A Study of I/O Techniques for Parallel Visualization

Hongfeng Yu Kwan-Liu Ma

University of California at Davis

Abstract

This paper presents two parallel I/O methods for the visualization of time-varying volume data in a high-performance computing environment. We discuss the interplay between the parallel renderer, I/O strategy, and file system, and show the results of our study on the performance of the I/O strategies with and without 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 methods effectively remove the I/O bottlenecks commonly present in time-varying data visualization, and therefore help significantly lower interframe delay. This high-performance visualization solution allows scientists to explore their data in the temporal, spatial, and visualization domains at high resolution. Such new explorability, likely not presently available to most computational science groups, will help lead to many new insights.

Key words: high-performance computing, massively parallel supercomputing, MPI, scientific visualization, parallel I/O, parallel rendering, time-varying data, volume rendering

1 Introduction

Parallel supercomputers allow scientists to model complex physical phenomena and chemical processes with a high degree of precision and sophistication. However, a complete run of such a high-resolution time-varying simulation can easily output hundreds of gigabytes to terabytes of data, which present tremendous challenges to the management and analysis of the data. Frequently, the data sets are either not fully explored or downsampled spatially or temporally to fit in the limited storage space, which defeat the original purpose of perform the high-resolution simulations.

Visualization transforms large amounts of data into vivid images revealing important aspects of the data, which often leads to profound levels of insight and understanding. Visualization becomes the most powerful when the scientists are allowed

to interactively explore the data. Interactive visualization is not generally available to scientists because rendering the highest-resolution data requires the most powerful parallel supercomputers.

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. Many parallel rendering algorithms have been introduced and shown to achieve high parallel efficiency, but the designs of most of these algorithms were done by considering rendering in isolation. In particular, I/O was mostly neglected. As a result, these renderers are either used in a limited way or totally not deployable for routine data visualization tasks.

In this paper, we discuss I/O support for parallel visualization, and present an experimental study of two parallel I/O strategies (1) designed specifically for rendering time-varying volume data in a typical high-performance computing environment. The volume data sets we used in our study were generated by the highest-resolution earthquake simulation performed to date (2). The I/O strategies 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 framerate can be obtained. All our tests were performed on the HP/Compaq AlphaServer operated at the Pittsburgh Supercomputing Center.

2 Parallel I/O

The widespread use of large-scale multiprocessor systems in scientific computing demands the advancement of parallel I/O technologies. MPI-IO was introduced to provide a common parallel I/O interface but its implementations did not consistently deliver desirable performance. Consequently, nowadays most parallel applications still use simple parallel file systems whose performance are largely determined by their hardware configurations and the network interconnect.

File I/O bottleneck mostly comes from mismatching between the data access patterns and the actual data storage order on disk. The situation becomes the worst when multiple processors make many small noncontiguous accesses to the disk. In this section, we review some of parallel I/O concepts and techniques introduced to address such a problem.

2.1 Collective I/O

Collective I/O is a technique with which processors cooperate to more efficiently fetch data from disks. The basic idea is to merge small, noncontiguous disk accesses into large, contiguous disk accesses by multiple processors. After each processor fetches a chunk of the data from disk, through the interconnection network data items are routed to the processor requests them. Such a two-phase I/O approach was first introduced by Del Rosario et al. (3), and then extended by Rajeev Thakur et al. in ROMIO (4).

That collective I/O helps is based on the assumption that the cost of unoptimized (numerous noncontiguous) I/O operations is greater than the combined cost of aggregating I/O operations and extra network communication. In practice, interconnection network speed is several orders of magnitude higher than the disk access speed so collective I/O can help improve I/O performance.

David Kotz introduced disk-directed I/O (5) which moves the I/O management job from the client program running on the compute processors to the disk server (i.e., I/O processors which the disks directly attached to). For collective I/O, the compute processors broadcast their data need to all the I/O processors. Then each I/O processor optimizes disk access according data residency. Each I/O processor uses double-buffering for transferring data between the disk and compute processors. Disk-directed I/O requires no communication among I/O processors nor among compute processors. Unlike the client-level, two-phase I/O, it also does not require an explicit data permutation phase. The performance of disk-direct I/O has been demonstrated to attain 93interface for conducting disk-directed I/O at logical file level, rather than the physical disk level (6). The I/O strategies we have used resemble disk-directed I/O.

