End-to-End Study of Parallel Volume Rendering on the IBM Blue Gene/P

Tom Peterka*, Hongfeng Yu†, Robert Ross*, Kwan-Liu Ma‡, Rob Latham*
*Argonne National Laboratory
Email: tpeterka@mcs.anl.gov
†Sandia National Laboratories, California
‡University of California, Davis

***Abstract*—In addition to their role as simulation engines, modern supercomputers can be harnessed for scientific visualization. Their extensive concurrency, parallel storage systems, and high-performance interconnects can mitigate the expanding size and complexity of scientific datasets and prepare for in situ visualization of these data. In ongoing research into testing parallel volume rendering on the IBM Blue Gene/P (BG/P), we measure performance of disk I/O, rendering, and compositing on large datasets, and evaluate bottlenecks with respect to system-specific I/O and communication patterns. To extend the scalability of the direct-send image compositing stage of the volume rendering algorithm, we limit the number of compositing cores when many small messages are exchanged. To improve the data-loading stage of the volume renderer, we study the I/O signatures of the algorithm in detail. The results of this research affirm that a distributed-memory computing architecture such as BG/P is a scalable platform for large visualization problems.**

Keywords–Distributed scientific visualization; Parallel volume rendering; image compositing, parallel I/O

I. Introduction

As data sizes and supercomputer architectures approach petascale, visualizing results directly on parallel supercomputers becomes a compelling approach for extracting knowledge from data. Benefits of this approach include the elimination of data movement between computation and visualization architectures; the economies of large-scale, tightly coupled parallelism; and the possibility of visualizing a simulation while it is running [1]. This paper is a continuation of ongoing work, where we examine the use of large numbers of tightly connected processor nodes in the context of a parallel ray casting volume rendering algorithm implemented on the IBM Blue Gene/P (BG/P) system at Argonne National Laboratory [2].

In this paper, we further evaluate the scalability of this method by testing on the largest structured grid volume data and system scales published thus far without resorting to out-of-core methods. Our goal is to expose bottlenecks in the three stages of the algorithm–I/O, rendering, and compositing–at very large scale. Performance results are analyzed in terms of scalability and time distribution between the three algorithm stages. The BG/P architecture and its parallel file system are examined in depth to help understand the results.

Our largest tests include 32K cores, 4480^3 data elements, and 4096^2 image pixels. At this scale, performance bottlenecks are the result of both algorithms and systems issues; algorithms must be tuned to take advantage of particular system characteristics such as I/O and interconnect parameters. Thus, we must be prepared to adapt rendering and compositing algorithms to high-performance computing (HPC) system characteristics such as communication topology and storage infrastructure, just as computational scientists do when using these resources.

Hence, this study is a combination of both visualization algorithms and HPC systems research; our contributions are reduced compositing time and improved I/O performance when reading time steps from storage, allowing us to scale parallel volume rendering to the largest in-core problem and system sizes published to date.

II. Background

Beginning with a description of our dataset, we then discuss prior parallel volume rendering literature. We review, in particular, sort-last compositing algorithms.

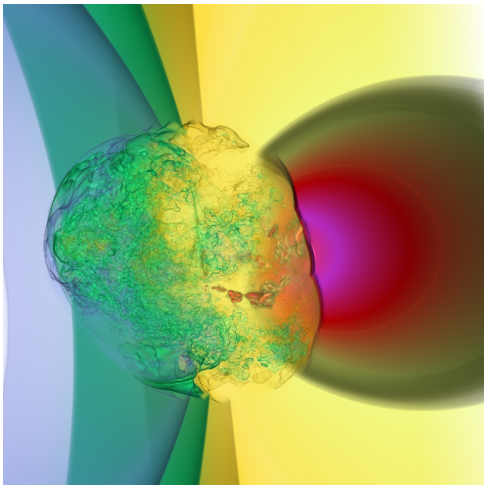

Fig. 1. Visualization of the X component of velocity in a core-collapse supernova.

TABLE I
PUBLISHED PARALLEL VOLUME RENDERING SYSTEM SCALES

Dataset	System Size(CPUs)	Billion Elements	Image Size	Year	Reference
Fire	64	14	800^2	2007	[3]
Blast Wave	128	27	1024^2	2006	[4]
Taylor-Raleigh	128	1	1024^2	2001	[5]
Molecular Dynamics	256	.14	1024^2	2006	[4]
Earthquake	2048	1.2	1024^2	2007	[1]
Supernova	4096	.65	1600^2	2008	[2]

A. Dataset

The dataset in Figure 1 shows X velocity from time step number 1530 of a supernova simulation. The data are courtesy of John Blondin at the North Carolina State University and Anthony Mezzacappa of Oak Ridge National Laboratory [6], through the U.S. Department of Energy's SciDAC Institute for UltraScale Visualization [7]. The model seeks to discover the mechanism of core collapse supernova, the violent death of short-lived, massive stars. The supernova dataset was chosen to be representative of the scale of computational problems encountered today.

