ParVox -- A Parallel Splatting Volume Rendering System for Distributed Visualization

P. Peggy Li, Jet Propulsion Laboratory
Scott Whitman, Equator Technologies, Inc.
Roberto Mendoza, Jet Propulsion Laboratory
James Tsiao, Jet Propulsion Laboratory

ABSTRACT

This paper presents a parallel volume rendering system, ParVox, for large volumes of 4-D data sets in regular structured grids. A parallel volume rendering API based on the splatting algorithm constitutes the core of the ParVox system. A network interface program takes commands from an X Window based GUI, calls the API to perform the rendering functions, compresses the rendered images and sends them back to the GUI window. The ParVox system is designed for interactive, distributed visualization of large multiple time steps, multiple parameters volume datasets. The parallel splatting algorithm employs both object space decomposition and image space decomposition; an asynchronous image compositing scheme based on the direct send model reduces both the communication overhead and the synchronization overhead. The ParVox system architecture, the parallel algorithm and its implementation on the Cray T3D and the parallel wavelet compression algorithm are discussed extensively in this paper. The performance results and some optimization techniques are also presented.

Keywords: Direct Volume Rendering, Wavelet Image Compression, Parallel Splatting Algorithm, Distributed Scientific Visualization.

1. INTRODUCTION

There has been a great deal of interest in implementing volume rendering algorithms on massively parallel supercomputer platforms in recent years. There are two main forces driving this development: first, the use of supercomputers for large scale scientific modeling and simulation enables the modelers to generate larger data volumes in greater speed. It becomes impractical and even impossible to visualize large volumes of datasets on workstations. Second, volume rendering of large datasets demands both extensive computational and storage resources which a workstation can hardly provide.

Parallel volume rendering algorithms are mainly based on three different sequential approaches: ray casting, shearing, and splatting. Ray casting is probably the most popular approach for direct volume rendering, which has been parallelized on many parallel architectures using either image space partition or object space partition [3,8,10,13,17,22]. Shearing performs view transformation by decomposing it into sequential 3D shear operations. Lacroute proposed an accelerated shearing algorithm, called shear-warp factorization [14]. He [15] then demonstrated a parallel implementation of the shear-warp algorithm on the SGI Power Challenge capable of real-time frame rate. While the shear-wrap approach is in general faster than ray casting and splatting, it does require a high initialization overhead to set up the run-length encoded volume and additional memory to store three copies of the encoded dataset. It also requires recalculation of the encoded volume when the volume is reclassified.

Splatting is a feed-forward voxel projection algorithm originated by Westover [26,27]. Several parallel splatting renderers have been implemented. Elvins [7] was the first to use splatting on a general purpose multiprocessor, the NCube machine. The volume is partitioned into individual slices and distributed across the processors in an interleaved fashion. Johnson and Genetti recently [12] implemented a similar splatting renderer on the Cray T3D with one set of contiguous slices distributed to each processor. Each processor renders its local data and parallel compositing is done by constructing an image manipulation pipeline where portions of the image (contiguous sets of scan lines) are passed in a round robin fashion among the processors.

Neumann [21] analyzed and implemented the three main volume rendering algorithms and concluded that splatting was consistently the fastest of the three. (Note that Neumann analyzed the shearing algorithm since the shear-warp factorization was not invented yet.) He also analyzed image space decomposition vs. object space decomposition as well as data partition choices, such as slices, slabs, and blocks. He concluded that an object space partition using block data regions required the least amount of communication. This is the method we have chosen to implement here.

Our approach is similar to Johnson's [12] in that we both use splatting with an object space partition. However, we improve the data decomposition using multiple interleaving blocks per processor to allow better load balancing. We also adapt a more sophisticated final image composition algorithm to reduce communication and synchronization overhead.

Building a parallel and distributed visualization system is far beyond the implementation of a parallel rendering algorithm alone. To integrate parallel renderers into a broader computing environment requires solving problems such as application interface, data representation and distribution, image transport, display and storage, and graphical user interface. Crockett proposed a software architecture for parallel rendering [6] which addresses issues and challenges for parallel rendering systems. In this paper, we present our solution toward a parallel, distributed volume visualization system, called ParVox.

In the next section, we give an overview of the ParVox system architecture and a brief description of each component. The parallel splatting algorithm and its implementation on the Cray T3D is described in detail in Section 3. Some earlier results of the parallel splatting algorithm were reported in [28]. In this paper, we present new results and more implementation details. The ERIC compression algorithm is discussed in Section 4 and the performance result is presented in Section 5.

2. PARVOX SYSTEM OVERVIEW

When we first started the ParVox project in 1995, we identified three requirements based on our discussion with scientists in various disciplines:

1. ParVox should be able to visualize a pre-generated volume dataset interactively in 3-D space or animate pre-generated multiple time-step volumes. The volume may be represented as a translucent color cube with selected colors representing different values or as iso-surfaces with designated threshold values.

2. ParVox should provide an API for applications to link with. It allows applications to convert 4-D volume sets into 2-D image frames with pre-determined rendering parameters, such as classification, viewing transformation, shading and lighting. The rendering parameters may be generated and recorded from an interactive session as described in 1. Storing intermediate simulation results as 2-D images greatly reduces the disk storage and the transmission time to bring the result back to the user's local computer.

3. ParVox should be able to run in concert with the application under the control of the GUI. The scientist is able to view the model result while it is generated and thus adjust the model parameters accordingly. In this mode, ParVox not only serves as a visualization system, but also a debugging tool. It saves both model development time and computing resources by viewing and analyzing results on the fly.

Most volume visualization systems act as post-processing tools, as noted in requirement #1 above, such as Vis5d [9], ParVis [4], and LinkWinds [11]. Some of the systems provide APIs for users to build their own rendering interface or link with applications, such as VolPack [15], and Vfleet [25]. None of them satisfies requirement #3 - interactive visualization while the application is running. Vis5d, and LinkWinds are workstation-based visualization systems. They implement volume rendering either in software or in graphic hardware depending on the availability of the hardware support on the particular platform. ParVis is a parallel ray-tracing volume renderer for structured grid volume datasets. It runs on Fujitsu AP 1000 machine with a GUI and display on an X Window workstation. VolPack is a volume rendering API based on the shear-wrap algorithm; it runs on shared-memory multiprocessor SGI Power Challenge series. Vfleet is a parallel ray-tracing volume rendering system with object space decomposition. It uses PVM for message passing and works on clusters of workstations and the Cray T3D.

