Interactive Volume Rendering of Time-Varying Data

Eric B. Lum* Kwan-Liu Ma*
University of California, Davis

Abstract—We present a scalable volume rendering technique that exploits lossy compression and low-cost commodity hardware to permit highly interactive exploration of time-varying scalar volume data. A palette-based decoding technique and an adaptive bit allocation scheme are developed to fully utilize the texturing capability of a commodity 3-D graphics card. Using a single PC equipped with a modest amount of memory, a texture capable graphics card, and an inexpensive disk array, we are able to render hundreds of time steps of regularly gridded volume data (up to 42 millions voxels each time step) at interactive rates. By clustering multiple PCs together we demonstrate the data-size scalability of our method. The frame rates achieved make possible the interactive exploration of data in the temporal, spatial, and transfer function domains. A comprehensive evaluation of our method based on experimental studies using data sets (up to 134 millions voxels per time step) from turbulence flow simulations is also presented.

Keywords—Compression, disk I/O, high performance computing, out-of-core processing, parallel rendering, PC, scalable algorithms, scientific visualization, texture hardware, time-varying data, transform encoding, volume rendering

I. INTRODUCTION

In many areas of science and engineering, studying time-varying phenomena is of paramount importance to understand the intrinsic properties of the underlying physical or chemical processes. Specific examples include studies of neuron excitability, material crack propagation, thunderstorm evolution, unsteady flow surrounding an aircraft, seismic reflections from geological strata, and even galaxy mergers. Recent advances in computing and instrumentation technologies have significantly increased scientists' capabilities to conduct these studies in higher dimensional space at greater resolution. However, cost-effective visualization solutions for scientists to quickly detect and explore the complex, dynamic phenomena contained within the resulting high-resolution time-varying data are lacking.

A typical time-varying data set from a computational fluid dynamics simulation can contain hundreds or even thousands of time steps and each time step may have hundreds of millions of grid points, each potentially containing multiple variables. As a result, a single data set can easily occupy terabytes of storage, creating a formidable challenge for subsequent analysis. Traditional statistical methods of analysis, though relatively easy to compute, tend to filter out information, computed at great expense, by reducing data to a relative few numbers. Visual data exploration techniques, such as direct volume rendering, have arisen as powerful aids to researchers in the analysis of these vast data sets. Interactive visualization can greatly facilitate and expedite the exploration of these data by exploiting the human brain's ability to process enormous amounts of visual information.

However, for visualization techniques to be most effective for enhancing the qualitative understanding of complex behavior, or for detection of features of importance, they must be interactive in every sense. The ability to change classification functions and color mappings quickly and easily, animate forward and backward in time, change viewpoints, zoom in and out on features of interest, all at interactive rates, is essential for maximizing scientific productivity [3].

In this paper, we present a low-cost, scalable solution for highly-interactive direct volume rendering of time-varying, scalar data. The basic algorithm, also described in our previous work [12], employs transform coding with adaptive bit allocation to temporally compress volumes, reducing both storage and bandwidth requirements at every stage in our pipeline. Interactive volume rendering then becomes possible with a palette-based decoding technique that utilizes the texturing capability of a commodity PC graphics card. With a single PC, consumer graphics card, and an inexpensive RAID storage system, we achieve a rendering capability in which all essential aspects of the visualization process are completely interactive.

We expand upon our previous work [12] in this paper, applying our compression method in a cluster computing environment. Using a small cluster of eight consumer PCs, each with a graphics card and IDE RAID, we demonstrate the scalability of our method by achieving highly-interactive rendering with large (5123), volumetric data sets. The aggregated memory across the cluster, combined with our compression technique, allow for the in-core rendering of moderate-length sequences of time varying volumetric data, while the parallel I/O from our distributed RAID storage, combined with the reduced I/O requirements of our compression technique, provides a method for interactively rendering out-of-core data whose temporal resolution is as large as we can store.

We have evaluated an implementation of our design using a number of time-varying data sets including several turbulence simulation data sets provided by scientists at the National Center for Atmospheric Research (NCAR) and the University of Colorado. The highly interactive rendering rates achieved by our PC-based system were previously available to NCAR scientists only through the use of costly supercomputers or high-end multiprocessor graphics workstations, thereby greatly limiting their access. In the following sections, a survey of time-varying data visualization research is followed by a detailed description of our design. The performance, scalability and explorability of the prototype system we have built are demonstrated with timing re-

* CIPIC & Department of Computer Science, University of California, One Shields Avenue, Davis 95616, [lum@ucdavis.edu]
† National Center for Atmospheric Research, 1850 Table Mesa Dr., Boulder, CO 80303, clyne@ucar.edu

* CIPIC & Department of Computer Science, University of California, One Shields Avenue, Davis 95616, [lum@ucdavis.edu]
† National Center for Atmospheric Research, 1850 Table Mesa Dr., Boulder, CO 80303, clyne@ucar.edu
sults, and images, using a number of data sets with sufficiently different characteristics.

