machine has four login nodes that support full Linux functionality.

3. Implementation

I/O

Our volume rendering application is written using MPI for both communication and I/O, and executes with one MPI process on each core. MPI-2 [23] (a.k.a. MPI-IO) collective file read calls perform data staging, t_{io} in equation 1, allowing each process to read its own portion of the volume in parallel with all of the other processes [7], [24]. This is more efficient than a single master process reading the entire dataset and distributing it to slave processes, and more importantly for large datasets, it does not require a single process to be able to fit the entire dataset into its memory.

For example, the largest dataset tested to date in this work consists of 864^3 voxels, or approximately 2.5 GB per time step. This is problematic for most workstations; even the BG/P has only 2GB of memory per node. However, with collective I/O, the total memory footprint of the entire machine, not just of one node, is the upper bound on the maximum data size that can be processed in-core. This memory limit on the current single-rack BG/P is 2TB, and will grow to 64 TB when the system is complete.

Underlying the MPI-2 collective I/O interface is a PVFS parallel file system [25]. By striping data across multiple volumes controlled by a number of file servers, application programs can access non-contiguous regions of a file in parallel. Performance varies depending on whether reads or writes are executed (reads in our case), on the number of I/O nodes being used, and on the size of the partition that each process reads. These issues will be revisited in next section as the key relationships between application performance and I/O throughput are exposed.

BG/P is still a new system undergoing development, and this holds true for its PVFS deployment, which has only been functioning for a short time as of this writing. Therefore, it is largely un-tuned and I/O throughput is expected to increase dramatically in the future. Even though our research is sometimes limited by currently available hardware capabilities, the early testing of this application is assisting both the BG/P and PVFS teams to expose and correct implementation problems.

When using PVFS, in particular when performance testing an application, it is important to realize that PVFS is a shared resource. Unlike the Blue Gene's compute nodes that start each job with a clean kernel and are dedicated to only one job, PVFS serves the entire machine and the login nodes as well. This means that PVFS performance for a given application is dependent on the total load of the system. We can see this in repeated trials of the same configuration; t_{render} and $t_{composite}$ repeat consistently, but t_{io} can vary depending on total I/O load. In our timing measurements, we have taken care to restrict other's PVFS usage, and confirmed results over multiple trials, but it should be remembered that I/O performance could vary in the context of everyday usage.

Rendering

The computation of local subimages, t_{render} in equation 1, is embarrassingly parallel – that is, it requires no inter-process communication and scales linearly with n. Its per-core performance is a function of the efficiency of the Blue Gene's compute node: clock speed, pipeline architecture, cache coherence, and the extent to which the code is tuned to optimize these features. Compiler optimizations thus far have netted 2X performance gains in t_{render}.

We are currently evaluating low-level performance counters to gauge the use of BG/P's double-hummer pipeline, but do not yet have a conclusive metric of the percentage of its use in this code. Another possibility we are considering is the storage in memory of the dataset as double precision instead of single precision, together with associated compiler directives concerning quad-word data alignment. This could better exploit IBM's double hummer dual floating unit, but it is unclear whether potential gains will be offset by the expanded memory requirements.

Peak FLOPS rates are theoretical and can be misleading. For example, measurements indicate the compute kernel running at approximately 200 MFLOPS per core, or 6% of the advertised 3.4 GFLOPS peak per core on BG/P. This is not surprising for actual code that contains loops, branches, etc. Performance tuning of the computation kernel is an ongoing aspect of this research.

Compositing

Compositing of parallel volume rendered subimages, $t_{composite}$ in equation 1, is implemented with direct send as follows. At the point of completion of the render stage and

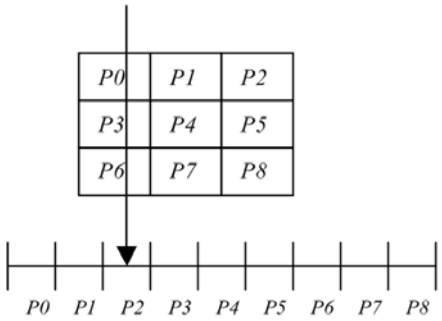

Figure 2: *Direct-send compositing divides both the object space and image space among processes.*

just prior to the beginning of compositing, each of the *n* processes owns a completed sub-image of its portion of the dataset. Next, each of the *n* processes is assigned responsibility for *1/n* of the final image area as well. For example, the final image can be divided into *n* scan lines or rectangles. There is no spatial correspondence between the location and size of a process' currently completed subimage from the rendering step, and the portion of the final image that the process is responsible for compositing during the compositing step.

