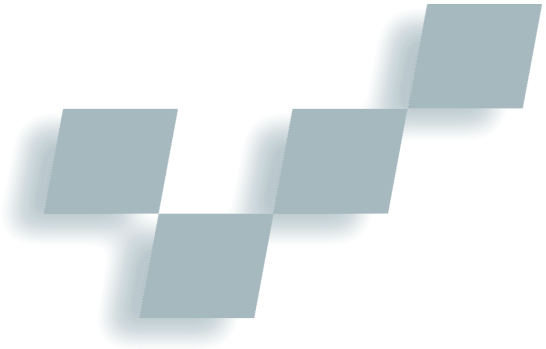


Massively Parallel Software Rendering for Visualizing Large-Scale Data Sets

Kwan-Liu Ma
University of California, Davis

Steven Parker
University of Utah

We describe two highly scalable, parallel software volume-rendering algorithms—one renders unstructured grid volume data and the other renders isosurfaces.

For some time, researchers have done production visualization almost exclusively using high-end graphics workstations. They routinely archived and analyzed the outputs of simulations running on massively parallel supercomputers. Generally, a feature extraction step and a geometric modeling step to significantly reduce the data's size preceded the actual data rendering. Researchers also used this procedure to visualize large-scale data produced by high-resolution sensors and scanners. While the graphics workstation allowed interactive visualization of the extracted data, looking only at a lower resolution and polygonal representation of the data defeated the original purpose of performing the high-resolution simulation or scanning.

To look at the data more closely, researchers could run batch-mode software rendering of the data at the highest possible resolution on a parallel supercomputer using the rendering parameters suggested by the interactive viewing. However, researchers frequently didn't do this for several reasons. First, a supercomputer is a precious resource. Scientists wanted to reserve their limited computer time on the supercomputers for running simulations rather than visualization calculations. Second, many of the parallel-rendering algorithms don't scale well, so the large number of massively parallel computer processors couldn't be fully and efficiently used. Third, most of the parallel-rendering algorithms were developed for meeting research curiosity rather than for production use. As a result, large and complex data couldn't be rendered cost effectively.

However, the current technology trend of cheaper, more powerful computers prompted us to revisit the option of using parallel software rendering (and in some cases, discarding hardware rendering totally). Most graphics cards were mainly optimized for polygon rendering and texture mapping. Scientists can now model physical phenomena with greater accuracy and complexity. Analyzing the resulting data demands advanced rendering features that weren't generally offered by commercial graphics workstations. In addition, the short lifespan, limited resolution, and high cost of graphics workstations constrain what scientists can do.

However, the decreasing cost and rapidly increasing performance of commodity PC and network technologies have let us build powerful PC clusters for large-scale computing. Supercomputing is no longer a shared resource. Scientists can build cluster systems dedicated to their own research. They can also build such systems incrementally to solve problems with increasing complexity and scale. More importantly, they can now afford to use the same machine for visualization calculations either for runtime visual monitoring of the simulation or postprocessing visualization calculations. Therefore, parallel software rendering is becoming a viable solution for visualizing large-scale data sets.

In this tutorial, we describe two highly scalable, parallel software volume-rendering algorithms. We designed one algorithm for distributed-memory parallel architectures to render unstructured-grid volume data.[1] We designed the other for shared-memory parallel architectures to directly render isosurfaces.[2] Through the discussion of these two algorithms, we address the most relevant issues when using massively parallel computers to render large-scale volumetric data. The focus of our discussion is direct rendering of volumetric data, so we don't consider other techniques for treating the large-scale data visualization problem such as feature extraction, multiresolution schemes, and compression.

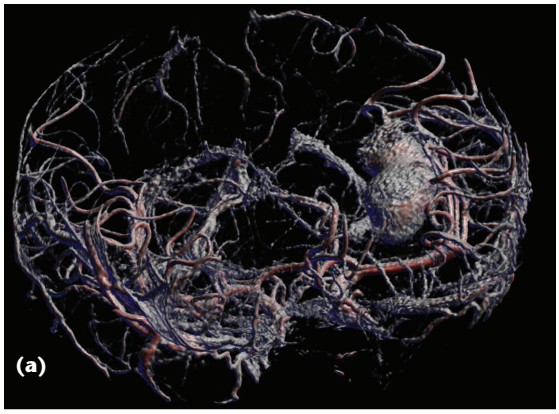
0272-1716/01/$10.00 © 2001 IEEE

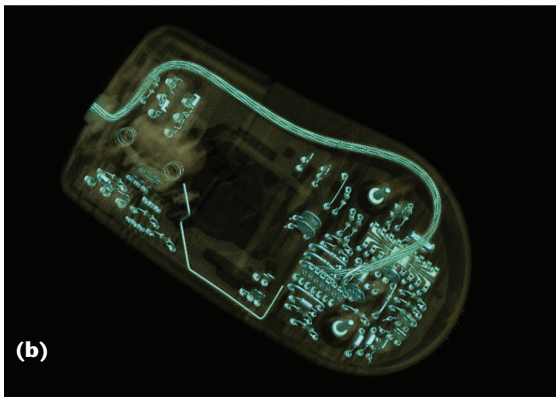
Parallel volume visualization

Volume rendering[3] is a powerful technique because it can potentially display more information in a single visualization than techniques such as isosurfacing or slicing. It's more flexible because we can also implement it to generate isosurfaces and cut planes or a mixture of them. Most importantly, direct volume rendering is particularly effective for visualizing fine features and those features that can't be defined analytically or with a binary classification of the data. Figure 1 shows two volume rendered images.

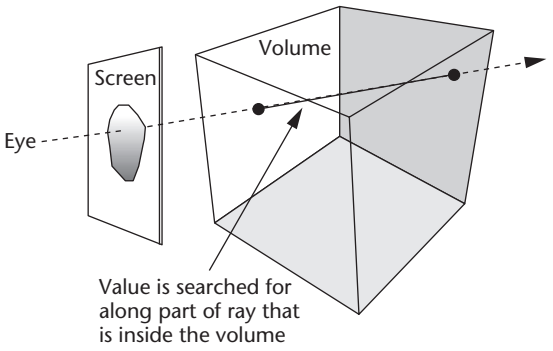
In Figure 2, the basic volume-rendering algorithm steps through the data volume, integrating color and opacity along a ray. A *transfer function* defines the relationship between values in the data set and the color and opacity at each point. Areas of high opacity will contribute more to the pixel's final color. Another method for performing volume rendering starts with the data set's cells and integrates color and opacity for multiple pixels under the data cell's projected area. The difference between these two algorithms is roughly analogous to the differences between ray tracing and z-buffer rendering. Other algorithms, such as the first one we describe in the following section, employ a hybrid approach between these two algorithms.

Volume rendering is computationally expensive because of the interpolation and shading calculations required for every sample point in the data's spatial domain. We can achieve interactive volume rendering of large-scale data sets by using graphics hardware[4,5] or a parallel computer.[6] Kniss et al.[4] used a hybrid parallel-software and texture-hardware capability of a multipipe SGI Onyx2 and rendered a time-varying 1024^3 volume at multiple frames per second. Lum, Ma, and Clyne[5] demonstrated that a single PC with a nVidia GeForce 2 card and a disk array (for less than $2,500) could render a time-varying $640 \times 256 \times 256$ volume at high frame rates, permitting interactive exploration in both the data's temporal and spatial domains. While hardware rendering has many advantages, parallel systems often use software rendering algorithms to provide maximum flexibility and to avoid any constraints on data size, image size, or image quality.





