Assessing and Improving Large Scale Parallel Volume Rendering on the IBM Blue Gene/P

Tom Peterka*, Robert Ross*, Hongfeng Yu†, Kwan-Liu Ma‡, Wesley Kendall§, and Jian Huang§
*Argonne National Laboratory
Email: tpeterka@mcs.anl.gov
†Sandia National Laboratories, California
‡University of California, Davis
§University of Tennessee, Knoxville

***Abstract*—Computational science's march toward the petascale demands innovations in analysis and visualization of the resulting datasets. As scientists generate terabyte and petabyte data, it is insufficient to measure the performance of visual analysis algorithms by rendering speed only, because performance is dominated by data movement. We take a systemwide view in analyzing the performance of software volume rendering on the IBM Blue Gene/P at over 10,000 cores by examining the relative costs of the I/O, rendering, and compositing portions of the volume rendering algorithm. This examination uncovers room for improvement in data input, load balancing, memory usage, image compositing, and image output. We present four improvements to the basic algorithm to address these bottlenecks. We show the benefit of an alternative rendering distribution scheme that improves load balance, and how to scale memory usage so that large data and image sizes do not overload system memory. To improve compositing, we experiment with a hybrid MPI - multithread programming model, and to mitigate the high cost of I/O, we implement multiple parallel pipelines to partially hide the I/O cost when rendering many time steps. Measuring the benefits of these techniques at scale reinforces the conclusion that BG/P is an effective platform for volume rendering of large datasets and that our volume rendering algorithm, enhanced by the techniques presented here, scales to large problem and system sizes.**

I. Introduction

In the face of the petascale era, innovative visualization technologies will be required to keep pace with dramatically increasing datasets. Supercomputer software rendering is a throwback to the past, but the methods presented in this paper are implemented and tested on one of the world's most powerful supercomputers: the IBM Blue Gene/P (BG/P). We test volume rendering on BG/P at massively parallel scales – over 10,000 cores – in the context of astrophysics data from the simulation of supernova core collapse (see Figure 1). This research, under the auspices of the U.S. Department of Energy's SciDAC Institute for Ultra-Scale Visualization [1] is conducted on the 550 teraflop BG/P at Argonne National Laboratory's Leadership Computing Facility (ALCF) [2].

While parallel volume rendering algorithms are well known, targeting a new architecture, large problem sizes, and extensive system scale can lead to a better understanding of the relative cost of the various stages of the method and uncovers the need for improvements in some areas. Much visualization research in recent years harnesses GPU hardware to increase performance. Our goal is not to compete with this technology, because software rendering rates are much lower than dedicated graphics hardware. Growing dataset size, however, implies that GPUs, while highly parallel at the hardware level, will require interchip and internode parallelism and be subject to many of the same performance bottlenecks as slower, software renderers. When raw rendering time is not the bottleneck in end-to-end performance, a modern supercomputer such as BG/P becomes a viable alternative to a graphics cluster. With its tightly coupled interconnection backbone, parallel file system, and massive number of cores, such an architecture scales more predictably once the problem size grows to billions and tens of billions of data elements. Factors such as I/O and communication bandwidth – not rendering speed – dominate the overall performance.

The opportunity for in situ visualization [3] [4] [5], or overlapping visualization with an executing simulation, is another advantage associated with visualizing data on the same architecture as the simulation. Because postprocessing and transporting data can dominate a scientific workflow as data sizes increase, moving the analysis and visualization closer to the data can be less expensive than the other way around.

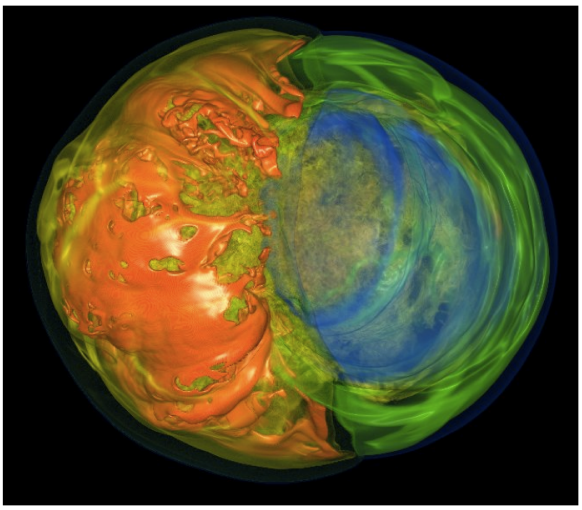

Fig. 1. Software volume rendering of the entropy from time step 1354 of a core collapse supernova simulation

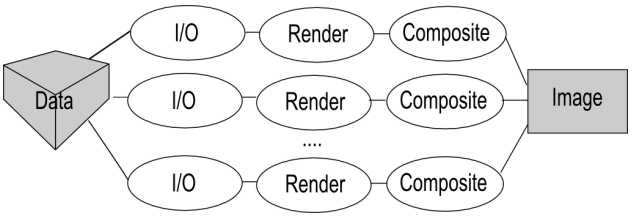

Fig. 2. Functional diagram of the parallel volume rendering algorithm

In situ techniques grow in importance as simulations grow in size. For example, Yeung et al. already routinely compute flow simulations at 2048^3 data elements [6], and their next target is 4096^3 using the IBM Blue Gene architecture. This equates to approximately 270 GB per time step per variable. Similar dataset sizes are currently being produced by Woodward et al. [7] in the area of fluid dynamics and by Chen et al. [8] in earthquake simulation.

In this paper we examine the costs of the various phases of our volume rendering algorithm; I/O, rendering, and compositing as we scale the problem and system size. Based on our findings, we examine four improvements to the algorithm and study how each technique affects performance. The techniques are: (a) load balancing via round-robin data block distribution, (b) memory conservation via parallel image output, (c) leveraging shared memory architecture via hybrid programming, and (d) hiding I/O latency through multiple time step pipelines. While these techniques are not new inventions, we test them at scales up to 16K cores and analyze the benefit of each to the whole as well as to individual parts of the parallel volume rendering algorithm.

II. Background

A brief background on parallel volume rendering is followed by a survey of parallel performance in large scale visualization. This section concludes with a look at trends in modern parallel architectures and the changing programming models to support them.

A. Parallel Volume Rendering

Figure 2 shows that the a parallel volume rendering algorithm consists of three main stages: I/O, rendering, and compositing. Each core executes I/O, rendering, and compositing serially, and this set of three operations is replicated in parallel over many cores. In the I/O stage, data are simultaneously read by all cores. The middle section, rendering, occurs in parallel without any intercore communication. The third stage, compositing, requires many-to-many communication using the direct-send method [9]. Finally, a completed image from a pipeline is either saved to disk or streamed over a network.

Sort-last parallel rendering methods [10] such as ours accommodate large data by dividing the dataset among nodes [11]. This approach can cause load imbalance in the rendering phase because of variations in scene complexity. Although it may seem that empty blocks should be fastest, the opposite is true in this algorithm. Early ray termination causes computation of the volume rendering integral along a ray to terminate once a maximum opacity is reached, but empty regions never reach this maximum opacity and are evaluated fully. Load imbalances occur only in the rendering phase; the compositing phase is inherently load balanced by dividing the resulting image into uniform regions such as scan lines [12]. Approaches to load balancing in the rendering phase can be either dynamic [13] or static [14], and distribution can occur in data space, image space, or both [15].