2.2 Prefetching

Prefetching is a technique to hide the cost of reading data from disk. After reading but before starting to process the current data block, the compute processor issues an asynchronous read request to fetch the next data block. In this way, the computation time of the current block can overlap with the I/O of the next block. This technique suits for applications whose access patterns can be predicted; that is, the I/O access patterns exhibit certain regularity. If the computation time is comparable to the I/O time, prefetching can result in significant performance improvement. Prefetching has been used in various systems (7; 8).

2.3 Data Sieving

The objective of data sieving is similar to that of Collective I/O, which is to replace many small nonconsecutive reads with fewer large consecutive reads. With data sieving, however, the fetching of several noncontiguous pieces of data is replaced by a single read of a large contiguous chunk of the data containing all those individual pieces. As a result, more data is read than actually needed. A buffer is used to hold the fetched data in memory before the pieces are extracted from it and distributed to the compute processors. In most cases, the benefit of reading large, contiguous chunks of data far outweighs the cost of reading unwanted data (9), but the memory overhead to store the unneeded data for data sieving can become excessive. ROMIO (4) provides a user-controllable parameter to define the maximum amount of contiguous data that a process can read at a time with data sieving.

2.4 MPI-IO

MPI-IO (10) provides both portability and convenience to perform parallel file I/O. It defines a set of routines for transferring data to and from external storage devices, and supports a number of useful parallel I/O features including collective I/O. It is possible to achieve good performance with MPI I/O by following a set of guidelines. In this paper, we also present our experimental study on MPI I/O using the parallel visualization application.

3 A Parallel Visualization Problem

One of the large-scale scientific simulations motiving our work is earthquake modeling. Simulating the earthquake response of a large basin is accomplished by numerically solving the partial differential equations (PDEs) of elastic wave propagation (11). 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 (12). 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 ve-

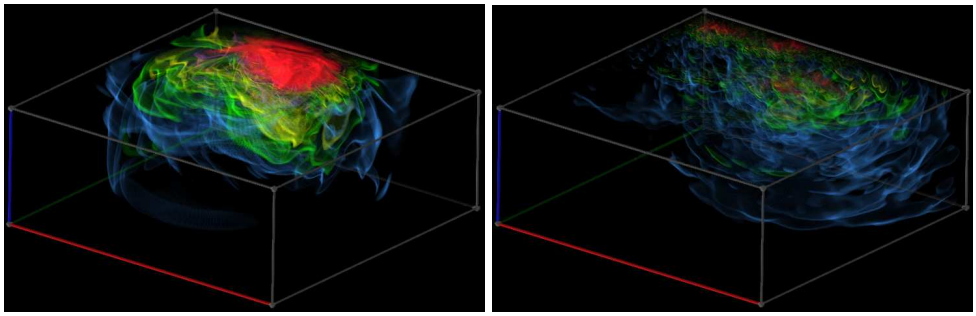


Fig. 1. Two selected time steps from the earthquake simulation.

locity 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. Figure 1 displays two selected time steps from the simulation.

Several strategies are commonly used to achieve high performance rendering of large time-varying volume datasets in a parallel computing environment. While this paper focuses on I/O issues, we first describe the parallel visualization method we have chosen to use and the corresponding I/O requirements.

3.1 Parallel and distributed rendering

Our approach to this 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 used thus must be highly efficient and scalable to a large number of processors because of the size of the dataset. Ma and Crockett (13) 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. (14) 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. (15) 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. (16) introduce a remote and distributed visualization architecture as a promising solution to very large scale data visualization.

3.2 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 (17). The rendering algorithm introduced by Ma and Crockett (18) 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 (19) 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.