Blondin et al.'s hydrodynamics code VH-1 stores five time-varying scalar variables, in 32-bit floating-point format in a single file for each time step. The file type is Network Common Data Format (netCDF), and in current large-scale runs, a single time step is 27 gigabytes (GB). The data are represented in a structured grid of 1120^3, or approximately 1.4 billion elements. To visualize a single variable, we can extract it during an offline preprocessing step and save it in a single, 32-bit raw data file of 5.3 GB. Or, we can read the entire 27 GB netCDF file in parallel, directly from the visualization code.

B. Parallel Volume Rendering

Parallel volume rendering is not new. Beginning with Levoy's classic ray casting in 1988 [8], parallel versions began to appear soon after, for example, [9]. Table I summarizes more recent examples of large dataset parallel volume rendering, on the order of one billion data elements.

Parallel algorithms can be classified according to when partial results are sorted: sort-first, sort-middle, or sort-last [10]. In sort-first, the image space is divided among processes, and the data space is replicated. The opposite occurs in sort-last: the data are divided among processes, and the entire image space is replicated in each process. Sort-middle is a hybrid combination of the two, but it is difficult to implement and uncommon in practice. In the case of large scientific visualizations, the sort-last approach is appropriate because data size is the dominant concern; data sizes are usually three orders of magnitude larger than image sizes.

C. Image Compositing

Image compositing is the reduction of multiple partial images created during the parallel rendering stage, into one final image. Like rendering, image compositing techniques have been widely published. We use the direct-send compositing approach [11], where each process takes ownership for a subregion of the final image, and receives partial results from those processes that contribute to this subregion. Cavin et al. [12] analyze relative theoretical performance of a number of compositing methods.

Because image compositing can be modeled as a data reduction problem, the image compositing methods used by the visualization community are similar to the collective communication algorithms in the message passing community. For example, tree methods such as Ma et al.'s binary swap algorithm [13] have counterparts in system communication space, such as the butterfly algorithm of Traff [14]. Analyses of the costs of collective algorithm can be found in [15]. Recent collective optimizations can be found in [16], [17].

There are also numerous studies specific to the Blue Gene architecture that evaluate the relationship between message sizes, numbers of messages, and link contention. Kumar and Heidelberger [18] show that in all-to-all tests when the message size drops below 256 bytes, bandwidth falls off rapidly away from the theoretical peak line. Davis et al. [19] report hot spots, where multiple senders send messages to the same recipient, and they report bandwidth at these locations to be three times slower than elsewhere. Hoise et al. [20] show that the fraction of peak bandwidth drops from near 100% to around 10% as the contention level increases. Almasi et al. [21] perform a benchmark of MPI_Allreduce(), and conclude that aggregate bandwidth drops by a factor of three when the message size decreases.

D. Parallel I/O

Traditionally, the subject of I/O has been ignored in the scientific visualization literature. This is beginning to change. For example, Yu and Ma [22] overview I/O techniques for visualization in [22]. As simulation and visualization grow in scale, data size and data movement become limiting factors in overall performance. Parallelism in accessing storage can improve this bottleneck; for example, multiple processes can read from the same file via collective I/O constructs.

Thus, visualization researchers need to educate themselves in systems storage concepts such as parallel I/O. This means understanding the parallel I/O software stack, beginning with the underlying storage architecture, parallel file systems such as PVFS [23]. I/O-related issues are further encountered in middleware and APIs such as ROMIO and MPI-IO [24], up to high-level storage libraries and file formats such as parallel netCDF [25] and HDF5 [26]. The I/O patterns in our algorithm will be examined in considerable detail in the remainder of this paper.

III. IMPLEMENTATION

To test scalability and performance at high numbers of processes, we implemented the volume rendering algorithm on a 160,000-core Blue Gene/P machine. A brief description

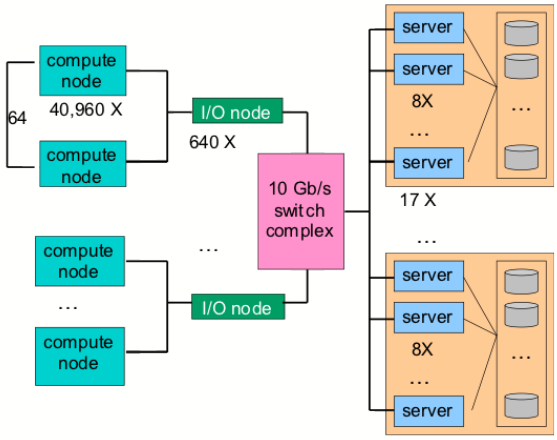

Fig. 2. The storage system and its connection to BG/P. The parallel file system includes 17 SANs; each contains four or eight servers for failover capability. Total storage capacity is 4.3 PB. The file system communicates with BG/P via I/O nodes. One I/O node handles 64 compute nodes.

of that architecture is followed by implementation details of this application on BG/P.

A. Blue Gene Architecture

The Blue Gene/P system at the Argonne Leadership Computing Facility is a 557-teraflop machine with ample opportunities to experiment with parallel rendering. Four PowerPC-450 850-MHz cores share 2 GB RAM to form one compute node. 1K nodes are contained within one rack, and the entire system consists of 40 racks. The total memory footprint is 80 TB.

