Volume Rendering on a Distributed Memory Parallel Computer

T. Todd Elvins
San Diego Supercomputer Center
Advanced Scientific Visualization Laboratory
P.O. Box 85608
San Diego CA 92186-9784

Abstract

Volume rendering ideally produces high-quality images at interactive rates. High CPU and memory requirements make this goal unreachable with current off-the-shelf technology. Exploiting highly parallel computers is one way that future systems may approach acceptable speeds. This paper discusses the adaptation of a known volume rendering algorithm to a new commercially available distributed memory MIMD parallel architecture.

1. Introduction

Scientific Visualization uses computer graphics techniques to give scientists greater insight into their scientific data. One sub-field of scientific visualization is volume visualization, the process of projecting a multi-dimensional dataset onto a two-dimensional image plane for the purpose of studying the structures contained within the volumetric data. To be useful, volume visualization techniques must allow scientific users to change viewing, lighting, and data classification parameters and to see the resultant image in a short amount of time. Scientists need these techniques not only for their own insight, but also to share their results with their colleagues, the institutions that support the scientist’s research, and the general public.

Most volume visualization software packages operate in dedicated-mode. Dedicated-mode software runs all of its components on the workstation-class machine in front of the user regardless of the capabilities of the machine. The workstation-class machines that most scientists have access to, however, are not sufficiently powerful to quickly generate high-quality volume renderings, and some datasets are so large that they cannot be imaged on even the largest workstation. One advantage of the dedicated-mode approach is that partial solutions can be stored in memory along the volume-rendering pipeline so that the incremental images can be generated with fewer calculations. This advantage will be discussed in more detail in the next section.

An alternative volume visualization approach is to use the scientist’s workstation-class machine as a visualization client and run the compute-intensive components on a remote specialized visualization server. Images are rendered and displayed on the scientist’s workstation in a much shorter time than if they were created by the workstation itself. This makes the scientist’s workstation appear more powerful than it really is. The San Diego Supercomputer Center (SDSC) has adopted this client/server approach in its NetV Distributed Volume Visualization package [Elv91]. An Alliant FX/2800 is the primary NetV volume rendering server. When the demand for the Alliant becomes too great, NetV will be revamped to distribute volume rendering jobs to alternative volume rendering servers, such as the nCUBE 2, based on the characteristics of the jobs.

In volume visualization images are typically created in one of two ways; using an image-order traversal of the pixels in the image plane or an object-order traversal of the volume elements (voxels) in the volumetric dataset. See [Deeb88] or [Elv92] for a more thorough overview of volume visualization techniques.

An image-order traversal usually casts one or more rays from each pixel through the dataset, then resamples the data at intervals along the ray to calculate the color of each pixel [Levo88]. The advantage of this approach is that all of the rays can be cast independently [Bado90]. However, it is difficult to implement for a distributed memory architecture because either the entire dataset has to be stored at every processor, or some method of passing rays or samples between processors has to be implemented. Since the number of rays and the number of samples are extremely large, this is not a practical solution.

Object-order traversals operate by calculating the projection and contribution that each voxel makes to the pixels in the image plane. This approach sometimes uses
intermediate surface primitives to approximate iso-value contour surfaces in the volumetric dataset [Wyi86][Lor87]. Surface-fitting is a powerful technique for some types of data but suffers from several problems such as occasional false positive and negative surface pieces, poor handling of amorphous data, and incorrect handling of small features and branches in the data. Two other object-order algorithms include the V-Buffer method [Upso88] and the more recent splatting method [West90], both of which perform a front-to-back object-order traversal of the data volume and composite voxel contributions into an image accumulation buffer in image-order. Both the V-Buffer and splatting algorithms were reported by their inventors to be readily parallelizable.

This paper describes a prototype implementation of a splatting volume renderer (SVR) on a commercially available distributed memory MIMD parallel processor.

