Visualization and parallel I/O at extreme scale

R B Ross[1], T Peterka[1], H-W Shen[2], Y Hong[2], K-L Ma[3], H Yu[3], and K Moreland[4]
[1]Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, USA
[2]Department of Computer Science and Engineering, The Ohio State University, Columbus, OH 43210, USA
[3]Department of Computer Science, University of California at Davis, Davis, CA 95616, USA
[4]Sandia National Laboratories, Albuquerque, NM 87185, USA

E-mail: rross@mcs.anl.gov

Abstract. In our efforts to solve ever more challenging problems through computational techniques, the scale of our compute systems continues to grow. As we approach petascale, it becomes increasingly important that all the resources in the system be used as efficiently as possible, not just the floating-point units. Because of hardware, software, and usability challenges, storage resources are often one of the most poorly used and performing components of today's compute systems. This situation can be especially true in the case of the analysis phases of scientific workflows. In this paper we discuss the impact of large-scale data on visual analysis operations and examine a collection of approaches to I/O in the visual analysis process. First we examine the performance of volume rendering on a leadership-computing platform and assess the relative cost of I/O, rendering, and compositing operations. Next we analyze the performance implications of eliminating preprocessing from this example workflow. Then we describe a technique that uses data reorganization to improve access times for data-intensive volume rendering.

1. Introduction

As with theory and experimentation, computation has become a critical component of scientific discovery. In the quest to find answers to ever more challenging questions through computational techniques, we find ourselves constructing ever larger and more capable systems. Today we are pushing the scale of compute systems to reach a petaflop and beyond, and it has become increasingly important that all resources in the system are used as efficiently as possible. While we commonly use the floating-point operation rate as a measure of the performance of large-scale compute systems, most members of the scientific computing community realize that other characteristics of the system, such as the rate of memory access, more often determine the resulting application performance.

Because advances in storage have trailed behind those of processors and networks, data access has become a serious challenge for many applications running at large scale. Storage systems are being hampered by a number of issues. First, storage hardware bandwidths are not increasing at nearly the rate

of processing hardware, meaning that more devices (disks) must be used in each successive generation of storage systems in order to keep pace with compute capabilities. Second, while significant algorithmic advances have taken place in scientific applications, much storage software – namely commercial parallel file systems – are using algorithms, interfaces, and architectures designed over a decade ago for much smaller scale systems. Working around the deficiencies of these old approaches can be very difficult. Third, scientists are often not using storage systems in the most effective manner. Tuning of data access is typically one of the last steps taken when improving an application code to operate more efficiently at scale, so many applications still perform I/O the same way they did many years ago, on very different systems. This might mean that all processes in a parallel application push data through a single node, or it might mean that every process creates its own independent data file when checkpointing, neither of which is likely to result in high performance. High-level libraries such as HDF5 [1] and Parallel netCDF [2] have been developed to provide more natural and efficient interfaces when used properly, but adopting these interfaces is a time-consuming task, and even with these libraries significant tuning is often necessary.

The analysis phase brings an additional set of I/O challenges. The conventional computational workflow has the simulation application deposit results on storage, and those results are processed by independent analysis tools as a separate step, after the simulation is complete. The separation of these two phases results in significant traffic to and from the storage system, because any data needed for analysis purposes must be stored for later access. The end result is that applications spend significant time waiting on I/O to complete, or they store less data, or store data less often, than they might otherwise. Moreover, the data written by the simulation code may not be in a form amenable to analysis, resulting in I/O patterns that make poor use of the storage hardware, even if the analysis code itself has been written using best practice access methods. These challenges require their own specific solutions, distinct from those developed to help with checkpointing and other output during the simulation phase, and developing these solutions typically involves both analysis and I/O expertise.

The SciDAC Institute for Ultra-Scale Visualization is working to enable extreme-scale knowledge discovery by advancing the state of visualization technologies, fostering awareness of and communication about new visualization technologies, and by putting these technologies into the hands of application scientists, especially SciDAC application teams. Institute research falls into one of three areas (figure 1). Work in architectures for visualization explores the viability of various types of compute resources with respect to scientific visualization. In improving and developing new fundamental algorithms, we seek to enable new visualization capabilities or apply existing capabilities to new types of data. Efforts in interactivity, interfaces, and tools seek to provide new ways to view and interact with data that help scientists more quickly and easily gain insight from their results. Overall, these three areas fit together to cover visualization challenges from end-to-end: from the underlying computational resources up to the user.