Application processes execute on top of a microkernel that provides basic OS services. The Blue Gene architecture has two separate interconnection networks: a 3D torus for interprocess point-to-point communication and a tree network for collective operations. The 3D torus maximum latency between any two nodes is 5 μs, and its bandwidth is 3.4 gigabits per second (Gb/s) on all links. BG/Ps tree network has a maximum latency of 5 μs, and its bandwidth is 6.8 Gb/s per link.

The tree also communicates with I/O nodes that serve as bridges to the storage system. BG/P has one I/O node for every 64 compute nodes. The total storage capacity of BG/P's parallel file system is 4.3 petabytes. The storage system consists of a physical array of storage nodes and servers, connected to BG/P by high-speed networks. The hardware structure, shown in Figure 2, consists of 17 racks of storage area network (SAN) storage. Each rack has eight servers attached, with a separate logical unit per server. File servers may take over on behalf of one another in the event of a server or network failure.

In theory, each SAN can deliver 5.5 gigabytes per second (GB/s) of peak storage bandwidth, and tests show that current aggregate peak bandwidth is approximately 50 GB/s. Our application exhibits considerably lower bandwidth, in part because it uses only 23% of the total system, and in part because it accesses a noncontiguous 3D data volume. We will analyze the I/O performance of individual file formats in greater detail later in this paper.

B. Algorithm Stages

Our parallel volume rendering algorithm is a sort-last implementation that divides the data space into regular blocks and statically allocates a small number of blocks to each process. The algorithm consists of three steps that occur sequentially: I/O, rendering, and compositing; this sequence is performed among many cores in parallel.

Both the first and last steps of this sequence occur collectively. That is, no single process reads the entire data file and redistributes it. Rather, each process logically reads only its portion of the data from the file. Likewise, no one process is responsible for compositing the results. Rather, compositing occurs via a many-to-many communication pattern.

To instrument the algorithm, we define the time that a frame takes to complete as the time from the start of reading the time step from disk to the time that the final image is completed. We further divide the frame time into three components: I/O time, rendering time, and compositing time.

1) I/O: MPI processes read the data file collectively. In the case of raw file format, MPI-2 [27] (MPI-IO) performs data staging. This allows each process to read its own portion of the volume in parallel with all of the other processes. In the case of netCDF format, the Parallel-NetCDF library performs similar collective functions. This way, data are read quickly in parallel, and there is no need for the application to communicate data between processes.

Moreover, collective I/O permits large datasets to be processed in-core because the entire dataset never resides within the memory of a single process. For example, in raw mode our file size is over 5 GB per variable, per time step; in netCDF mode the size is 27 GB. BG/P contains only 2 GB of memory per node. With collective I/O, the total memory footprint of the entire machine (80 TB) dictates the maximum data that can be processed in-core, without resorting to processing the data in serial chunks.

Underlying the programming interface is a parallel file system such as PVFS. By striping data across multiple disks controlled by a number of file servers, application programs can access different regions of a file in parallel. Performance varies with the number of I/O nodes being used and on the processes' pattern and size of accesses. Performance can also be tuned by means of hints passed to MPI-IO, adjusting such parameters as internal buffer sizes and number of I/O aggregators [28].

2) Rendering: The computation of local subimages requires no interprocess communication. To render its subimage, a process casts a ray from each pixel through its data subdomain. As data values are sampled along the ray in front-to-back order, they are converted into color and opacity values according to a transfer function and accumulated. The resulting pixel value is the blending of sampled colors and opacities along the ray. This is the pixel value for a single process. This rendering step

is strictly local; all of the processes still need to blend their local values for that pixel in order to determine its final value. This blending occurs in the next stage of the algorithm, image compositing.

3) Image Compositing: Compositing of parallel volume rendered subimages is implemented with the direct-send method as follows. At the start of compositing, each of n processes (*renderers*) owns a completed subimage of its portion of the dataset. Of the n rendering processes, m processes (*compositors*) are also assigned responsibility for $1/m$ of the final image area. Normally, direct-send implementations use $n = m$. Each of the renderers send their subimage to all of the compositors that require it. The average number of messages in the entire compositing process is $O(mn^{1/3})$ because on average, $n^{1/3}$ messages are sent to each of m recipients.

IV. End-to-End Performance Analysis

To evaluate the performance of our supernova dataset on BG/P, we measured the time to render a frame across a range from 64 to 32K cores. The first set of results are for a single variable in raw data format, a single time step, 1120^3 grid size, and 1600^2 pixels image size. We then evaluate larger data and image sizes.

A. Total and Component Time

Figure 3 shows a log-log plot of time to read, render, and composite one time step using raw I/O mode. The best all-inclusive frame time of 5.9 s was achieved with 16K cores. For comparison with other published data that may exclude I/O time, our visualization-only time (rendering + compositing) is 0.6 s. The rendering curve is approximately linear. Rendering is embarrassingly parallel–no interprocess communication is required. Minor deviations in the curve are due to load imbalances between rendering processes.

