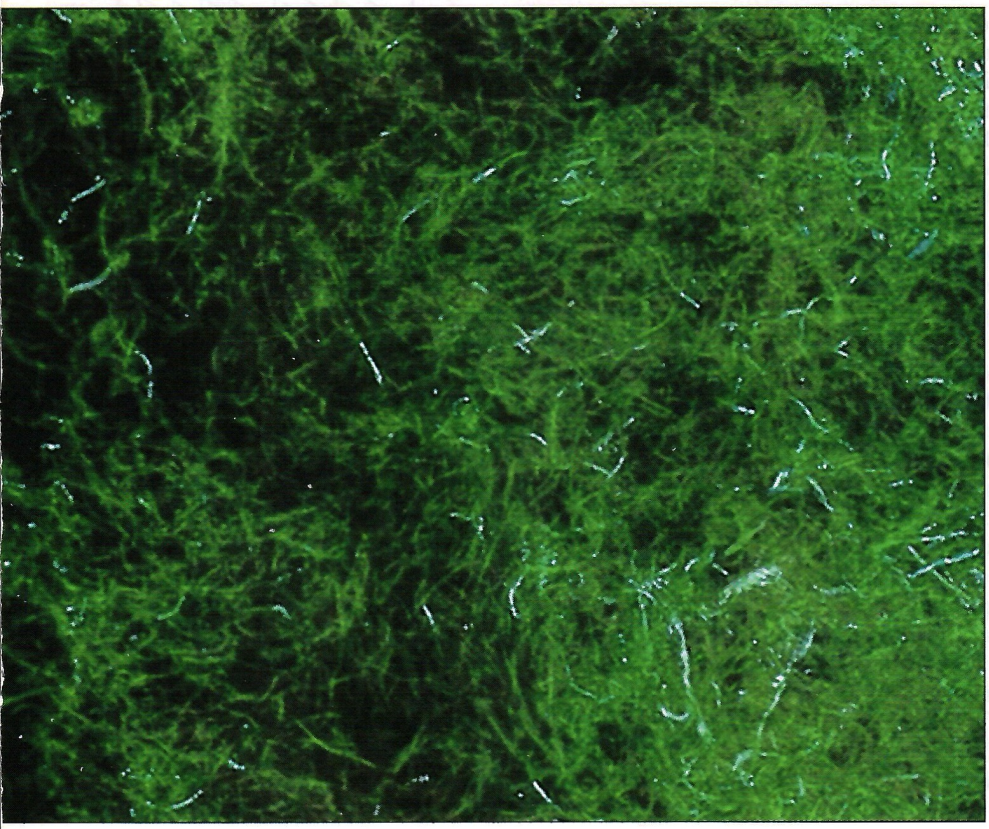
Parallel Volume Rendering Using Binary-Swap Compositing



Kwan-Liu Ma
NASA Langley Research Center
James S. Painter
University of Utah
Charles D. Hansen and Michael F. Krogh
Los Alamos National Laboratory

This algorithm distributes data and computations to individual processing nodes for rendering subimages, then composites the final image with a method that uses all nodes at all times.

Existing volume rendering methods, though capable of very effective visualizations, are computationally intensive and therefore fail to achieve interactive rendering rates for large data sets. Although computing technology continues to advance, computer processing power never seems to catch up to the increases in data size.

Several observations motivated the work we describe here. First, volume data sets can be quite large, often too large for a single-processor machine to hold in memory at once. Moreover, high-quality volume renderings normally take minutes to hours on these machines, 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 inferior visualization results. Second, many acceleration and data exploration techniques for volume rendering trade memory for time, which increases memory use by another order of magnitude. Third, motion is one of the most effective visualization cues, but an animation sequence of volume visualization normally takes hours to days to generate. Finally, we notice the availability of massively parallel computers and hundreds of high-performance workstations in our computing environments. These workstations frequently sit idle for many hours a day.

These observations led us to investigate ways of distributing the increasing amount of data as well as the time-consuming rendering process to the tremendous distributed computing resources available to us. In this article, we describe the resulting parallel volume-rendering algorithm, which consists of two parts: parallel ray tracing and parallel compositing. In our current implementation on Connection Machine's CM-5 and networked workstations, the parallel volume renderer evenly distributes data to the computing resources available. Without the need to communicate with other processing units, each subvolume is ray traced locally and generates a partial image. The parallel compositing process then merges all resulting partial images in depth order to produce the complete image.

Our compositing algorithm is particularly effective for massively parallel processing, as it always uses all processing units by repeatedly subdividing the partial images and distributing them to the appropriate processing units. Our test results on both the CM-5 and the workstations are promising. They do, however, expose different performance issues for each platform.

Background

Many parallel algorithms for volume rendering have been developed recently.[1-4] The major algorithmic strategy for parallelizing volume rendering is the divide-and-conquer paradigm.

0272-17-16/94/$4.00 © 1994 IEEE

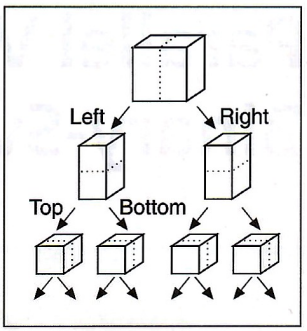


Figure 1. k-D tree subdivision of a data volume.

The subdivision can occur either in data space or in image space. Data-space subdivision assigns the computation associated with particular subvolumes to processors, while 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, while image-space subdivision is often used in shared-memory multiprocessing environments. Our method and similar methods developed independently by Hsu,[1] Camahort,[2] and Neumann[3] can be considered hybrid methods because they subdivide both data space (during rendering) and image space (during compositing).

The basic idea behind our algorithm and other similar methods is very simple: Divide the data into smaller subvolumes and distribute the subvolumes to multiple processors, render them separately and locally, and combine the resulting images in an incremental fashion. The memory demands on each processor are modest, since each one holds only a subset of the total data set.