B. Performance

To measure performance, we compile the time required for each of the three stages in Figure 2, which sum to the total frame time, or the latency between viewing one time step to the next. (Frame rate is the reciprocal of frame time.) The relative costs of the three phases shift as the number of cores increases, but ultimately the algorithm is I/O bound [16]. Rendering performance scales linearly assuming perfect load balance, because it requires no interprocess communication. Cost of the direct-send compositing portion of the algorithm is dominated by many-to-many communication and becomes a significant factor beyond 2K cores and eclipses rendering time beyond 8K cores.

We do not use compression or multiple levels of detail, which can impose load imbalance and degrade visual quality. We do not preprocess the data to detect empty blocks, nor do we hierarchically structure the data as in [17]. Because our data are time varying, we attempt to simply visualize "on-the-fly," without doing any preprocessing between time steps.

In [18], Peterka et al. tested scalability up to 4K cores and concluded that leadership class supercomputers are viable volume rendering platforms for large datasets, that I/O cost dominates performance and must be mitigated, that rendering inefficiency is caused by load imbalance, and that compositing strategies such as direct-send cannot scale indefinitely. To summarize some other large visualization results, we survey a few notable examples from the literature in Table I.

From this list, one may conclude that it is possible to visualize structured meshes of 1 billion elements at interactive rates of several frames per second, including I/O for time varying data. Unstructured meshes can be visualized in tens of seconds, excluding I/O and preprocessing time. Large structured meshes of tens of billions of elements fall in between, requiring several seconds excluding I/O time. Lighting is typically not possible at these performance levels, and image sizes are usually limited to 1 megapixel. Our goal is to show scalability rather than interactive rates, but performance cannot be neglected either. Our results will demonstrate that the performance of this method fits within the context of the work in Table I. For example, we will show that 11 billion elements on a structured grid can be volume rendered, with lighting, in 16 seconds, including I/O.

C. Parallel Architectures and Programming Models

In addition to passing messages between nodes, sharing memory among cores is another way to achieve concurrency.

TABLE I
PREVIOUSLY PUBLISHED LARGE VOLUME RENDERING RESULTS

Dataset	Billion Elements	Mesh Type	Image Size	Time (s)	I/O incl.	Ref.
Molecular Dynamics	.14	unstructured	1Kx1K	30	no	[19]
Blast Wave	27	unstructured	1Kx1K	35	no	[19]
Taylor-Raleigh	1	structured	1Kx1K	0.2	yes	[20]
Fire	14	unstructured	800x800	16	no	[21]

Multicore architectures are ubiquitous today as chip makers strive to satisfy Moore's Law without the ability to increase clock speed and power indefinitely. Parallel graphics and visualization is evolving with this changing hardware. For example, Santos et al. [22] remark that the parallel visualization community began to address these architectures in 2007. [23] is one such paper that specifically targets multicore architectures to perform quality interactive rendering of large data sets. This trend continues in 2008 with many more papers targeting multicore hardware, for example, volume rendering on the cell broadband engine [24]. The number of cores in the Blue Gene PowerPC processor has doubled in the last generation of machines, and this trend is likely to continue. These hardware changes suggest that programming models will evolve to best harness the intra- and internode parallelism that is available. MPI has been the *de facto* parallel programming model for many years, but OpenMP [25], Intel Thread Building Blocks [26], and other libraries are gaining popularity for writing multithreaded programs.

III. METHOD

In this section we elaborate on details of the implementation. We begin by describing the nature of the dataset and BG/P architecture. We then discuss the program parameters that vary in order to generate results, and discuss how performance data are collected and analyzed.

A. Dataset

Our datasets originate from Anthony Mezzacappa of Oak Ridge National Laboratory and John Blondin of North Carolina State University and represent physical quantities during the early stages of supernova core collapse [27]. Variables such as density, pressure, and velocity are stored in netCDF [28] file format, in structured grids of size 864^3 and 1120^3. The two data sizes contain 0.65 and 1.4 billion elements per time step, respectively. Additionally, we created two simulated larger datasets by doubling each of the actual datasets in three dimensions. These simulated data are eight times larger than the original, or 5.2 and 11.2 billion data elements, respectively. In order to visualize the data, each time step is preprocessed to extract a single variable from the netCDF file and written in a separate file in 32 bit floating point format. With a single variable extracted, the file sizes for one time step of the actual and simulated datasets are 2.6, 5.6, 20.8, and 44.8 GB, respectively.

B. Architecture

Argonne National Laboratory's Leadership Computing Facility (ALCF) [2] operates a 557 teraflop (TF) BG/P supercomputer. Four PowerPC450 cores that share 2 GB of RAM constitute one BG/P node. Peak performance of one core is 3.4 gigaflops (GF), or 13.6 GF per node. One rack contains 1K nodes (4K cores), has 2 TB memory, and is capable of 13.9 TF peak performance. Eight racks equate to 16 TB RAM and 108 TF. The complete system contains 40 racks (40K nodes, 160K cores) 80 TB RAM, with peak performance of 557 TF. These relationships are diagrammed in Figure 3. IBM provides extensive online documentation of the BG/P architecture, compilers, and users' and programmers' guides [29].

C. Program Parameters

We built many controls into the volume rendering application. For example, the image mode is variable: not just the size of the finished image but whether it is saved to a file or streamed to a remote display. The number of time steps run in succession, including looping indefinitely over a finite time series, can be set. Lighting can be enabled or disabled; we enable lighting in order to demonstrate performance for high quality rendering. The density of point sampling along each ray can be controlled with a parameter as well.

A few more controls govern the arrangement of processes into parallel pipelines, multiple threads, and collective writers. The number of pipelines is variable up to the limit of the total number of cores available. For example, 8K cores can be configured in a single pipeline, two pipes of 4K cores each, etc. The rendering portion of an MPI process can be multithreaded with up to four POSIX pthreads. Each pipeline can output the final image in parallel, using a number of writer processes. Table II lists a sample of the parameters used to control the volume rendering code. With these parameters, the test setup is flexible and configurable. The meaning of most of these parameters will be clear when they are used to generate the results in Section IV.

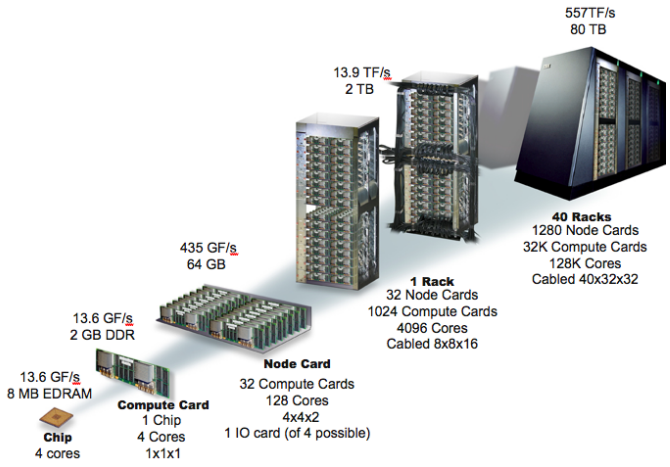

Fig. 3. Argonne's BG/P architecture.

