A Data Distributed, Parallel Algorithm for Ray-Traced Volume Rendering

Kwan-Liu Ma[†], James S. Painter[‡], Charles D. Hansen[§], Michael F. Krogh[§]
[†]ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, Virginia 23681
[‡]Department of Computer Science, University of Utah, Salt Lake City, Utah 84112
[§]Advanced Computing Laboratory, Los Alamos National Laboratory, Los Alamos, New Mexico 87545
kma@icase.edu, jamie@cs.utah.edu, hansen@acl.lanl.gov, krogh@acl.lanl.gov

Abstract

This paper presents a divide-and-conquer ray-traced volume rendering algorithm and a parallel image compositing method, along with their implementation and performance on the Connection Machine CM-5, and networked workstations. This algorithm distributes both the data and the computations to individual processing units to achieve fast, high-quality rendering of high-resolution data. 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. Test results on the CM-5 and a group of networked workstations demonstrate the practicality of our rendering algorithm and compositing method.

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

1 Introduction

Existing volume rendering methods, though capable of making very effective visualizations, are very computationally intensive and therefore fail to achieve interactive rendering rates for large data sets. 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 of 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. 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 parallel volume rendering algorithm and a image compositing method along with their implementations and performance on the CM-5 and networked workstations. For a homogeneous computing environment, a computing environment with uniformly distributed processing and memory units, this parallel 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 in the proper order through a new parallel compositing algorithm to achieve the correct final image. Our test results on both the homogeneous and heterogeneous computing environments are promising, and expose different performance tuning issues for each environment.

0-8186-4920-8/93 $3.00

2 Related Work

An increasing number of parallel architectures and algorithms 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, the Pixel-Planes 5 system [5] 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 [22], variations of parallel volume rendering implemented on this system are presented. One approach similar to the idea we proposed earlier in [11] 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 20 frames per second. In the following sections, we survey most recent research results from other algorithmic approaches.

2.1 Montani

Montani *et al.* [13] 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 the 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 [14] 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 [9]. 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 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 256^3 data set can be generated in subsections; for nonadaptive sampling, the speedup over uniprocessor rendering is 40.

2.3 Schröder

Schröder and Stoll [18] develop a data-parallel ray-traced volume rendering algorithm that exploits ray parallelism. They describe the ray tracing steps as discrete line drawing. This algorithm is both more memory efficient and less communications bound than an algorithm introduced earlier by the first author [17]. They have implemented this algorithm on both the Connection Machine CM-2 and the Princeton Engine, which consists of 2048 16-bit DSP processors arranged in a ring. 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.

2.4 Vézina

Vézina, et al. [21] implement a multi-pass algorithm similar to Schröder's on MP-1, which is a massively data-parallel SIMD computer with a 2D array of processing elements (PEs). Their algorithm, based on work done by Catmull and Smith [2], and Hanrahan [7], converts both 3D rotation and perspective transformations into only four 1D 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

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 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-tracing technique presented by Levoy [8]. 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 on the ray, a color and an opacity are computed using trilinear interpolation from the data values at each of the eight nearest voxels.

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. The color/opacity compositing is based on Porter and Duff's **over** operator [16]. 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.

3.2 Data Subdivision/Load Balancing

The divide-and-conquer algorithm requires that we partition the input data into subvolumes. There are many ways

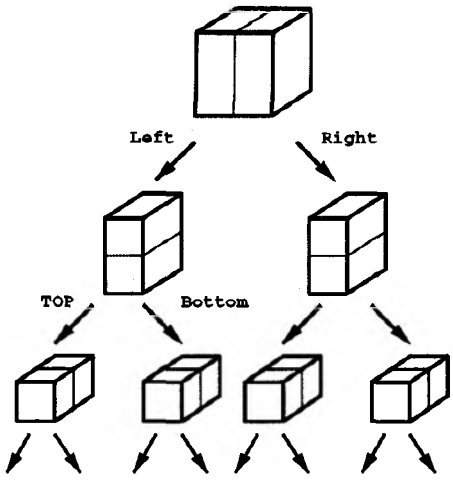

Figure 1: k-Dtree Subdivision of a Data Volume

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.

