Scalable Data Servers for Large Multivariate Volume Visualization

Markus Glatter, *Student Member, IEEE*, Colin Mollenhour,
Jian Huang, *Member, IEEE*, and Jinzhu Gao, *Member, IEEE*

Abstract—Volumetric datasets with multiple variables on each voxel over multiple time steps are often complex, especially when considering the exponentially large attribute space formed by the variables in combination with the spatial and temporal dimensions. It is intuitive, practical, and thus often desirable, to interactively select a subset of the data from within that high-dimensional value space for efficient visualization. This approach is straightforward to implement if the dataset is small enough to be stored entirely in-core. However, to handle datasets sized at hundreds of gigabytes and beyond, this simplistic approach becomes infeasible and thus, more sophisticated solutions are needed. In this work, we developed a system that supports efficient visualization of an arbitrary subset, selected by range-queries, of a large multivariate time-varying dataset. By employing specialized data structures and schemes of data distribution, our system can leverage a large number of networked computers as parallel data servers, and guarantees a near optimal load-balance. We demonstrate our system of scalable data servers using two large time-varying simulation datasets.

Index Terms— Parallel and distributed volume visualization, large Data Set Visualization, multi-variate Visualization, volume Visualization.

1 INTRODUCTION

Today's scientific research frequently involves large datasets that are hundreds of gigabytes or more. In order for visualization to adequately handle datasets of that scale, techniques targeting specific applications are often needed. In this work, we would like to focus on large multivariate volume visualization, in which the variables may either come directly from the simulation or be subsequently derived from the simulated variables. This application is important due to its utility to basic scientific research. It is also interesting because it brings forth a grand challenge for today's computing infrastructure.

The community already faces a growing disparity between the available bandwidth in the computing infrastructure and the total amount of data to be processed for visualization. Introducing k variables on each voxel further exacerbates the disparity by a linear factor of k . As a real-world example, a recent astrophysics simulation of supernova generates a 300 time step dataset of a spatial resolution of 864^3 . This size is already considered large by current standards. However, with research questions of significance in astrophysics simultaneously involving variables such as velocity, tangential velocity, angular momentum, entropy, and various derivatives of those variables, there are indeed few existing systems that can adequately support the ensuing visualization needs.

In designing our system, we leverage two basic approaches: parallelism and data culling. To handle datasets on the scale of hundreds of gigabytes, parallelism with excellent scalability seems to be the most plausible approach. In addition, our undertaking aims to intelligently reduce the required amount of data processing and be able to do so as early during the visualization pipeline as possible, i.e. effective data culling.

Our main design concept is to develop parallel data culling on the granularity of individual voxels. The data culling process is driven by compound boolean queries. Very recently, researchers have demonstrated the great potential of this design and reported early successes

in using compound queries in the context of multivariate volume visualization [20]. However, they have not yet extended their system to operate in parallel and handle very large datasets.

We explore methods to use a large number of networked but independent computers as parallel data servers to support efficient visualization, with two conceptual steps: (i) a selection module in the form of compound boolean range queries to cull unnecessary data processing early in the visualization pipeline and (ii) any visualization approach to finally render the relevant voxels.

After all voxels relevant to specified queries have been "selected" through parallel data culling, there could be several potential ways to visualize or render the selected voxel data. As an example, we have implemented hardware accelerated splatting [6] with a procedural transfer function module and parallel coordinate visualization [23]. Both techniques, in their classic forms, had difficulty dealing directly with large datasets. Other uses of our scalable parallel data servers could include, for instance, support for procedurally defined high-dimensional transfer functions, first proposed by Kniss et. al. [10]. In that context, we could easily adapt our system to handle procedural Gaussian transfer functions. All that is needed is to add a selection step to query all voxels in the rectangular domain of each high-dimensional Gaussian function and then use the Gaussian functions to assign color and opacity to voxels on the fly.

As for the actual data management, the entire dataset is distributed among all data servers according to space filling curve order in the high-dimensional attribute space. Load-balancing among data servers can be achieved, independent of the types of compound boolean range queries. Each data server leverages a data structure similar to B-tree to maintain all voxels distributed to it. As a result, we achieve a very compact structure of meta data (usually less than 0.5% in size, as compared to the size of the entire dataset). A client can query the data servers and receive queried results over the network. We refer to the selected voxels as *relevant voxels*. Regardless of dataset size and query type, we achieve a query rate of 4.2 million relevant voxels/second across a local area network.

Since our focus here is to handle large multivariate data, the main results presented are from visualizing the supernova simulation we described earlier. Specifically, our visualization runs involve eleven variables per voxel, with $864 \times 864 \times 864$ spatial resolution, over 6 time steps. The entire set used is over 100 GB in size. In order to demonstrate that the scalability of our system is not dataset-sensitive, we also included results obtained from a $1024 \times 1024 \times 960$ version of the well-known Richtmyer-Mevhkov Instability (RMI) simulation done at Lawrence Livermore National Lab.

We discuss previous works in Section 2 and then present our system in Section 3. In Section 4 we describe the deployment requirements

- Markus Glatter is with The University of Tennessee, E-mail: glatter@cs.utk.edu.
- Colin Mollenhour is with The University of Tennessee, E-mail: mollenho@cs.utk.edu.
- Jian Huang is with The University of Tennessee, E-mail: huangj@cs.utk.edu.
- Jinzhu Gao is with Oak Ridge National Lab, E-mail: gaoj@ornl.gov.