Our visualization solution couples the parallel rendering algorithm of Ma and Crokett (18) with our new I/O strategies to form a highly efficient parallel visualization pipeline for near-interactive browsing of large time-varying volume data.

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 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

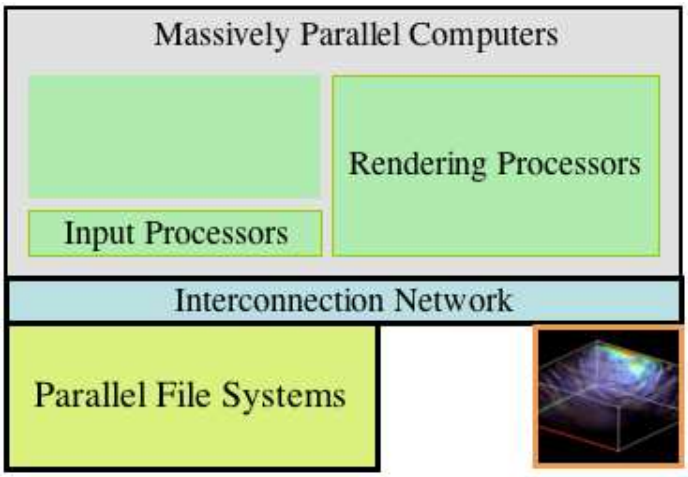


Fig. 2. The architecture of the parallel visualization solution.

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.

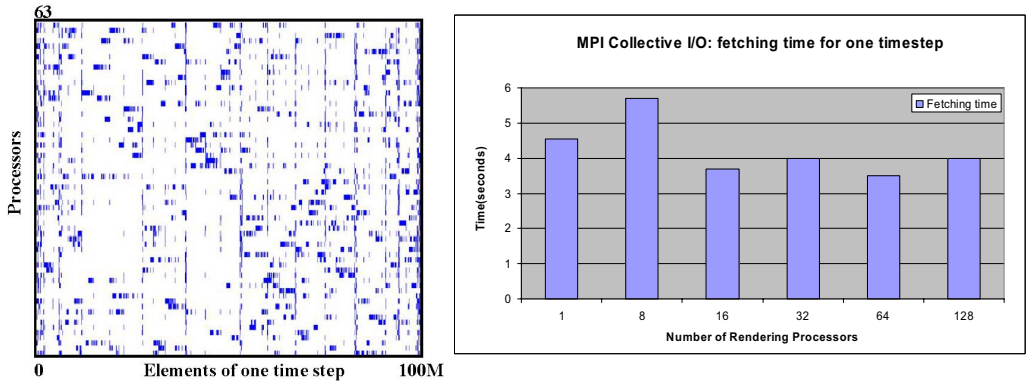
4.1 Adaptive rendering

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. 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 (20; 21; 22) but their efficiency is mostly limited to the use of specific network topology or number of processors. We have adopted the SLIC algorithm (23) 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 (24; 25; 18) 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



(a) (b)

Fig. 3. (a) Data distribution pattern;(b) using MPI Collective I/O to read and distribute one time step among rendering processors.

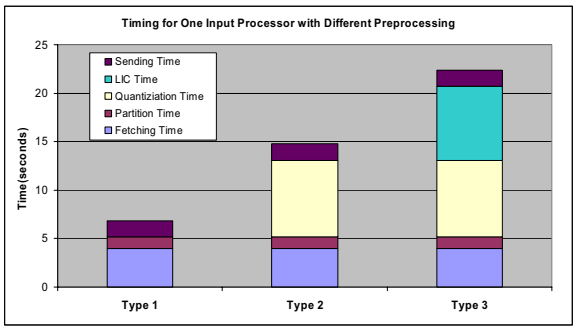
improve parallel efficiency.

