

Parallel Volume Rendering Unstructured Data: A DISTRIBUTED APPROACH

Kwan-Liu Ma

Institute for Computer Applications in Science and Engineering

Mail Stop 132C

NASA Langley Research Center

Hampton, VA 23681-0001

kma@icase.edu

Abstract

The development of effective parallel rendering algorithms for unstructured volume data is challenging due to the irregular and adaptive nature of the corresponding meshes. Most of the algorithms developed previously have been mainly for shared-memory architectures. Only a distributed approach can better meet the computational and memory requirements of the rendering calculations. This paper presents a volume rendering algorithm that distributes both the data and the rendering process among the processors. At each processor, ray-casting of local data is performed independent of the other processors. The global image compositing processes, which require inter-processor communication, are overlapped with the local ray-casting processes to achieve better parallel efficiency. In theory, this algorithm should attain high parallel efficiency but its implementation on the Intel Paragon shows otherwise. Besides the added ray-casting overhead, a critical factor is the imbalanced load due to the highly adaptive nature of typical unstructured meshes and the selection of transfer functions. The causes, effects and possible cures of the imbalanced load are studied.

1 Introduction

Unstructured meshes are used to model scientific and engineering problems with complex geometries; an example is the design of aircraft. The problem domain is adaptively decomposed into small cells, called elements. Popular element types include the tetrahedron, triangular prism and hexahedron. While adaptive techniques provide the promise of reliably and efficiently solving a problem to the desired level of accuracy, the resulting irregular meshes introduce additional difficulties to visual interpretations of the solution data.

Many visualization techniques have been developed for the interrogation and analysis of unstructured data. While exterior face rendering and cutting plane methods remain the most common and affordable techniques, three-dimensional methods such as direct volume rendering have received considerable attention because they can capture the overall data domain in a single image, and are capable of revealing complex features in the data that traditional surface-oriented graphics techniques fail to represent.

As computing technology continues to advance, the level of complexity and the resolution of the problems that can be studied increase. For example, a typical aerodynamics calculations, can generate data with hundreds of thousands of elements or more. The absence of a simple indexing scheme for three-dimensional unstructured meshes makes direct volume rendering a computationally expensive process. Since parallel processing enables the solution of many other compute-intensive problems, computer graphics and visualization researchers have also been exploiting various parallel methods for volume rendering.

This paper describe a parallel volume ray-casting algorithm for visualizing unstructured data. This algorithm is completely distributed which distinguishes from previous efforts. Both the data and the rendering computation are distributed across the available processing nodes. Inter-processor communication is only needed for the image compositing step. At each processor, ray-casting of local data is performed independent of other processors. Image compositing is overlapped with the ray-casting processes to achieve higher parallel efficiency. The algorithm is also very flexible. Although the current implementation handles only tetrahedral cells, the algorithm can be generalized to handle a mix of cell types and arbitrary object geometry, such as objects with holes and concavities.

Tests are performed on the Intel Paragon, a distributed-memory MIMD parallel computer, by using from two to 128 processors. The purpose of the testing is to study how characteristics of unstructured mesh, data (mesh) partitioning, and transfer functions affect the parallel rendering efficiency when using the algorithm proposed here. When a volume mesh is partitioned for parallel processing, the area of boundary surface increases and so does the corresponding ray-casting cost, since a ray begins at the boundary surface. Therefore, it is not possible to achieve optimal parallel efficiency. Moreover, both data partitioning and the selection of transfer functions determine the actual computational load on each processor. Test results show that imbalanced load further degrades the rendering efficiency. We investigate a static load balancing technique and show that it fails to improve performance sufficiently. A summary of the test results suggests a dynamic

approach for load balancing. This paper provides useful hints for future implementors of an unstructured volume renderer on parallel distributed-memory architectures.