In earlier work we used this approach to render high-resolution data sets in a computing environment that had many midrange workstations (for example, equipped with 16 Mbytes of memory) on a local area network.[5] Many computing environments have an abundance of such workstations, which could be harnessed for volume rendering if the memory usage on each machine was reasonable.

A divide-and-conquer algorithm

The starting point of our algorithm is the volume ray-tracing technique presented by Levoy.[6] This technique constructs an image in *image order* by casting rays from the eye through the image plane and into the volume. One ray per pixel is generally sufficient, provided the image sample density exceeds the data sample density. The technique uses a discrete rendering model to sample the data at evenly spaced points along the ray, usually at a rate of one to two samples per voxel. The data is interpolated to these sample points, generally using a trilinear interpolant. Color and opacity are determined by applying a transfer function to the interpolated data values—a table lookup can do this. Intensity is assigned by applying a shading function such as the Phong lighting model. You can use the normalized gradient of the data volume as the surface normal for shading calculations.

Sampling continues until the 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. We base the color/opacity compositing for our algorithm on Porter and Duff's "over" operator.[7] It is easy to verify that this operator is associative—that is,

a over (b over c) = (a over b) over c

The associativity of the over operator lets us break a ray into segments, process the sampling and compositing of each segment independently, and combine the results from each segment via a final compositing step.

Data subdivision/load balancing

The divide-and-conquer algorithm requires us to partition the input data into subvolumes. There are many ways to do this. For example, Neumann compared block, slab, and shaft data distribution.[3] Ideally, we would like each subvolume to require about the same amount of computation. We would also like to minimize the amount of data that must be communicated between processors during compositing.

The simplest method is probably to partition the volume along planes parallel to the coordinate planes of the data. If the viewpoint is fixed and known when partitioning the data, we can determine the coordinate plane most nearly orthogonal to the view direction and subdivide the data into "slices" orthogonal to this plane. Orthographic projection tends to produce subimages with little overlap and, therefore, little communication during compositing. If the view point is not known a priori, or if perspective projection is used, it is better to partition the volume equally along all coordinate planes. This *block data distribution* can be done by gridding the data equally along each dimension.[1,2]

We instead use a k-D tree structure for data subdivision,[8] with alternating binary subdivision of the coordinate planes at each level in the tree as indicated in Figure 1. When the number of processors is a power of eight, the volume divides equally among all three dimensions. Hence, this is equivalent to the gridding method described above for block data distribution. If the number of processors is not a power of eight, the volume splits unevenly in the three dimensions, but never by more than a factor of two.

As shown later, the k-D tree structure provides a convenient hierarchical structure for image compositing. Note that with trilinear interpolation, the data lying on the boundary between two subvolumes must be replicated and stored with both subvolumes.

Parallel rendering

Each processor performs local rendering independently—that is, no data communications are required during subvolume rendering. We use ray-casting-based volume 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.

In principle, we could use any volume-rendering algorithm for local rendering. However, some care must be taken to avoid visible artifacts where subvolumes meet. For example, in ray casting, we sample along each ray at a fixed predetermined interval. We must ensure consistent sampling locations for all

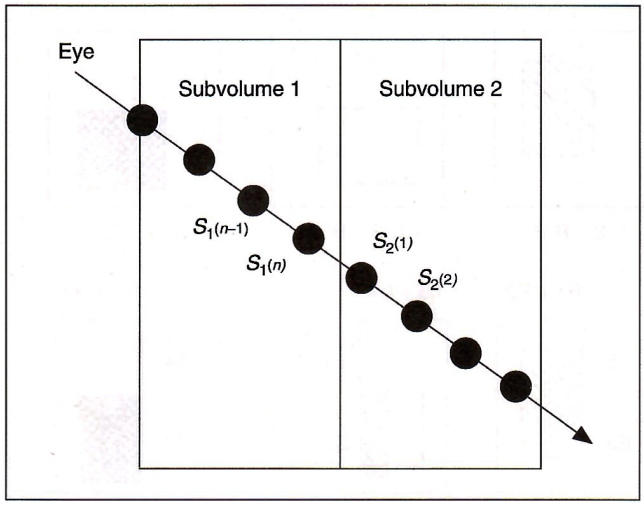


Figure 2. Correct ray sampling.

subvolumes so that we can reconstruct the original volume. As shown in Figure 2, for example, we should calculate the location of the first sample $S_2(1)$ on the ray shown in subvolume 2 correctly so that the distance between $S_2(1)$ and $S_1(n)$ equals the predetermined interval. Without careful attention to the sample spacing, even across subvolume boundaries, the subvolume boundaries can become visible as artifacts in the final image.

Image composition

The final step of our algorithm is to merge ray segments and thus all partial images into the final image. To merge subimages, we must store not only the color at each pixel but also the accumulated opacity. As described earlier, we base the rule for merging subimages 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, we can determine this front-to-back image compositing order 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.[9] In addition, the hierarchical structure provides a natural way to accomplish the compositing in parallel: It lets us process sibling nodes in the tree concurrently.

Binary compositing is a naive approach for parallel merging of the partial images. By pairing up processors in order of compositing, this method produces a new subimage for each disjoint pair. Thus, after the first stage, we are left with the task of compositing only $n/2$ subimages. Then we use half the number of the original processors and pair them up for the next level of compositing. Continuing similarly, we obtain the final image after $\log n$ stages. One problem with this method is that many processors become idle during the compositing process. At the top of the tree, only one processor is active, doing the final composite for the entire image. We found that compositing two 512×512 images required 1.44 seconds on one CM-5 scalar processor. One of our goals was interactive volume rendering, which requires subsecond rendering times, so this method was unacceptable.

The compositing phase must exploit more parallelism. To this end, we generalized the binary compositing method so that every processor participates in all 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 binary-swap algorithm, each processor is responsible for a large portion of the image area, but the data coverage in the image area is usually sparse because only a few processors have contributed to it. In later phases of the algorithm, as we move up the compositing tree, the processors are responsible for a smaller and smaller portion of the image area, but the density of data coverage increases because 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.