5 Parallel I/O Strategies and Test Results

We have developed and experimentally studied two 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 Pittsburgh Supercomputing Center adopts Quadrics PFS. There are 64 file servers with 300MB per second data maximum data transfer rate each piece, and the maximum throughput is about 18GB per second. Currently the 64 file server nodes are divided into 4 independent filesystems, each containing 16 nodes. By default a file is striped across 8 nodes. Quadrics Elan3 (dual-rail) is the high-speed network connecting the file systems. Each rail is capable of around 250MB per second. The message latency is under 10 $/mu$secs. We have studied how to effectively utilize these high-performance computing resources for the visualization of time-varying ground motion simulation data consisting of 100 million hexahedral elements. Each time step of the data to be transfer is about 400 megabytes.

Figure 3(a) shows the assignment of the 100 million data elements among 64 processors, which is rather random suggesting the data access pattern must be very irregular. We first tested the performance of MPI-IO collective I/O for fetching one time. Figure 3 (b) demonstrates that the fetching time for one time step is not scalable with the number of the rendering processors. As a result, even though the rendering time can be reduced by using more rendering processors, it is impossible to achieve multiple frames per second rendering rates due to the high read cost with MPI-IO.

Our designs use parallel pipelining. In addition to employing multiple rendering



Three preprocessing types

	Partition	Quantization	LIC
Type 1	√		
Type 2	√	√	
Type 3	√	√	√

Fig. 4. Using one input processor, the fetching time is 4 seconds, and sending times is 1.7 seconds. The preprocessing times vary: data partition takes 1.15 seconds; quantization requires 7.91 seconds; and LIC rendering takes 7.6 seconds for 512×512 image.

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, t he 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.

5.1.1 Test results

We first measured how different preprocessing tasks impact the overall performance when a single input processor is used. The timing results also allow us to predict the performance of using more input processors, and to select the optimal input processor number. Figure 4 presents the test results for three different types of preprocessing tasks: data partition, quantization calculations, and LIC rendering. As more preprocessing tasks are added, the time for transferring and preparing the data increases from 7 to 23 seconds.

The second set of tests was performed using 64 rendering processors with different rendering and preprocessing requirements. In the first case, the image size is 512×512. The rendering is about 2.13 seconds, and the total time due to I/O and preprocessing is about 15 seconds if only a single input processor is used. Preprocessing cost includes the time to do data partition and quantization. Figure 5 (a) shows when using 8 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:

$$\frac{(4.0+9.06)}{2.13} + 1 = 7.13$$

which matches Figure 5 (a).

If the image size is 1024×1024, since the rendering time would increase, the number of input processors needed would decrease. Out test results verify this as shown in Figure 5 (b). Only 5 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:

$$\frac{(4.0+9.06)}{3.63} + 1 = 4.60$$

which matches the test results.

Furthermore, if adding lighting effect, it requires calculations of gradient information to approximate local surface orientation plus solving the lighting equation at each sample point. Using 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. A much smaller number of input processors are needed because of the higher cost of rendering. Substituting the new rendering time 8.4 seconds into our model, we obtain 2.55, which is consistent with our test result for rendering 512×512 image as shown in Figure 5 (c).

Finally, we tested 2D LIC images which can be done as a preprocessing step and handled by the Input processors. Figure 5 (d) shows the cost of making simultaneous surface LIC and volume visualization using 64 rendering processors with 1DIP strategy. When 11 input processors are used, computing the LIC images, other preprocessing, and I/O essentially become free. Given the new preprocessing time, using our model, we obtain:

$$\frac{(4.0+16.7)}{2.13} + 1 = 10.72$$

which is consistent with our test results.