TABLE II
SAMPLE PROGRAM PARAMETERS

Parameter	Example	Notes
DataSize	1120^3	Grid size (data elements)
ImageSize	1600^2	Image size (pixels)
NumProcs	16384	Total MPI processes
NumPipes	16	16 pipelines of 1K processes each
NumThreads	1	Threads per MPI process
NumNodes	4096	BG/P nodes (eg, vn mode)
NumWriters	64	Parallel image output processes
BlockingFactor	8	Round-robin data blocks per process

D. Emphasis on Quality

It is important for visualizations of scientific data to faithfully replicate detail and provide the highest fidelity, most enlightening view of the data possible. For example, when data are computed at high spatial resolution, high image resolution should be used to view the result so that the image area is comparable to the view area of the data volume. Besides image size, careful choice of ray sample spacing and the use of lighting models affect the quality of output images.

Lighting and shading add a high degree of information content and realism to volume rendering. For example, compare the two images of supernova angular momentum with and without lighting in Figure 4. The illuminated model, complete with specular highlights, resembles the appearance of isosurface rendering. This quality comes at a price, and although straightforward to compute, interactive volume renderers often omit lighting in order to maintain frame rate. With the extended potential for scaling that leadership class machines offer, we compute lighting as a matter of course. By estimating gradient direction from differences of neighboring vertex values, a normal direction is calculated on the fly for each data point, and a standard lighting model, including ambient, diffuse, and specular components, is computed [30].

Just as image size should be proportional to the length and width of the data volume, the sample spacing along each ray should be of the same order as the volume data spacing in the depth direction. This way, larger, higher resolution data are reproduced faithfully. Our algorithm generates samples along rays spaced at an adjustable fraction of the data spacing. We always set this parameter to one. Thus, the number of samples along each ray grows or shrinks with the number of data voxels, ensuring that image acuity matches data detail.

IV. RESULTS

This section quantifies gains from our main improvements to the volume rendering algorithm: load balancing, parallel output, multithreaded rendering, and parallel pipelines. Tests on large data and high numbers of cores validate the results.

A. Load Balancing

Load balancing is a difficult problem when it is in the context of large problem sizes, large numbers of cores, and time varying data. Uniform load balance is crucial to good scalability and efficiency, and much research has been published in this area of parallel computing. However, many of the published methods do not scale, particularly their communication costs, to thousands of cores under the performance constraints of time dependant data. Marchesin et al. [13] provide a dynamic load balancing method that requires data to be replicated on all cores; data replication is unacceptable for our problem scale. Childs et al. [19] describe a two phase rendering algorithm that balances work load by dividing a portion of it within object space and the rest in image space, but the cost of an additional many-to-many communication between the two stages may be prohibitive for our purposes. Ma et al. [31] show good scalability and parallel efficiency through a round-robin static distribution method.

Choosing a low cost but effective strategy, we implemented and tested the round-robin approach. Round-robin data partitioning is inexpensive because it is still a static balancing scheme but it has a good probability of achieving a roughly uniform load balance. The dataset is divided into more blocks than processors and the blocks are assigned to processors in

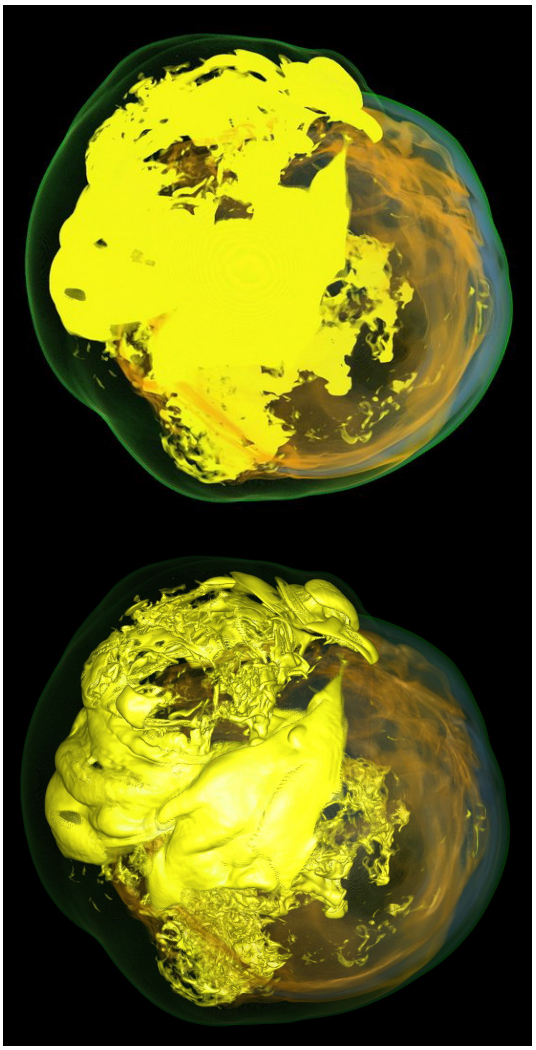

Fig. 4. Unlit (top) rendering vs. lit (bottom) rendering. Image quality that resembles triangle mesh isosurfacing can be produced with direct volume rendering.

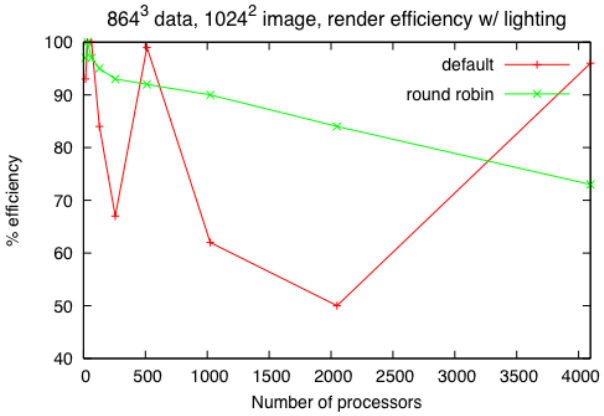

Fig. 5. Efficiency of the rendering phase can be improved by distributing data blocks in a more equitable manner. Round-robin balancing assigns many blocks to each core, distributed in a round-robin fashion. Compare efficiency the default distribution of one block per core.