This approach can exploit the sparsity of image data, since compositing needs to occur only where nonblank image data is present. Each processor maintains a screen-aligned bounding rectangle of the nonblank subimage area. The processors only store and composite within this bounding rectangle. Two forces affect the size of the bounding rectangle as we move up the compositing tree: The bounding rectangle grows due to the contributions from other processors, but it shrinks due to the subdivision of the image plane as we move up the tree. The net effect is analyzed in greater detail in the next section.

Figure 3 (on the next page) illustrates the binary-swap compositing algorithm graphically for four processors. When all four processors finish rendering locally, each processor holds a partial image, as depicted in Figure 3a. Each partial image is subdivided into two half-images by splitting along the x axis. As shown in Figure 3b, processor 1 keeps only the left half-image and sends its right half-image to its immediate-right sibling, which is processor 2. Conversely, processor 2 keeps its right half-image and sends its left half-image to processor 1. Both processors then composite the half image they keep with the half image they receive. A similar exchange and compositing of partial images occurs between processors 3 and 4.

After the first stage, each processor holds only a partial image, half the size of the original one. In the next stage, processor 1 alternates the image subdivision direction. This time it keeps the upper half-image and sends the lower half-image to its second-to-right sibling, which is processor 3, as shown in Figure 3c. Conversely, processor 3 trades its upper half-image for processor 1's lower half-image for compositing. Concurrently, a similar exchange and compositing occurs between processor 2 and 4. After this stage, each processor holds only one-fourth of the original image. This completes the processing for this example, so each processor sends its image to the display device. The final composited image is shown in Figure 3d. It has been brought to our attention that Mackerras[10] independently developed a similar merging algorithm.

In our current implementation, the number of processors 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

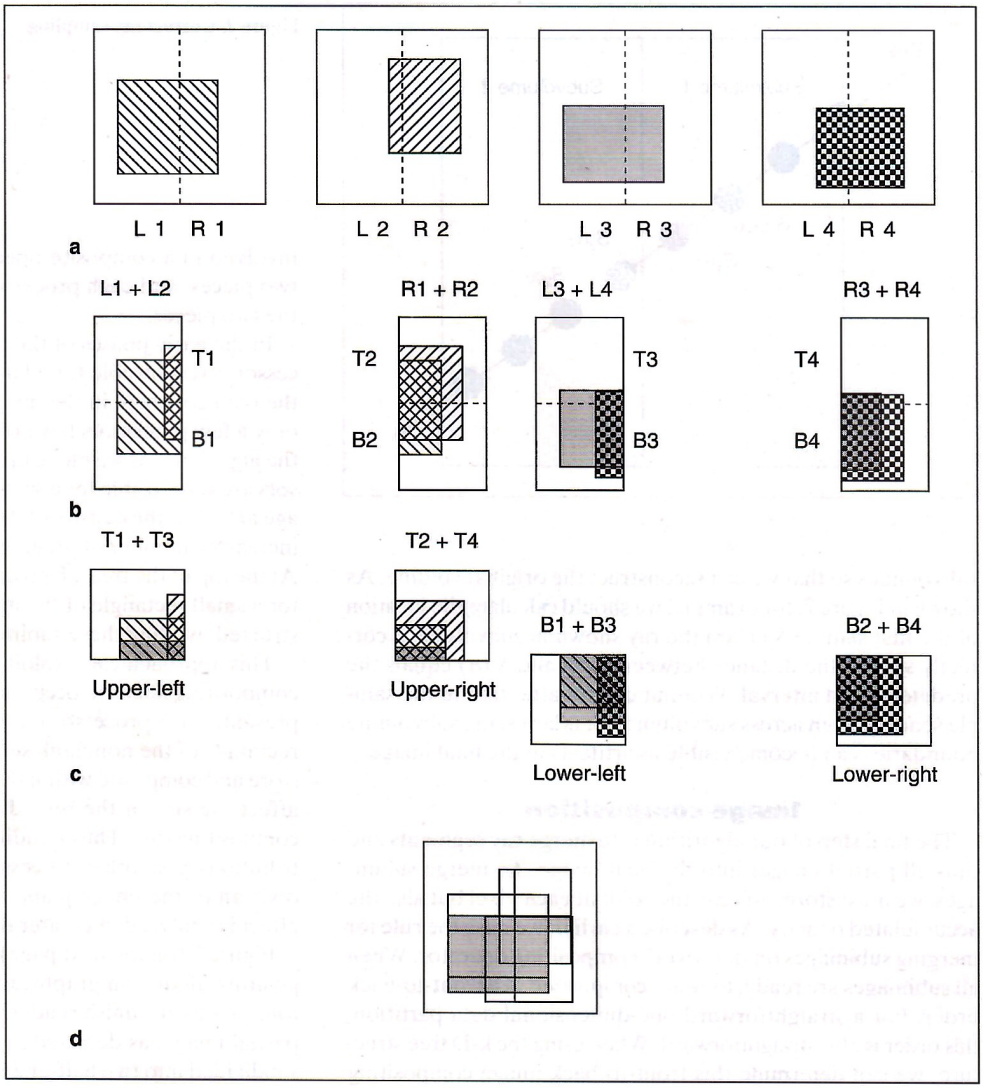


Figure 3. Parallel compositing process.

every compositing phase. We can generalize the algorithm to relax this restriction if we keep the compositing tree as a *full* (but not necessarily complete) binary tree. This generalization causes some additional complexity in the compositing partner computation and leaves some processors idle during the first compositing phase.