Manuscript received 31 March 2006; accepted 1 August 2006; posted online 6 November 2006.

For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org.

of our system, followed by our testing results. We conclude with a discussion of future work in Section 5.

2 BACKGROUND

Query-driven data visualization is still a relatively new approach in visualization research, yet much potential has already been demonstrated with this approach [20] for the intuitive interface it offers for facilitating understanding of complex multivariate datasets. In an effort to handle larger datasets in parallel, we find previous works in volume rendering (particularly sparse volume rendering), parallelism and search data structures most relevant to our research. We now summarize those works below.

Volume rendering has been widely studied, primarily targeting datasets with a single scalar variable. To date, a variety of algorithms have been proposed to render a volume, including raycasting [12, 22], splatting [6, 15], 3D texture hardware based rendering [2, 8] and shear-warp [11]. Among those, ray-casting, splatting and 3D texture based rendering offer the best rendering quality, with each being most efficient for specific scenarios [14].

Splatting is most efficient when only a sparse subset of an entire dataset is non-transparent. This is advantageous for all applications that are primarily concerned with features composed of interval volumes as shown in [8]. Unfortunately, a shortcoming with previous splatting algorithms stems from its lack of well-studied methods to dynamically construct the sparse volume on the fly, which hinders its application in volume rendering systems requiring dynamic user interactions. However, with the support of advanced query-based techniques like the ones demonstrated by Stockinger et. al. in [20] and our work presented here, splatting is an ideal front-end renderer to run on client machines.

In order to handle large datasets efficiently, parallelism is indispensable. Many previous researchers in parallel visualization resorted to partitioning large datasets into volume blocks or bricks; a block is the finest level of granularity of parallel I/O [25], parallel rendering [9], data culling [4] and multi-resolution data management [18]. Although block-based data organization has already significantly improved memory efficiency when handling large data, we have found that for our system, using blocks as a unit of data processing can still be too expensive. In this paper, we address this issue by delivering only the voxels relevant to a user-input query on the fly. Due to this, we choose to use hardware accelerated splatting instead of 3D texture-memory based rendering techniques. After all, the size of texture memory on today's graphics boards is still quite limited.

Although few previous works have focused on interactive volume rendering of datasets having 8 to 10 variables, efficient searching within a two-variable space formed by the min/max values of each voxel cell has been well studied and used predominantly to accelerate isosurface extraction. Near optimal performance can be achieved with methods such as span space, interval tree, extremal graph [19, 1, 13, 7], etc. While these methods can be efficient in execution time, the storage needs incurred is often high. For instance, as reported in [3], an interval tree may require 4 times the storage of the scalar volume itself.

These widely used data structures are predominately based on the binary tree and its extensions to spatial and temporal dimensions. Examples include interval tree [3], k-d tree, quad-tree, octree [17], branch-on-demand octree [21], etc. The binary tree has been favorable for reasons including: (i) simplicity and (ii) optimal asymptotic performance while balanced. Unfortunately, the storage overhead is a major technical hurdle against its use for handling large numbers of attributes. For instance, in the case of 10 attributes, to perform a binary partition in all dimensions, there are 2^{10} cases. The total storage for the pointers to the 2^{10} cases increases exponentially as the depth of the tree increases. This occurs whether these partitions are simultaneous, as in the case of octree for three dimensions, or interleaved, as with k-d tree. Given that each pointer requires four bytes of memory, any 32-bit computer will be overwhelmed by a tree of depth three. In addition, with a tree of such complexity, balancing the entire tree would be a grand challenge. But without a balanced tree, optimal performance cannot be guaranteed. Thus it is not feasible to use a binary tree based

data structure in our system.

Methods that are not based on trees to accelerate searches have also been proposed, such as bitmap indexing used in [20]. However, the use of bitmap indexing in visualization is still relatively new, has not yet been extended to operate in parallel, and seems to work best if the complete dataset as well as the meta data used for searching fits entirely within main memory.

In this work we design a parallel system configuration and a space-efficient server-side data structure to support parallel multi-variate data visualization.

3 OUR SYSTEM

3.1 Design Considerations

Large, multivariate, time-varying datasets sized at hundreds of gigabytes are increasingly common. Such datasets are too large to be held in a single machine's main memory on most computers. Since the transfer rates offered by local disk storage are considerably slower than today's high speed networks, it is often advantageous to stream data from networked data servers which hold a portion of a large dataset in main memory for fast data retrieval.

We present a scalable system of parallel data servers that provide efficient parallel data culling on large quantities of data. In our system, the data is distributed among a system of parallel data servers, each of which holds a part of the data in main memory. The system achieves high throughput by employing specialized, efficient data structures and load-balancing schemes on the server side. The user's queries to the servers consist of compound boolean range queries where multiple ranges can be defined per data variable. Please note that time and spatial coordinates of voxels are not treated differently from other attributes.

3.2 Data Distribution and Preprocessing

3.2.1 Attribute-Space Hilbert Curve

For our system, we chose individual voxels to be the targeted level of granularity. A voxel is identified by its *attributes*. These attributes include all the *variables*, such as density or entropy, on the voxel as well as the *time step* t and *coordinates* (x, y, z) of the voxel. In any dataset, no two voxels will have all of the attributes being exactly the same, because (x, y, z, t) already uniquely identifies a voxel. However, voxels with a subset of attributes being of the same values would still be common.