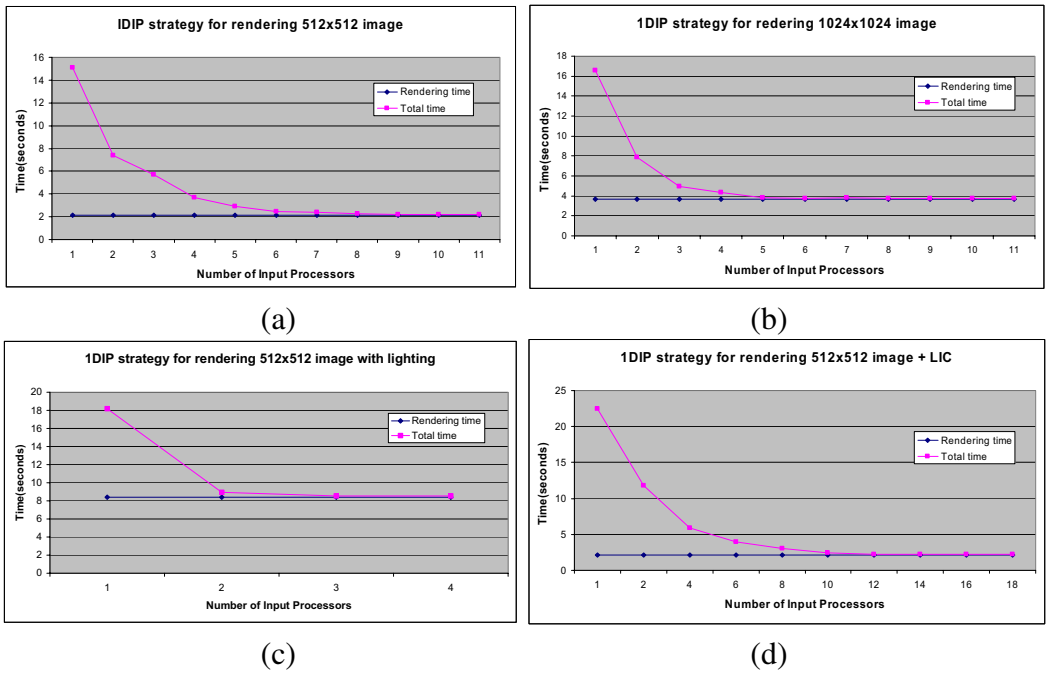


Fig. 5. Using 64 rendering processors with 1DIP strategy. Our 1DIP model was tested under four visualization scenarios with different rendering and preprocessing requirements. To hide the I/O and preprocessing cost, the input processor numbers obtained by our model are consistent with the test results.

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

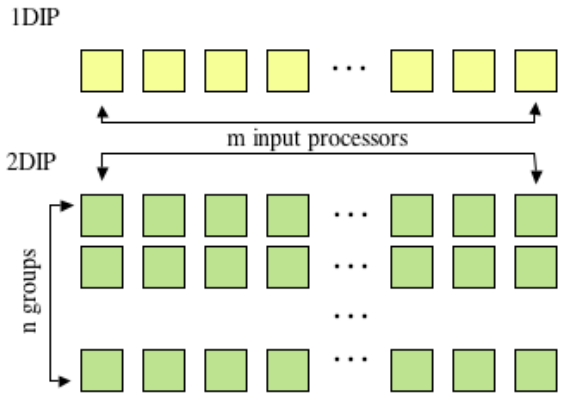


Fig. 6. The 1DIP and 2DIP configurations.

2DIP configurations.

5.2.1 File reading strategies

MPI-IO, the I/O part of the MPI-2 standard (10), 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.

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

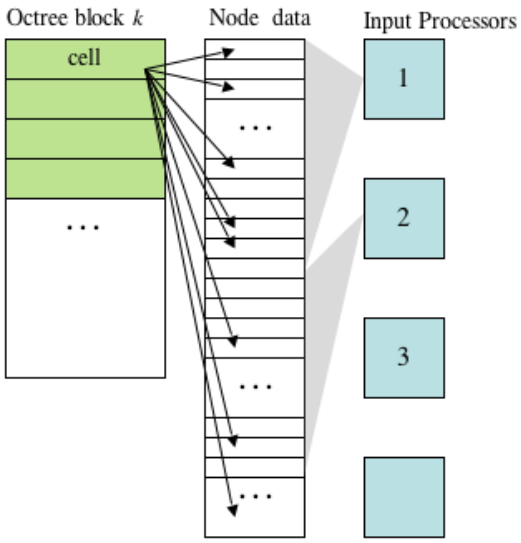


Fig. 7. Octree blocks are assigned to rendering processors according to a load balancing strategy. Using the second reading method, the node data belonging to the octree block k likely spread multiple input processors. There is a merging process at every rendering processor to gather all the relevant node data.