Communications costs

At the end of local rendering, each processor holds a subimage of size approximately $p\,n^{-2/3}$ pixels, where p is the number of pixels in the final image and n is the number of processors.[3] The total number of pixels over all n processors is therefore $p\,n^{1/3}$ pixels. The first phase of the binary-swap algorithm communicates half these pixels. At this point, each processor performs a single binary compositing step with the data from the neighboring processors, operating only on the nonempty pixels in the portion of the image plane assigned to it. Some reduction in the total number of pixels will occur due to the depth overlap resolved in this compositing stage.

On average, resolution of the depth overlap reduces the total number of pixels at each compositing phase by a factor of $2^{-1/3}$. To see this, consider what happens over three compositing phases. The k-D tree partitioning of the data set will split each of the coordinate planes in half over three levels in the tree. Orthogonal projection onto any plane will have an average depth overlap of 2. For example, assume the viewpoint is selected so that we are looking straight down the z axis. The x and y axis splits of the data will completely eliminate depth overlap in the image plane, while the z split will result in complete overlap. This cuts the total number of pixels in half. Thus, over three compositing phases, the image size is reduced by a factor of 1/2. The average over each phase then is $2^{-1/3}$, so when three stages are invoked (cubing the per-stage factor), we get the required factor of 1/2.

This process repeats through $\log n$ phases. If we number the phases from $i = 1$ to $\log n$, each phase begins with $2^{-(i-1)/3} n^{1/3} p$ pixels and ends with $2^{-i/3} n^{1/3} p$ pixels. The last phase therefore ends with $2^{-(\log n)/3} n^{1/3} p = n^{-1/3} n^{1/3} p = p$ pixels, as expected. At each phase, half the pixels are communicated. Summing up the pixels communicated over all phases:

$$\text{pixels transmitted} = \sum_{i=1}^{\log n} \left(\frac{1}{2} 2^{-(i-1)/3} n^{1/3} p \right)$$

The $2^{-(i-1)/3}$ term accounts for depth overlap resolution. The $n^{1/3} p$ term accounts for the initial local rendered image size, summed over all processors. The factor of 1/2 accounts for the fact that only half the active pixels are communicated in each phase. This sum can be bounded by pulling out the terms that don't depend on i and noticing that the remaining sum is a geometric series which converges:

$$\begin{aligned} \text{pixels transmitted} &= \sum_{i=1}^{\log n} \left(\frac{1}{2} 2^{-(i-1)/3} n^{1/3} p \right) \\ &= \frac{1}{2} n^{1/3} p \sum_{i=0}^{\log n - 1} 2^{-i/3} \\ &\le 2.43 n^{1/3} p \end{aligned}$$

Comparisons with other algorithms

Alternatives for parallel compositing have been developed simultaneously and independently of our work. One, which we will call *direct send*, subdivides the image plane and assigns each processor a subset of the total image pixels. Hsu[1] and Neumann[3] use this approach. It sends each rendered pixel directly to the processor assigned that portion of the image plane. Processors accumulate these subimage pixels in an array and composite them in the proper order after all rendering is completed. The total number of pixels transmitted with this method is $n^{1/3}\ p\ (1 - 1/n)$, as reported by Neumann.[3] Asymptotically, this is comparable to our result, but with a smaller constant factor.

In spite of the somewhat higher count of pixels transmitted, our method has some advantages over direct send. Direct send requires that each rendering processor send its rendering results to, potentially, every other processor. Indeed, Neumann recommends interleaving the image regions assigned to different processors to ensure good load balance and network utilization. Thus, direct-send compositing could require transmitting $n(n - 1)$ messages. In binary-swap compositing, each processor sends exactly $\log n$ messages, albeit larger ones, so the total number of messages transmitted is $n \log n$. When per-message overhead is high, it can be advantageous to reduce the total message count.

Furthermore, binary-swap compositing can exploit faster nearest neighbor communications paths when they exist. In early phases of the algorithm, processors exchange messages with nearest neighbors. This is exactly when the number of pixels transmitted is largest, since little depth resolution has occurred. On the other hand, binary-swap compositing requires $\log n$ communications phases, while the direct-send method sends each partial ray segment result only once. In an asynchronous message passing environment, direct-send latency costs are $O(1)$. In a synchronous environment, they are $O(n)$ because the processor must block until each message is received and acknowledged. Binary-swap latency costs grow by $O(\log n)$ with either synchronous or asynchronous communications.

Camahort and Chakravarty have developed a different parallel compositing algorithm, which we call the *projection* method.[2] Their rendering method uses a 3D grid decomposition of the volume data. It accomplishes parallel compositing by propagating ray segment results front-to-back along the path of the ray through the volume to the processors holding the neighboring parts of the volume. Each processor composites the incoming data with its own local subimage data before passing the results on to its neighbors in the grid. The final image is projected onto a subset of the processor nodes—those assigned outer back faces in the 3D grid decomposition.

Like the other methods, the projection method requires a total of $O(n^{1/3}\ p)$ pixels to be transmitted. Camahort and Chakravarty observe that each processor sends its results to, at most, three neighboring processors in the 3D grid. Thus, by buffering pixels, the projection method can be implemented with only three message sends per processor as compared to $\log n$ for binary swap and $n - 1$ for direct send. However, it requires the routing of each final image pixel through $n^{1/3}$ processor nodes, on average, on its way to a face of the volume. This means that the message latency costs grow by $O(n^{1/3})$.

Implementation of the renderer

We implemented two versions of our distributed volume rendering algorithm: one on the CM-5 and another on groups of networked workstations. Our implementation consists 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 it to each participating machine. Alternatively, each node program can read its piece from disk directly if parallel I/O facilities exist.

Our renderer is a basic one and not highly tuned for the best performance. Data-dependent volume rendering acceleration techniques tend to be less effective in parallel volume renderers than in uniprocessor implementations, since they may accelerate 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 loads on all processors.

For shading the volume, we use central differencing to approximate surface normals as local gradients. We trade memory for time by precomputing and storing the three components of the gradient at each voxel. For example, a data set of size $256 \times 256 \times 256$ requires more than 200 megabytes 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.

