

Distributed Data Management for Large Volume Visualization

Jinzhao Gao*
Oak Ridge National Lab

Jian Huang†
The Univ. of Tennessee

C. Ryan Johnson‡
The Univ. of Tennessee

Scott Atchley§
The Univ. of Tennessee

James Arthur Kohl¶
Oak Ridge National Lab

ABSTRACT

We propose a distributed data management scheme for large data visualization that emphasizes efficient data sharing and access. To minimize data access time and support users with a variety of local computing capabilities, we introduce an adaptive data selection method based on an “*Enhanced Time-Space Partitioning*” (ETSP) tree that assists with effective visibility culling, as well as multiresolution data selection. By traversing the tree, our data management algorithm can quickly identify the visible regions of data, and, for each region, adaptively choose the lowest resolution satisfying user-specified error tolerances. Only necessary data elements are accessed and sent to the visualization pipeline. To further address the issue of sharing large-scale data among geographically distributed collaborative teams, we have designed an infrastructure for integrating our data management technique with a distributed data storage system provided by *Logistical Networking* (LoN). Data sets at different resolutions are generated and uploaded to LoN for wide-area access. We describe a parallel volume rendering system that verifies the effectiveness of our data storage, selection and access scheme.

CR Categories: I.3.2 [Graphics Systems]: Distributed/network graphics—; I.3.6 [Methodology and Techniques]: Graphics data structures and data types—;

Keywords: large data visualization, distributed storage, logistical networking, visibility culling, volume rendering, multiresolution rendering

1 INTRODUCTION

Large-scale computer/software simulations are increasingly prevalent in many fields of science. Such simulations can produce hundreds of gigabytes to terabytes, including multiple variables on fine spatial and temporal scales. For computational scientists, visualization can be a crucial tool to validate scientific models and seek novel domain-specific understanding. The increasingly collaborative nature of scientific research across geographically distributed groups creates additional demands on visualization technology. In distributed settings, high performance and scalability must be maintained while preserving high throughput and ease of use, particularly for very large data sets.

The sheer size of simulation data is often a significant technical hurdle for distributed teams. The volume of data precludes storing multiple copies at each site, especially given the typical frugality of most local resource management. The common alternative is to maintain a centralized storage archive, yet this requires use of

some efficient data management middleware to dynamically access and stream data to the end users. Such a system must be careful to deliver a minimally necessary subset of any given data set to any user requesting an interactive visualization session. Further, the specific data management schemes will likely need to be different for each front-end visualization algorithm.

This paper presents such a prototype data management middleware system, known as the *Distributed Data Management System* (DDMS). This system is applied for volume rendering, being one of the more general visualization approaches, and also a significantly rigorous case study for validation of our design. The DDMS prototype is based on the assumption that entire data sets are held en masse on centralized storage devices, which we generally refer to here as “*depots*” (implying a single disk or some integrated networked file system). The main data structure of our DDMS is an *Enhanced Time-Space Partitioning* (ETSP) tree that provides a novel capability to efficiently discover, at run-time, all *non-transparent, un-occluded* (visible) portions of a given volume, that are at the right *level of detail* (according to the current viewing frustum and viewport resolution, in conjunction with the display capability of end user’s computer). This middleware applies the ETSP tree under a family of transfer functions, and by exploiting various data coherence characteristics, minimizes the amount of data movement using a conservative (optimistic) scheme.

To optimize scalability, in terms of both performance and the number of simultaneous users, our DDMS has been designed to operate in a distributed system, specifically, employing networked computers running *Internet Backplane Protocol* (IBP) [3, 20] for data depots. Each data set is uploaded, possibly with replicas, to IBP depots, which collectively act as an open distributed wide-area file system. Any user can access the depot data without reservation, scheduling or authentication [3]. The collective storage capacity of all IBP depots currently exceeds 31 terabytes, with over 331 locally maintained depots across the US and 22 other countries. The IBP infrastructure provides sufficient performance and flexibility for data uploading and downloading. Our DDMS automatically stripes data across multiple IBP depots; this redundancy not only facilitates reliability but also enables strategic data placement on depots convenient to each participating site. Using DDMS, users at different locations can easily access and visualize only relevant subsets of a large time-varying data set, according to a specified transfer function.

The rest of the paper is organized as follows. In Section 2, we describe previous work on data management and distributed visualization. Our DDMS data management middleware, and its extension for deployment on distributed systems, are described in Sections 3 and 4, respectively. We present experimental results of a parallel volume rendering system using our data management schemes in Section 5, and summarize our contribution and future directions in Section 6.

2 BACKGROUND

Several previous projects have designed specialized data structures for data management, particularly for time-varying data sets. One of the most widely known is the Time-Space Partitioning (TSP) tree

*gaoj@ornl.gov

†huangj@cs.utk.edu

‡cjohnson@cs.utk.edu

§atchley@cs.utk.edu

¶kohlja@ornl.gov

proposed by Shen *et al.* [21]. The skeleton of a TSP tree is a complete octree, or a Space-Partitioning Tree (SPT), which is built by recursively subdividing the spatial decomposition of the data volume, with each octree node covering a certain solid subvolume in space. To capture temporal coherence of each octree node, a binary tree, or a Time-Partitioning Tree (TPT), is built by recursively bisecting the time span. At each time partitioning tree node, the mean value as well as the spatial and the temporal errors are stored.