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 shown later, 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 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

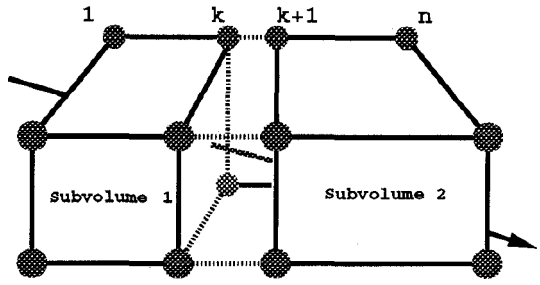

Figure 2: Volume Boundary Replication.

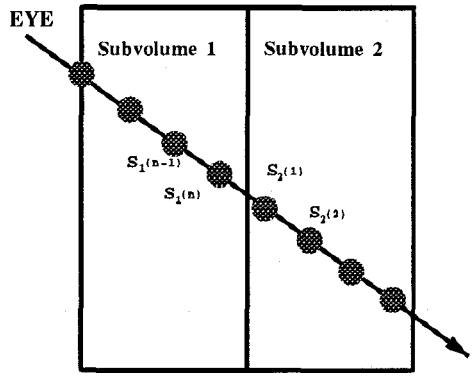

Figure 3: Correct Ray Sampling.

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.

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 [4] and octrees [3]. In addition, the hierarchical structure provides a natural way to accomplish the compositing in parallel: sibling nodes in the tree may be processed concurrently.

A naive approach for merging the partial images is to do binary compositing. By pairing 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 pair 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 compositing process 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.

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.

Figure 4 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 (a). Then each partial image is subdivided into two half-images by splitting along the X axis. In our example, as shown in (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 done between Computer 3 and 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 (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 computer holds only one-fourth of the original image. For this example, we are done and each computer sends its image to the display device. The final composited image is shown in (d). It has been brought to our attention that a similar merging algorithm has been developed independently by Mackerras [12].

In our current implementation, the number of processors

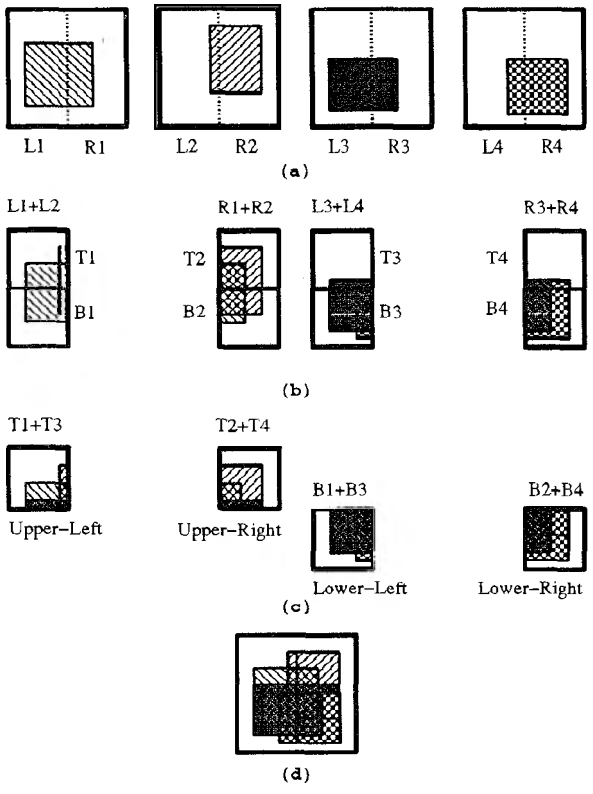

Figure 4: Parallel Compositing Process.

(**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.

4 Implementation of the Renderer

We have implemented two versions of our distributed volume rendering algorithm: one on the CM-5 and another on groups of networked workstations. Our implementation is composed of three major pieces of code: a data distributor, a renderer, and an image compositor. Currently, the data distributor runs as a single "host" process that determines the partitioning of the data set, reads the data set piece by piece from disk and distributeds it to a set of "node" processes that perform the actual rendering and compositing. Alternatively, each node program could read their piece from disk directly.

The renderer implements a conventional ray-traced volume rendering algorithm [8] using a Phong lighting model [15]. 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 [10], we estimate that the current implementation of the renderer can be further improved in speed by 10%-15%. In fact, data dependent optimization methods might affect load balancing decisions by accelerating the progress on some processors more than others. For example, a pro-

cessor 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.