ParVox is designed to fulfill all the above three requirements. As a post-processing distributed visualization system, it provides an X Window based GUI program for display and viewing control, a parallel input library for reading 4-D volume datasets in NetCDF [24] format, a network interface program that interfaces with the GUI running on the remote workstation and a parallel wavelet image compression library capable of supporting both lossless and lossy compression. NetCDF is chosen as the input file format for ParVox due to its popularity in the scientific computing community. The data format could be double precision or single precision floating point, integer, or one-byte character. Meta information such as data range, variable name, spacing, time tag and null data value are also included in the ParVox data file.

We provide a rendering API in the form of OpenGL including lighting, shading, classification, data redistribution, multiple-isosurface and direct volume rendering, multiple-variable and multiple time-step volume rendering. Up to five white light sources are supported; each one can be individually disabled or enabled. A data volume may have multiple variables; each variable may have its own material which defines its light reflection property. The Blinn shading model [2] is used for shading and Levoy's algorithm [16] is used for iso-surface classification.

In order to run ParVox together with a scientific simulation as an interactive analysis tool, we need to take into account the speed difference and memory requirement for simulation, rendering and image output and display. Most simulations demand extensive computation and memory resources. One simulation step may take several minutes or tens of minutes on an MPP. Rendering also requires a lot of memory and computing resources, but rendering one frame usually takes a fraction of a second. In addition, the data partition for the simulation may not be the optimal one for the renderer and a data redistribution may be necessary while switching from simulation to rendering. Therefore, we propose a functional pipelining architecture for supporting this type of application. Figure 1 shows the architecture.

In Figure 1, three functional units are running as separate partitions in one supercomputer or on different computers. The simulator, the slowest module, requires most of the resources; thus it has the largest partition. The simulated results in the form of 3-D volume sets are transferred to the rendering module via a fast and parallel communication channel. The data redistribution occurs while data is transferred from the simulator to the renderer. The renderer performs the rendering, responding to the commands from a remote GUI. The rendered images are then sent to a compressor module to perform wavelet compression. Last, the compressed images are sent to the GUI on the user's desktop via a low bandwidth link. The functional pipeline allows all three modules to run at the same time: the simulator calculates the next simulation step while the user is viewing the result from the previous time step. The renderer can render the next frame while the compressor is compressing and sending the previous frame. The three functional modules may be running on three different computers as long as the bandwidth between modules supports the required data rate.

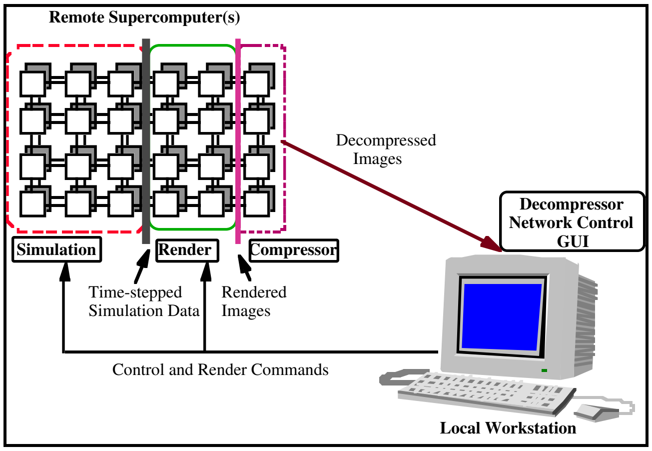


Figure 1. ParVox System Architecture

In our current implementation, we have all the utilities and libraries for post-processing distributed visualization and batch-processing animation generation. We have not implemented the functional pipelining nor designed the interface between functional units yet. The set of controls to the application will require more understanding of each individual simulation; whether there exists a generic set of controls that can be built in the GUI still remains an open question.

X Window User Interface

The ParVox GUI is designed to run on a remote workstation capable of X window display. The GUI provides a user-friendly, interactive environment for viewing and rendering control. Multiple control panels are provided to control the direct volume rendering, iso-surface classification, viewing transformation, lighting and material setting. A trackball is implemented on a small window in the viewing control panel showing a bounding box and a 3-D coordinate. ParVox also integrates the color and opacity editing program, icol [1], developed by Army High Performance Computing Research Center, University of Minnesota. Both the color map and the opacity map may be loaded, edited, and stored in the ParVox GUI.

3. PARALLEL SPLATTING ALGORITHM

Splatting, as defined by Westover [26,27], is a feed-forward voxel projection algorithm in which a filter is applied to each voxel and the resultant filtered voxel is projected onto the screen.

The reader is referred to Westover's paper for an excellent overview of the actual splatting process. Briefly, the voxels are processed in sheets -- a set of voxels in a 2-d grid of the rectilinear volume which is most orthogonal to the current viewing direction. The sheets can be processed either front-to-back or back-to-front as long as the order is consistent. Each voxel in a sheet is projected to the screen and its contribution to the underlying pixels is determined according to the applied filter kernel (Figure 2).

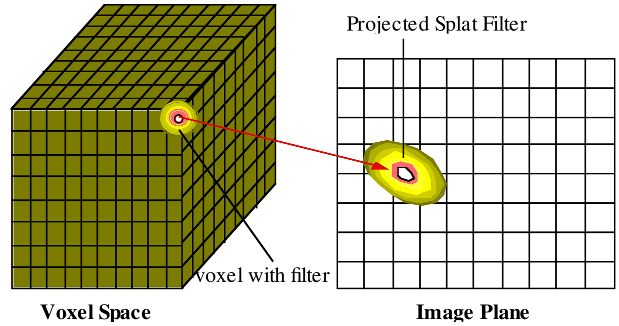


Figure 2. The Splatting Process With Filter

The parallel algorithm is very similar to the sequential algorithm with a few changes. First, each processor receives a block portion of the volume which is then rendered locally -- splatting and compositing to the local processor's accumulation buffer. This results in a correct image for this portion of the volume. This sub-image is then composited with other sub-images from other processors. Communication is required to obtain the sub-image data from the appropriate processors. The rendered sub-images will overlap each other in depth as well as in the X and Y directions. Then, final compositing of the sub-images occurs and the distributed image (among all the processors) is reconstructed to a single image for final display. Below, we describe in detail each portion of the parallel algorithm.