To accelerate isosurface extraction in time-varying data sets, Sutton and Hansen [22] developed a temporal Branch-on-Need tree. Recently, Wang and Shen [23] introduced a Wavelet-based Time-Space Partitioning (WTSP) tree that uses data compression to support multiresolution rendering of a time-varying data. It is now generally agreed that more coherence can be exploited by performing a spatial partition before a temporal partition, as done in these pioneering methods. Usually, these methods measure coherence by variances of the voxel values in a block. However, in volume rendering, a large variance does not directly correlate to a significant difference in the resulting color or opacity of an octree node. Our DDMS systematically extends TSP trees by incorporating more coherence metrics, such as transfer function dependent opacity and visibility. In this paper, we refer to “opacity” as a transfer function specific metric that solely depends on information local to a voxel, while “visibility” of a voxel depends on the opacity of the collective set of voxels or volume blocks between this voxel and the viewer.

Previously researchers have studied how to provide general remote visualization capabilities. Typically, a remote visualization system consists of a chain of servers, at least one client and possibly some intermediate computing nodes. The data set exists entirely on the server, with graphics primitives being streamed across the wide-area network towards the client node. Caching and prefetching are common in remote visualization systems, on both client or intermediate nodes, to hide communication costs. The fundamental problem is to design algorithms that transmit minimal amounts of data over the network, with the network latency hidden to users on the client side as much as possible. Most remote visualization algorithms rely on transforming raw data to a more compact representation, possibly offering a good approximation to the visual appearance of the raw data representation [14, 4]; others employ compression-based methods [13, 12, 9, 15]. Our DDMS employs neither data transformation nor compression, and focuses on using the capacity of the wide-area network and general distributed computing architectures.

Using the wide-area network, for example, Bethel *et al.* [4] leveraged a network protocol offering higher performance than primitive TCP. Engel *et al.* [7, 8] demonstrated a remote visualization infrastructure based on the Common Object Request Broker Architecture (CORBA). Recently, Brodlie *et al.* [5] developed a Grid-based problem solving environment, complete with modules for visualization, analysis and computational steering. Most of these systems require specialized reservation and scheduling to obtain optimal performance, thereby limiting the number of simultaneous users. An infrastructure for distributed computing that does not require such mechanisms is the Logistical Networking infrastructure. As evidenced by [6], gigabyte-level data sets can be efficiently streamed across a wide-area network. With some caching and buffering, Logistical Networking can be applied as a distributed file system, providing users with a perceived latency comparable to the access from the local network. Our DDMS is independent of any specific distributed data communication infrastructure, and can be integrated with a number of systems, including GridFTP and the middleware frameworks mentioned above. For this case study, our DDMS prototype system was integrated with Logistical Networking, due to its efficiency, ease-of-use and inherent scalability to support large numbers of simultaneous users. Section 4 provides a more detailed discussion of this infrastructure.

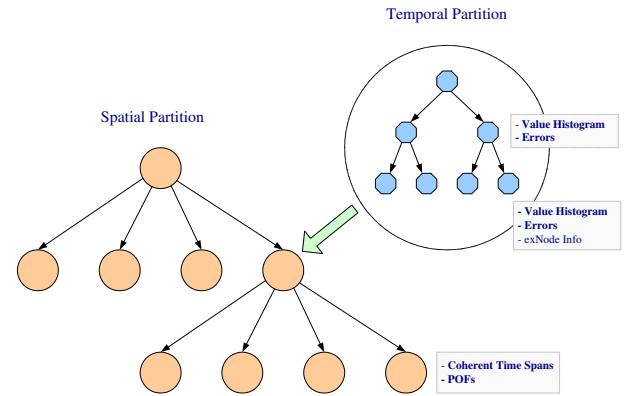


Figure 1: An Enhanced TSP (ETSP) tree.

3 OUR DATA MANAGEMENT MIDDLEWARE

The key components in our DDMS include an Enhanced TSP (ETSP) tree for data selection and a cache system for data prefetching. The details of the ETSP tree and our cache design are presented in the following subsections.

3.1 ETSP Tree

The ETSP tree (Figure 1) has the same skeleton of a TSP tree with more information stored, which only requires *one-time* construction. Our DDMS assumes a coarse level of granularity and only deals with blocks of voxels. Typical voxel block sizes are 32 cubed, 64 cubed, etc. All voxel blocks, throughout a time-varying data set, are managed using an ETSP tree. At runtime, given an arbitrary transfer function (from a family of an unlimited number of transfer functions), the same ETSP tree can be applied to support dynamic discovery of voxel blocks that are un-occluded and non-transparent, at adaptive levels of detail. Our DDMS also assumes a consistent data access pattern when navigating time-varying data sets, i.e., that the user will maintain the same viewing setup, including view angle, frustum and viewport, to render a sequential set of time steps in the simulation [21, 23].