All attributes are stored as unsigned shorts through a binning process, or quantization, process. This allows 2^{16} different bins for each attribute. Thus, data with a precision of 16 bits or lower can be converted without a loss of precision, while data with a greater precision will have to be scaled to match the range of the unsigned short datatype.

Our system could easily switch to use unsigned integer as the precision of the binning process. However, we have found 16-bit fixed point precision to be sufficient for most scenarios. With 11 attributes (i.e. 7 simulated or later derived variables, 3 spatial coordinates and 1 time step id), a voxel is referred to as a *data item*, which requires 22 bytes of storage.

A selection is composed of compound boolean range queries and constitutes specific criteria for data culling. Each range corresponds to a particular attribute of interest. To process a "selection", we must quickly search for voxels with the corresponding attributes within the specified ranges. To do so, all data items residing on a server are sorted by each voxels attributes, with the first attribute being the most significant value, and each following attribute being less significant than its predecessor. For instance, a data item with three attributes $(1, 2, 3)$ is sorted in front of a data item containing $(1, 2, 4)$, but behind a data item $(1, 1, 2)$. Upon sorting, all data items are indexed by a search data structure similar to B-tree.

The storage overhead of the search structure is typically only a very small fraction, e.g. 0.5%, of the entire dataset's storage space. This storage need is, however, independent on the dataset's size. (see 3.3 for details). By making use of this adapted data structure we achieve very

Fig. 1. In-situ Preprocessing.

low overhead during searching and data culling, so that both querying and visualization are greatly accelerated.

A straightforward way to distribute data items to parallel data servers is to do so randomly. However, randomness does not necessarily ensure good load-balancing in all scenarios. Since a selection is most likely to query for data items with similar values, we would like the data items' distribution to satisfy the following requirement: If the attributed values of two data items do not differ significantly, they should be distributed on different servers.

This heuristic implies that data items adjacent to each other in the attribute space should be distributed to different servers. We implement this heuristic by traversing the attribute space in a locality-preserving manner, and apply a round-robin distribution scheme during the traversal. This way, we minimize the problem of data items with similar attribute values “piling up” on a few servers.

In particular, we traverse the high dimensional space by discrete Hilbert space-filling curve, which comes close to achieving optimal locality [5]. As demonstrated in [16], the great locality of Hilbert space-filling curves can indeed be leveraged to achieve great performance when handling a high-dimensional space in visualization.

A traversal along Hilbert curve can be considered as mapping, or serializing, k -dimensional vectors of integer values into a 1-D space. Here k is the number of attributes per voxel (including time step, spatial dimensions, and all original and derived data attributes). Assuming 16-bit precision per attribute, such a Hilbert curve will be of an exponential length (2^{16^k}), i.e. (65536^k) . For practical reasons, instead of traversing through such a long curve, we sort all data items using the inverse function of Hilbert curve, which computes the order on the Hilbert space-filling curve between any two data items. This sorting step must be implemented as an external algorithm due to the large quantity of data involved.

Because the Hilbert curve only comes close to optimal locality, it would be hard to prove that our data distribution scheme based on Hilbert curve in k -dimensional attribute space is optimal. However, as we will show in Table 1 (Section 4.3), our data distribution method manages to achieve a close approximation to a perfectly load-balanced system.

3.2.2 Data Preprocessing

In addition to computing attributes in parallel, there are more time consuming tasks to prepare a large multi-variate data visualization. Particularly, in our system it is a time-consuming process to organize data, originally indexed by temporal and spatial coordinates, according to order in attribute space. For hundreds of gigabytes of data, external sort algorithms implemented on hard-drives still can not offer sufficient performance.

In our system, we preprocess the input data utilizing a group of networked computers in parallel. In fact, it is exactly the same set of networked computers that will later be used as the parallel data servers. From this perspective, the external mergesort is computed *in situ*.

As illustrated in Figure 1, (1) the dataset is initially partitioned and stored on the file system as spatial blocks arranged according to time

Fig. 2. An illustrated M-ary search tree.

steps. (2) We then distribute the volume blocks to all networked computers in a round-robin fashion. (3) Each computer then locally computes all derived attributes (e.g. gradients, etc.) and sorts all data items it has into Hilbert curve order. At the end of this process, each computer, C_i , holds a different linear list of sorted data items.

After each data server is ready, (4) the client machine, acting as the orchestrator of an external *mergesort*, requests the first 32 MB from the linear list stored on each networked computer. (5) The client machine then starts an internal mergesort. (6) The resulting sorted list is then re-distributed in a round-robin fashion to all the networked computers. All network communication is buffered, in order to reduce overhead of small messages. There is a separate 32MB buffer on the client for each networked computer.

Operations 4, 5 and 6 take concurrently on all computers. After those three operations complete, each networked computer will have received an equal share of the entire dataset. (7) Each of these computers will then act as a data server. The data items are then organized locally on each server using our special search data structure (detailed in Section 3.3). No additional data movement is necessary.

3.3 Server-side Data Structure

3.3.1 Design Goals

After each data server has received its portion of the data, an efficient data structure is needed to maintain those data items. A selection request, or data culling (the inverse of selection), is formally specified by

$$\{\forall x | lb(i) \leq x(i) \leq ub(i), i \in [0, k-1] \subset N\}$$

Here lb and ub are the lower and upper bounds for each attribute, respectively. k is the number of attributes in each data item. We would like to support scenarios where $k \geq 5$, using a space-efficient data structure while offering fast runtime performance. In addition, we would like the performance to be dependent on the number of relevant voxels, but independent of the size of the entire dataset. This property would be greatly advantageous since visualization algorithms usually focus on iso-ranges or intervals based on different variables. Such is the case with isosurface extraction and volume rendering.

