Eurographics Symposium on Parallel Graphics and Visualization (2004)
Dirk Bartz, Bruno Raffin and Han-Wei Shen (Editors)

I/O Strategies for Parallel Rendering of Large Time-Varying Volume Data

Hongfeng Yu[1], Kwan-Liu Ma[1], and Joel Welling[2]

[1]University of California at Davis
[2] Pittsburgh Supercomputing Center

Abstract

This paper presents I/O solutions for the visualization of time-varying volume data in a parallel and distributed computing environment. Depending on the number of rendering processors used, our I/O strategies help significantly lower interframe delay by employing a set of I/O processors coupled with MPI parallel I/O support. The targeted application is earthquake modeling using a large 3D unstructured mesh consisting of one hundred millions cells. Our test results on the HP/Compaq AlphaServer operated at the Pittsburgh Supercomputing Center demonstrate that the I/O strategies effectively remove the I/O bottlenecks commonly present in time-varying data visualization. This high-performance visualization solution we provide to the scientists allows them to explore their data in the temporal, spatial, and visualization domains at high resolution. This new high-resolution explorability, likely not presently available to most computational science groups, will help lead to many new insights.

1. Introduction

In many areas of science and engineering, the capability to accurately model and understand time-varying physical phenomena or chemical processes enables new discoveries and is thus crucial to making continued innovation. The numerical modeling generally requires very high temporal and spatial resolutions, and, hence, demands the most capability from a massively parallel supercomputer. While the field of high-performance scientific computing intends to address such requirements, the associated data analysis and visualization tasks present even bigger challenges. A complete run of a time-varying simulation could output hundreds of gigabytes to terabytes of data even though not every time step of the data is stored. In addition to the overall size of the data set, what makes large time-varying 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.

We have previously developed a parallel renderer [MSB*03] for visualizing 3D unstructured volume data generated from a time-varying earthquake simulation code [Qua] in the high-performance computing environment of the Pittsburgh Supercomputing Center (PSC). The renderer performs satisfactorily for modest data sizes like tens of millions cells. As the data size grows the primitive I/O scheme we chose to use does not work well any more. Even though a parallel file system is used the I/O cost can become so high that it totally dominates the overall cost. In this paper, we present I/O strategies that adapt to the data size and parallel system performance such that I/O and data preprocessing costs could be effectively hidden. Interframe delay becomes completely determined by the rendering cost. Consequently, as long as a sufficient number of rendering processors are used, the desired framerates can be obtained. We demonstrate this new parallel I/O solution for making a volume visualization of the highest resolution earthquake simulation performed to date.

2. Driving Applications

Our work has been driven by several large-scale scientific applications including earthquake modeling, supernova modeling, ocean modeling, and turbulence modeling. While this paper places a focus on earthquake modeling, the visualization requirements and challenges are common to all applications. Simulating the earthquake response of a large basin is accomplished by numerically solving the par-

tial differential equations (PDEs) of elastic wave propagation [BBG*98]. An unstructured mesh finite element method 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 [TOL02]. A massively parallel computer must be employed to solve the resulting dynamic equations. The specific data set we used in our tests was from the modeling of the 1994 Northridge mainshock to 1Hz resolution, the highest resolution obtained to date, requiring a discretization of the greater LA basin to 10 meters finest resolution with 100 million unstructured hexahedral finite elements.

A typical dataset generated by the ground motion simulation may consist of thousands of time steps and the spatial domain is composed of 10-100 million elements. Each mesh node outputs six values, three displacement components and three velocity components. To efficiently browse both the temporal and spatial domains of the data, the corresponding visualization challenge is thus concerned with transferring and rendering large time-varying data possibly with multiple variables.

Several strategies are commonly used to achieve high performance rendering of large time-varying volume datasets in a parallel computing environment. Good load balancing can generally be achieved by distributing data and work load in an interleaving fashion. Rendering-time, per-time-step preprocessing calculations should be avoided or replaced with one-time preprocessing. Whenever possible, overlap communication and computation to hide data transfer overhead. Unless bandwidth is unlimited, buffering of intermediate rendering results can always effectively amortize communication overheads. Finally, compression can often significantly reduce the data that must be transferred, leading to lower communication cost. We have designed our parallel visualization solutions by closely following these guidelines.