2. The platform

The purpose of this project was to investigate the possibility of using an nCUBE 2 parallel computer as an alternative NetV volume rendering server. nCUBE Corporation builds distributed memory MIMD machines based on a proprietary processor. There are more than 250 nCUBE systems installed worldwide with a maximum installed nCUBE configuration of 1024 processors. The SDSC nCUBE 2 has 128 processors. 64 of these processors have 4MB of local memory, and the remaining processors have 16MB of local memory. There is no shared memory on the nCUBE. The peak speed of one processor is 3.3Mflops and the peak processor-to-processor communication rate is 2MB/sec. nCUBE computers are hosted by and accessed from a Sun SPARCSStation.

The nCUBE compiler includes command-line directives to control heap, stack, and communication buffer (comm-space) size. The sum of the sizes of the three must fit in the processor's local physical memory in nCUBE90. The nCUBE processors do not have virtual memory. Heap is the area where program memory is dynamically allocated and deallocated. Stack is where the program and fixed-size variables are kept. Comm-space is the area where processor-processor and processor-host communication messages are staged.

Since the architecture has no shared memory, message-passing is necessary to get data resident on one processor's memory to another processor's memory. The comm-space must be declared at compile time to be large enough to hold the sum of all messages passed to an individual processor. Using a message-passing protocol that limits the number of messages in a processors read-queue allows a small comm-space to be used. Dedicating a small portion of memory to comm-space leaves more memory free for heap space which is desirable for most computer graphics programs.

3. Splatting

The procedure is called splatting because it can be likened to throwing a snowball (voxel) at a glass plate. The snow contribution at the center of impact will be high and the contribution will drop off further away from the center of impact. A serial version of an SVR was developed as a basis for the future parallel version of the code. An SVR proceeds in several steps. The first step determines in what order to traverse the volume. By passing the coordinates of the corners of the volume through the viewing matrix, the face of the volume and corner of the face closest to the image plane can be found. Voxels are splatted according to their distance from the image plane, with the closest voxel being splatted first. The voxels in one slice are all splatted before the next slice is started. This traversal ensures that voxels close to the viewpoint obscure voxels far from the viewer.

Each voxel in each slice is visited. The value at the voxel is checked in user-specified classification tables to determine what substance is contained in the voxel. The substance's color and opacity values are used to determine a voxel's contribution to the image. For example, the user may have specified that values in the range 200-255 are bone, are colored white, and have an opacity of 0.9 (nearly opaque). Next the voxel is shaded. The gradient at the voxel is calculated using a central difference formula and is used to approximate the normal for the voxel in the shading formula. The final shading value is used to attenuate the voxel's color tuple before it is splatted into the image plane.

The next steps project the voxel into image-space to find how the voxel's RGB tuple, and associated opacity, contribute to an image buffer. A round filter called a reconstruction kernel is used to find the pixel extent of this contribution. For orthogonal viewing, a circular kernel (usually a Gaussian) can be calculated once and used for all of the voxels. However, for perspective viewing, a new oblique kernel must be calculated for every voxel. The projection of the kernel into the image buffer is called a footprint. The size of the footprint is made proportional to the size of the volume and the size of the image to be generated, so that a small volume can fill a large image. Once the footprint is calculated, its center is placed at the center of the voxel's projection in the image buffer. Note that there is not necessarily going to be a pixel center at this location.

The shaded color tuple and opacity values are blended into the image buffer at every pixel that falls within the
area covered by the circular Gaussian footprint. Before a tuple is blended with a pixel, it is attenuated by the value of the Gaussian footprint at that particular pixel center. This has the effect of making voxel contributions higher when near the center of the projection and lower when far from the center. After all of the voxels have been splatted into the image buffer, the image is complete. When the opacity at a pixel in the image buffer reaches unity, then no further splats will have an effect on it.

Although the splatting technique operates much differently than either surface-fitting or ray-casting, it quickly produces high-quality images that are similar to the images produced by other algorithms.

4. Parallel Implementation

It is best to execute and control the programs running on the nCUBE from a program running on the nCUBE's host computer. Each nCUBE processor may be running a unique nCUBE executable or, as in this case, the processors may all be running the same executable.