4.1 CM-5 and CMMD 3.0

The CM-5 is a massively parallel supercomputer which supports both the SIMD and MIMD programming models [19]. The CM-5 in the Advanced Computing Laboratory at Los Alamos National Laboratory has 1024 nodes, each of which is a Sparc microprocessor with 32 MB of local RAM and four 64-bit wide vector units. 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 nodes can be divided into partitions whose size must be a power of two. A user's program is constrained to operating within a partition. Our CM-5 implementation of the parallel volume renderer takes advantages of the MIMD programming features of the CM-5. MIMD programs use CMMD, a message passing library for communications and synchronization, which supports either a hostless model or a host/node model [20].

We chose 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. Finally, 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 its 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 step, 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 for display.

4.2 Networked Workstations and PVM 2.4.2

Unlike a massively parallel supercomputer dedicating uniform and intensive computing power, a network computing environment provides nondedicated and scattered computing cycles. Thus, using a set of high performance workstations connected by an Ethernet, our goal is to set up a volume rendering facility for handling large data sets and batch animation jobs. That is, we hope that by using many workstations concurrently, the rendering time will decrease linearly and we will be able to render data sets that are too large to render on a single machine. Note that real-time rendering is generally not achievable in such environment.

We use PVM (Parallel Virtual Machine) [6], a parallel program development environment, to implement the data communications in our algorithm. PVM allows us to portably implement our algorithm for use on a variety of workstation platforms. 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.

A host/node model has also been used. As a result, the way it has been implemented is very similar to that of CM-5's. In fact, the only distinct difference between the workstation's and CM-5's implementation (source program) is the communication calls. For most of the basic communication functions, PVM 2.4.2 and CMMD 3.0 have one-to-one equivalence.

5 Tests

We used three different data sets for our tests. The *vorticity* data set is a $256 \times 256 \times 256$ voxel CFD data set, computed on a CM-200, showing the onset of turbulence. The *head* data set is the now classic UNC Chapel Hill CT head at a size of $128 \times 128 \times 128$. The *vessel* data set is a $256 \times 256 \times 128$ voxel Magnetic Resonance Angiography (MRA) data set showing the vascular structure within the brain of a patient. Plate 1 illustrates the compositing process described in Figure 4, using the images generated with this *vessel* data set. Similarly, 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 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. All times are given in seconds. For the *vorticity* data set, we show complete timing results in Table 1 and the speedup graph in Figure 5. The times shown are the broadcast time (data) and the maximum times for all the nodes for the two steps of the core algorithm: the rendering step (rend) and the compositing step (comp), followed by the actual communication component (comm) in the compositing step and lastly the image gathering time (send). Note that the speedup was measured for the core algorithm and it is a function of the 32 node running time. Due to limited space, for the *head* and *vessel* data sets, we show only the corresponding speedup graphs in Figure 6 and 7, respectively.

Looking at Table 1, 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. We expect much faster computation rates in the renderer when the vectorized code is completed. As there is no communication in the rendering step, one might expect linear speedup when utilizing more processors. As can be seen from the three speedup graphs, this is not always the case due to the load balance problems. The *vorticity* data set is relatively dense (i.e. it contains few empty voxels) and

size	opt	32	64	128	256	512
	data	89.87	93.516	83.185	94.326	49.157
	rend	0.8038	0.3995	0.2072	0.1116	0.0597
	comp	0.0137	0.0125	0.0101	0.0101	0.0094
	comm	0.0013	0.0008	0.0006	0.0005	0.0003
64^2	send	0.0161	0.0168	0.0187	0.0218	0.0280
	rend	3.1446	1.5974	0.8247	0.4086	0.2041
	comp	0.0473	0.0406	0.0300	0.0279	0.0235
	comm	0.0030	0.0026	0.0018	0.0012	0.0011
128^2	send	0.0608	0.0615	0.0657	0.0687	0.0734
	rend	12.334	6.3133	3.2305	1.6158	0.8063
	comp	0.1807	0.1466	0.1108	0.1001	0.0836
	comm	0.0210	0.0075	0.0052	0.0037	0.0027
256^2	send	0.2406	0.2417	0.2615	0.2470	0.2537
	rend	48.200	24.430	12.697	6.3434	3.1878
	comp	0.7152	0.5810	0.4272	0.3874	0.3310
	comm	0.0843	0.0231	0.0181	0.0138	0.0097
512^2	send	0.9918	0.96500	0.9645	1.0151	0.9849

Table 1: CM-5 Results on the *Vorticity* Data Set

therefore exhibits nearly linear speedup. On the other hand, both the *head* and the *vessel* data sets contain many empty voxels which unbalance the load and therefore do not exhibit the best speedup. Figure 5 demonstrates that for the *vorticity* data set, our implementation achieves very good speedup 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.

