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

Tom Peterka[1], Robert Ross[1], Hongfeng Yu[2], Kwan-Liu Ma[2]

[1]Argonne National Laboratory, [2]University of California, Davis

Direct correspondence to tpeterka@mcs.anl.gov

Abstract

As computational science progresses toward petascale, innovations in analysis and visualization of the resulting datasets are necessary. Rather than evaluating visualization performance by rendering speed only, we take a systemwide view and evaluate where bottlenecks actually are when scaling parallel software volume rendering on the IBM Blue Gene/P (BG/P) to over 10,000 cores. We first examine the relative costs of I/O, rendering, and compositing for a variety of dataset sizes, and we show the benefit of an alternative rendering distribution scheme that improves load balance. Next we examine the cost of adding high-quality lighting in the rendering phase. We also investigate the use of multiple parallel pipelines to partially hide the I/O cost when rendering many time-steps. We conclude that BG/P is an effective platform for volume rendering of large-scale datasets and that the rendering capabilities of BG/P cores are not the limiting factor in performance.

Classification, Keywords

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

In the face of the petascale era, innovative visualization technologies will be required to keep pace with dramatically increasing datasets. One idea is not an innovation at all: supercomputer software rendering is a throwback to the past, before the proliferation of hardware-accelerated graphics. Figure 1 shows an example of modern software volume rendering applied to astrophysics data. This research through the U.S.

Department of Energy's SciDAC Institute for Ultra-Scale Visualization [4] applies software volume rendering to a very new supercomputer, the IBM Blue Gene/P (BG/P) at Argonne National Laboratory.

While parallel volume rendering algorithms are well known, unprecedented scales are tested here: over 10,000 cores are employed to target ever-increasing data sizes. Our testing at these scales has produced several new optimizations to the method and uncovered the need for others. The new contributions of this paper are threefold: a significant increase in efficiency due to improved load balancing, use of high-quality lighting and shading to add visual realism and visual content, and multiple levels of parallelism to hide I/O cost. We demonstrate the implementation of these contributions at scales of up to 16K cores.

Much of the visualization research in recent years has focused on the exploitation of GPU hardware to increase performance by representing volumetric data as 3D textures. Software rendering rates are several orders of magnitude slower than graphics hardware and clearly software volume rendering cannot compete with GPU methods in the same space of problems. However, GPU methods, while highly parallel at the hardware level, are limited by video memory size and degrade in performance the farther the data are removed from the rendering hardware: across the CPU-GPU bus, on another physical node, or in storage.

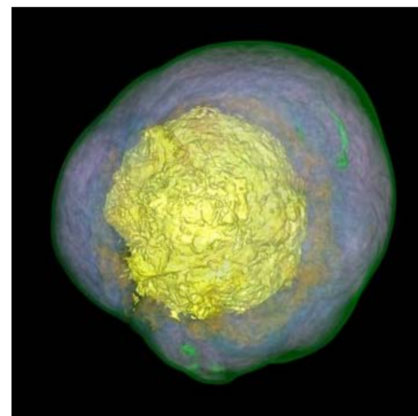

Figure 1: Software volume rendering of the early stages of supernova core collapse

When raw rendering time is not the bottleneck in end-to-end performance, a modern supercomputer such as the BG/P becomes a viable alternative to a graphics cluster. With its tightly coupled interconnection backbone, parallel file system, and massive amounts of cores, such an architecture scales more predictably once the problem size grows to billions and tens of billions of data elements, where factors such as I/O and communication bandwidth - not rendering speed - dominate the overall performance.

The opportunity for in situ visualization is another advantage associated with enabling visualization on the same architecture as the simulation. Because postprocessing and transporting data can take as long as its computation, moving the analysis and visualization closer to the data is usually less expensive than the other way around. In situ visualization [16], [24], [30], or overlapping visualization with an executing

simulation, is a largely uncharted area ripe for discoveries; we hope that the results presented here will contribute to the advancement of in situ visualization and data analysis.

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 [28], and their next target is 4096^3 using the IBM Blue Gene architecture. This equates to approximately 270 GB per time-step per variable. At these scales, every extra iteration through the dataset is extremely costly in terms of both time and storage space; hence, overlapping computation with analysis into a single step is the only practical way to extract information from the simulation results. Similar dataset sizes are currently being produced by Woodward et al. [26], again in the area of fluid dynamics, and by Chen et al. [7] in earthquake simulation. Both of these groups are working within the context of shared-memory architectures. Chen et al., for example, perform both simulation and visualization using the NEC Earth Simulator.

Concepts implemented in this paper such as round-robin load distribution or multiple parallel pipelines have been published earlier. Our contribution is to evaluate these concepts in the context of massive parallelism, improve them as necessary to operate in this environment, and note additional areas needing improvement. In so doing, we address common challenges that appear across architectures and problem domains in parallel computing: parallel I/O, load balancing, and compositing of partial results to a final solution. These parallelization problems are stress-tested in this work and the lessons learned may be broadly applied to other computing, analysis, and visualization algorithms such as unstructured meshes or adaptive mesh refinement (AMR) [12], [25]. Other supercomputer architectures and large graphics clusters face similar parallelization problems as they scale up, and perhaps so will future architectures of massively parallel collections of GPUs or cell processors.