CM-5 and CMMD 3.0

The CM-5 is a massively parallel supercomputer that supports both the single instruction, multiple data (SIMD) and multiple instruction, multiple data (MIMD) programming models.[11] The CM-5 in the Advanced Computing Laboratory at Los Alamos National Laboratory has 1,024 nodes, each a Sparc microprocessor with 32 Mbytes of local RAM and four 64-bit-wide vector units. With four vector units, each node can perform up to 128 operations from a single instruction. This yields a theoretical speed of 128 Gflops for a 1,024-node CM-5. The nodes can be divided into partitions whose size must be a power of two. Each user program operates within a single partition.

Our CM-5 implementation of the parallel volume renderer takes advantage of the CM-5's MIMD programming features. MIMD programs use CMMD (Connection Machine multiple instruction, multiple data), a message-passing library for communications and synchronization, which supports either a hostless model or a host/node model.

We chose the host/node programming model of CMMD be-

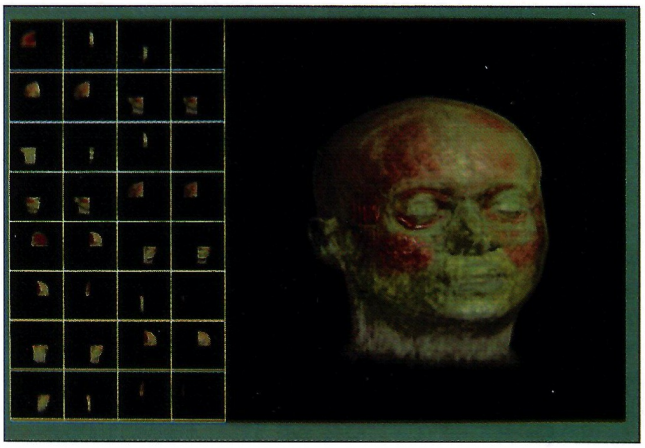


Figure 4. Head data set and parallel compositing process.

cause 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 each node's 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 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 copies 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 it then sends back to the host for display.

Networked workstations and PVM 3.1

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 was to set up a volume rendering facility for handling large data sets and batch animation jobs. That is, we hope that using many workstations concurrently will allow us to decrease the rendering time linearly and 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 an environment.

We use PVM (Parallel Virtual Machine),[12] a parallel program development environment, to implement the data communications in our algorithm. PVM lets us implement our algorithm portably 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.

Figure 5. Vessel data set.

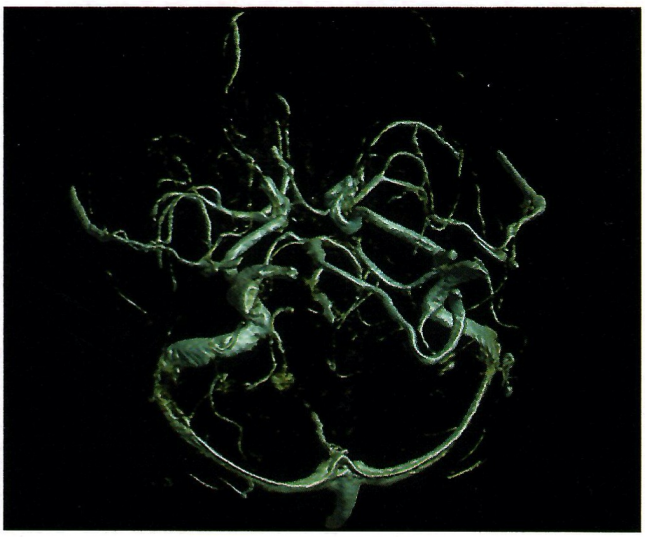
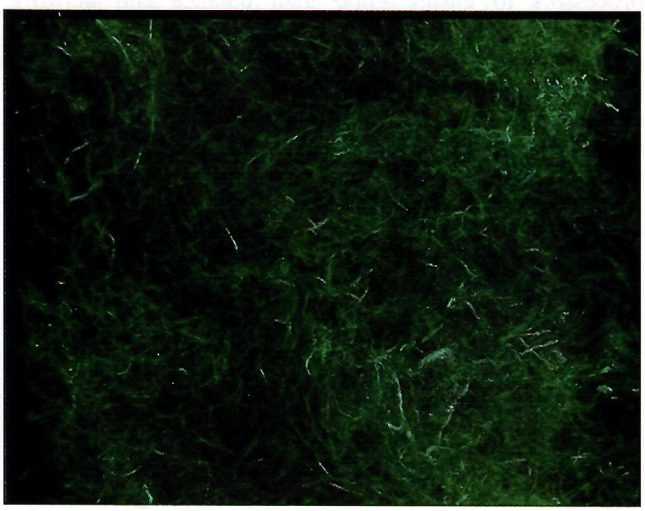


Figure 6. Vorticity data set.



We also used a host/node model. As a result, the implementation is nearly identical to that on the CM-5. In fact, the only distinct difference between the workstation and CM-5 implementations (source program) is the communication calls. Basically, for most of the basic communication functions, PVM 3.1 and CMMD 3.0 have one-to-one equivalence.

Tests

We used three different data sets for our tests. The *vorticity* data set is a 256 × 256 × 256 voxel computational fluid dynamics (CFD) data set, computed on a CM-200, showing the onset of turbulence. The *head* data set is the now-classic University of North Carolina at Chapel Hill magnetic resonance (MR) 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.