For example, consider the 9-process 2D example in Figure 2. The squares represent the volume divided into 9 subvolumes, and the line along the bottom represents the image divided into 9 regions. (The image need not be aligned with the subvolume axes.) Looking at process *P2*, it is responsible for one subvolume and one portion of the image. Through a global data structure that all processes share, *P2* knows that it must get the subimages from *P6, P3,* and *P0*, composite them in front-to-back order, to form its portion of the final image. Equations 2 and 3 recursively compute color and opacity during compositing,

$$i = (1.0 - a_{old}) * i_{new} + i_{old} \quad (2)$$
$$a = (1.0 - a_{old}) * a_{new} + a_{old} \quad (3)$$

Where *i* represents the intensity (r,g,b) premultiplied by its associated alpha-value, and *a* represents the accumulated alpha-value or opacity.

The last step is for processes *P1* thru *P8* to send their final results to process *P0*, which tessellates them together into one image. The average communication complexity of $t_{composite}$ is $O(n^{4/3} + n)$. The first term, $n^{4/3}$, is because on average, $n^{1/3}$ messages must be sent to each of *n* recipients in order for the *n* processes to composite their portion of the final image. The second term, *n*, represents the gathering of final subimages at the root process.

Streaming and Prefetching

When resulting images are streamed to a remote display device, rather than being stored on disk, the path requires several steps. This is because the Blue Gene connects to the outside world only through the front-end login nodes. So, to send an image from one of the compute nodes, it first passes via a socket to the IP address of one of the login nodes. Physically, it actually travels from the compute node to the I/O node assigned to that compute node, and from the I/O node to the login node, but the connection between compute node and associated I/O node is transparent to the programmer. Finally, a daemon running on the login node forwards the data stream to the remote display via a separate socket connection. The connectivity is diagrammed in Figure 3.

Prefetching of time steps can be harnessed when the optimal number of cores is significantly less than the total number available. For example, when processing 300^3 data in order to stream images to a remote site, the optimal number of cores dedicated to one frame was 512. Therefore, the balance of the machine can be applied toward processing the next or several next time steps. This results in a multi-pipe application structure, as in Figure 4. Each of the pipes functions independently according to the previous description. In the example of Figure 4, four separate sockets send images out of the Blue Gene to a remote display. A token is passed between the pipes to ensure that images are sent in order. The receiving graphics application regulates frame rate so that images appear on the screen at a constant rate.

4. Performance data

In November 2007, real-time streaming of the volume rendering application from BG/L was demonstrated, generating and streaming a series of 200 time steps repeatedly from ANL in Chicago, Illinois to the Supercomputing conference exhibit floor in Reno, Nevada. A single time step is 103 MB, and over the course of the one-hour demo, approximately 500 GB of data was processed in real time. The optimal setting for this data size was 512 cores. Figure 5 shows more recent tests of the same data on BG/P, out to 4K processes.

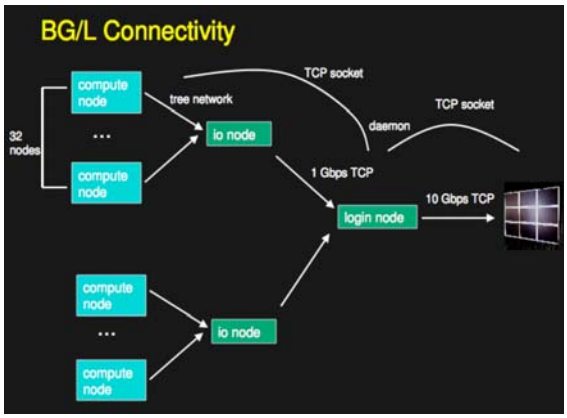

Figure 3: *Connecting a compute node to a remote display is a several-step process.*