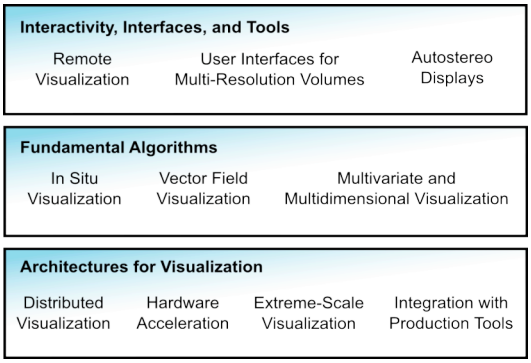

Figure 1. Research topics in the Institute.

Because of the technical challenges and the importance of effective visual analysis tools for extreme scale data, the Institute has a number of ongoing efforts in this area. One such effort is the addition of a parallel I/O layer to Paraview/VTK. This layer allows for the use of MPI-2 [3] I/O routines to improve performance on large-scale systems by eliminating bottlenecks in the data path, while serial I/O options are provided for portability. A small set of readers using this new parallel I/O layer are under

development, and preliminary benchmarking is under way. By enhancing the widely used Paraview tool, we hope to enable more effective I/O for a broad range of applications.

Additional research is necessary to develop solutions for the analysis of data at extreme scales. In the following two sections we discuss Institute efforts in visualization for extreme scale data. In Section 2 we cover our work in volume rendering on the IBM Blue Gene/P architecture, and in Section 3 we describe our work in data organization for efficient I/O during visualization. In Section 4 we conclude and point to possible future approaches for reducing the I/O demands of analysis.

2. Volume rendering of large-scale data

One example of the Institute's research efforts is parallel volume rendering on leadership-class resources such as the Blue Gene/P at Argonne National Laboratory. Using the same compute architecture for visualization as for simulation can provide three benefits for large-scale data:

- Data movement from the simulation architecture to a visualization architecture can be avoided.
- The massive resources of a leadership machine, including high-speed interconnection network and parallel file system, can be brought to bear on the visualization problem.
- This is an intermediate step toward in situ visualization.

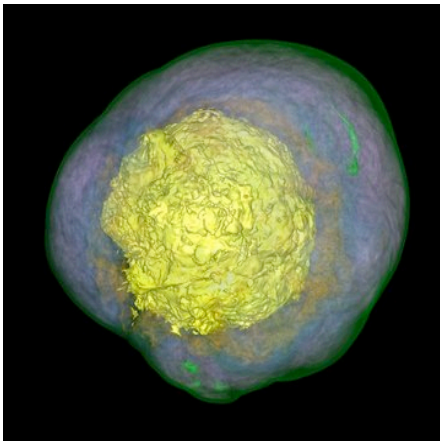

Figure 2. Image resulting from volume rendering of astrophysics dataset on IBM Blue Gene/P. Original image is 1600x1600 pixels.

At first glance, using a leadership computing resource for analysis might not make sense. The vector parallelism provided by modern GPUs enables peak rendering rates that are on average 100 to 1000 times faster than the software rendering rate of a low-power general-purpose processor such as BG/P's PowerPC. However, modern datasets that are gigabytes or larger per time step demand parallel processing for visualization, including parallel I/O. In this scenario, the bottleneck is not rendering time, but data movement between storage and processors. Thus, we want a visualization system that excels at these tasks, not solely by rendering. Fortunately, such visualization architectures, with robust parallel I/O and communication infrastructures already exist in the BG/P and Cray XT systems.

This work is in collaboration with John Blondin of North Carolina State University and Anthony Mezzacappa of Oak Ridge National Laboratory, who are working to understand the origin of spherical accretion shock instability (SASI) in supernova core collapse [4]. The simulation seeks to understand how the SASI is driven by global acoustic modes trapped in the supernova interior. By manipulating the transparency of rendered data (i.e., the transfer function) in our volume rendering code, the scientists can quickly see different combinations of variables and isolate features of interest. In their dataset, five scalar variables are stored in regular grid locations up to 1120^3, in netCDF format. Up to 200 time steps are stored in a simulation run, each in a separate file.

