A Parallel Visualization Pipeline for Terascale Earthquake Simulations

Hongfeng Yu   Kwan-Liu Ma
University of California at Davis
{yuho,ma}@cs.ucdavis.edu

Joel Welling
Pittsburgh Supercomputing Center
welling@psc.edu

ABSTRACT

This paper presents a parallel visualization pipeline implemented at the Pittsburgh Supercomputing Center (PSC) for studying the largest earthquake simulation ever performed. The simulation employs 100 million hexahedral cells to model 3D seismic wave propagation of the 1994 Northridge earthquake. The time-varying dataset produced by the simulation requires terabytes of storage space. Our solution for visualizing such terascale simulations is based on a parallel adaptive rendering algorithm coupled with a new parallel I/O strategy which effectively reduces interframe delay by dedicating some processors to I/O and preprocessing tasks. In addition, a 2D vector field visualization method and a 3D enhancement technique are incorporated into the parallel visualization framework to help scientists better understand the wave propagation both on and under the ground surface. Our test results on the HP/Compaq AlphaServer operated at the PSC show that we can completely remove the I/O bottlenecks commonly present in time-varying data visualization. The high-performance visualization solution we provide to the scientists allows them to explore their data in the temporal, spatial, and variable domains at high resolution. The new high-resolution explorability, likely not available to most computational science groups, will help lead to many new insights.

Keywords

High-performance computing, massively parallel supercomputing, MPI, scientific visualization, parallel I/O, parallel rendering, time-varying data, vector field visualization, volume rendering

1. INTRODUCTION

Large-scale computer modeling of the earthquake-induced ground motion in large heterogeneous basins and analysis of the soil-structure interaction can help understand earthquake and reduce its risk to the general population. The

simulation results can guide development of more rational seismic provisions for building codes, leading to safer, more efficient, and economical structures in earthquake-prone regions. However, a complete quantitative understanding of strong ground motion in large basins requires simultaneous consideration of 3D effects of earthquake source, propagation path, and local site conditions. The large scale associated with the modeling places enormous demands on computational resources.

A multidisciplinary team of researchers [24, 3] has been developing tools to model ground motion and structural response in large heterogeneous basins, and apply these tools to characterize the seismic response of large populated basins such as Los Angeles. To model at the needed scale and accuracy, they have created some of the largest unstructured finite element simulations ever performed by utilizing massively parallel supercomputers. Consequently, a serious challenge they face is visualizing the output of these very large, highly unstructured simulations.

An effective way to understand earthquake wave propagation is to volume render the time history of the 3D displacement and velocity fields. However, interactive rendering of time-dependent unstructured hexahedral datasets with $10^7 - 10^8$ elements (anticipated to grow to 10^9 over the next several years) presents a major challenge. In particular, what makes time-varying volume data visualization hard is the need to constantly transfer each time step of the data from disk to memory to carry out the rendering calculations. This I/O requirement, if not appropriately addressed can seriously hamper interactive visualization and exploration for discovery. Past visualizations were limited to downsized versions of the data on a regular grid. The development of advanced algorithms and software for parallel visualization of unstructured hexahedral datasets that scale to the very large grid sizes required will significantly assist their ability to interpret and understand earthquake simulations.

This paper presents the design and performance of a parallel visualization pipeline for time-varying unstructured volume data generated from terascale earthquake simulations. In our previous work [16], a parallel volume renderer was developed for visualizing 3D unstructured volume data generated from the same, but smaller scale, earthquake simulation [24]. The renderer performed satisfactorily for modest data sizes containing ten-million cells and could deliver rendering rates

of about 2 seconds per fame when using up to 128 processors of the HP AlphaServer operated at PSC. As the data size grows the primitive I/O scheme we chose ceases to work well. Even though a parallel file system was used, the I/O cost became so high that it totally dominated the overall cost. The interframe delay for rendering 100 million data cells became 15-20 seconds, which is not acceptable.

Our new parallel visualization pipeline design not only removes the I/O bottleneck but also facilitates the preprocessing calculations required to derive more sophisticated or expressive visualization rendering lighting, feature enhancement, and vector fields. The pipeline incorporates I/O strategies that adapt to the data size and parallel system performance such that I/O and data preprocessing costs can be effectively hidden. Interframe delay becomes completely determined by the rendering cost. Consequently, as long as a sufficient number of rendering processors are used, desired framerates can be obtained. We demonstrate this new parallel I/O solution implemented in MPI I/O [8] for making volume visualization of the highest resolution earthquake simulation performed to date.

2. PREVIOUS WORK

The research problem we address has multiple facets including large time-varying data, parallel I/O, parallel rendering, unstructured grids, and vector fields, none of which can be neglected if our goal is to derive a usable solution. Little previous research has been done to address all aspects of the problem in the context of visualization.

2.1 Time-varying data

Visualizing time-varying data presents two challenges. The first is the need to periodically transfer sequences of time steps to the processors from disk through a data server. The second is the need for an exploration mechanism accompanied by an appropriate user interface for tracking and interpreting the temporal aspects of the data. We have focused on I/O and aim to hide the I/O cost to reduce interframe delay. For interactive browsing in both the spatial and temporal domains of the data, a minimum of 2–5 frames per second is needed. McPherson and Maltrud [22] develop a visualization system capable of delivering realtime animation of large time-varying ocean flow data. The system exploits the high performance volume rendering of texture-mapping hardware on four InfiniteReality pipes attached to an SGI Origin 2000 with enough memory to hold thousands of time steps of the data. The ParVox system [13] is designed to achieve interactive visualization of time-varying volume data in a high-performance computing environment. Highly interactive splatting-based rendering is achieved by overlapping rendering and compositing, and by using compression.

A survey of time-varying data visualization strategies developed more recently is given in [17]. One very effective strategy is based on a hardware decoding technique that keeps the data compressed until reaching the video memory for rendering [14]. Even though encoding methods can significantly reduce the data size, the preprocessing cost and additional data storage requirements are not always desirable and affordable. In the absence of support for a high-speed network and parallel I/O, a particularly promising strategy for achieving interactive visualization is to perform pipelined rendering. Ma and Camp [18] show that by properly grouping processors according to the rendering loads, compressing images before delivering, and completely overlapping the uploading, rendering, and delivering of the images, interframe delay can be kept to a minimum. Reinhard et al. [25] use a data partitioning approach to enable highly efficient ray traced isosurface visualization of time-varying data.

2.2 Parallel I/O

In the study of parallel rendering algorithms, I/O cost is often ignored. The most common strategy is to overlap communication and computation, which does not solve the problem of disk contention. In our previous work [16], we show that the use of multiple I/O nodes can maximize bandwidth and reduce latency. We experimentally determine the number of I/O nodes required. In this work, we study this I/O issue further and develop two parallel I/O strategies. In particular, we show the number of I/O nodes can be analytically computed.

The MPI I/O interface [8] supports a suite of parallel I/O operations but very little use of MPI I/O has been found in parallel visualization applications. As described in Section 5.3, our work relies on MPI I/O extensively.

Data file formats such as HDF [9] and netCDF [12] that are widely used by scientific applications have parallel I/O support . However, our earthquake simulation data files are in neither HDF nor NetCDF so we had to develop new parallel I/O strategies through MPI I/O.