1 Volume visualizations: (a) A volume rendering of the human brain, showing an aneurysm near the right of the image. (Image courtesy of Gordon Kindlmann.) (b) A volume rendering of a CT Microsoft computer mouse. (Data courtesy of Tony Davis and Matthew Sheats.)



2 A ray traverses a volume, integrating color and opacity through the entire volume.

A parallel volume-rendering algorithm for 3D unstructured-grid data

Increasingly, we use unstructured meshes to model some of the most challenging scientific and engineering problems, from aerodynamics calculations, accelerator simulations, and blast simulations to bioelectric field simulations. Applying finer meshes only to regions with complex geometry or requiring high accuracy can significantly reduce computing time and storage space. This adaptive approach results in computational meshes containing irregular data cells (in both size and shape). The lack of a simple indexing scheme for traversing the data cells makes visualization calculations on such meshes more expensive than on regular meshes. In a distributed computing environment, irregularities in cell size and shape make balanced load distribution especially difficult.

Ma and Crockett[1] introduced a parallel cell-projection volume rendering algorithm for visualizing unstructured-grid data. The basic algorithm performs the following sequence of tasks:

1. distribute data and visualization parameters,
2. space partitioning,
3. view transformation,
4. scan conversion of data cells,
5. merge of ray segments, and
6. assemble and output final images.

In this tutorial, we discuss the design philosophy behind this algorithm and the implementation strategies we used to achieve high scalability. Our test results show that an implementation of this algorithm on the Cray T3E can achieve above 75 percent parallel efficiency when rendering 18 million tetrahedra using up to

The algorithm introduced here sacrifices a small amount of performance in favor of enhanced usability. An optimal renderer design should seek a balance between preprocessing cost and parallel-rendering performance.

512 processors. To learn complete performance numbers on several different parallel architectures, you should refer to the Ma and Crockett article.[1]

Key design criteria

One problem shared by many visualization algorithms for unstructured data is the need for a significant amount of preprocessing. One step extracts additional information about the mesh, such as connectivity, to speed up later rendering calculations. Another step may be needed to partition the data based on the particular parallel computing configuration being used (number of processors, communication parameters, and so on). To reduce the user-hassle factor as much as possible and avoid increasing the data size or replicating data, we suggest avoiding offline preprocessing whenever possible—especially for rendering large-scale data sets.

While eliminating preprocessing provides flexibility and convenience, it also means less information is available for optimizing the rendering computations. The algorithm introduced here sacrifices a small amount of performance in favor of enhanced usability. An optimal renderer design should seek a balance between preprocessing cost and parallel-rendering performance. One good design criterion is to conduct low-cost preprocessing calculations (for example, for view-dependent optimization) prior to the rendering time on the same parallel computer to increase parallel rendering efficiency. That is, the preprocessing is also parallelized to incur less storage requirements and data transport.

In addition to offering flexibility, the parallel renderer must be highly scalable so that whenever many processors are available they can be used to render large-scale data efficiently. As we describe in the next section, we can achieve high scalability by using fine-grain load interleaving coupled with an asynchronous communication strategy that overlaps the rendering calculations with interprocessor communication.

Load distribution

Ideally, data should be distributed so that every processor requires the same amount of storage space and incurs the same computational load. Several factors affect this. For the sake of concreteness, let's consider meshes composed of tetrahedral cells. First, there's some cost for scan converting each cell. Variations in the number of cells assigned to each processor will produce variations in workloads. Second, cells come in different sizes and shapes. The difference in size can be as large as several orders of magnitude due to the mesh's adaptive nature. As a result, a cell's projected image area can vary dramatically, which produces similar variations in scan-conversion costs. Furthermore, a cell's projected area also depends on the viewing direction. Finally, voxel values are mapped to both color and opacity values to user-specified transfer functions. An opaque cell can terminate a ray early, thereby saving further merging calculations but introducing further variability in the workload.

One common approach assigns groups of connected cells to each processor so that exploiting cell-to-cell coherence can optimize the rendering process. But connected cells are often similar in size and opacity, so that grouping them together exacerbates load imbalances, making it difficult to obtain satisfactory partitionings. For parallel rendering of large-scale unstructured-grid data, the opposite approach—dispersing connected cells as widely as possible among the processors—is in fact better. That is, each processor is loaded with cells taken from the whole spatial domain rather than from a small neighborhood.

We can generally achieve satisfactory scattering of the input data with a simple round–robin assignment policy. With enough cells, the computational requirements for each processor tend to average out, producing an approximate load balance. This approach also satisfies our requirement for flexibility, since the data distribution can be computed trivially for any number of processors, without the need for an expensive preprocessing step.

Dispersing the grid cells among processors also facilitates an important visualization operation for unstructured data—zoom-in viewing. Because of the highly adaptive nature of unstructured meshes, the most important simulation results are usually associated with a relatively small portion of the overall spatial domain. The viewer normally takes a peek at the overall domain, then immediately focuses on localized regions of interest such as areas with high velocity or gradient values. This zooming operation introduces challenges for visualizing a distributed computing environment efficiently. First, locating all the cells that reside within the viewing region can be expensive. Our solution, described in the next section, employs a spatial partitioning tree to speed up this cell searching. Second, if data cells are distributed to processors as connected components, zooming in on a local region will result in severe load imbalances, because only a few processors handle all the rendering calculations while others are idle.

Although the round–robin distribution discourages data sharing, this rendering algorithm only requires minimum data—the cell and node information. It doesn't need connectivity data. Each cell takes 16 bytes to store four node indices, and each node takes 16 bytes to store three coordinates and a scalar value. As a result, in the worst case of not sharing any node information, $80n$ bytes of data must be transferred for distributing n cells per processor. For example, on average, about 640,000 bytes of data are transferred to and stored at each proces-

sor when distributing a data set consisting of 1 million cells to 128 processors.

In addition to the object–space operations on mesh cells, pixel-oriented ray-merging computations must also be evenly distributed to processors. Local variations in cell sizes within the mesh lead directly to variations in depth complexity in image space. We also need an image partitioning strategy that disperses the ray-merging operations. *Scanline interleaving*, which assigns successive image scanlines to processors in round–robin fashion, generally works well as long as the image's vertical resolution is several times larger than the number of processors. When using numerous processors, we need finer grained pixel interleaving to more effectively distribute high-density regions over more processors, resulting in better load balancing and improved scalability. Our test results show we can speed up the overall rendering by nearly 10 percent with pixel interleaving when using a large system.

Parallel space partitioning

The round–robin data distribution scheme helps to achieve flexibility and produces an approximate static load balance. However, it destroys the spatial relationship between mesh cells, making an unstructured data set even more irregular. Certain ordering must be restored so that the rendering step can be performed more efficiently.