The host program will not attempt to start the nCUBE if the dataset is too big. Additionally, the host will allocate only those processors required for the renderer to run. Since the number of processors to be used must be specified at compile time, a number of host programs are maintained for varying nCUBE configurations. The host program calculates approximately the sum of memory that will be needed to hold dataset slices and image contribution buffers. The host program then allocates a subcube of the processors with 4MB of memory, if this sum is less than 4MB, and a subcube of processors with 16MB of memory otherwise.

The final image is passed from the master nCUBE processor to the host in RGB tuple floating-point. The host computer puts the RGB values into a virtual frame-buffer [Nade91] and invokes a library routine that writes the pixelmap to an image file.

The initial nCUBE version of the SVR uses one nCUBE processor as a master processor. The master is responsible for reading the dataset from disk, distributing slices of data to a group of slave processors, and then collecting image contributions from the slavers after they finish splatting the slice data.

The master processor actually distributes three slices of data to each slave processor. This is because the slice in-front-of the slice to be splatted and the slice behind the slice to be splatted are needed to calculate gradients. After splatting the middle slice, each slave returns an image-size image contribution buffer to the master processor. If there are more slices than processors, then some slaves are given a second packet of three slices after the first packet is finished. If there are more processors than slices, then some processors are idle. The first and last slices in the data volume are not splatted because their gradients cannot be estimated with any certainty. It is important to note that the master must collect and composite image contributions in front-to-back order to arrive at a correct image.

Since each data-slice has to be accompanied by its two neighbors, the initial nCUBE implementation requires distributing each data slice to three processors (the exception to this is the first and last slice are only distributed to one processor and the second and second-to-last slices are only distributed to two processors). Splatting is into a 16-byte deep (one float for each of red, green, blue, opacity) image contribution buffer. Splatting each slice of 8-bit per voxel data into its own 16-byte per pixel image buffer, then passing the whole buffer to another processor, is clearly not a good final solution because the message-passing overhead is too high. The run-times of the initial implementation are shown in wall-clock time on line one in Table 1.

5. Optimizations

Several optimizations have been implemented to cut down on message-passing requirements.

5.1. Exploiting image coherence

The first optimization uses image coherence to reduce the size of the image contribution packets that have to be passed from the slave processors to the master processor. Often only a small, coherent portion of an image contribution buffer contains non-black, non-transparent pixels.

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>1. none</td>
<td>5:42</td>
</tr>
<tr>
<td>2. image bounding box</td>
<td>6:00</td>
</tr>
<tr>
<td>3. two master processors</td>
<td>NA</td>
</tr>
<tr>
<td>4. group slices</td>
<td>NA</td>
</tr>
<tr>
<td>5. unbuffered I/O</td>
<td>NA</td>
</tr>
</tbody>
</table>

Table 1. Timings for four additive optimizations. Time in minutes & seconds.
For many datasets, cutting the rectangular bounding-box containing the non-black pixels out of the image buffer, and passing this, along with the location it was cut from, results in a speed-up in message-passing. The compositing step on the master processor is also faster because a smaller number of pixels is being composited.

Line two in Table 1 shows the timings for a volume containing all non-black, semi-opaque voxels. Although this particular dataset does not exhibit coherent image properties, an overall speed-up of approximately 15% is realized for coherent images, such as those generated from CT data.

5.2. Splitting the master’s tasks

The second optimization splits the data-distribution and image-compositing tasks among two master processors. The reasons for this are twofold. The first reason is that the master processor runs out of physical memory for large data volumes. The entire dataset has to be in memory at one time so the volume can be sliced along any of the three axes. For this reason, the single master processor scheme has to maintain the entire dataset, an image contribution buffer, and a final image accumulation buffer in memory all at the same time.