2.1. Phases of a visual analysis algorithm

Most parallel visual analysis algorithms can be divided into four main tasks: input, render, composite, and output. In classical parallel visualization with smaller datasets and systems, it was customary to ignore input costs and focus on rendering and compositing. However, the expanding size and complexity of data shift the burden onto the I/O system, meaning this cost must be recognized and included in our analyses as an integral component of parallel visualization [5].

Parallel rendering algorithms are usually grouped into one of three types: sort-first, sort-middle, and sort-last [6]. We use the sort-last algorithm, which distributes the dataset among processors first and then gathers the partial results last. This is appropriate when the size of data dominates the problem size, as it does in most modern scientific visualizations that are data-intensive.

During the input phase, cores read a time step in parallel. By using MPI-2 features such as collective I/O as an interface to the parallel file system, all cores simultaneously read only their designated portion of the dataset. This approach increases I/O bandwidth due to parallel data paths between compute cores and storage, compared to a single data pathway from storage to a root node. Moreover, the entire dataset need not fit into the memory of a single core; the memory requirement is distributed among all of the cores.

Parallel input of a dataset tacitly assumes that a predefined domain decomposition of the data has already been established. Distribution can be static or dynamic; while dynamic distribution has been shown to produce good load balance at small data and system scales [7], it is costly at large scales and we do not believe it appropriate for tens of thousands of cores. Rather, we implement a static scheme that decomposes a time step into block-shaped subvolumes. By dividing into a multiple of blocks per processor core (usually 4–16) and allocating cores to blocks in a round robin manner, we have balanced load to greater than 50% rendering efficiency across 16K cores. As the next paragraph explains, some decrease in efficiency is unavoidable even with perfect load balance, because optimizations act locally, not globally across the data volume.

The rendering algorithm within a single core is classic software ray casting [8] with some optimizations and quality improvements, with the goal of maximizing rendering quality and minimizing per-frame preprocessing costs. Once the core has its subvolume of data and has applied the transfer function to map a physical quantity into a chromatic one, it casts rays from all pixels through its data, sampling the data at evenly spaced intervals along each ray. The spacing along a ray is controllable and proportional to the data spacing. Lighting is computed and includes diffuse, specular, and ambient components [9]. Next, the sample points along each ray are blended front-to-back using the "over" operator [10] to compute a single (R, G, B, A) pixel value. This process terminates prematurely if a maximum opacity is reached, called early ray termination. Other optimizations such as hierarchical levels of detail, octree data structuring, or empty space skipping are not used in our algorithm.

Compositing, like other classical parallel visualization tasks, has been extensively published but not at the system scales that we address in our research. We currently use the direct-send algorithm, mainly for its simplicity [11]. Direct-send is a two-step process. First, each process receives subimages from those processes that contribute to the portion of the image that the process is responsible for compositing. The process performs the same "over" blending operation that is performed for an individual ray during the rendering phase, but this time blending completed pixels from the subimages. Second, all processes send their completed subimages to the root. Unlike rendering, which is only approximately load balanced, the work in the compositing stage is evenly divided among processes. We have demonstrated effective capability of direct-send compositing up to 16K cores. Our results show nearly constant compositing time through 8K cores, and then a gradual increase beyond that. Network contention accounts for the majority of the compositing time at this point. In the first step a large number of small messages are exchanged, and each process has only a few pixels to composite. In the second step the finished subimages are gathered at the root process using MPI_Gather. Both communication steps are expensive at this scale. Because binary swap compositing [12] has lower complexity in the number of messages, we plan to analyze its performance at these scales in the future.

Thus far, image sizes have been limited to 2K x 2K pixels, or 4 megapixels, so output bandwidth to storage has not been a bottleneck. This will change as images become larger, and we will experiment with collective writes of the output image to eliminate the need to send all subimages to one root process, which can be a point of contention in the communication system.