2.3 Parallel and distributed rendering

Our approach to the large data problem is to distribute both the data and visualization calculations to multiple processors of a parallel computer. In this way, we not only can visualize the dataset at its highest resolution, but also achieve interactive rendering rates. The parallel rendering algorithm thus must be highly efficient and scalable to a large number of processors. Ma and Crockett [20] demonstrate a highly efficient, cell-projection volume rendering algorithm using up to 512 T3E processors for rendering 18 million tetrahedral elements from an aerodynamic flow simulation. They achieve over 75% parallel efficiency by amortizing the communication cost and using a fine-grain image-space load partitioning strategy. Parker et al. [23] use ray tracing techniques to render images of isosurfaces. Although ray tracing is a computationally expensive process, it is highly parallelizable and scalable on shared-memory multiprocessor computers. By incorporating a set of optimization techniques and advanced lighting, they demonstrate interactive, high-quality isosurface visualization of the Visible Woman dataset using up to 124 nodes of an SGI Reality Monster with 80%–95% parallel efficiency. Wylie et al. [30] show how to achieve scalable rendering of large isosurfaces (7–469 million triangles) and rendering performance of 300 million triangles per second using a 64-node PC cluster with a commodity graphics card on each node. The two key optimizations they use are lowering the size of the image data that must be transferred among nodes by employing compression, and performing compositing directly on compressed data. Bethel et al. [4] introduce a very unique remote and distributed visualization architecture as a promising solution to very large scale data visualization.

2.4 Unstructured-grid data

To efficiently visualize unstructured data, additional information about the structure of the mesh needs to be computed and stored, which incurs considerable memory and computational overhead. For example, ray tracing needs explicit connectivity information for each ray to march from one element to the next [15]. The rendering algorithm introduced by Ma and Crockett [19] requires no connectivity information. Since each tetrahedral element is rendered independently of other elements, data distribution can be done in a more flexible manner. Chen, Fujishiro, and Nakajima [6] present a hybrid parallel rendering algorithm for large-scale unstructured data visualization on SMP clusters such as the Hitachi SR8000. Their three-level hybrid parallelization consists of message passing for inter-SMP node communication, loop directives by OpenMP for intra-SMP node parallelization, and vectorization for each processor. A set of optimization techniques are used to achieve maximum parallel efficiency. In particular, due to their use of an SMP machine, dynamic load balancing can be done effectively. However, their work does not address the problem of rendering time-varying data.

2.5 Vector field

A variety of techniques have been developed for rendering of vector fields. We adopt a texture-based method called Line Integral Convolution (LIC) [5]. The input to LIC is a vector field and a white noise image. Visualization is generated by first tracing a streamline both forward and backward for each data point and then convolving in one dimension along the streamline. By using a periodic filter kernel, an animation giving an impression of the flow direction and structure can be made. Texture-based methods can also be used to efficiently depict time-dependent vector fields [26, 10, 7] and can be made highly interactive [29].

3. EARTHQUAKE-INDUCED GROUND MOTION MODELING

Modeling and forecasting earthquake ground motion in large basins is a challenging and complex task. The complexity arises from several sources. First, multiple spatial scales characterize the basin response: the shortest wavelengths measure in tens of meters, whereas the longest measure in kilometers, and basin dimensions are on the order of tens of kilometers. Second, temporal scales vary from the hundredths of a second necessary to resolve the highest frequencies up to several minutes of shaking within the basin. Third, many basins have highly irregular geometry. Fourth, the soil properties are highly heterogeneous. Fifth, strong earthquakes give rise to nonlinear material behavior. And sixth, geology and source parameters are only indirectly observable, and thus introduce uncertainty into the modeling process.

Simulating the earthquake response of a large basin is accomplished by numerically solving the partial differential equations (PDEs) of elastic wave propagation [2]. An finite element method employing an unstructured mesh is used for spatial approximation, and an explicit central difference scheme is used in time. The mesh size is tailored to the local wavelength of propagating waves via an octree-based mesh generator [28]. Even though using an unstructured mesh may yield three orders of magnitude fewer equations than with structured grids, a massively parallel computer still must be employed to solve the resulting dynamic equations.

The earthquake modeling team is currently performing simulations in the greater LA basin to 10 meters at the finest resolution with 100 million unstructured hexahedral finite elements, a factor of 4000 smaller than a regular grid would require. These include simulations of the 1994 Northridge mainshock to 1 Hz resolution, the highest resolution obtained to date. Despite the large degree of irregularity of the meshes, the codes are highly efficient: close to 90% parallel efficiency is regularly obtained in scaling up from 1 to 2048 processors on the HP/Compaq AlphaServer-based parallel system at the Pittsburgh Supercomputing Center. Node performance is also excellent for an unstructured mesh code, permitting sustained throughputs of nearly one teraflop per second on 2048 processors. A typical simulation requires 25,000 time steps to simulate 40 seconds of ground shaking, and requires wall-clock time on the order of several hours, depending on the material damping model used, size of the region considered, number of processors (between 512 and 2048), and output statistics required. Figure 1 displays eight selected time steps from the simulation rendered using our visualization system.

4. THE PARALLEL RENDERING METHOD

The basic architecture of our parallel visualization solution is shown in Figure 2. It is essentially a parallel pipeline and becomes the most efficient when all pipeline stages are filled. The input processors read data files from the storage device which in our design must be a parallel file system, prepare the raw data for rendering calculations, and distribute the resulting data blocks to the rendering processors. The rendering processors produce volume-rendered images for its local data blocks, perform image compositing, and deliver the images to the output processors which then send the images to a display or storage device.

Since the mesh structure never changes throughout the simulation, a one-time preprocessing step is done to generate a spatial (octree) encoding of the raw data. The input processors use this octree along with a workload estimation method to distribute blocks of hexahedral elements among the rendering processors. Each block of elements is associated with a subtree of the global octree. This subtree is delivered to the assigned rendering processor for the corresponding block of data only once at the beginning since all time steps data use the same subtree structure. Non-blocking send and receive operations are used for the blocks distribution.