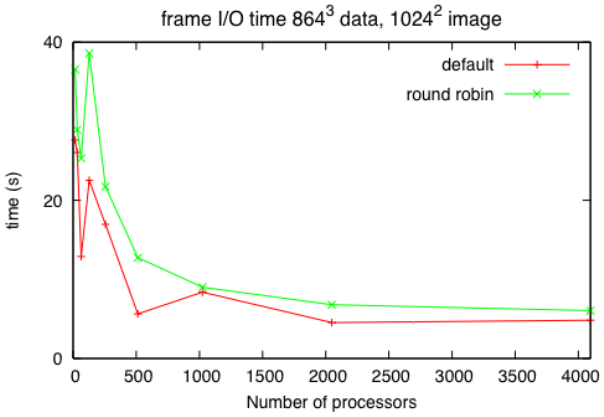

Fig. 6. There is additional I/O cost associated with round robin, since each process must read many more noncontiguous blocks. However, this cost is offset by increases in rendering efficiency.

a round-robin fashion. The number of blocks per processor is called the blocking factor: we have found 4, 8, or 16 to be sufficient for most cases.

The round-robin load distribution is surprisingly effective for increasing rendering efficiency, especially considering its low cost. Occasionally we need to hand tune the blocking factor to increase efficiency further. For example, in our tests we found 32 blocks per process to be better at 128 and 256 cores, but we can now smooth out previous bumps in efficiency curves easily through appropriate choice of blocking factor. Figure 5 compares rendering efficiency with and without round-robin balancing. The default case represents the uniform data division into the same number of blocks as processors. The round-robin case represents the improved load balance scheme.

Figure 5 shows that the round robin distribution is not perfect. For example, there are instances such as 512 and 4K cores where the default distribution is coincidentally very good, reaching over 90%. On average, however, the round-robin distribution is more predictable and in most cases better than the default, ranging between 70 and 90% over several thousand cores. The raw rendering time shows this improvement as well, completing faster than before in most cases. There is an I/O cost, however, in reading a number of nonadjacent blocks sequentially within each process, instead of just one or sometimes two. See Figure 6, which measures the I/O portion of time only. In all but two cases, however, the increase in I/O time was offset by a reduction in rendering time.

In those cases where the increased I/O cost remains, we will show later that I/O can be hidden effectively through multiple pipeline parallelism. Another potential optimization is to write a more intelligent I/O read function that batches the sequential reads that a process makes into a higher level collective read that can occur simultaneously. MPI-IO dictates that any collective file I/O operations occur in monotonically nondecreasing byte order at the file level. Hence, blocks in the subvolume would need to be decomposed into strings of contiguous bytes, and these strings sorted with respect to file byte order. Then a processes can read all of its blocks collectively, followed by reassembly into subvolume blocks once in memory. We are considering implementing this in the future.

To further test the overall performance of the algorithm with load balancing, we present overall timing results for four data sizes, 864^3, 1120^3, 1728^3, and 2240^3. The first two sizes are the actual data from Blondin et al.'s computational runs. We generated the latter two sizes by doubling the size of the original datasets in each direction, producing a file size eight times as large as the original.

The 1120^3 time step contains 1.4 billion data values. Figure 7 shows total frame rate (including I/O) for this data size, as a function of the number of cores. The total frame time at 8K cores is 7.9 s. Blocking factors for the round robin distribution range from 16 at lower numbers of cores to 4 at the upper end. The slope of the curve gradually decreases as compositing cost

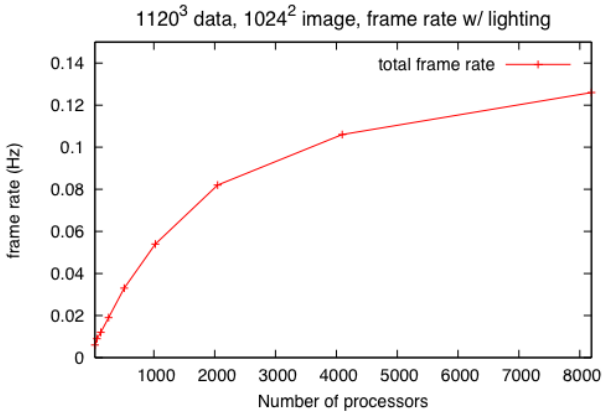

Fig. 7. The total frame rate with lighting is plotted for the 1120^3 dataset.

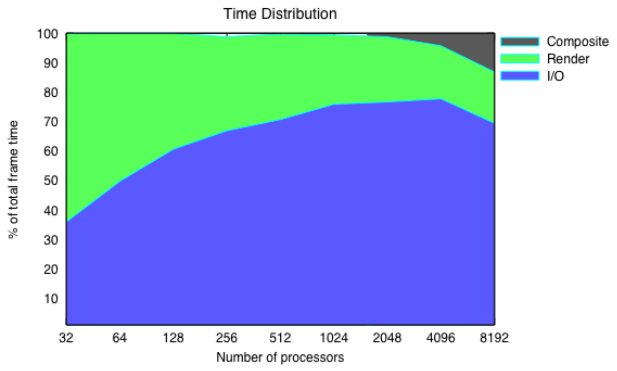

Fig. 8. At large data sizes and many cores, the relative contributions of the three stages of the algorithm are dominated by I/O. Compositing cost also grows.

grows with increased numbers of cores.

For the same test conditions in Figure 7, Figure 8 shows the distribution of time spent in I/O, rendering, and compositing. I/O dominates over the vast majority of the plot, and at 8K cores compositing is nearly as expensive as rendering. Extrapolating these these trends further out, rendering will be the least expensive of the three stages beyond approximately 10K cores. At the far right side of the time distribution, the relative percentage of I/O decreases. This is caused by the increasing cost of compositing, not because the I/O rate itself improved.

The total, end-to-end times for all four data sizes are listed in Table III. For comparison to other published results, we also include the visualization-only time in the right-hand column; this is the rendering time plus the compositing time, excluding I/O time. The image size is a constant 1024^2 pixels for these tests. At the scale of 16K cores, rendering has become the fastest of the three stages and compositing has grown to nearly 30% of the total frame time. Data movement, whether network communication or storage access, dominates the process at large scale.

B. Memory Conservation Through Parallel Writers

Memory is a scarce resource on BG/P. As we saw in Section III.B, 2GB are divided among 4 cores, or 512 MB per core in virtual node mode. Even though the compute node kernel only requires a few megabytes, it is still easy for an application to exceed the remaining 510 megabytes or so. Applications must conserve memory.

TABLE III
SIMULATED LARGE DATA SIZES

Grid Size	Elements (billion)	File Size (GB)	Cores	End-End Time (s)	Vis-Only Time (s)
864^3	.64	2.4	2K	3.6	0.8
1120^3	1.4	5.2	8K	7.9	2.4
1728^3	5.2	19.2	16K	15.6	5.2
2240^3	11.2	41.9	16K	16.4	5.2

The size of the data and the size of the image both determine how much memory is required by the volume rendering application. Memory requirements grow with problem size; that is unavoidable. We must ensure, nonetheless, that the amount of required memory decreases linearly with the number of cores, so that large problems can fit into memory by allocating more cores. Otherwise, the program will run out of memory at some data or image size, regardless of the number of cores.

That was exactly our situation. Even after optimizing memory usage to dynamically grow memory buffers only as needed, an underlying problem remained: the entire image needed to eventually reside on one core at the end of the process. The data structure that accompanies each ray requires significant memory per pixel, much more than just the R,G,B,A color values. The compositing process concludes with an MPI_Gatherv() operation that funnels all of the completed subimages to one core. This core tessellates the pieces into one image and saves it to disk. This serialization is a choke point in the algorithm. Performance suffers, memory usage does not scale, and image sizes beyond 1600^2 pixels crash the program.

