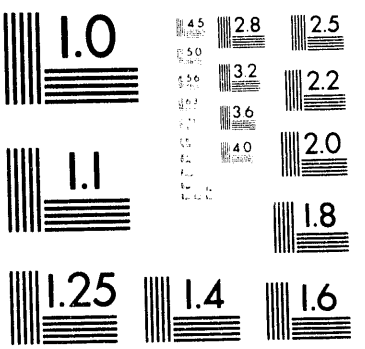

1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6

1 of 1

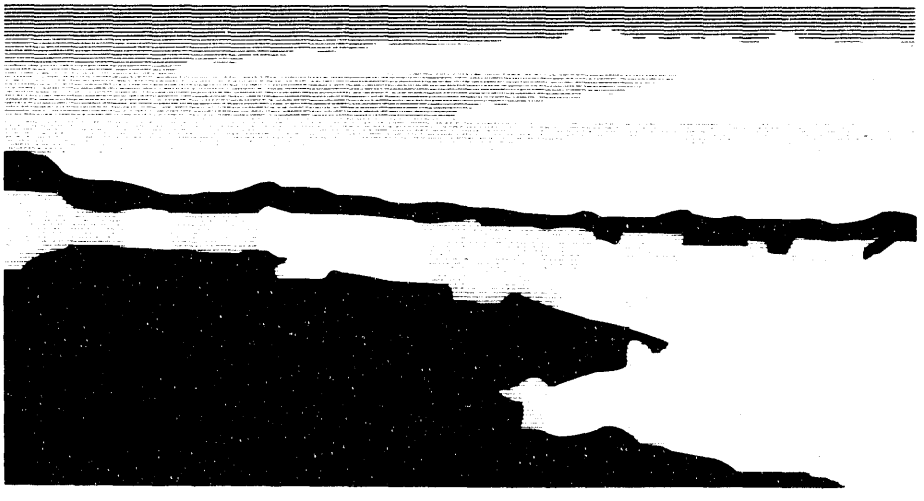
CONF-9310153--1

LA-UR- 93-3138

Title: A DATA DISTRIBUTED, PARALLEL ALGORITH FOR RAY-TRACED VOLUME RENDERING

Author(s): K.-L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh

Submitted to: Parallel Rendering Symp.,
San Jose, CA
October 25, 1993



A DATA DISTRIBUTED, PARALLEL ALGORITHM FOR RAY-TRACED VOLUME RENDERING

Kwan-Liu Ma and James S. Painter
Department of Computer Science
University of Utah
Salt Lake City, Utah 84112
kma@cs.utah.edu
jamie@cs.utah.edu

Charles D. Hansen and Michael F. Krogh
Advanced Computing Laboratory
Los Alamos National Laboratory
Los Alamos, New Mexico 87545
hansen@acl.lanl.gov
krogh@acl.lanl.gov

March 30, 1993

Abstract

This paper presents a divide-and-conquer ray-traced volume rendering algorithm and its implementation on networked workstations and a massively parallel computer, the Connection Machine CM-5. This algorithm distributes the data and the computational load to individual processing units to achieve fast, high-quality rendering of high-resolution data, even when only a modest amount of memory is available on each machine. The volume data, once distributed, is left intact. The processing nodes perform local raytracing of their subvolume concurrently. No communication between processing units is needed during this locally ray-tracing process. A subimage is generated by each processing unit and the final image is obtained by compositing subimages in the proper order, which can be determined *a priori*. Implementations and tests on a group of networked workstations and on the Thinking Machines CM-5 demonstrate the practicality of our algorithm and expose different performance tuning issues for each platform. We use data sets from medical imaging and computational fluid dynamics simulations in the study of this algorithm.

Key Words: Scientific Visualization, Volume Rendering, Distributed Algorithms, Network Computing, Massively Parallel Processing.

1 Introduction

The advance of computing technology has given scientists new opportunities to attempt very large-scale problems that were previously impossible to solve. In addition, the advance of data acquisition instrument technology has provided scientists with very high resolution sensory and monitoring devices to observe the physical world around us at a scale that was previously impossible to attain. The increase in computer processing power, memory capacity, acquisition instrument resolution,

and thus the scale of problems result in data of a size that cannot be handled efficiently by traditional data analysis methods. The use of computer graphics techniques has been a very effective way to convert vast amounts of data to economical visual forms that may convey the most important information in the data sets.

As a consequence of simulating the physical phenomena in three-dimensional space of our ordinary everyday life, a majority of these data sets consist of samples of scalar or vector fields in three spatial dimensions. These are known as volume data sets. Existing volume rendering methods, though capable of making very effective visualizations, are very computationally intensive and thus fail to achieve interactive rendering rates for large data sets. As the size of the data continues to increase and the speed of light physically limits the performance of a single processing element in a computer, multiple computers working in parallel on different data sets or different parts of the same data set offers new promise to unlimited computing power. As a result, in the past two years the development of parallel architectures and algorithms for volume data visualization has been an area of great activity in the research community as well as in industry [20, 22].

Our work was motivated by the following observations: First, volume data sets can be quite large, often too large for a single workstation to hold in memory at once. Moreover, high quality volume renderings normally take minutes to hours on a single processor machine and the rendering time usually grows linearly with the data size. To achieve interactive rendering rates, users often must reduce the original data, which produces poor visualization results. Second, many acceleration techniques and data exploration techniques for volume rendering trade memory for time. Third, motion is one the most effective visualization techniques. An animation sequence of volume visualization normally takes hours to days to generate. Finally, we notice the availability of hundreds of high performance workstations in our computing environment, which are frequently sitting idle for many hours a day, especially after midnight. This lead us to consider ways to distribute the increasing amount of data as well as the time-consuming rendering process to the tremendous distributed computing resources available to us.

In this paper, we describe the resulting divide-and-conquer volume rendering algorithm along with its implementations and performance on a set of networked workstations and the Thinking Ma-

chines CM-5. For a homogeneous computing environment, a computing environment with uniformly distributed processing and memory units, this parallel ray-traced volume rendering algorithm evenly distributes data to the computing resources available. Each subvolume is then ray-traced locally and generates a partial image, without the need to communicate with other processing units. These partial images are merged through a parallel compositing algorithm that composites them in the proper order to achieve the correct final image. The communications costs during compositing is small compared to the cost of the volume rendering itself, so a near linear speedup is attained. Data sets from medical imaging and computational fluid dynamics simulations are used for testing this algorithm in both homogeneous and heterogeneous computing environments.

2 Related Work

An increasing number of parallel algorithms and architectures for volume rendering have been developed. The major algorithmic strategy for parallelizing volume rendering is the divide-and-conquer paradigm. The volume rendering problem can be subdivided either in data space or in image space. While data-space subdivision assigns the computation associated with particular subvolumes to processors, image-space subdivision distributes the computation associated with particular portions of the image space. Data-space subdivision is usually applied to a distributed-memory parallel computing environment. On the other hand, image-space subdivision is simple and efficient for shared-memory multiprocessing. Hybrid methods are also feasible.

Among the parallel architectures developed which are capable of performing interactive volume rendering [8, 11], the Pixel-Planes 5 system [8], an example of a hybrid method, is a heterogeneous multiprocessor graphics system using both MIMD and SIMD parallelism. The hardware consists of multiple i860-based Graphics Processors, multiple SIMD pixel-processors arrays called Renderers, and a conventional 1280×1024-pixel frame buffer, interconnected by a five-gigabit ring network. In [23], variations of parallel volume rendering implemented on the Pixel-Planes 5 system are presented. In one approach, the volume data set is distributed to the Renderers, where shading and syntactic classification are done. Each Graphics Processor is assigned a subimage, performs corresponding ray sampling and requests needed voxel values from the Renderers. A sequence

of high resolution images can be generated at three frames per second for a 128^3 data set using twenty Graphics Processors and eight Renderers. A much more efficient approach, similar to the idea we proposed earlier in [15] and now elaborate in this paper, distributes data as well as ray casting among separate Graphics Processors and reconstructs the ray segments into coherent rays. Incorporating dynamic load balancing, lookup tables and progressive refinement, this approach can render shaded images from 128×128×56 volume data at twenty frames per second.

In the following sections, we survey most recent research results from other algorithmic approaches.

2.1 Montani

Montani et al. [17] propose a hybrid ray-traced method for running on distributed-memory parallel systems like a nCUBE, in which processing nodes are organized into a set of *clusters*, each of them composed of the same number of nodes. The image space is partitioned and a subset of pixels is assigned to each cluster, which will compute pixel values independently. Data to be visualized is replicated in each cluster, and is partitioned among the local memory of the cluster's nodes. A static load balancing strategy based on estimated work load of each processor is used to improve efficiency, and on average a twenty percent speedup in rendering time can be obtained. In addition, a mechanism for preventing deadlock is necessary to handle the dependency between processing nodes in the same cluster. The best efficiency reported by the authors while using a single cluster of 128 nodes is 0.74. However, when increasing the number of clusters, the efficiency drops significantly. For example, using 16 clusters with 8 nodes per cluster, the efficiency reported is only 0.31.