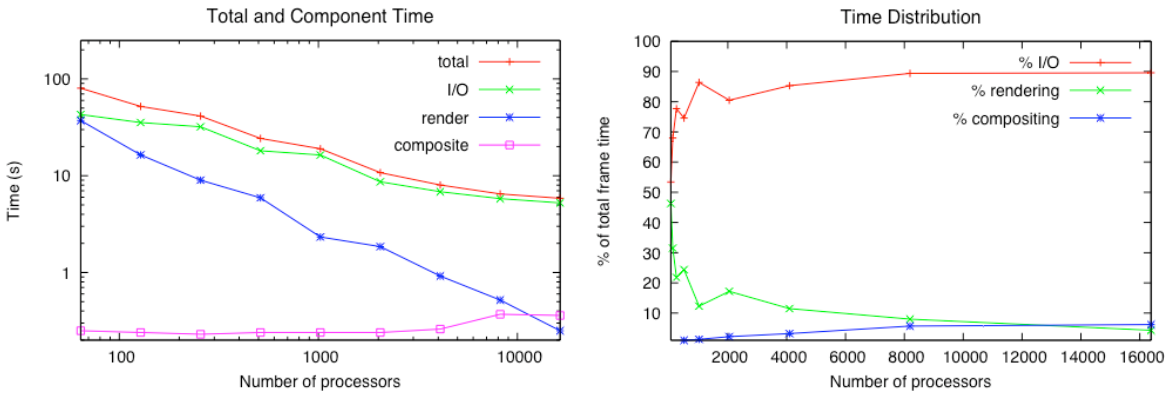

Figure 3. The time distribution between three stages of the volume-rendering algorithm for a 1600^2-pixel image, in seconds (left) and as a percentage of run time (right). Time is dominated by I/O.

2.2. Experimental results

Figure 3 shows how the duration that one time step requires is distributed among I/O, rendering, and compositing, from 64 to 16K cores. The jagged steps at smaller numbers of cores are caused by imperfect load balancing at certain number of processes. Overall, the time is dominated by I/O, consuming nearly 90% at 16K cores. Beyond 16K, compositing is more expensive than rendering, which is one indication that a new compositing algorithm is needed if we expect to scale further.

The dependence of large-scale visualization on I/O performance, made clear by figure 3, demonstrates the need to (1) continue to improve the I/O bandwidth and latency of the parallel file systems of architectures such as BG/P, and (2) devise ways to hide or mitigate the effects of I/O latency within visualization applications. While storage researchers attempt to improve storage performance overall through techniques such as advanced I/O forwarding [13], in Section 3 we discuss one effort in the Institute to reduce the I/O demands of visual analysis algorithms.

2.3. To preprocess or not to preprocess

Simulation output files are often preprocessed in order to reduce data sizes prior to moving data across the wide area, to extract variables of interest during analysis, or to transform the data into a format more amenable to processing by general-purpose analysis tools. In this context we consider this operation "preprocessing" because it must be performed before the analysis phase (in other contexts it might be termed "postprocessing" because it occurs after the simulation). This preprocessing step doesn't generate any new results, and as such it can be thought of as overhead in the scientific workflow. Reducing or completely eliminating preprocessing, especially for multivariate time varying data, presents an opportunity to improve the I/O performance of an application.

In our example astrophysics dataset, each time step is a netCDF file consisting of five record variables: pressure, density, and velocity in three directions. Each of these five quantities is a scalar value on a regular 3D Cartesian grid. For a grid size of 1120^3, one time step is approximately 25 GB. In the past, we would preprocess each time step to extract a single variable of interest, for example pressure, and write this into a separate, 32 bit raw data format. We would then proceed to volume render the sequence of pressure time steps, as described above.

This approach is expensive and needs to be streamlined if we expect it to scale further. First, there is the cost of preprocessing time. The extraction of a single variable from a single time step of the above