Figure 3 shows two compositing curves: original and improved. The original compositing time remains constant through 1K cores. Beyond that, compositing time increases sharply due to the large number of messages, and beyond 8K cores, the compositing time is greater than the rendering time. To address the performance degradation beyond 1K cores, we limit the number of compositors. The total time curve reflects the improved compositing method.

Customarily in direct-send, the number of compositors is the same as the number of renderers, but this need not be the case. So, we used 1K compositors when the number of renderers is between 1K and 4K and then 2K compositors beyond that. We arrived at these values empirically after testing combinations of renderers and compositors. From the original compositing times, we knew that contention was not an issue below 1K compositors. After a small number of trials, we determined that 2K compositors were sufficient for up to 32K renderers, and that 4K renderers was an appropriate point to increase the number of compositors. Finer control over the number of compositors did not improve the results.

The number of compositors is known at initialization time, and the schedule of messages is built around this number from the beginning. During compositing, n renderers transmit their messages to m compositors, where $n \geq m$. The reduction from n to m occurs automatically as part of the compositing step and incurs no additional cost. The number of messages still has the same theoretical complexity, but the smaller constant is significant.

At 32K renderers, the compositing time improved by a factor of 30 times over the original scheme. By limiting the number of compositors, the overall frame time decreases by 24%. We presume that link contention causes the original compositing performance to degrade at large system scale, because communication bandwidth degrades with large numbers

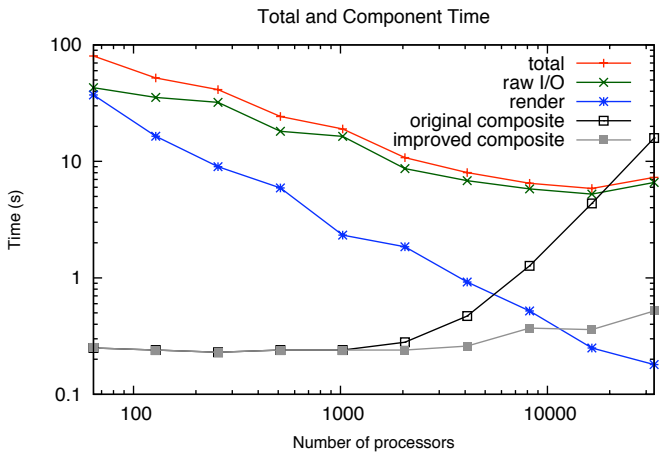

Fig. 3. Total frame time as well as individual components I/O, rendering, and compositing times plotted on a log-log scale. Two versions of compositing time are shown; the total frame time includes the faster, improved compositing. The file is raw data format, 1120^3, and the image size is 1600^2.

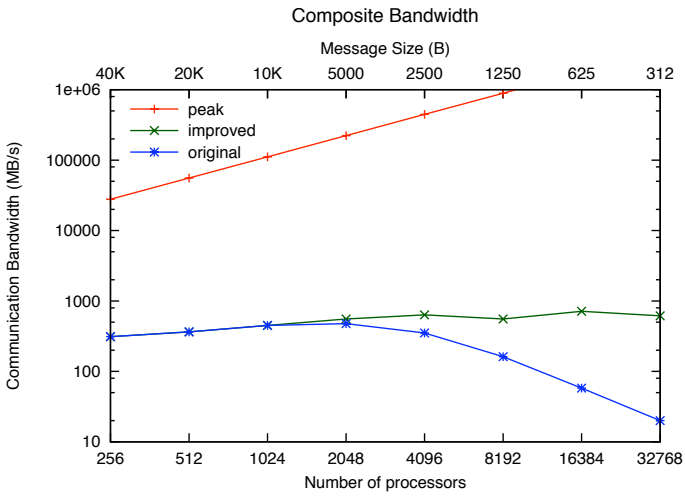

Fig. 4. Communication bandwidth plotted against message size and number of processors. As the number of processors increases and message size decreases, the bandwidth falls away from the peak theoretical curve. The drop-off is more severe in the original compositing scheme and alleviated by limiting the number of compositors.

of small messages. Figure 4 shows this trend, plotting communication bandwidth against message size for the compositing portion of the previous test results. For comparison, BG/P's peak bandwidth is also shown.

B. Large Data and Image Size

The preceding subsection documents the performance of the volume rendering algorithm on a data size of 1120^3 and an image size of 1600^2. This represents the current scale of hydrodynamics simulations such as VH-1. In an effort to look ahead to future growth of simulations, we tested the volume rendering algorithm on two larger data sizes. Because data in the desired scale do not exist, at least from any of our collaborators, we upsampled the existing supernova raw data format. Upsampling preserves the structure of the data, and resulting images are similar to those from the original data. The upsampling was performed efficiently, in parallel, with the same BG/P architecture and collective I/O, but as a separate step prior to executing the visualization.

We produced a time step of the supernova data in two new grid sizes: 2240^3 with 11 billion elements and 4480^3 with 90 billion elements. Each time step occupies 41 GB and 335 GB of storage space on disk, respectively. In order to faithfully reproduce the resolution of the dataset, the size of the image should scale with data size. Thus, for these tests, we generated 2048^2 images for the 2240^3 volume and 4096^2 images for the 4480^3 volume. To our knowledge, these are some of the largest in-core volume rendering results published to date.