II. PREVIOUS WORK

Time-varying data visualization has been an active research area. Various approaches have been proposed to reduce the storage, corresponding I/O, and rendering demands for visualizing time-varying data in a more efficient way.

Using proper encoding and exploiting temporal coherence or spatial coherence or both, the storage requirements and rendering cost of the data may be significantly reduced. Shen and Johnson [27] take advantage of temporal coherence, using difference encoding to significantly diminish storage and rendering requirements. Westermann [30] performs wavelet encoding of each time step separately to generate a compressed multiscale tree structure. Further compression can be obtained by examining the resulting tree structures and wavelet coefficients. Ma and Shen [16] discuss how non-uniform quantization along with octree and difference encoding can be employed to speed up rendering of time-varying volume data. They show that the octrees for consecutive time steps can be merged to share subtrees. Consequently, during rendering, partial images built from subtrees that have not changed over time may be reused in later time steps.

Wilhelms and Van Gelder [32] design hierarchical data structures for controlled compression and volume rendering. They extend octrees and a branch-on-need (BON) subdivision strategy [31] to handle multi-dimensional data. Sutton and Hansen [29] propose a temporal branch-on-need tree (T-BON) as an extension to the 3-D BON tree for time-varying isosurface extraction. Shen, Chiang, and Ma [26] introduce a hierarchical data structure, called Time-Space Partitioning (TSP) tree, for a better utilization of both spatial and temporal coherence. In essence, the skeleton of a TSP tree is a standard complete octree, which recursively subdivides the volume spatially until all subvolumes reach a predefined minimum size. To store the temporal information, each TSP tree node itself is a binary tree. Every node in the binary tree represents a different time span for the same subvolume in the spatial domain. Most importantly, TSP trees allow the renderer to use data from subvolumes of different spatial and temporal resolutions, which is not possible for 4-d octrees. The TSP tree data structure has been also used by Ellsworth, Chiang, and Shen [6] to facilitate large scale volume rendering using 3-D texture hardware.

Further speed up of rendering may be accomplished by utilizing parallel computers or graphics hardware. However, even though a parallel computer can render images at multiple frames per second, without high-speed network and parallel I/O support, these two bottlenecks can still make it impossible to achieve interactive viewing. One bottleneck is the need to stream large volume files throughout the course of the visualization process. The other is the delay due to transferring the resulting images over a potentially non-dedicated network. Ma and Camp [15] develop a post-processing parallel visualization strategy based on a pipelined rendering. They demonstrate remote visualization of time-varying volume data on a PC cluster over a wide-area network. Pipelining and careful grouping of processors are used to hide I/O time and to maximize processor utilization. Visually lossless compression is used to significantly cut down the cost of transferring output images from the PC cluster to a display device through a wide-area network. Clyne and Dennis [4] employ similar techniques, using double buffering to help mask both the costs of volume data I/O over a high-bandwidth channel and hide image transmission over a TCP/IP network.

The methods introduced in this paper can work as well, and by many metrics better, as most of the aforementioned techniques. Furthermore they have the added advantage of being scalable and running on low-cost, commodity hardware making them far more accessible to a broad range of researchers.

III. ALGORITHM

Commodity PC graphics cards are capable of performing rendering that only a few years ago required a high-end graphics workstation. In particular, the 2-D texture hardware that helps generate impressive graphics for video games can also be exploited for data visualization. For example, commodity PC graphics cards have been effectively used for volume rendering static volumetric data [2], [26]. Volume rendering requires the loading of the volumetric data into the texture memory of the video card prior to rendering. The resolution of the volume that can be rendered is often limited by the amount of video memory the card contains since the access and transfer of data from main memory across the graphics bus is relatively slow compared to the direct access of graphics memory.

A. Compression

The interactive rendering of time-varying volumetric data sets offers a number of challenges because of the sheer size of the data being visualized. These data sets can be reduced in size and therefore made more manageable through the use of compression. The advantages of compressing volumetric data are twofold. First, it reduces the storage requirements needed for the data. This could allow a data set to fit in main memory that might otherwise not fit, eliminating the need for transferring data from disk. The reduction in storage can also be used to fit relatively small compressed data sets entirely in texture memory, thus eliminating the need for transferring data across the graphics bus. The other benefit of compression is a reduction in I/O. If a compressed volume fits entirely in main memory, the cost of transferring compressed data to the graphics card is lower than the cost of transferring uncompressed data. If a data set does not fit into main memory, the transferring of compressed data from disk can be substantially faster than with uncompressed data, allowing for interactive visualization from disk.

Video and main memory can be thought of as a two-level cache for volume rendering. The compression of volumetric data not only increases the amount of data that can fit in each level, but also decreases the I/O costs of transfers between these levels. Through the use of compression, and careful management of the time costs associated with the transfers between levels, it is possible to load texture maps representing volume data into video memory at rates suitable for interactivity on a commodity PC.