3. Previous Work

The research problem we intended to solve has multiple facets ranging from large time-varying data, parallel I/O, parallel rendering, and unstructured grids, 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.

3.1. 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 of an exploration mechanism accompanied by an appropriate user interface for tracking and correct interpretation of the temporal aspects of the data. We have mainly looked into the I/O issues 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 [MM98] develop a visualization system capable of delivering realtime viewing of large time-varying ocean flow data by exploiting the high performance volume rendering of texture mapping hardware of four InfiniteReality pipes attached to an SGI Origin 2000 with enough memory to hold thousands of time steps of the data. The ParVox system [LWMT97] 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 [Ma03]. One very effective strategy is based on a hardware decoding technique such that data stay compressed until reaching the video memory for rendering [LMC02]. 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 high-speed network and parallel I/O support, a particularly promising strategy for achieving interactive visualization is to perform pipelined rendering. Ma and Camp [MC00] show that by properly grouping processors according to the rendering loads, compressing images before delivering, and completely overlapping uploading each time step of the data, rendering, and delivering the images, interframe delay can be kept to a minimum. Garcia and Shen [GS03] develop a dynamic load balancing strategy based on asynchronous communication for more efficiently rendering time-varying volume data on a PC cluster. Improved load balancing is achieved by cleverly and dynamically distributing the image compositing job.

3.2. 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, however, does not solve the problem of disk contention. Our previous system for rendering the earthquake simulation data [MSB*03] experimentally selected and used multiple I/O nodes to maximize bandwidth and reduce latency. The MPI I/O interface [GLT99] would be attractive but surprisingly very little use of MPI I/O has been found in parallel visualization applications. Parallel I/O support has also been made to those data file formats widely used by scientific applications such as HDF [HDF] and netCDF [LLC*03]. Our earthquake simulation data files are in neither HDF nor NetCDF so we have to develop new parallel I/O strategies through MPI I/O.

3.3. Parallel rendering

Our approach to the large data problem is to distribute both the data and visualization calculations to multiple proces-

sors 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 used thus must be highly efficient and scalable to a large number of processors because of the size of the dataset. Ma and Crockett [MC99] demonstrate a highly efficient, cell-projection volume rendering algorithm using up to 512 T3E processors for rendering 18 millions tetrahedral elements from an aerodynamic flow simulation. They achieve over 75% parallel efficiency by amortizing the communication cost as much as possible and using a fine-grain image space load partitioning strategy. Parker et al. [PPL*99] 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 very 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. [WPLM01] show how to achieve scalable rendering of large isosurfaces (7-469 million triangles) and a 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. [BTL*00] introduce a very unique remote and distributed visualization architecture as a promising solution to very large scale data visualization.

3.4. 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 rendering needs explicit connectivity information for each ray to march from one element to the next [Ma95]. The rendering algorithm introduced by Ma and Crockett [MC97] requires no connectivity information. Since each tetrahedral element is rendered completely independent of other elements, data distribution can be done in a more flexible manner facilitating load balancing. Chen, Fujishiro, and Nakajima [CFN02] present a hybrid parallel rendering algorithm for large-scale unstructured data visualization on SMP clusters such as the Hitachi SR8000. The three-level hybrid parallelization employed 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.