Table II shows the detailed timing results of volume rendering these datasets using 8K, 16K, and 32K cores. One time step of the 2240^3 dataset can be volume rendered end-to-end in 35 seconds. Of this time, 96% is I/O, with an aggregate I/O bandwidth of 1.3 GB/s. Compositing and rendering both complete in less than 1 second. For the 4480^3 dataset, a frame requires 211 seconds or about 3-1/2 minutes to visualize end-to-end. I/O again requires 96% of the total time, at 1.6 GB/s aggregate storage bandwidth.

TABLE II
VOLUME RENDERING PERFORMANCE AT LARGE SIZES

Grid Size	Time step (GB)	Image (pixels)	# Procs	Tot. Time (s)	% I/O	% Composite	Read B/W (GB/s)
2240^3	42	2048^2	8K	51.35	96.1	1.0	.87
			16K	43.11	97.4	1.0	1.02
			32K	35.54	95.8	2.7	1.26
4480^3	335	4096^2	8K	316.41	96.1	0.5	1.13
			16K	272.63	96.8	1.5	1.30
			32K	220.79	95.6	2.6	1.63

Figure 5 summarizes the overall frame time, including I/O, rendering, and compositing, for all three data and image sizes, over all of the system sizes that we tested. The graph shows that even at 2K or 4K cores, any of the problem sizes can be visualized, given enough time. The configuration that produces the shortest run time might not always be viable; for example, one might choose to wait a few more minutes longer for a result to execute if it means that a smaller subset of the machine is available earlier.

V. I/O ANALYSIS

Figure 6 is a stacked graph of the time spent in the three stages of the volume rendering algorithm as the system scale increases. It shows that rendering time is not the bottleneck when parallelizing large-scale volume rendering on the BG/P. Rather, the most critical stage is I/O, which is why much of our visualization research is directed at understanding and increasing I/O performance.

Thus far we have concentrated on the results of reading a single variable in raw 32-bit format. Blondin et al. store their dataset in netCDF record variable format. Reading the netCDF file directly in the visualization has advantages, such as eliminating preprocessing and having multiple variables simultaneously available for rendering. Figure 7 compares the I/O rates for raw and netCDF modes. NetCDF is approximately 4-5 times slower than raw mode at low numbers of cores. At high core counts, netCDF mode is 1.5 times slower.

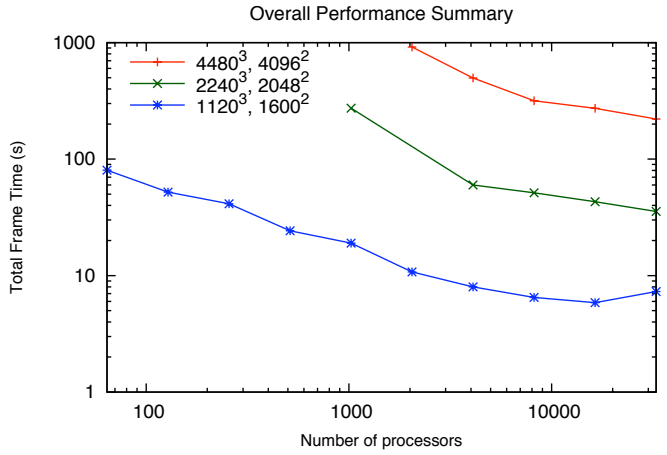

Fig. 5. Total frame time for three data and image sizes on a log-log scale.

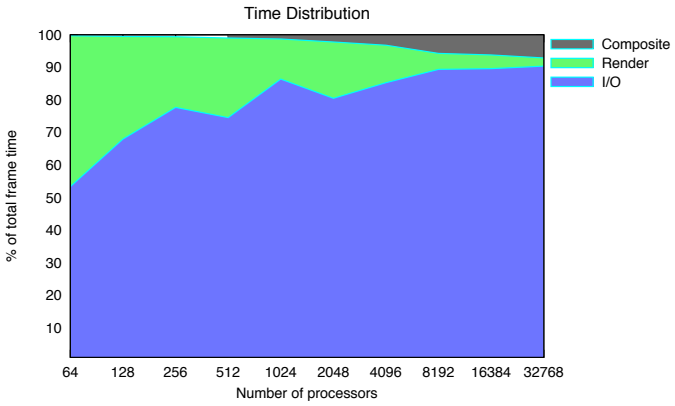

Fig. 6. Percentage of time spent in I/O, rendering, and compositing. I/O dominates the overall algorithm's performance.

To understand the causes of the slowdown in netCDF access rate compared to raw format, and before attempting to improve it, we need to understand how netCDF writes multiple variables within a file and the implications to accessing a single variable. Then, we compare the performance of netCDF to other file formats such as HDF5.

A. NetCDF File Format

The netCDF file format allows for storage of multiple multidimensional, typed variables in a single file, along with attributes assigned to that data. Variables stored in this format may be defined as one of two types: nonrecord or record.