If a compressed volume is to be rendered directly from video memory, it must also be uncompressed using the graphics hard-
ware. This is a significant constraint since the operations supported in video hardware are extremely limited compared to a general-purpose CPU. Another constraint is imposed by the fact that it is very desirable to encode the scalar voxel values in terms of their scalar value rather than as a red, green, blue, alpha (opacity) set. Using scalar values and color indexed textures allows a user to manipulate the color palette to interactively change the opacity and color maps, permitting exploration of the data’s transfer function space. Storing voxels in terms of RGBA would require recompressing the entire data set as parameters are changed, which can be impractical for very large data sets. In addition, storing a single scalar value, rather than four color scalars reduces the amount of data by a factor of four.

Unfortunately, using indexed values puts a number of limitations on how graphics hardware can be used to decode data. Most screen and texture combining operations supported in hardware, such as Register Combiners, work in terms of the manipulation of RGBA values and not the manipulation of scalar map index values. In particular, one might consider compression methods that would deal with difference images or volumes. These differences, however, would need to be combined in terms of RGBA and not indexed scalars. This would make the difference images color map and opacity map dependent, since the difference between two volumes, in terms of RGBA, depends extensively on the transfer function being used.

**B. Palette based decoding**

With these limitations in mind, we present a method for the temporal encoding of indexed volumetric data that can quickly be decoded in hardware. The method makes extensive use of hardware support for the changing of color palettes without the reloading of textures. The cycling of color palettes can be used to create simple animations from static images. In our work we use color palette manipulation to allow a single scalar index to represent grid points at several times steps.

With paletted textures, a single scalar index is used to represent an RGB or RGBA color. The palette consists of a limited set of colors that sample the RGBA color space. Each of these colors is encoded in a single value, often a single byte. In our approach we encode a sequence of temporally changing scalar values into a single index. In this way, the value stored in each texel represents an approximation of a sequence of scalar values. Each index is therefore a sample in the space of possible time varying scalar values. The scalar values that an indexed texel represents is decoded to its temporally changing values through the frame to frame manipulation of the palette. For each frame, the color for each palette entry is set to the color found in the transfer function for the scalar encoded by that index value during that frame, as shown in the following pseudocode which renders N time steps using a single indexed texture. Note that 8-bit indexed textures are assumed.

```cpp
// the color palette to be calculated for each time step
Color palette[256];

for each timestep t (0 to N-1) {
    for each palette entry i (0 to 255) {
        palette[i]=colormap[decoder[t][i]];
    }
    // set the palette for current frame
    setPalette(palette);
    renderTexture();
}
```

The textures are rasterized to the screen using linear interpolation. Unlike dependent textures, for paletted textures linear interpolation occurs in terms of RGBA values after the table lookup. If interpolation occurred in terms of palette indices, the resulting images would show severe artifacts, since the mapping between palette indices and decoded scalar values is far from linear.

**C. Temporal encoding**

The encoding process consists of mapping sequences of scalars into single scalar indices. This operation can be approached as a vector quantization problem. We perform this process using transform encoding, specifically using the Discrete Cosine Transform (DCT) [9],[7],[25]. Transform encoding is a compression method that transforms data into a set of coefficients that are then quantized to create a more compact representation. The transform by itself is reversible, and does not compress the data. Rather, a transform is selected that puts more energy into fewer coefficients, thus allowing the less important, lower energy coefficients to be quantized more coarsely, thus requiring less storage.

The DCT is defined by:

$$C(u) = \alpha(u) \sum_{x=0}^{N-1} f(x) \cos\left(\frac{(2x+1)u\pi}{2N}\right)$$

and

$$\alpha(u) = \begin{cases} \\
\sqrt{\frac{2}{N}} & \text{for } u = 0 \\
\sqrt{\frac{2}{N}} & \text{for } u = 1, 2, \ldots, N-1 \\
\end{cases}$$

where $C(u)$ are the transformed coefficients, $N$ is the number of input samples, and $f(x)$ are the input samples. We use the DCT since it is known to have good information packing qualities and tends to have less error at the boundaries of a sequence [7]. Boundary performance is important in order to avoid discontinuities during the transition between blocks. Since our application compresses temporally, discontinuities would appear in the form of flashes between compressed sequences.

The encoding process is shown in Figure 1. First a window size is selected, which will be the length of the time sequence that will be encoded into a single value. The longer the window size, the greater the compression that will be achieved at the expense of temporal accuracy. For each window of time-evolving scalars the DCT is applied. The result is a set of coefficients equal in number to the size of the window used. The first coefficient stores the average value over the window, and tends to be
DCT. Rather than using the inverse DCT, however, each palette entry is mapped to the average scalar value that the index represents for each time step. The encoding process is repeated for every window in time.

D. 2-D texture & sub-volume optimizations

As described in the previous section, the quantization step is adapted based on the characteristics of transformed coefficients. Since the temporal properties of a data set can vary widely across a volume, it is advantageous to adaptively quantize small sections of data at a time. Our volume rendering implementation uses axis-aligned 2-D textures. To minimize the amount of error introduced in the quantization step, we encode each texture slice independently. This permits bit allocation to vary based on the temporal characteristics of each slice, as well as allowing for the quantization levels to vary based on the characteristics of the coefficients for each slice. We have found per-slice encoding shows noticeably fewer compression artifacts for a given bit rate. One negative effect of this process, however, is that an uncompressed scalar value can be encoded into different compressed values depending on the slice in the volume. In practice, however, we have found this effect to be minimal, in part because quantization occurs in slices that are nearly perpendicular to the viewing direction, thus variations from slice to slice of a scalar value are softened by the volume rendering integral.