Object Space Decomposition

Typically, each processor is given *1/P* portion of the input data volume where *P* is the number of processors. However, to support load balancing, we assign each processor multiple sub-volumes (blocks), interleaved over the entire volume. Classification and re-classification by the user will result in different work loads per block. By assigning multiple blocks per processor, better load balancing is achieved. Karia [13] implemented a very similar data decomposition on the Fujitsu AP1000 for a parallel ray casting modification of the Ma, et al. algorithm [17]. He claims almost 50% improvement vs. 1 block per processor. We have not specifically analyzed the improvement in our approach. A dynamic approach was taken by Neumann [21] which involved copying additional data to each processor and adjusting the volume partition boundaries between the processors. The adjustment is based on the rendering times from the previous frame for all processors. This approach could easily be adapted to our implementation. Our goal, though, was to obtain a working test-bed for scientific use rather than specific optimization for load balancing.

In general, the local processor's splatting operation is no different than a sequential splatting algorithm. However, there is one important difference. The screen space projection of a splat block does not stay within its own boundary per se. This is because of the filter applied during the splatting process. Typically, the filter used has a width that extends anywhere from 20% to 70% into the adjacent voxel. Westover visually shows that a filter of size 160% of the voxel produces very satisfying results. As shown in Color Plate 2, the filters applied for the voxels on the block's border cross over to the pixels in screen space from an adjacent block rendered in another processor. This must be handled in such a way that the processor borders in object space are not readily detected in the final rendering. The solution involves sharing (or in our case copying) the border voxels from adjacent blocks. In order to do this correctly, these border voxels need to splat in such a way that they contribute to the local processor's block associated screen projection pixels but not outside of that region. This screen projection is an integration of a set of parallelogram projections of voxel sheets. In Plate 2, two such parallelograms projected from the first sheet of two adjacent blocks are shown in the screen space. A red border voxel from processor *i* will be copied to processor *j* but only splat within processor *j*'s screen space projection. The same is true for the border voxels from processor *j* (such as the green voxel in Plate 2) to processor *i*'s screen space projection.

Image Space Decomposition

After each processor has splatted and composited its local blocks, the resultant images must be composited with other processor's images. These images may overlap in depth as well as in the X and Y directions. This final compositing phase is also distributed among the processors. The work load involves partitioning image space so that each processor receives multiple screen space regions in which it performs final compositing. Multiple regions per processor helps to balance the load in this section of the algorithm. Switching from object space partition to image space partition requires data redistribution across processors. In a distributed MPP system, communication is used for data redistribution during the final compositing phase. Binary swap [8,17] requires synchronization of the processors which then alternate between compositing and communication. The solution we adopted avoids this synchronization and allows the performance for this phase to scale with the rest of the algorithm.

As shown in Color Plate 3, each processor sends the image regions from its local processor to a buffer on a "compositing" processor which performs final compositing. The three processors on the left have created the three sub-images. We illustrate the image space decomposition with colored square region lines where the arrows indicate one particular image region that happens to overlap all three processors. This image region is later composited by the compositing processor to form the final image region on the right. Rather than waiting until all local rendering of all blocks is completed, we utilize asynchronous one-sided communication during this rendering. Recall that each processor has multiple volume blocks to render in the first stage. After a block has been rendered locally, the subimage is then broken up into the corresponding screen space decomposition and each image region is sent to the appropriate processor for later compositing. One-sided communication facilitates this without the receiving processor involved. After all processors are finished rendering all local blocks, the processors turn their attention to final compositing. In other words, each processor now becomes a compositing processor. Note that all communication has already occurred because the image regions were sent during the rendering stage and no waiting or synchronization is required. More importantly, the communication is interleaved with other processors' rendering, which results in less network contention than if all communication would occur at the end. This communication is discussed in detail below.

Communication

As soon as a processor is finished rendering a block, it sends the individual final compositing regions off to the appropriate pre-assigned "compositing" processor, and then it proceeds to render the next block. In order to save memory and communication, we only send the valid pixels in an image region by putting a bounding box around the relevant data. The one-sided communication is accomplished using the Cray shmem library with the shmem_put call. Other manufacturers are implementing this type of mechanism and it is already part of the new MPI-2 standard [20]. Shmem_put involves a non-coherent DMA write to another processor's memory. All that is necessary is that the memory be available on the remote processor and consistent so that both processors know about it. However, with this approach, there is a complication if more than one processor needs to send data to a given compositing processor simultaneously. This is remedied by a semaphore mechanism. Each processor maintains a pointer to the memory buffer where the image region data will be put onto the remote compositing processor. The pointer is updated to point to the next available location by a remote processor using an atomic operation. The CRAY shmem_swap routine is used to implement this atomic operation. Once the pointer has been updated, the image region can be put to the remote processor's memory since there is no possible intrusion by any other processor.

This one-sided communication is very fast on the T3D, particularly for a put operation (up to 120 MB/sec) as opposed to a get. Since it involves no overhead on the other processor, communication can be easily overlapped with computation. In fact, because the image data is not used until much later, no cache coherence need be maintained until later. Therefore, the data is put directly into memory and not into the cache of the remote processor. Later, after all data has been put, each processor flushes its cache and starts final compositing. It is true that each processor will have to send image regions to many others but this communication will be spread out over time, thus reducing network contention. Also, once a put has been executed, the processor need not wait until the data has arrived at its destination so it can immediately go back to work.

Shading and Iso-surface Extraction

There are two basic methods to compute the lighting for the volume. The first is to compute the lighting prior to rendering based on a single eye point. This method results in shading the appropriate voxels once and as a result, when the image is rotated, it appears that the light rotates with the image. Multiple lights are needed so that the back side is illuminated. The second approach involves re-computing lighting for each frame resulting in a light which appears stationary. The second approach is very costly even in a parallel implementation and we found that the benefit gained was not worthwhile Therefore, we use the first approach, but still allowing the user to interactively change the light. After all changes are made, the volume lighting is re-computed, the shaded voxel colors are stored, and the image is re-rendered.