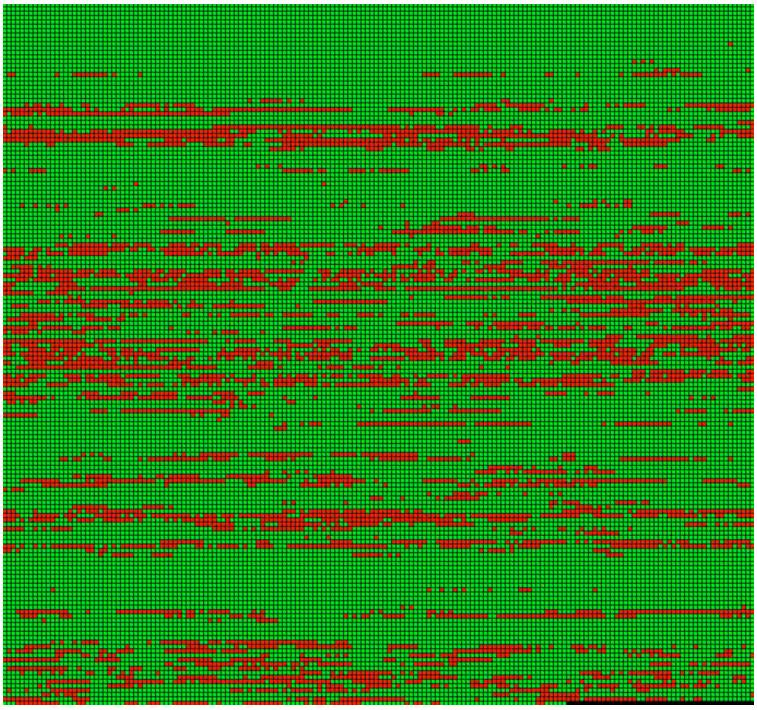

Figure 4. Depiction of a 2-dimensional representation of data blocks from the Visible Woman dataset stored on a single I/O server. Green (light) boxes represent data blocks that were not accessed during the sample operation. Red (dark) blocks show regions that were accessed. This view of data access highlights the sparse nature of data accesses on each server during a single visualization operation. This type of access will result in many disk seeks, reducing I/O throughput, and significantly reduces the effectiveness of prefetching.

dataset takes 2.5 minutes, using a serial netCDF reader on a single process of the same BG/P machine (and storage system). While this may not seem expensive because it is done offline, a dataset consisting of multiple variables and hundreds of time steps can require hours or days of preprocessing time. Additional storage space is also required: typically the original netCDF files are kept in addition to the extracted files.

To avoid these overheads, we have implemented direct parallel reading of netCDF data files within our volume rendering code. Using the parallel netCDF library [2], we can perform collective I/O from multiple cores as we did previously from a file holding a single variable. Our initial tests indicate that this parallel read takes 10.8 seconds for the 1120^3 volume on 16K cores, which is approximately twice the time taken to read the preprocessed data file. We are currently investigating ways to improve this performance; but given how quickly analysis occurs for this dataset and the simplicity from the workflow perspective, this approach of reading directly from the simulation output seems appropriate in this case.

3. Organizing data for visualization

In the previous section we discussed the benefits of eliminating one type of preprocessing from the scientist's workflow. When we explored directly reading data from the astrophysics dataset in the above example, the performance degradation seen was due to the fact that we were accessing only 20% of the data in the time step and that data was spread out in the file, causing noncontiguous I/O access, where accesses for a single process are spread out across the file [14]. In datasets with more variables, in unstructured datasets where data might be spread less regularly throughout the file, or in cases where some data might not be needed for algorithmic reasons, the performance gap could be much wider between simulation data and preprocessed data access rates. While storage researchers are investigating methods to improve performance for analysis workloads with noncontiguous access patterns [15], we are examining methods to reorganize data within preprocessed data files to improve locality of access and, in doing so, improve I/O access rates.

Our algorithm is based on knowledge of data distribution in parallel file systems such as the Parallel Virtual File System (PVFS) [16], Lustre [17], and GPFS [18]. In these systems, data are striped across I/O servers in a round-robin manner. Figure 4 shows the I/O access patterns from a sample I/O server for a typical rendering run on a dataset of 512x512x1728 voxels stored on a PVFS file system with 16 I/O servers and 1 Kbyte strip size. This figure shows that the data I/O access patterns can be very sparse. When the data blocks needed for a visualization query are scattered across a large file, two problems can occur. First, these blocks may be striped onto only one or a few I/O servers, which can result in I/O congestion. Second, the data blocks on individual I/O servers may be stored noncontiguously, which will increase the number of disk seeks on that server during access, reducing effective throughput.

Generally, when volume rendering is applied to a large dataset, for a given user-defined transfer function and a particular view direction, there can be up to 70% of data blocks that are outside the value range of interest, and hence meaningless to load. Among those meaningful blocks, half of them can be occluded by other data blocks and thus become invisible. These blocks are also unnecessary to load. Our approach is to deploy a more efficient data distribution strategy to group together data blocks that are more likely to be accessed together to avoid noncontiguous I/O requests when visualizing large scale datasets. To group data blocks, we analyze the visibility pattern of the data blocks using *visibility feature vectors* for data blocks as signatures to identify data blocks that have similar access patterns. By reorganizing these data blocks based on the calculated access patterns, the number of necessary noncontiguous I/O can be greatly reduced.