3.1.1 Encoding Visibility in ETSP Tree

As shown by [10], temporal occlusion coherence is an effective metric for culling occluded portions of a volume, early in the visualization pipeline. To leverage temporal occlusion coherence, first an *Optimized Plenoptic Opacity Function* (OPOF) needs to be computed for each volume block. An OPOF encodes the opacity of a volume block for all possible viewing angles around the block, under a family of transfer functions that share the same set of basis functions. By giving each basis transfer function a different weight, one can effectively control the opacity of different features that have been identified in the data set.

Temporal occlusion coherence is straightforward to apply. For instance, one can partition the entire spatial domain of a volume into blocks. For each block, the OPOF function is computed for each time step. Then, by comparing the opacity information between neighboring time steps for the same block, coherent time intervals can quickly be identified. To accelerate the searching process, such coherent time intervals are stored in our Space-Partitioning Tree. The coherent time intervals of an octree internal node can be computed from the combination of the intervals stored at its child nodes, in a bottom-up manner. Using the coherent time intervals, the overhead to conservatively (optimistically) esti-

mate visibility status of each volume block is limited solely to those blocks behind some block whose block-wise opacity has changed from the previous time step.

At runtime, we first traverse the Space-Partitioning tree to discover all blocks that are un-occluded. Then, we use methods described in the next section to further cull away volume blocks that are highly transparent. For blocks that are not completely transparent, but sufficiently uniform in the voxel *RGBA* assignment according to the transfer function in use, we choose a lower level of detail. This reduces the total amount of data required for the visualization.

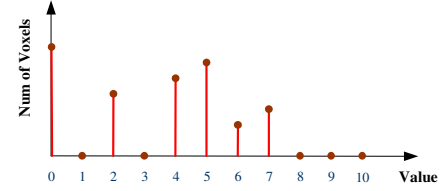
3.1.2 Encoding Value Histograms in ETSP

The goal for a multi-resolution data selection algorithm is to use as low a resolution as possible for a region, without losing too much image quality. Intuitively, if a volume block at a lower level of detail (LOD) has no noticeable difference in appearance compared to that of a higher resolution, we can reduce data movement significantly by using the lower resolution version of the block. This problem is handled much more simply in object space than in image space.

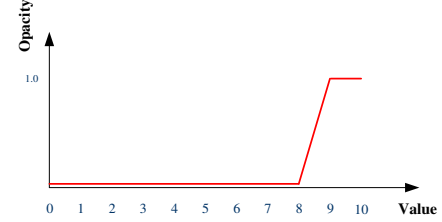
Traditionally, the resolution selection for a data region is based entirely on the raw data [23], often as a function of the variance of the values inside the region. When the variance is low (according to some user provided threshold), the block is considered uniform and rendered using a lower resolution. This approach operates directly on the raw voxel values, and works well to reduce data movement in regions of the volume showing homogeneous voxel values. However, there are exceptions to this heuristic, especially when the transfer function assigns contrasting colors to voxel values within a small range. Conversely, a block with a large variance may be classified to be of uniform color under some transfer functions, and the variance will produce an overly conservative measure.

In addition to using variance (or spatial and temporal errors) in the ETSP tree [23], we introduce a novel technique that uses *Value Histograms* for reducing data movement in regions classified as homogeneous in color and opacity. In a value histogram, the distribution of voxel values in a block is captured over the possible range of values. An example is shown in Figure 2. Suppose the value range of the block is $[0, 10]$. The value histogram that records the number of voxels with each value (on a per-block basis) is shown in Figure 2(a). Using the value distribution, we see that the given block varies widely in value. Here, we have two opacity functions that may be selected by the user. If the opacity transfer function *OpaFunc1* is chosen, all values in the block map to the same small opacity value, yielding a block that is highly transparent and visually uniform, and can be rendered at a lower resolution. If the opacity transfer function *OpaFunc2* is used, the block has high visual variation and needs to be rendered in the original high resolution. (In this example, we show only the opacity. The actual resolution selection also relies on other aspects of a transfer function.)

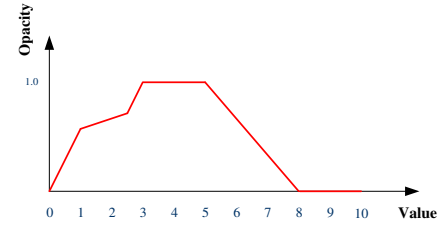
Since a value histogram is computed based solely on the values in a voxel block, we can pre-compute value histograms for all blocks *a priori*. The resolution of the value histogram usually corresponds to the number of entries used in the lookup table of the transfer function. Without loss of generality, consider this resolution as being k . Typically, k would assume a value of 256 when 8-bit voxel values are dealt with, for example. As a result, the value histogram describes the number of voxels in a block that correspond to each entry in the transfer function lookup table. To minimize the storage cost, instead of storing an integer for each entry in the value histogram, we use only one bit for each value histogram entry – 1 for “having voxels taking this value” and 0 for “no voxels taking this value”. Besides offering the advantage of a compact storage format, the bit vector also allows us to use efficient bit-wise operations during run-time rendering computation. In addition, to speedup the searching process, we compute the value histogram