Similarly, iso-surface extraction requires re-classification of the voxels for each iso-surface. Our approach is to let the user choose multiple iso-surfaces up front. We then classify the entire volume prior to rendering. The user can then turn on or off each iso-surface. A bit mask is used to encode in a voxel which iso-surfaces a given voxel represents. During interactive viewing, the appropriate iso-surface voxels are splat according to the user's input with no extra work to choose among the surfaces. Of course, if the user desires to change a surface during rendering, this is facilitated as well but requires the classification step to be performed again. There is some error that may be possible using this type of encoding. The benefit to the user though is that the error does not appear in visible artifacts and changing iso-surfaces is quicker than re-classification would normally require. Of course, both shading and iso-surface extraction are performed in parallel on the local blocks. In order to calculate the gradient and splat the boundaries of each block correctly, we need to duplicate two sheets of voxels on each side of the block. No communication between processors is required during these stages.

4. WAVELET COMPRESSION ALGORITHM

The algorithm employed by this system is called Efficient Reversible Image Compression (ERIC) [18,19] and it is based on a fast integer implementation of the wavelet transform. In the lossless mode, it can achieve compression ratios that are comparable to or better than the best standard lossless compression algorithms, which do not allow for progressively accurate approximations as ERIC does.

There are two basic steps in the compression algorithm: forward Integer Wavelet Transform (IWT), in which the original image is decomposed into sub-images containing information at various levels of spatial resolution; and entropy coding, in which the sub-images are entropy coded separately.

Typically, the two-dimensional IWT is applied iteratively up to n number of levels. One level of a one-dimensional forward IWT decomposes an image into a low-frequency sub-image and a high-frequency sub-image, both of which have lower resolution. One level of a two-dimensional forward IWT is computed in two passes. First, the algorithm applies a one-level forward IWT in the horizontal dimension, which produces one low-frequency sub-image and one high-frequency sub-image. Then, a one-level forward IWT in the vertical dimension is applied to both of these sub-images, resulting in four sub-images: one low-frequency sub-image and three high-frequency sub-images. Subsequent levels of decomposition are performed on the low-frequency sub-image produced by the previous decomposition.

The next step, entropy coding, consists of decomposing the resulting transform sub-images according to the compression mode. In the lossy mode, a quantized transform image is obtained. In the lossless mode, in addition to obtaining a truncated transform image, the algorithm generates l bit planes of information, which correspond to l levels of approximation to the integer transform coefficients.

The quantized image and, in the case of lossless compression, l bit planes are now entropy coded separately. First, the quantized coefficients and bit plane information are run-length encoded (RLE) to remove redundancy. The resulting symbols are then used to compute a histogram and a Huffman table associated with this histogram. The total number of histograms and Huffman tables generated is $l+1$. Finally, the run-length encoded sequences are Huffman encoded.

Decompression basically consists of performing the reverse of the compression operations. We may choose to decompress to the full accuracy available in the compressed image or to a lesser accuracy by decompressing up to a number of bytes or a number of encoded levels less than the total available. Each entropy-coded level is Huffman decoded and run-length decoded. Transform coefficients are then composed using the resulting information. Finally, the inverse IWT is applied to decompress the image. One level of an inverse two-dimensional IWT can be computed in two steps: (1) two one-level inverse IWT in the vertical dimension, producing two integer output sub-images, followed by (2) one one-level inverse IWT in the horizontal dimension, producing one integer output image. Further levels of the reverse IWT are again computed on the full sub-image produced at the previous level and the three high-frequency sub-images at the current level.

Parallelization

Data parallelism is achieved by dividing the original image among a number of processors and applying independent compression on each subarea. The number of processors employed for compression depends on the number of spectral channels or image planes and the minimum subblock size. Each channel is compressed by a different subset of processors. The subset size is constrained by the minimum subblock size, which is $2^n \times 2^n$, where n represents the total number of levels of IWT decomposition. For this parallel implementation, the IWT uses four levels of decomposition, which gives a minimum subarea size of 16 x 16. If the size falls below this minimum, the algorithm automatically decreases the number of levels of decomposition used.

For each level of approximation, local histograms from all the processors within a channel subset are combined into a global histogram, which is used by each processor to build a global Huffman table.

In the lossless mode, the compression rate is applied globally, rather than locally, to control the inter-processor data-quality variation. Within each channel subset, each processor compares the desired compressed data to the bit plane sizes to determine the number of bit planes that can be transmitted completely. For any bit plane that has to be transmitted partially, the partial amount per processor is computed based on equal rationing.

The value of a IWT coefficient depends on d_n adjacent data points lying along the horizontal and vertical axes centered at the coefficient's location: $d_n = 2^n + d_{n-1}$ above and to the left, and $d_n = 2^{n-1} + d_{n-1}$ below and to the right, where $d_0 = 0$. When computing a coefficient which has fewer than d_n adjacent data points in any of these directions, as when close to a boundary, a form of data extension or the equivalent has to be provided. If the transform coefficients are quantized, an artificial data extension yields a different level of loss at each boundary between adjacent processors. This can result in unacceptable discontinuities if the loss is high enough.

Color plate 1 shows three different images rendered by ParVox, each one accompanied by two compressed images with different compression ratios. All the images are compressed using eight processors for each channel. The compressed image quality varies according to image content. The more opaque and lower-frequency images such as the Atlantic Ocean temperature image, (a) to (c), and the isosurfaces for the Temperature Convective model, (g) to (i), yield a better image quality at higher compression ratios. The more translucent and higher-frequency images such as the Atlantic Ocean current velocity image in (d) to (f) tend to compress less and the image quality degrades more quickly as the compression ratio increases. The subarea boundaries are clearly visible for high compression ratios, such as (c), (f), and (i).