In addition to determining the partitioning and distribution of data blocks, each input processor also performs a set of calculations to prepare the data for rendering. Typical calculations include quantization (from 32-bit to 8-bit), differencing to derive gradient vectors for lighting or rates of change for temporal domain enhancement, and texture synthesis for depicting vector fields. Lighting and temporal domain enhancement are optional. As we will show later, the number of preprocessing calculations can influence the optimal system configuration for rendering. Note that it is often more convenient and economical to conduct these preprocessing tasks at the input processors rather than the
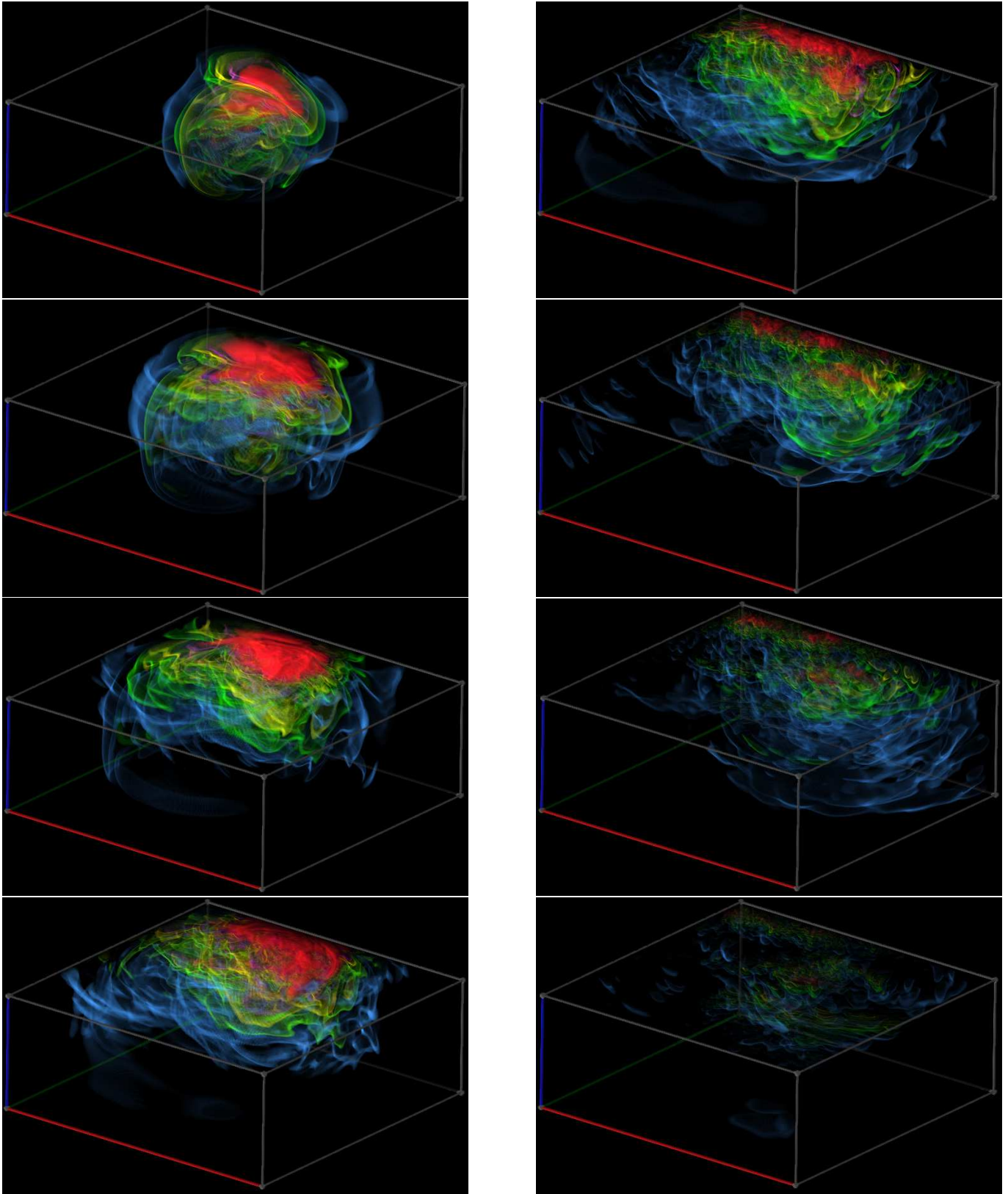
Figure 1: Visualization of velocity magnitude for selected time steps from the earthquake simulation. Left: time steps 50, 75, 100, 125. Right: time steps 150, 200, 250, 350.
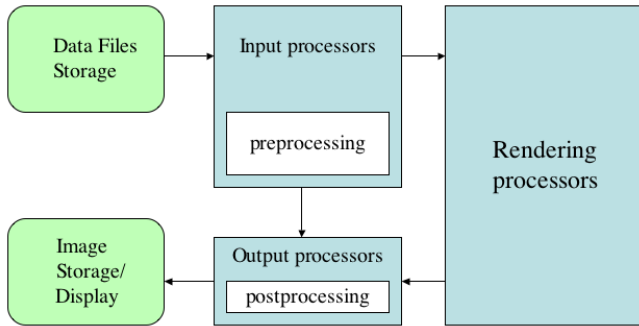
Figure 2: The architecture of the parallel visualization solution. The input processors are mainly responsible for uploading each time steps of the data. They are also used for calculations that are easier or only possible to do before the data are split and distributed to the rendering processors. The Output processors deliver the final rendered images to a display or storage device.
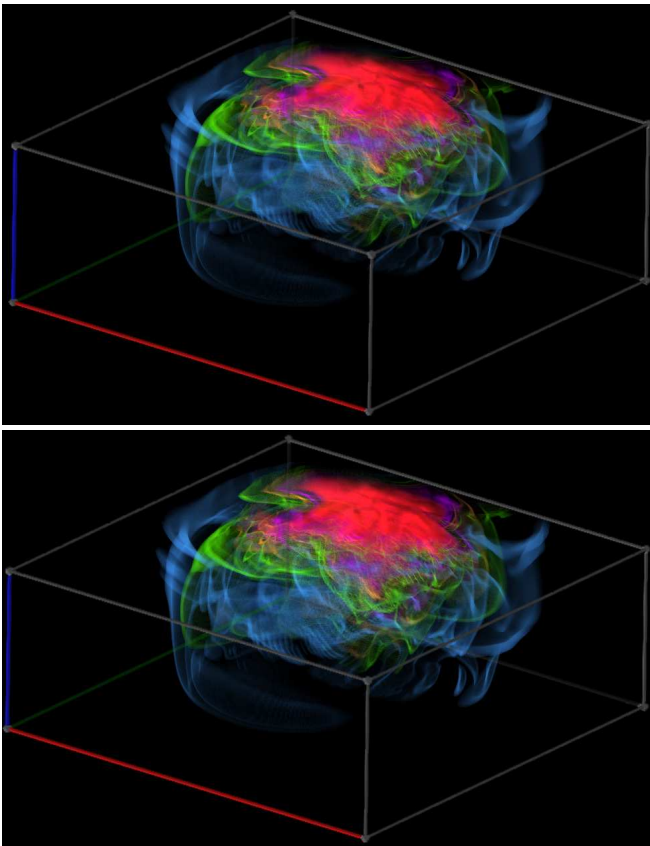
Figure 3: Top: high-resolution rendering (level 13). Bottom: Adaptive rendering (level 8). Both images were rendered at 1024×1024 pixels. The bottom image reveals almost the same details as the top image while being generated 3–4 times faster.

rendering processors. First, data replication is avoided because the input processors have access to all the needed data. Second, as with I/O, the calculations become free because of the parallel pipelining.

The number of rendering processors used is selected based on the rendering performance requirements. After each rendering processor receives a subset of the volume data through the input processors, our parallel rendering algorithm performs a sequence of tasks: view-dependent preprocessing, local volume rendering, image compositing, and image delivery. Before the local rendering step begins, each rendering processor conducts a view-dependent preprocessing step whose cost is very small and thus negligible. As described later, this preprocessing is used to optimize the image compositing step. While rendering calculations are carried out, new data blocks for subsequent time steps are continuously transferred from the input processors in the background. As expected, overlapping data transport and rendering helps lower interframe delay.

4.1 Adaptive rendering

Rendering cost can be cut significantly by moving up the octree and rendering at coarser-level blocks instead. This is done for maintaining the needed interactivity for exploring in the visualization parameter and data spaces. A good approach is to render adaptively by matching the data resolution to the image resolution while taking into account the desired rendering rates. For example, when rendering tens of millions of elements to a 512×512 pixels image, rendering at the highest resolution level does not reveal more details unless a close-up view is selected. One of the calculations that the view-dependent preprocessing step performs is to choose the appropriate octree level. The savings from such an adaptive approach can be tremendous and there is very little impact on the level of information presented in the resulting images, as shown in Figure 3. Presently the appropriate level to use is computed based on the image resolution, data resolution, and a user-specified limit to the number of elements that project to the same pixel.