3.3.2 Design of Data Structure

Our primary design concept is to use an M-ary balanced search tree of depth N . A practical combination could be $M = 256$ and $N = 3$; in all situations, M should always be significantly larger than N . This is one of the primary differences between this search data structure and previous data structures, such as interval tree, k-d tree, quadtree and octree.

The M-ary search tree (Figure 2) is a classic concept which we have adapted for use in multivariate visualization. In the tree, each tree node contains a list of keys and pointers to its children. From the list of keys stored in a tree node, the child containing a given target search key can be quickly identified. The M-ary tree provides meta data to guide a search process. Due to the large branching factor, M , the M-ary search tree requires little storage space. Actual data items are not stored in the tree, but in a linear list sorted by a key function, described

in Section 3.3.3. The leaf nodes of the tree only store pointers to the respective records in a sorted linear list.

M-ary search trees, like binary search trees, have been used in other disciplines. The most noteworthy example is B-trees used frequently in database systems. Our use of the M-ary search trees has several unique aspects [24]. First, most disciplines use search trees in situations where records are dynamically inserted and deleted, requiring complex operations to keep the entire search tree balanced. The overhead of operations such as tree node adoption, rotation, etc., are fairly significant. Fortunately, with multivariate visualization, runtime insertion and deletion are not often necessary. Therefore, our algorithms for tree construction and maintenance are simpler and handle gigabyte sized volume data more efficiently. Second, we have discovered that conventional methods for record access by traversing the tree are too expensive, both in terms of the large number of addressing operations and caching performance. To address this, we use a novel method to accelerate range searches in an M-ary tree for assistance (detailed in Section 3.4.1), optimized specifically for multivariate datasets.

3.3.3 Tree Construction

While general methods for mapping arbitrary multi-field records to a single numeric key do not exist, it is quite straightforward when dealing with multivariate visualization data that has been quantized. We avoid using scalar valued hash functions to obtain keys; rather, we directly use attributes of each data item as the “key function”. This allows a much larger range of possible keys than using only single scalar values as the key. To determine the order between two compound keys, attribute a_0 is the most significant, while a_{k-1} is the least significant.

For any given dataset, the tree is a static data structure that can be pre-computed in a one-time preprocessing step. After each data server has received all data items, we compute the key for each voxel and sort the data items into an array based on the key. On a typical 2.8 GHz P4 processor, quicksort, as part of the standard C library can finish sorting 32 million voxels within 40 seconds. This overhead as a one-time preprocessing cost is quite reasonable.

From the sorted list, we can efficiently construct the tree. Given a combination of M and N values, the total number, T_n , of leaf nodes is equal to M^N . In the case of $M = 256$, $N = 3$, T_n is 16 million. Then, the average number of voxels to be stored in each leaf node, to ensure a balanced tree, can be conveniently obtained as $1/T_n$ of the whole set of data items.

In Figure 2, a tree of $M = 6$, $N = 1$ is illustrated, with the sorted array of data items shown at the bottom. The array is partitioned into M^N consecutive runs. The data item separating two neighboring runs is then used as the key in the corresponding leaf node. For instance, the run started by element v_j should have elements with keys valued between k_2 and k_3 . v_j should be the data item with the smallest key larger than k_2 , while v_k has the smallest key larger than k_5 . In a bottom-up fashion, we sequentially build the leaf nodes in left to right order and recursively fill in key values in intermediate nodes based on the range of key values within each child node.

Our experiments on the tradeoff between search performance and storage overhead have shown that keeping the number of data items pointed to by a leaf node on the order of 10 to 100 is close to optimal.

3.4 Query Handling and Complexity of Query

3.4.1 Server Side Query Handling

Traditionally, one accesses all entries in a search tree by first referencing the root node. While this might be sufficient for querying single items, we have found this conventional approach to be quite expensive for querying many relevant voxels from a large dataset at the same time. A large portion of the incurred overhead is due to addressing operations and cache misses during tree traversal.

To achieve maximal performance, we designed a special scheme to search through the M-ary tree. At the core of the entire process, we use a multi-digit counter with a carrying mechanism. For instance, we illustrate k attributes in (Figure 3). The 1^{st} attribute is the most significant, and the k^{th} is the least significant. For each attribute a range is specified, shown as the non-empty region in each column. The

Fig. 3. An illustration of server-side query handling using an M-ary search tree.

```

/* a[i] is the compound key */
for (i = k-1, carry = 0; i >= 0; i --) {
    a[i] += carry;
    carry = 0;

    if (a[i] < lb[i]) {
        a[i] = lb[i];
        for (j = i; j <= k; j ++){
            a[j] = lb[j];
        }
    }

    if (a[i] > ub[i]) {
        a[i] = lb[i];
        carry = 1;
    }
}

```

Fig. 4. Pseudo code describing how an out-of-bound compound key is “carried” back into range.

bottom of Figure 3 is the same array of sorted data items in Figure 2. In the array, the leftmost data item has the lowest key value, and the values of compound keys monotonically increase from left to right.