2. Background

Sort-last parallel rendering methods [19] accommodate large data by dividing the dataset among nodes [27]. This approach can cause load imbalance in the rendering phase because of variations in scene complexity. Although it might seem that empty blocks should be fastest, the opposite is true in this algorithm. Early ray termination causes the volume rendering integral computation 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 [23]. Approaches to load balancing in the rendering phase can be either dynamic [17] or static [14], and distribution can occur in data space, image space, or both [10].

Figure 2 shows that the 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 is simultaneously read by all cores of a pipeline. The middle section, rendering, occurs in parallel without any intercore communication. The third stage, compositing, requires many-many communication using the direct-send method [21]. Finally, a completed image from a pipeline is either saved to disk or streamed over a network.

Times are compiled for each of the three stages and add up to a 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 [29]. Rendering performance scales linearly assuming perfect load balance because it requires no interprocess

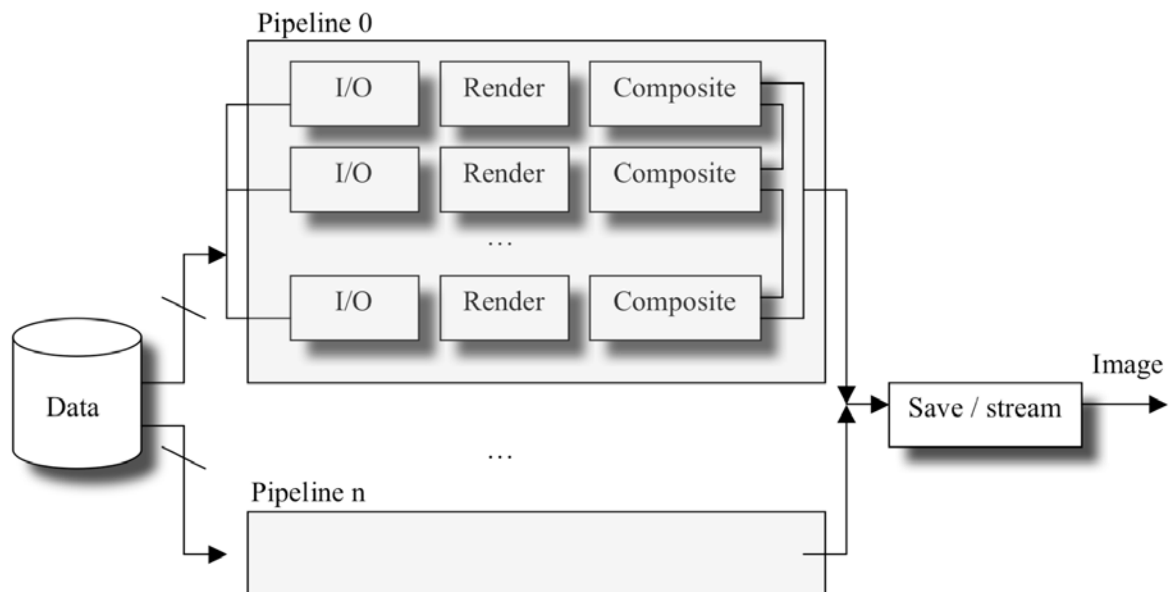

Figure 2: Functional diagram of the parallel volume rendering algorithm

communication. Cost of the direct-send compositing portion of the algorithm is dominated by many-to-many communication and becomes a significant factor beyond 4K cores and eclipses rendering time beyond 8K cores. Both strong scaling and weak scaling are tested: performance for the same dataset is tabulated across both increasing numbers of cores and increasing data sizes.

We do not use compression or multiple levels of detail, which can impose extreme 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 [9] because the preprocessing costs are incompatible with our time-varying data context.

A single-pipeline version of the structure in Figure 2, tested by Peterka et al. [22], resulted in the following conclusions:

1. Leadership-class supercomputers are viable volume rendering platforms for data > 1 billion elements.
2. I/O cost dominates performance and must be mitigated.
3. Rendering inefficiency is caused by load imbalance.
4. With enough cores, the performance budget allows higher-quality illumination models to be used.
5. Many-many compositing strategies such as direct-send will not scale effectively beyond 8K cores.

This paper presents solutions to items 2, 3, and 4, further reinforcing the claim made in item 1. Work is ongoing for item 5.

As a benchmark for our results, we review the landscape of comparable large dataset visualization results in the literature. Table 1 lists some of those results. From a survey of current performance we conclude that it is possible to visualize structured meshes of 1 billion elements at interactive rates of several frames per second, including I/O time 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.