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.

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

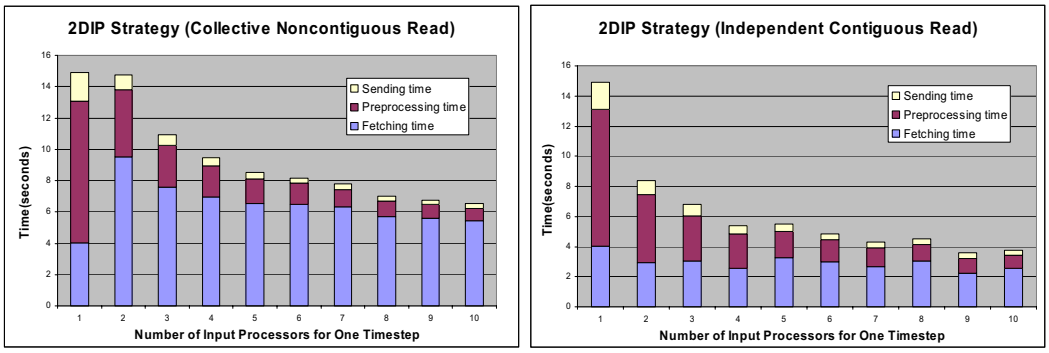


Fig. 8. Using 2DIP strategy Left: with Collective Noncontiguous Read, sending time is reduced to under 1 second which makes interactive visualization possible. Right: with Independent Contiguous Read, not only the sending time is reduced to under 1 second, but the total time becomes under 4 seconds.

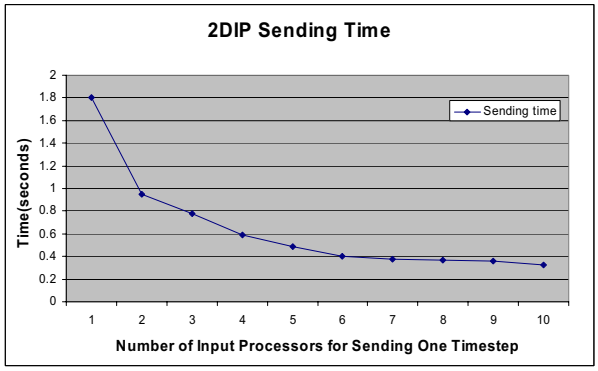


Fig. 9. The cost of sending one time step when using 2DIP decreases steadily as more Input processors are used. This plots indicates that it is possible to reach multiple frames per second rendering rates.

strategy is superior if the overhead of collective I/O would become too high.

5.2.2 Test results

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. Figure 8 presents our test results. Note that using Independent Contiguous Read, the total time can be reduced to under 4 seconds when using 9 or more input processors to read one time step of the data. (Recall in the 1DIP case, it takes around 15 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 9 which plots the sending time. Our test results also demonstrate that the Independent Contiguous Read is superior than the Collective Noncontiguous Read.

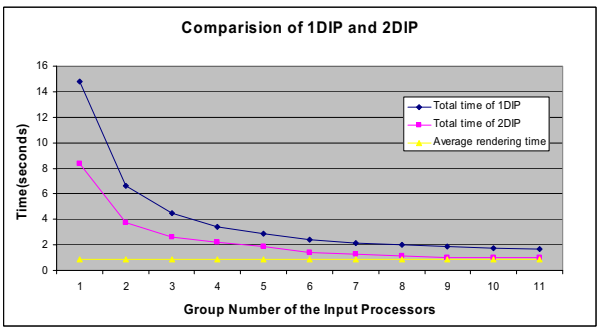


Fig. 10. Comparing 1DIP and 2DIP using 128 rendering processors for rendering 512×512 images. Preprocessing includes data partition and quantization. The rendering time is reduced to about 1 second. In this case, overlapping rendering and I/O is only possible with 2DIP.

Finally, Figure 10 compares 1DIP and 2DIP. When the rendering time is reduced to under a second, using more than 2 input processors per group (i.e., m=2) 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. All tests were done using Type 2 preprecessing.