Usually, when bit allocation occurs, most bits are used for storing the average value over a windowed sequence. As a result, when the transition occurs between two compressed sequences, the shift in average value can cause a perceived jump in the animation. We therefore interleave the starting times of the windows for each slice. Figure 2 shows such an interleaving scheme. This decorrelates temporal transitions so that the jump occurs during every frame but for interleaved slices in the volume, rather than the whole volume. This is analogous to interlaced video, except rather than being interlaced vertically, the textures are interleaved along the viewing direction. As with per-slice quantization, the volume rendering integral helps to soften the interleaving effect.

For a transform window of length $N$, without interleaving an entire new compressed volume must be loaded every $N$ frames. Since the loading of data across the graphics bus is relatively slow, this can cause a substantial drop in frame rate every $N$ frames. This problem can be solved by loading $\frac{1}{N}$ of the next compressed volume every frame, but requires storing a copy of the next volume in texture memory. This, however, is not necessary if the textures are interleaved, since for every frame $\frac{1}{N}$ of the volume can be flushed from texture memory and replaced with a new texture. Thus by amortizing data movement costs, interleaving allows for a more consistent frame rate without the expense of needing the texture memory to store a second compressed volume. If the user moves to an arbitrary frame non-sequentially, then all textures must be reloaded and there is a drop in frame rate.

![Fig. 1. DCT-based encoding. In this example, the window size is 4 and only the first 3 coefficients are stored into an 8-bit value.](image1)

![Fig. 2. When 2-D texture interleaving is utilized, for every time step, every $N$th 2-D texture is replaced starting with the $t$ modulo $N$th texture slice, where $t$ is the time step and $N$ is the compression ratio. In this example, $N$ is four. The numbers on each slice indicate which time steps the texture stores. The shaded slice is the slice that is updated at time $t$.](image2)
IV. PARALLEL IMPLEMENTATION

The method we have just described is adequate for highly interactive rendering of time-varying data sets of moderate spatial resolution. However, extremely large grids, like those arising from fluid dynamics simulations at many national laboratories, are problematic. The available texture memory in the graphics board and the I/O bandwidth, particularly between disk and main memory, constrain the number of grid points in a single volume that may be rendered interactively. To address both of these limitations we explore a parallel implementation that again employs low-cost commodity hardware.

A. Hardware

Our parallel rendering environment consists of a cluster of eight rendering nodes and a ninth node used for control and display. Each node is a PC configured with a 1.3GHz AMD Athlon processor, one gigabyte of main memory, Nvidia GeForce3 graphics with 64 megabytes of texture memory, and a level 0 IDE RAID with two 7200 RPM disk drives. Thus our parallel rendering platform has an aggregate of eight GBs of main memory and 512 MBs of video memory. Each node is attached to it’s own RAID storage unit. Aggregate bandwidth from disk to main memory and from main memory to video memory is 8 times that of a single node. The interconnect between nodes is switched 1000BaseT gigabit ethernet. Communication between nodes is performed using MPI over TCP/IP sockets.

B. Algorithm

Our low-cost, distributed memory computing platform, with it’s relatively high-latency/low-bandwidth interconnect, and the use of hardware texture mapping for volume rendering lead us to an object-space task partitioning [13],[20]. Each node in the cluster is responsible for rendering an equal-sized subset of the volume. The volume decomposition is chosen such that back-to-front visibility ordering can easily be maintained. After rendering is complete, the partial images are read back from the frame buffer and blended together to yield the final, composited image. A single node is then responsible for displaying the result. The only synchronization that is required occurs during the image composition phase. This approach allows us to fully exploit the aggregate bandwidth and capacity at each stage in our storage hierarchy, from magnetic disk to video memory. Thus we are able to render data at 8 times the spatial resolution allowed by a single node.

We elected to use a simple work decomposition strategy, dividing each volume into \(N\) equal slabs, where \(N\) is the number of rendering nodes in our cluster. Recall that our use of 2-D texture hardware requires us to slice the volume orthogonal to each of the three grid axes (object-aligned slices). Hence three slabs from each time-step are distributed to each node, one slab aligned orthogonally to each data grid axis. Each node is thus responsible for three different regions of the volume depending on which viewing direction is chosen. The slab selected for a given viewing angle is the one most orthogonal to the viewing direction. The advantage of this decomposition strategy is it’s simplicity; voxels do not have to be replicated at subregion boundaries to maintain visual fidelity. The disadvantage is that compositing costs of the partial images are higher for slabs than for other partitioning schemes (e.g blocks). In fact, because we always use slabs that are most orthogonal to the viewing direction the costs are worst case [20]. These higher costs manifest themselves in two ways: there are more active pixels to transmit between nodes (higher network costs) and more active pixels to composite (higher computational costs).