4.2 Enhancement rendering

Because of the large dynamic range of the data, it is often difficult to follow time-varying phenomena. For example, half way into the simulated period, direct volume rendering reveals very little variation in the domain without modifying the opacity mapping used. We have employed a new temporal domain filtering method to enhance the wave propagation throughout the whole time period [16]. The enhancement is done locally by using values in either the previous or the next time step, or both. As a result, both large-scale and small-scale wave propagations are captured in the picture. The user can turn the enhancement on and off during interactive viewing to ensure a correct interpretation of the data. The cost of computing this enhancement is small, and the input processors are excellent for such preprocessing calculations. Figure 4 contrasts the images rendered with and without enhancement for one of the later time steps.

4.3 Vector field rendering

In the earthquake ground motion modeling, the computational mesh is most dense near the ground surface. More
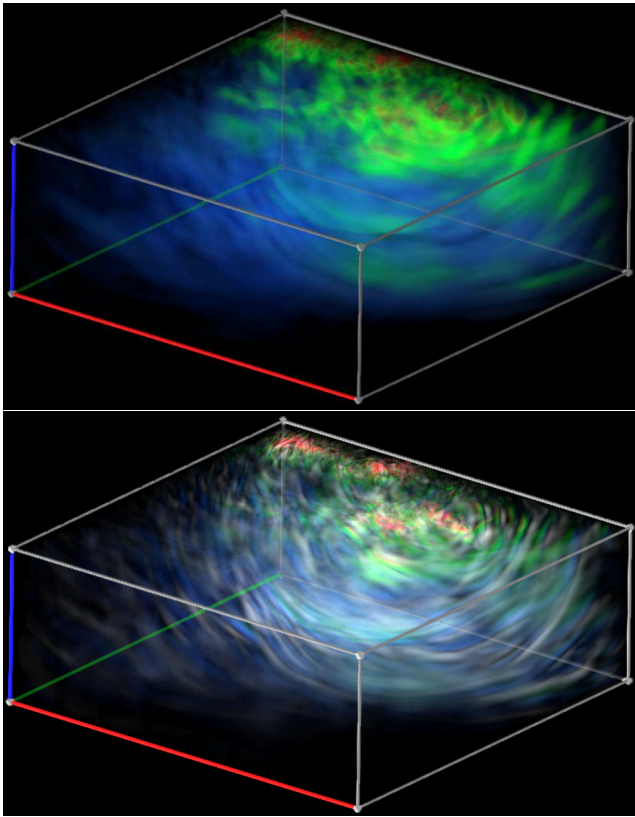
Figure 4: Visualization of time step 200. Top: without enhancement. Bottom: with enhancement. The enhancement brings out the wave propagation.

than 20 percents of mesh points are near the surface regions. There is thus a strong interest in understanding the characteristics of the various scalar and vector fields near the ground surface. We have attempted to add the capability of visualizing vector field on the surface using LIC.

In our approach, a quadtree is first constructed to organize all nodes on the top surface. For each time step, the 2D vector field on the surface is extracted from the raw 3D vector fields. Since the extracted vector field is on an irregular grid, to simplify the later LIC calculations a 2D regular-grid vector field is derived using the underlying quadtree. This step can done either as a preprocessing step or on the fly. The resolution of the 2D regular-grid vector filed is determined by the image size and the adaptive levels selected by the user. During rendering time, the LIC images can be directly computed from the vector field on the Input processors. The resultant images are then sent to the Output processors to be composited with the volume rendered images (see Figure 2). Since the I/O processors execute concurrently with the rendering processors, it is possible to hide the cost of vector field rendering if a sufficient number of Input processors are used.

4.4 Parallel image compositing
The parallel rendering algorithm is sort-last which thus requires a final compositing step involving inter-processor communication. Most parallel image compositing algorithms

were designed to achieve high efficiency on specific network topology [21, 11, 1]. We have adopted SLIC [27] which is an optimized version of the *direct send* compositing method to offer maximum flexibility and performance. The direct send method has each processor send pixels directly to the processor responsible for compositing them. This approach has been widely used; it is easy to implement and does not require a special network topology. With direct send compositing, in the worst case there are $n(n-1)$ messages to be exchanged among n compositing nodes. For low-bandwidth networks, care should be taken to avoid many-to-one or many-to-many communication.

SLIC uses a minimal number of messages to complete the parallel compositing task. The optimizations are achieved by using a view-dependent precomputed compositing schedule. Reducing the number of messages that must be exchanged among processors should be beneficial since it is generally true that communication is more expensive than computation. The preprocessing step to compute a compositing schedule for each new view introduces very low overhead, generally under 10 milliseconds. With the resulting schedule, the total amount of data that must be sent over the entire network to accomplish the compositing task is minimized. According to our test results, SLIC outperforms previous algorithms, especially when rendering high-resolution images, like 1024×1024 pixels or larger. Since image compositing contributes to the parallelization overheads, reducing its cost helps improve parallel efficiency.

5. I/O STRATEGIES
Our objective is to make the rendering performance independent of the I/O requirements. This is possible if some form of parallel I/O support is available. The computing environment at PSC has several parallel file systems connected by high-speed networks. We have studied how to effectively utilize these high performance computing resources. Our designs use parallel pipelining. In addition to employing multiple rendering processors, multiple input processors are used to maximize data rates with concurrent reads and writes [31]. The parallel pipelining becomes the most efficient when the I/O costs are hidden so that the rendering time dominates the overall turnaround time and interframe delay.

5.1 1D input processors (IDIP)
To maximize bandwidth utilization of the parallel file system, it is advantageous to use multiple I/O processes with each processor reading and preprocessing a complete, single time step of the data. In this way, best performance can be achieved if $T_f + T_p = T_s(m-1)$ where T_f is the time to fetch the data, T_p the preprocessing time, T_s the time to send the data to a rendering processor, and m the number of processors used. As a result, the number of input processors which should be used is $m = \frac{T_f + T_p}{T_s} + 1$. This would eliminate the idle time of a rendering processor between receiving two consecutive time steps, as shown in Figure 5. When T_s is smaller than the rendering time T_r which normally is the case, we can let $m = \frac{T_f + T_p}{T_r} + 1$ instead, which allows us to use fewer input processors but still keep the rendering processors busy.
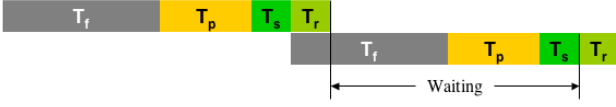
Figure 5: Overlapping I/O and rendering calculations. Only when I/O time is not larger than the rendering time can we effectively hide the I/O cost.

5.2 2D input processors (2DIP)

The strategy 1DIP works well until T_s become larger than T_r. That is, even though we can increase the rendering rates by using more rendering processors, the 1DIP approach limits how much we can reduce T_s. We have investigated an alternative design which uses a two-dimensional configuration of input processors. Basically, there are n groups of m input processors. Each group of processors is responsible for reading, preprocessing, and distributing one complete time step of the data.

Since each time step of the data is distributed among all the rendering processors, with m input processors working on one time step, it takes about $T_s' = \frac{T_s}{m}$ time for the m input processors to deliver the data blocks. Now we can control m to keep T_s' smaller than T_r so it becomes possible to make the rendering processors busy all the time. Note that in this way we also spread the preprocessing cost and $T_p' = \frac{T_p}{m}$.