2.2 Nieh

Nieh and Levoy [18] implement ray-traced volume rendering on Stanford DASH Multiprocessors, a scalable shared-memory MIMD machine. Their method employs algorithmic optimizations such as hierarchical opacity enumeration, early ray termination, and adaptive image sampling [13] and the shared-memory architecture providing a single address space allows straightforward implementations. The parallel algorithm distributes volume data in an interleaved fashion among the local

memories to avoid hot spotting. The ray tracing computation is distributed among the processors by partitioning the image plane into contiguous blocks and each processor is statically assigned an image block. Each block is further divided into square image tiles for load balancing purposes. When a processor is done computing its block, instead of waiting, it steals tiles from a neighboring processor's block to keep itself busy. Experiment results show this load balancing scheme cuts the variation of execution times across the the 48 processors used by 90%. Currently, each processor in DASH is a 33 MHz MIPS R3000. Using all 48 processors available, a 416×416-pixel image for a 256×256×226 data set can be generated in subsections; for nonadaptive sampling, the speedup over uniprocessor rendering is 40.

2.3 Schröder

Schröder and Salem [20] describe a multi-pass shear decomposition algorithm implemented on the Connection Machine CM-2 to approximate interactive rotation of volume data. The algorithm distributes data among processors and rotates the volume in place. Forward pointers are used to keep track of the location of a neighbors' piece of data. The parallel computation constructs offered by the CM-2, like the combiner operator **Max**, can produce maximum intensity projected images very efficiently. However, for performing semitransparent rendering with sophisticated shading effects, data transposition must be done which needs to use the general router and thus adds a significant overhead cost.

More recently, Schröder and Stoll describe in their paper [21] a more interesting data-parallel ray-traced volume rendering algorithm that is both more memory efficient and less communications bound than the algorithm presented in [20]. This new technique expioits ray parallelism. They implement the algorithm on both the CM2 and the Princeton Engine, which consists of 2048 16-bit DSP processors arranged in a ring. They describe the ray tracing steps as discrete line drawing. To allow for a SIMD implementation, rays initially enter only the front-most face of the volume and proceed in lock step. Consequently, each sample has the same local coordinates in a voxel. When rays exit the far face, a toroidal shift of the data is performed and new rays are initialized to enter the visible side face of the volume. As a result, the rotation angle selected influences about

10% of the runtime of the algorithm. Tests using a 128^3-voxel data set on both the CM2 from 8K to 32K processors in size and the Princeton Engine of 1024 processors show subsecond rendering time. However, the Princeton Engine performs better because of its simpler communication system to facilitate nearest neighbor shifts.

2.4 Vézina

Vézina, et al. [22] implement a multi-pass algorithm similar to Schröder's on MP-1, which is a massively data-parallel SIMD computer with a two-dimensional array of processing elements (PEs). Their algorithm, based on work done by Catmull and Smith [2], and Hanrahan [10], converts both three-dimensional rotation and perspective transformations into only four one dimensional shear/scale passes, compared to Schröder's eight-pass rotation algorithm composed exclusively of shear operations. Volume transposition is then performed to localize data access. MP-1 provides a global router which allows efficient moving of data between PEs. On a 16K-PE MP-1, a 128×128-pixel volume rendered image of a 128^3-voxel data can be generated in subseconds. However, it seems that if either a smaller number of PEs or larger data sets are used, the data transposition time can degrade the performance significantly.

3 A Divide-and-Conquer Algorithm

Parallel processing is essentially a divide-and-conquer approach to problem solving. Thus the idea behind our algorithm is very simple: divide the data up into smaller subvolumes distributed to multiple computers, render them separately and locally, and combine the resulting images in an incremental fashion. While multiple computers are available, the memory demands on each computer are modest since each computer need only hold a subset of the total data set. This approach can be used to render high resolution data sets in an environment, for example, with many midrange workstations (*e.g.* equipped with 16MB memory) on a local area network. Many scientific and engineering computing environments have an abundance of such workstations which could be harnessed for volume rendering provided that the memory usage on each machine is reasonable.

3.1 Ray-traced Volume Rendering

The starting point of our algorithm is the volume ray-traced technique presented by Levoy [12]. An image is constructed in *image order* by casting rays from the eye through the image plane and into the volume of data. One ray per pixel is generally sufficient, provided that the image sample density is higher than the volume data sample density. Using a discrete rendering model, the data volume is sampled at evenly spaced points along the ray, usually at a rate of one to two samples per voxel. At each sample point $S(i)$ on the ray, a color $C(i)$ and an opacity $\alpha(i)$ are computed using trilinear interpolation from the data values at each of the eight nearest voxels. Here we assume that $C(i)$ is pre-multiplied by its opacity.

The color is assigned by applying a shading function such as the Phong lighting model. A color map is often used to assign colors to the raw data values. The normalized gradient of the data volume can be used as the surface normal for shading calculations. The opacity is derived by using the interpolated voxel values as indices into an opacity map. Sampling continues until the data volume is exhausted or until the accumulated opacity reaches a threshold cut-off value. The final image value corresponding to each ray is formed by compositing, front-to-back, the colors and opacities of the sample points along the ray. Considering $S(i)$ as a *pair* $[C(i), \alpha(i)]$, the color/opacity compositing based on Porter and Duff's **over** operator [19] for two consecutive samples $S(i)$ and $S(j)$ can be described as:

$$S(i) \textbf{ over } S(j) \quad = \quad S(i) + (1 - \alpha(i))S(j);$$

and the composited color is $C(i) + (1 - \alpha(i))C(j)$. Thus the contribution of n samples along the ray for a pixel p on the image plane is

$$S(p) = S(1) \textbf{ over } (S(2) \textbf{ over } (S(3) \textbf{ over } (S(4)...S(n))), \tag{1}$$

and the corresponding color and opacity values are

$$C(p) = \sum_{i=1}^{n} C(i) \prod_{j=1}^{i-1} (1 - \alpha(j)) \quad (2)$$

$$\alpha(p) = 1 - \prod_{i=1}^{n} (1 - \alpha(i)) \quad (3)$$

It is easy to verify that the **over** is *associative*; that is,

$$a \textbf{ over } (b \textbf{ over } c) = (a \textbf{ over } b) \textbf{ over } c.$$

The associativity of the **over** operator allows us to break a ray up into segments, process the sampling and compositing of each segment independently, and combine the results from each segment via a final compositing step. This is the basis for our parallel volume rendering algorithm. For example, using the associativity of **over** we can rewrite equation 1 above as follows:

$$S(p) = (S(1) \textbf{ over } S(2) \textbf{ over } \cdots S(k)\ ) \textbf{ over } (S(k+1) \textbf{ over } S(k+2) \textbf{ over } \cdots S(n)\ )$$

A simple example illustrates the operations involved. Suppose we break a ray into two segments, segment 1 the front-half ray and segment 2 the back-half ray. Applying Equation 2, the color of the ray segment 1 is

$$C_f = \sum_{i=1}^{\frac{n}{2}} C(i) \prod_{j=1}^{i-1} (1 - \alpha(j)),$$

and the color of the ray segment 2 is

$$C_b = \sum_{i=\frac{n}{2}+1}^{n} C(i) \prod_{j=\frac{n}{2}+1}^{i-1} (1 - \alpha(j)).$$

According to the **over** operation, the color of the full ray is

$$C = C_f + (1 - \alpha_f) C_b$$

where

$$\alpha_f = 1 - \prod_{j=1}^{\frac{n}{2}}(1-\alpha(j))$$

Therefore, we can derive the composite as

$$\begin{aligned}
C &= C_f + (1-\alpha_f)C_b \\
&= C_f + (\prod_{j=1}^{\frac{n}{2}}(1-\alpha(j)))C_b \\
&= C_f + (\prod_{j=1}^{\frac{n}{2}}(1-\alpha(j)))\sum_{i=\frac{n}{2}+1}^{n} C(i)\prod_{j=\frac{n}{2}+1}^{i-1}(1-\alpha(j)) \\
&= \sum_{i=1}^{\frac{n}{2}} C(i)\prod_{j=1}^{i-1}(1-\alpha(j)) + \sum_{i=\frac{n}{2}+1}^{n} C(i)\prod_{j=1}^{i-1}(1-\alpha(j)) \\
&= \sum_{i=1}^{n} C(i)\prod_{j=1}^{i-1}(1-\alpha(j))
\end{aligned}$$

which gives us back Equation 2.

3.2 Data Subdivision/Load Balancing

The divide-and-conquer algorithm requires that we partition the input data into subvolumes. There are many ways to partition the data; the only requirement is that an unambiguous front-to-back ordering can be determined for the subvolumes to establish the required order for compositing subimages. Ideally we would like each subvolume to require about the same amount of computation. In practice, this is generally not something that we can always control well. For example, if the viewpoint is known and fixed, we could partition the volume in a manner that minimizes the overlap between the images resulting from the subvolumes. This will reduce the cost of the merging since compositing need only be applied where subimages overlap as shown later. For an animation sequence, this technique can not be applied since the viewpoint changes with each frame. We can also partition the volume based on an estimation of the distribution of the amount of computation within the volume by preprocessing the volume to identify high gradient regions or empty regions. In addition, we may partition and distribute the volume according to the performance of individual computers when using a heterogeneous computing environment.