The central idea is to have all processors render the cells in the same neighborhood at about the same time. Ray segments generated for a particular region would consequently arrive at their image–space destinations within a relatively short window of time, letting them merge early. This early merging tends to limit the length of the ray-segment list that each processor maintains, which benefits the rendering process in two ways:

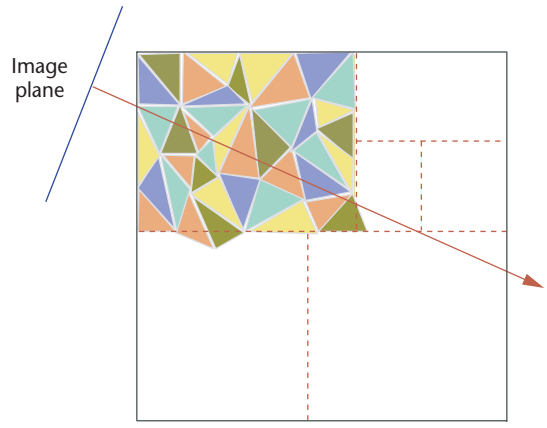
- a shorter list reduces the cost of inserting a ray segment in its proper position within the list and
- the memory needed to store unmerged ray segments is reduced.

To provide the desired ordering, we can group data cells into local regions using a hierarchical spatial data structure such as an octree or *k*-d tree. Figure 3 shows rendering a region within such a partitioning, where different processors store the different colored cells and scan converts them.

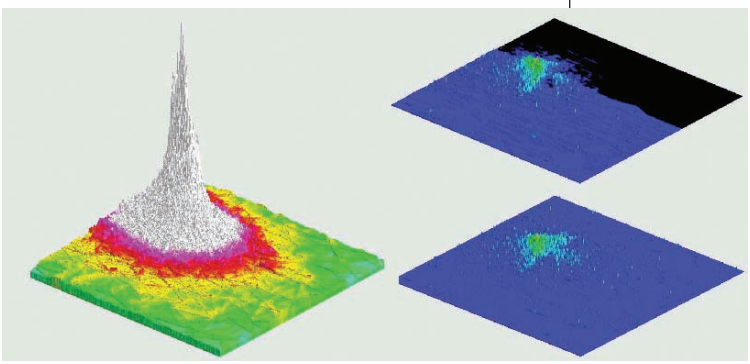
The tree should be constructed cooperatively so that the resulting spatial partitioning is exactly the same on every processor. After the host computer initially distributes the data cells, all processors participate in such a synchronized parallel partitioning process. The algorithm works as follows:

- Each processor examines its local collection of cells and establishes a cutting position so that the two resulting subregions contain about the same number of cells. The cut's direction is the same on each processor and alternates at each level of the partitioning.
- The proposed local cutting positions are reported by the processors to a designated host node, which averages them to obtain a global cutting position. This information is then broadcast by the host node to each processor, along with the host's choice of the next subregion to be partitioned. We assign a cell crossing the cut boundary to the region containing its centroid.
- The procedure repeats until the desired number of regions has been generated.

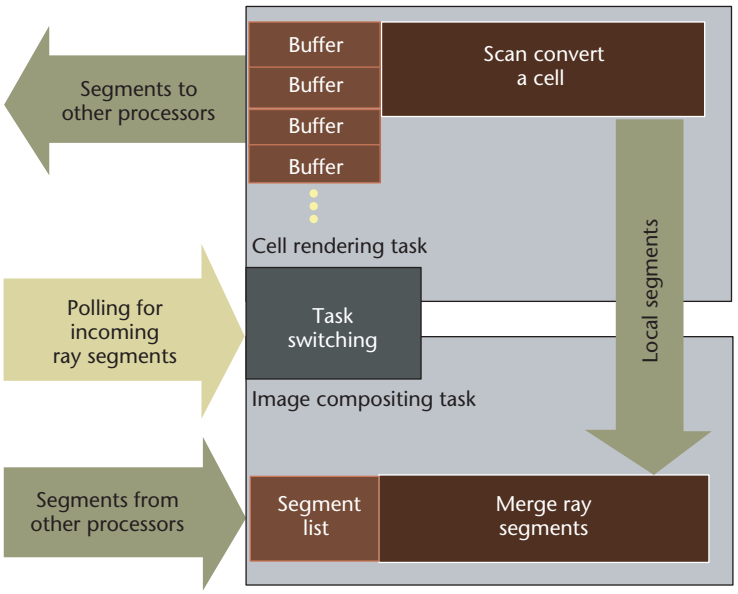
At the end of the partitioning process, each processor has an identical list of regions, with each region representing approximately the same rendering load as the corresponding region on every other processor. If all processors render their local regions in the same order, loose synchronization can be achieved because of the similar workloads, letting early ray merging take place within the local neighborhoods. Test results show that for each pixel the maximum number of ray segments that can't be premerged and therefore must be stored into the temporary list is always less than 10 percent of the total number of ray segments received for the same pixel. Figure 4 displays plots of the total number of the ray segments received for each pixel and the actual number stored. The *k*-d tree also allows for fast searching of



3 A spatial data structure directs all processors to work on cells in the same neighborhood at about the same time. Different processors handle the different colored cells.



4 Runtime memory consumption for a data set from modeling a flow field surrounding an airplane wing. The left image shows the total number of ray segments received for each pixel. The spike coincides with the fine-mesh region. The right images show the number of ray segments actually stored at two different times during the rendering. It's also apparent from the images how using a spatial partitioning tree affects the order in which ray segments are generated and delivered.



5 Task management with asynchronous communication.

cells within a spatial region specified by a zoom-in view. In addition, the spatial regions can also serve as workload units should we ever need to perform dynamic load balancing.

Cell-projection rendering

We use a cell-projection rendering method to render the volume data because it offers more flexibility in data distribution. During rendering, processors follow the same path through the spatial partitioning tree, processing all the cells at each leaf node of the tree. Each processor scan converts local cells independent of other processors and sends the resulting ray segments to the processor that owns the corresponding image scanline. Ray segments received by each processor are merged according to the depth order using the standard Porter–Duff over operator.

Because the ray segments that contribute to a given pixel arrive in an unpredictable order, each ray segment must contain not only a sample value and pixel coordinates but starting and ending depth values for sorting and merging within the pixel's ray segment list. For the typical applications we envision, 10^6 to 10^8 ray segments may be generated for each image; at 16 bytes per segment, aggregate communication requirements are approximately 10^7 to 10^9 bytes per frame. Clearly, managing the communication efficiently is essential to this approach's viability.

Task management with asynchronous communication

Good scalability and parallel efficiency can only be achieved if we can keep the parallelization penalty and communication cost low. To manage communication costs, we use an asynchronous communication strategy. Its features include

- asynchronous operation, which allows processors to proceed independently of each other during the rendering computations;
- multiplexing of the object–space cell computations with the image–space ray merging computations;
- overlapped computation and communication, which hides data transfer overheads and spreads the communication load over time; and
- buffering of intermediate results to amortize communication overheads.

During the course of rendering, two main tasks must be performed: scan conversion and image compositing. We can attain high efficiency if we can keep all processors busy doing either of these two tasks. Logically, the scan conversion and merging operations represent separate threads of control, operating in different computational spaces and using different data structures. For the sake of efficiency and portability, however, it's a good strategy to interleave these two operations using a polling strategy, as Figure 5 illustrates. Each processor starts by scan converting one or more data cells. Periodically the processor checks to see if incoming ray segments are available; if so, it switches to the merging task, sorting and merging incoming rays until no more input is pending.