(a) Value histogram



(b) Opacity transfer function *OpaFunc1*



(c) Opacity transfer function *OpaFunc2*

Figure 2: Value Histogram of a sample block and two opacity transfer functions, *OpaFunc1* and *OpaFunc2*. When *OpaFunc1* is used, the block is highly transparent and visually uniform, and can be rendered in the lower resolution. If *OpaFunc2* is used for high opaqueness and visual variation, the block needs to be rendered in the original higher resolution.

for an internal node of the Space-Partitioning Tree by applying the bit-wise OR operations to all value histograms stored at its child nodes. For a time-varying data set, value histograms are computed first for the leaf nodes in each Time-Partitioning Tree as we described above, and are then used to calculate value histograms for the internal nodes, in a bottom-up manner.

Similar to the value histogram, from the transfer function, we also generate four k -bit vectors for *R*, *G*, *B* and *Alpha* respectively. In the same way as with the value histogram bit vectors, 0 corresponds to a “zero value” and 1 means that this voxel value maps to a non-zero *R*, *G*, *B* or *Alpha*. By taking a bit-wise AND operation between the value histogram and each of the 4 transfer function bit-vectors, we can quickly identify and compare all voxel values that lead to a non-zero *RGBA* tuple.

Using the value histogram, resolution selection for a block is also straightforward, essentially consisting of checking whether a given block is uniform or transparent after classification. If either is true, then a lower LOD is used for the block. Specifically, for each block we compute 4 bit-wise AND operations, i.e., masking the value histogram against the *R*, *G*, *B* and *Alpha* bit-vectors. For all voxel values with non-zero *Alpha* assignments, if the resulting *Alpha* values are below a user-specified threshold (say, 0.05), the block can be considered transparent and rendered at a lower resolution. If the variance between any two resulting *RGB* tuples is within a certain difference threshold, then a lower LOD can also be used, since the voxels in the block assume very similar colors.

3.1.3 Summary of ETSP

In summary, in an Enhanced TSP tree we store coherent time intervals, OPOFs, value variances (or spatial and temporal errors) as well as value histograms. (Figure 1)

Then using an ETSP for reducing data movement, given a specific transfer function, involves the following steps. First, the leaf nodes corresponding to the current time step (identified by traversing the Time-Partitioning Tree) are accessed, and the un-occluded nodes are identified. Second, three metrics are used to decide whether a lower LOD rendering of the block should be used: (i) low variance in raw voxel values, (ii) low opacity, hence less distinguishable visual differences in the block, and (iii) uniform color assignment to non-transparent voxels in the block, resulting from the user specified transfer function.

3.2 Cache Design for Data Prefetching

Large data visualization routinely deals with data sets that have to be stored out-of-core. Therefore, after all blocks have been selected for rendering the current time step, it would be beneficial if the latency required to load the chosen data from out-of-core storage could be further minimized. To meet this need, a caching module for dynamic access to out-of-core data has been designed for our data management middleware. The strategy here is to overlap data transfer time with rendering time as much as possible, using a separate cache management thread.

In our cache system, each block is identified uniquely with two values: a block identifier and a level of detail value. These values form a compound key which is used to insert and retrieve blocks from the cache. To increase the performance of certain cache-related operations, we maintain two distinct data structures. One is a hash table of cache elements is used for quick storage and retrieval. The other is a red-black tree, keyed on the last time the block was requested. Such tree structure allows for efficient removal of the least recently used (LRU) block. The elements of each of these structures are linked, to work cooperatively, as shown in Figure 3.

When an access request is made for a block, the cache first checks the hash table for the block's key. If the key is found, the block's "last requested time" is updated and reinserted into the red-black tree, and the block data is sent to the front-end. This exploits the temporal locality supported by ETSP tree, that is, a block accessed at time t is likely to be accessed again at a time following t . If the key is *not* found, we first examine our current cache size. If the cache is at its maximum capacity, we must remove those blocks whose last requested time is farthest in the past. The red-black tree is used to find and remove these blocks in $O(\log(n))$ time. With room now available in the cache, the data block is requested from the depot and its information is stored in both the hash table and the red-black tree, with the block's last request time set to the current time.