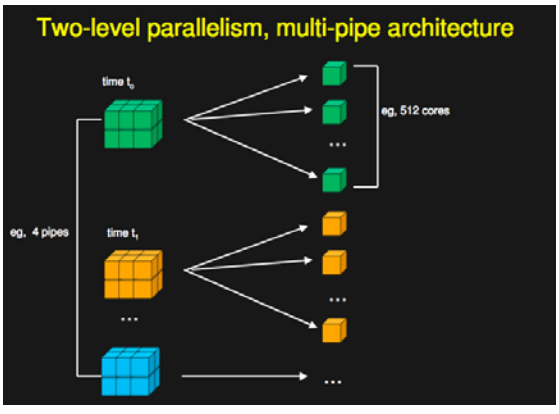

Figure 4: *Processing several frames simultaneously can extend the degree of parallelism.*

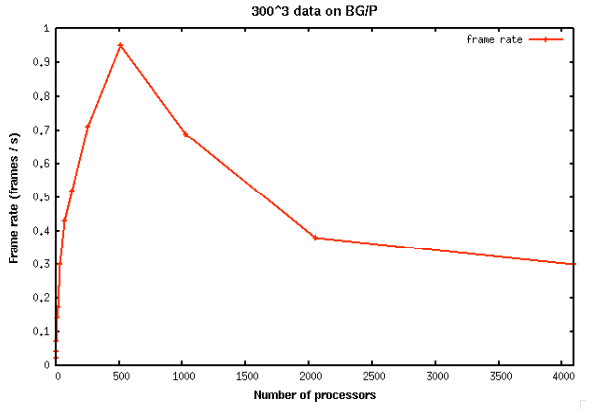

Figure 5: *Performance of BG/P on 300^3 downsampled dataset.*

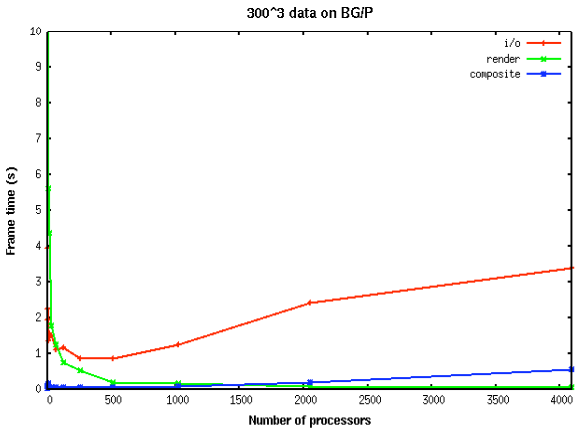

Figure 6: *Comparison of t_{io}, t_{render}, and $t_{composite}$ is shown in terms of absolute frame time.*

The BG/P is capable of executing one, two, or four processes per node. In IBM terminology, these are called *smp* mode, *dual* mode, *vn* mode, respectively. In smp mode, one core performs computation while the other cores idle, with the exception of low-level OS tasks. The total memory footprint of 2GB per node is shared among the four cores in smp mode.

Our tests show approximately 10% slower performance in dual and vn modes, compared to smp mode. The largest increase is in t_{io}, because the number of I/O nodes assigned to a job is a fraction of the number of compute *nodes*, not compute *cores*. On the BG/P, this number is 64 compute nodes to one I/O node. Using twice as many compute nodes means that twice as many I/O nodes are available for t_{io}.

Figure 6 compares the contribution to t_{frame} of each of t_{io}, t_{render}, and $t_{composite}$ for the same 300^3 dataset on BG/P. It is clear that I/O time dominates beyond 64 processes, but this plot of absolute times actually masks some important features. It would appear that t_{render} drops so quickly and that $t_{composite}$ grows so slowly that it does not make sense to optimize them. Both of these assumptions are proved false by Figure 7, which shows the same performance data plotted as relative percentages of the total time, t_{frame}.