The broadcast time includes the time it takes to read the data over NFS at Ethernet speeds on a loaded Ethernet. The broadcast time for the 512-node case is substantially less than for the smaller partitions because while the timings were being gathered for partitions smaller than 512 nodes, the other partitions were also running other jobs causing both disk and Ethernet contention. The image gathering time (send) is the time it takes for the nodes to send their composited image tiles to the host. As can be seen, the image gathering time 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 and the use of the HIPPI frame buffer.

5.2 Networked Workstations

For our workstation tests, we used a set of 32 high performance workstations. The first four machines were IBM RS/6000-550 workstations equipped with 512 MB of memory. These workstations are rated at 81.8 SPECfp92. The next 12 machines were HP9000/730 workstations, some with 32 MB and others with 64 MB. These machines are rated at 86.7 SPECfp92. The remaining 16 machines were Sun Sparc-10/30 workstations equipped with 32 MB, which are rated at 45 SPECfp92. The tests on one, two and four workstations used only the IBM's. The tests with eight and 16 used a combination of the HP's and IBM's. The 16 Sun's were used for the tests on 32. It was not possible to assure absolute quiescence on each machine because they are in a shared environment with a heavily used Ethernet and large files systems. During the period of testing there was a network traffic from NSF activity and across-the-net tape backups. The four IBM's were all on the same subnet, while the remaining nodes lie on different subnets. Thus, we expect the communication performance for the one, two and four machines to be better than for the eight or more.

In a heterogeneous environment, it is less meaningful to use speedup graphs to study the performance of our algorithm and implementation. Thus in Figure 8, 9 and 10, for the rendering step and the compositing step, varying the number of workstations and the image size, we display the maximum times from the tests on the *vorticity*, *head* and *vessel* data sets, respectively. Note that we use a *log* scale along the y axis. The solid lines show the time for both steps and the dotted lines show the time for the rendering step only.

In a *shared* computing environment, the communication costs are highly variable due to the use of the local Ethernet shared with hundreds of other machines. 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. Moreover, communications costs are expected to drop with higher speed interconnection networks (e.g. FDDI) and on clusters isolated from the larger local area network.

Unlike the CM-5's results, tests on workstations show that the communication component is the dominant factor in the compositing costs. This can be seen by comparing the solid lines with the dotted lines in the graphs. On the average, communication takes about 97% of the overall compositing time. However, while using eight or fewer workstations, the rendering time still dominates the compositing time in most cases. Again, the significant performance degradation for rendering smaller images is due to the overhead costs associated with the rendering and compositing steps. These graphs 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. Preliminary results with PVM 3.1 indicate much lower communications costs.

6 Conclusions

We have presented a parallel volume ray-tracing algorithm for a massively parallel computer or a set of interconnected workstations. The algorithm divides both the computation and memory load across all processors 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 (*binary-swap*) compositing method was developed to combine the independently rendered results from each processor. The *binary-swap* compositing method has merits which make it particularly suitable for massively 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 processors busy doing useful work. Finally, it is simple to implement with the use of the k-D tree structure described earlier.

The algorithm has been implemented on both the CM-5 and a network of scientific workstations. The CM-5 imple-