Nonrecord variables are variables for which all dimensions are specified. These variables are stored as a contiguous block of data in the file, which favors high-performance access. Unfortunately, the current netCDF format limits the total size of a nonrecord variable to 4 GB. The current output size for our application exceeds this limit, forcing the scientists to use record variables.

In the netCDF record variable format, the logical 3D variables are stored as records of 2D data and interleaved record by record for each variable (Figure 8). Our dataset contains five variables: pressure, density, and velocity in X, Y, and Z. The result of formatting is that individual variables are split into noncontiguous regions (about 5 MB in size) spread out regularly through the file, with a file size approximate five times as large as a single variable in our raw format.

To understand I/O performance in the context of this file format, we need to recognize that the Parallel netCDF library is using MPI-IO internally and that the MPI-IO implementation (ROMIO) uses a two-phase optimization that reads in a large contiguous region of data and then distributes the small, noncontiguous regions of interest to the appropriate MPI process [24], [28]. In the ideal case, the "density" of relevant data in the contiguous region is high; but in our case, only one record out of every five contains the data we want, so a great deal of undesired data are read, reducing our effective bandwidth.

Besides showing original (untuned) netCDF performance, Figure 7 shows that tuning I/O parameters to a particular data layout can result in significant gains. Setting MPI-IO hints based on our knowledge of the file layout allows us to optimize the read pattern over the default behavior of the MPI-IO library. In this case, setting the read buffer size to the netCDF record size (1120^2 x 4 bytes) improved the netCDF I/O performance in some cases by a factor of two over the untuned performance, eliminating reads of data we would not be processing. (Recall from Figure 8 that the record size is that of a 2D slice of the volume.)

In the case of raw data, we also experimented with the size of the read buffer, but in the case of the netCDF data, understanding the file layout allowed us to exactly match the buffer size to the data record size. We are continuing to study the effects of this hint, as well as others such as the number of collective aggregators. Tuning parallel I/O is an active research area in our group, and we plan to continue to develop and share I/O performance models that can be used to optimize an applications' collective I/O requests.

To better grasp how much extra data are read when accessing data stored in the netCDF record variable format, and how tuning can improve this overhead, we graphically depicted the access patterns of our I/O options in Figure 9. The dark regions are the file blocks that are read, while light blocks are untouched. The left panel depicts the file accesses, without any tuning, required to read a single variable, pressure, from the netCDF file. Clearly most of the file is being accessed in order to read only one out of five variables contained within. In this case, the collective read of the pressure variable results

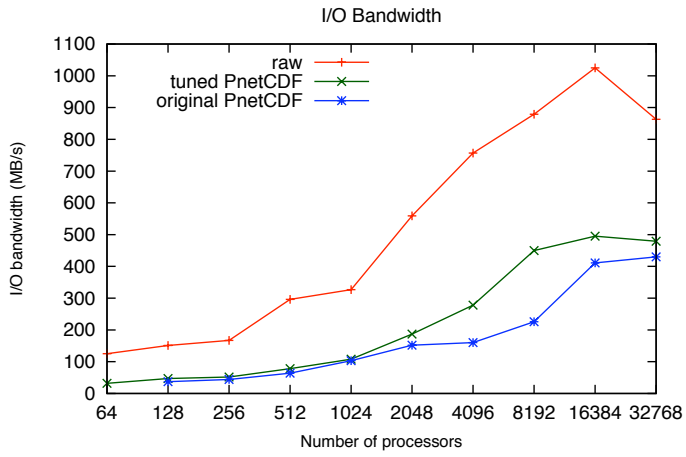

Fig. 7. Our application's I/O performance for raw mode, original PnetCDF, and tuned PnetCDF. Data size is 1120^3.

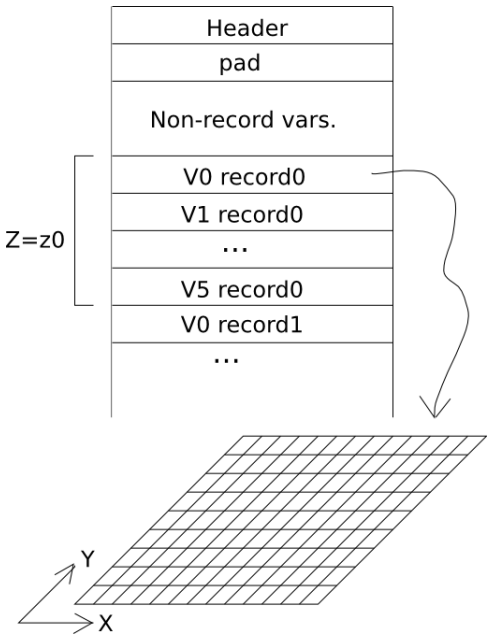

Fig. 8. The organization of variables within the netCDF file.

in approximately 3,000 actual accesses, each roughly 15 MB in size.

The center panel of Figure 9 shows the improved access pattern that can be achieved by setting MPI-IO hints described above. Now 2,600 accesses are performed, but each has an average size of 4.5 MB. In total, 11 GB are accessed in order to read 5 GB of useful data. This is still a significant overhead, but it is four times less than the untuned access pattern. The amounts of extra access needed in order to read usable data help to explain the overall I/O bandwidth in Figure 7 that tuning enables. A more sophisticated two-phase optimization could perform even better [29], and we are looking to improving the two-phase algorithm in ROMIO.