In the attribute space, the query is specified by the lower bound, lb_i , and upper bound ub_i . From the first (leftmost) element of the array, we search for data items meeting the compound query. Specifically, we start by searching for the lower bound lb_i in the M-ary tree. The first set of relevant voxels should correspond to a consecutive run, e.g. run A in Figure 3. We traverse run A until we encounter a voxel with its compound key a_i falling out of the specified range. To determine which data item to search next, we use the method described in Figure 4 to increase the key a_i “back into” the specified range via a “carrying” mechanism.

We note here that the resulting key, a_i , may or may not exist in the data. But we search for it in the M-ary tree regardless. If the key a_i does exist, the M-ary tree will identify its location in the array, and provide the starting point of the next consecutive run of relevant voxels, e.g. run B in Figure 3, meeting the boolean query. If not, the M-ary tree points to the next immediate data item with a compound key larger than a_i . We then repeat the above process to know whether or not this data item is in the bound. The entire process is iterative until the whole dataset has been searched.

In essence, our search mechanism performs a linear search on the sorted voxel list, and uses the M-ary tree only to skip voxels that are out of the specified range. While traversing a linear list in contiguous blocks is efficient, the M-ary tree also provides great cache coherence, because of its compact size and shallow tree-depth. In contrast, traditional tree traversal may incur a much higher overhead. Without know-

ing a priori how many data items would exist in a range, or whether such data items exist at all, one would need to make an overwhelming $\prod_{i=0}^{k-1} r_i$ number of individual queries, where r_i is the size of the range of a_i .

As a side note, the pseudo code described in Figure 4 can be adapted to handle floating point data. All that is needed is to set the “carry” variable to the smallest unit of floating point in the dataset. Finally, if a user is solely querying the last attribute, a_{k-1} , those runs shown in Figure 3 could contain as few as just one voxel. This will cause each search in the M-ary tree to generate just one data item, and thus reduce efficiency. Fortunately, by leveraging user’s domain knowledge and putting attributes queried less often as the last few attributes in our system, efficiency of the system can generally be maintained.

3.4.2 Query Handling on Client Side

A user query consists of compound boolean range queries entered via an interactive user interface on the client side. The client then simultaneously sends request to all data servers which send the query results back to the client upon data retrieval. During the data retrieval process on the server side, the data items are further reduced in size by simply sending only data parts necessary to render the current query on the client. For example, if the client’s transfer function only uses a single value from the data item to determine color and opacity of a voxel, then only that will be sent, along with the inevitable spatial coordinates and first degree gradients. This reduces the data to be sent over the network to little more than 10 bytes per voxel. After the client has gathered all incoming data, it can then use any volume visualization approach, e.g. splatting, to display the result to the user. Additionally, queries will be cached locally at the client, so that recurrent requests can be dealt with locally by obtaining the data from the cache.

3.5 The Communication Interface

The server/client interface has been designed to keep communication overhead low by using several different protocols depending on what information needs to be communicated.

In our system, a server opens a TCP port and starts listening for a client to connect to it. Upon establishing a connection, the first few bytes sent by the client indicate what kind of task it needs the server to perform. The server refers to its look-up table of functions to find the appropriate component and hands over control to it. From that moment on, the client communicates with the appropriate component directly. Different components implement different protocols where needed. For instance, the “QUERY” component, responsible for retrieving the data matching a client’s query, sends back most of its data as a plain memory copy of what it finds in the server’s data storage. There is no need for additional tags or other overhead as the client agreed upon the selected protocol by calling that particular server function.

This interface design has another advantage. As the actual core of the server simply decides what component needs to be activated, it can fork off a process to run the component and close the socket right away. This enables the server to handle multiple connections simultaneously, e.g. for the use of multi-display devices, where more than one client needs to connect to a server.

4 SYSTEM DEPLOYMENT AND RESULTS

4.1 Deployment / Requirements

Our system is relatively easy to deploy on networked commodity computers, whether part of a cluster or not. One computer will function as a rendering client, and the rest as data servers. The necessary number of parallel data servers depends on the size of the dataset. As long as more computers are available to serve as data servers, our system could be scaled to handle larger and larger datasets.

However, we do have a couple minimal requirements if interactive visualization is desired. First, all data servers combined must have enough main memory to hold the entire preprocessed data in-core. Our system is still operable without meeting this requirement, however, the additional out-of-core overhead incurred by each data server would hamper the query performance. Second, the client machine should be

able to hold the queried data in-core locally. In fact, any standard PC with 2GB of memory easily meets this requirement for most practical queries. Our client software also includes a cache to avoid retransmission of recurrent requests. Even with the cache resident in memory on the rendering client, most queries in our experiment still did not exceed the limit of local memory.

In all experiments, our data servers and the rendering client are commodity computers of the same configuration (Dual AMD Opteron 1.6 GHz, 2GB RAM), connected by a Quadrics Elan3 network offering a theoretical single-direction bandwidth of 400MB per second. No additional special hardware or software was used.

4.2 Datasets and Pre-Processing

Our system has made it feasible to examine more variables interactively within the same framework than previously possible. For experimental evaluation, we use two datasets: the TSI dataset and the RMI dataset.

The TSI dataset is the main testing dataset, since it is inherently multivariate, with time-varying density, pressure, and flow velocity simulated on each voxel. The spatial resolution is $864 \times 864 \times 864$. In astrophysics research, density and pressure are often converted to a more useful variable, entropy. Besides commonly-used attributes such as gradients of various degrees, application scientists are also interested in tangential velocity and angular momentum. The angular momentum with respect to a chosen origin for a given vector field (denoting velocity and direction of velocity) is given by $L = v \times r$, at every point where r is denoting the vector from the chosen origin to the current point, and v is the vector from the velocity field.