To minimize the data access time required by this visualization system, we employ a data prefetching scheme supported by our cache system. Instead of requesting a single block, the list of blocks that will likely be accessed in the future is requested, with descending priority. Based on the OPOFs and the value histograms, we are able to know such list of blocks beforehand for a small cost. We also key the red-black tree on the sum of the last request time and the priority. The first block in the list is considered the target block and is given the highest priority. After the cache request is finished, this target block must be in the cache. However, no such guarantee is made for the remaining blocks. An attempt is made to prefetch each of these blocks, but if one's sum is less than the current minimum in the red-black tree, the block is ignored. This scheme allows blocks from previous requests, that are still near the

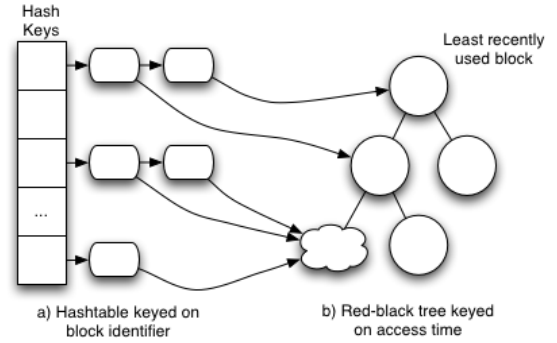


Figure 3: Caching system for quick data retrieval. A hash table is used for efficient storage and retrieval, and a red-black tree manages blocks' importance in $O(\log(n))$ time.

target block, to stay in the cache until the next miss, which benefits users who advance and retreat through data, a common habit.

4 DISTRIBUTED DATA MANAGEMENT

To support distributed sharing of large data sets, we have deployed our data management scheme into the wide-area network, leveraging the infrastructure provided by Logistical Networking (LoN). We chose LoN because of a few unique features. First, the current LoN infrastructure already provides 31 terabytes total of distributed storage in the US and 22 other countries. These resources are unorchestrated, i.e. they do not require reservation and authentication (therefore there is no need to obtain individual user accounts). Second, there are inherent redundancies, similar to how RAID disks incorporate fault-tolerance. This redundancy scheme within LoN is transparent to the end users. Third, the API used to access the data stored is relatively simple, and offers substantial performance. As demonstrated by [6], it is practical to dynamically stream a gigabyte-level image database from California to Tennessee, for supporting interactive remote visualization with image-based rendering. More background concepts on LoN can be found in [3] and implementation details, performance and fault-tolerance of LoN can be found in [20].

The overall configuration of DDMS is illustrated in Figure 4. We do not assume a specific type of local computing facility for the end users. As the necessary data is discovered and delivered to the user, on-the-fly by the data management middleware, the front-end visualization can be performed on a PC cluster, a personal computer, etc. Our prototype results have all been obtained using a moderate PC cluster.

Where LoN differs from most distributed storage systems is that data replication is internal to the file object rather than using a separate replica catalog to maintain replica availability. In LoN, the file is represented by the exNode [2]. Like the Unix inode, the exNode maintains the mapping from logical file to the stored blocks. Unlike the inode, it allows for replication and striping over multiple IBP servers. The higher-level APIs take advantage of the replication and striping to improve performance and fault-tolerance, which is critical in distributed systems. Our viz middleware depends on the performance and fault-tolerance of the LoN software to achieve good performance and overall reliability.

To deploy our ETSP tree based data management middleware on LoN infrastructures, we build a multiresolution data hierarchy for each time step. This construction is done in a bottom-up manner. Conceptually, the construction will produce an octree. The leaf nodes correspond to the data blocks in the raw data. The data block

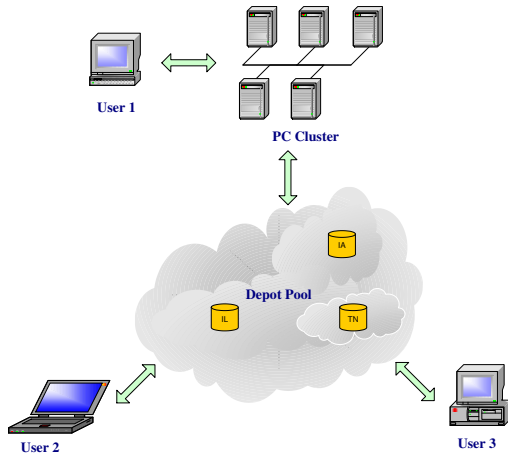


Figure 4: The system architecture.

at any internal tree node is a lower resolution combination of the data blocks from its eight children. This lower resolution data is generated using the method proposed by Guthe et al. [11]. After the data hierarchy is built, data blocks at different resolutions are uploaded to the LoN and then made accessible via exNodes. For run-time data access, each leaf node of a Time-Partitioning Tree is mapped to a corresponding exNode. Because of the large storage capacity supported by the LoN, and the high data reconstruction cost shown in [23], instead of compressing each data block and storing only coefficients, we store the full data blocks "as is." In this way, we avoid the dependencies among data blocks, as well as expensive data reconstruction at runtime. Then a common data set can efficiently be shared among a group of collaborators, each using their own transfer functions.

Typically, visualization software relies on local resources that are highly reliable (e.g. local disk or NFS). Since we are working with data that is distributed over the wide-area, we have to consider fault-tolerance. Although individual IBP depots can be highly reliable, they are in practice only as reliable as the underlying network infrastructure. A client may be unable to access stored data due to local network congestion, wide-area network congestion, congestion in the remote IBP depot's LAN, congestion on the IBP depot (due to a large number of requests) or simply failure of the depot or its host. To overcome potential faults, we use the replication capabilities of the exNode. For this paper, we stored a complete replica of the data at the University of Vanderbilt. We then create partial replicas on two additional servers at University of Iowa and StarLight in Chicago to provide redundancy. We chose partial replicas to simulate intermittent unavailability of portions of the data.