Because of the large number of ray segments generated, the overhead for communicating each of them individually would be prohibitive in most architectures. Instead, it's better to buffer them up locally and send many ray segments together in one operation. To supplement this, we use asynchronous send and receive operations, which let us overlap communication and computation, reduce data copying overheads in message-passing systems, and decouple the sending and receiving tasks. This strategy proves most effective when each destination has two or more ray-segment buffers. While a send operation is pending for a full buffer, the scan conversion process can be placing additional ray segments in its companion buffer. In the event that both buffers for a particular destination fill up before the first send completes, rendering can switch to the ray-merging task and process incoming segments while waiting for the outbound congestion to clear (in fact, this is essential to prevent deadlock).

Users may specify two parameters to control the frequency of task switching and communication. The first parameter is the polling interval—that is, the number of cells to be processed before checking for incoming ray segments. If polling occurs too frequently, excessive overheads will be introduced; if not, the asynchronous communication scheme will perform poorly as outbound buffers clog up due to pending send operations.

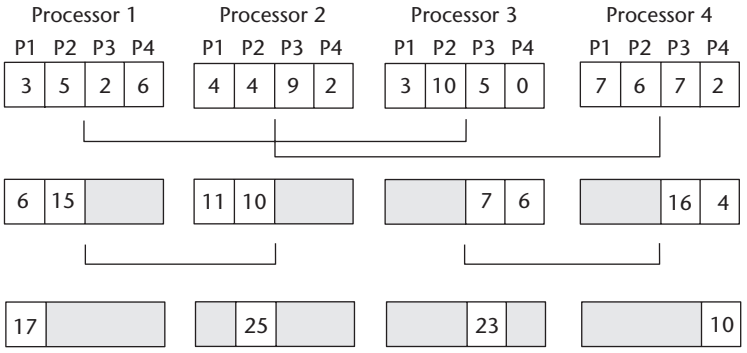
The second parameter is the buffer depth, which indicates how many ray segments should be accumulated before the system posts an asynchronous send. If the buffer size is too small, the overheads for initiating send

and receive operations will be excessive, resulting in lowered efficiency. On the other hand, large buffers can introduce delays for processors that have finished their scan conversion work and are waiting for ray segments to merge. Large buffers are also less effective at spreading the communication load across time, resulting in contention delays in bandwidth-limited systems.

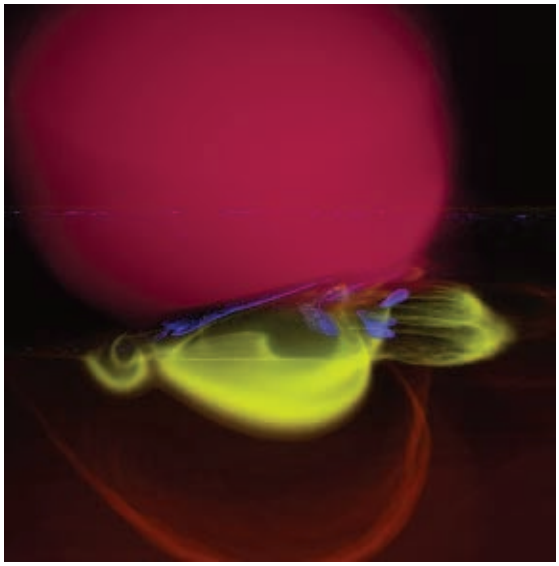
When choosing an ideal buffer size, we must take into account the number of processors in use, the number of ray segments to be communicated, and the target architecture's characteristics. As you might suspect, the polling frequency should be selected in accordance with the buffer size. As a general rule, polling should be performed more frequently with smaller buffer sizes or larger numbers of processors. Empirical results agree with this rule.

Because of the rendering algorithm's asynchronous nature, individual processors can't determine when a frame is complete. Hence, we must use a distributed termination detection protocol. A straightforward approach would use a designated node (like the host processor) to coordinate the termination process by collecting local termination messages and broadcasting a global termination signal. A final global synchronization operation then ends the overall rendering process. When using a large number of processors, this approach would prolong the termination step.

A more scalable solution calls for using a binary merging algorithm based on ray-segment counts. That is, because each processor knows exactly how many ray segments it has sent to each of the other processors, a binary-swap summing process would result in the total number of ray segments each processor should eventually receive. At the end of this globally synchronous process, each processor should hold exactly the number it needs. If this number is the same as the number of segments the processor has actually received, the processor can stop receiving. Figure 6 illustrates such a summing process for a four-processor case. This binary-swap summing process can be carried out as soon as all processors finish projecting local cells. Such an algorithm runs in logarithmic, rather than linear, time and doesn't involve the host, making it more efficient and scalable to a larger number of processors.



6 Termination is based on a binary-swap summing process. At the end of the process, the total number of ray segments each processor should receive becomes available locally.



7 Visualization of flow surrounding an aircraft wing with the wing taken out.

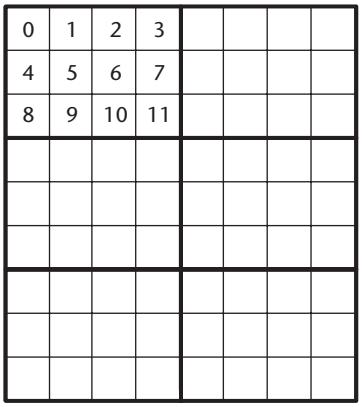
Discussion

We performed tests of the parallel volume-rendering algorithm using an 18 million cell data set obtained from a simulation of flow surrounding an aircraft wing (see Figure 7). Direct volume rendering of the flow speed (with the wing taken out) elicits many fine features in the flow field that would be invisible with conventional 2D or 3D contour plots. In particular, we can verify the low-pressure region (red spherical cloud) above the wing and the high-pressure region (yellow and orange blobs) below the wing. We can also see the extreme low-velocity values on the flaps (white stripes) and the complex flow patterns ahead of the leading edge and behind the trailing edge of the wing. None of these detailed phenomena could be seen with either low-resolution data or rendering.

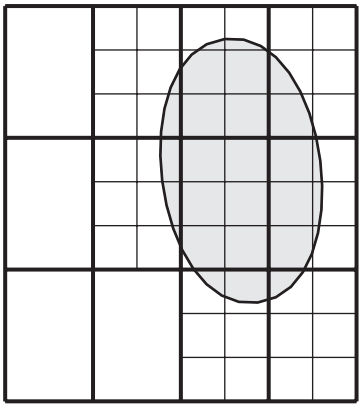
The algorithm scales well on large parallel systems. As we mentioned earlier, we obtained more than 75 percent parallel efficiency on the Cray T3E using up to as many as 512 processors. Even on a cluster of Sun Ultra 5 and Ultra 60 CPUs over Fast Ethernet, our experimental results show more than 63 percent parallel efficiency using up to 128 processors. A PC cluster with a 1-Gbit network should easily outperform the T3E for comparable numbers of processors and network performance. Nevertheless, we should point out that this algorithm doesn't suit shared-memory architectures. Our experiments on the SGI Origin 2000 architecture show the algorithm doesn't scale beyond 32 processors.[7] The algorithm we describe next was designed specifically for shared-memory architectures.

A parallel ray-tracing algorithm for isosurface rendering

The parallel rendering system we discussed in the previous section operates from object space, rendering individual volume elements to a screen. The other approach is to do the opposite: for each pixel on the screen, determine which object or objects contribute to that pixel's final color. This process is ray tracing.



8 We can organize cells into tiles or bricks in memory to improve locality. The numbers in the first brick represent layout in memory. Neither the number of voxels nor the number of bricks need be a power of two.