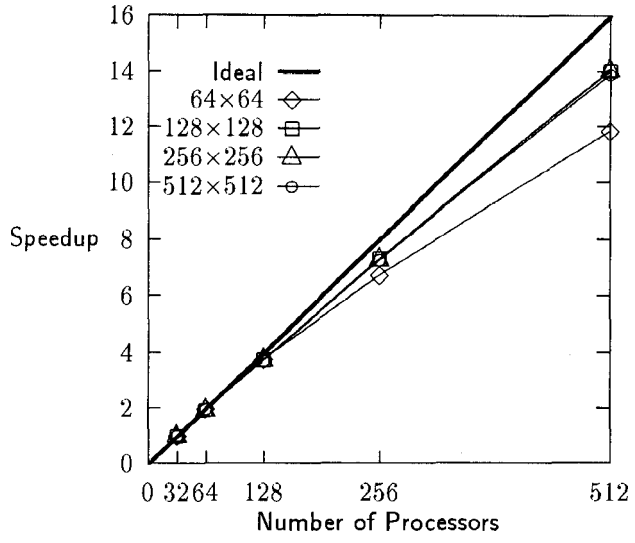

Figure 5: CM-5 Speedup for the *Vorticity* Data Set.

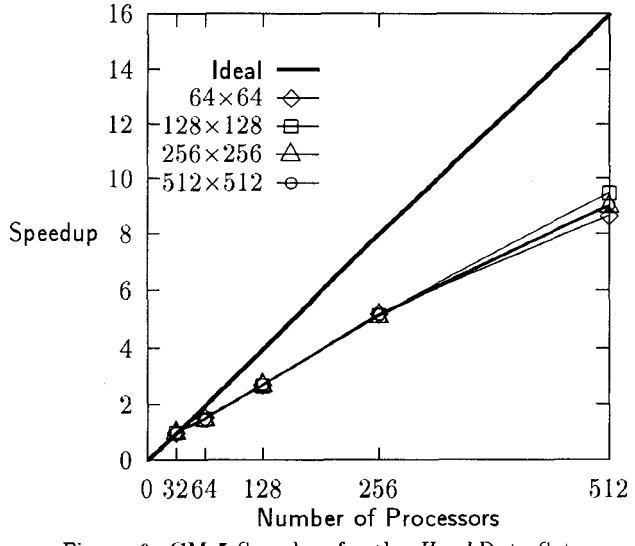

Figure 6: CM-5 Speedup for the *Head* Data Set.

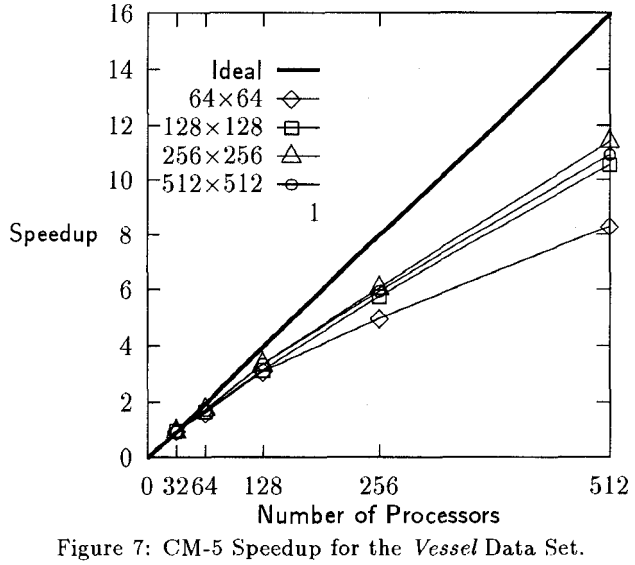

Figure 7: CM-5 Speedup for the *Vessel* Data Set.

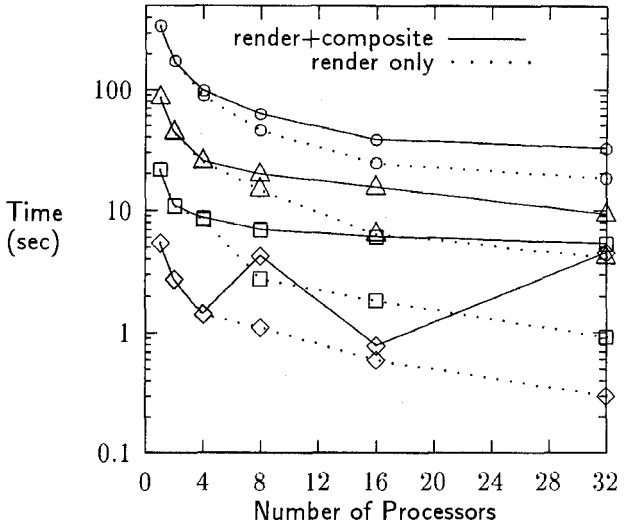

Figure 8: PVM Results on the *Vorticity* Data Set.