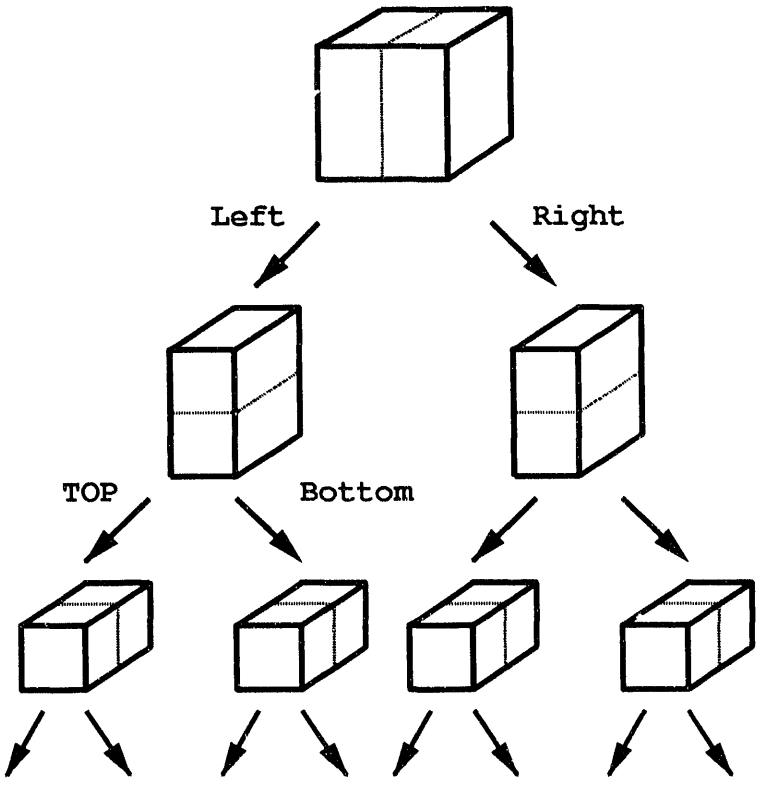


Figure 1: k-Dtree Subdivision of a Data Volume

The simplest method is probably to partition the volume along planes parallel to the coordinate planes of the data. Again, if the viewpoint is fixed and known when partitioning the data, the coordinate plane most nearly orthogonal to the view direction can be determined and the data can subdivided into "slices" orthogonal to this plane. When orthographic projection is used, this will tend to produce subimages with little overlap. If the view point is not known, or if perspective projection is used, it is better to partition the volume equally along *all* coordinate planes. This can be accomplished using a k-D tree structure [1], with alternating binary subdivision of the coordinate planes at each level in the tree as indicated in Figure 1. As we will discuss shortly, this structure provides a nice mechanism for image compositing.

As shown in Figure 2, when a volume of grid points (voxels) is evenly subdivided into, for example, two subvolumes, each subvolume may contain half of the total grid points. Note that each voxel is located at a corner of the grid. Consequently, those ray samples that lie in the cut boundary region (the dotted region) are lost. If the view vector is parallel to the cut plane, a black

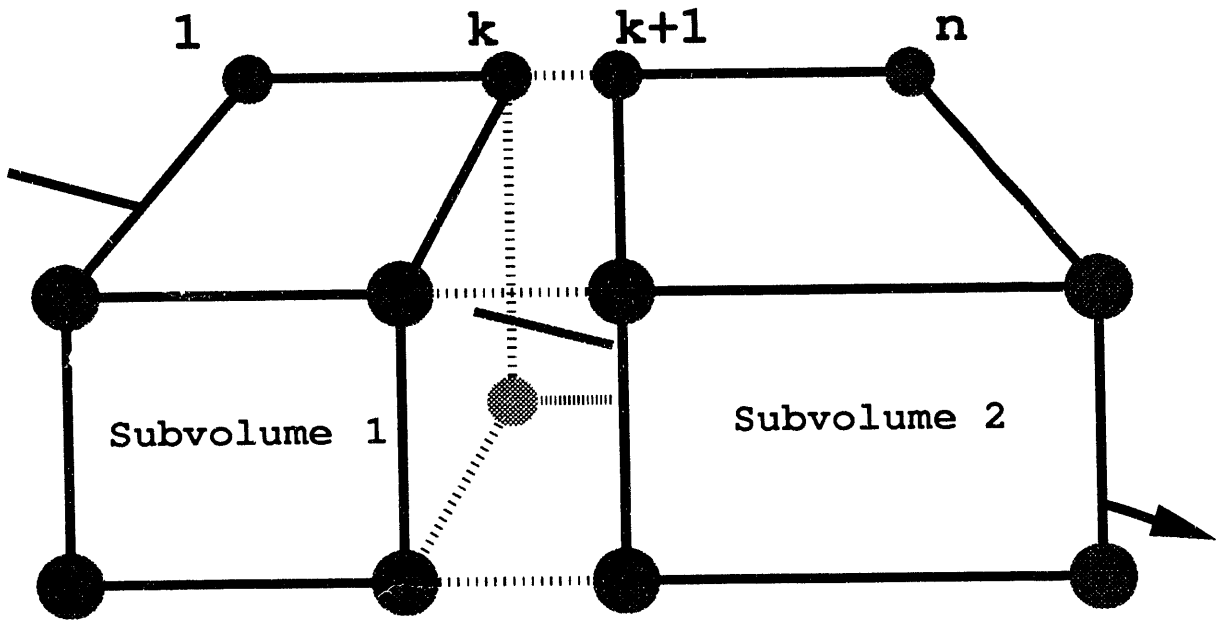


Figure 2: Volume Boundary Replication.

strip will appear at each cut boundary in the composited image. In order to avoid this problem, we need to replicate one layer of the boundary grid at each subvolume so the composited ray-casting image does not drop out features originally in the volume. For the case shown in Figure 2, one possible arrangement is that Subvolume 1 includes layer *1* to layer *k* and Subvolume 2 includes layer *k* to layer *n*; that is, in Subvolume 2, layer *k* is replicated.

3.3 Parallel Rendering

We use ray-casting based volume rendering. Each computer can perform raytracing independently; that is, there is no data communication required during the subvolume rendering. All subvolumes are rendered using an identical view position and only rays within the image region covering the corresponding subvolume are cast and sampled. Since we sample along each ray at a predetermined interval, consistent sampling locations must be ensured for all subvolumes so we can reconstruct the original volume. As shown in Figure 3, for example, the location of the first sample $S_2(1)$ on the ray shown in Subvolume 2 should be calculated correctly so that the distance between $S_2(1)$ and $S_1(n)$ is equivalent to the predetermined interval. Otherwise, small features in the data might be lost or enhanced in an erroneous way.

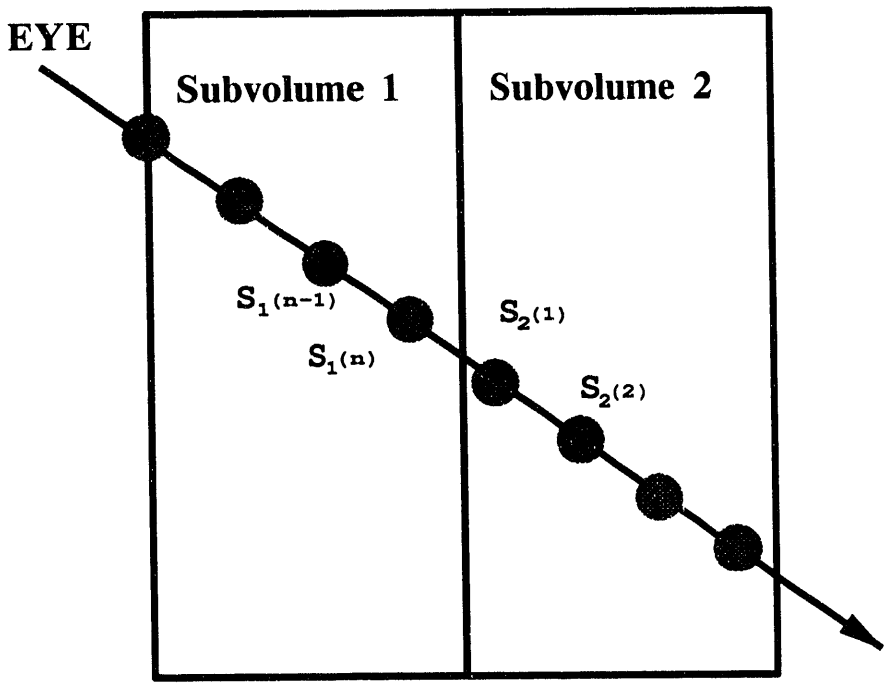


Figure 3: Correct Ray Sampling.

3.4 Image Composition

The final step of our algorithm is to merge ray segments and thus all partial images into the final total image. In order to merge, we need to store not only the color at each pixel but also the accumulated opacity there. As described earlier, the rule for merging subimages is based on the **over** compositing operator. When all subimages are ready, they are composited in a front-to-back order. For a straightforward one-dimensional data partition, this order is also straightforward. When using the k-D tree structure, this front-to-back image compositing order can then be determined hierarchically by a recursive traversal of the k-D tree structure, visiting the "front" child before the "back" child. This is similar to well known front-to-back traversals of BSP-trees [7, 6] and octrees [5, 16]. In addition, the hierarchical structure provides a natural way to accomplish the compositing in parallel: sibling nodes in the tree may be processed concurrently.

The actual compositing can be done in a totally sequential manner such that the computer with the front-most subimage sends its image to the computer with the next-front-most subimage, their composite is sent to the next computer, and so on, until the final total image is obtained. A slightly more efficient way is to do binary compositing. A naive approach is to pair up computers

in order of compositing. Each disjoint pair produces a new subimage. Thus after the first stage, we are left with the task of compositing only $\frac{n}{2}$ subimages. Then we use half the number of the original computers, and pai them up for the next level compositing. Continuing similarly, after $log\ n$ stages, the final image is obtained.