9 With a two-level hierarchy, rays can skip empty space by traversing larger cells. We used a three-level hierarchy for most of the examples in this article.

Conventional wisdom holds that ray tracing is too slow to be competitive with hardware z-buffers. However, when rendering a sufficiently large data set, ray tracing should be competitive because its low time complexity ultimately overcomes its large time constant.[8] Because of the asymptotic complexity of each algorithm, ray tracing becomes a faster algorithm for rendering images with large numbers of primitives. This crossover will happen sooner on a multiple CPU computer because of ray tracing's high degree of intrinsic parallelism. The same arguments apply to the volume traversal problem.

Figure 2 shows the basic ray-volume traversal method we describe here. This framework lets us implement volume visualization methods that find exactly one value along a ray. Here, we describe an isosurfacing algorithm for parallel ray tracing. Maximum-intensity projection is a direct volume-rendering technique where the opacity is a function of the maximum intensity seen along a ray.

We know that ray tracing is accelerated through two main techniques:[9] accelerating or eliminating ray–voxel intersection tests and parallelization. Acceleration is usually accomplished by a combination of spatial subdivision and early ray termination.[3]

Ray tracing for volume visualization naturally lends itself toward parallel implementations.[10] The computation for each pixel occurs independently of all other pixels, and the data structures used for casting rays are usually read only. These properties have resulted in many parallel implementations. Researchers have used a variety of techniques to make such systems parallel and have built many successful systems.[11,12] (Whitman[13] surveyed these techniques.)

Our system organizes the data into a shallow rectilinear hierarchy for ray tracing. For unstructured or curvilinear grids, we imposed a rectilinear hierarchy over the data domain. Within a given level of the hierarchy, we traverse from cell to cell using the incremental method that Amanatides and Woo[14] described.

Memory bricking

The first optimization is to improve data locality by organizing the volume into bricks, which are analogous to using image tiles in image-processing software and other volume rendering programs[15] (see Figure 8).

Effectively using the cache hierarchy proves a crucial task in designing algorithms for modern architectures. Bricking, or 3D tiling, has been a popular method for increasing locality for ray-cast volume rendering. The system reorders the data set into $n \times n \times n$ cells, which then fill the entire volume. On a machine with 128-byte cache lines and using 16-bit data values, n is exactly 4. However, using float (32-bit) data sets, n is closer to 3.

Using an effective translation look-aside buffer (TLB) is also becoming a crucial factor in algorithm performance. The same bricking technique can be used to improve TLB hit rates by creating $m \times m \times m$ bricks of $n \times n \times n$ cells. For example, a $40 \times 20 \times 19$ volume could be decomposed into $4 \times 2 \times 2$ macrobricks of $2 \times 2 \times 2$ bricks of $5 \times 5 \times 5$ cells. This corresponds to $m = 2$ and $n = 5$. Because 19 can't be factored by $mn = 10$, we need one level of padding. We use $m = 5$ for 16-bit data sets and $m = 6$ for 32-bit data sets.

The resulting offset q into the data array can be computed for any x, y, z triple with the expression

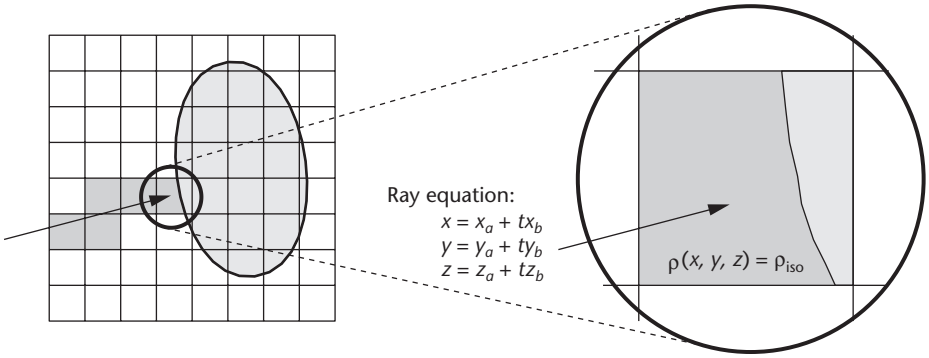
$$\begin{aligned} q = {} & ((x \sqrt{} n) \sqrt{} m)n^3m^3((N_z \sqrt{} n) \sqrt{} m)((N_y \sqrt{} n) \\ & \sqrt{} m) + ((y \sqrt{} n) \sqrt{} m)n^3m^3((N_z \sqrt{} n) \sqrt{} m) \\ & + ((z \sqrt{} n) \sqrt{} m)n^3m^3 + ((x \sqrt{} n) \bmod m)n^3m^2 \\ & + ((y \sqrt{} n) \bmod m)n^3m + ((z \sqrt{} n) \bmod m)n^3 \\ & + (x \bmod n \times n)n^2 + (y \bmod n) \times n + (z \bmod n) \end{aligned} \quad (1)$$

where N_y and N_z are the respective sizes of the data set.

Equation 1 contains many integer multiplication, divide, and modulus operations. On modern processors, these operations are extremely costly (32 or more cycles for the MIPS R10000). Where n and m are powers of two, we can convert these operations to bit shifts and bit-wise logical operations. However, the ideal size is rarely a power of two. Thus, we need a method that addresses arbitrary sizes. We can convert some of the multiplications to shift–add operations, but the divide and modulus operations prove more problematic. Computing the indices incrementally would require tracking nine counters, with numerous comparisons and poor branch prediction performance. Note that we can write Equation 1 as

$$q = F_x(x) + F_y(y) + F_z(z) \quad (2)$$

where



10 The ray traverses each cell (left), and when a cell is encountered that has an isosurface in it (right), the system performs an analytic ray-isosurface intersection computation.

$$F_x(x) = ((x \surd n) \surd m)n^3m^3((N_z \surd n) \surd m)((N_y \surd n) \surd m) + ((x \surd n) \bmod m)n^3m^2 + (x \bmod n \times n)n^2$$

$$F_y(y) = ((y \surd n) \surd m)n^3m^3((N_z \surd n) \surd m) + ((y \surd n) \bmod m)n^3m + (y \bmod n) \times n$$

$$F_z(z) = ((z \surd n) \div m)n^3m^3 + ((z \surd n) \bmod m)n^3 + (z \bmod n)$$

We tabulate F_x, F_y, and F_z and use x, y, and z, respectively, to find three offsets in the array. We sum these three values to compute the index into the data array. These tables will consist of N_x, N_y, and N_z elements, respectively. The total sizes of the tables will fit in the processor's primary data cache even for large data-set sizes. Using this technique, we note that you could produce more complex mappings than the two-level bricking we describe here, although it's not obvious which would achieve the highest cache use.

For many algorithms, each iteration through the loop examines the eight corners of a cell. To find these eight values, we need to only look up $F_x(x)$, $F_x(x+1)$, $F_y(y)$, $F_y(y+1)$, $F_z(z)$, and $F_z(z+1)$. This consists of six index table lookups for each of the eight data value lookups.