Table 1: Previously published large volume rendering results

Dataset	Billion Elements	Mesh Type	Image Size	Time (s)	I/O	Reference
Molecular dynamics	0.14	unstructured	1K x 1K	30	no	[8]
Blast wave	27	unstructured	1K x 1K	35	no	[8]
Taylor-Raleigh	1	structured	1K x 1K	0.2	yes	[11]
Fire	14	unstructured	800x800	16	no	[20]

3. Method

In this section we elaborate on details of the implementation. We begin by describing the nature of the dataset and of the 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.

3.1 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 supernovae core collapse [6]. Variables such as entropy, angular momentum, density, pressure, and velocity are stored in netCDF [3] file format. Data elements are stored in structured grids, although we expect the majority of our research to be applicable to unstructured grid and AMR data in the future as well.

In order to visualize the data, each time-step is preprocessed to extract a single variable from the netCDF file and to write it in 32-bit floating-point format into a separate file. Currently we have two sizes of actual data, 864^3 and 1120^3 data elements, or 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, making them 8 times larger, or 5.2 and 11.2 billion data elements, respectively. File sizes of the two actual datasets and two simulated datasets are 2.6, 5.6, 20.8, and 44.8 GB per time-step, respectively.

3.2 Architecture

Argonne National Laboratory is installing a 557 teraflop (TF) BG/P supercomputer, designated by the U.S. Department of Energy as one of two leadership computing facilities in the nation. The Argonne Leadership Computing Facility (ALCF) [1] currently has a single-rack testing and development machine and an eight-rack 100 teraflop machine. Both are functional and available for early adopters and INCITE users now, subsets of the 557 TF BG/P.

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 or 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 final system is projected to have 40 racks (40K nodes, 160K cores) 80 TB RAM, and 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 [2].

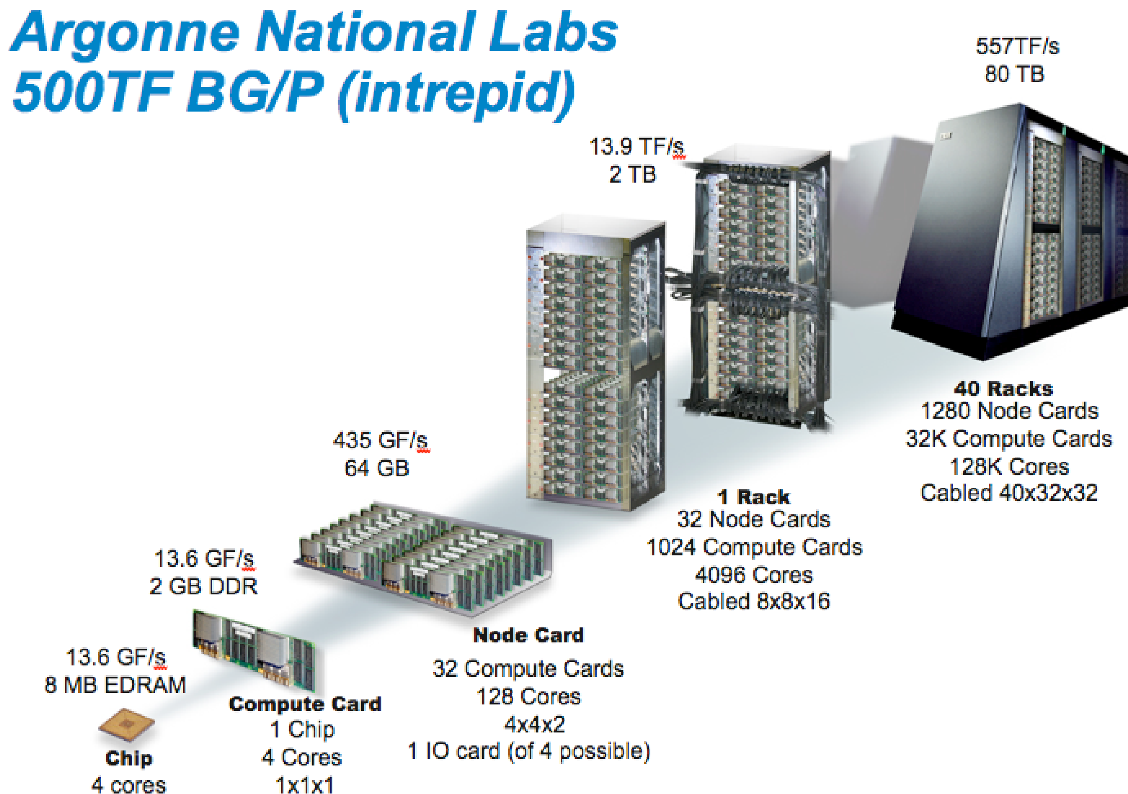

Figure 3: Argonne's BG/P architecture

3.3 Program Parameters

Several controls for testing performance under a number of conditions have been designed 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 on the network. 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. We always set this to full density; as with lighting, we do not trade quality for speed. Unlike [8], the sample spacing along a ray is a factor of the data spacing, not a fixed value. Hence, the final resulting sample spacing along a ray depends on the data size and the image detail faithfully reproduces the data.