The solution is to write images out in parallel with MPI-IO at the end of the algorithm, similar to the way that data are read in parallel at the start of the algorithm. For performance reasons, we want to control how many writing cores ("writers") we use. Too few writers result in the situation described previously. Too many writers hinder performance, because each writer writes very little. In this discussion and in Figure 9, terms such as writers, renderers, and compositors refer to the roles certain processes play at a given time. The same process takes on different roles during various stages of the algorithm. All processes act as renderers and compositors; later, a subset of processes become writers during the last stage.

We implemented a flexible scheme diagrammed in Figure

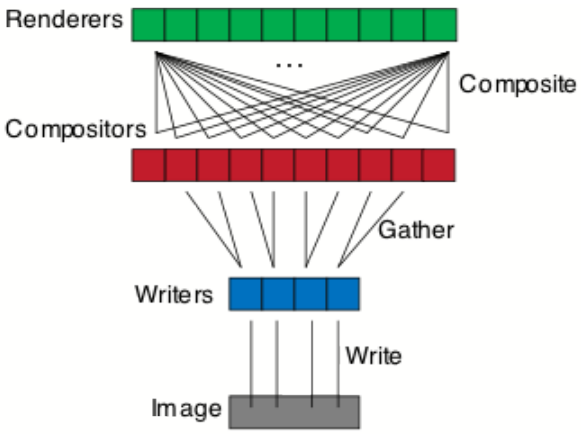

Fig. 9. The tail end of the algorithm consists of three substages: compositing, gathering down to a possibly smaller number of writers, and collectively writing the image to disk. Parallel image output is more scalable in terms of memory usage. No core needs to allocate space for the entire image, along with its associated data structures.

9. By creating several MPI communicators, compositing is followed by a partial gather down to the desired number of writers, and concludes with a collective image write to disk. Because the number of writers is a parameter, it can be any value from one to the total number of cores. Figure 10 shows that the optimal number of writers is 64 for this example. This test included 2 K cores, the 1120^3 dataset, and an image size of 2048^2 pixels.

In fact, now that memory is more scalable, we have successfully generated images as large as 4096^3, or 16 megapixels. After these improvements, the final memory usage model for this algorithm is:

$$m = 70M + 2.5Kp/c + 4v/c$$

Where:

1) m is the total memory usage in bytes
2) p size is the total number of pixels in the image
3) v is the total number of data elements in the volume
4) c is the total number of cores being used
5) *M, K* are one megabyte and one kilobyte, respectively

In this model, the memory requirements for data size and image size are divided by the number of cores. 70 MB are allocated by the MPI library, but we can control the rest of the memory usage by applying more or fewer cores to the volume rendering task.

Many data structures depend on the image size, including ray casting, compositing schedule, and the partial image itself. That is why the coefficient of p is so much larger than that of v. However, v itself is generally three orders of magnitude larger than p, so both image and data size end up contributing to the memory usage. We tested the accuracy of this model between 256 and 1024 cores, image sizes from 256^2 to 2048^2, and volume sizes up to 1120^3.

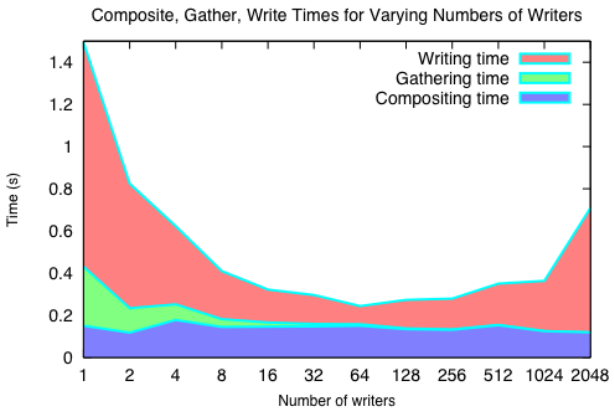

Fig. 10. Different numbers of writers affect overall output performance. We divide this stage into three substages: composite, gather, and write. The stacked graph shows the relative time that each of these substages takes when the number of writers is varied. 2 K compositors are used, and the image size is 2048^2 pixels.

C. Hybrid MPI - Multithreaded Rendering

Multicore computer architectures are ubiquitous today. BG/P is no exception: each node contains four cores that can be operated individually (virtual node mode), in pairs (dual mode), or as one unit (SMP mode). In SMP mode, 2 GB of memory constitute a common address space that is shared among the four cores. Thus far, we have not attempted to exploit this mode of operation, limiting ourselves to virtual node mode where each core acts as a separate processor with its own process space of 512 MB. With current architectures tending toward multi and many cores, however, it is necessary to develop new programming models that can exploit changing technology. We were curious what advantages could be gained by combining message passing and shared memory parallelism in our volume renderer. As usual, we wanted to test this at large system scale, so we modified a portion of the volume rendering code to include a hybrid multithreaded / MPI component, with the goal of measuring how the I/O, rendering, and compositing time compared to the original.

We changed the rendering portion of the volume renderer only, dictating whether an MPI process runs the rendering stage with one thread or with 4 threads within that process. In the first case, one process per *core* executes in virtual node mode, while in the second case, one process per *node* executes in SMP mode. BG/P's microkernel statically maps each of the four threads to one of the four cores within the node. Several programming APIs are available for thread management, including OpenMP and POSIX pthreads. We chose the latter for simplicity: thread creation was straightforward with pthreads and there was no scheduling advantage that OpenMP could provide within the context of BG/P's static allocation of threads to cores. In the following performance graphs, the first case is labeled "4 procs" (processes) while the second case is labeled "4 threads." "4 procs" implies the conventional MPI-only method and "4 threads" is the hybrid MPI - multithread method.

In the 4 threads method, the distribution of data space and image space is hybrid as well as the programming model. Among nodes and MPI processes, the data space is still divided into blocks, but among threads within a node, the image space is divided into 4 image regions. Ray casting is a natural algorithm to parallelize in image space, because each ray is independent of the others. The threads do not communicate with one another and the results of the cast rays are written to different addresses in the same data structure without need for synchronization or mutual exclusion. The only restriction is that all four threads complete the rendering stage before the program can proceed to the next stage, compositing.