From Figure 7, we see that at smaller numbers of processes, rendering time dominates the frame time, but I/O cost dominates beyond 64 processes. This underlines the need to further optimize parallel I/O operation on BG/P. Rendering time decreases more slowly as a percentage of the total time, compared to Figure 6, and is a significant concern out to at least 2048 processes. Compositing time is still a relatively small fraction of the total time, reaching a maximum of 14% and usually less than 10%. However, Figure 7 clearly shows its relative contribution steadily increasing, hence it cannot be ignored indefinitely, especially if one expects to scale to tens of thousands of processes.

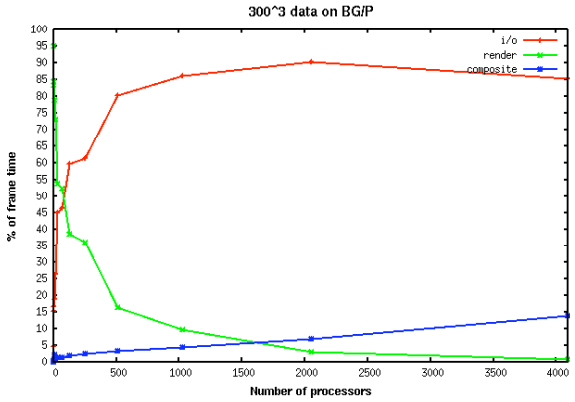

Figure 7: *Relative contribution to t_{frame} of each of t_{io}, t_{render}, and $t_{composite}$ is shown.*

Even when PVFS is optimized on BG/P, there will likely be configurations that are more efficient than others in terms of I/O. For example, 8 I/O nodes seems to be a sweet spot in terms of throughput. At 64 compute nodes per I/O node, this corresponds to 512 processes, 1 process per node. Also, it is known that PVFS does not perform well when the partition size per process becomes too small; 1 or 2 MB seems to be a good target according to past experience, although MPI-IO optimizations can mitigate the impact of small partition sizes.

With such reliance in this application on I/O rates, it is worthwhile to study the I/O performance in more detail, and to continue to reevaluate it as PVFS on BG/P becomes more mature. Figure 8 shows current read bandwidth for three dataset sizes: 300^3, 600^3, and 800^3 voxels. In each case, the best throughput occurs at approximately 256-512

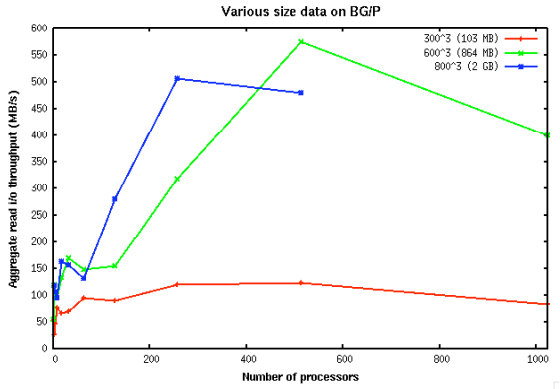

Figure 8: *Aggregate read I/O throughput is plotted for various data sizes and numbers of processes.*

processes, and the optimal partition size appears to be between 2-8 MB. PVFS performs much better on larger data sizes than on small ones, as Figure 8 shows. More detailed tests will be required to confirm these results, once PVFS has been dialed-in on BG/P. The resulting throughput values reflect end-to-end I/O times, including file open and file close, not the raw time to perform the actual read.

The full size supernova dataset is 864^3 voxels, which we reduced slightly to 800^3 for our final test. For our largest scale result thus far, an output image of 1600^2 pixels was rendered from 800^3 voxels in a frame time of approximately 7 seconds. In this test, each time step is 2 GB, or ½ billion voxels, and each resulting image is 2.5 Megapixels.

5. Conclusions

The Blue Gene architecture can be an appropriate platform for high quality software visualization algorithms such as classical direct volume rendering by ray casting. Its salient features with respect to this application are: large numbers of tightly connected cores, a flexible programming API (MPI), a high-bandwidth connection to the parallel I/O system (MPI-IO and PVFS), and the ability to connect via sockets to remote displays. Software rendering cannot produce better performance than graphics clusters for small to medium size problems, but if current trends in data size [26], [27] continue, massively parallel supercomputer software volume rendering may become a predominant method in the future.