Given $T_s' \leq T_r$ and $T_s' = \frac{T_s}{m}$, we can obtain $m \geq \frac{T_s}{T_r}$. Similarly as with 1DIP, we let $T_f' + T_p' = T_s'(n-1)$. Consequently, $n = \frac{(T_f' + T_p')}{T_s'} + 1$. When $T_s' = T_r$, $m = \frac{T_s}{T_r}$ and $n = \frac{(T_f' + T_p')}{T_r} + 1$. Assume each input processor deals with exactly $\frac{1}{m}$ of the data. Then ideally $T_p' = \frac{T_p}{m}$ and $T_f' = \frac{T_f}{m}$. Thus, $n = \frac{(T_f/m + T_p/m)}{T_r} + 1 = \frac{(T_f + T_p)}{T_s} + 1$. In summary, to render a time-varying dataset, we can therefore use 1DIP when T_r is greater than T_s; otherwise, 2DIP should be used. Figure 6 contrasts 1DIP and 2DIP configurations.

5.3 File reading strategies

MPI-IO, the I/O part of the MPI-2 standard [8], is an interface designed for portable, high-performance I/O. For example, it provides Data Sieving to enable more efficient read of many noncontiguous data and Collective I/O to allow for merging of the I/O requests from different processors and servicing the merged request. Our designs use both Data Sieving and Collective I/O for 2DIP. However, we have also developed an alternative approach which experimentally proves to be more efficient for reading noncontiguous data. Our design requires a parallel file system with a high bandwidth.

In the 2DIP case, m input processors fetch, preprocess, and distribute one time step dataset. Recall that, as a load bal-
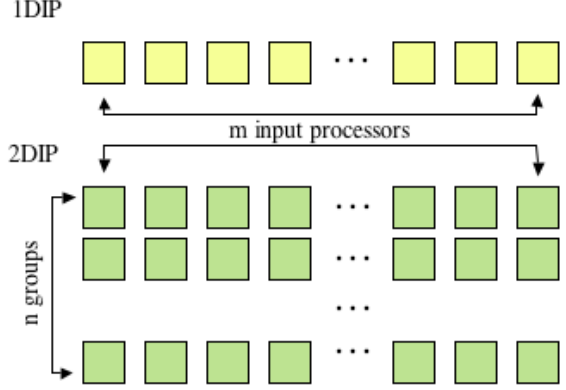

Figure 6: The 1DIP and 2DIP configurations. With 1DIP, each input processor fetches one time step of the data so m time steps are fetched concurrently. With 2DIP, each group of input processors fetches one time step so n time steps are read concurrently.

ancing strategy, each rendering processor receives multiple octree blocks which spread the spatial domain of the data. In order to make data subsets ready for each rendering processor, each input processor must reconstruct the hexahedral cell data from the node data according to the octree data. Since the node data is stored as a linear array on the disk, each processor must make noncontiguous reads to recover the cell data for each octree block. The parallel I/O support offered by MPI-IO makes this task easier.

The biggest bottleneck is reading data from the disk storage system to the input processors. While it is clear using multiple input processors helps increase the bandwidth, we are interested in determining the minimal number of input processors that must be used for a preselected renderer size to achieve the desired frame rates. Parallel reads may be done in the following two ways.

5.3.1 Single collective and noncontiguous read.

In the first strategy, we rely on MPI-IO support. All input processors fetch a roughly equal number of hexahedral cells from the disk. Grouping of the cell data is done according to the octree data and the load balancing strategy. To avoid duplicating node data, octree data are merged for each rendering processor. Each of the m input processors uses

- MPI_TYPE_CREATE_INDEXED_BLOCK to derive a data type (e.g., an array of node data) from the octree data. The derived data type describes one reading pattern;

- MPI_FILE_SET_VIEW to set the derived data type as the reading pattern of the current input processor; and

- MPI_FILE_READ_ALL to collectively read the data along with other input processors.

At the end, each input processor has a subset of the current time step of the cell data to be distributed among the
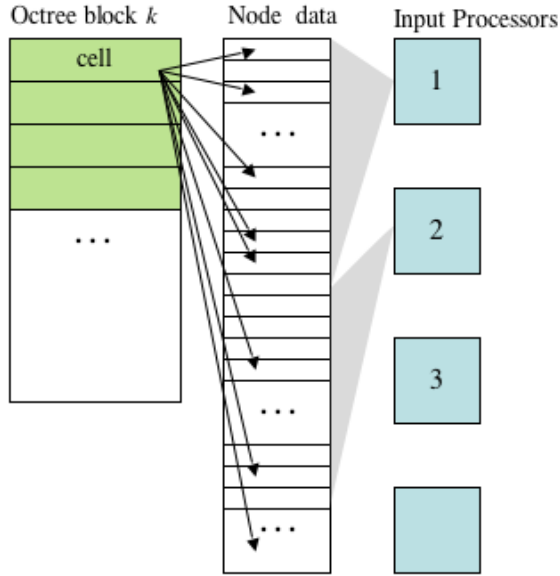
Figure 7: Octree blocks are assigned to rendering processors according to a load balancing strategy. Using the second reading method, the node data belonging to the octree block k is likely to be spread across multiple input processors. There is a merging process at every rendering processor to gather all the relevant node data.

rendering processors.

5.3.2 Independent contiguous read.

In this case, each input processor independently reads the contiguous $\frac{1}{m}$ of a time step of the node data. Both the node data and the octree data are 1D arrays as shown in Figure 7. The node data of a particular octree block k likely spread across multiple input processors. Each input processor therefore scans through the octree data and creates a mapping between its local node data and the corresponding octree blocks. Each input processor then forwards both the node data and the map to the rendering processors according to a load balancing strategy. Each rendering processor has to merge the incoming data to form complete local octree blocks of data. No communication between processors are needed for the merge operations. This strategy is superior if the overhead of collective I/O would become too high.

6. TEST RESULTS

We present the performance of our parallel visualization pipeline on LeMieux, an HP/Compaq AlphaServer with 3,000 processors operated at the Pittsburgh Supercomputing Center for the visualization of time-varying ground motion simulation data consisting of 100 million hexahedral elements. Each time step of the data to be transferred is about 400 megabytes.

Figure 8 shows rendering performance using 64 rendering processors with the 1DIP strategy. The image size is 512×512. The rendering time is about 2 seconds, and the total time
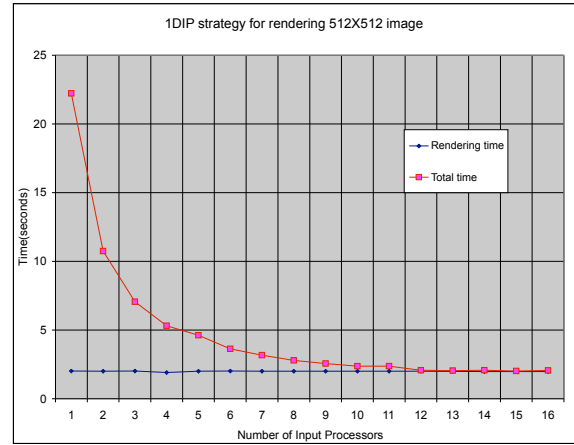

Figure 8: 64 rendering processors using the 1DIP strategy. Image resolution is 512×512. With 12 input processors, the total time due to I/O and preprocessing is reduced to about 2 seconds, very close to the rendering time.

due to I/O and preprocessing is about 22 seconds if only a single input processor is used. Preprocessing cost includes the time to do volume partitioning, load balancing, and quantization. In Figure 8, we can see when using 12 input processors the total time due to I/O and preprocessing becomes very close to the rendering time, making possible hiding of the I/O and preprocessing cost.