When plotting results, we align values along the horizontal axis based on the same physical number of BG/P nodes, to enable a fair comparison. This way, the physical amount of hardware is the same. Some of the results matched our expectations, while others were a surprise. We predicted that even though the rendering step is where the code was changed, the rendering time should remain roughly constant in both

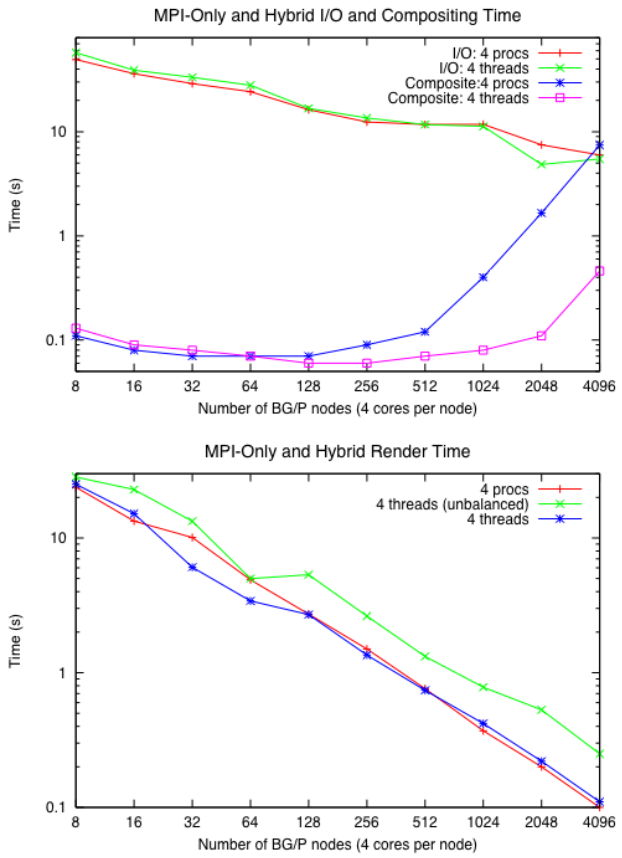

Fig. 11. Top: I/O and composite time vs. number of nodes when rendering is performed with 4 processes per node and 4 threads per node. Bottom: Similar test of rendering time. Rendering can be adversely affected by load imbalance, as the "4 threads (unbalanced)" curve shows.

scenarios. The total number of rays to be cast is the same, and since rendering requires no communication between either processes or threads, there should be little difference between the two approaches. On the contrary, we predicted that the I/O and compositing steps would be affected by the way that the program is executed. In virtual node mode, there are four times as many processes performing collective reading and four times as many participants exchanging messages.

Figure 11 shows our results for a test of one time step of the 1120^3 dataset, rendered to a 1024^2 pixel image. Both graphs compare aspects of the two programming models as the number of nodes increases. The top graph illustrates I/O and compositing time while the bottom graph depicts rendering time. The I/O time did not improve much with multithreading. At higher numbers of nodes, one might expect to see some I/O improvement because the dataset is divided into fewer, larger blocks. However, storage traffic passes through the same number of I/O nodes in BG/P, irrespective of the number of processes per compute node. Furthermore, MPI-IO executes collective I/O calls with the help of "aggregators." These can be thought of as a smaller number of actors that actually carry out I/O requests on behalf of MPI processes. The number of aggregators is also based on the number of I/O nodes, usually eight per I/O node, and this is a way for MPI-IO to avoid small, individual reads by grouping them into larger sets. These factors appear to have more significant impact on I/O rates than the changes that we made to the code.

The real win is in compositing. With fewer numbers of processes that need to exchange messages, compositing performance begins to improve at 128 nodes. With multithreading, the hybrid division of labor into both data space and image space takes advantage of the natural parallelism in ray casting. Rays within the same node exchange no information, and multithreading allows four subsets of rays to be computed in roughly the same time. This is the ideal parallel scenario. The compositing curves in the upper graph of Figure 11 show this improvement.

The lower graph confirms our hypothesis that rendering time does not improve with programming model, but it reveals a surprising fact. While multithreading did not speed up rendering, it can degrade rendering performance if not done carefully. We see that the "4 procs" and "4 threads" performance is virtually identical, except for minor deviations at 32 and 64 nodes. However, we also include a "4 procs unbalanced" result that is substantially slower. The cause of this slowdown, as the name suggests, is load imbalance between the four threads. Since all four threads must join before proceeding to compositing, the slowest thread dictates performance for the node. There is little to be gained from concurrency if the threads are imbalanced. As the lower graph shows, this can actually be slower than the MPI-only case, which has no such barrier per node.

Fortunately, we already solved this problem in another context. We can achieve interthread load balance using as similar technique as in Subsection A: round-robin distribution. This time, the distribution is in image space, but the idea is the same. Each thread operates on more than one region of the image, and those regions are interleaved in round-robin fashion. We call these regions stripes, and found that a striping factor of eight stripes per thread was sufficient to balance rendering load. The curve labeled "4 threads" in the bottom graph of Figure 11 is generated with eight stripes per thread. The interleaving of threads and stripes within a subimage follows the pattern below:

- Thread 0, Stripe 0
- Thread 1, Stripe 0
- Thread 2, Stripe 0
- Thread 3, Stripe 0
- Thread 0, Stripe 1
- ...
- Thread 3, Stripe 8

D. Multiple Parallel Pipelines

The previous sections demonstrate good scalability, but they also point out two problems. One is the diminishing return from higher numbers of cores in Figure 7, and the other is the widening gap between visualization-only time and end-to-end time in Table 2. Rendering time is not the bottleneck at scale: I/O dominates the frame time and further progress depends

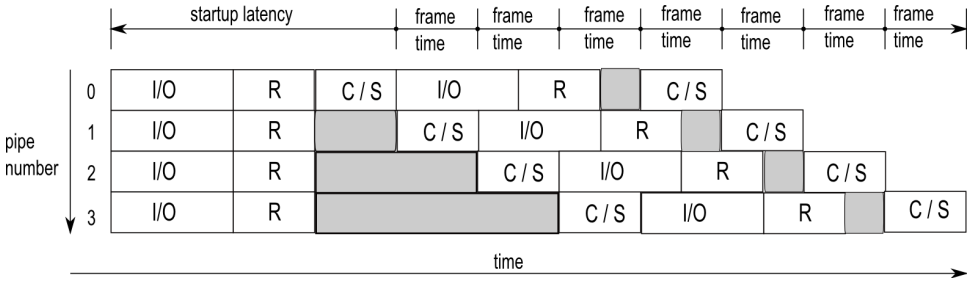

Fig. 12. Example of how four parallel pipelines can reduce frame time and hide I/O cost. I/O = file read, R = render, C/S = composite and send.

on hiding this I/O cost. In a time varying dataset, we cannot choose to simply ignore I/O time when measuring the frame rate because each time step or image frame requires a new file to be read. The I/O cost, however, can be mitigated and even completely hidden if sufficient I/O bandwidth, rendering resources, and communication bandwidth exist to process multiple time steps simultaneously.

The solution is two levels of parallelism: inter- time step and intra- time step parallelism using multiple parallel pipelines. In fact, even some of the rendering costs can be absorbed if the pipelines have sufficient overlap. As long as final frames are sent out in order, the client display software can buffer frames to smooth out any discrepancies in interframe latency and present final frames at a consistent frame rate.

Collections of cores are grouped into parallel pipelines [31]. Within any pipeline, many cores operate in parallel. Figure 12 shows a simplified diagram of a multipipe architecture for this application with four pipelines. Each pipeline is actually a collection of many cores operating in parallel on the same time step. The boxes are labeled I/O for file reading, R for rendering, and C/S for compositing and sending. All four pipelines begin at the same time because there is no need to synchronize them until the final stage. The only synchronization requirement is that images are output in order.