In our experiments, there are 11 attributes visualized for the TSI dataset: time step id, entropy, magnitudes of tangential velocity, angular momentum gradient, entropy gradient, first derivative of entropy in x-, y-, and z-directions, and x, y and z coordinates. This is also the order (significance) of the attributes, when the server side M-ary tree is constructed, with entropy being the most significant. The spatial and temporal coordinates are needed, because application scientists routinely do clipped or cut-away views of their data. Our tests used six time steps of the TSI dataset totaling 105GB of data.

The most expensive overhead incurred during data preprocessing (3.2.2) is not to construct the search data structure, but actually to compute and coalesce the attributes of a volume partition into one common file, with each voxel stored in 22 contiguous bytes (11 attributes). This step is a one-time process which takes 5 to 8 hours in our experiments, since some computation needs to take place out-of-core. Nonetheless, this step could be significantly accelerated with more sophisticated parallel computing software, which is out of scope for our research.

The one-time process that distributes the whole dataset onto multiple, e.g. 20, parallel servers is the sort in Hilbert curve order and the construction of the M-ary tree (Figure 1), which only takes about 2 hours.

For scalability experiments, we have also used 3 time steps of the larger RMI simulation data. However, since the RMI data only contains a single variable, it is not the ideal dataset for multivariate visualization. We use this data strictly for scalability evaluation. The version of RMI data we used is of $1024 \times 1024 \times 960$ spatial resolution. Again, we included 11 attributes in the following order: time step id, density (the original variable in the dataset), correlation between directly neighboring voxels in the +z direction, correlation between voxels that are one voxel apart in the +z direction, norm of density gradient, first derivative of density in x-, y-, and z-directions, and x, y and z coordinates.

Although it is well-known that the RMI dataset is highly turbulent, we were able to find some rather interesting views of the dataset Figure 5. Rendering results of the TSI dataset is provided in Section 4.4.

4.3 Timing Results

After preprocessing, including binning and discarding uninteresting voxels such as empty space, we end up with about 20GB of data items with the TSI dataset. We experimented with the TSI dataset using

Fig. 5. Two imagery results of RMI dataset. The top 3% range of density, with RGB channels of transfer function modulated by (left) correlation between directly neighboring voxels and (right) density gradient in y-direction. In both views, a query selects a single time step, which is then clipped to a slab with x-coordinate between 0 ~ 140.

configurations of 20, 30 and 40 data servers, respectively. With the 20-server configuration, each server holds roughly 980MB of data in memory. In 30- and 40-server configurations, that in-core amount of data decreases to 653MB and 490MB, accordingly.

The RMI dataset is larger and also has a much smaller portion of data constituting empty space. After preprocessing, each of 40 servers still holds about 1.6GB of data. For that reason, we did not have scalability results of the RMI data using 20- or 30-server configuration.

Running queries locally on a single data server, we are able to retrieve between 4 and 13 million data items per second. The variance is caused by the significance levels of the attributes when the M-ary tree is constructed. If a user primarily queries on the least significant (the last) attribute, 4 million data items is the maximal rate. When the most significant attributes are targeted, the rate of data retrieval increases to 13 million data items per second.

We have found in the parallel system, however, that the network is the bottleneck regardless of which attributes are being queried (see Figure 6). Due to the balanced workload among servers (shown in Table 1), the total local querying capability amounts to 80 ~ 260, 120 ~ 390, and 160 ~ 520 million data items per second, with 20, 30 and 40 data servers operating in parallel, respectively. Generally, the local query time accounts for only a fraction of the total latency, even if a server has to search through more than 1.5GB of data to find voxels matching the query.

When a query only results in a small number of relevant voxels, smaller messages are sent across the network, thus a generally lower level of performance is obtained. As the size of queries increases, the obtained query rate approaches 9 million data items per second, which corresponds to a 144 MB/sec sustained bandwidth on the network (only 16 of the 22 bytes are transmitted per data item, see 3.4.2), which is considerably less than the local query rate. Thus, the network transfer time accounts for most of the total latency.

In a separate experiment, we have run the same test using a standard Gigabit Ethernet as the interconnect. There, the obtained maximal query rate in our system is 4.2 million data items per second (roughly a 536Mbps per second sustained bandwidth).

From Figure 6, we see that the system's performance is rather stable even with differing numbers of servers. We attribute the relatively lower throughput when using 40 servers to the fact that more concurrent TCP streams are on the network, with smaller messages being sent. While the RMI dataset has almost double the amount of data for the system to handle, the achieved throughput was not reduced significantly.

In Table 1, we recorded the exact number of data items found to be relevant for queries resulting in varying number of relevant voxels. Although perfect load-balancing between data servers cannot be theoretically proven, we find the measured load imbalance negligible. The load imbalance ranged between a mere 0.012% to 0.155% for medium and large queries. For very small queries (10 – 10,000) the load imbalance increased to a reasonable 4%, since load-balancing tends to be more important for large scale queries. When more data servers are used, e.g. 30 or 40, the relative levels of load imbalance is similar to those shown in Table 1.