B. Alternative File Formats

A number of scientists use the HDF5 standard in their work, so we converted the netCDF file to HDF5 and retested. The right panel of Figure 9 shows the HDF5 performance. Log files show that the data blocks for a single variable are better collocated in HDF5 compared to netCDF record variable access; the HDF5 read requires 8 GB of physical I/O in order to read a 5 GB variable from the file. We did no tuning to attain this rate. There is some initial overhead when the dataset is opened, and every process performs 11 very small metadata accesses of no more than 600 bytes. Beyond that, the data appear to be written contiguously within the file, so that accesses are more efficient.

Some of the members of our team, in conjunction with K. Gao, W.-K. Liao, and A. Choudhary of Northwestern University, are working on a future netCDF format that addresses some of the limitations of the current version of netCDF. By expanding all of the fields to 64 bits, the new implementation permits nonrecord variables of virtually unlimited size. We also tested this version using nonrecord variables, and the result was the same as HDF5. The right panel of Figure 9 shows a single image for both HDF5 and the new netCDF with 64-bit addressing.

Figure 10 reinforces the conclusion that file layout influences the number of blocks needed to be read, and in turn I/O performance. These are results of a synthetic benchmark based on the I/O pattern of the volume rendering code. It shows the comparison between five data formats: raw, untuned netCDF, tuned netCDF, HDF5, and the future release of the new netCDF format. We read 1120^3 data elements using 2K cores. There is a strong correlation between the overall read time and the *data density*, which we define as the physical size in bytes of the desired data divided by the number of bytes that are actually read by the underlying collective I/O infrastructure.

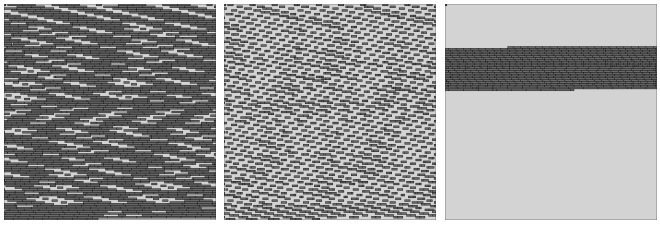

Fig. 9. Reading netCDF without tuning (left) results in very inefficient access, apparent in this visualization of the data access pattern generated from I/O logs of a PnetCDF read of the 1120^3 dataset by 2K cores. The dark regions signify file blocks that were read in order to access a single variable. Using MPI-IO hints (center) to tune the access pattern results in a more efficient access pattern. The best patterns result from HDF5 and a new release of netCDF that features 64-bit addressing (right).

VI. Discussion and Future Work

Our tests show that parallel software volume rendering can be performed on the IBM Blue Gene/P at scales of tens of billions of data elements, millions of pixels, and tens of thousands of processor cores. We presented performance data for several problem sizes, across a range of processor cores spanning nearly three orders of magnitude and examined how total and component times vary, as well as timing distributions between component phases of the algorithm.

I/O limits performance. We studied five I/O modes in an effort to understand and optimize the performance of multivariate data formats such as netCDF. Reading these formats directly in the visualization eliminates the need for costly preprocessing and affords the possibility to perform multivariate visualizations in the future.

In compositing, we improved link contention by realizing that the number of renderers need not be the same as the number of compositors. When the number of rendering cores grows to several thousand, compositing time can be reduced significantly by compositing with fewer cores. We also upsampled our dataset to produce larger time steps and increased the image size commensurate with volume size.

BG/P has the needed tools to perform large volume rendering. These include numerous processor cores each capable of

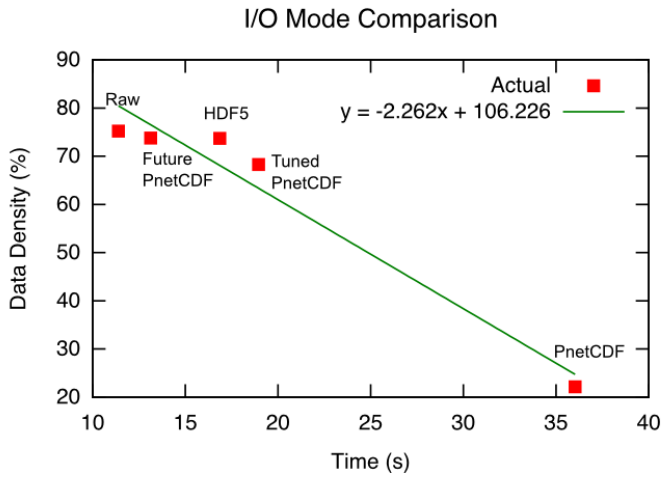

Fig. 10. From a synthetic benchmark, five I/O modes appear in order from fastest to slowest for a test read of 1120^3 data elements using 2K cores to perform the collective I/O. We define the data density as the number of blocks needed divided by the number of blocks actually read. There is a strong correlation between the time and the data density.

software rendering a small subset of the total data, interconnected by an efficient network, and served by a high-capacity parallel storage system. These characteristics are necessary enablers for future in situ visualization. We hope that in situ techniques will enable scientists to see early results of their computations, as well as eliminate or reduce expensive storage accesses, because, as our research shows, I/O dominates large-scale visualization.