The second reason for splitting the master processor’s tasks is that distributing data and collecting image contributions is too much work for one processor. Since the image contributions must be received and composited in a particular order, each slave processor waits until it receives a ready token from the master processor before it sends its image contribution. This wait is sometimes unreasonably long if the master is in the middle of distributing data.

The speed-up realized from splitting the master’s responsibilities between two processors is fairly dramatic for less than 64 processors. The rendering times after this optimization are shown on line three of Table 1.

5.3. Distributing groups of slices

The next optimization attempts to further reduce message passing requirements. Instead of sending one slice of data with its two adjacent slices, then gathering a number of image contributions equal to the number of slices (minus two), a group of slices is sent to each processor so that fewer data messages are sent and fewer image contribution messages are received.

In this version of the software, the slave processors each get one packet of \( n \) data slices, where \( n \) is the total number of slices divided by the number of slave processors. For example, if there are two slave processors and nine slices of data, the first processor gets the first six contiguous slices of data and the second processor gets the remaining five slices of data (the extra slices are for gradient calculations). Note again that the first and last slices are not splatted. A slave processor splats all of its data slices into one image contribution buffer and then passes this buffer back to a master processor. This scheme unchokes the image contribution message-passing bottleneck. The number of image contributions is reduced from the number of slices to the number of pro-

<table>
<thead>
<tr>
<th>Machine</th>
<th>Blob 15x15x15 byte data 3KB</th>
<th>GeoPhysics 51x33x38 byte data 98KB</th>
<th>Hipip 64x64x64 float data 1MB</th>
<th>Dolphin 200x180x91 byte data 3.2MB</th>
<th>Brain 256x256x90 short data 12MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sun 4/SPARC I</td>
<td>0:21</td>
<td>1:33</td>
<td>1:49</td>
<td>TooBig</td>
<td>TooBig</td>
</tr>
<tr>
<td>SGI 4D/320VGX</td>
<td>0:06</td>
<td>0:17</td>
<td>0:12</td>
<td>8:13</td>
<td>15:40</td>
</tr>
<tr>
<td>Alliant FX/2800</td>
<td>0:07</td>
<td>0:18</td>
<td>0:10</td>
<td>2:40</td>
<td>2:45</td>
</tr>
<tr>
<td>Cray YMP/864</td>
<td>0:05</td>
<td>0:21</td>
<td>0:23</td>
<td>2:13*</td>
<td>TooBig</td>
</tr>
<tr>
<td>nCUBE 4 proc</td>
<td>0:18</td>
<td>0:51</td>
<td>0:39</td>
<td>TooBig</td>
<td>TooBig</td>
</tr>
<tr>
<td>nCUBE 8 proc</td>
<td>0:18</td>
<td>0:25</td>
<td>0:23</td>
<td>TooBig</td>
<td>TooBig</td>
</tr>
<tr>
<td>nCUBE 16 proc</td>
<td>0:27*</td>
<td>0:26</td>
<td>0:28</td>
<td>1:48</td>
<td>TooBig</td>
</tr>
<tr>
<td>nCUBE 32 proc</td>
<td>0:48*</td>
<td>0:32</td>
<td>0:53</td>
<td>1:34</td>
<td>2:14</td>
</tr>
<tr>
<td>nCUBE 64 proc</td>
<td>1:37*</td>
<td>1:24*</td>
<td>1:55</td>
<td>2:05</td>
<td>2:53</td>
</tr>
</tbody>
</table>

Table 2. Wall-clock execution time given in minutes and seconds

(•) Time shown is time to execute. The job was submitted via the large-job batch queue.

(*) Since groups of data slices are distributed to processors, attempting to use more processors than there are data slices leaves many processors completely idle. There is overhead associated with starting up processors even if they are not used.
cessors. The times for this optimization are shown on line four of Table 1.

5.4. Unbuffered message-passing

The final optimization uses unbuffered message-passing instead of buffered message-passing. This requires additional message-passing coordination between processors but does not require copying each message into comm-space before sending it. Receiving processors must be waiting for the message. The speed-up using unbuffered message-passing is shown on line five of Table 1.