Fig. 6. Measured rates of query using queries of different numbers of relevant voxels.

voxels per query	voxels per server	load imbalance	
		abs.	relative
228,838	11,464 ± 178	356	0.155%
1,335,685	66,968 ± 533	1,066	0.079%
6,099,624	304,957 ± 825	1,650	0.027%
7,737,111	386,964 ± 863	1,726	0.022%
9,688,584	484,541 ± 936	1,872	0.019%
10,989,382	549,588 ± 1,100	2,200	0.020%
12,837,754	642,000 ± 1,361	2,722	0.021%
17,700,906	885,162 ± 1,711	3,422	0.019%
19,241,715	962,156 ± 1,713	3,426	0.017%
20,304,821	1,015,461 ± 1,549	3,098	0.015%
22,330,131	1,116,666 ± 1,614	3,228	0.014%
23,565,859	1,178,372 ± 1,804	3,608	0.015%
24,467,368	1,223,571 ± 1,416	2,832	0.011%
28,021,080	1,401,034 ± 1,749	3,498	0.012%
30,085,660	1,504,109 ± 1,932	3,864	0.012%

Table 1. Load imbalance for different queries from TSI data (6 time steps, 864 × 864 × 864, 11 attributes per voxel) in a configuration with 20 data servers.

4.4 Rendering Results

All images shown in this section have been rendered using our interactive query renderer, obtained typically within a few minutes of user interaction to refine a query. Our client software enables the user to select compound ranges on the fly, while the transfer function can be procedurally controlled from a GUI similar to that in [10]. Only queried relevant voxels are rendered.

Our front-end sparse volume renderer uses hardware accelerated splatting, where each fragment undergoes two texture mapping operations (one to modulate the alpha channel using a footprint, and the other to implement the transfer function). In addition, each splat is composited onto the framebuffer. Due to those reasons, hardware accelerated splatting is usually fill-limited. On an nVidia QuadroFX 1000, a consistent rate of 3 million splats per second can be achieved, whereas on an nVidia GeForceFX 5900 card, the rate drops to 2.1 million per second.

Latency is another metric of performance. It is a measurement of the time between when a user inputs a query and when an image is rendered on the screen of the client machine. In our system, $latency = query_overhead + time_to_render_one_frame$. Since querying rates are often faster than rendering rates, the latencies we observe during experiments are predominantly rendering costs. This is espe-

Fig. 7. Illustration to demonstrate flexibility of visualization: parallel co-ordinate style interface.

Fig. 8. (Left) Slice of the dataset colored by entropy. (Right) Medium range entropy colored by angular momentum.

cially true with queries resulting in 10 million relevant voxels or more.

Please note that our system enables a user to interactively examine different combinations of various attributes. This unique advantage allows users to interactively “pick and choose” among 10 or more attributes. In addition, our systems efficiency allows us to render a parallel coordinate display with volume rendered images without too much cluttering. In Figure 7, we display a parallel coordinate style graphical user interface along with the volume rendered image to aid the user in finding interesting ranges for each attribute.

By allowing simulated variables and coordinates to be queried differently, clipping becomes a convenient operation. Figure 8 (left) shows a very thin slice of the x-dimension at the dataset’s center. One can clearly recognize several layers of shockwaves at the outer boundary as well as other internal structures in the upper half of the image where the shockwave folds in. In Figure 8 (right) we selected a medium range for entropy and color voxels by the norm of angular momentum. This way we can identify several levels of subsurfaces inside the outer shockwave.

In Figure 9 (left), we query for voxels with medium range of gradient strength and highlight two time steps (red and violet). Analyzing the image we can see how the dataset evolves over time. On the left-hand side of the image the outside surface moves inward with time (the violet voxels are closer to the center than the red ones), and on the upper right-hand side the shockwave moves outward with the yellow voxels being the outer layer followed by green and then blue layers on the inside. In Figure 9 (right), we show the same query on medium tangential velocity from six contiguous time steps, but this time using the same transfer function for all time steps.

At last, one can also combine multiple independent queries in the same visualization. For instance in Figure 10, two queries are shown.

Fig. 9. (Left) Medium gradient strength over time, red and violet time steps highlighted. (Right) A query on medium tangential velocity on six contiguous time steps.

Fig. 10. Two simultaneous queries.

One is bottom 15% range of tangential velocity from the 5th time step, assigning color channels (a rainbow type of color map) with entropy values. The other is medium range of entropy from the 1st time step, with the same color map indexed by angular momentum. We do not understand the physical meaning of this rendering, if there is any. This image is meant only to demonstrate the functionality of our system.

5 CONCLUSION AND FUTURE WORK

By maintaining a nearly optimal load-balance for almost all types of compound queries, our system is efficient and scalable. It relies on external sorting according to a high-dimensional space filling curve order in attribute space, and an efficient M-ary search tree to efficiently skip irrelevant voxels in the dataset. These enabling techniques make it feasible to use more attributes than before by simply increasing the number of servers when required. As long as a sufficient number of servers are available, the amount of data that can be handled is rather straightforward.

For a user of our system, e.g. an application scientist, it is common to interact with the dataset of a simulation to verify correctness or to explore newly formulated hypotheses. To this end, one should be able to navigate not only in time and space, but also in the high-dimensional attribute space. By treating simulated and derived variables, as well as temporal and spatial coordinates, uniformly as attributes, some types of queries (e.g. Figures 8, 9) that have not been widely used in current visualization production are now convenient operations. Additionally, in a way, every server in our system accepts queries in the same way that web servers function. This allows parallel servers to operate persistently, while the client can dynamically start, stop or simply shut-down.