To retrieve the data, we incorporated the LoRS API into our middleware software, specifically the `lorsSetLoad()` function. This function allows us to retrieve individual data blocks ($64 \times 64 \times 64$ each). If multiple replicas are available, `lorsSetLoad()` will try each copy until the data is retrieved. If none of the replicas are available, it will return an error. If data from one replica is not available, `lorsSetLoad()` takes less than 0.2 seconds to switch from one replica to another. If a replica is available but the performance is too slow, `lorsSetLoad()` can try another source if the first source exceeds the timeout set by our viz middleware.

5 EXPERIMENTAL RESULTS

To verify the effectiveness of our data management schemes, we tested our DDMS with a parallel volume rendering system that is similar to the system reported in [10]. Our parallel software ren-

derer runs on a PC cluster with 32 2.4GHz Pentium IV Xeon processors connected by Dolphin Networks. Data produced from a Richtmyer-Mevhkov Instability (RMI) simulation at Lawrence Livermore National Laboratory are used in our experiment. We selected 64 time steps from the entire data set. Each time step is sized at about 1GB with a spatial resolution of $1024 \times 1024 \times 960$. As described before, we assume a fixed viewing direction throughout all time steps, and the time steps are rendered in a sequential order.

For this experiment, we generated and partitioned the multiresolution data for each time step, and resulting data blocks ($64 \times 64 \times 64$ each) were uploaded to three public IBP depots at Vanderbilt University, University of Iowa and StarLight in Chicago. The Vanderbilt depots provide over 3 TB of storage and run Linux 2.6 on an IBM x335 with dual 2.8 GHz Xeon, 1 GB of RAM and gigabit Ethernet. Vanderbilt's Abilene connection is 500 Mbps. The University of Iowa depots provide 1.5 TB of storage and run MacOS X Server 10.3.8 on a dual 2.0 GHz G5 Xserve with 4 GB of RAM and gigabit Ethernet. The UIowa Abilene connection is 1 Gbps. The StarLight depot provides 7 TB of storage and is a YottaYotta NetStorager running Linux 2.4 on a dual 2.2 GHz Xeon with 2 GB of RAM and a gigabit connection to the Abilene backbone. Each exNode had a complete replica at Vanderbilt. The exNodes also had partial replicas at UIowa and/or StarLight. Hence, we had three incomplete replicas accessible via exNodes. The data access code used the LoRS API to retrieve the data using an adaptive algorithm that tries each available source until the data are retrieved or no more replicas exist [20]. Usually this process takes less than 0.2 seconds if block is not available and up to the timeout value specified by the user if the block is available but the throughput is too low. In our experiment, the depots didn't participate in the actual computation or rendering.

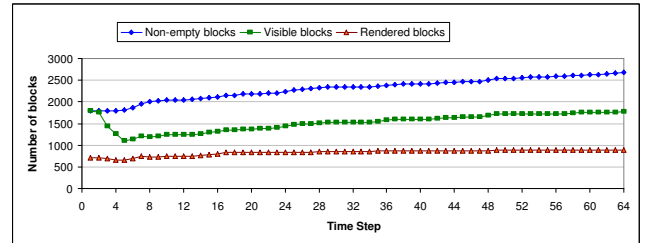


Figure 5: The comparison of the number of nonempty blocks, visible blocks, and the actually rendered multiresolution data blocks at each of 64 time steps.

The construction cost of the ETSP tree is very small. In our experiment, it took us about 21 seconds to build the whole tree for the RMI data set with 64 time steps. With the information stored at the ETSP tree, we were able to significantly reduce the number of data blocks transmitted across the network into the front-end visualization pipeline. Figure 5 compares the number of nonempty blocks, visible blocks, and multiresolution blocks selected for rendering at each of 64 time steps. In the worst case, only about 40% of non-empty blocks are actually rendered at each time step. Although lower resolutions were used for some regions, we could still maintain high image quality through error tolerance selection. Figure 6 compares the rendering result of the original data with that of a multiresolution data. Like any multiresolution algorithm, there is a tradeoff between the performance and the image quality. With a higher error tolerance the image quality drops, but fewer blocks are accessed and rendered.

When dealing with large data sets, disk I/O and network communication always incur high overheads. To smooth and hide such latencies, we used our caching scheme to prefetch the data at the next $N(\geq 1)$ time steps. In our experiments, we have found the