One problem for the above methods is that during the process of image compositing, many computers become idle. At the top of the tree, only one processor is active, doing the final composite for the entire image. When running on a massively parallel computer like CM-5 with thousands of processors, this would significantly affect the overall performance; consequently, the compositing process would become a bottleneck when interactive rendering rates are desired. To avoid this problem, we have generalized the binary compositing method so that every processor participates in all the stages of the compositing process. We call the new scheme *binary-swap* compositing. The key idea is that, at each compositing stage, the two processors involved in a composite operation split the image plane into two pieces and each processor takes responsibility for one of the two pieces.

Figure 5 illustrates the binary-swap compositing algorithm graphically for four processors. When all four computers finish ray-tracing locally, each computer holds a partial image, as depicted in Figure 5 (a). Then each partial image is subdivided into two half-images by splitting along the X axis. In our example, as shown in Figure 5 (b), Computer 1 keeps only the left half-image and sends its right half-image to its immediate-right sibling, which is Computer 2. Conversely, Computer 2 keeps its right half-image, and sends its left half-image to Computer 1. Both computers then composite the half image they keep with the half image they receive. A similar exchange and compositing of partial images is also done between Computer 3 and Computer 4. After the first stage, each computer only holds a partial image that is half the size of the original one. In the next stage, Computer 1 alternates the image subdivision direction. This time it keeps the upper half-image and sends the lower half-image to its second-immediate-right sibling, which is Computer 3, as shown in Figure 5 (c). Conversely, Computer 3 trades its upper half-image for Computer 1's lower half-image for compositing. Concurrently, a similar exchange and compositing between Computer 2 and 4 are done. After this stage, each computers hold only one-fourth of the

```
Initialize image plane to entire image
for(stride=1; stride<nproc; stride * = 2)
{
        partner = self XOR stride;
        Subdivide image plane;
        Exchange image data with partner;
        Composite our part of the remaining
        image plane with partners image data;
}
```

Figure 4: Psuedo Code for Binary Split Compositing

original image. For this example, we are done and each computer sends its image to the display device. Figure 5 (d) shows the final composited image.

Figure 4 illustrates the binary-swap compositing algorithm when the number of processors (**nproc**) is a perfect power of two. We assume that processors are numbered from 0 to **nproc**-1 and that **self** is an integer containing the current processor number. There are $\log_2$(**nproc**) phases, a phase corresponding to each level in the compositing tree. During each phase, each processor exchanges data with its partner which is **stride** away from it. The **stride** value steps from 1 up to **nproc**/2 in powers of 2. In the early phases of the algorithm, each processor is responsible for a large portion of the image area, but the image area is usually sparse since it includes contributions only from a few processors. In later phases, as we move up the compositing tree, the processors are responsible for a smaller and smaller portion of the image area, but the sparsity decreases since an increasing number of processors have contributed image data. At the top of the tree, all processors have complete information for a small rectangle of the image. The final image can be constructed by tiling these subimages onto the display.

In our current implementation, the number of processors (**nproc**) must be a perfect power of two. This simplifies the calculations needed to identify the compositing partner at each stage of the compositing tree and ensures that all processors are active at every compositing phase. The algorithm can be generalized to relax this restriction if the compositing tree is kept as a **full** (but not necessarily complete) binary tree, with some additional complexity in the compositing partner computation and with some processors remaining idle during the first compositing phase.

The binary-swap compositing method has merits which make it particularly suitable for mas-

sively parallel processing. First, while the parallel compositing proceeds, the decreasing image size for sending and compositing makes the overall compositing process very efficient. Next, this method always keeps all processing units busy doing useful work. Finally, it is simple to implement with the use of the k-D tree structure described earlier.

4 Implementation of the Renderer

We have implemented two versions of our distributed volume rendering algorithm: one on a set of networked workstations and another for the Thinking Machines CM-5. Our implementation is composed of three major pieces of code: a data distributor, a renderer, and an image compositor. Currently, the data distributor is a part of the host program which reads data piece by piece from disk and distributes to each machine participating. Alternatively, each node program could read their piece from disk directly.

The renderer implements a conventional ray-traced volume rendering algorithm [12] using a Phong lighting model. Our renderer is a basic renderer and is not highly tuned for best performance. Compared to a performance tuned ray-traced volume rendering program we implemented previously [14], we estimate that the current implementation of the renderer can be further improved in speed by 10-15%. Data dependent optimization methods might in fact affect load balancing decisions by accelerating the progress on some processors more than others. For example, a processor tracing through empty space will probably finish before another processor working on a dense section of the data. We are currently exploring data distribution heuristics that can take the complexity of the subvolumes into account when distributing the data to ensure equal load on all processors.

For shading the volume, surface normals are approximated as local gradients using central differencing. We trade memory for time by precomputing and storing the three components of the gradient at each voxel. As an example, for a data set of size $256 \times 256 \times 256$, more than 200 megabyte are required to store both the data and the precomputed gradients. This memory requirement prevents us from sequentially rendering this data set on most of our workstations.

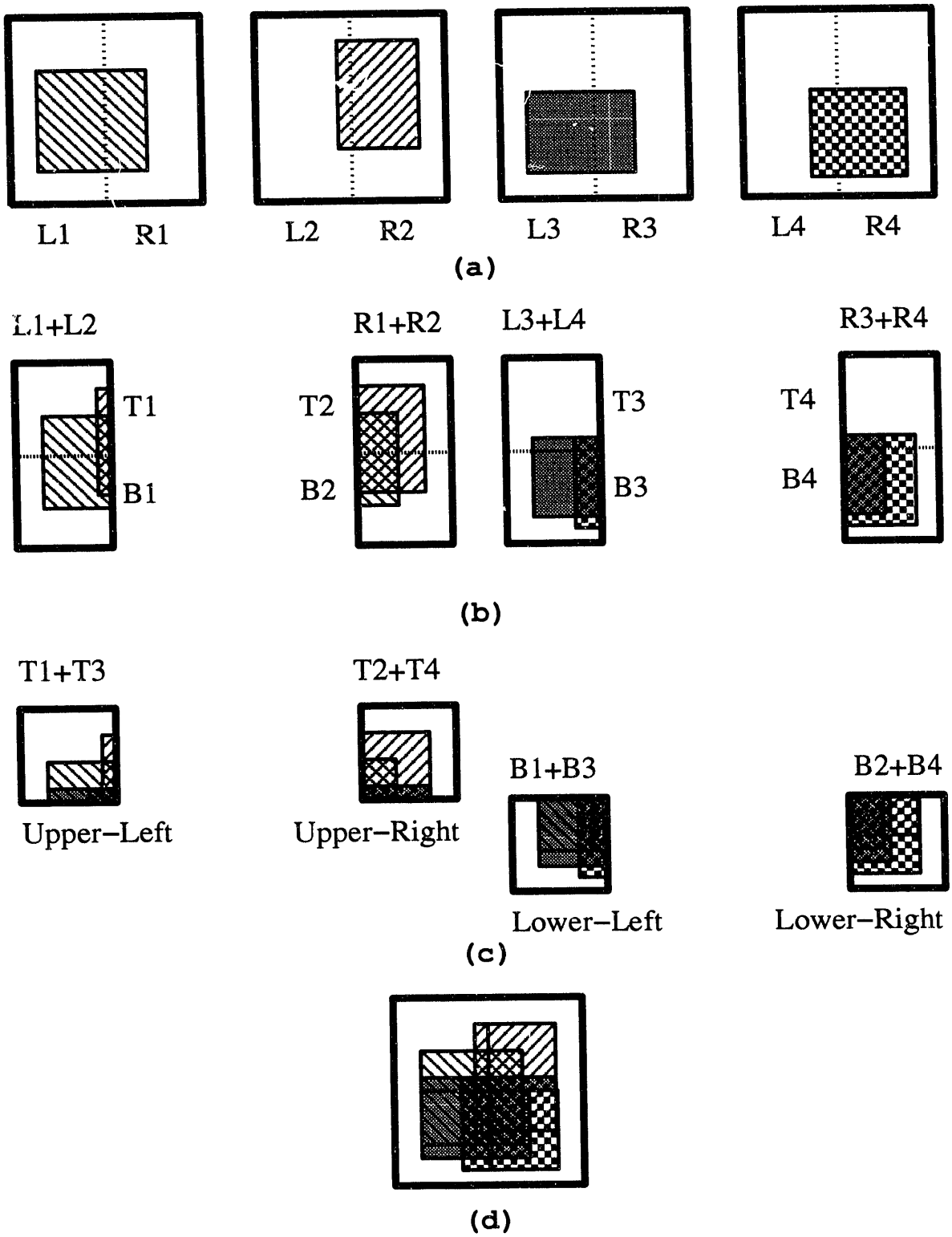


Figure 5: Parallel Compositing Process.

4.1 CM-5 and CMMD

In addition to multiple networked workstations, the Advanced Computing Laboratory (ACL) at Los Alamos National Laboratory has a 1024-node CM-5. There were several goals which lead us to implement the parallel volume renderer on the CM-5. First, we wanted the capability to render very large data sets. Currently, scientific users are generating data sets on the order of 512×512×512 floating point numbers. Their intentions are to increase the resolution of their models to 1K×1K×1K in the near future. Secondly, we wanted to obtain rendering rates as close to real-time as possible. Currently, images are displayed using X Windows which inhibits real-time display. However, in the near future we will have HIPPI framebuffers directly connected to the CM-5 which will support real-time animation rates. Thirdly, we wanted to have a batch animation capability.