Multilevel grid

The other basic optimization we use is a multilevel spatial hierarchy to accelerate the traversal of empty cells as Figure 9 shows. Cells are divided into equal portions and then the system creates a "macrocell," which contains the minimum and maximum data value for its children cells. This is a common variant of standard ray-grid techniques[16] and resembles previous multilevel grids.[17] Others have shown minimum–maximum caching to be useful.[18,19]

The ray–isosurface traversal algorithm examines the min and max at each macrocell before deciding whether to recursively examine a deeper level or to proceed to the next cell. The typical complexity of this search will be $O(\sqrt[3]{n})$ for a three-level hierarchy. While the worst case complexity is still $O(n)$, it's difficult to imagine an isosurface approaching this worst case. Using a deeper hierarchy can theoretically reduce the average case of complexity slightly but also dramatically increases the storage cost of intermediate levels.

We've experimented with modifying the number of levels in the hierarchy and empirically determined that a trilevel hierarchy (one top-level cell, two intermediate macrocell levels, and the data cells) is efficient. This optimum may be data dependent and is modifiable at program startup. Using a trilevel hierarchy, the storage overhead is negligible (typically less than 0.5 percent of the data size). The cell sizes used in the hierarchy are independent of the brick sizes used for cache locality in the first optimization.

We can index macrocells with the same approach that we used for memory bricking the data values. However, in this case each macrocell will have three table lookups. This, combined with the significantly smaller memory footprint of the macrocells, made the effect of bricking the macrocells negligible.

Rectilinear isosurfacing

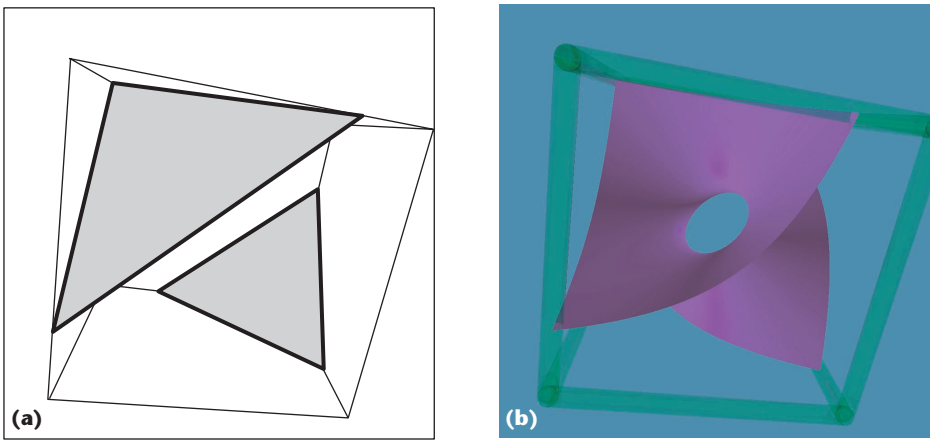
Once the data have been organized into bricks and macrocells, the system can use these data structures to render isosurfaces interactively. Our algorithm has three phases:

1. traversing a ray through cells that don't contain an isosurface,
2. analytically computing the isosurface when intersecting a voxel containing the isosurface, and
3. shading the resulting intersection point.

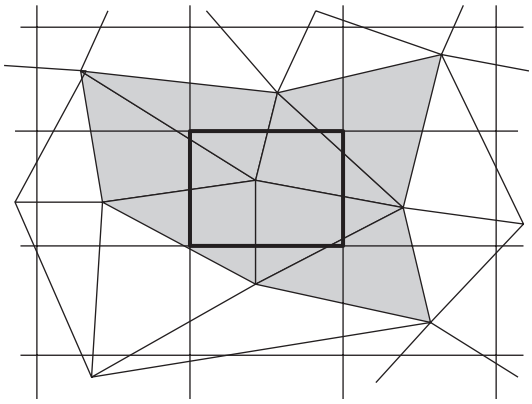
This process repeats for each pixel on the screen. A benefit is that adding incremental features to the rendering has only incremental cost. For example, if you visualize multiple isosurfaces with some of them rendered transparently, the correct compositing order is guaranteed because we traverse the volume in a front-to-back order along the rays. Additional shading techniques, such as shadows and specular reflection, can easily be incorporated for enhanced visual cues. Another benefit is the ability to exploit texture maps without being limited by a fixed quantity of special texture memory. We've demonstrated results of 5.1-Gbyte 3D textures interactively.

If we assume a regular volume with even grid-point spacing arranged in a rectilinear array, then ray–isosurface intersection is straightforward. Analogous simple schemes exist for the intersection of tetrahedral cells.

To find an intersection (see Figure 10), the ray $\mathbf{a} + t\mathbf{b}$ traverses cells in the volume, checking each cell to see if its data range bounds an isovalue. If it does, the system performs an analytic computation to solve for the ray



11 (a) The isosurface from the marching cubes algorithm. (b) The isosurface resulting from the true cubic behavior inside the cell.



12 For a given leaf cell in the rectilinear grid, indices to the shaded elements of the unstructured mesh are stored.

Table 1. Explicit bone isosurface extraction times.

Number of CPUs	Noise Build (seconds)	Noise Extract (seconds)	Marching Cubes (seconds)
1	4,838	110	627
2	2,109	81	324
4	1,006	56	171
8	885	31	93
16	437	24	49
32	118	14	26
64	59	12	24

parameter t at the intersection with the isosurface:

$$\rho(x_a + tx_b, y_a + ty_b, z_a + tz_b) - \rho_{iso} = 0 \quad (3)$$

When approximating ρ with a trilinear interpolation between discrete grid points, this equation will expand to a cubic polynomial in t. This cubic can then be solved in closed form to find the intersections of the ray with the isosurface in that cell. We use the closed form solution because its stability and efficiency haven't been major issues for the data we've used in our tests. The system examines only the roots of the polynomial contained in the cell. There may be multiple roots, corresponding to multiple intersection points. In this case, we use the smallest t (closest to the eye). There may also be no roots of the polynomial, in which case the ray misses the isosurface in the cell. Note that using trilinear interpolation directly will produce more complex isosurfaces than is possible with a marching cubes algorithm. Figure 11 shows an example of this, which illustrates case 4 from Lorensen and Cline's paper.[20] Techniques such as the Asymptotic Decider[21] could disambiguate such cases but they would still miss the correct topology because of the isosurface interpolation scheme.

Unstructured isosurfacing

For unstructured meshes, we use the same memory hierarchy as we did in the rectilinear case. However, we can control the cell size's resolution at the finest level. We chose a resolution that uses approximately the same number of leaf nodes as there are tetrahedral elements. The leaf nodes store a list of references to overlapping tetrahedra (see Figure 12). For efficiency, we store these lists as integer indices into an array of all tetrahedra.

Rays traverse the cell hierarchy in a manner identical to the rectilinear case. However, when the system detects a cell that might contain an isosurface for the current isovalue, it tests each of the tetrahedra in that cell for intersection. The system doesn't use connectivity information for the tetrahedra; instead it treats them as independent items, just as in a traditional surface-based ray tracer.