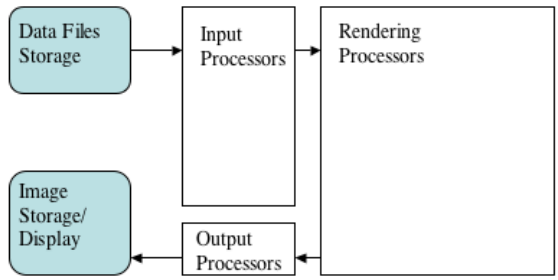

Figure 1: *The architecture of the parallel visualization solution.*

4. The Parallel Rendering Method

The basic architecture of our parallel visualization solution is shown in Figure 1. It is essentially a parallel pipeline and become the most efficient as soon as 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 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 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), central differencing to derive gradient vectors for lighting, and one-side differencing to derive rates of change for temporal domain enhancement. Lighting and temporal domain enhancement are optional. As we will show later, the amount of preprocessing calculations can influence the setting of an optimal system configuration for rendering. Note that it is more convenient and economical to conduct these preprocessing tasks at the input processors rather than the rendering processors. First, data replication is avoided because the input processors have access to all the needed data. Second, like 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 delivering. 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 for optimizing 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 space and the data space. 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 elements to a 512×512 pixels image, unless a close-up view is selected, rendering at the highest resolution level would not reveal more details. One of the calculations that the view-dependent preprocessing step performs is to choose the appropriate octree level. The saving from such an adaptive approach can be tremendous and there is virtually very little impact on the level of information presented in the resulting images as shown in Figure 2. Presently the appropriate level to use is computed based on the image resolution, data resolution, and a user-specified number that limits the number of elements allowed to be projected into a pixel.

4.2. Parallel image compositing

The parallel rendering algorithm is sort-last which thus requires a final compositing step involving inter-processor communication. Several parallel image compositing algorithms are available [MPHK94, LRN96, AP98] but their efficiency is mostly limited to the use of specific network topology or number of processors. We have adopted the SLIC algorithm [SML*03] 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 used in [Hsu93, Neu94, MC97] because 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 both a high speed network and parallel I/O support are 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. 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 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. 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.

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

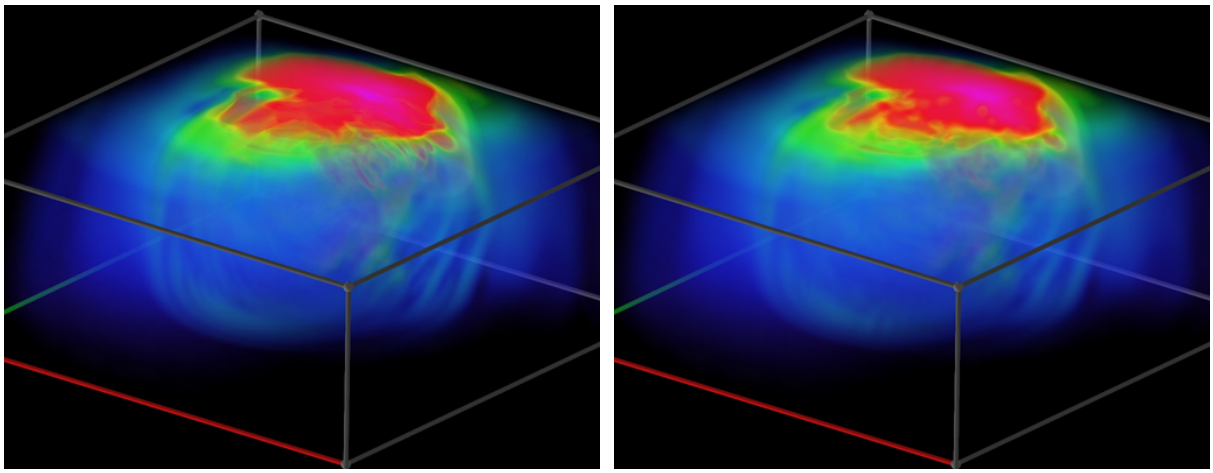

Figure 2: *Left: high-resolution rendering (level 11). Right: Adaptive rendering (level 6). The image on the right provides enough high level information about the data while it can be generated about 3 times faster than the high resolution one.*

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 3 contrasts 1DIP and 2DIP configurations.