4.1.1 CM-5

The CM-5 is a commercially available massively parallel supercomputer built by Thinking Machines[3]. The CM-5 consists of 1024 RISC-based processors (Sparc microprocessors) each with 16MB of local RAM. Each processor also has four 64-bit wide vector units which assist in math coprocessing and contain an additional 4MB RAM each for a total of 32GB of main memory for the entire machine. With four vector units up to 128 operations can be performed by a single instruction. This yields a theoretical speed of 128 GFlops for a 1024-node CM-5.

The 1024 node processors can be divided into partitions whose size must be a power of 2. Each partition is controlled by a partition manager (also Sparc microprocessors). The partition managers are responsible for system administration tasks and executing non-parallel code. A user's program is constrained to operating within a partition.

The CM-5 has three internal high-speed networks: the control network, the data network, and the diagnostic network. The control network is used for data operations, such as broadcasts, global operations, and combining operations. It is also used for synchronization and error handling. All selected processors participate in control network operations. The data network is used for routing data between nodes. The diagnostic network is not available to user programs. High-speed I/O

devices, such as parallel disk arrays, HIPPI network interfaces, and frame buffers are also attached to the CM-5 data network.

The CM-5 supports SIMD and MIMD programming models. The SIMD (Single Instruction, Multiple Data) model performs the same operation on all the selected data elements[1]. For example given an array of numbers, a constant could be added to each number. When using the SIMD model, this operation would logically occur simultaneously on each element of the array. Actual hardware may or may not perform this operation simultaneously on all selected data elements. This would depend on whether or not enough physical processors exist for each element in the array. If there are fewer processors than data elements, then multiple elements are assigned to processors.

The MIMD (Multiple Instruction/Multiple Data) model, divides a task up into a number of subtasks that can run concurrently and independently. Some subtasks can occur in parallel while others might occur serially. For example a set of processors might be used to factor numbers to search for primes. Each processor could be assigned a number to factor asynchronously from the others. When a processor has finished with a given number, it could request another.

Currently, SIMD style programs can be developed using data parallel Fortran (CMF) or data parallel C (C*). MIMD programs are written in C, C++, and Fortran, and use a message passing library (CMMD) for communications and synchronization.

4.1.2 CMMD 3.0

When using the MIMD model, the application developer utilizes the message passing facilities provided by CMMD. The developer must pick either the host/node model or the hostless model. In the host/node model, the user provides explicit communications between the nodes and the host (partition manager). The nodes can communicate with each other as well as with the host. Since the partition manager runs a full UNIX kernel, the host/node model allows access to any software which normally runs on a Sun computer such as system calls, I/O calls, X11 routines, and calls to other specialized libraries[2]. Thus, the application has a component that executes on the

[1]SIMD implies the processors execute in lockstep. Where the processors have their own copy of the program instructions and the instructions don't execute in strict lockstep, the model is known as SPMD (Single Program, Multiple Data)

[2]The nodes run a striped down CMOS kernel which does not provide such facilities.

partition manager (the host) and other components that run on the nodes. The host is responsible for initiating computation on the nodes.

In the hostless mode, an application uses a standard host program supplied by the CMMD library. The host merely initiates the execution of the node programs, and thereafter acts as an I/O server for the nodes. The node program takes advantage of the CMOS kernel which runs on each of the CM-5 nodes. The application developer writes programs which compute and communicate strictly on the nodes and do not explicitly communicate with the host. Each node runs its own code asynchronously from its local memory. It synchronizes with other nodes with explicit instructions (i.e. send/receive messages, participate in a global instruction, etc).

CMMD is the native message passing library supplied by Thinking Machines. CMMD, which sits on top of the CM-5 network interface, provides high level message passing primitives similar to those provided by other message passing libraries. CMMD provides for both synchronous and asynchronous communications, for polled or interrupt driven messages, for global operations among all nodes, and virtual channels (optimized communication between nodes) [4].

4.1.3 CM-5 Implementation

The CM-5 massively parallel implementation of the parallel volume renderer takes advantage of the MIMD programming features of the CM-5. We choose the host/node programming model of CMMD because we wanted the option of using X-windows to display directly from the CM-5. The host program determines which data-space partitioning to use, based on the number of nodes in the CM-5 partition, and sends this information to the nodes. The host then optionally reads in the volume to be rendered and broadcasts it to the nodes. Alternatively, the data can be read directly from the DataVault or Scalable Disk Array into the nodes local memory. The host then broadcasts the opacity/colormap and the transformation information to the nodes. Following this step, the host performs an I/O servicing loop which receives the rendered portions of the image from the nodes.

The node program begins by receiving it's data-space partitioning information and then its portion of the data from the host. It then updates the transfer function and the transform matrices.

Following this, the nodes all execute their own copy of the renderer. They synchronize after the rendering and before entering the compositing phase. Once the compositing is finished, each node has a portion of the image that they then send back to the host.

4.2 Networked Workstations and PVM

The University of Utah Computer Science computing laboratory consists of groups of workstations connected with an Ethernet network. Our goal is to set up a volume rendering facility for handling large data sets and batch animation jobs. We hope that by using many workstations concurrently, the rendering time will decreases linearly and we will be able to render data sets that are too large to render on a single machine. We use PVM (Parallel Virtual Machine) [9], a parallel program development environment, to implement the data communications in our algorithm. PVM allows us to implement our algorithm portably for use on a variety of workstation platforms.

4.2.1 PVM

PVM, supporting an asynchronous message-passing model, is a network-based concurrent computing environment developed at Oak Ridge National Laboratory. It grants the utilization of a heterogeneous network of parallel and serial computers as a parallel virtual machine. To run a program under PVM, the user first executes a daemon process on the local host machine, which in turn initiates daemon processes on all other remote machines used. Then the user's application program (the node program), which should reside on each machine used, can be invoked on each remote machine by a local host program via the daemon processes. Communication and synchronization between these user processes are controlled by the daemon processes, which guarantee reliable delivery. There is some overhead associated with the use of PVM. Direct communications between processing nodes is likely to be faster than communications through the PVM daemon processes. Nevertheless, the portability and ease of use considerations simplified the porting of the CM-5 implementation to the workstation environment and allows us to easily utilize a variety of workstation platforms.

5 Tests

We used three different data sets for our performance measurements. The **head** data set is the now classic UNC Chapel Hill CT head at a size of 128×128×128. The **vessel** data set is a 256×256×128 voxel Magnetic Resonance Angiography (MRA) data set showing the vascular structure within the brain of a patient. The **vorticity** data set is a 256×256×256 voxel CFD data set, computed on the CM-200, showing the onset of turbulence.

In Figure 6, we illustrate the compositing process described in Figure 5, using the images generated with the vessel data set. Each column shows the images from one processor, while the rows are the phases of the compositing algorithm. The final image is displayed at the bottom.

5.1 Tests on the CM-5

We performed multiple experiments on the CM-5 using partition sizes of 32, 64, 128, 256, and 512. When these tests were run, a 1024 partition was not available. As previously noted, the partition sizes must conform to powers of two. The run-time system on the CM-5 provides mechanisms for utilization of all nodes regardless of the partition size. Identical programs were run on the different partitions using the three data sets previously described.

All times are given in seconds. Table 1 shows the results of the volume rendering of the head data. Table 2 shows the results of the volume rendering of the MRA (vessel) data. Table 3 shows the results of the volume rendering of the vorticity data. The times shown are the maximum times for all the nodes for the two steps of the core algorithm: the rendering step and the compositing step.

There is no data communication between the rendering step and the compositing step since the rendering is performed in the k-D data-space decomposed sets and the compositing utilizes these results directly. That is, the compositing step does not require any data exchange from the rendering step. Furthermore, the rendering step does not require any internal communication. The node synchronization step between the rendering and compositing steps is bounded by the node which takes the longest time to render its data partition. For this case, the minimum wait time was less than 0.001 seconds since the last node to finish waits for little time. The maximum wait