3.1. Visibility feature vectors

When volume rendering is implemented by using the standard ray casting algorithm, a block's visibility can be estimated if we are given a list of sample values collected by the viewing rays starting from the image plane and before they reach the volume block. This is possible because we can evaluate the discrete volume rendering equation using the sample values and estimate the accumulated opacity. To represent the sample values for each ray, we can use a *per-ray sample histogram*. For a volume block under a particular view direction, we can combine the per-ray sample histograms from all the pixels within the projection footprint of the block. This combined sample histogram can characterize the block's visibility in the sense that if two blocks share a very similar sample histogram, they are likely to have similar visibility. This is the basic idea behind our visibility feature vector.

A visibility feature vector is a n-dimensional tuple in the format of $[h_1, h_2, ..., h_n]$, where each dimension represents a sample view direction. The entry h_i is the corresponding histogram with respect to the ith view direction. The purpose of the visibility feature vector is to keep an overall description of a block's visibility status for all possible view directions. Each block has a visibility feature vector, based on which it is possible to compare the visibility characteristics among multiple blocks. The blockwise visibility feature vector is constructed based on the data only and does not depend on run time transfer functions. By clustering data blocks based on the visibility feature vectors, it is possible to identify those blocks with approximately the same visibility status, and hence the same usage pattern. In addition to volume rendering, visibility feature vectors can be applied to other scalar data visualization techniques such as isosurfaces by configuring an appropriate transfer function. Isosurfacing can be implemented by setting the desired isovalue completely opaque while leaving other isovalues transparent. Invisible isosurfaces can be quickly estimated as in volume rendering using the per-ray sample histograms and the run-time transfer function.

To improve the I/O performance and minimize the number of noncontiguous I/O requests, we group the data blocks that share similar visibility characteristics and place them together in a file. When copying the file onto the parallel file system, data blocks are striped along various I/O servers in terms of pre-calculated clusters: the blocks within the same cluster are distributed in a round-robin manner into a

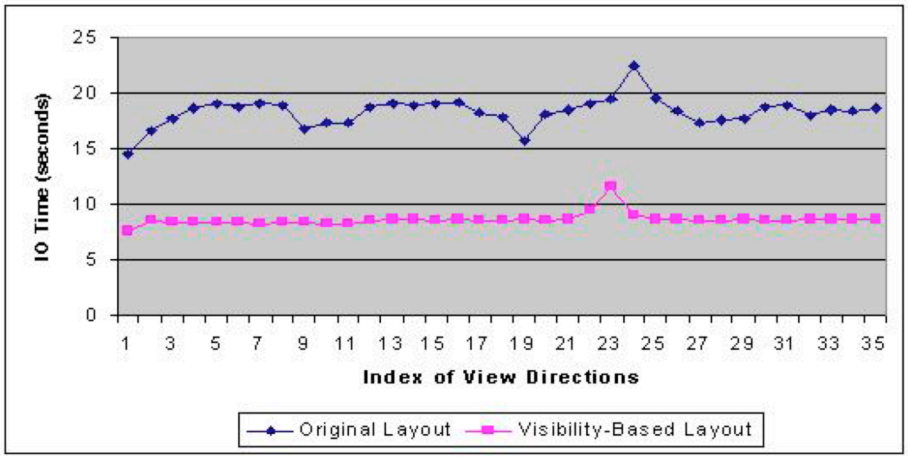

Figure 5. Comparison of I/O time between original file layout and visibility-based file layout using the Visible Woman dataset and a 1 Kbyte PVFS strip size.

sequence of I/O servers. This new layout strategy has the advantage that since the data blocks in the same cluster are likely to be accessed together, distributing the I/O load to all servers can avoid placing the load among only a few I/O servers. It can also avoid issuing multiple noncontiguous small IO requests in the individual I/O servers. Our experimental tests showed that the new file layout greatly reduces the I/O cost.

3.2. Experimental results

The test dataset for our I/O performance experiment is a 512x512x1728 dataset called the "Visible Woman." All our tests were run on an IBM Blue Gene/L supercomputer at Argonne National Laboratory, which has up to 2048 700 MHz PowerPC 440 cores, each with a double-pipeline-double-precision floating-point unit (FPU), a cache subsystem with built-in DRAM controller and the logic to support multiple communication sub-systems. The dual FPUs give each Blue Gene/L node a theoretical peak performance of 5.6 GFLOPS. In our experiments, the Visible Woman volume dataset is partitioned into 110,592 16^3 volume blocks and stored on a PVFS file system across all 16 I/O servers. All of our experiments were conducted using 64 processors.