A closely related metric to data size is image resolution. In our tests, the image size is chosen such that the number of pixels in one dimension of the image is twice the number of voxels in one dimension of the volume, to satisfy the Nyquist Sampling Theorem.

We believe that our research will prove useful for data sizes larger than 1000^3 voxels (several gigavoxels) in conjunction with image sizes larger than 2000^2 pixels (several megapixels). The method is also promising for in situ visualization [28], or in general when a very large dataset resides on the system already. As data sizes increase, transporting data between machines becomes non-trivial.

The relative cost of the three phases of the algorithm changes with the number of processes, although ultimately the application is I/O bound. There is a certain tension between applying enough processes to reduce the rendering time, but not so many as to force the I/O system to read many partitions of a very small size, which is inefficient. A similar trade-off, but to a lesser degree, exists between rendering and compositing. Therefore, it is unlikely that this method alone can effectively produce highly interactive performance, for example, 30 frames per second. It is more likely that its niche will be for very large data sets that cannot be accommodated by graphics clusters, and can produce frame times on the order of a few seconds for such data.

Within the rendering kernel, performance tuning is still important. The ability to apply parallelism should augment, not replace, good algorithms, coding practices, and optimizations. Minimizing the number of processes by optimizing code using compiler flags, directives, and quad-word alignment can speed up execution, so that larger, more efficient I/O partition sizes can be used among fewer processes.

The spare capacity of the machine can be applied to the application in several ways. Prefetching of several frames through a multi-pipeline layout is one alternative. This produces two levels of parallelism; the first level is a gross division of the machine into, for example, four pipes, each pipe processing one frame. The second level of parallelism is the division of a pipe into, for example, 512 processes.

The next approach for utilizing more of the machine capacity is to improve the quality of the rendering, for example to enable lighting and shading calculations. In the performance results, lighting was disabled, but Figure 1 shows that very high quality images can result through the addition of lighting. On the 300^3 dataset, lighting slowed down performance by approximately 50%. However, the cost of lighting will be less significant at higher process counts, because lighting is a part of the rendering cost.

Another quality adjustment is the sample spacing along each ray. This is also adjustable and is set to twice the voxel spacing in these tests. However, slight reductions in sample spacing can increase performance significantly, but at the expense of small "holes" throughout the resulting image. However, this technique can be used as a type of level-of-detail reduction for improving interaction rates.

6. Future Work

We are continuing work to scale data size to gigavoxels and image size to megapixels, and to improve image quality through lighting and shading in our next tests. In order to do so, each of the three components of t_{frame}, t_{io}, t_{render}, and $t_{composite}$, must be further improved. I/O performance relies mainly on improvements to the PVFS and I/O forwarding implementations on BG/P. Rendering performance can be further improved with code tuning and optimization.

We are planning to experiment with tree-based compositing as a replacement for direct-send. This may include binary swapping per [20] as a way to balance the number of messages with the size of a message, and to keep more processes busy during the late stages of compositing. This also implies that several processes, instead of a single root process, will write the output image collectively to disk. This is similar to the way data is now being read from disk. In the case of streaming images, several sockets can transmit in parallel, similar to the way that the multi-pipe configuration now sends images.

Besides propelling optimizations and algorithmic improvements, extending the scale of an application also flushes out implementation issues that may be hidden at smaller scales. While it can be painful to perform application testing on a machine still undergoing acceptance testing with a nascent parallel I/O implementation, in the long run it benefits both the application and systems researchers.

For example, we recently discovered an MPI-2 implementation issue at large numbers of nodes with the 864^3 dataset, forcing our current largest results to be performed at 800^3. In the past, we have discovered another MPI-2 implementation bug and a BG/P memory allocation bug through this research. Likewise, several application

errors were discovered when scaling to hundreds and thousands of processes.