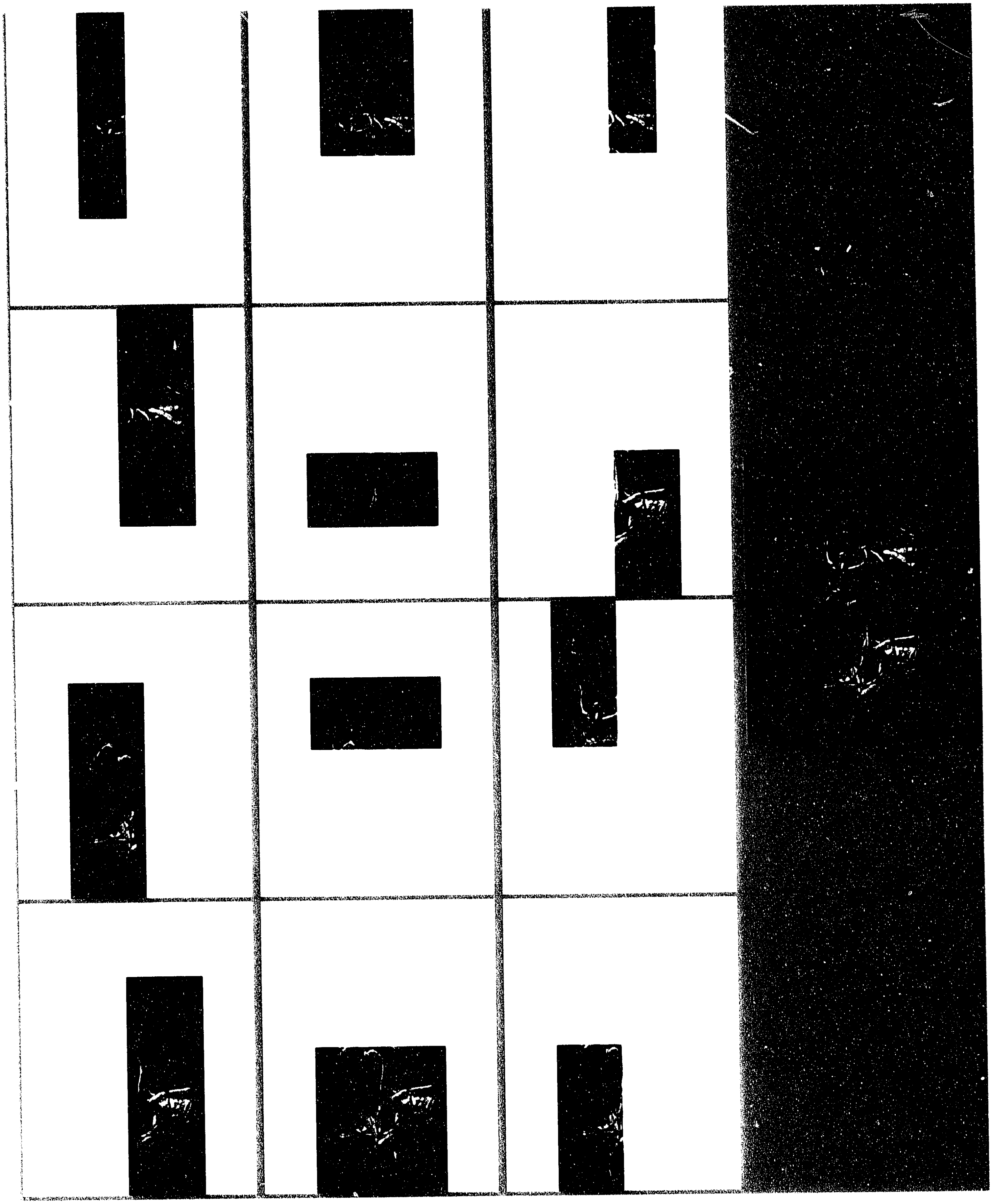


Figure 6: Illustration of the Image Compositing Process Using Actual Images.

size	function	32	64	128	256	512
64 x 64	render	0.5839	0.3723	0.2071	0.1043	0.0593
	composite	0.0165	0.0150	0.0133	0.0113	0.0101
128x128	render	2.3033	1.5393	0.8459	0.4278	0.2223
	composite	0.0576	0.0497	0.0322	0.0325	0.0269
256x256	render	9.2600	6.1558	3.3663	1.7344	0.9536
	composite	0.1679	0.1932	0.1287	0.1090	0.0945
512x512	render	36.3685	24.1807	13.1200	6.7355	3.7107
	composite	0.63320	0.77810	0.47660	0.4029	0.3782

Table 1: CM-5 Results on Head Data Set

size	function	32	64	128	256	512
64 x 64	render	0.4346	0.2627	0.1350	0.0806	0.0454
	composite	0.0097	0.0087	0.0085	0.0086	0.0081
128x128	render	1.6138	0.9500	0.4988	0.2643	0.1390
	composite	0.0303	0.0237	0.0233	0.0213	0.0167
256x256	render	6.4522	3.6532	1.8698	1.0084	0.5193
	composite	0.1146	0.0897	0.0835	0.0741	0.0554
512x512	render	26.0314	14.9057	7.5980	4.1720	2.2034
	composite	0.46060	0.34600	0.3278	0.2931	0.2167

Table 2: CM-5 Results on Vessel Data Set

size	function	32	64	128	256	512
64 x 64	render	0.8038	0.3995	0.2072	0.1116	0.0597
	composite	0.0137	0.0125	0.0101	0.0101	0.0094
128x128	render	3.1446	1.5974	0.8247	0.4086	0.2041
	composite	0.0473	0.0406	0.0300	0.0279	0.0235
256x256	render	12.3345	6.3133	3.2305	1.6158	0.8063
	composite	0.1807	0.1466	0.1108	0.1001	0.0836
512x512	render	48.2005	24.4303	12.697	6.3434	3.1878
	composite	0.71520	0.58100	0.4272	0.3874	0.3310

Table 3: CM-5 Results on Vorticity Data Set

size	function	32	64	128	256	512
64 x 64	composite	0.0137	0.0125	0.0101	0.0101	0.0094
	communication	0.0013	0.0008	0.0006	0.0005	0.0003
128x128	composite	0.0473	0.0406	0.0300	0.0279	0.0235
	communication	0.0030	0.0026	0.0018	0.0012	0.0011
256x256	composite	0.1807	0.1466	0.1108	0.1001	0.0836
	communication	0.0210	0.0075	0.0052	0.0037	0.0027
512x512	composite	0.7152	0.5810	0.4272	0.3874	0.3310
	communication	0.0843	0.0231	0.0181	0.0138	0.0097

Table 4: CM-5 Compositing Communication Times

time for the vorticity data set on a 512 CM-5 partition was 0.588 seconds. The maximum wait time varies depending on both the data set and the viewpoint.

Some processors are assigned data partitions which consist of mainly empty voxels while others have partitions which are comprised of non-empty voxels. The partitions with empty voxels have very little computation to perform whereas the partitions with non-empty voxels perform more work. Similarly, as the viewpoint is rotated, some processors will have fewer ray-samples passing through the voxels which comprise their subset of the data and some will have more. The processors with the most ray-samples passing through non-empty voxels will perform the most computation thereby taking the maximum rendering time. Since there is no communication in the rendering step, one might expect linear speedup when utilizing more processors. As can be seen from the tables, this is not the case due to the load balance problems. The vorticity data set is relatively dense[3] and therefore exhibits nearly linear speedup. The head data set contains many empty voxels which unbalances the load and therefore does not exhibit the best speedup. The load balance affects the wait-time which limits the linearity of the speedup.

The compositing stage requires communication between pairs of nodes to perform the actual compositing. In the case of the vorticity data set, Table 4 shows the combined compositing and communication time and the actual communication component for the different sized CM-5 partitions. As can be seen, the communication time varies from about 10 percent to about 3 percent of the total compositing time. As the image size increases, both the compositing time and the

[3] The vorticity data set contains few empty voxels.

size	function	32	64	128	256	512
	broadcast	89.87	93.516	83.185	94.326	49.157
64 x 64	send image	0.0161	0.0168	0.0187	0.0218	0.0280
128x128	send image	0.0608	0.0615	0.0657	0.0687	0.0734
256x256	send image	0.2406	0.2417	0.2615	0.2470	0.2537
512x512	send image	0.9918	0.96500	0.9645	1.0151	0.9849

Table 5: CM-5 Host Communication Results on Vorticity Data Set

communication time also increase. For a fixed image size, increasing the partition size lowers the communication time because each node contains a proportionally smaller piece of the image.

Looking at the tables, it is easy to see that rendering time dominates the process. It should be noted that this implementation does not take advantage of the CM-5 vector units. All floating point operations are done on the Sparc node floating point units. These are much slower than the vector units and we expect much faster computation rates in the renderer when the vectorized code is completed. The communication times will not be affected.

Table 5 shows the broadcast and image gather times for the vorticity data. The broadcast time includes the time it takes to read the image over NFS at ethernet speeds on a loaded ethernet. While the timings were being gathered for partitions smaller than 512 nodes, the other partitions were also running causing both disk and ethernet contention. Thus, the 512 broadcast time is substantially less than for the smaller partitions. Reading the file from disk dominates this time. The image gather time (send image) is the time it takes for the nodes to send their composited image tiles to the host. As can be seen, the image gather time (send image) is only slightly slower for larger partitions which have more image-tiles. Both of these times will be mitigated by use of the parallel storage (DataVault or Scalable Disk Array) and the use of the HIPPI framebuffer. With the parallel storage, the nodes can load their portion of the data directly. The nodes can also write their sub-images concurrently to the framebuffer via HIPPI.

Figure 7 and Figure 8 show speedup versus number of processors used for the vorticity data set and the head data set. The diagonal line represents linear speed up whereas the four curves show the speedup results obtained for various image resolutions. The speedup was measured for the core algorithm: the rendering step and the compositing step. Speedup is a function of the

32 node run time. The graph demonstrates that for the vorticity data set, our implementation achieves very good speed up for all image sizes except 64×64. The rendering of the 64×64 image exhibits less speedup than larger image sizes due to overhead costs associated with the rendering and compositing steps. In particular, the compositing step showed a speedup of only 1.46 when going from 32 nodes to 512 nodes. For all image resolutions above 64×64, the overall speedup was nearly the same.

For the head data set, the speed up is not nearly as linear. This is due to the load balance issue previously discussed. With this dataset and opacity map, many rays are terminated early since opacity reaches 1.0 where ever the isosurface is encountered. Rendering a 512×512 image size for the head data set, the rendering time varied from 0.2931 to 3.7771 seconds. For the same image size, rendering of the vorticity data set yielded rendering times from 2.5729 to 3.1924 seconds. The more consistent rendering times are indicative of better load balancing.