A few more controls concern the multiple-pipeline architecture for processing several time-steps simultaneously. 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. Additionally, the amount of overlap between pipeline execution is adjustable by permitting the I/O and compositing phases of the pipes to occur simultaneously or in alternating order. The purpose of this parameter is to gauge the amount of contention between the pipes for storage and communication bandwidth. We call the two pipeline modes *staged* and *unstaged.*

3.4 Performance Data Collection

Performance data are collected by using timing measurements as outlined in [22], including the total frame time as well as its constituents: I/O time, render time, and composite time. When streaming to a remote display, the final net frame rate at the display device is also measured. This is the frame rate that the scientist would see when viewing the data. Since the dataset is time-varying, the end user is seeing the end-to-end time, including file I/O, in the final frame rate at the display.

Profiling tools assist in gathering high-frequency performance data, in understanding subtle relationships such as percentage of execution time spent in various program functions or the time distribution across all cores of a particular section of the algorithm, and in querying all of the hardware counters that BG/P makes available. Our favorite tool is TAU (Tuning and Analysis Utilities) [5] from the University of Oregon. Figure 4 shows several of the outputs from TAU. The upper half of the figure shows a call-graph representation: the size of the function blocks represents the fraction of execution time spent in each.

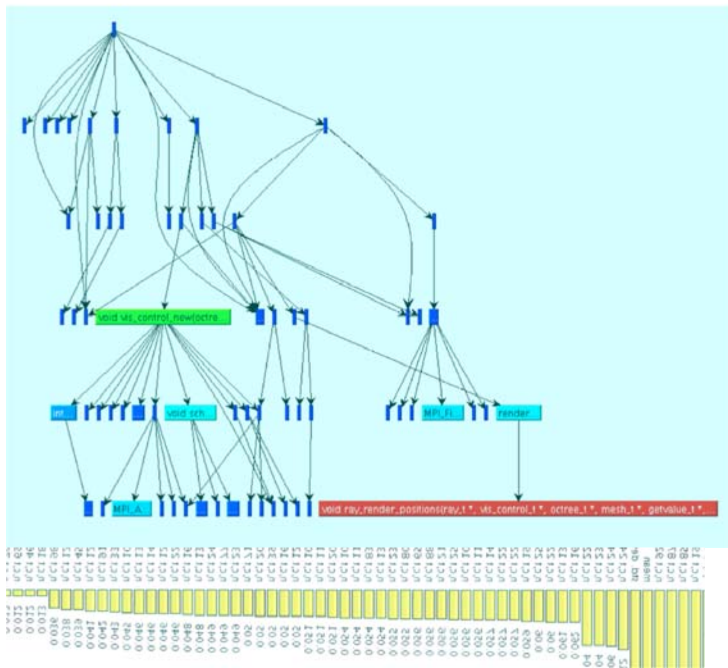

Figure 4: Profiling tools assist in comparing performance between program modules (top) and between cores executing the same module (bottom).

Comparing call-graphs between runs illuminates how the balance of time spent in different parts of the code changes with scale. The lower half of Figure 4 shows a time distribution of the rendering phase in a single run across 256 cores. Load imbalances are quickly evident from the uneven distribution. Statistics such as mean and standard deviation are also accessible.

4. Results

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

4.1 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. Clearly a uniform load balance is crucial to good scalability and efficiency,

and much research has been published in this area of parallel computing. However, it is not clear whether any of the published methods are lightweight enough, particularly their communication costs, at thousands of cores under the performance constraints of time-dependant data. Marchesin et al. [17] 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. [8] describe a two-phase rendering algorithm that balances work load by dividing a portion of the workload 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. [14] show good scalability and parallel efficiency through a round-robin static distribution method.

We take a low-cost but effective strategy that improves efficiency in some cases by a factor of 2 compared to a uniform data distribution. Round-robin data partitioning is inexpensive because it is still a static balancing scheme but it has a better probability of achieving a roughly uniform load balance. The dataset is divided into many more blocks than processors and the blocks are assigned to processors in a round-robin fashion. The number of blocks per processor is called the *blocking factor*; we have found 16 to be sufficient for most cases.

The round-robin load distribution is surprisingly effective for increasing rendering efficiency, given 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, while the round-robin

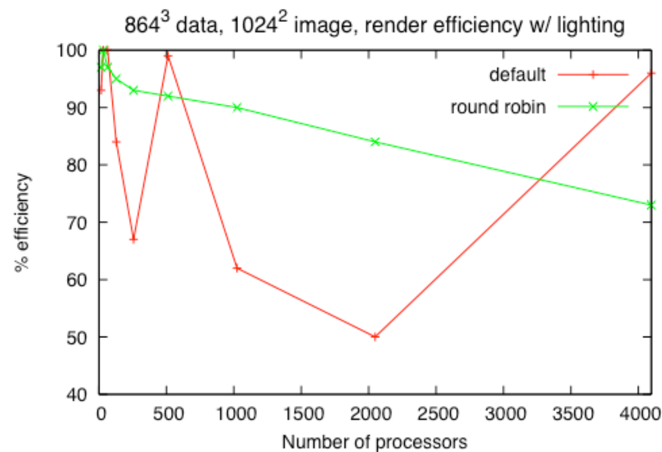

Figure 5: Efficiency of the rendering phase can be improved by distributing data blocks in a more equitable manner. Round-robin balancing assigns many blocks to each core, distributed in a round-robin fashion. Compare with the default distribution of one block per core.

case represents the improved load balance scheme.