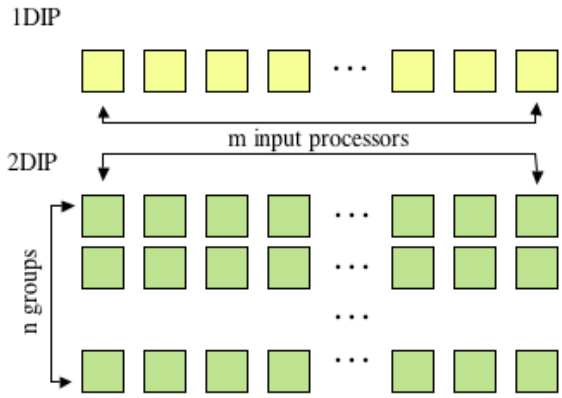

Figure 3: *The 1DIP and 2DIP configurations.*

5.3. File reading strategies

MPI-IO, the I/O part of the MPI-2 standard [GLT99], 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 balancing 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 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 4. 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.

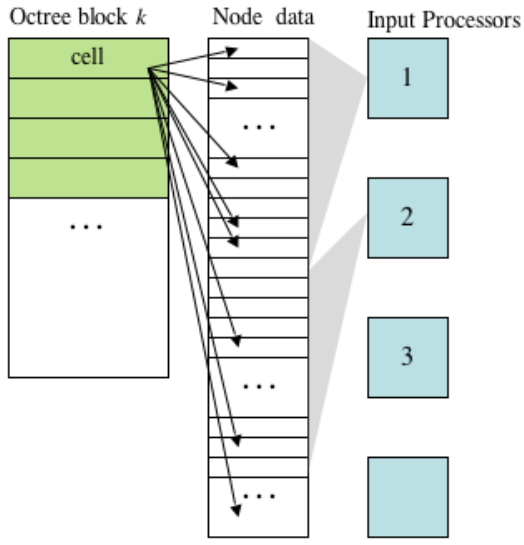

Figure 4: *Octree blocks are assigned to rendering processors according to a load balancing strategy. Using the second reading method, the node data belonging to the octree block k likely spread multiple input processors. There is a merging process at every rendering processor to gather all the relevant node data.*

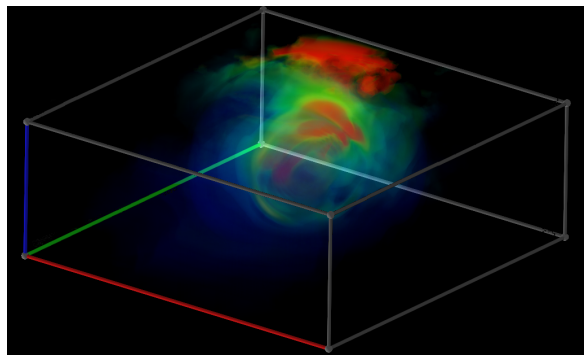

Figure 5: *Time step 100 of the velocity magnitude data.*

6. Test Results

We present the performance of our parallel I/O strategies on LeMieux, an HP/Compaq AlphaServer with 3,000 processors operated at the Pittsburgh Supercomputing System 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 5 displays the time step 100 of the velocity magnitude data.

The first set of tests was performed 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 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 load balancing and quantization. Figure 6 shows 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. Recall that $m = \frac{T_f+T_p}{T_r} + 1$. Using the actual values for T_f, T_p, and T_r, we obtain:

$$\frac{(12.32+9.06)}{2.0} + 1 = 11.69$$

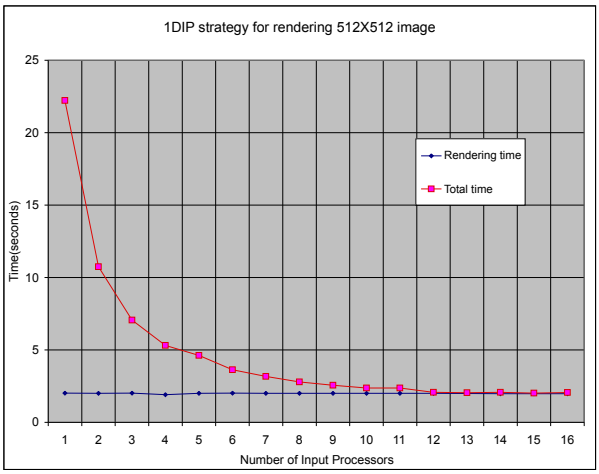

Figure 6: *64 rendering processors using the 1DIP strategy. Image resolution is 512times512. When 12 input processors, the total time due to I/O and preprocessing is reduced to about 2 seconds, very close to the rendering time.*