We are continuing to study the I/O signature, that is, the striping pattern across I/O servers, of this and other algorithms. We are also comparing against other codes and benchmarks. The effect of the file system on performance is an active area of research; we are conducting similar experiments on Lustre.

In the future, we plan to implement and test other visualization algorithms at these scales. We plan to extend these algorithms to unstructured and adaptive mesh refinement grid data. We will research how to best overlap executing simulations with visualizations as we apply the lessons learned to in situ visualization. We plan to also conduct similar experiments on other supercomputer systems such as the Cray XT.

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] *SciDAC Institute for Ultra-Scale Visualization*, 2008, http://ultravis.ucdavis.edu/.

[8] M. Levoy, "Display of surfaces from volume data," *IEEE Computer Graphics and Applications*, vol. 8, no. 3, pp. 29–37, 1988.

[9] U. Neumann, "Parallel volume-rendering algorithm performance on mesh-connected multicomputers," in *Proc. 1993 Parallel Rendering Symposium*, San Jose, CA, 1993, pp. 97–104.

[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] W. M. Hsu, "Segmented ray casting for data parallel volume rendering," in *Proc. 1993 Parallel Rendering Symposium*, San Jose, CA, 1993, pp. 7–14.

[12] X. Cavin, C. Mion, and A. Fibois, "Cots cluster-based sort-last rendering: Performance evaluation and pipelined implementation," in *Proc. IEEE Visualization 2005*, 2005, pp. 111–118.

[13] 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.

[14] J. L. Traff, "An improved algorithm for (non-commutative) reduce-scatter with an application," in *Proc. EuroPVM/MPI 2005*, Sorrento, Italy, 2005, pp. 129–137.

[15] E. Chan, M. Heimlich, A. Purkayastha, and R. van de Geijn, "Collective communication: theory, practice, and experience: Research articles," *Concurr. Comput. : Pract. Exper.*, vol. 19, no. 13, pp. 1749–1783, 2007.

[16] E. Chan, R. van de Geijn, W. Gropp, and R. Thakur, "Collective communication on architectures that support simultaneous communication over multiple links," in *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2006, pp. 2–11.

[17] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in mpich," *International Journal of High Performance Computing Applications*, vol. 19, pp. 49–66, 2005.

[18] S. Kumar and P. Heidelberger, *Performance Analysis of All-to-All Communication on the Blue Gene/L Supercomputer*, 2007, http://domino.research.ibm.com/library/cyberdig.nsf/papers.

[19] K. Davis, A. Hoisie, G. Johnson, D. J. Kerbyson, M. Lang, S. Pakin, and F. Petrini, "A performance and scalability analysis of the blue gene/l architecture," in *Proc. Supercomputing 2004*, Pittsburgh, PA, 2004, p. 41.

[20] A. Hoisie, G. Johnson, D. J. Kerbyson, M. Lang, and S. Pakin, "A performance comparison through benchmarking and modeling of three leading supercomputers: Blue gene/l, red storm, and purple," in *Proc. Supercomputing 2006*, Tampa, FL, 2006, p. 3.

[21] G. Almasi, C. J. Archer, C. C. Erway, P. Heidelberger, X. Martorell, J. E. Moreira, B. Steinmacher-Burow, and Y. Zheng, "The optimization of mpi collective communication on blue gene/l systems," in *Proc. ICS'05*, Boston, MA, 2005, pp. 253–262.

[22] 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.

[23] P. Carns, W. B. I. Ligon, R. Ross, and R. Thakur, "Pvfs: A parallel file system for linux clusters," in *Proc. 4th Annual Linux Showcase & Conference*, Atlanta, GA, 2000, p. 28.

[24] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective i/o in romio," in *Proc. 7th Symposium on the Frontiers of Massively Parallel Computation*, 1999, pp. 182–189.

[25] J. Li, W.-k. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, "Parallel netcdf: A high-performance scientific i/o interface," in *Proc. Supercomputing 2003*, Phoenix, AZ, 2003.

[26] M. Folk, A. Cheng, and K. Yates, "Hdf5: A file format and i/o library for high performance computing applications," in *Proc. Supercomputing 1999*, Portland, OR, 1999.

[27] A. Geist, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, W. Saphir, and T. Skjellum, "Mpi-2: Extending the message-passing interface," in *Proc. Euro-Par'96*, Lyon, France, 1996.

[28] R. Thakur and A. Choudhary, "An extended two-phase method for accessing sections of out-of-core arrays," *Scientific Programming*, vol. 5, no. 4, pp. 301–317, 1996.

[29] K. Coloma, A. Ching, A. Choudhary, R. Ross, R. Thakur, and L. Ward, "New flexible mpi collective i/o implementation," in *Proc. Cluster 2006*, 2006.