The ideal extension at a boundary between processors is an overlapping area of height d_n, because then coefficient values near the boundary in the original area will be the same as if no segmentation of the original data had been performed (as when using only one processor). Overlapping areas of height less than d_n will help smooth the transition, but a boundary may still be visible, depending on the amount of compression. We have observed that the distortion is not detectable for compression ratios of about 30 or less. For compression ratios of less than 100, one or two overlapping lines significantly reduce the boundary discontinuity.

Decompression is performed sequentially and again consists of performing the reverse of the compression operations.

5. PERFORMANCE

The speedup chart for the rendering is shown in Figure 3. Two datasets are used for timing measurement. The ocean dataset is a volume of 640x624x37 voxels. It is a temperature dataset for the Atlantic Ocean generated from the Parallel Ocean Modeling Program on the Cray T3D. In this dataset, 30% of the voxels are land and there is no value associated with the land voxels. The data is rendered as a translucent volume with a constant opacity 0.4 assigned to each voxel. Color Plate 1(a) shows an image of the Gulf of Mexico and Northwest Atlantic Ocean where the temperature is mapped to a spectral color map ranged from purple to red. The second dataset is a 128 cubic volume derived from a parallel 3-D thermal convective model. It represents the normalized temperature distribution in a closed environment when one side of the cube is heated by a constant heat source. The volume is rendered as three iso-surfaces with threshold values 0.1, 0.3 and 0.6 as shown in Plate 1(g). The same color map used in the ocean data is also used for the iso-surface colors. The purple surface has a threshold of 0.1, blue surface a threshold of 0.3, and green surface a threshold of 0.6. All three surfaces are assigned an opacity of 0.7. The volume is decomposed into equal-sized blocks and distributed to processors interleavingly. In this benchmark, four blocks are assigned to each PE regardless of the total number of PEs. The image size is 512x512 and the final image is divided into square regions with 5 regions per PE. The rendering algorithm scales well from one PE all the way to our largest configuration, 256 PEs. More significant load imbalance is observed in the ocean dataset with small number of processors due to the fact that the land voxels are grouped together in the data set and it is very likely that an entire block contains no valid data. The valid data in the iso-surfaces (with non-zero opacity), although more sparse, scatters more evenly in the entire dataset; thus, it has better load balancing with small number of processors. The speedup curve of the iso-surface tails off more rapidly with higher number of processors because with finer decomposition of the dataset and very sparse data to calculate, the load becomes more unbalanced.

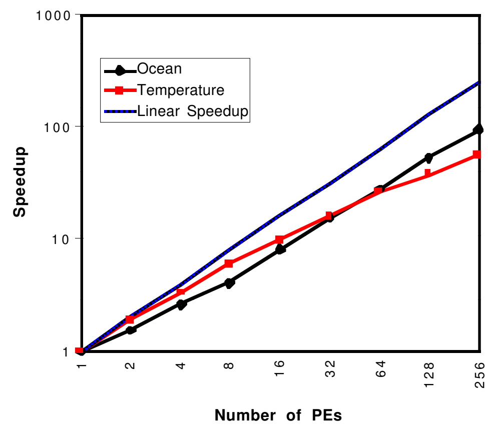


Figure 3. The Speedup for the Rendering Algorithm

Color Plate 4 is another example of ParVox images. This is a North Atlantic Ocean velocity dataset at 1/6 degree resolution and 37 layers in depth. The same color map used in the temperature datset is used to represent the magnitude of the velocity. The opacity of ech voxel is proportional to the voxel value, thus most of the image is transparent where the velocity is low. The white background is the topological map of the ocean floor. The cone shaped eddies are evident flowing from the equator to the Gulf of Mexico. A five year animation is available in the CDROM.

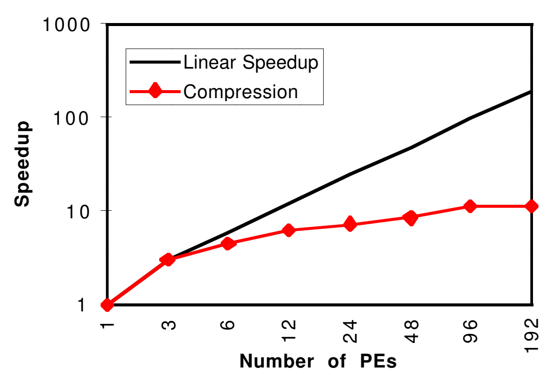


Figure 4. The Speedup for the Compression Algorithm

Figure 4 shows the speedup curve for the ERIC compression algorithm. Unlike the rendering algorithm, this algorithm does not scale well with more than 24 processors. The poor speedup performance is due to the fact that two steps in the entropy coding process are not scalable: histogram generation, which requires a global operation, and Huffman table generation, which is not parallelized. Global histogram generation time increases slightly with increasing number of processors and the time to generate the Huffman table remains constant. With more than 24 processors, these two times become dominant contributors to the entropy encoding time, which itself becomes the main contributor to overall compression time.

Both the compression time and the decompression time decrease slightly with the increase of the compression ratio. It takes about 0.1 seconds to compress a 512x512 RGB image using 96 processors (with 32 processors per color channel) and about 0.2 seconds using 3 processors (one processor per channel) for compression ratios 15 or above. The decompression time measured on an SGI Indigo2 drops from 0.6 seconds for a low compression ratio to about 0.3 seconds for compression ratios greater than 15. It is noted that the decompression speed is much slower than the compression speed, thus becoming the limiting factor of this algorithm. In this example, the maximal frame rate that can be displayed is about 3.3 frames per second.