It is easy to see the difference between the behavior of the two curves in Figure 5. The efficiency is based on rendering time only, which requires no communication. 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 16 or 32 nonadjacent blocks sequentially within each process, instead of just one or 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. This has not been implemented yet.

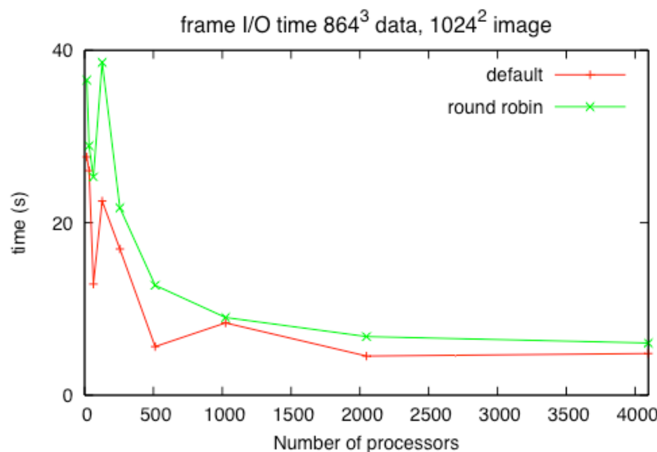

Figure 6: There is additional I/O cost associated with round robin, since each process must read many more non-contiguous blocks. However, this cost is offset by increases in rendering efficiency.

4.2 Lighting Large Data

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 7. The illuminated model, complete with specular highlights, closely resembles the appearance of isosurface rendering. This quality comes at a steep price, however, which is why most parallel volume renderers cannot afford to use it.

By estimating gradient direction from differences of neighboring vertex values, a normal direction is calculated for each data point. A standard lighting model, including ambient, diffuse, and specular components, is computed [18]. Although straightforward to compute, lighting is often omitted from other volume renderers in order to boost performance. For example, the 864^3 dataset in Figure 7, rendered to a 1024^2 image requires approximately seven times longer to compute the effects of lighting. With the extended potential for scaling that leadership-class machines offer, however, we can now compute lighting within volume rendering as a matter of course.

The next largest dataset that we have available is 1120^3, or 1.4 billion, vertices. Currently we have one time-step of each of five variables: density, pressure, and x,y,z components of velocity (approximately 5.3 GB per variable). Once again this is part of a time-varying dataset and we will eventually transfer more time-steps to Argonne's BG/P. This performance test visualizes the pressure variable to a 1024^2 image size, with lighting. The resulting image appears in Figure 1. For this test, the number of cores scales from 32 to 8K by factors of 2.

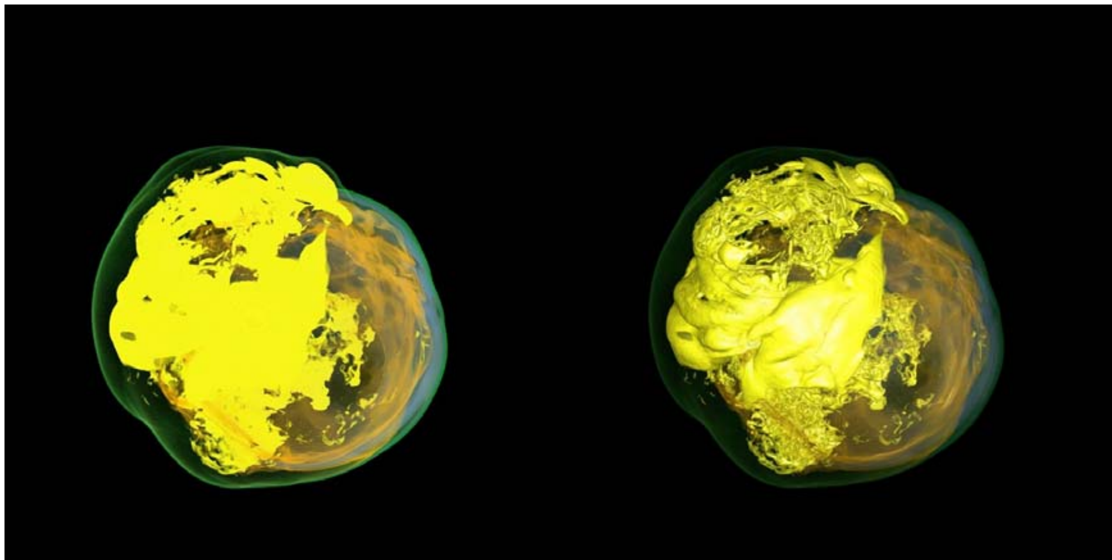

Figure 7: Unlit (left) rendering vs. lit (right) rendering. Image quality that resembles triangle mesh isosurfacing can be produced with direct volume rendering.