In the future, we plan to parallelize the rendering capabilities to support the use of multi-display devices, e.g. powerwalls. Also, we would like to adapt our server system to be run on distributed computing fa-

cilities, and to make use of certain characteristics of the underlying system structure. For example, we plan to utilize parallel I/O capabilities on a cluster system hosting a parallel file system to further reduce the amount of time spent for preprocessing.

ACKNOWLEDGMENTS

This work was supported in part by DOE Early Career PI grant DE-FG02-04ER25610, NSF grant ACI-0329323, and in part by the DOE MICS Office, under contract DE-AC05-00OR22725 and DE-FG02-04ER25651. The TSI data provided by John Blondin, at North Carolina State University, and Anthony Mezzacappa, at Oak Ridge National Lab, under the auspices of the DOE SciDAC Terascale Supernova Initiative. The RMI data provided by Mark Duchaineau at Lawrence Livermore National Lab.

REFERENCES

- [1] C. Bajaj, V. Pascucci, and D. Schikore. Fast isocontouring for improved interactivity. In *Symposium on Volume Visualization '96*, pages 39–46, 1996.
- [2] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *IEEE/ACM Symposium on Volume Visualization '94*, pages 91–98, 1994.
- [3] P. Cignoni, P. Marino, C. Montani, E. Puppo, and R. Scopigno. Speeding up isosurface extraction using interval trees. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):158–170, 1997.
- [4] J. Gao, J. Huang, C. R. Johnson, S. Atchley, and J. A. Kohl. Distributed Data Management for Large Volume Visualization. In *Proceedings of IEEE Visualization '05*, pages 183–189, 2005.
- [5] C. Gotsman and M. Lindenbaum. The metric properties of discrete space-filling curves. *IEEE Transactions on Image Processing*, 5(5):794–797, 1996.
- [6] J. Huang, K. Mueller, N. Shareef, and R. Crawfis. Fastplats: Optimized splatting on rectilinear grids. In *IEEE Visualization'00*, pages 219–227, 2000.
- [7] T. Itoh and K. Koyamada. Automatic isosurface propagation using an extrema graph and sorted boundary cell lists. *IEEE Transactions on Visualization and Computer Graphics*, 1:319–327, 1995.
- [8] J. Kniss, G. Kindlmann, and C. Hansen. Interactive volume rendering using multi-dimensional transfer functions and direct manipulation widgets. In *IEEE Visualization'01*, pages 255–262, 2001.
- [9] J. Kniss, P. McCormick, A. McPherson, J. Ahrens, J. Painter, A. Keahey, and C. Hansen. Interactive texture-based volume rendering for large data sets. *IEEE Computer Graphics and Applications*, 21(4):52–61, 2001.
- [10] J. Kniss, S. Premoze, M. Ikits, A. Lefohn, C. Hansen, and E. Praun. Gaussian transfer functions for multi-field volume visualization. In *IEEE Visualization'03*, pages 65–73, 2003.
- [11] P. Lacroute and M. Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *Proc. SIGGRAPH94*, pages 451–458, 1994.
- [12] M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(5):29–37, 1988.
- [13] Y. Livnat, H.-W. Shen, and C. Johnson. A near optimal isosurface extraction algorithm using the span space. *IEEE Transactions on Visualization and Computer Graphics*, 2(1):73–84, 1996.
- [14] M. Meissner, J. Huang, D. Bartz, K. Mueller, and R. Crawfis. A practical evaluation of the four most popular volume rendering algorithms. In *IEEE/ACM Symposium on Volume Visualization'00*, pages 81–90, 2000.
- [15] K. Mueller, N. Shareef, J. Huang, and R. Crawfis. High-quality splatting on rectilinear grids with efficient culling of occluded voxels. *IEEE Transactions on Visualization and Computer Graphics*, 5(2):116–135, 1999.
- [16] V. Pascucci and R. Frank. Global static indexing for real-time exploration of very large regular grids. In *ACM/IEEE Conference on Supercomputing (SC'01)*, pages 2–2, 2001.
- [17] H. Samet. The design and analysis of spatial data structures. New York, Addison Wesley, 1990.
- [18] 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.
- [19] H.-W. Shen, C. Hansen, Y. Livnat, and C. Johnson. Isosurfacing in span space with utmost efficiency (ISSUE). In *IEEE Visualization '96*, pages 287–294, 1996.
- [20] K. Stockinger, J. Shalf, K. Wu, and W. Bethel. Query-driven visualization of large data sets. In *IEEE Visualization'05*, page 22, 2005.
- [21] P. Sutton and C. Hansen. Accelerated isosurface extraction in time-varying fields. *IEEE Transactions on Visualization and Computer Graphics*, 6:98–107, 2000.
- [22] U. Tiede, T. Schiemann, and K. H. Hoehne. High quality rendering of attributed volume data. In *IEEE Visualization'98*, pages 255–262, 1998.
- [23] M. K. Tory, S. Potts, and M. T. A parallel coordinates style interface for exploratory volume visualization. *IEEE Trans. on Visualization and Computer Graphics*, 11:71–80, 2005.
- [24] M. Weiss. In *Data Structures and Algorithm Analysis in C++*, 2 edition, ISBN 0-201-36122-1, Addison-Wesley. Addison-Wesley, 1998.
- [25] H. Yu, K.-L. Ma, and J. Welling. A parallel visualization pipeline for terascale earthquake simulations. In *ACM/IEEE Conference on Supercomputing (SC'04)*, page 49, 2004.