In the future we will also begin to study how this research can be extended to encompass adaptive mesh refined (AMR) time-varying datasets [29], [30]. Varying levels of spatial resolution encoded in AMR data provide a compromise between the rigidity of completely structured data, and the randomness of entirely unstructured data. This may significantly impact the initial setup time. This step was ignored up to now because it was a one-time cost; the structure did not change from one frame to the next. However, in the worst AMR case, each time step could have a different spatial layout. This could add tens of seconds to the frame time, without some method of rapidly restructuring the initial data structures.

Another goal is to collate the performance data into a coherent model for predicting future performance. This model may take the form of a smooth, high-dimensional manifold, or a set of governing equations, or various rules and heuristics. The determination of what input criteria, such as processor speed, data size, number of processes, network bandwidth, memory bandwidth, aggregate I/O throughput, etc, should be included into such a model is an open question. The result should be a relatively simple-to-use module that can analyze a parallel volume rendering problem and suggest an optimal configuration and predict its performance.

Finally, one of our long-term goals is to study a supercomputer architecture can be used to support interactive rendering. The research so far has not included any elements of interactivity, and performance data reveals that reaching interactive rates is difficult because of the tradeoffs between t_{io}, t_{render}, and $t_{composite}$. The next steps toward interactive rates may include LOD rendering as well as local view interpolation at the display machine(s). The ideal configuration may be the supercomputer and the graphics machine(s) sharing responsibilities in a client-server architecture.

Acknowledgments

The authors wish to thank John Blondin and Anthony Mezzacappa for making their dataset available for this research. This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

References

[1] BLONDIN, J. M., MEZZACAPPA, A. DEMARINO, C.: Stability of Standing Accretion Shocks, with an Eye Toward Core Collapse Supernovae. *The Astrophysics Journal*, 584, 2, (2003), 971.

[2] SciDAC Institute for Ultra-Scale Visualization. http://ultravis.ucdavis.edu/ 2007.

[3] LEVOY, M.: Display of Surfaces from Volume Data. *IEEE Computer Graphics and Applications*, 8, 3, (May 1988), 29-37.

[4] LEVOY, M.: Efficient Ray Tracing of Volume Data. *ACM Transactions on Graphics*, 9, 3, (July 1990), 245-261.

[5] MA, K.-L., PAINTER, J. S., HANSEN, C. D. KROGH, M. F.: A Data Distributed, Parallel Algorithm for Ray-Traced Volume Rendering. *Proceedings of 1993 Parallel Rendering Symposium*, San Jose, CA, (October 1993), 15-22.

[6] NEUMANN, U.: Parallel Volume-Rendering Algorithm Performance on Mesh-Connected Multicomputers. *Proceedings of 1993 Parallel Rendering Symposium*, San Jose, CA, (October 1993), 97-104.

[7] YU, H., MA, K.-L. WELLING, J.: A Parallel Visualization Pipeline for Terascale Earthquake Simulations. *Proceedings of Supercomputing 2004*, (November 2004), 49.

[8] CAVIN, X., MION, C. FIBOIS, A.: COTS Cluster-based Sort-last Rendering: Performance Evaluation and Pipelined Implementation. *Proceedings of IEEE Visualization 2005*, (October 2005), 111-118.

[9] BIDDISCOMBE, J., GEVECI, B., MARTIN, K., MORELAND, K. THOMPSON, D.: Time Dependent Processing in a Parallel Pipeline Architecture. *IEEE Transactions on Visualization and Computer Graphics*, 13, 6, (October 2007), 1376-1383.

[10] MORELAND, K., AVILA, L. FISK, L. A.: Parallel Unstructured Volume Rendering in ParaView. *Proceedings of IS&T SPIE Visualization and Data Analysis 2007*, San Jose, (January 2007).

[11] CHILDS, H. R., BRUGGER, E. S., BONNELL, K. S., MEREDITH, J. S., MILLER, M. C., WHITLOCK, B. J. MAX, N. L.: A Contract Based System for Large Data Visualization. *Proceedings of IEEE Visualization 2005*, Minneapolis, MN, (October 2005), 190-198.

[12] CHILDS, H., DUCHAINEAU, M. MA, K.-L.: A Scalable, Hybrid Scheme for Volume Rendering Massive Data Sets. *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization 2006*, Braga, Portugal, (May 2006), 153-162.