The image composition stage is performed using a variation of the binary-swap method [13]. Despite the higher compositing costs imposed by our choice of slab partitioning, we elected to use a software-based implementation of the binary-swap. The software approach allows us to blend using higher precision arithmetic to avoid artifacts due to accumulated numerical error with the narrower 8-bit arithmetic available on the graphics chip.

V. RESULTS

In the sections that follow, we present results obtained using both the serial and parallel implementations of our method. We note that the serial rendering experiments were conducted on a PC with a slightly different hardware configuration than that of our cluster nodes, previously described. Different data sets, with appropriate spatial resolutions, were also chosen for the serial and parallel experiments. For all out-of-core experiments the kernel buffer cache was flushed prior to each run.

A. Serial Rendering

Our serial rendering results were obtained on a low-cost (under $1500) PC configured with an AMD 1.2 GHz Athlon processor, 768 megabytes of main memory, an Nvidia GeForce 3 based graphics card with 64 megabytes of texture memory, and an IDE level 0 Raid (4 drives). Figure 3 displays the storage configuration of our serial implementation testbed. Using our compression method we are able to render moderate resolution, time-varying volumetric data sets at interactive rates.

Table I lists the three data sets that were chosen for this part of our study. Figure 4 shows one frame from each data set.
An animation of the turbulent vortex jets data displays a fairly nonuniform pattern over time as the vortices spread the whole domain. In NCAR’s quasi-geostrophic (QG) turbulent flow data, we witness the formation of coherent turbulent structures akin to Jupiter’s red spot. The QG calculations simulate large-scale motions in the Earth’s atmosphere and oceans and are representative in size and complexity of many Earth Sciences turbulent fluid flow simulations. The LBL shock-bubble data are simulation results of a shock wave impacting a spherical bubble of helium. The shock-bubble flow data exhibits a slowly developing structure starting from one end of the domain and eventually reaching the other end.

Tables II and III show frame rates for different compression cases using NCAR’s QG data set and LBL’s shock-bubble data set, respectively. Compressing each time step of a $256^3$ QG data set takes between 5 and 15 seconds depending on the level of compression, using an approximation of Lloyd-Max Quantization [9]. Our implementation uses eight-bit paletted textures, although our technique could be applied to hardware that supports higher precision textures for encoding strategies that allocate more bits to each transformed coefficient. The results were obtained when rendering the volume to a $512 \times 512$ window, with the volume occupying approximately one third of the window area.

If a compressed data set fits entirely in main memory, then the bottleneck in the rendering process is the transfer of textures from main memory to the graphics card. Compression helps with both of these limitations, increasing not only the number of time steps that fit in main memory, but also decreasing the amount of time necessary for transferring data across the graphics bus. If only one set of axis-aligned textures is stored in main memory, then the number of time steps that can be stored in memory increases by a factor of three at the expense of the user not being able to view the data set from an arbitrary angle without swapping data from disk.

In the case of the $256^3$ volumetric QG data, using a compression factor of four and $256$ axis-aligned textured polygons we can fit $140$ time steps into main memory and sustain a frame rate of approximately $25.8$ frames per second. If $128$ axis-aligned textured polygons are used instead, which requires transferring and drawing only half the data, the frame rate doubles and we can render $280$ time steps from memory. Without compression,
Fig. 5. Selected visualizations of the quasi-geostrophic data set produced by varying transfer functions.
the same 140 time steps no longer fit into main memory. A memory-resident subset of the uncompressed data can be rendered at only about 11.5 frames per second, compared to the 25.8 frames per second with compression. We note that although the amount of data transferred with compression is one fourth of that without, the frame rate does not scale linearly. This is due to the time required to rasterize the textured polygons to the screen. The performance would scale more linearly if a graphics card with a higher fill-rate were used, or if the fill-rate requirements were reduced by projecting the volume to a lower resolution display.

Often the temporal resolution of a data set is too large to fit the desired number of time steps into main memory even with compression. In this case it is necessary to load and render the volume from disk. Compression can substantially decrease the amount of data that must be loaded for each frame, resulting in a noticeably higher frame rate, as shown in Table II. For example, all 1492 time steps of the $256^3$ QG data set can be rendered at 13.4 fps when compressed by a factor of eight versus only 2.0 frames per second when rendered uncompressed from disk. Once the user finds a shorter temporal region of interest, that data can then be loaded into main memory and rendered at a faster frame rate, or higher image fidelity. Figure 5 shows select visualizations of the QG data set for the same time step using different transfer functions defined through interactive exploration. Figure 6 displays images of time steps 32, 39, 46, 53 of the shock-bubble data set.