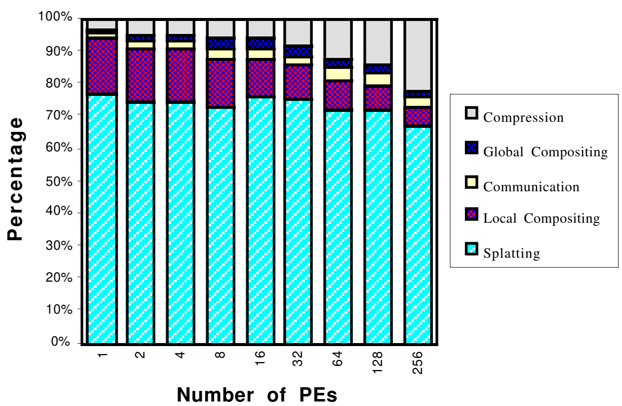


Figure 5. Timing Distribution of the entire rendering process

Figure 5 shows the timing distribution of an average rendering cycle. A rendering cycle is measured on the Cray T3D without the image transmission, decompression, and display time. Splatting takes about 75% of the total rendering time regardless of the number of processors. Local compositing is the time to composite each sheet of a processor's local block in a front-to-back order. This is part of each processor's local splatting process. A block is splatted into a sub-image after splatting and local compositing. The sub-image is then partitioned into image regions and sent to the destination processor for final compositing. The communication time is the time to send the image regions to their destination. The time is very small because the measurement only includes the time to actually make the call. We did not add any additional time to make sure the message was received since communication overlaps with computation in our algorithm. The communication does increase in percentage with the increase of the number of processors. That is because with finer decomposition of the object space and the image space, more smaller messages get sent and the average depth of the final compositing increases. Final compositing was very small in proportion to the entire rendering. The reason for this is that each processor only has to deal with several regions of the screen space and the depth complexity corresponding to those regions is very low compared to the number of sheets that are composited locally during the splatting phase. The image compression does not scale well with more than 32 processors. It becomes a significant portion (20%) of the total rendering time with 256 processors. This suggests that compression should be separated from the rendering processor and functional pipelining should be used to overlap the two modules as we proposed in Figure 1.

We measured the end-to-end rendering time from the GUI for the above two datasets. Figure 6 shows the timing results in seconds from 4 PE to 256 PEs. The end-to-end rendering time includes the time for GUI to send the render command to the network interface, the time for the network interface program to broadcast the command to all the rendering processors, the time to send the compressed image from the T3D, via the YMP frontend, and the long-haul Internet to the GUI on the local SGI Indigo2, and the time for the GUI to decompress and assemble the image and display it to the X Window. The overall overhead is about 0.37 seconds per frame regardless of the number of processors being used. This overhead is very significant for interactive visualization. Although we can pipeline the rendering, compression, data transmission, and decompression process, we still have to suffer a big latency for interactive viewing.

	4 PE	8 PE	16 PE	32 PE	64 PE	128 PE	256 PE
Ocean Dataset	14.4	9.09	5.56	3.51	1.67	1.58	1.09
Thermal Data	7.21	4.47	2.60	1.60	1.31	0.99	0.89

Figure 6. The end-to-end rendering time in seconds

6. CONCLUSION

In this paper, we presented a parallel volume rendering system, ParVox, for remote visualization of time-varying, multiple-variable 3-D scientific datasets. A parallel splatting algorithm with both static object space and static image space decomposition was designed and implemented on the Cray T3D using the shmem library. An asynchronous image compositing scheme which can be fully overlapped with the splatting process reduced the communication overhead to a minimal extend. We also proposed a system architecture for an integrated application/rendering system and presented our implementation of some key components in this architecture.

ParVox is still far from mature as a general purpose visualization tool. We need to improve the end-to-end rendering speed, enhance the functionality, increase its portability, and design the functional pipeline interface.

In the visualization pipeline, the slowest module will be the one that determines the frame rate. In a distributed system with low bandwidth links, such as ISDN or long-haul Internet, the data transmission will be the slowest part, thus it is beneficial to use ERIC to get high compression ratio and reduce the transmission time. However, for low compression ratios, it may be more beneficial in terms of speed to use other compression algorithms, such as zlib used in [25] and RLE with frame differencing used in PGL [5]. They are both loseless algorithms with compression ratio less than 10. These types of compression algorithms result in a better end-to-end performance when using a relatively fast network, such as ATM or Ethernet. Therefore, we also would like to incorporate different types of compression algorithms into the ParVox system and be able to select the proper algorithm based on the network speed and the local workstation speed.

ParVox is still missing some important functionalities in order to make it a more versatile tool, such as slicing, arbitrary clipping planes, and vector field visualization. We also would like to port ParVox to MPI-2 using its one-sided communication API. Once it is complete, we can define a generic application interface to support the functional pipelining among multiple partitions of an MPP machine or across multiple MPP platforms.

ACKNOWLEDGMENT

The work presented in this paper was jointly sponsored by the National Aeronautics and Space Administration (NASA), High Performance Computing and Communications (HPCC) Program Earth and Space Science (ESS) Project, and the SGI/Cray PATP Program. Access to the Cray T3D computer was provided by Jet Propulsion Laboratory. We would like to thank Yi Chao and Ping Wang for providing us the Atlantic Ocean temperature datasets and the 3-D thermal convective dataset, respectively.

REFERENCE

[1] "AHPCRC Graphics Software: Icol", http://www.arc.umn.edu/gvl-software/icol.html.,1995.

[2] James Blinn, "Models of Light Reflection for Computer Synthesized Pictures", Proceedings of Siggraph 1977.

[3] Emilio Camahort, Indranil Chakravarty, "Integrating Volume Data Analysis and Rendering on Distributed Memory Architectures", Proceedings of 1993 Parallel Rendering Symposium, pp. 89-96.

[4] Brian Corrie and Paul Mackerras, "Parallel Volume Rendering and Data Coherence", Proceedings of 1993 Parallel Rendering Symposium, October 1993, pp. 23-26.

[5] Thomas Crockett, "Design Considerations for Parallel Graphics Libraries", ICASE Report No. 94-49 (NASA CR-194935).

[6] Thomas Crockett, "Beyond the Renderer: Software Architecture for Parallel Graphics and Visualization", Proceedings of First Eurographics Workshop on Parallel Graphics and Visualization, September 1996, pp. 1-15.

[7] T. Todd Elvins, "Volume Rendering on a Distributed Memory Parallel Computer", Proceedings of Visualization '92, 1992.

[8] Chuck Hansen, et. al., "Binary-Swap Volumetric Rendering on the T3D", Proceedings of the Cray Users Group 1995 Spring Conference, March 1995, pp. 61-69.

[9] W. Hibbard and D. Santek, "The VIS-5D System for Easy Interactive Visualization", Proceedings of 1990 IEEE Visualization Conference, pp 28-35.