[13] DREBIN, R. A., CARPENTER, L. HANRAHAN, P.: Volume Rendering. *ACM SIGGRAPH Computer Graphics*, 22, 4, (August 1988), 65-74.

[14] MAX, N. L.: Optical Models for Direct Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1, 2, (June 1995), 99-108.

[15] PORTER, T. DUFF, T.: Compositing Digital Images. *Proceedings of 11th Annual Conference on Computer Graphics and Interactive Techniques*, (1984), 253-259.

[16] STOMPEL, A., MA, K.-L., LUM, E. B., AHRENS, J. PATCHETT, J.: SLIC: Scheduled Linear Image Compositing for Parallel Volume Rendering. *Proceedings of IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, Seattle, WA, (October 2003), 33-40.

[17] HSU, W. M.: Segmented Ray Casting for Data Parallel Volume Rendering. *Proceedings of 1993 Parallel Rendering Symposium*, San Jose, CA, (1993), 7-14.

[18] NEUMANN, U.: Communication Costs for Parallel Volume-Rendering Algorithms. *IEEE Computer Graphics and Applications*, 14, 4, (July 1994), 49-58.

[19] MA, K.-L. INTERRANTE, V.: Extracting Feature Lines from 3D Unstructured Grids. *Proceedings of IEEE Visualization 1997*, Phoenix, AZ, (October 1997), 285-292.

[20] MA, K.-L., PAINTER, J. S., HANSEN, C. D. KROGH, M. F.: Parallel Volume Rendering Using Binary-Swap Compositing. *IEEE Computer Graphics and Applications*, 14, 4, (July 1994), 59-68.

[21] LEE, T.-Y., RAGHAVENDRA, C. S. NICHOLAS, J. B.: Image Composition Schemes for Sort-Last Polygon Rendering on 2D Mesh Multicomputers. *IEEE Transactions on Visualization and Computer Graphics*, 2, 3, (September 1996), 202-217.
[22] IBM Redbooks. http://www.redbooks.ibm.com/redpieces/abstracts/sg247287.html?Open 2007.
[23] GEIST, A., GROPP, W., HUSS-LEDERMAN, S., LUMSDAINE, A., LUSK, E., SAPHIR, W. SKJELLUM, T.: MPI-2: Extending the Message-Passing Interface. *Proceedings of Euro-Par'96*, Lyon, France, (October 1996).
[24] YU, H. MA, K.-L.: A Study of I/O Methods for Parallel Visualization of Large-Scale Data. *Parallel Computing*, 31, 2, (February 2005), 167-183.
[25] CARNS, P., LIGON, W. B. I., ROSS, R. THAKUR, R.: PVFS: A Parallel File System for Linux Clusters. *Proceedings of 4th Annual Linux Showcase & Conference*, Atlanta, GA, (2000), 28.
[26] MOUNT, R.: The Office of Science Data-Management Challenge. Report from the DOE Office of Science Data-Management Workshops, 2004.
[27] JOHNSON, C. ROSS, R.: Visualization and Knowledge Discovery: Report from the DOE/ASCR Workshop on Visual Analysis and Data Exploration at Extreme Scale, 2007.
[28] TU, T., YU, H., RAMIREZ-GUZMAN, L., BIELAK, J., GHATTAS, O., MA, K.-L. O'HALLARON, D. R.: From Mesh Generation to Scientific Visualization: An End-to-end Approach to Parallel Supercomputing. *Proceedings of Supercomputing 2006*, Tampa, FL, (November 2006).
[29] MA, K.-L.: Parallel Rendering of 3D AMR Data on the SGI/Cray T3E. *Proceedings of 7th Annual Symposium on the Frontiers of Massively Parallel Computation 1999*, Annapolis MD, (February 1999), 138-145.
[30] WEBER, G. H., HAGEN, H., HAMANN, B., JOY, K. I., LIGOCKI, T. J., MA, K.-L. SHALF, J. M.: Visualization of Adaptive Mesh Refinement Data. *Proceedings of IS&T/SPIE Visual Data Exploration and Analysis VIII*, San Jose, CA, (2001), 121-132.