Figure 9 compares 1DIP and 2DIP. Recall that the purpose of using 2DIP is to employ multiple input processors to fetch a single time step of the data for further cutting down the sending time, in contrast to 1DIP which concurrently reads multiple time steps. The test results show that when rendering time is low 2DIP should be used so that the I/O and preprocessing cost can be effectively hidden. Note that in this set of tests, Independent Contiguous Read was used for 2DIP since according to our previous study it is superior to Collective Noncontiguous Read. A thorough performance study of the 1DIP and 2DIP I/O strategies is presented in [31].

When adaptive rendering is used, I/O cost can be reduced significantly by using adaptive data fetching. That is, only data cells at the selected level are fetched from the disk by the input processors using MPI I/O. Our test results show that when rendering 512×512 images using 1DIP and adaptive level 8 with 64 rendering processors, only four input processors are needed to reach the best possible parallel pipelining. Without using adaptive fetching, as shown previously in Figure 8, 12 input processors are needed.

In 3D graphics, lighting is employed to help convey shape and orientation information. In the context of flow visualization, lighting helps illustrate feature surfaces and their spatial relationship. Adding lighting requires calculations of gradient information to approximate local surface orientation plus solving the lighting equation at each sample point.
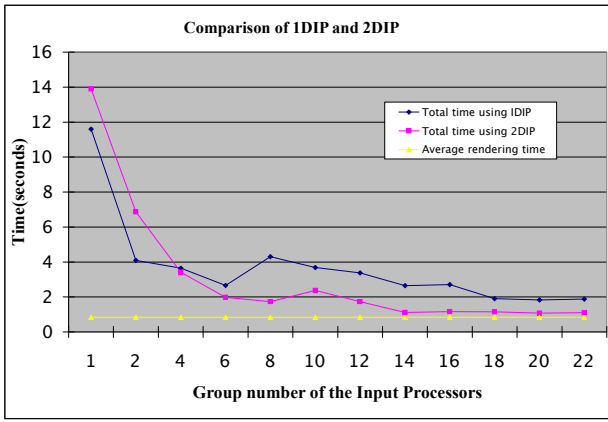
Figure 9: Comparing 1DIP and 2DIP using 128 rendering processors for rendering 512×512 images. The rendering time is reduced to about 1 second. In this case, overlapping rendering and I/O is only possible with 2DIP.
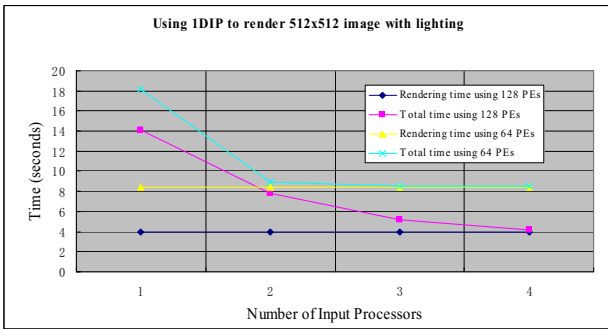

Figure 10: Rendering 256×256 images with lighting and adaptive fetching. The cost of adding lighting is high so fewer input processors, 3 and 4, are required when using 64 and 128 rendering processors, respectively.

Using the input processors to compute gradients requires transmitting the gradient vectors to the rendering processors. It is thus advantageous to compute gradient on the rendering processors. Figure 10 shows the cost of rendering with lighting using 64 and 128 rendering processors. In both cases, a much smaller number of input processors are needed because of the higher cost of rendering and using adaptive feathing. Figure 11 contrasts the images rendered with and without lighting.

Adding 2D LIC images can be done as a preprocessing step and handled by the Input processors. Figure 12 shows the cost of making simultaneous surface LIC and volume visualization using 64 rendering processors with 1DIP strategy. When 16 input processors are used, computing the LIC images, other preprocessing, and I/O essentially become free. Figure 13 displays four selected time steps of simultaneous scalar and vector fields visualization. Figure 14 shows the LIC image for time step 200 and two close-up views. When
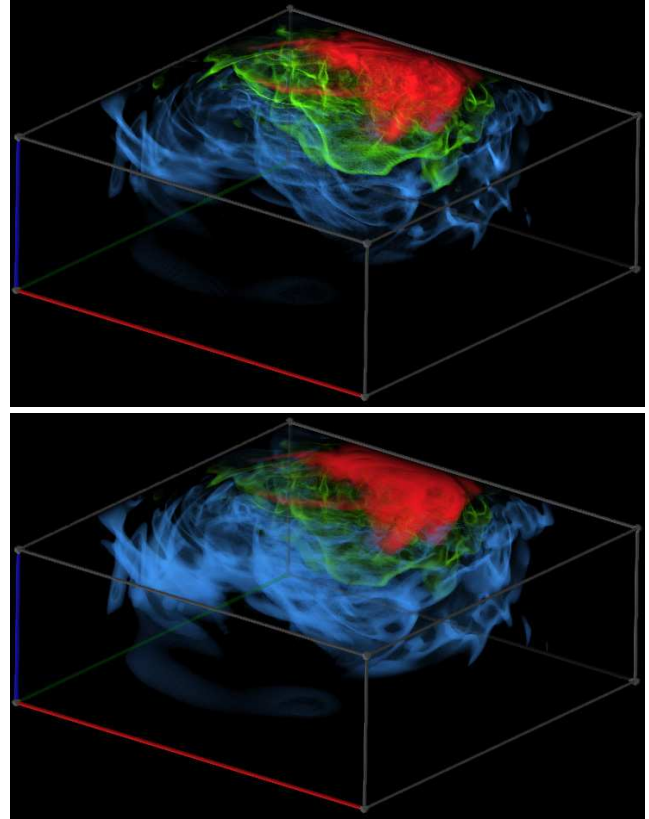

Figure 11: Top: with lighting. Bottom: without lighting. Adding lighting results in visualization showing the flow structure with greater clarity.
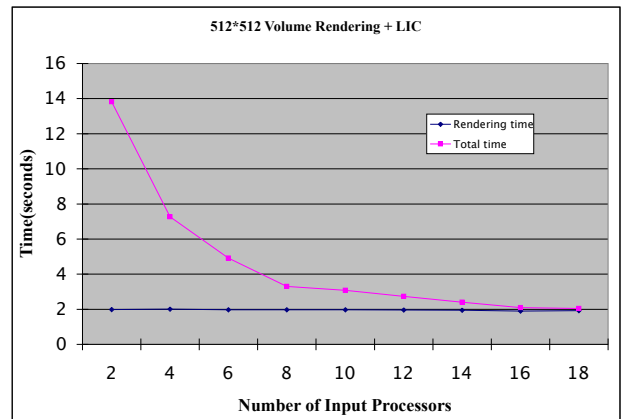

Figure 12: The cost of making simultaneous surface LIC and volume visualizations using 64 rendering processors with 1DIP strategy. When 16 input processors are used, the cost of computing the LIC images and the I/O can be completely hidden.

the interframe delay can be cut below 1-2 seconds, it is possible for the user to explore the temporal, spatial, and variable domains of the simulation. Such a capability was not previously available for datasets at this scale.