5.2 Networked Workstations

For our workstation tests, we used a set of 16 high performance workstations. The first three machines where IBM RS/6000-550 workstations equipped with 512 MB of memory. These workstations are rated at 81.8 SPECfp92. The fourth machine was an IBM RS/6000-530 with 384 MB. This is rated at 64.6 SPECfp92. The remaining 12 machines were HP9000/730 workstations, some with 32MB and others with 64 MB. These machines are rated at 86.7 SPECfp92.

The tests on 1, 2 and 4 workstions used only the IBM's. The tests with 8 and 16 used a combination of HP and IBM workstations. It was not possible to assure absolute quiescence on each machine because these machines are in a shared environment with a large shared ethernet and shared files systems. During the period of testing there was a network traffic from network file system activity (NFS) and across-the-net tape backups. The first four nodes were all on the same subnet, while the remaining nodes lie on a different subnet. Thus, we expect the communications performance for the 1, 2 and 4 processors test to be better than for the 8 and 16 process cases.

Table 6 shows the results of the volume rendering of the head data at several image sizes and with 1-16 processors. Table 7 shows the results of the volume rendering of the MRA (vessel) data.

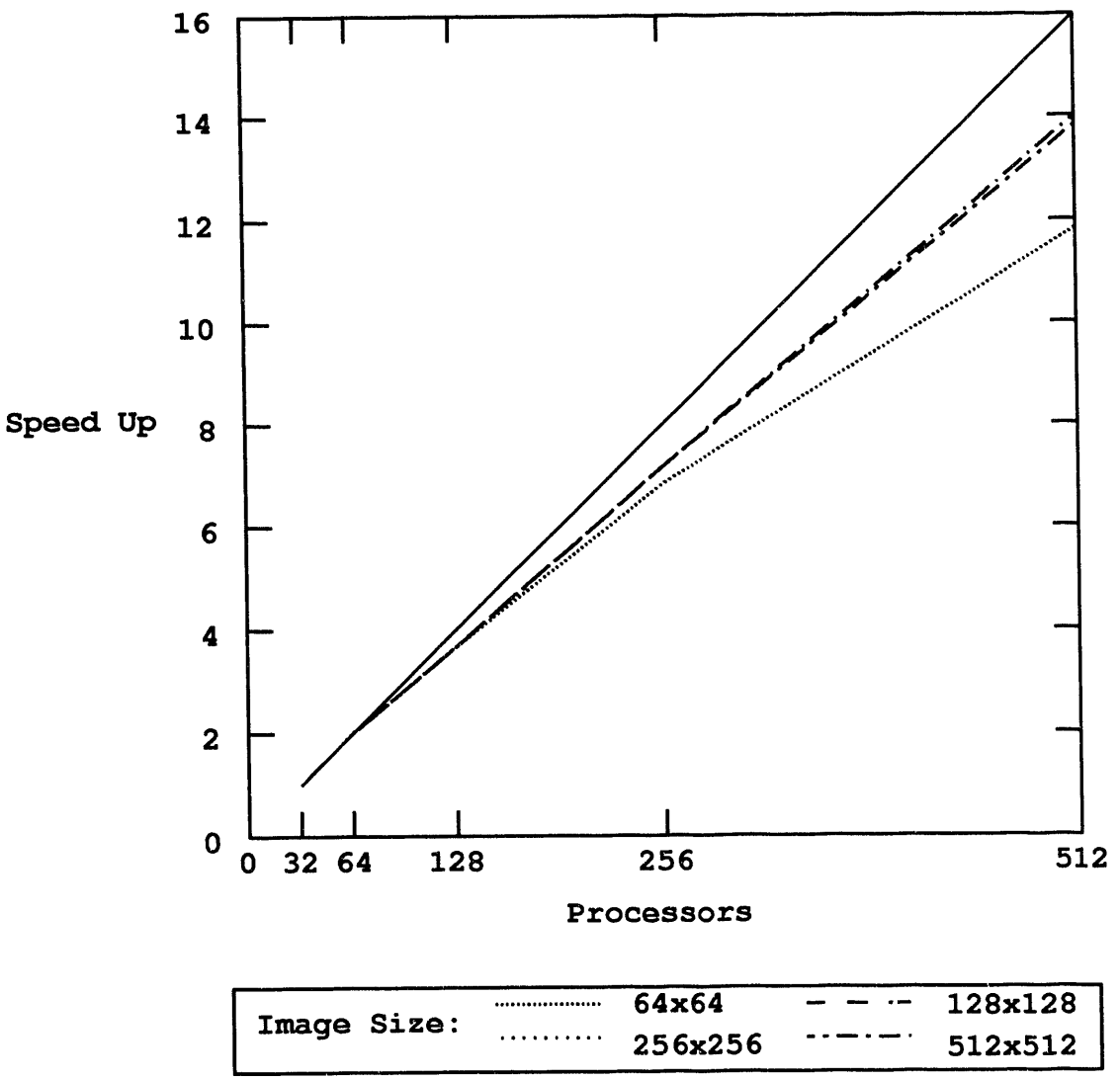


Figure 7: CM-5 Speedup for the Vorticity Data Set

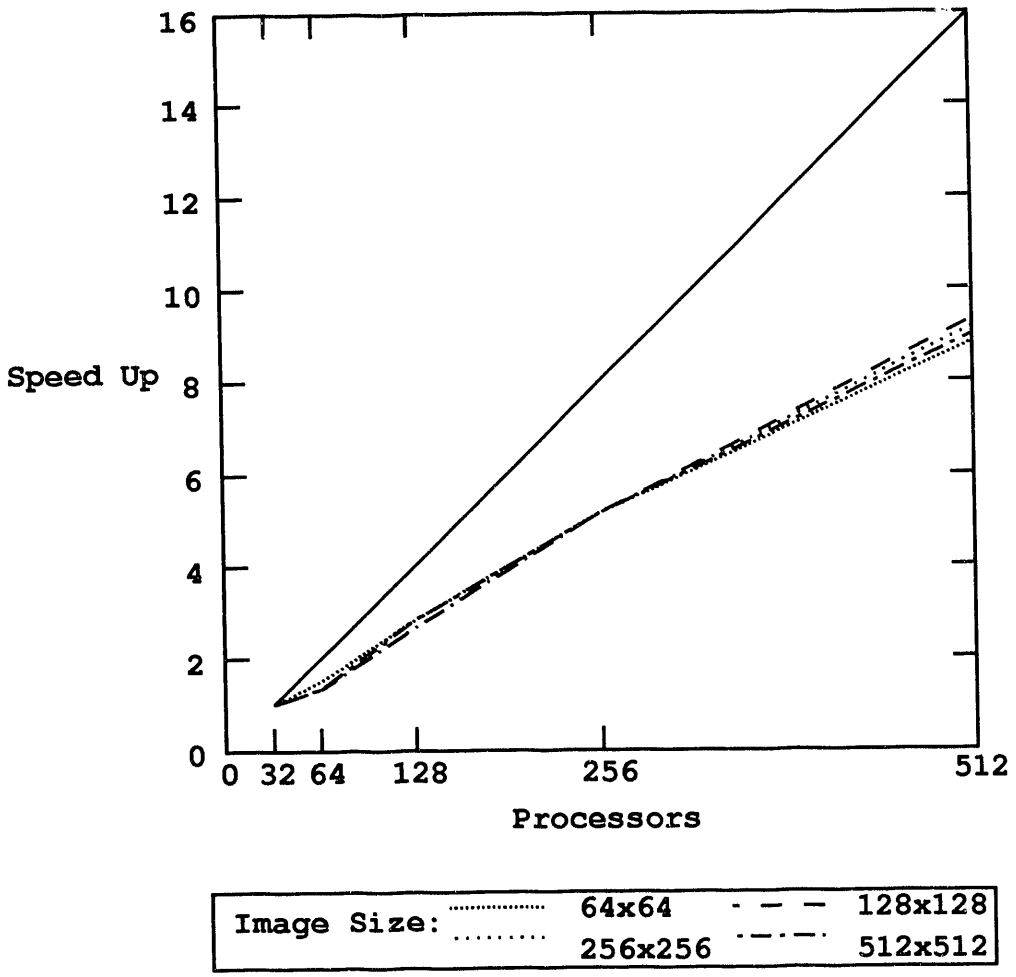


Figure 8: CM-5 Speedup for the Head Data Set

size	function	1	2	4	8	16
64 x 64	render	2.2200	1.2700	1.1030	0.4920	0.3250
	composite	0.0000	0.0420	0.0900	0.0990	0.1570
128x128	render	8.4040	4.9540	4.1890	1.8270	1.1160
	composite	0.0010	0.1240	0.2170	0.2370	4.3490
256x256	render	33.2260	19.9480	17.0840	8.2230	4.3660
	composite	0.0030	0.4200	0.4810	0.6160	8.1090
512x512	render	134.4880	80.5170	70.2450	44.2310	20.9980
	composite	0.0110	2.3730	1.5470	5.5280	13.1470

Table 6: PVM Results on Head Data Set

size	function	1	2	4	8	16
64 x 64	render	2.9790	1.4560	1.1730	0.6840	0.3490
	composite	0.0010	0.0280	0.0740	0.0880	0.1670
128x128	render	11.7850	5.9760	4.7910	2.5460	1.2610
	composite	0.0000	0.0660	0.1240	0.1210	0.2400
256x256	render	44.5380	23.8390	18.2220	9.5590	6.5850
	composite	0.0020	0.2320	0.2940	0.5350	4.3000
512x512	render	183.9460	94.9210	72.7390	38.5960	20.3100
	composite	0.0060	0.7560	1.1350	5.1510	4.5740

Table 7: PVM Results on Vessel Data Set

Table 8 shows the results of the volume rendering of the vorticity data. The times shown are for the two steps of the core algorithm: the rendering step and the compositing step. The times shown are the maximum times in seconds for all the machines. As can be seen, the rendering time dominates the compositing time in most cases.