By changing the window size used in the encoding step, the compression ratio and quality can be varied. Tables V, VI, and VII show the peak signal to noise ratio that results from compressing each data set over 50 time steps. Figure 7 and Figure 8 show volumes that have been rendered using varying degrees of compression. As the amount of compression increases, some of the more subtle features as well as the faster moving features can become blurred. Thus, there is a distinct trade off between the compression ratio and rendering performance versus the quality of the compressed volume. This gives users a degree of flexbility in choosing compression ratios that best meet their needs. For example, if a scientist is interested in viewing a short time sequence at high quality, a lower compression ratio can be used. On the other hand, to view a very long sequence of data at high speeds, a higher compression rate can be selected. The scientist can combine compression ratios to preview a data set at a coarser temporal resolution and then view a specific time sequence of interest with less compression.

### B. Parallel Rendering

For our parallel rendering study we use two different data sets to evaluate the performance of our implementation as listed in Table VIII. Both data sets have a spatial resolution of $512^3$. The first data set contains approximately 350 time steps and was generated from a numerical simulation of stratified decaying turbulence. The second data set, also numerically generated, represents solar turbulent compressible thermal convection. There are 200 time steps in the second data set. Since our algorithm takes no advantage of any data-dependent optimizations (e.g. early ray termination), performance on both data sets is identical. All of our timings reported below come from the longer stratified decaying turbulence simulation. Figure 9 and Figure 10 show these two data sets at different compression ratios. Tables IX and X show the peak signal to noise ratio for these data sets over 50 time steps.

In the results below each experiment was run three times. All per-frame timings are the average of the times for the slowest node in each run. A static viewpoint was used with a 20 degree rotation about both the X and Y coordinate axes. Final display was performed using a $512 \times 512$ window with the projected image occupying about 90% of the display. Recall that in our serial experiments the projected image occupied only approximately one third of the display.

Table XI shows the frame rates for different compression ratios, both in-core and out-of-core, running on all eight cluster nodes. We observe that frame rates for the in-core data are independent of compression; a maximum of 5 fps is achieved. The

<table>
<thead>
<tr>
<th>compression ratio</th>
<th>PSNR (dB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2x</td>
<td>41.1</td>
</tr>
<tr>
<td>4x</td>
<td>35.5</td>
</tr>
<tr>
<td>8x</td>
<td>32.1</td>
</tr>
</tbody>
</table>

**TABLE V**

NCAR QUASI-GEOSTROPHIC DATA SET ERROR

<table>
<thead>
<tr>
<th>compression ratio</th>
<th>PSNR (dB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2x</td>
<td>38.8</td>
</tr>
<tr>
<td>4x</td>
<td>35.3</td>
</tr>
<tr>
<td>8x</td>
<td>30.8</td>
</tr>
</tbody>
</table>

**TABLE VI**

VORTEX DATA SET ERROR

<table>
<thead>
<tr>
<th>compression ratio</th>
<th>PSNR (dB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2x</td>
<td>48.9</td>
</tr>
<tr>
<td>4x</td>
<td>45.0</td>
</tr>
<tr>
<td>8x</td>
<td>41.9</td>
</tr>
</tbody>
</table>

**TABLE VII**

LBL SHOCK BUBBLE DATA SET ERROR

<table>
<thead>
<tr>
<th>compression ratio</th>
<th>PSNR (dB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2x</td>
<td>41.9</td>
</tr>
</tbody>
</table>

**TABLE VIII**

TEST DATA SETS FOR PARALLEL IMPLEMENTATION.

<table>
<thead>
<tr>
<th>data set</th>
<th>time steps</th>
<th>spatial resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decaying turbulence</td>
<td>350</td>
<td>$512^3$</td>
</tr>
<tr>
<td>Compressible convection</td>
<td>200</td>
<td>$512^3$</td>
</tr>
</tbody>
</table>

**TABLE IX**

DECAYING TURBULENCE

<table>
<thead>
<tr>
<th>compression ratio</th>
<th>PSNR (dB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2x</td>
<td>38.7</td>
</tr>
<tr>
<td>4x</td>
<td>33.2</td>
</tr>
<tr>
<td>8x</td>
<td>28.4</td>
</tr>
</tbody>
</table>
Fig. 6. Visualizations of the shock bubble data set at different time steps.

<table>
<thead>
<tr>
<th>compression ratio</th>
<th>PSNR (dB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2x</td>
<td>46.9</td>
</tr>
<tr>
<td>4x</td>
<td>40.4</td>
</tr>
<tr>
<td>8x</td>
<td>37.2</td>
</tr>
</tbody>
</table>

TABLE XI

<table>
<thead>
<tr>
<th>compression ratio</th>
<th>fps (in core)</th>
<th>fps (out of core)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8x</td>
<td>5.1</td>
<td>3.5</td>
</tr>
<tr>
<td>4x</td>
<td>4.9</td>
<td>2.9</td>
</tr>
<tr>
<td>2x</td>
<td>4.9</td>
<td>1.6</td>
</tr>
<tr>
<td>1x</td>
<td>4.8</td>
<td>1.1</td>
</tr>
</tbody>
</table>

frame rates for the out-of-core results perform more as expected, improving with increasing compression. However, as with our serial case, the speed-up is not ideal.