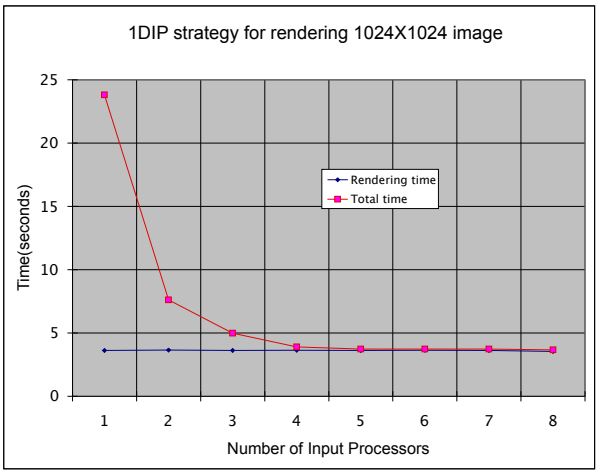

Figure 7: *64 rendering processors using the 1DIP strategy. Image resolution is 1024times1024. Only 7 input processors are needed to make the time due to I/O and preprocessing close to the rendering time.*

which matches Figure 6. If the image size is 1024×1024, since the rendering time would increase, the number of input processors needed would decrease. Our test results verify this as shown in Figure 7. Only 7 input processors are needed to make the total time similar to the rendering time which is 3.63 seconds. Compute m using our model, we obtain:

$$\frac{(12.32+9.06)}{3.63}+1=6.889$$

which matches.

The cost of preprocessing can vary significantly depending on the visualization requirements. In the first set of tests, we used a rather complex function to quantize the volume data. When we switched to a simple linear quantization method, the preprocessing cost was cut from 9.06 to 1.53 seconds. Substituting this new number into our model, we obtain 7.9, which is consistent with our test results for rendering 512×512 image as shown in Figure 8.

The second set of tests, we studied the 2DIP strategy. Recall that the purpose of using 1DIP 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 2DIP which concurrently reads multiple time steps. Figures 9 and 10 present our test results. Note that the total time can be reduced to under 10 seconds when using 8 or more input processors to read one time step of the data. (Recall in the 1DIP case, it takes more than 20 seconds to read and preprocess a single time step of the data.) More importantly, the sending time is reduced to under 1 second making possible displaying rates at multiple frames per second, as revealed in Figure 11 which only plots the sending time. Our test results

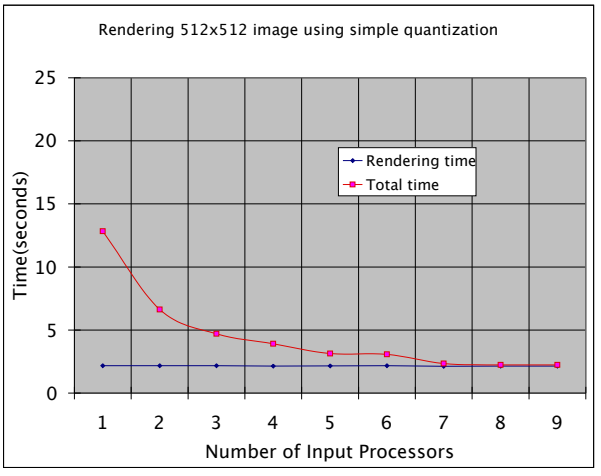

Figure 8: *Preprocessing time is significantly reduced because of using a simple linear quantization method instead. As a result, the number of input processors required decreases.*