Figure 8 shows total frame rate (including I/O), as a function of the number of cores. The total frame time at 8K cores is 7.9 s. Round-robin data distribution is used, with blocking factors that range from 16 at lower numbers of cores to 4 at the upper end. The slope of the curve gradually decreases because compositing overhead assumes a more prominent role with increased numbers of cores.

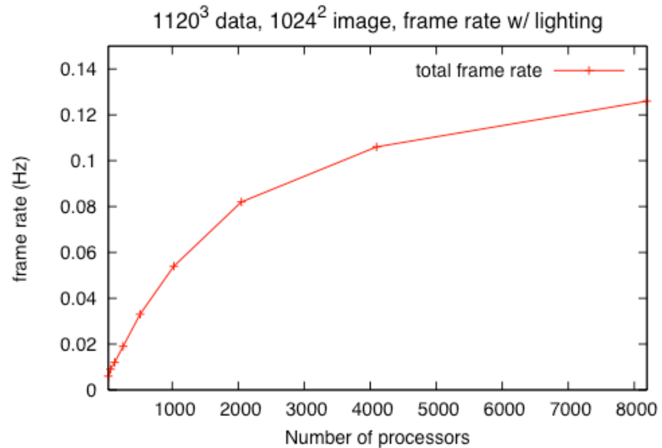

Figure 8: The total frame rate with lighting is plotted for the 1120^3 dataset.

This underlines the need to rewrite the compositing algorithm before scaling much further. Figure 9 shows the distribution of time spent in I/O, rendering, and compositing for this same dataset. Clearly, the I/O dominates over the vast majority of the plot, and at 8K cores compositing is nearly as expensive as rendering. Beyond approximately 10K cores, rendering will be the least expensive of the three stages.

As a temporary measure until larger data are available, we have taken each of the 864^3 and 1120^3 datasets and doubled the volume in each of the three dimensions, extending total size by a factor of eight. The performance on these simulated datasets are in Table 2 and constitute our largest scale to date: 16K cores. These values include lighting. At this scale, rendering is the fastest of the three stages; compositing consumes nearly 30% of the total frame time. The right-most column includes rendering and compositing time, but excludes I/O. As we shall see in the next section, it is possible to mitigate

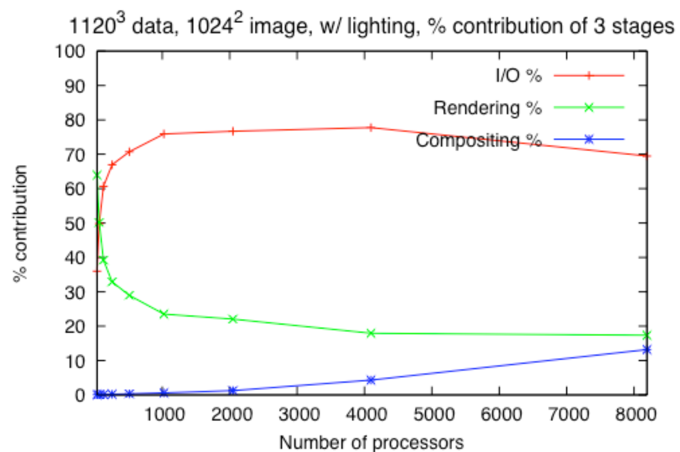

Figure 9: At large data sizes and many cores, the relative contributions of the three stages of the algorithm are dominated by I/O. Compositing cost also grows.

Table 2: Simulated very large data sizes

Original Size	Concatenated Size	Elements (billion)	File Size (GB)	Image Size (pixels)	End-End Time (s)	Vis-Only Time (s)
864^3	1728^3	5.2	20	1K x 1K	15.6	5.2
1120^3	2240^3	11.2	42	1K x 1K	16.4	5.2

the cost of I/O time through the use of multiple parallel pipelines.

4.3 Multiple Parallel Pipelines

The previous section demonstrates extreme levels of scalability, but it also points out two problems. One is the diminishing return from higher and higher amounts of cores in Figure 8, and the other is the widening gap between visualization-only time and end-to-end time in Table 2. We conclude that rendering time is not the bottleneck at this problem scale: I/O dominates the frame time and further progress depends 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 receiving-end display software can buffer frames to smooth out any discrepancies in interframe latency and present final frames at a consistent frame rate.

In Figure 2, collections of cores are shown grouped into parallel pipelines [13]. Within any pipeline, many cores operate in parallel. Figure 10 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 in order that sending occurs in order.