The nCUBE C compiler has four levels of compiler optimization. Level 0 is deemed very safe and level 3 is called experimental and is not guaranteed to work at all. All four levels of compiler optimization were tried with little speed-up gained. The program continued to function properly at every level.

6. Results

Table 2 shows nCUBE, Sun, SGI, Alliant, and Cray rendering times for five datasets. A serial version of the SVR was run on one processor on the Sun, SGI, Alliant, and Cray. The times in table 2 are for a relatively small (200x200 pixel) image. Not all datasets could be run on all nCUBE configurations due to physical memory limitations on the individual processors. Note that the 32 processor configuration is faster than the other architectures on the dolphin dataset and on the brain dataset. Although the nCUBE rendering times are approximately the same as the rendering times on the more traditional architectures in most cases, these initial timings do demonstrate that parallel volume rendering offers potential. Future parallel volume rendering implementations will probably be most useful for imaging larger datasets where the benefit of having additional rendering processors best justifies the cost of passing data to and from those processors.

The timings in Table 1 show that each nCUBE processor must be given about eight slices of data to overcome the message-passing overhead, the processor-starting overhead, and to keep the processors from being data-starved. If a processor is given fewer than eight slices of data to splot, the processor will be able to render the slices faster than it will be able to send its image contribution buffer to a master processor.

7. Future work

Lee Westover made a correction to his original splotting algorithm in his recent thesis [West91]. It deals with how image contribution buffers are accumulated. This correction should be incorporated in the SDSC SVR.

The maximum gradient magnitude should be found and broadcast to all processors so that normalized gradient magnitudes can be used to attenuate splats. This has the effect of detecting iso-surfaces in the volume.

Instead of performing all of the image contributions composited on the master processor, images should be composited in a hierarchical fashion as image contribution buffers are passed up a tree of processors.

An interesting experiment would be to try running the volume renderer in dedicated-mode. This method leaves all of the data on the processors and just changes rendering parameters such as the viewing, lighting, or data classification. In this way, the processors are only started once, the data is only distributed once, and the gradients are only calculated once. The SVR currently runs in a batch mode because that fits best with the SDSC NetV distributed volume visualization scheme.

Restricting the view to axis-aligned parallel projections would ensure that a horizontal slice of data perpendicular to the image plane would project into a few rows of pixels. The number of rows of pixels would be the same as the height of the splot kernel. Passing a few rows of pixels between processors would consume a small fraction of the message-passing time that is spent passing entire image pixel blocks.

8. Conclusion

Some relatively good rendering times are achieved with the nCUBE SVR. Message-passing bottlenecks occur when large numbers of floating-point values have to be collected from every processor for every picture. For large images this is a severe limitation. This initial implementation of a SVR on a distributed memory parallel computer demonstrates the need for parallel computers with high-bandwidth connections between processors and also for new parallelizable volume rendering algorithms.

An increasing number of CPU-intensive visualization tasks will be handled in the future by adapting existing visualization algorithms to take advantage of powerful new parallel computers.

9. References

Acknowledgements

This work was supported by the National Science Foundation under grant ASC8902825 and ASC9008413 to the San Diego Supercomputer Center. Additional support was provided by the State of California.

Special thanks to the director of the SDSC visualization group, Mike Bailey, and the rest of the SDSC viskids, Phil Mercurio, Dave Nadeau, Jim McLeod, Zahid Ahmed, Richard Frost, Kevin Landel, and Charlotte Smart.

Thanks to Ulrich Neumann at the University of North Carolina Chapel Hill for his helpful suggestions and to Chuck Niggley and Brent Swartz at nCUBE for their patience and support.

Figure 1: Dolphin head. Ted Cranford, University of California, Santa Cruz; Todd Elvins and Phil Mercurio, SDSC.

Figure 2: Geophysics simulation data. Louise Pellerin and Gerald Hohmann, University of Utah; Todd Elvins, SDSC.

(See color plates, p. CP-12.)