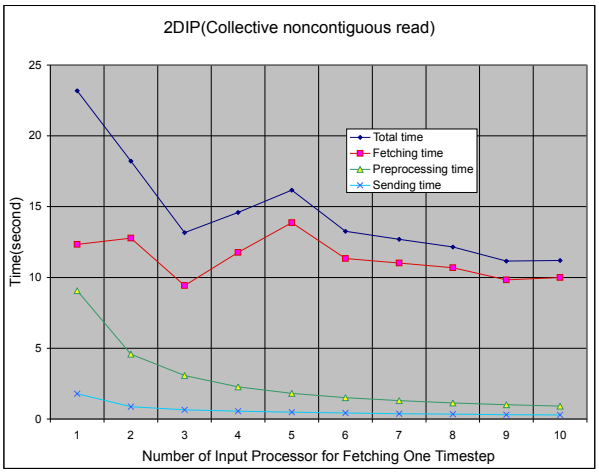

Figure 9: *Using 2DIP strategy with Collective Noncontiguous Read, sending time is reduced to under 1 second which makes interactive visualization possible.*

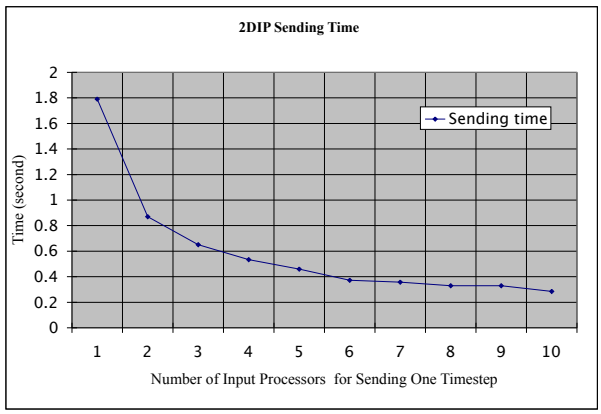

Figure 11: *The cost of sending one time step when using 2DIP decreases steadily as more Input processors are used. This plots indicates that it is possible to reach multiple frames per second rendering rates.*

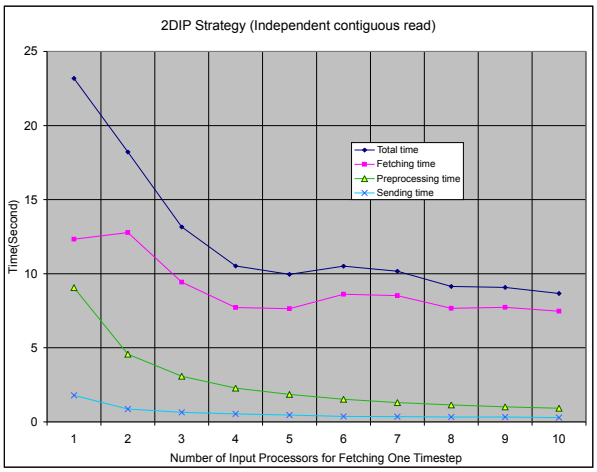

Figure 10: *Using 2DIP strategy with Independent Contiguous Read, not only the sending time is reduced to under 1 second, but the total time becomes under 10 seconds.*

also demonstrate that the Independent Contiguous Read is superior than the Collective Noncontiguous Read.

Finally, our further tests using 256 rendering processors show that the 1DIP strategy is less scalable. While the rendering time is reduced to under a second, using more than 14 input processors per group (i.e., m=14) with 2DIP, we can still completely hide the I/O and preprocessing cost, but not with IDIP. Adaptive rendering can significantly reduce both the rendering time and the amount of data that must be transferred from disk to the rendering processors. The effectiveness of the IDIP or 2DIP strategy stays the same regardless of using adaptive rendering or not.

7. Conclusions

We have experimentally studied new I/O strategies for the parallel visualization of large-scale earthquake simulations. This parallel visualization solution incorporates adaptive rendering, a highly efficiently parallel image compositing algorithm, and the new I/O strategies to make possible near-interactive visualization of large-scale time-varying data. Our performance study using up to 276 processors of LeMieux at the PSC demonstrates convincing results, and also reveals the interplay between data transport strategy used and interframe delay.