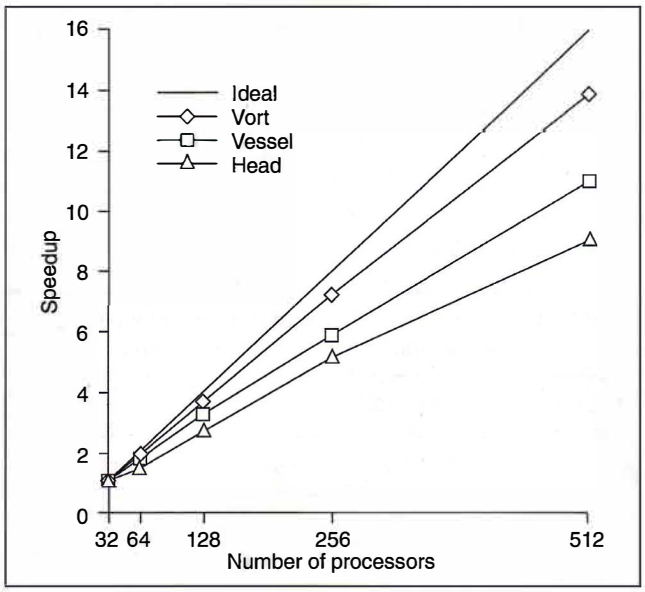


Figure 4 illustrates the compositing process described in Figure 3, using the images generated with the head data set and eight processors. In Figure 4, each row shows the images from one processor, while the columns show—from left to right—the intermediate images before each composite phase. The rightmost column shows the final results, still divided among the eight processors. The final tiled image is blown up and displayed on the right. Figures 5 and 6 show images of the other two data sets rendered in parallel using the algorithm described here.

CM-5

We performed multiple experiments on the CM-5 using partition sizes of 32, 64, 128, 256, and 512 nodes. When these tests were run, a 1,024 partition was not available. Figure 7 shows the speedup results for a 512 × 512 image on each data set. Note that the speedup is relative to the 32-node running time.

Because there is no communication in the rendering step, you might expect linear speedup when utilizing more processors. As the three speedup graphs show, this is not always the case due to load-balance problems. The vorticity data set is relatively dense (that is, it contains few empty voxels) and therefore exhibits nearly linear speedup. On the other hand, both the head and the vessel data sets contain many empty voxels

Table 1. CM-5 time breakdown (in seconds), vorticity data set, 512 × 512 image size.

Function	32	64	128	256	512
dist	89.87	93.516	83.185	94.326	49.157
rend	48.2005	24.4303	12.697	6.3434	3.1878
comp	0.6309	0.5579	0.4091	0.3736	0.3213
comm	0.0843	0.0231	0.0181	0.0138	0.0097
send	0.9918	0.965	0.9645	1.0151	0.9849

Figure 7. CM-5 speedup for 512 × 512 image size.

Figure 8. CM-5 runtimes by data set, 512 × 512 image size.

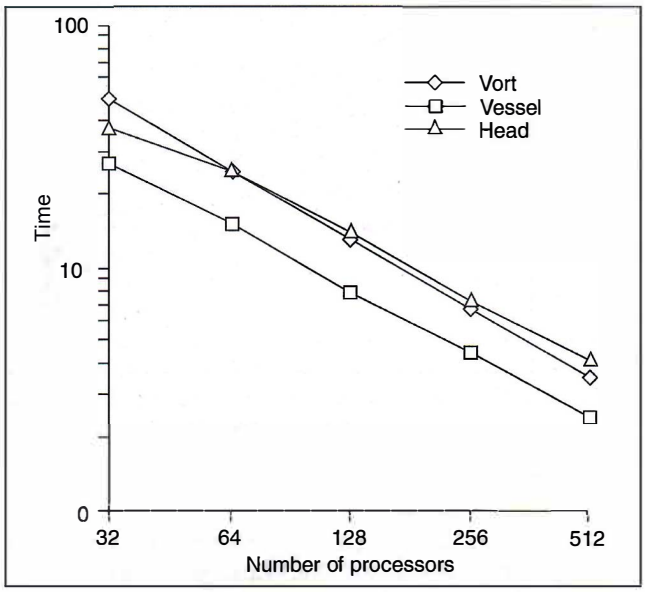
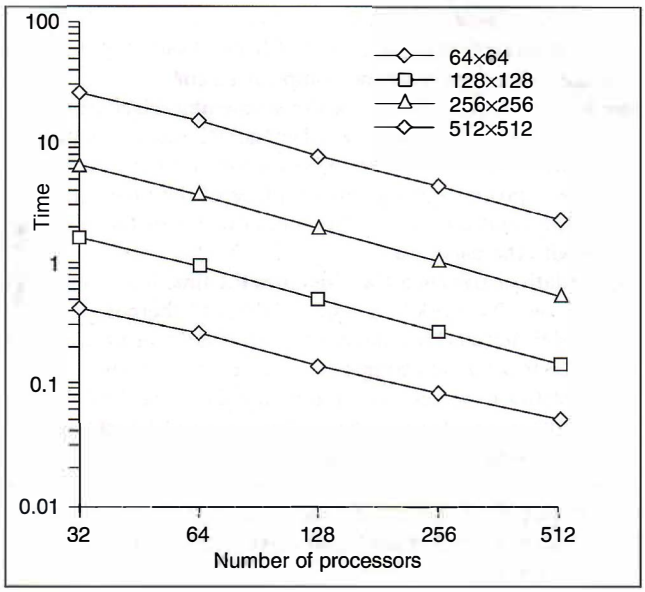


Figure 9. CM-5 runtimes by image size, vessel data.



that unbalance the load. These data sets therefore do not exhibit the best speedup.

Figure 8 shows timing results for all data sets, using an image size of 512 × 512, and Figure 9 shows the results for the vessel data set at several image sizes. All times are given in seconds. The times shown in the graphs are the maximum times for all the nodes for the two steps of the core algorithm: rendering and compositing. The graphs do not include times for data distribution or image gathering.