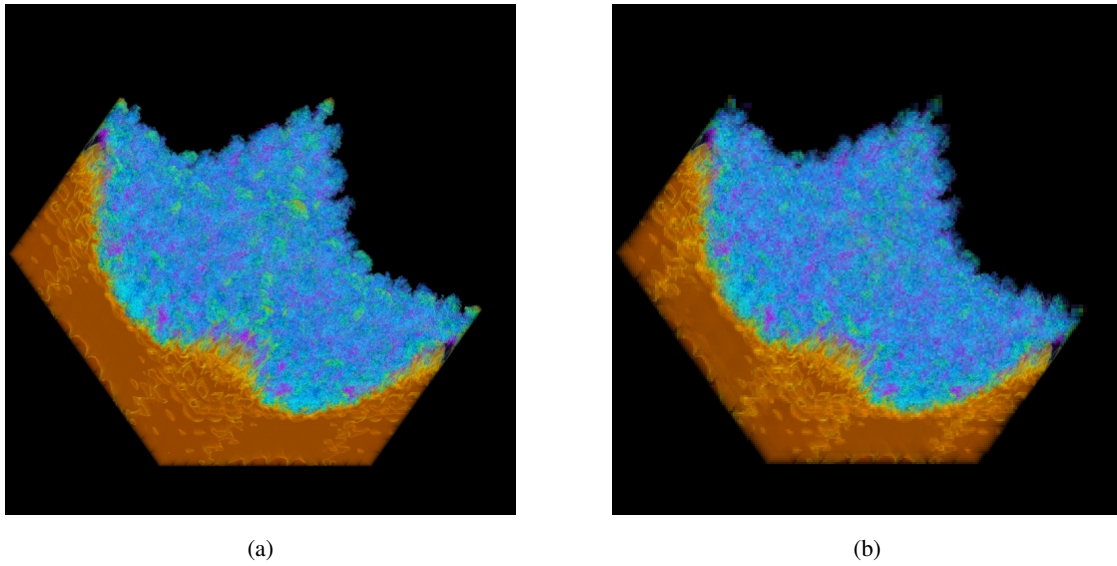


Figure 6: The image of (a) the original volume (2678 blocks) and (b) a multiresolution volume (887 blocks selected by spatial error tolerance = 700, visual error tolerance = 0.001.)

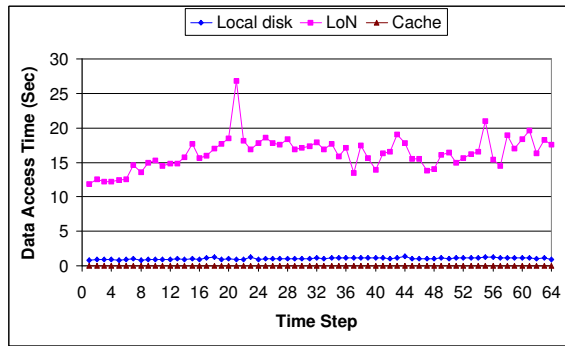


Figure 7: The comparison of the data access time needed at each of 64 time steps if the data is stored at local disks, at the distributed storage system (LoN), and prefetched through our cache system.

time to prefetch the necessary data for a time step to be usually less than the time that it takes to render that time step. Specifically, at each computing node, prefetching one data block took less than 0.16 seconds, while it usually takes 0.2 seconds to render a block (roughly 8×8 pixels per block, 0.5 step size). In such cases, the network overhead is completely hidden.

We ran additional tests for three different scenarios: data is stored locally at the rendering cluster, data is stored at distributed storage depots, and data is prefetched into our cache system. As shown in Figure 7, orders of magnitude difference exists in the resulting performances. The worst case is obviously when the entire data set is stored in the distributed storage depots without using caching. The best case is using the cache when the latency incurred by either local disk I/O or remote network traffic can all be effectively hidden completely. In our experiment, less than 0.001 second was observed for this case. The scenario of not using cache but having the entire data local is moderate in terms of performance. We note here that the regular access pattern in our rendering software allows great cache coherence so that prefetching effectively precludes possibilities of cache misses. For the best prefetching

performance, the rendering time of each time step should be longer than the time to transfer the data. Thus, without the advantage offered by DDMS, both the network transmission cost and rendering overhead will increase proportionally. Network congestion, longer rendering time and possibly thrashing the main memory on local machines are all potential system bottlenecks.

6 CONCLUSION AND FUTURE WORK

In this paper, we propose a distributed data management system for large-scale volume visualization that allows researchers at geographically distributed locations to share data and research results in an efficient and flexible way. Several data structures including value histograms, an ETSP tree and a cache are introduced to assist in efficiently identifying visible region, selecting LOD and accessing the data. To avoid unnecessary data replication at different locations, we integrate our data management system with a distributed data storage system, LoN, for the easy sharing of large-scale data. A multiresolution hierarchy of the data is constructed to support adaptive data selection and rendering. The whole data management middleware as well as a parallel volume rendering system are implemented and tested to show the effectiveness of proposed schemes and data structures.

Much work still needs to be done. More visualization components will be added to support various fields of applications. Our system needs to support users with different system resources, and more intelligence will be provided to help the user decide the suitable visualization techniques or data resolution. Other frequency based analysis methods need to be tried to improve our LOD selection. As software for hardware accelerated rendering become portable and more widely used, the rendering time of data blocks could be greatly reduced in distributed systems. To still be able to sufficiently hide network latency from front-end users, we will seek to further reduce data transfer costs by incorporating techniques such as data compression and optimized network protocols.