We have addressed the I/O problem of the massively parallel rendering. The next subject of study is load balancing. We plan to investigate a fine grain load redistribution method and study how to reduce its overhead as much as possible. We have demonstrated that using multiple data servers helps not only remove the I/O bottleneck but also hide preprocessing cost. Using input processing to also handle load balancing could be a promising solution.

Presently the image compositing cost is about constant. We believe compression can help lower communication cost to possibly make the overall compositing scalable to large machine size. 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 previously, the full rendering and adaptive rendering can result in visually indistinguishable results but the saving in rendering cost can be tremendous. Our study in this direction will focus on how adaptive rendering can be done with minimal user intervention and perception of level switching. Further optimization might be possible with adaptive fetching according to the selected level.

Even though our performance study was conducted with batch mode rendering the test results show that interactive visualization is possible. The subsequent task is to design appropriate user interface and interaction techniques for interactive browsing in both the spatial and temporal domains of the data. Scientists therefore can conduct interactive data exploration on their desktop. A buffering mechanism is likely needed for the user to conduct spatial domain exploration of a selected time step, which would defer the rendering of incoming time steps. To complicate the problem further, it could be desirable to create a single visualization by making use of multiple variables and/or multiple time steps [SLM02].

Finally, it is advantageous to run the parallel simulation and renderer simultaneously on either the same machine or two different machines connected with high-speed network interconnect such as the Quadrics network which has a bandwidth over 300 megabytes per second, permitting remote interaction with the simulation and visualization.

Acknowledgments

This work has been sponsored in part by the U.S. National Science Foundation under contracts ACI 9983641 (PECASE award), ACI 0325934 (ITR), ACI 0222991, and CMS-9980063; the U.S. Department of Energy under Memorandum Agreements No. DE-FC02-01ER41202 (SciDAC) and No. B523578 (ASCI VIEWS); the LANL/UC CARE program; and the National Institute of Health. Pittsburgh Supercomputing Center (PSC) provided time on their parallel computers through AAB grant BCS020001P. The authors would like to thank Rajeev Thakur for the discussion on MPI I/O, and Jacobo Bielak, Omar Ghattas, and Eui Joong Kim for providing the earthquake simulation datasets. Thanks especially go to Paul Nowoczynski and John Urbanic for their assistance on setting up the needed system support at PSC.

References

[AP98] AHRENS J., PAINTER J.: Efficient sort-last rendering using compression-based image compositing. In *Proceedings of the 2nd Eurographics Workshop on Parallel Graphics and Visualization* (1998), pp. 145–151.

[BBG*98] BAO H., BIELAK J., GHATTAS O., KALLIVOKAS L. F., O'HALLARON D. R., SHEWCHUK J. R., XU J.: Large-scale simulation of elastic wave propagation in heterogeneous media on parallel computers. *Computer Methods in Applied Mechanics and Engineering 152*, 1–2 (Jan. 1998), 85–102.

[BTL*00] BETHEL W., TIERNEY B., LEE J., GUNTER D., LAU S.: Using high-speed WANs and network data caches to enable remote and distributed visualization. In *Proceedings of Supercomputing 2C00* (November 2000).

[CFN02] CHEN L., FUJISHIRO I., NAKAJIMA K.: 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* (2002), pp. 133–140.

[GLT99] GROPP W., LUSK E., THAKUR R.: *Using MPI-2: Advanced Features of the Message Passing Interface*. The MIT Press, 1999.

[GS03] GARCIA A., SHEN H.-W.: Asynchronous rendering for time-varying volume datasets on PC clusters. In *Proceedings of the IEEE Visualization 2003 Conference (to appear)* (October 2003).

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

[Hsu93] HSU W. M.: Segmented ray casting for data parallel volume rendering. In *Proceedings of 1993 Parallel Rendering Symposium* (1993), pp. 7–14.

[LLC*03] LI J., LIAO W.-K., CHOUDHARY A., ROSS R., THAKUR R., GROPP W., LATHAM R., SIEGEL A., GALLAGHER B., ZINGALE M.: Parallel netCDF: A high-performance scientific I/O interface. In *Proceedings of Supercomputing 2003 Conference* (November 2003).