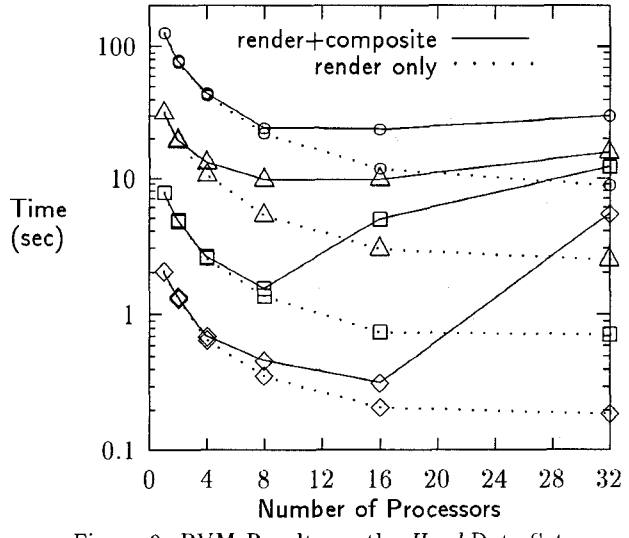

Figure 9: PVM Results on the *Head* Data Set.

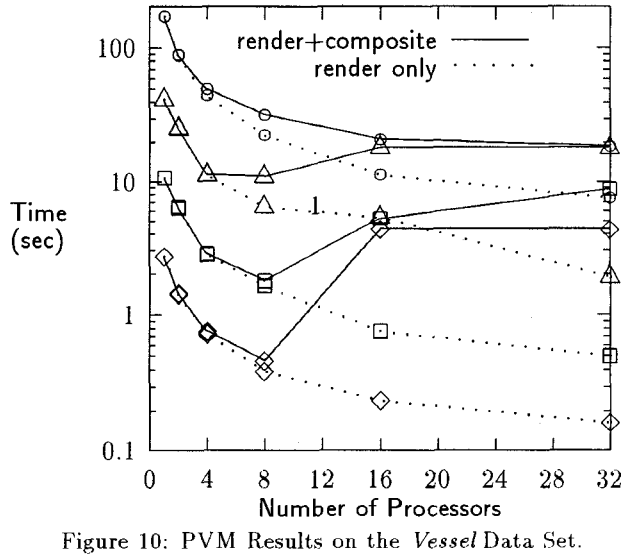

Figure 10: PVM Results on the *Vessel* Data Set.

mentation 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, indicating the success of the parallel compositing method. 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 alleviated by exploiting the parallel I/O capabilities of the CM-5. Rendering 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 for 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. The finer subdivision of the data volumes would improve load balancing during rendering at the cost of some additional compositing time due to more levels in the compositing tree.

Acknowledgments

The MRA vessel data set was provided by the MIRL at the University of Utah. 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. The Alpha_1 and CSS groups at the University of Utah provided the workstations for our performance tests. Thanks go to Elena Driskill for comments on a draft of this paper. This work has been supported in part by NSF/ACERC and NASA/ICASE.

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

[4] FUCHS, H., ABRAM, G., AND GRANT, E. D. Near Real-Time Shade Display of Rigid Objects. In *Proceedings of SIGGRAPH '83* (1983), pp. 65–72.

[5] 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.

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

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

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

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

[10] 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.

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

[12] MACKERRAS, P. A Fast Parallel Marching Cubes Implementation on the Fujitsu AP1000. Tech. Rep. TR-CS-92-10, Department of Computer Science, Australian National University, 1992.

[13] 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.

[14] 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.

[15] PHONG, B. Illumination for Computer-Generated Pictures. *Commun. ACM 18*, 6 (June 1975), 311–317.

[16] PORTER, T., AND DUFF, T. Compositing Digital Images. In *Proceedings of SIGGRAPH '84* (July 1984), pp. 253–259.

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

[18] 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.

[19] THINKING MACHINES CORPORATION. *The Connection Machine CM-5 Technical Summary*, 1991.

[20] THINKING MACHINES CORPORATION. *CMMD Reference Manual; Preliminary Documentation for Version 3.0 Beta*, February 1993.

[21] 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.

[22] 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.