Figure 11 and Figure 12 show the timing distribution of the rendering pipeline for the in-core and out-of-core results, respectively. The draw time measures the time it takes to load the textures from main memory into video memory and then render the textured polygons into the frame buffer. In the case of the in-core data, all of the textures reside in main memory and the OpenGL driver manages the paging of textures between main memory and video memory. In the out-of-core case the textures reside on disk and must be explicitly loaded from disk to main memory and then from main memory to video memory as needed. The Disk I/O indicates the time to read textures into main memory. The compositing times shown include the time to read the intermediate images out of the frame buffer, encode the images, swap with other nodes, and blend in software.

The composition time remains close to constant across all experiments, regardless of compression or whether the data reside in or out of core. Our binary-swap image composition method employs run-length encoding to avoid transmitting transparent image regions between nodes. However, changing compression levels does not impact the image significantly and thus has negligible impact on overall image composition time. Hence the composition times remain constant.

We observe from the in-core results that the draw time does not improve with the increasing level of compression as we would hope. We speculate that the renderer has become limited by the fill-rate of the graphics card. Despite the lack of increase in frame rate for the in-core case, the use of compression does allow substantially more time steps to be rendered in-core without the need to swap from disk. We also observe that image composition is dominating the results, placing an upper limit on overall interactivity regardless of individual node rendering rates. The out-of-core results behave more as we would expect, demonstrating improved performance with increasing compression. The much slower disk I/O dominates the overall costs at lower compression ratios but exhibits near linear speed-up as compression is increased. The draw time also benefits from the reduced main memory to video memory bandwidth requirements. However, the constant composition time again begins to dominate at the highest level of compression. Further compression, if it were possible, would be of little benefit. Overall linear
speed-up is not possible.

To further explore the role of compositing time in our overall performance we ran a second set of experiments where compositing was not performed. Table XII shows the frame rates for both the in-core and out-of-core data running on all eight nodes. The in-core data is limited by fill-rate and receives no benefit from compression, but the overall frame rate without compositing improves significantly. The out-of-core data, on the other hand, exhibit near perfect linear speed-up as compression is increased.

We conducted one final performance experiment to look at the benefits of increasing levels of parallelism. Table XIII shows frame rates for different numbers of cluster nodes. Figure 13 and 14 show timing distributions for increasing numbers of nodes for the in-core and out-of-core data, respectively. A compression factor of four was used in all cases. For the single node case, binary-swap is not required so the composite time is zero.

The in-core results demonstrate fair overall speed-up, albeit they are again constrained by the composite time which begins to dominate the performance as the number of nodes is increased. The draw time does, however, exhibit near ideal

---

**Fig. 7.** Visualizations of Time Step 210 of the quasi-geostrophic data set at different compression levels. As the level of compression increases, some of the finer features become blurred.

**TABLE XII**

<table>
<thead>
<tr>
<th>compression</th>
<th>fps (in core)</th>
<th>fps (out of core)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8×</td>
<td>12.1</td>
<td>11.0</td>
</tr>
<tr>
<td>4×</td>
<td>11.0</td>
<td>5.7</td>
</tr>
<tr>
<td>2×</td>
<td>11.7</td>
<td>2.7</td>
</tr>
<tr>
<td>1×</td>
<td>12.2</td>
<td>1.5</td>
</tr>
</tbody>
</table>

Frame rates without binary-swap image compositing.
VI. Discussion

Our implementation uses commodity PC hardware graphics accelerators. Although PC graphics cards are emerging with hardware support for 3-D volumetric textures, and we expect that our method would work with volumetric indexed textures, we have elected to use more commonly available 2-D texture mapping. The use of 2-D textures has the limitation that for a volume to be viewable from an arbitrary angle, three copies of the textures must be stored for each of the principle viewing directions. However, this limitation is tempered somewhat by the fact that compression can reduce the amount of texture storage required by beyond a factor of 3. When a volume is 

speed-up. This is expected as the amount of rendering a node is required to perform is reduced in proportion to the number of nodes. The composition time again remains constant The out-of-core data demonstrate even better overall speed-up. This is due in part to the fact that they never achieve the same overall frame rate as the in-core data and are therefore less impacted by the composite time. The I/O time and drawing time both display ideal linear speed-up.

### Table XIII

<table>
<thead>
<tr>
<th>Nodes</th>
<th>fps (in core)</th>
<th>fps (out of core)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>4.9</td>
<td>2.9</td>
</tr>
<tr>
<td>4</td>
<td>3.3</td>
<td>1.8</td>
</tr>
<tr>
<td>2</td>
<td>2.1</td>
<td>1.0</td>
</tr>
<tr>
<td>1</td>
<td>1.8</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Fig. 8. Visualizations of Time step 980 of the quasi-geostrophic data set at difference compression levels. As the level of compression increases, some of the finer features become blurred.
viewed out-of-core, storing extra compressed copies of the data set becomes much less of an issue since the copies of the data are stored on disk. Static data sets volume rendered with 2-D textures are more susceptible to image artifacts. However, our use of 2-D textures for time-varying data permit us to perform per-slice encoding, an important aid in reducing compression artifacts. Lastly, 2-D textures permit us to interleave texture downloads, amortizing the I/O costs across multiple time steps. Without this interleaving scheme, our peak bandwidth requirements would be substantially greater.