[LMC02] LUM E., MA K.-L., CLYNE J.: A hardware-assisted scalable solution for interactive volume rendering of time-varying data. *IEEE Transactions on Visualization and Computer Graphics 8*, 3 (2002), 286–301.

[LRN96] LEE T.-Y., RAGHAVENDRA C. S., NICHOLAS J. B.: Image composition schemes for sort-last polygon rendering on 2d mesh multicomputers. *IEEE Transactions on Visualization and Computer Graphics 2*, 3 (1996), 202–217.

[LWMT97] LI P., WHITMAN S., MENDOZA R., TSIAO J.: ParVox – a parallel spaltting volume rendering system for distributed visualization. In *Pro-*

ceedings of 1997 Symposium on Parallel Rendering (1997), pp. 7–14.

[Ma95] MA K.-L.: Parallel volume ray-casting for unstructured-grid data on distributed-memory architectures. In *Proceedings of the Parallel Rendering '95 Symposium* (1995), pp. 23–30. Atlanta, Georgia, October 30-31.

[Ma03] MA K.-L.: Visualizing time-varying volume data. *IEEE Computing in Science & Engineering 5*, 2 (2003), 34–42.

[MC97] MA K.-L., CROCKETT T.: A scalable parallel cell-projection volume rendering algorithm for three-dimensional unstructured data. In *Proceedings of 1997 Symposium on Parallel Rendering* (1997), pp. 95–104.

[MC99] MA K.-L., CROCKETT T.: Parallel visualization of large-scale aerodynamics calculations: A case study on the Cray T3E. In *Proceedings of 1999 IEEE Parallel Visualization and Graphics Symposium* (1999), pp. 15–20.

[MC00] MA K.-L., CAMP D.: High performance visualization of time-varying volume data over a wide-area network. In *Proceedings of Supercomputing 2000 Conference* (November 2000).

[MM98] MCPHERSON A., MALTRUD M.: POPTEX: Interactive ocean model visualization using texture mapping hardware. In *Proceedings of the Visualization '98 Conference* (October 18-23 1998), pp. 471–474.

[MPHK94] MA K.-L., PAINTER J. S., HANSEN C., KROGH M.: Parallel Volume Rendering Using Binary-Swap Compositing. *IEEE Computer Graphics Applications 14*, 4 (July 1994), 59–67.

[MSB*03] MA K.-L., STOMPEL A., BIELAK J., GHATTAS O., KIM E.: Visualizing large-scale earthquake simulations. In *Proceedings of the Supercomputing 2003 Conference* (2003).

[Neu94] NEUMANN U.: Communication costs for parallel volume-rendering algorithms. *IEEE Computer Graphics and Applications 14*, 4 (July 1994), 49–58.

[PPL*99] PARKER S., PARKER M., LIVNAT Y., SLOAN P., HANSEN C.: Interactive Ray Tracing for Volume Visualization. *IEEE Transactions on Visualization and Computer Graphics 5*, 3 (July-September 1999), 1–13.

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

[SLM02] STOMPEL A., LUM E., MA K.-L.: Visualization of multidimensional, multivariate volume data using hardware-accelerated nonphotorealistic rendering techniques. In *Proceedings of Pacific Graphics 2002 Conference* (2002), pp. 394–402.

[SML*03] STOMPEL A., MA K.-L., LUM E., AHRENS J., PATCHETT J.: SLIC: scheduled linear image compositing for parallel volume rendering. In *Proceedings of IEEE Sympoisum on Parallel and Large-Data Visualization and Graphics* (October 2003), pp. 33–40.

[TOL02] TU T., O'HALLARON D., LOPEZ J.: Etree: A database-oriented method for generating large octree meshes. In *Proceedings of the Eleventh International Meshing Roundtable* (September 2002), pp. 127–138.

[WPLM01] WYLIE B., PAVLAKOS C., LEWIS V., MORELAND K.: Scalable rendering on PC clusters. *IEEE Computer Graphics and Applications 21*, 4 (July/August 2001), 62–70.