7. CONCLUSIONS AND FUTURE WORK

We have developed a highly efficient parallel visualization pipeline based on an effective processor partitioning scheme which facilitates overlapping I/O operations, preprocessing, and rendering. This parallel visualization solution also incorporates adaptive rendering, a highly efficient parallel image compositing algorithm, and new I/O strategies to make possible near-interactive visualization of terascale earthquake simulations. Our performance study using LeMieux at the PSC demonstrates convincing results, and also reveals the interplay between data transport strategy and interframe delay.

We have addressed the I/O problem of massively parallel rendering. We have demonstrated that using multiple data servers, adaptive fetching, and MPI I/O helps not only removes the I/O bottleneck, but also hides preprocessing cost. Presently, the input processors also handle load balancing statically. We plan to investigate a fine-grain load redistribution method and study how to reduce its overhead as much as possible.

Presently, the image compositing cost is about constant. We believe compression can help lower communication cost to make the overall compositing scalable to large machine sizes. Our preliminary test results show a 50% reduction in the overall image compositing time with compression.

We have not exploited the SMP features of LeMieux, which we believe could allow us to accelerate the rendering calculations while reducing communication cost. The result will be a more scalable renderer offering higher frame rates.

Adaptive rendering will continue to play a major role in our subsequent work. As shown, full rendering and adaptive rendering often result in visually indistinguishable results, but the savings in rendering cost can be tremendous. Our further study in this direction will focus on how adaptive rendering can be done with minimal user intervention and perception of level switching.

Our ultimate goal is to perform simulation-time visualization allowing scientists to monitor the simulation, make immediate decisions on data archiving and visualization production, and even steer the simulation. To achieve such an ambitious goal, we have started by first developing a highly efficient parallel visualization algorithm that is capable of delivering interactive rendering of terascale datasets, scalable to large MPP systems, and easily coupled with extended capabilities such as vector field rendering. It is also essential to develop appropriate user interfaces and interaction techniques for interactive browsing in both the spatial and temporal domains of the data.

Finally, the parallel simulation and renderer will run simultaneously on either the same machine or two different machines connected with a high-speed network interconnect permitting remote interaction with the simulation and vi-
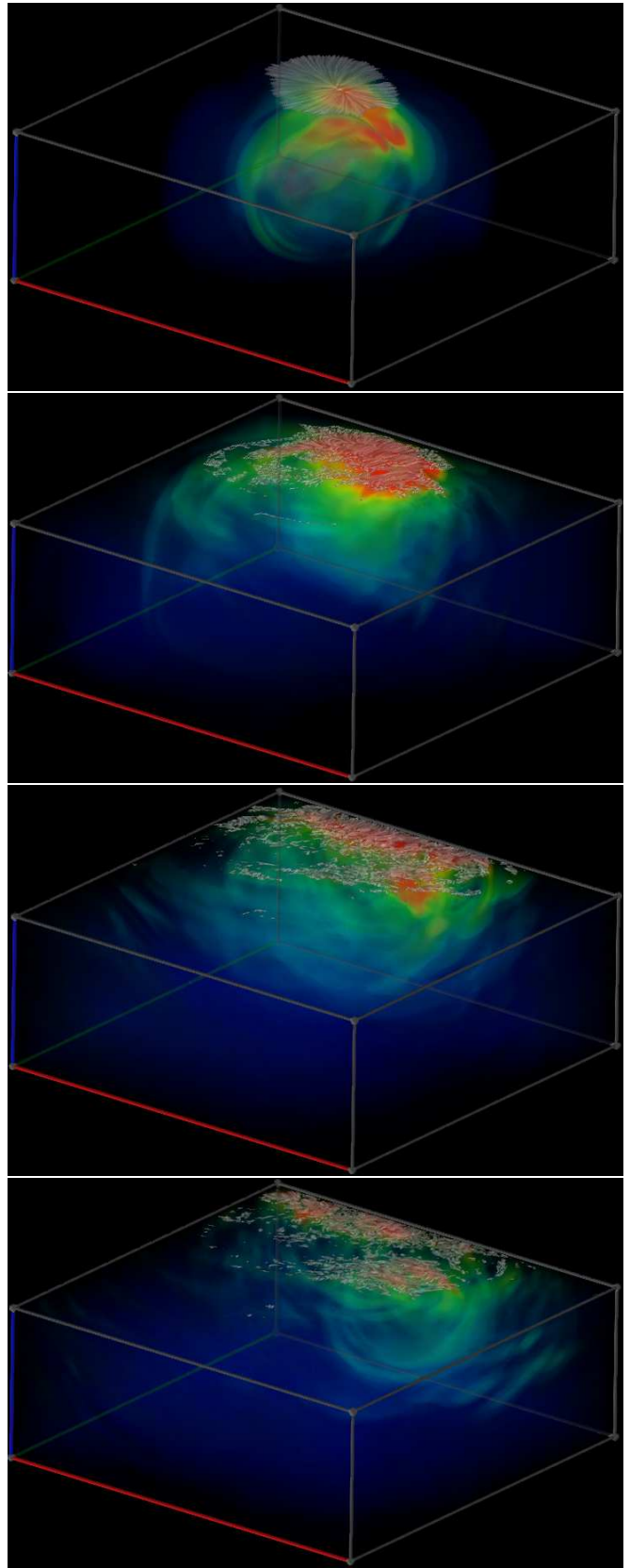

Figure 13: Simultaneous volume rendering (without lighting) and surface LIC visualization for selected time steps: 50, 100, 150, and 200.
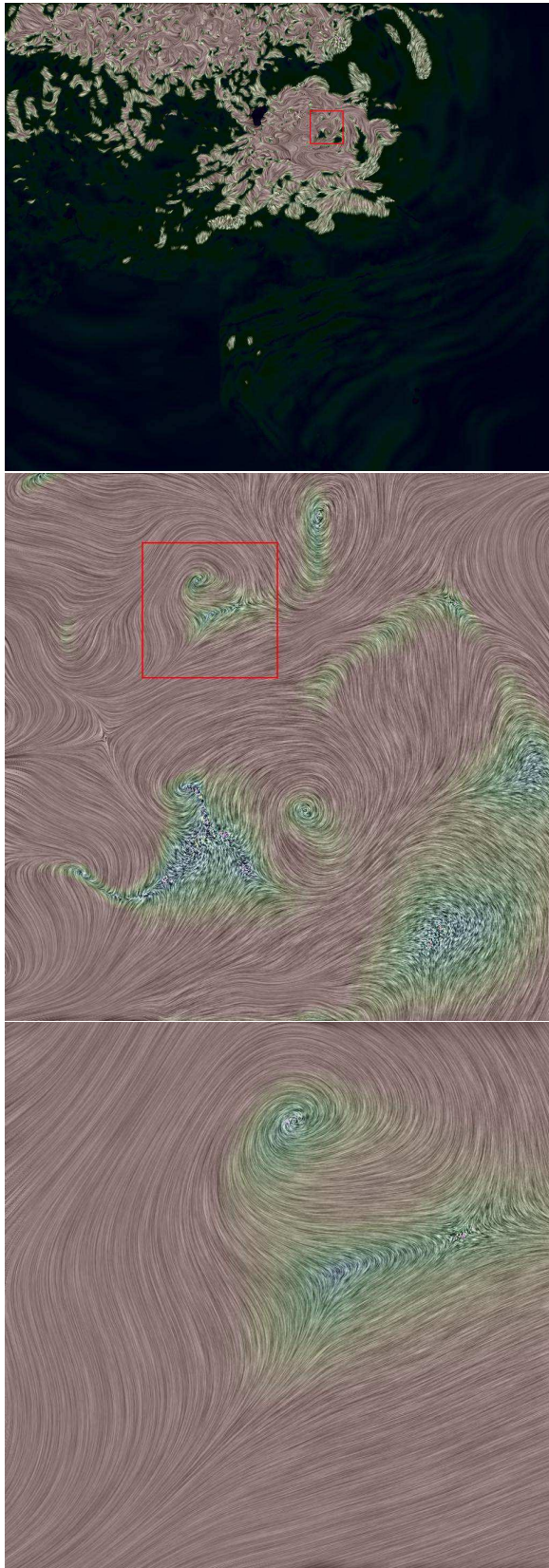
sualization. We will therefore also investigate the use of a graphics-enhanced PC cluster as a dedicated visualization server. The question then is whether our I/O strategies can keep up with hardware accelerated rendering.