When I/O or compositing are regulated so that one pipeline at a time has control of the storage or communication network, 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, so we assume that storage and network bandwidths are not saturated at scales up to 8K cores. We are currently examining peak aggregate I/O rates for the parallel file system and studying the communication patterns of the BG/P torus to verify this assumption. 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. The total number of processes must be divisible by the number of pipes, and usually these are both powers of two. Therefore, the number of pipes will usually result in imperfect

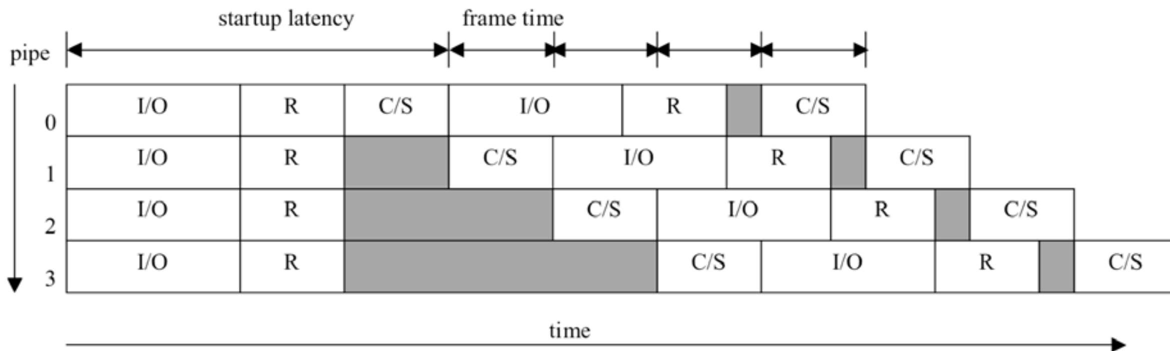

Figure 10: Example of how four parallel pipelines can reduce frame time and hide I/O cost. I/O = file read, R = render, C/S = composite and send.

utilization and some dwell time. 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 11 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 11 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.

In all cases, performance improves 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.

One might argue that performance improves because so many more total cores are being used. However, Figure 11 shows that the difference in frame rate is significant even for the same total number of cores, depending on how many pipes the cores constitute. For example, 8K total cores arranged as 16 pipes of 512 cores produces a frame rate that is six times faster than 8K cores in a single pipe. Similar but less dramatic improvements appear in Figure 11 for the other combinations

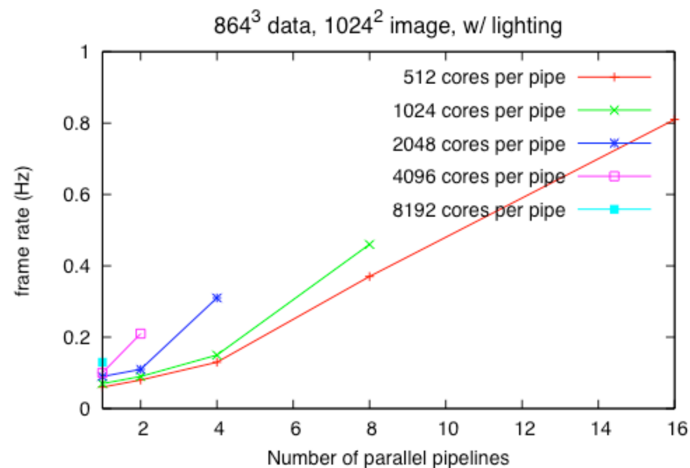

Figure 11: Multiple pipelines can provide several times faster performance, even if the same total number of cores is distributed into several pipes instead of a single pipe.

of the same total number of cores arranged in different ways. The overlap of operations in Figure 10 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. Hiding the I/O time via multiple pipelines is an effective tool to counterbalance I/O cost.

5. Discussion

By improving load balancing, adding lighting, 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 up to the order of 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 does naïve single-block allocation. With extreme numbers of cores, maybe this is 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.

Lighting adds a new degree of visual fidelity to volume rendering but the computational expense has limited its use to small data until now. While the expense is the same, software volume rendering on leadership architectures such as BG/P provides new opportunities for scalability, far beyond sizes of graphics clusters. Moreover, lighting calculations do not fit graphics clusters' texture-based hardware accelerations very well. Storing of normal vectors as textures can require more video memory than the original data, and computing them per pixel is expensive because it requires normalization.

The multiple pipeline organization effectively hides I/O costs in time-varying datasets by processing multiple time-steps simultaneously. Even if multiple pipelines of the original number of cores cannot be formed, for example if the cores are not available, it is still more efficient to allocate existing cores into more than one pipeline of fewer cores each than to group all of the cores into a single pipeline.

We have shown that it is technically feasible to apply leadership-class machines at large scales to visualization and analysis problems and we have explained how to do so. The remaining question is, "*Why* should others consider doing this?" There are several reasons why 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 the availability of 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 resolution effectively down-sample the dataset to a much coarser level of detail. Data is discarded just as if the dataset had originally been much smaller, except that detail is lost at the end of the visualization workflow, a waste of all of the previous computational resources. 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 can be interleaved easily within other computational runs. Our runs are typically short, lasting 10-30 minutes. Over the course of a week, these short runs accrue no more than a few hours of total time. Even if 16K cores are required for a total of 10 hours per week (much more than we have used to date), this is still only a fraction of one percent of the utilization of a 500TF machine. Since visualization runs are much shorter than simulations, the scheduler can easily back-fill unused cycles between scheduled computational runs with analysis and visualization tasks. These cycles would be wasted otherwise, so the visualization is essentially free.