6 Conclusion

Our work has been driven by large-scale scientific applications such as earthquake modeling, supernova modeling, ocean modeling, and turbulence modeling. While this paper presents our experimental study of parallel I/O strategies for rendering time-varying data from an earthquake simulation, the visualization requirements and challenges are common to all applications.

Our 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 shown that using dedicated input processors helps not only remove the I/O bottleneck but also hide preprocessing cost. We believe it is also possible to use input processors to achieve dynamic load balancing. In addition, adaptive rendering will continue to play a major role in our subsequent work. Further I/O optimization might be possible with adaptive fetching according to the selected level.

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, John Urbanic, and Joel Welling for their assistance on setting up the needed system support at PSC.

References

[1] H. Yu, K.-L. Ma, J. Welling, I/O strategies for parallel rendering of large time-varying volume data, in: Eurographics/ACM SIGGRAPH Symposiumm Proceedings of Parallel Graphics and Visualization 2004, 2004, pp. 31–40.

[2] K.-L. Ma, A. Stompel, J. Bielak, O. Ghattas, E. Kim, Visualizing large-scale earthquake simulations, in: Proceedings of the Supercomputing 2003 Conference, 2003.

[3] J. M. del Rosario, R. Bordawekar, A. Choudhary, Improved parallel I/O via a two-phase run-time access strategy, ACM SIGARCH Computer Architecture News 21 (5) (1993) 31–38.

[4] R. Thakur, W. Gropp, E. Lusk, Data sieving and collective I/O in ROMIO, in: Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation, 1999.

[5] D. Kotz, Disk-directed I/O for MIMD multiprocessors, ACM Transactions on Computer Systems 15 (1) (1997) 41–74.

[6] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, M. Winslett, Server-directed collective I/O in Panda, in: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM), ACM Press, 1995.

[7] M. Arunachalam, A. Choudhary, A prefetching prototype for the parallel file systems on the Paragon, in: Proceedings of the 1995 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems, ACM Press, 1995, pp. 321–322.

[8] R. Thakur, R. Boardawekar, A. Choudhary, R. Ponnusamy, T. Singh, PASSION runtime library for parallel I/O, in: Proceedings of the Scalable Parallel Libraries Conference, IEEE Computer Society Press, 1994, pp. 119–128.

[9] R. Thakur, W. Gropp, E. Lusk, A case for using MPI's derived datatypes to

improve I/O performance, in: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM), IEEE Computer Society, 1998, pp. 1–10.

[10] W. Gropp, E. Lusk, R. Thakur, Using MPI-2: Advanced Features of the Message Passing Interface, The MIT Press, 1999.

[11] H. Bao, J. Bielak, O. Ghattas, L. F. Kallivokas, D. R. O'Hallaron, J. R. Shewchuk, 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) (1998) 85–102.

[12] T. Tu, D. O'Hallaron, J. Lopez, Etree: A database-oriented method for generating large octree meshes, in: Proceedings of the Eleventh International Meshing Roundtable, 2002, pp. 127–138.

[13] K.-L. Ma, 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, 1999, pp. 15–20.

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

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

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

[17] K.-L. Ma, 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.

[18] K.-L. Ma, T. Crockett, 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.

[19] L. Chen, I. Fujishiro, 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, 2002, pp. 133–140.

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

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

[22] J. Ahrens, J. Painter, 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.

[23] A. Stompel, K.-L. Ma, E. Lum, J. Ahrens, J. Patchett, SLIC: scheduled linear image compositing for parallel volume rendering, in: Proceedings of IEEE Sympoisum on Parallel and Large-Data Visualization and Graphics, 2003, pp. 33–40.

[24] W. M. Hsu, Segmented ray casting for data parallel volume rendering, in: Proceedings of 1993 Parallel Rendering Symposium, 1993, pp. 7–14.
[25] U. Neumann, Communication costs for parallel volume-rendering algorithms, IEEE Computer Graphics and Applications 14 (4) (1994) 49–58.