We can also impose further serialization within the pipeline interior, so that only one pipeline has control of the storage or communication network at any time. We call this a staged pipeline. It imposes a higher degree of serialization in exchange for less contention. An unstaged pipeline allows the maximum parallelism without concern for contention, ordering only the final sending of the resulting images. In our tests, staging of I/O or compositing did not significantly affect the frame time, but we have retained the staging feature in the code and will retest whether this has an effect at still larger scales.

Some idle time may occur within each pipe between the completion of rendering (R) and the start of either compositing or sending (C/S). This depends on the exact sum of the component times with respect to the number of pipelines. In the limit, however, multiple pipelines can reduce the frame time from the sum of I/O + R + C/S to just the C/S time, a significant savings.

Figure 13 shows the results of our experiments with 1, 2, 4, 8, and 16 pipelines arranged as follows:

- 1 pipe of 8 K cores
- 1 and 2 pipes of 4 K cores
- 1, 2, and 4 pipes of 2 K cores
- 1, 2, 4, and 8 pipes of 1 K cores
- 1, 2, 4, 8, and 16 pipes of 512 cores

The frame rate in Figure 13 is measured at the receiving display device; images are streamed to it as they are completed. An average frame rate is computed over all of the time steps received, so this is an end-to-end value that includes the entire system including I/O, rendering, compositing, and streaming. No compression is used for streaming or elsewhere in these tests. The graph shows five curves of various size pipelines, 512 to 8192 cores per pipeline. The horizontal axis shows the total number of cores used in the test. The number of pipelines for a given data point is the total number of cores divided by the number of cores per pipeline.

Performance improves along each curve with each doubling of the number of pipes. In fact, with 512 cores per pipe, the frame time (reciprocal of the frame rate) improved from nearly eighteen seconds for a single pipe to just over one second for sixteen pipes. At this point all of the I/O time is hidden along with a portion of the original single-pipe visualization time.

Figure 13 also demonstrates that performance improves for the same total number of cores, depending on how many pipelines they are arranged into. For example, 8K total cores

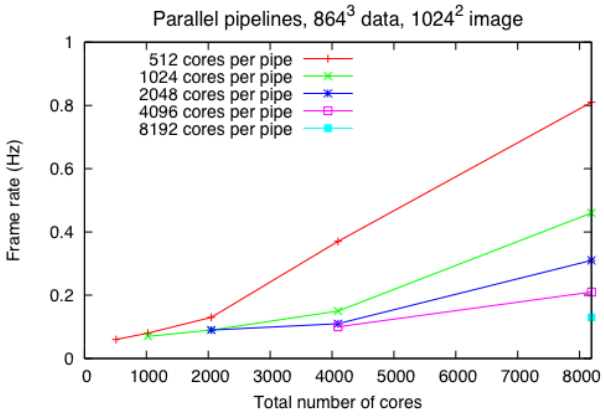

Fig. 13. Multiple pipelines can provide several times faster performance, even if the same total number of cores is distributed into several pipes instead of a single pipe.

arranged as 16 pipes of 512 cores produces a frame rate that is six times faster than 8K cores in a single pipe. Intersecting the curves of 13 with any vertical grid line produces a similar result: for a constant total number of cores, faster times are attained by arranging the cores into a greater number of smaller size pipelines. We conclude that the overlap of operations in Figure 12 contributes more than simple scaling of the number of cores because the total end-to-end time is I/O bound and does not scale linearly. In other words, hiding the I/O time via multiple pipelines is an effective tool to counterbalance I/O cost.

V. Discussion

By improving load balancing, conserving memory through parallel output, combining MPI with multithreaded programming, and employing multiple pipelines, we have extended the scale of high quality time varying volume rendering to over 10 billion data elements per time step. By scaling to over 10,000 cores, we can generate results at frame times on the order of several seconds, including I/O and lighting.

A simple round-robin load distribution scheme achieves two times better balance than naive single-block allocation. With extreme numbers of cores, this may be the best that can be achieved without the cost of load balancing outweighing its benefit. More complex redistribution of data at these scales, within performance constraints, has yet to be achieved. Even round-robin distribution carries increased I/O costs, but these can be offset through improved rendering efficiency and multipipe parallelism.

Changes in memory allocation can extend the scalability of our volume renderer. By outputting the image in parallel, memory use scales inversely with the number of rendering cores. Now, arbitrarily large volumes and images can be accommodated by allocating enough cores to the problem. We implemented this feature using several MPI communicators and gathering the number of compositors down to a smaller number of writers. By making the number of writers a program parameter, we can test variable numbers of writers, from one to the total number of cores.

Computer architectures are becoming increasingly multicore. Another way to improve the performance of parallel machines is to exploit the intranode parallelism that exists on chip. By allocating one MPI process per node and one thread per core for the rendering portion of the algorithm, we have found that a hybrid programming model can increase the I/O and compositing performance at large system scale. Rendering cost remains approximately constant whether threads or processes are used, but the overall program performance can improve due to the improvements in the other parts of the algorithm.

The multiple pipeline organization effectively hides I/O costs in time varying datasets by processing multiple time steps simultaneously. It is often more efficient to arrange a fixed number of cores into more pipelines of fewer cores each, than to group all of the cores into a single pipeline.

With these improvements, it is technically feasible to apply leadership-class machines at large scales to visualization and analysis problems such as volume rendering. The remaining question is, "Is this use of resources justified?" While such nontechnical questions are outside of the scope of this research, the remainder of this section presents a few arguments why we believe that this use of valuable resources is desirable, justifiable, and surprisingly economical. Foremost, we are creating the foundation for in situ visualization. Not only does this offer significant savings in terms of time and data movement, but having access all of the simulation data in situ affords new capabilities, such as simultaneous analysis of multiple variables. Interacting with the simulation, not just the visualization, is another advantage. Numerous other possibilities exist in this exciting new research area.

In order to faithfully resolve detail in large datasets, large display devices such as tiled walls and accompanying large image sizes (tens and hundreds of megapixels) are required. Otherwise, the display size and image resolution effectively down-sample the dataset to a much coarser level of detail. Data are discarded just as if the dataset had originally been much smaller, except that information is discarded at the end of a potentially long and expensive visualization workflow. Larger display and image sizes require orders of magnitude larger visualization systems than are currently available in the graphics cluster class of architectures. Leadership-class machines are currently the only choice when considering not only large data size, but also high image resolution.

In terms of scheduling and machine utilization, in our experience, visualization jobs interleave easily within other computational runs. Our visualization runs are short, usually only a few minutes. Over the course of a week, these short runs accrue no more than a few hours of total CPU time. Even if 16K cores are required for a total of 10 hours per week (more than we have used to date), this is still only a fraction of one percent of the utilization of a 500 TF machine. A job scheduler can back-fill the unused cycles between scheduled computational runs with analysis and visualization tasks. These cycles would be wasted otherwise, so the visualization is essentially free.