Table 1 shows a time breakdown by algorithm component—data distribution (dist), rendering (rend), compositing compu-

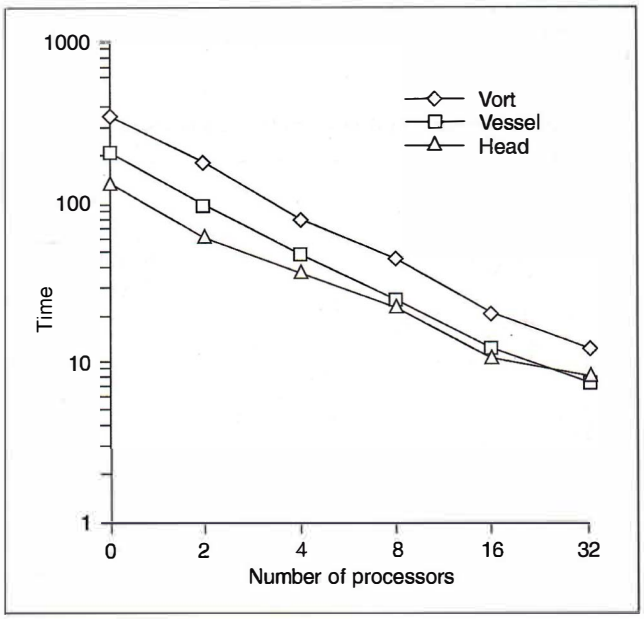


Figure 10. PVM runtimes by data set, 512 × 512 image size.

Table 2. PVM time breakdown (in seconds), vorticity data set, 512 × 512 image size.

Function	1	2	4	8	16	32
rend	350.24	180.15	79.54	45.01	20.59	12.50
comp	0.03	0.17	0.09	0.10	0.12	0.12
comm	0.00	0.57	0.39	2.04	3.11	1.37

tation time (comp), compositing communications (comm), and image gathering (send)—on a 512 × 512 rendering of the vorticity data. It is easy to see that rendering time dominates the process. Note that this implementation does not take advantage of the CM-5 vector units. We expect much faster computation rates for both the renderer and compositor when the vectorized code is complete.

The communication time varied from about 10 percent to about 3 percent of the total compositing time. As the image size increases, both the compositing time and the 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 and because the total communications bandwidth of the machine scales with the partition size.

The data distribution time includes the time it takes to read the data over Network File System (NFS) at Ethernet speeds on a loaded Ethernet. The image gathering time is the time it takes the nodes to send their composited image tiles to the host. While other partitions were also running, the data distribution time could vary dramatically due to the disk and Ethernet contention. Taking the vorticity data set as an example, the data distribution varied from 40 to 90 seconds regardless of the partition size. Both the data distribution and image gathering times will be mitigated by use of the parallel storage and the use of the HiPPI frame buffer.

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 Mbytes of memory. These workstations are rated at 81.8 SPECfp92. The next 12 machines were HP9000/730 workstations, some with 32 Mbytes and others with 64 Mbytes. These machines are rated at 86.7 SPECfp92. The remaining 16 machines were Sun Sparc-10/30 workstations equipped with 32 Mbytes, which are rated at 45 SPECfp92.

The tests on one, two, and four workstations used only the IBMs because of their memory capacity. The tests with eight and 16 used a combination of the HPs and IBMs. We used the 16 Suns for the 32-machine tests. We could not assure absolute quiescence on each machine, because they are in a shared environment with a large shared Ethernet and file systems. During the testing period, there was network traffic from network file system activity and across-the-net tape backups. In addition, the workstations lie on different subnets, increasing communications times when the subnet boundary must be crossed. Thus, the communication performance was highly variable and difficult to characterize.

Figure 10 shows timing using all three data sets and an image size of 512 × 512. Again, the graphs do not include data distribution and image gathering times. In a heterogeneous environment, it is less meaningful to use speedup graphs to study the performance of our algorithm and implementation, so speedup graphs are not provided.

For large images (say, 512 × 512) in the workstation environment, it is worthwhile to compress the subimages used in the compositing process. We incorporated a compression algorithm into our communications library using an algorithm described in Williams.[13] The compression ratio was about four to one, resulting in about 80 percent faster communication rates for the 32-workstation case. With fewer processors, computation tends to dominate over communications and compression is not as much of an advantage. The timing results shown in Figure 10 include the effects of data compression.

Table 2 shows a time breakdown by algorithm component: rendering (rend), compositing computation time (comp), and compositing communications (comm). From the test results, we see that the rendering time still dominates when using eight or fewer workstations. It is also less beneficial to render smaller images due to the overhead costs associated with the rendering and compositing steps. Unlike the CM-5 results, tests on workstations show that the communication component is the dominant factor in the compositing costs. On average, communication takes about 97 percent of the overall compositing time. On the CM-5, a large partition improved the overall communications time, partly because the network bandwidth scales with the partition size. This is *not* true for a local area network,

such as an Ethernet, that has a fixed bandwidth available regardless of the number of machines used. On a LAN, communication costs of the algorithm increase with increasing numbers of machines.

The data distribution and image gathering 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 gathering times varied from an average of 0.06 seconds for a 64×64 image to a high of 8 seconds for a 512×512 image. These test results were based on Version 3.1 of PVM. Our initial tests using PVM 2.4.2 show a much higher communication cost, more than 70 percent higher.

In a shared computing environment, the communication costs for our algorithm are highly variable due to the many factors over which we have no control. For example, an overloaded network and other users' processes competing with our rendering process for CPU and memory usage can greatly degrade the algorithm's performance. We could improve performance by carefully distributing the load to each computer according to data content, the computer's performance, and its average usage by other users. Moreover, communications costs are expected to drop with higher speed interconnection networks (for example, FDDI) and on clusters isolated from the larger local area network.

Conclusions

Our algorithm can render data sets that are too large to fit into memory on a single uniprocessor. Its binary-swap compositing method is particularly suitable for massively parallel processing, and it is simple to implement with the use of the k-D tree structure. The algorithm's implementation on a CM-5 showed good speedup characteristics, with only a small fraction of the total rendering time 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. We can alleviate these bottlenecks 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 hope to achieve real-time rendering rates at medium to high resolution with these improvements.