6. 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 tree-based binary swap [15]. This code is being written but is not ready for testing yet. Although aware of the issue for some time, we have successfully avoided this problem up until now–but no longer. Efficient compositing is one of the priorities for successful operation at tens of thousands of cores.

Improving I/O performance is another focus area. Motivated by round-robin data distribution, we are writing a collective I/O routine that enables each process to read multiple blocks in a single operation. 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 will also be exploring other programming models to leverage the memory that is shared by the four cores of single BG/P node. This technique should combine MPI with OpenMP or other thread-level parallelism methods and may result in a hybrid message passing / shared memory model of parallel visualization and analysis.

Acknowledgments

We 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. 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] Argonne Leadership Computing Facility. http://www.alcf.anl.gov/ 2008.
[2] IBM Redbooks. http://www.redbooks.ibm.com/redpieces/abstracts/sg247287.html?Open 2007.
[3] NetCDF. http://www.unidata.ucar.edu/software/netcdf/ 2008.
[4] SciDAC Institute for Ultra-Scale Visualization. http://ultravis.ucdavis.edu/ 2007.
[5] Tuning and Analysis Utilities. http://www.cs.uoregon.edu/research/tau/home.php 2008.
[6] 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.
[7] CHEN, L., FUJISHIRO, I. NAKAJIMA, K.: Optimizing Parallel Performance of Unstructured Volume Rendering for the Earth Simulator. *Parallel Computing*, 29, 3, (March 2003), 355-371.
[8] 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.
[9] GAO, J., WANG, C., LI, L. SHEN, H.-W.: A Parallel Multiresolution Volume Rendering Algorithm for Large Data Visualization. *Parallel Computing*, 31, 2, (February 205), 185-204.
[10] GARCIA, A. SHEN, H.-W.: An Interleaved Parallel Volume Renderer with PC-clusters. *Proceedings of Eurographics Workshop on Parallel Graphics and Visualization 2002*, Blaubeuren, Germany, (2002), 51-59.
[11] KNISS, J., MCCORMICK, P., MCPHERSON, A., AHRENS, J., PAINTER, J. S., KEAHEY, A. HANSEN, C. D.: Interactive Texture-Based Volume Rendering for Large Data Sets. *IEEE Computer Graphics and Applications*, 21, 4, (July/August 2001), 52-61.

[12] 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.
[13] MA, K.-L. CAMP, D. M.: High Performance Visualization of Time-Varying Volume Data over a Wide-Area Network. *Proceedings of Supercomputing 2000*, Dallas, TX, (November, 2000), 29.
[14] MA, K.-L. CROCKETT, T. W.: A Scalable Parallel Cell-Projection Volume Rendering Algorithm for Three-Dimensional Unstructured Data. *Proceedings of Parallel Rendering Symposium 1997*, (1997), 95.
[15] 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.
[16] MA, K.-L., WANG, C., YU, H. TIKHONOVA, A.: In-Situ Processing and Visualization for Ultrscale Simulations. *Journal of Physics*, 78, (June 2007).
[17] MARCHESIN, S., MONGENET, C. DISCHLER, J.-M.: Dynamic Load Balancing for Parallel Volume Rendering. *Proceedings of Eurographics Symposium of Parallel Graphics and Visualization 2006*, Braga, Portugal, (May 2006).
[18] MAX, N. L.: Optical Models for Direct Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1, 2, (June 1995), 99-108.
[19] MOLNAR, S., COX, M., ELLSWORTH, D. FUCHS, H.: A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Applications*, 14, 4, (July 1994), 23-32.
[20] 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).
[21] NEUMANN, U.: Communication Costs for Parallel Volume-Rendering Algorithms. *IEEE Computer Graphics and Applications*, 14, 4, (July 1994), 49-58.
[22] PETERKA, T., YU, H., ROSS, R. MA, K.-L.: Parallel Volume Rendering on the IBM Blue Gene/P. *Proceedings of Eurographics Parallel Graphics and Visualization Symposium 2008*, Crete, Greece, (April 2008).
[23] 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.
[24] 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).
[25] 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.
[26] Laboratory for Computational Science and Engineering. http://www.lcse.umn.edu/index.php?c=home 2008.
[27] WYLIE, B., PAVLAKOS, C., LEWIS, V. MORELAND, K.: Scalable Rendering on PC Clusters. *IEEE Computer Graphics and Applications*, 21, 4, (July/August 2001), 62-69.
[28] YEUNG, P. K., DONZIS, D. A. SREENIVASAN, K. R.: High-Reynolds-Number Simulation of Turbulent Mixing. *Physics of Fluids*, 17, 081703, (August 205).
[29] 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.
[30] YU, H., MA, K.-L. WELLING, J.: A Parallel Visualization Pipeline for Terascale Earthquake Simulations. *Proceedings of Supercomputing 2004*, (November 2004), 49.

The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne"). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.