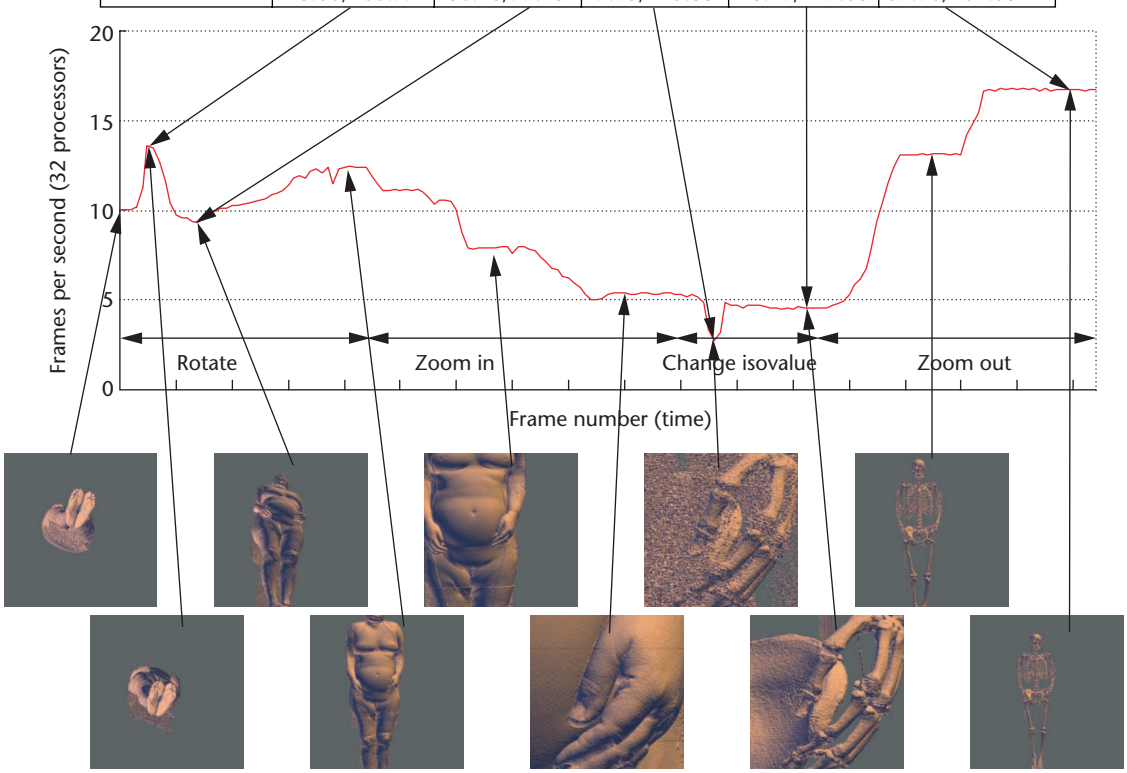
The system computes the isosurface for a tetrahedron implicitly by using barycentric coordinates. It computes the intersection of the parameterized ray and the isoplane directly, using the implicit equations for the plane and the parametric equation for the ray. The system checks the intersection point to see if it's still within the bounds of the tetrahedron by ensuring that the barycentric coordinates are all positive. Parker et al. describe the details of this intersection.[4]

Discussion

We contrasted applying this algorithm to explicitly extracting polygonal isosurfaces from the Visible Woman data set. For the skin isosurface, we generated 18,068,534 polygons. For the bone isosurface, we generated 12,922,628 polygons. These numbers are consistent with those reported by Lorensen given that he used a cropped version of the volume.[22] With this number of polygons, it would be challenging to achieve interactive rendering rates on conventional high-end graphics hardware.

Our method can render a ray-traced isosurface of this

Number of processors	Frame rate/Speedup				
1	0.427/1.00	0.304/1.00	0.084/1.00	0.155/1.00	0.568/1.00
2	0.84/1.97	0.60/1.98	0.17/1.99	0.31/2.00	1.13/1.98
3	1.26/2.94	0.89/2.93	0.25/2.95	0.46/2.96	1.68/2.96
4	1.67/3.91	1.19/3.92	0.33/3.96	0.62/3.97	2.24/3.94
6	2.45/5.73	1.76/5.77	0.50/5.97	0.93/5.96	3.29/5.80
8	3.20/7.50	2.32/7.61	0.67/7.94	1.23/7.93	4.36/7.67
12	4.81/11.26	3.44/11.30	1.00/11.89	1.84/11.88	6.51/11.47
16	6.38/14.93	4.59/15.08	1.33/15.84	2.45/15.80	8.64/15.21
24	9.54/22.33	6.84/22.48	1.98/23.54	3.65/23.49	12.92/22.76
32	12.65/29.61	9.12/29.96	2.63/31.38	4.88/31.47	17.09/30.10
48	18.85/44.13	13.52/44.39	3.92/46.72	7.30/47.02	25.27/44.50
64	24.73/57.90	17.72/58.19	5.18/61.78	9.64/62.14	32.25/56.80
96	35.38/82.82	25.04/82.23	7.67/91.38	14.28/92.02	45.50/80.14
124	43.06/100.79	30.28/99.45	9.73/115.88	18.17/117.08	57.70/101.63



13 Variation in frame rate as the viewpoint and isovalue changes.

data at roughly 10 frames per second using a 512 × 512 image on 64 processors. Table 1 shows the extraction time (in seconds) for the bone isosurface using both near-optimal isosurface extraction (Noise)[23] and marching cubes.[20] Note that these numbers would improve if we used a dynamic load balancing scheme. However, this wouldn't let us interactively modify the isovalue while displaying the isosurface. Using a downsampled or simplified detail volume would allow interaction at the cost of some resolution. Simplified, precomputed isosurfaces could also yield interaction but storage and precomputation time would be significant. Triangle stripping could improve display rates by up to a factor of three because isosurface meshes are usually transform bound. Note that we gain efficiency for both the extraction and rendering components by not explicitly extracting the geometry. Therefore, our algorithm doesn't suit applications that will use the geometry for nongraphics purposes.

Our system's interactivity lets us explore the data by interactively changing the isovalue or viewpoint. For example, you could view the entire skeleton and interactively zoom in and modify the isovalue to examine the detail in the toes all at about 10 fps. Figure 13 shows the variation in frame rate.

Conclusions

We described two different algorithms for rendering large-scale data sets on a parallel machine. Each of these algorithms have their own strengths. We've pushed the first algorithm to a higher number of processors (512), while the second is limited by the number of processors available in a shared-memory system (currently tested

to 128). For larger image sizes, the first algorithm may be more appropriate, as large images will slow down the ray-tracing process. However, for large data sets and smaller images, the ray-tracing method can outperform high-end graphics hardware even on a modest number of processors.

Parallel rendering is reaching a stage where it's usable for interactive rendering of large-scale data sets, not just for offline generation of high-quality images. The two algorithms we presented here are just a sample of those that are forming a reality of interactive large-scale scientific visualization. We anticipate data sizes will continue to grow at such a rate that direct parallel rendering alone would not suffice to guarantee interactive rendering. Additional techniques such as compression should be applied whenever possible to accelerate both rendering and communication. A multiresolution framework is particularly desirable when there is a mismatch between the data size and the processor size. That is, when the number of processors available can't render the data at the full resolution, the user should be allowed to choose interactivity over accuracy by using a coarser version of the data. Finally, we'll extend these algorithms to less expensive commodity workstation clusters. The distributed memory architecture and the relatively slow communication networks associated with such clusters will require novel new techniques to achieve interactive rates for large data sets. ■

Acknowledgments

The Visible Woman data set is available through the National Library of Medicine as part of its Visible Human Project. The Los Alamos National Laboratory released the CT data set, which was generated with FlashCT, a software product of Hytec. Dimitir Mavriplis provided the aerodynamic-flow data set. Peter Shirley helped generate several of the explanatory figures. Tom Crockett performed the tests on the 128-CPU Sun cluster.