[10] William M. Hsu, "Segmented Ray Casting for Data Parallel Volume Rendering", Proceedings of 1993 Parallel Rendering Symposium, pp 7-14

[11] A.S. Jacobson, A.L. Berkin, and M.N. Orton, "LinkWinds: Interactive Scientific Data Analysis and Visualization", Communications of the ACM, Vol.37, No.4, April 1994, pp. 42-52.

[12] Greg Johnson and Jon Genetti, "Medical Diagnosis using the Cray T3D", Proceedings of 1995 Parallel Rendering Symposium, pp 70-77

[13] Raju J. Karia, "Load Balancing of Parallel Volume Rendering with Scattered Decomposition", Proceedings of the 1994 Scalable High Performance Computing Conference, Knoxville, TN, May 1994.

[14] Philippe Lacroute and Marc Levoy, "Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation", Proceedings of SIGGRAPH '94,

[15] Philippe Lacroute, "Analysis of a Parallel Volume Rendering System Based on the Shear-Warp Factorization", IEEE Transactions on Visualization and Computer Graphics, Vol. 2, No. 3, September 1996, pp 218-231.

[16] Marc Levoy, "Display of Surfaces from Volume Data", IEEE Computer Graphics and Applications, Vol. 8, No. 3, May, 1988, pp. 29-37.

[17] Kwan-Liu Ma, et. al. "A Data Distributed, Parallel Algorithm for Ray-Traced Volume Rendering", Proceedings of 1993 Parallel Rendering Symposium, pp 15-22

[18] Eric E. Majani. "ERIC: An Algorithm for Efficient Reversible Image Compression", JPL New Technology Report 20141/9777, January 1997.

[19] Roberto Mendoza, et. al. "A Prototype Distributed Visualization System", JPL New Technology Report 20137/9773, http://www-msim.jpl.nasa.gov/HPCC/Vis/report.html, January 1997.

[20] MPI Forum, "MPI-2: Extensions to the Message-Passing Interface", http://www.mcs.anl.gov/mpi/mpi2/mpi2-report/mpi2-report.html.

[21] Ulrich Neumann, "Volume Reconstruction and Parallel Rendering Algorithms: A Comparative Analysis", Ph.D. Dissertation, UNC Chapel Hill, 1993.

[22] Jason Nieh and Marc Levoy, "Volume Rendering on Scalable Shared-Memory MIMD Architectures", Proceedings of 1992 Workshop on Volume Visualization, Oct. 1992, pp. 17-24.

[23] James S. Painter, "Mantle Convection Visualization on the Cray T3D", Proceedings of Visualization '96, October 1996, pp. 409-412.

[24] R. K. Rew and G. P. Davis, "NetCDF: An Interface for Scientific Data Access", IEEE Computer Graphics and Applications, Vol. 10, No. 4, July 1990, pp. 76-82.

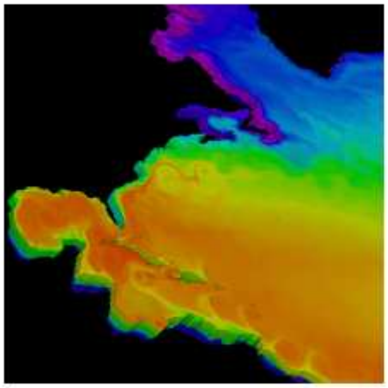
[25] Joe Welling, "Vfleet User's Guide", http://www.psc.edu/general/software/packages/vfleet/development/vfleet.html.

[26] Lee, Westover, "Interactive Volume Rendering", Proceedings of Chapel Hill Workshop on Volume Visualization Workshop, May 1989, pp. 9-16.

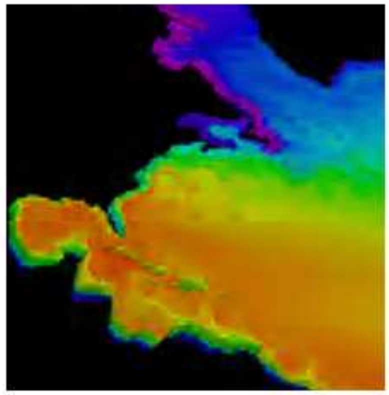
[27] Lee Westover, "Footprint Evaluation for Volume Rendering", Computer Graphics, Proceedings of Siggraph, Vol. 24, No. 4, August 1990, pp. 367-376.

[28] Scott Whitman, Peggy Li, and James Tsiao, "Volume Rendering of Scientific Data on the T3D", Proceedings of the CUG 1996 Spring Conference, March 1996.

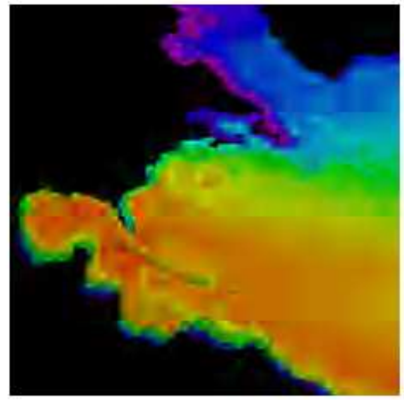
ParVox -- A Parallel Splatting Volume Rendering System for Distributed Visualization
P. Peggy Li, Scott Whitman, Roberto Mendoza, James Tsiao