VI. Future Work

We continue to scale up to larger data and more cores. In so doing, new bottlenecks appear. The next hurdle to overcome requires rewriting the compositing part of the algorithm to employ binary swap [32]. More efficient compositing is one of the priorities for successful operation at tens of thousands of cores.

Improving I/O performance is another. The ALCF researchers continue to improve aggregate I/O bandwidth and I/O scalability; these improvements are welcome because they directly affect the performance of this application. We are investigating different I/O file formats and ways to increase the aggregate I/O bandwidth of our application as well.

ACKNOWLEDGMENT

We thank John Blondin and Anthony Mezzacappa for making their dataset available for this research. This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357. Work is also supported in part by NSF through grants CNS-0551727, CCF-0325934, and by DOE with agreement No. DE-FC02-06ER25777.

REFERENCES

[1] (2008) Scidac institute for ultra-scale visualization. [Online]. Available: http://ultravis.ucdavis.edu/

[2] (2008) Argonne leadership computing facility. [Online]. Available: http://www.alcf.anl.gov/

[3] K.-L. Ma, C. Wang, H. Yu, and A. Tikhonova, "In-situ processing and visualization for ultrascale simulations," *Journal of Physics*, vol. 78, 2007.

[4] T. Tu, H. Yu, L. Ramirez-Guzman, J. Bielak, O. Ghattas, K.-L. Ma, and D. R. O'Hallaron, "From mesh generation to scientific visualization: An end-to-end approach to parallel supercomputing," in *Proc. Supercomputing 2006*, Tampa, FL, 2006.

[5] H. Yu, K.-L. Ma, and J. Welling, "A parallel visualization pipeline for terascale earthquake simulations," in *Proc. Supercomputing 2004*, 2004, p. 49.

[6] P. K. Yeung, D. A. Donzis, and K. R. Sreenivasan, "High-reynolds-number simulation of turbulent mixing," *Physics of Fluids*, vol. 17, no. 081703, 2005.

[7] P. Woodward. (2008) Laboratory for computational science and engineering. [Online]. Available: http://www.lcse.umn.edu/index.php?c=home

[8] L. Chen, I. Fujishiro, and K. Nakajima, "Optimizing parallel performance of unstructured volume rendering for the earth simulator," *Parallel Computing*, vol. 29, no. 3, pp. 355–371, 2003.

[9] U. Neumann, "Communication costs for parallel volume-rendering algorithms," *IEEE Computer Graphics and Applications*, vol. 14, no. 4, pp. 49–58, 1994.

[10] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs, "A sorting classification of parallel rendering," *IEEE Computer Graphics and Applications*, vol. 14, no. 4, pp. 23–32, 1994.

[11] B. Wylie, C. Pavlakos, V. Lewis, and K. Moreland, "Scalable rendering on pc clusters," *IEEE Computer Graphics and Applications*, vol. 21, no. 4, pp. 62–69, 2001.

[12] A. Stompel, K.-L. Ma, E. B. Lum, J. Ahrens, and J. Patchett, "Slic: Scheduled linear image compositing for parallel volume rendering," in *Proc. IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, Seattle, WA, 2003, pp. 33–40.

[13] S. Marchesin, C. Mongenet, and J.-M. Dischler, "Dynamic load balancing for parallel volume rendering," in *Proc. Eurographics Symposium of Parallel Graphics and Visualization 2006*, Braga, Portugal, 2006.

[14] K.-L. Ma and T. W. Crockett, "A scalable parallel cell-projection volume rendering algorithm for three-dimensional unstructured data," in *Proc. Parallel Rendering Symposium 1997*, 1997, p. 95.

[15] A. Garcia and H.-W. Shen, "An interleaved parallel volume renderer with pc-clusters," in *Proc. Eurographics Workshop on Parallel Graphics and Visualization 2002*, Blaubeuren, Germany, 2002, pp. 51–59.

[16] H. Yu and K.-L. Ma, "A study of i/o methods for parallel visualization of large-scale data," *Parallel Computing*, vol. 31, no. 2, pp. 167–183, 2005.

[17] J. Gao, C. Wang, L. Li, and H.-W. Shen, "A parallel multiresolution volume rendering algorithm for large data visualization," *Parallel Computing*, vol. 31, no. 2, pp. 185–204, 2005.

[18] T. Peterka, H. Yu, R. Ross, and K.-L. Ma, "Parallel volume rendering on the ibm blue gene/p," in *Proc. Eurographics Parallel Graphics and Visualization Symposium 2008*, Crete, Greece, 2008.

[19] H. Childs, M. Duchaineau, and K.-L. Ma, "A scalable, hybrid scheme for volume rendering massive data sets," in *Proc. Eurographics Symposium on Parallel Graphics and Visualization 2006*, Braga, Portugal, 2006, pp. 153–162.

[20] J. Kniss, P. McCormick, A. McPherson, J. Ahrens, J. S. Painter, A. Keahey, and C. D. Hansen, "Interactive texture-based volume rendering for large data sets," *IEEE Computer Graphics and Applications*, vol. 21, no. 4, pp. 52–61, 2001.

[21] K. Moreland, L. Avila, and L. A. Fisk, "Parallel unstructured volume rendering in paraview," in *Proc. IS&T SPIE Visualization and Data Analysis 2007*, San Jose, CA, 2007.

[22] L. P. Santos, D. Reiners, and J. Favre, "Parallel graphics and visualization," *Computers and Graphics*, vol. 32, no. 1, pp. 1–2, 2008.

[23] C. P. Gribble, C. Brownlee, and S. G. Parker, "Practical global illumination for interactive particle visualization," *Computers and Graphics*, vol. 32, no. 1, pp. 14–24, 2008.

[24] J. Kim and J. Jaja, "Streaming model based volume ray casting implementation for cell broadband engine," in *Proc. Eurographics Parallel Graphics and Visualization Symposium*, Crete, Greece, 2008.

[25] (2008) Openmp. [Online]. Available: http://openmp.org/wp/

[26] (2008) Intel thread building blocks. [Online]. Available: http://www.threadingbuildingblocks.org/

[27] J. M. Blondin, A. Mezzacappa, and C. DeMarino, "Stability of standing accretion shocks, with an eye toward core collapse supernovae," *The Astrophysics Journal*, vol. 584, no. 2, p. 971, 2003.

[28] (2008) Netcdf. [Online]. Available: http://www.unidata.ucar.edu/software/netcdf/

[29] (2008) Ibm redbooks. [Online]. Available: http://www.redbooks.ibm.com/redpieces/abstracts/sg247287.html?Open

[30] N. L. Max, "Optical models for direct volume rendering," *IEEE Transactions on Visualization and Computer Graphics*, vol. 1, no. 2, pp. 99–108, 1995.

[31] K.-L. Ma and D. M. Camp, "High performance visualization of time-varying volume data over a wide-area network," in *Proc. Supercomputing 2000*, Dallas, TX, 2000, p. 29.

[32] K.-L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh, "Parallel volume rendering using binary-swap compositing," *IEEE Computer Graphics and Applications*, vol. 14, no. 4, pp. 59–68, 1994.