We computed the visibility feature vectors described above. In the case of this data set, the alignment of the data set is such that view direction variations along the z-axis produce only small variations in resulting images. For this reason the sampled view directions were rotated along the x-axis and y-axis with a 20-degree increment in each step, with a fixed z-axis position, which produced a total of 64 sampled view directions. It took approximately 2.3 hours using 64 nodes of the Blue Gene/L to sample the view directions and build the corresponding visibility feature vectors. Based on the visibility feature vectors, a clustering process was conducted to group those blocks with similar feature vectors. Data blocks in the same cluster are placed contiguously in a new file; generating this new file took an addition 13 minutes. The PVFS file system will stripe the data in a round-robin manner to several I/O servers, placing these blocks near one another on individual servers. Assuming that we have correctly clustered data, I/O requests in this format should be serviced significantly more quickly than they would have been serviced in the original configuration.

We conducted a series of experiments to test the I/O performance of our algorithm. Two file layouts were tested on the Visible Woman dataset: one was the original layout where the data blocks were stored sequentially according to their spatial locations, and the other was the new layout using our visibility-based clustering. In order to ensure that the I/O performance improvement was stable under different

transfer functions, four transfer functions were used for each layout, and 35 view directions were sampled to test with each transfer function. During the run-time rendering process, after visibility culling, each processor reads the needed data blocks using MPI-IO. In our implementation we first use MPI_File_set_view() to adjust individual process's view of the data in the file. Then we call MPI_File_read_all() to utilize the noncontiguous I/O optimizations provided in MPI-IO. For each view direction, the I/O time was collected for each processor and the average time was calculated as the metric to compare between the two layouts. We tested using 16 I/O servers with three strip sizes – 1 Kbyte, 4 Kbyte, and 8 Kbyte – to explore the relationship between strip size and our layout algorithm.

Figure 5 shows a comparison of I/O performance between the two file layouts with the 1 Kbyte PVFS strip size for a sample transfer function, clearly demonstrating the improvements achieved by the visibility-based file layout. For each view direction the same number of data blocks were read in both file layouts. PVFS performed more efficiently when using the visibility-based layout. Results show up to 50% reduction of I/O time by using the visibility-based file layout under different view directions. Similar results were found for the other three transfer functions and with the 4 Kbyte and 8 Kbyte strip sizes, although access times improved slightly for both layouts as the strip size became more closely matched to the data block size. Moreover, the variances of I/O time under different view directions with the visibility-based layout were far less than those with the original layout. This is because in the visibility-based layout, there existed fewer numbers of noncontiguous I/O requests compared with the original file layout.

4. Conclusions

Optimizing access to storage is becoming increasingly important, not just during simulation run-time but also during analysis. Numerous approaches have been pursued to date, each with its own advantages and disadvantages. Directly reading simulation output makes sense when the simulation output is relatively well suited to analysis, as was the case in our astrophysics volume rendering example. In other cases aggressive preprocessing prior to analysis can provide substantial performance improvement, as seen in the Visible Woman example. Another option is to avoid this postsimulation analysis altogether, by performing analysis at simulation run-time.

In situ processing is the reduction, transformation, analysis, or viewing of data as it is being computed, using the same architecture as the computation [19]. This technique has two distinct advantages. First, this technique allows analysis routines to operate on data while it is still in memory, potentially significantly reducing the I/O demands of the overall workflow. Second, the technique naturally enables the application scientist to see the results of the simulation as it proceeds, because these results are generated as the simulation progresses. The Institute has explored in situ processing previously [19], including a study of in situ visualization for steering of a massively parallel earthquake simulation [20].

Solutions exist to enable in situ analysis in applications, but current portable solutions require that applications be developed in specific problem solving environments [21–25]. This requirement prevents in situ analysis from being adopted by a wider community of users, despite the technique's substantial benefits in time to solution and ability to track progress. A more accessible approach to in situ processing is desperately needed to address the performance limitations of storage resources in supercomputing systems. Because applications are increasingly using more descriptive I/O interfaces to interact with storage, from database interfaces such as SQL to multidimensional data interfaces such as the Hierarchical Data Format (HDF5) and Network Common Data Format (netCDF) libraries, one option might be to incorporate analysis techniques into the I/O path, hiding analysis routines under interfaces already in place in the applications and transparently enabling in situ processing.