References

1. K.-L. Ma and T.W. Crockett, "A Scalable, Cell-Projection Volume Rendering Algorithm for 3D Unstructured Data," *Proc. 1997 Symposium on Parallel Rendering*, IEEE CS Press, Los Alamitos, Calif., 1997, pp. 95-104.
2. S. Parker et al., "Interactive Ray Tracing for Isosurface Rendering," *Proc. Visualization 98*, CD-ROM, ACM Press, New York, Oct. 1998.
3. M. Levoy, "Display of Surfaces from Volume Data," *IEEE Computer Graphics and Applications*, vol. 8, no. 3, May 1988, pp. 29-37.
4. J. Kniss et al., "Interactive Texture-Based Volume Rendering for Large Data Sets," *IEEE Computer Graphics and Applications*, vol. 21, no. 4, Jul./Aug. 2001, pp. 52-61.
5. E.B. Lum, K.-L. Ma, and J. Clyne, "Texture Hardware Assisted Rendering of Time-Varying Volume Data," to appear in *Proc. IEEE Visualization 2001,* CD-ROM, ACM Press, New York, Oct. 2001.
6. K.-L. Ma and D. Camp, "High Performance Visualization of Time-Varying Volume Data over a Wide-Area Network," *Proc. Supercomputing 2000 Conf.*, CD-ROM, ACM Press, New York, 2000.
7. C. Hofsetz and K.-L. Ma, "Multi-Threaded Rendering Unstructured-Grid Volume Data on the SGI Origin 2000," *Proc. Third Eurographics Workshop on Parallel Graphics and Visualization*, Eurographics Assoc., Switzerland, 2000, pp. 133-140.
8. J.T. Kajiya, "An Overview and Comparison of Rendering Methods," *A Consumer's and Developer's Guide to Image Synthesis*, ACM Siggraph 1988 Course 12 Notes, ACM Press, New York, 1988, pp. 259-263.
9. E. Reinhard, A.G. Chalmers, and F.W. Jansen, "Overview of Parallel Photo-Realistic Graphics," *Proc. Eurographics 98*, Eurographics Assoc., Switzerland, 1998, pp. 1-25.
10. K.-L. Ma et al., "Parallel Volume Rendering Using Binary-Swap Image Composition," *IEEE Computer Graphics and Applications,* vol. 14, no. 4, July 1994, pp. 59-68.
11. G. Vézina, P.A. Fletcher, and P.K. Robertson, "Volume Rendering on the MasPar MP-1," *Proc. 1992 Workshop on Volume Visualization*, Oct. 1992, pp. 3-8.
12. M.J. Muuss, "Towards Real-Time Ray-Tracing of Combinatorial Solid Geometric Models," *Proc. Ballistic Research Laboratories Computer-Aided Design (BRL-CAD) Symp.*, Army Research Lab, Adelphi, Md., 1995.
13. S. Whitman, *Multiprocessor Methods for Computer Graphics Rendering*, Jones and Bartlett Publishers, London, 1992.
14. J. Amanatides and A. Woo, "A Fast Voxel Traversal Algorithm for Ray Tracing," *Proc. Eurographics 87*, Eurographics Assoc., Switzerland, 1987, pp. 3-10.
15. M.B. Cox and D. Ellsworth, "Application-Controlled Demand Paging for Out-of-Core Visualization," *Proc. Visualization 97*, ACM Press, New York, Oct. 1997, pp. 235-244.
16. J. Arvo and D. Kirk, "A Survey of Ray Tracing Acceleration Techniques," *An Introduction to Ray Tracing*, A.S. Glassner, ed., Academic Press, San Diego, 1989.
17. K.S. Klimansezewski and T.W. Sederberg, "Faster Ray Tracing Using Adaptive Grids," *IEEE Computer Graphics and Applications*, vol. 17, no. 1, Jan./Feb. 1997, pp. 42-51.
18. J. Wilhelms and A. Van Gelder, "Octrees for Faster Isosurface Generation," *ACM Trans. Graphics*, vol. 11, no. 3, July 1992, pp. 201-227.
19. A. Globus, *Octree Optimization*, tech. report RNR-90-011, NASA Ames Research Center, Moffett Field, Calif., 1990.
20. W.E. Lorensen and H.E. Cline, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," *Computer Graphics* (Proc. Siggraph 87), vol. 21, no. 4, ACM Press, New York, July 1987, pp. 163-169.
21. G. Nielson and B. Hamann, "The Asymptotic Decider: Resolving the Ambiguity in Marching Cubes," *Proc. Visualization 91*, IEEE CS Press, Los Alamitos, Calif., 1991, pp. 83-91.
22. W.E. Lorensen, *Marching through the Visible Woman*, http://www.crd.ge.com/cgi-bin/vw.pl, 1997.
23. Y. Livnat, H. Shen, and C.R. Johnson, "A Near Optimal Isosurface Extraction Algorithm Using the Span Space," *IEEE Trans. Visualization and Computer Graphics*, vol. 2, no. 1, Mar. 1996, pp. 73-84.

Kwan-Liu Ma *is an associate professor of computer science at the University of California, Davis, where he teaches and conducts research in computer graphics and scientific visualization. His career research goal is to improve the overall experience and performance of data visualization through more effective user-interface designs, interaction techniques, and high-performance computing. He received his PhD in computer science from the University of Utah in 1993. He served as co-chair for the 1997 IEEE Symposium on Parallel Rendering, Case Studies of the IEEE Visualization Conference in 1998 and 1999, and the first National Science Foundation/US Department of Energy Workshop on Large-Scale Data Visualization. He recently received the Pecase award for his work in parallel visualization and large-scale data visualization.*

Readers may contact Ma at the Dept. of Computer Science, Univ. of California, Davis, One Shields Ave., Davis, CA 95616-8562, email ma@cs.ucdavis.edu.

Steven Parker *is currently a research assistant professor in the School of Computing at the University of Utah. Additionally, he is a faculty member at the university's Scientific Computing and Imaging (SCI) Institute. His primary research interests are in component architectures, large-scale scientific visualization, and high-performance computing. Parker is the primary developer of the problem-solving environment SCIRun, which was the basis of his PhD dissertation and is also the foundation for a number of projects at the SCI Institute. He is also the lead software architect for the Center for Accidental Fires and Explosions at the University of Utah, where he is developing a problem-solving environment for a complex multiphysics simulation running on thousands of processors. He received a PhD in computer science from the University of Utah.*

Coming Soon

IEEE Visualization 2001
San Diego, California, USA
21–26 October 2001
This conference focuses on interdisciplinary methods and collaboration among developers and users of visualization methods across all areas of science, engineering, medicine, and commerce. The Symposium on Information Visualization 2001 (InfoVis 2001) and the Symposium on Parallel and Large-Data Visualization and Graphics (PVG 2001) will also be held in conjunction with IEEE Visualization 2001 this year. Visit http://vis.computer.org for more information.

SPIE Conference on Visualization and Data Analysis 2002
San Jose, California, USA
20–25 January 2002
This conference covers all aspects of visualization and issues affecting successful visualizations. Example topics include biomedical visualization and applications, image processing, data explorations using classical and novel approaches, medical imaging, interactive platforms, and virtual environments. Visit http://www.futurevisions.net/SPIE/vda2002 or email spie@spie.org for more information.