Better load balancing could improve the performance achieved in the distributed workstation implementation of the algorithm. While linear speedup in a heterogeneous environment with shared workstations is difficult, we are investigating data distribution heuristics that account for varying workstation computation power and workload. ❑

Acknowledgments

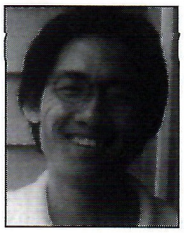
The Medical Imaging Research Laboratory at the University of Utah provided the MRA vessel data set. Shi-Yi Chen of T-Division at Los Alamos National Laboratory provided the vorticity data set. David Rich of the Advanced Computing Laboratory and Burl Hall of Thinking Machines helped tremendously with the CM-5 timings. Alpha_1 and Center for Software Sciences at the University of Utah provided the workstations for our performance tests. Steve Parker helped in the analysis of binary-swap compositing by pointing out the factor of $2^{-(i-1)/3}$ due to depth overlap resolution.

Thanks go to Elena Driskill and the reviewers for comments on earlier versions of this article.

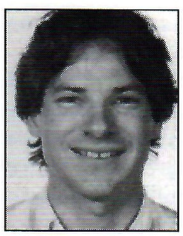
This work has been supported in part by NSF/ACERC, NASA/ICASE, and NSF (CCR-9210587). All opinions, findings, conclusions, or recommendations expressed in this article are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

References

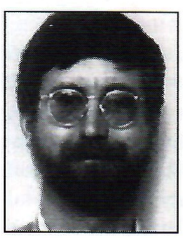
1. W.M. Hsu, "Segmented Ray Casting for Data Parallel Volume Rendering," in *Proc. Parallel Rendering Symp.*, ACM, New York, 1993, pp. 7-14.
2. E. Camahort and I. Chakravarty, "Integrating Volume Data Analysis and Rendering on Distributed Memory Architectures," in *Proc. Parallel Rendering Symp.*, ACM, New York, 1993, pp. 89-96.
3. U. Neumann, "Parallel Volume-Rendering Algorithm Performance on Mesh-Connected Multicomputers," *Proc. Parallel Rendering Symp.*, ACM, New York, 1993, pp. 97-104.
4. B. Corrie and P. Mackerras, "Parallel Volume Rendering and Data Coherence," *Proc. Parallel Rendering Symp.*, ACM, New York, 1993, pp. 23-26.
5. K.-L. Ma and J.S. Painter, "Parallel Volume Visualization and Data Coherence," *Computers and Graphics*, Vol. 17, No. 1, 1993, pp. 31-37.
6. M. Levoy, "Display of Surfaces from Volume Data," *IEEE CG&A*, Vol. 8, No. 3, May 1988, pp. 29-37.
7. T. Porter and T. Duff, "Compositing Digital Images," *Computer Graphics* (Proc. Siggraph), Vol. 18, No. 3, July 1984, pp. 253-259.
8. J. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Comm. ACM*, Vol. 18, No. 9, Sept. 1975, pp. 509-517.
9. H. Fuchs, G. Abram, and E.D. Grant, "Near Real-Time Shade Display of Rigid Objects," *Computer Graphics* (Proc. Siggraph), Vol. 17, No. 3, July 1983, pp. 65-72.
10. P. Mackerras, "A Fast Parallel Marching Cubes Implementation on the Fujitsu AP1000," Tech Report TR-CS-92-10, Dept. of Computer Science, Australian National University, 1992.
11. Thinking Machines Corp., *The Connection Machine CM-5 Technical Summary*, Cambridge, Mass., 1991.
12. G. Geist and V. Sunderam, "Network-based Concurrent Computing on the PVM System," *Concurrency: Practice and Experience*, Vol. 4, June 1992, pp. 293-312.
13. R. Williams, "An Extremely Fast Ziv-Lempel Data Compression Algorithm," in *Proc. IEEE Computer Society Data Compression Conf.*, J. Storer and J. Reif, eds., IEEE Computer Society, Los Alamitos, Calif., 1991.



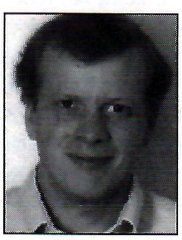
Kwan-Liu Ma is a staff scientist in the Institute of Computer Applications for Science and Engineering (ICASE) at NASA Langley Research Center. His research interests include computer graphics, scientific visualization, and parallel and distributed computing. Ma received BS, MS, and PhD degrees in computer science from the University of Utah. He is a member of IEEE, ACM, and Phi Kappa Phi.



James S. Painter is an assistant professor in the Computer Science Department at the University of Utah. His research interests include scientific visualization, photorealistic image synthesis, and image processing. Painter received a BS in mathematics from the University of Washington in 1979. After severals years in industry at Boeing Computer Services, he returned to graduate school in computer science at the University of Washington, receiving MS and PhD degrees in 1986 and 1990, respectively. He is a member of ACM.



Charles D. Hansen is a technical staff member at the Advanced Computing Laboratory, at Los Alamos National Laboratory, where he is project leader for the scientific visualization environment for the Department of Energy's High Performance Computing Research Center. He also serves as an adjunct faculty member at the University of New Mexico and New Mexico Institute of Technology. His research interests include scientific visualization, parallel rendering, 3D shape representation, and computer vision. He received his BS in computer science from Memphis State University in 1981 and a PhD in computer science from the University of Utah in 1987. He is a member of the IEEE Computer Society and ACM Siggraph.



Michael F. Krogh is a visualization researcher at the Advanced Computing Laboratory at Los Alamos National Laboratory. His research interests include scientific visualization, massively parallel processing, distributed systems, and software engineering. He received a BS in computer science/mathematics in 1987 and an MS in computer science in 1992, from the University of Illinois, Urbana-Champaign. He is a member of IEEE and ACM.

Readers may contact Kwan-Liu Ma at ICASE, M.S. 132C, NASA Langley Research Center, Hampton, VA 23681-0001, e-mail kma@icase.edu.