2 Visualizing Unstructured Volume Data

Data visualization on unstructured meshes requires multiple steps to obtain maximum efficiency due to the irregularity of the meshes. Yarmarkovich and Gelberg [25] describe *preprocessing* methods to achieve interactive visualization for techniques like exterior-face rendering, slicing and iso-surface rendering. Gallagher and Nagtegaal [4] present an algorithm based on the *marching cubes* iso-surface extraction method [10]. However, unlike the basic marching cubes algorithm, the surface patches derived are represented by parametric bi-cubic polynomials to achieve visually smooth appearance. These surface patches are then subdivided into planar polygons for rendering using hardware Gouraud shading. Koyamada and Nishio [9] describe two approaches for extracting iso-surfaces from tetrahedral elements. One is based on the *marching cubes* algorithm and the other is a ray-tracing method. In the ray-tracing approach, a diffusive iso-surface is made by raising the opacity in the region containing the iso-value. More recently, Shen and Johnson [17] develop a fast iso-surface extraction algorithm for interactive probing of scalar fields.

For direct volume rendering, there are generally two approaches: ray-tracing and projection. Garrity [5] uses a ray-tracing approach for rendering tetrahedra. Elements of other types are handled by first subdividing them into tetrahedra. For each ray, exterior faces are tested to find the first intersection point, and subsequent intersection points can be efficiently calculated by using the *connectivity* between elements. Giertsen [6] proposes a different ray-tracing approach by slicing the data along each horizontal scanline. The intersection of the slicing plane and the data elements results in a set of planar polygons, which are triangulated. Each resulting triangle is broken into segments and pixel-aligned segments are composited to form an image.

In the projection approach, elements are first sorted in visibility order. Then each element is projected onto the screen in either front-to-back or back-to-front order. The element's contribution to each pixel is calculated and blended with existing values. Williams [23] develops an algorithm for determining the visibility order of elements. Max, Hanrahan and Crawfis [14] describe an accurate, but computationally expensive, analytic illumination model for use with their projection algorithm. Shirley and Tuchman [18] propose a splatting algorithm which provides a fast approximation to the projection process. Williams [22] further approximates Shirley and Tuchman's splatting algorithm to achieve better interactive rendering rates. These fast approximation methods are suggested for use in data previewing, rather than producing realistic or accurate visualization.

On the other hand, the development of massively parallel rendering algorithms for irregular data has been rather sparse. Notably, Williams [24] developed a cell-projection volume rendering algorithm for finite element data running on a single SGI multiprocessor workstation. Uelton [21] implemented a volume ray-tracing algorithm for curvilinear grids on a similar platform.

The tests were performed for up to eight processors and high parallel efficiency was obtained. Challenger [2] developed a parallel volume ray-tracing algorithm for nonrectilinear grids and implemented it on the BBN TC2000, a multiprocessor architecture with up to 128 nodes. Note that all three of these used shared-memory parallel computers. Finally, Giertsen and Petersen [7] designed a scanline volume rendering algorithm based on Giertsen's previous work [6] and implemented it on a network of workstations. Data are replicated on each workstation, and a master-slave scheme is used to achieve dynamic load balance. However, tests were performed with a maximum of four workstations, so the scalability of the algorithm and the implementation for massively parallel processing has yet to be demonstrated.

Next section gives an overview of the basic parallel algorithm developed in this research for rendering unstructured volume data. The same algorithm and its preliminary test results have been reported in [11]. This paper offers a more thorough discussion and, at the same time, focuses more on how data partitioning and the imbalanced load affect the overall rendering performance.

3 Overview of the Basic Algorithm

Figure 1 depicts the parallel rendering process which consists of multiple steps:

- data partitioning
- data distribution
- view-independent preprocessing
- view-dependent preprocessing
- local rendering
- global image compositing
- image display

Each step is described in the following sections, except data distribution and image display which are ignored in current research. In addition, we presently consider rendering as a postprocess only. For runtime visualization, rendering data in place on the same computer where the parallel simulation runs involves additional considerations; for example, data partitioning should comply with and be done by the simulation. As described in [12], these considerations do not change the basic algorithm and are not discussed here. Handling meshes that change dynamically at runtime would be a future research topic.