The rendering component of the algorithm is purely computation: there is no communications. The compositing component does require communications between nodes as illustrated in Table 9. This tables shows the combined compositing and communication time and the component due to communications overhead. Note that communications time is the dominant factor in the compositing costs. These communications costs are highly variable due to the use of the local ethernet shared with hundreds of other machines. Communications costs are expected to drop with higher speed interconnection networks (e.g. FDDI) and on clusters isolated from the larger local area network.

size	function	1	2	4	8	16
	render	5.7130	2.8110	2.5740	1.1550	0.6440
64 x 64	composite	0.0000	0.0440	0.0890	0.1290	4.3200
	render	22.6530	11.4960	10.6490	5.2760	2.5670
128 x 128	composite	0.0010	0.1150	0.1470	0.2740	4.2890
	render	93.2510	47.7220	44.5610	23.8740	11.5390
256 x 256	composite	0.0060	0.4160	0.5860	4.7460	3.6730
	render	371.7900	180.4650	174.7770	71.6980	44.1010
512 x 512	composite	0.0160	1.4000	1.9560	9.1320	5.2020

Table 8: PVM Results on Vorticity Data Set

size	function	1	2	4	8	16
64 x 64	composite	0.0000	0.0440	0.0890	0.1290	4.3200
	communications	0.0000	0.0430	0.0870	0.1280	4.3200
128x128	composite	0.0010	0.1150	0.1470	0.2740	4.2890
	communications	0.0000	0.1080	0.1380	0.2730	4.2870
256 x 256	composite	0.0060	0.4160	0.5860	4.7460	3.6730
	communications	0.0000	0.4040	0.5750	4.7390	3.6680
512 x 512	composite	0.0160	1.4000	1.9560	9.1320	5.2020
	communications	0.0000	1.3250	1.9240	9.1 30	5.1740

Table 9: PVM Compositing Communication Times

In a *shared* computing environment, there are many factors that we have no control over that are influential to our algorithm. For example, an overloaded network and other users' processes competing with our rendering process for CPU and memory usage could greatly degrade the performance of our algorithm. Improved performance could be achieved by carefully distributing the load to each computer according to data content, and the computer's performance as well as its average usage by other users.

The tables above exclude the data distribution and image gather times. These times varied greatly, due to the variable load on the shared ethernet. The data distribution times varied from 17 seconds to 150 seconds while the image gather times varied from an average of .06 seconds for a 64×64 image to a high of 8 seconds for a 512×512 image.

6 Conclusions

We have presented a parallel algorithm for volume rendering on distributed memory parallel machines or a set of interconnected workstations. The algorithm divides both the computation and memory load across all processing nodes and can therefore be used to render data sets that are too large to fit into the memory system of a single uniprocessor. A parallel compositing method was developed to combine the independently rendered results from each processor. The algorithms were implemented on the Thinking Machines CM-5 massively parallel supercomputer and on a network of scientific workstations using PVM. The CM-5 implementation showed good speedup characteristics out to the largest available partition size of 512 nodes. Only a small fraction of the total rendering time was spent in communications, indicated the success of the parallel compositing algorithm.

Several directions appear ripe for further work. The host data distribution, image gather, and display times are bottlenecks on the current CM-5 implementation. These bottlenecks can be aliviated by exploiting the parallel I/O capabilities of the CM-5. Render and compositing times on the CM-5 can also be reduced significantly by taking advantage of the vector units available at each processing node. We are hopeful that real time rendering rates will be achievable at medium to high resolution with these improvements.

Performance of the distributed workstation implementation could be further improved by better load balancing. In a heterogeneous environment with shared workstations, linear speedup is difficult. A simple approach is to do static load balancing. The data subdivision can be done unevenly, taking into account the predicted capacity on each machine to try to balance the load. Alternatively, the data can be subdivided into a larger number of equal sized subvolumes and the more capable machines can be assigned more than one subvolume. The later approach has the advantage that it can be generalized to a dynamic load balancing approach: divide the data into many subvolumes and assign them to processors in a demand driven fashion. Each processors asks the master host for more data when it has completed rendering of the previous subvolume. The finer subdivision of the data volumes improves load balancing during rendering at the cost of some additional compositing time due to additional levels in the compositing tree.

Acknowledgments

This work has been supported in part by NSF/ACERC and an IBM grant for Scientific Visualization. The Medical Imaging Laboratory at the University of Utah provides the MRA data set. The vorticity data set was provided by Shi-Yi Chen of T-Div at Los Alamos National Laboratory. David Rich, of the ACL, and Burl Hall, of Thinking Machines, helped tremendously with the CM-5 timings.Professor Bob Kessler, Jay Lepreau and the Center for Software Sciences group at Utah provide the HP workstations for some of our performance tests. Thanks go to Elena Driskill for comments on a draft of this paper.

References

[1] BENTLEY, J. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM 18*, 8 (September 1975), 509–517.

[2] CATMULL, E., AND SMITH, A. R. 3-D Transformations of IMages in Scanline Order. *Computer Graphics 14*, 3 (1980), 279–285.

[3] CORPORATION, T. M. The connection machine CM-5 technical summary, 1991.

[4] CORPORATION, T. M. Cmmd reference manual; preliminary documentation for version 3.0 beta, February 1993.

[5] DOCTOR, L., AND TORBORG, J. Display Techniques for Octree-Encoded Objects. *IEEE Comput. Graphics and Appl. 1* (July 1981), 29–38.

[6] FUCHS, H., ABRAM, G., AND GRANT, E. D. Near Real-Time Shade Display of Rigid Objects. In *Proceedings of SIGGRAPH 1983* (1983), pp. 65–72.

[7] FUCHS, H., KEDEM, Z. M., AND NAYLOR, B. F. On Visible Surface Generation by A Priori Tree Structures. In *Proceedings of SIGGRAPH 1980* (1980), pp. 58–67.

[8] FUCHS, H., POULTON, J., EYLES, J., GREER, T., GOLDFEATHER, J., ELLSWORTH, D., MOLNAR, S., TURK, G., TEBBS, B., AND ISRAEL, L. Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories. *Computer Graphics 23*, 3 (July 1989), 111–120.

[9] GEIST, G., AND SUNDERAM, V. Network-based Concurrent Computing on the PVM System. *Concurrency: Practice and Experience 4*, 4 (June 1992), 293 312.

[10] HANRAHAN, P. Three-Pas Affine Transforms for Volume Rendering. *Computer Graphics 24*, 5 (1990). Special issue on San Diego workshop on Volume Rendering.

[11] KAUFMAN, A. Volume Rendering Architectures. *Volume Visualization Algorithms and Architectures* (August 1990), 189–198. ACM SIGGRAPH Course Notes.

[12] LEVOY, M. Display of Surfaces from Volume Data. *IEEE Computer Graphics and Applications* (May 1988), 29–37.

[13] LEVOY, M. Efficient Ray Tracing of Volume Data. *ACM Transactions on Graphics 9*, 3 (July 1990), 245–261.

[14] MA, K.-L., COHEN, M., AND PAINTER, J. Volume Seeds: A Volume Exploration Technique. *The Journal of Visualization and Computer Animation 2* (1991), 135–140.

[15] MA, K.-L., AND PAINTER, J. S. Parallel Volume Visualization on Workstations. *Computers and Graphics 17*, 1 (1993).

[16] MEAGHER, D. Geometric Modeling Using Octree Encoding. *Comput. Graphics and Image Process. (USA) 19* (June 1982), 129–147.

[17] MONTANI, C., PEREGO, R., AND SCOPIGNO, R. Parallel Volume Visualization on a Hypercube Architecture. In *1992 Workshop on Volume Visualization* (1992), pp. 9–16. Boston, October 19-20.

[18] NIEH, J., AND LEVOY, M. Volume Rendering on Scalable Shared-Memory MIMD Architectures. In *1992 Workshop on Volume Visualization* (1992), pp. 17–24. Boston, October 19-20.

[19] PORTER, T., AND DUFF, T. Compositing Digital Images. *Computer Graphics (Proceedings of SIGGRAPH 1984) 18*, 3 (July 1984), 253–259.

[20] SCHRÖDER, P., AND B., S. J. Fast Rotation of Volume Data on Data Parallel Architectures. In *Proceedings of Visualization'91* (October 1991), pp. 50–57.

[21] SCHRÖDER, P., AND STOLL, G. Data Parallel Volume Rendering as Line Drawing. In *1992 Workshop on volume Visualization* (1992), pp. 25–31. Boston, October 19-20.

[22] VÉZINA, G., FLETCHER, P. A., AND ROBERTSON, P. K. Volume Rendering on the MasPar MP-1. In *1992 Workshop on volume Visualization* (1992), pp. 3–8. Boston, October 19-20.

[23] YOO, T., NEUMANN, U., FUCHS, H., PIZER, S., CULLIP, T., RHOADES, J., AND WHITAKER, R. Direct Visualization of Volume Data. *IEEE Computer Graphics and Applications* (July 1992), 63–71.

DATE

FILMED

11/17/93

END