Acknowledgments

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 and CCF-0325934, and DOE with agreement No. DE-FC02-06ER25777.

References

[1] *The Hierarchical Data Format*, http:// hdf.ncsa.uiuc.edu/HDF5/
[2] Li J, Liao W-K, Choudhary A, Ross R, Thakur R, Gropp W, Latham R and Siegel A, Gallagher B, and Zingale M 2003 Parallel netCDF: A high-performance scientific I/O interface *Proc. Supercomputing 2003*
[3] The MPI Forum 1997 MPI-2: extensions to the message-passing interface
[4] Blondin J M, Mezzacappa A, and Demarino C 2003 Stability of standing accretion shocks, with an eye toward core collapse supernovae *The Astrophysics J.* **584**(2):971
[5] Peterka T, Yu H, Ross R and Ma K-L 2008 Parallel volume rendering on the IBM Blue Gene/P *Proc. Eurographics Parallel Graphics and Visualization Symposium 2008*
[6] Molnar S, Cox M, Ellsworth D and Fuchs H 1994 A sorting classification of parallel rendering *IEEE Computer Graphics and Applications* **14**(4):23–32
[7] Marchesin S, Mongenet C and Dischler J-M 2006 Dynamic load balancing for parallel volume rendering *Proc. Eurographics Symp. Parallel Graphics and Visualization 2006*
[8] Levoy M 1988 Display of surfaces from volume data *IEEE Computer Graphics and Applications* **8**(3):29–37
[9] Max N L 1995 Optical models for direct volume rendering *IEEE Trans. Visualization and Computer Graphics* **1**(2):99–108
[10] Porter T and Duff T 1984 Compositing digital images *Proc. 11th Annual Conf. on Computer Graphics and Interactive Techniques* 253–259
[11] Hsu W M 1993 Segmented ray casting for data parallel volume rendering *Proc. 1993 Parallel Rendering Symposium* 7–14
[12] Ma K-L, Painter J S, Hansen C D and Krogh M F 1994 Parallel volume rendering using binary-swap compositing *IEEE Computer Graphics and Applications* **14**(4):59–68
[13] Iskra K, Romein J W, Yoshii K and Beckman P 2008 ZOID: I/O-forwarding infrastructure for petascale architectures *Proc. 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming* 153–162
[14] Ching A, Choudhary A, Liao A, Ross R and Gropp W 2004 Evaluating structured I/O methods for parallel file systems *International J. High Performance Computing and Networking* **2**:133–145
[15] Gu P, Wang J and Ross R 2008 Bridging the gap between parallel file systems and local file systems: A case study with PVFS *Proc. International Conf. on Parallel Processing 2008* (to appear)
[16] Carns P, Ligon W B, Ross R and Thakur R 2000 PVFS: A parallel file system for Linux clusters. *Proc. 4th Annual Linux Showcase & Conference*
[17] Cluster File Systems 2002 Lustre: A scalable, high performance file system *White paper*
[18] Schmuck F and Haskin R 2002 GPFS: A shared-disk file system for large computing clusters *Proc. File and Storage Technologies Conference* 231–244
[19] Ma K-L, Wang C, Yu H and Tikhonova A 2007 In-situ processing and visualization for ultrascale simulations *J. Physics* 78
[20] Tu T, Yu H, Ramirez-Guzman L, Bielak J, Ghattas O, Ma K-L and O'Hallaron D R 2006 From mesh generation to scientific visualization: An end-to-end approach to parallel supercomputing *Proc. ACM/IEEE Supercomputing 2006 Conference*
[21] Johnson C, Parker S, Hansen C D, Kindlmann G L and Livnat Y 1999 Interactive simulation and visualization *IEEE Computer* **32**(12):59–65
[22] Parker S and Johnson C 1995 SCIRun: A scientific programming environment for computational steering *Proc. Supercomputing 1995*
[23] Tuchman A, Jablonowski D and Cybenko G 1991 Run-time visualization of program data *Proc. IEEE Visualization 1991* 255–261
[24] Jablonowski D, Bruner J D, Bliss B and Haber R B 1993 VASE: The Visualization and Application Steering Environment *Proc. Supercomputing 1993*
[25] Geist G A, Kohl J A and Papadopoulos P M 1997 CUMULVS: Providing fault-tolerance, visualization, and steering of parallel applications *International J. High Performance Computing Applications* **11**:224–236