Since our system can render volumes from disk at interactive rates we feel it is very scalable with respect to the temporal size of a data set. With regard to the size of the data set in the spatial domain, the amount of texture memory on a single card can be a limiting factor. Since our work compresses temporally, it does not reduce the amount of texture memory utilized to below that which would be required to render a single static volume. With next generation graphics cards having increasingly larger amounts of texture memory, this single-card limitation should become diminished. For out-of-core rendering, the cost of swapping textures from the graphics card to main memory is much lower than the cost of reading from disk, thus texture memory capacity restraints become less of a concern.

Supporting data sets whose spatial resolutions exceed that which a single graphics card is capable of handling can readily be addressed by clustering multiple PCs together. Clustering effectively increases the amount of aggregate texture memory available, and just as importantly, increases the aggregate bandwidth between all the levels in the storage hierarchy. Although we were able to achieve interactive rates for in-core and out-of-core data, the one shortcoming we observed in our cluster approach is the dependency on compositing for combining inter-
mediate image results. We saw that composition times quickly dominated our overall rendering costs as other components of the rendering pipeline were sped up. Different data partitioning strategies, resulting in smaller intermediate images, could improve the composition times by reducing communication and computational requirements of compositing. However, these would complicate the rendering stage and might introduce performance problems elsewhere. We feel that the best hope for addressing compositing performance is offered by the emergence of out-board hardware image compositors such as Sepia [18], Lightning-2 [28], and SGI’s Scalable Graphics [10], as well as our own efforts [19].

The use of compression by our methods presents two potential shortcomings that are worth addressing. First, since our compression scheme is lossy there is the potential for modest, but noticeable, image quality degradation that increases with the degree of compression. However, a moderate loss of image fidelity due to compression or other optimization strategies is an acceptable tradeoff for enabling interactive exploration of temporal data, provided the gross features of evolving structures is preserved as it is in our test cases [22],[1]. It is worth noting that many NCAR researchers commonly perform crude data reduction using simple zero-order subsampling in order to accommodate interactive exploration with the tools presently available to them. In essence, they have already demonstrated a willingness to sacrifice image quality to gain interactive exploration capabilities that are essential to maximizing scientific productivity [3]. Once a feature of importance is detected in the reduced data set, the full resolution data may be further analyzed if necessary. Second, compression requires additional storage (for maintaining both the raw and compressed versions of the data), and it takes time to perform the encoding. Similarly to loss of image
produce less error, particularly when the volumetric features are significantly less error than spatial downsampling for a given bitrate. Sets with fine features, temporal compression produces significantly improved spatial resolutions. Our test results show that for data sets with the error produced by storing volumetric data sets at reduced spatial resolutions. We have combined with TSP based techniques to store textures at varying degrees of both spatial and temporal resolution. We have presented a scalable, hardware texture assisted technique for rendering time-varying volume data, and demonstrated it with experimental results. This technique is very attractive to the scientists with whom we are working because of its low cost.

VII. CONCLUSIONS

We have presented a scalable, hardware texture assisted technique for rendering time-varying volume data, and demonstrated it with experimental results. This technique is very attractive to the scientists with whom we are working because of its low cost.
and interactive rendering rates. It is now feasible to put a PC-based system, such as we described, on every scientist’s desktop, making interactive exploration of large data sets accessible to a far broader group of scientists and engineers. Researchers can browse through data in a highly interactive manner, efficiently filtering unimportant from important features, and obtain valuable qualitative information about their data content. The compression scheme used is controllable and results in visualizations suitable for interactive data exploration.

For data sets with spatial resolutions beyond what a single PC can accommodate, we have demonstrated how our system can easily be scaled by clustering together multiple PCs. Though our object-space task partitioning necessitates a relatively costly final image composition, we feel that the approach is justified as the algorithm is simple and despite the additional composition costs we are still able to achieve frame rates suitable for interactivity. Furthermore, we expect to see inexpensive, out-board compositing hardware become more widely available as cluster-based graphics solutions become more commonplace.

ACKNOWLEDGEMENTS

This work has been sponsored by the National Science Foundation under contract ACI 9983641 (PECASE Award) and through the Large Scientific and Software Data Set Visualization (LSSDSV) program under contract ACI 9982251. The quasi-geostrophic turbulent flow data set were generated by Jeffrey Weiss and Clive Baillie at University of Colorado at Boulder, James McWilliams at University of California at Los Angeles, along with Irad Yavneh at Technion. The shock-bubble flow data set was provided by the Center for Computational Sciences and Engineering at the Lawrence Berkeley National Laboratory (http://seesar.lbl.gov/ccse). We obtained the turbulent vortex flow data set through the VIZLAB of CAIP at Rutgers University. The decaying turbulence data was provided by Rodney James at NCAR and Yoshi Kimura at Nagoya University, Japan. Mark Rast, at NCAR, provided the turbulent compressible thermal convection data. The authors are grateful to all of these individuals and institutions for helping make our study possible.

REFERENCES