ACKNOWLEDGMENTS

This work was supported in part by NSF grants ACI-0329323 and NGS-0437508, and in part by the Mathematics, Information and Computational Sciences Office, Office of Advanced Scientific Computing Research, U. S. Department of Energy, under contract No. DE-AC05-00OR22725 with UT-Battelle, LLC. Special thanks to Professor Micah Beck, Professor Jack Dongarra and Clay England at University of Tennessee. The RMI data set was provided by Mark Duchaineau at Lawrence Livermore National Laboratory. The authors would like to thank Kevin McCord and the rest of the Accre team at Vanderbilt and Boyd Knosp and Michael Jenn at the University of Iowa for letting us use their depots.

REFERENCES

- [1] High performance storage system. In [http : //www.hpss – collaboration.org/hpss/](http://www.hpss-collaboration.org/hpss/).
- [2] A. Bassi, M. Beck, and T. Moore. Mobile management of network files. In *3rd International Workshop on Active Middleware Services*, 2001.
- [3] M. Beck, T. Moore, and J.S. Plank. An end-to-end approach to globally scalable network storage. In *ACM SIGCOMM*, 2002.
- [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*, 2000.
- [5] K. Brodlie, D. Duce, J. Gallop, Sagar M., Walton J., and J. Wood. Visualization in grid computing environments. In *Proceedings of IEEE Visualization*, pages 155–162, 2004.
- [6] J. Ding, J. Huang, M. Beck, S. Liu, T. Moore, and S. Soltesz. Remote visualization by browsing image based databases with logistical networking. In *Proceedings of Supercomputing*, 2003.
- [7] K. Engel, O. Sommer, C. Ernst, and T. Ertl. Remote 3d visualization using image-streaming techniques. In *ISIMADE'99*, pages 91–96, 1999.
- [8] K. Engel, O. Sommer, and T. Ertl. An interactive hardware accelerated remote 3d-visualization framework. In *Proceedings of Data Visualization*, pages 167–177, 2000.
- [9] J. Gao and H.-W. Shen. Parallel view-dependent isosurface extraction using multi-pass occlusion culling. In *2001 IEEE Symposium in Parallel and Large Data Visualization and Graphics*, 2001.
- [10] J. Gao, H.-W. Shen, J. Huang, and J. A. Kohl. Visibility Culling for Time-Varying Volume Rendering Using Temporal Occlusion Coherence. In *Proceedings of IEEE Visualization '04*, pages 147–154, 2004.
- [11] S. Guthe, M. Wand, J. Gonser, and W. StraBer. Interactive Rendering of Large Volume Data Sets. In *Proceedings of IEEE Visualization '02*, pages 53–60, 2002.
- [12] Z. Liu, A. Finkelstein, and K. Li. Progressive view-dependent isosurface propagation. In *Vissym'01*, 2001.
- [13] Y. Livnat and C. Hansen. View dependent isosurface extraction. In *IEEE Visualization '98*, pages 175–180, 1998.
- [14] E.J. Luke and C.D. Hansen. Semotus visum: a flexible remote visualization framework. In *Proceedings of IEEE Visualization*, pages 61–68, 2002.
- [15] K.-L. Ma and D. M. Camp. High performance visualization of time-varying volume data over a wide-area network. In *Proceedings of Supercomputing*, 2000.
- [16] N. Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2), 1995.
- [17] O.E.B. Messer, M. Liebendoerfer, W. R. Hix, A. Mezzacappa, and S. W. Bruenn. The Impact of Improved Weak Interaction Physics in Core-Collapse Supernovae Simulations. In *Proceedings of the ESO/MPA/MPE Workshop*, 2002.
- [18] A. A. Mirin, R. H. Cohen, B. C. Curtis, W. P. Dannevik, A. M. Dimits, M. A. Duchaineau, D. E. Eliason, D. R. Schikore, S. E. Anderson, D. H. Porter, P. R. Woodward, L. J. Shieh, and S. W. White. Very High Resolution Simulation of Compressible Turbulence on the IBM-SP System. In *Proceedings of Supercomputing Conference '99*, 1999.
- [19] A. Neeman, P. Sulatycke, and K. Ghose. Fast remote isosurface visualization with chessboarding. In *Proceedings of Parallel Graphics and Visualization*, pages 75–82, 2004.
- [20] J. S. Plank, S. Atchley, Y. Ding, and M. Beck. Algorithms for high performance, wide-area, distributed file downloads. *Parallel Processing Letters*, 13(2):207–224, 2003.
- [21] H.-W. Shen, L.-J. Chiang, and K.-L. Ma. A fast volume rendering algorithm for time-varying fields using a time-space partitioning (tsp) tree. In *IEEE Visualization '99*, pages 371–377, 1999.
- [22] P. M. Sutton and C. D. Hansen. Isosurface extraction in time-varying fields using a temporal branch-on-need tree(t-bon). In *IEEE Visualization '99*, pages 147–154, 1999.
- [23] C. Wang and H.-W. Shen. A framework for rendering large time-varying data using wavelet-based time-space partitioning (wtsp) tree. In *Technical Report No. OSU-CISRC-1/04-TR05, Department of Computer Science and Engineering, The Ohio State University*, 2004.