Acknowledgments

8.   REFERENCES

[1] J. Ahrens and J. Painter. Efficient sort-last rendering using compression-based image compositing. In *Proceedings of the 2nd Eurographics Workshop on Parallel Graphics and Visualization*, pages 145–151, 1998.

[2] H. Bao, J. Bielak, O. Ghattas, L. F. Kallivokas, D. R. O'Hallaron, J. R. Shewchuk, and J. Xu. Large-scale simulation of elastic wave propagation in heterogeneous media on parallel computers. *Computer Methods in Applied Mechanics and Engineering*, 152(1–2):85–102, Jan. 1998.

[3] H. Bao, J. Bielak, O. Ghattas, D. R. O'Hallaron, L. F. Kallivokas, J. R. Shewchuk, and J. Xu. Earthquake ground motion modeling on parallel computers. In *Supercomputing '96*, Pittsburgh, Pennsylvania, Nov. 1996.

[4] W. Bethel, B. Tierney, J. Lee, D. Gunter, and S. Lau. Using high-speed WANs and network data caches to enable remote and distributed visualization. In *Proceedings of Supercomputing 2C00*, November 2000.

[5] B. Cabral and L. Leedom. Imaging vector fields using line integral convolution. In *SIGGRAPH '93 Conference Proceedings*, pages 263–270, August 1993.

[6] L. Chen, I. Fujishiro, and K. Nakajima. Parallel performance optimization of large-scale unstructured data visualization for the earth simulator. In *Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization*, pages 133–140, 2002.

[7] W. Daniel, E. Gordon, and E. Thomas. A texture-based framework for spacetime-coherent visualization of time-dependent vector fields. In *Proceedings of IEEE Visualization 2003 Conference*, pages 107–114, 2003.

[8] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2–Advanced Features of the Message Passing Interface*. MIT Press, 1999.

Figure 14: LIC image of the ground surface at time step 200. The bottom 2 images show increasingly close-up views of the field.

[9] HDF5 home page, the national center for supercomputing applications. http://hdf.ncsa.uiuc.edu/HDF5.

[10] B. Jobard, G. Erlebacher, and M. Hussaini. Lagrangian-eulerian advection for unsteady flow visualization. In *Proceedings of IEEE Visualization 2001 Conference*, pages 53–60, 2001.

[11] T.-Y. Lee, C. S. Raghavendra, and J. B. Nicholas. Image composition schemes for sort-last polygon rendering on 2d mesh multicomputers. *IEEE Transactions on Visualization and Computer Graphics*, 2(3):202–217, 1996.

[12] 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 *Proceedings of Supercomputing 2003 Conference*, November 2003.

[13] P. Li, S. Whitman, R. Mendoza, and J. Tsiao. ParVox – a parallel spaltting volume rendering system for distributed visualization. In *Proceedings of 1997 Symposium on Parallel Rendering*, pages 7–14, 1997.

[14] E. Lum, K.-L. Ma, and J. Clyne. A hardware-assisted scalable solution for interactive volume rendering of time-varying data. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):286–301, 2002.

[15] K.-L. Ma. Parallel volume ray-casting for unstructured-grid data on distributed-memory architectures. In *Proceedings of the Parallel Rendering '95 Symposium*, pages 23–30, 1995. Atlanta, Georgia, October 30-31.

[16] K.-L. Ma. Visualizing large-scale earthquake simulations. In *Proceedings of the Supercomputing 2003 Conference*, 2003.

[17] K.-L. Ma. Visualizing time-varying volume data. *IEEE Computing in Science & Engineering*, 5(2):34–42, 2003.

[18] K.-L. Ma and D. Camp. High performance visualization of time-varying volume data over a wide-area network. In *Proceedings of Supercomputing 2000 Conference*, November 2000.

[19] K.-L. Ma and T. Crockett. A scalable parallel cell-projection volume rendering algorithm for three-dimensional unstructured data. In *Proceedings of 1997 Symposium on Parallel Rendering*, pages 95–104, 1997.

[20] K.-L. Ma and T. Crockett. Parallel visualization of large-scale aerodynamics calculations: A case study on the cray t3e. In *Proceedings of 1999 IEEE Parallel Visualization and Graphics Symposium*, pages 15–20, 1999.

[21] K.-L. Ma, J. S. Painter, C. Hansen, and M. Krogh. Parallel Volume Rendering Using Binary-Swap Compositing. *IEEE Computer Graphics Applications*, 14(4):59–67, July 1994.

[22] A. McPherson and M. Maltrud. Poptex: Interactive ocean model visualization using texture mapping hardware. In *Proceedings of the Visualization '98 Conference*, pages 471–474, October 18-23 1998.

[23] S. Parker, M. Parker, Y. Livnat, P. Sloan, and C. Hansen. Interactive Ray Tracing for Volume Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 5(3):1–13, July-September 1999.

[24] The Quake project, Carnegie Mellon University and San Diego State University. http://www.cs.cmu.edu/~quake.

[25] E. Reinhard, C. Hansen, and S. Parker. Interactive ray tracing of time-varying data. In *Proceedings of the 4th Eurographics Workshop on Parallel Graphihcs and Visualization*, 2002.

[26] H.-W. Shen and D. Kao. A new line integral convolution algorithm for visualizing time-varying flow fields. *IEEE Transactions on Visualization and Computer Graphics*, 4(2), 1998.

[27] A. Stompel, K.-L. Ma, E. Lum, J. Ahrens, and J. Patchett. SLIC: scheduled linear image compositing for parallel volume rendering. In *Proceedings of IEEE Sympoisum on Parallel and Large-Data Visualization and Graphics (to appear)*, October 2003.

[28] T. Tu, D. O'Hallaron, and J. Lopez. Etree: A database-oriented method for generating large octree meshes. In *Proceedings of the Eleventh International Meshing Roundtable*, pages 127–138, September 2002.

[29] J. van Wijk. Image based flow visualization. In *Proceedings of SIGGRAPH 2002 Conference*, 2002.

[30] B. Wylie, C. Pavlakos, V. Lewis, and K. Moreland. Scalable rendering on PC clusters. *IEEE Computer Graphics and Applications*, 21(4):62–70, July/August 2001.

[31] H. Yu, K.-L. Ma, and J. Welling. I/O strategy for parallel rendering of large time-varying volume data. In *Proceedings of 2004 Parallel Graphics and Visualization Symposium (to appear)*, 2004.