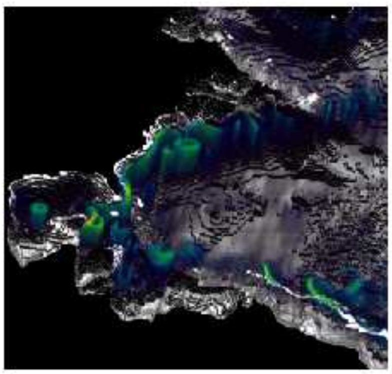
(a) Original



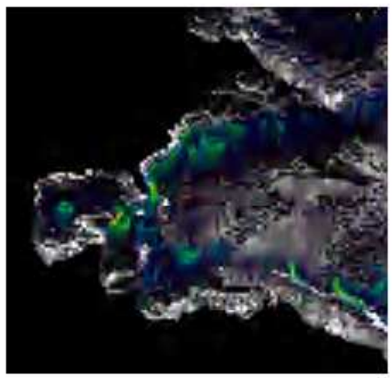
(b) Compressio Ratio:81



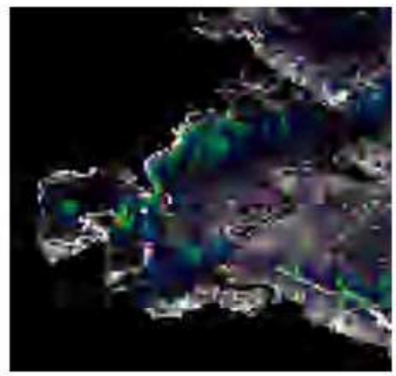
(c) Compression Ratio: 135



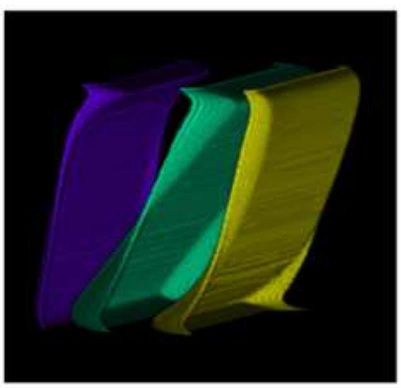
(d) Original



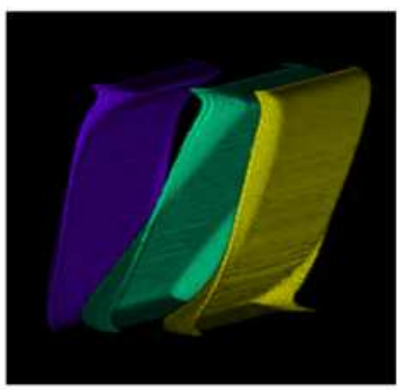
(e) Compression Ratio: 35



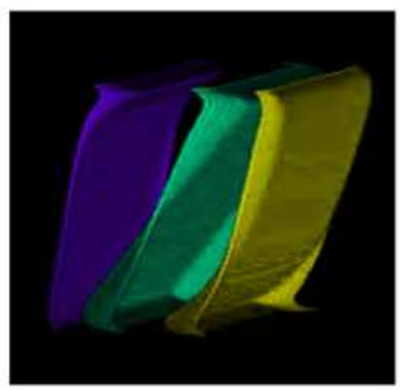
(f) Compression Ratio: 60



(g) Original



(h) Compression Ratio: 87



(i) Compression Ratio: 145

Plate 1. Three ParVox images with differnt compression ratios:
(a)-(c): Atlantic Ocean temperature volume,
(d)-(f): Atlantic Ocean current velocity with ocean floor topological map,
(g)-(i): Temperature convective data showing as isosurfaces.

ParVox -- A Parallel Splatting Volume Rendering System for Distributed Visualization

P. Peggy Li, Scott Whitman, Roberto Mendoza, James Tsiao

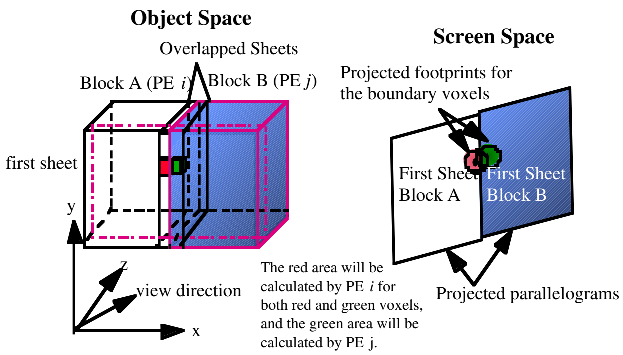


Plate 2. Overlapped splatting of block boundary voxels

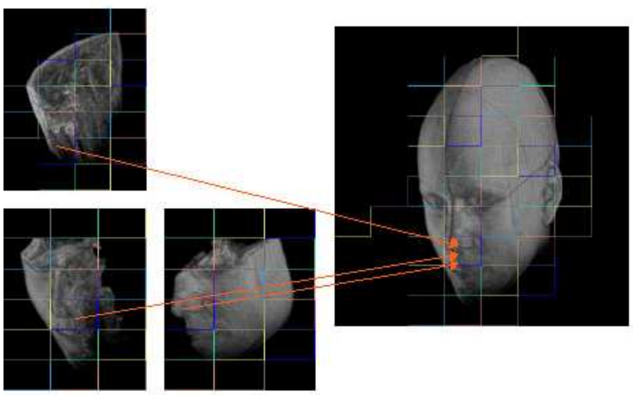


Plate 3: Final Compositing Process

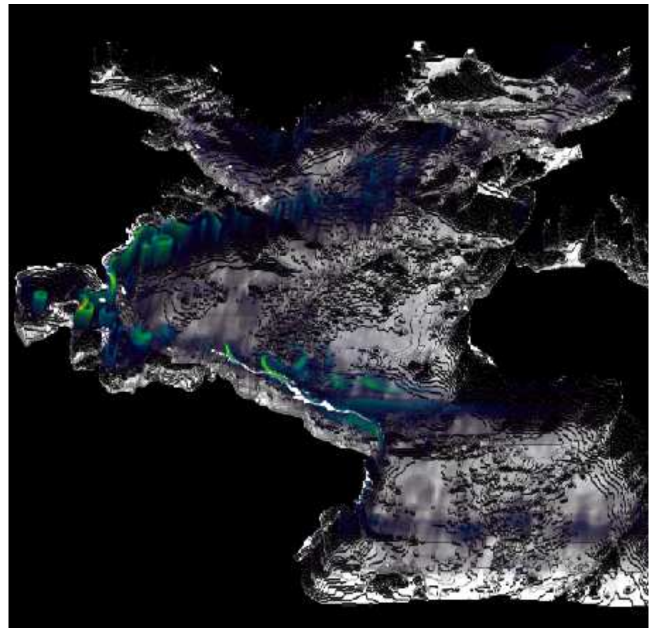


Plate 4. Ocean Velocity with Ocean Floor topological map