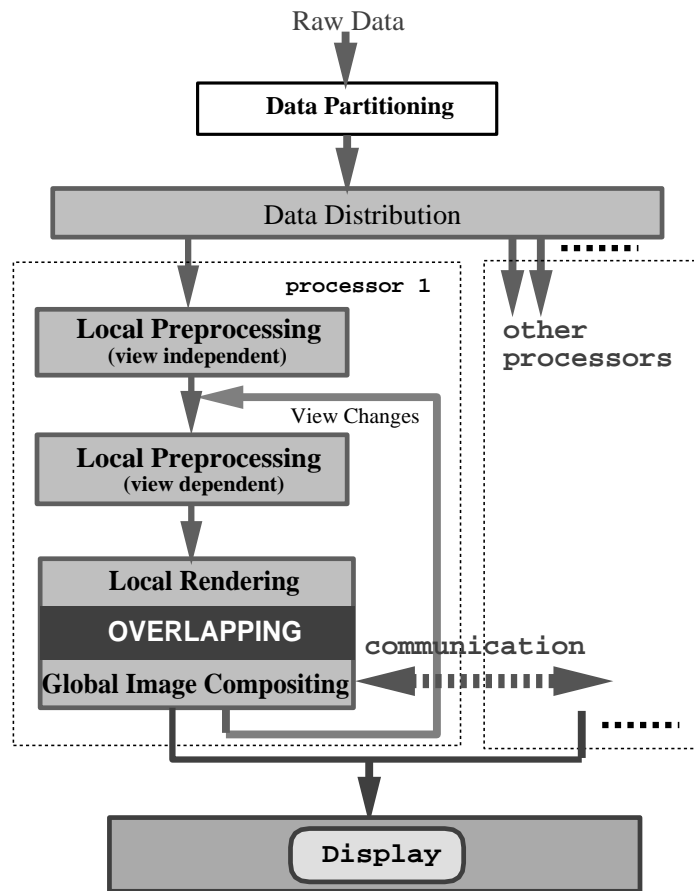


Figure 1: A Distributed Parallel Rendering Process.

3.1 Data Partitioning

Partitioning of the computational domain and its associated data structures is also an important problem for computational researchers implementing large-scale unstructured-mesh calculations on parallel distributed-memory computers. A partition of the domain into subdomains leads to a decomposition of the data as well as tasks associated with the computational problem, and the resulting partitions are mapped to individual processors. Most of the partitioning algorithms developed are recursive and based on graph bisection [8]. In essence, the computational domain, represented as an undirected graph, is subdivided into two subdomains based on some criterion; then the same criterion is applied to the subdomains recursively. A good partitioning for parallel simulation results in subdomains of about equal size (e.g. the same number of elements or vertices) and minimizes boundary area between subdomains. While the first property helps load balancing, the second reduces inter-processor communication. Alternatively, a weight can be assigned to each vertex in the mesh and then partitioning can be done according to, for example, the size of surface area or solution values at each vertices.

To take full advantage of a distributed-memory parallel computer like the Intel Paragon, the first step is to partition the data volume into p subvolumes, where p is the number of processors used. A perfect subdivision should produce subvolumes with identical memory and processing requirements to achieve good load balancing. Nevertheless, the requirements of visualization calculations are generally different from the simulation's. The criterion used by the parallel simulation may not be applicable. Essentially, the objection functions to be considered which may affect the performance of a parallel distributed renderer include:

- the number of elements (or vertices)
- the size of the subdomain
- the number of boundary vertices
- the shape of the subdomain (e.g. the aspect ratios)
- the associated solution values

The number of elements mainly determines the amount of memory space needed on each processor. For large data sets, memory load must be as even as possible just to make the rendering possible. The size of the subdomain and the number of boundary vertices are somewhat related. As described later, local rendering begins at the boundary of a subvolume. More rays are cast for larger boundary area. The shape of the subdomain also has some influence to the overall rendering process. a narrow shape in one direction reduces the amount of coherence of which the rendering process (both the ray-casting and ray-compositing process) can take advantage. Finally, the associated solution values are mapped to color and opacity values. Since a ray entering an opaque element terminates immediately, work load changes with different mappings.

In this research, a graph-based data-partitioning software package is used. The major objective function used is the number of elements. As shown later, although reasonably even subvolumes (i.e. subvolumes containing about the same number of elements) are obtained, computational load is imbalanced because of other factors.

3.2 Data Preprocessing

An unstructured data set is composed of at least a set of nodes and a list of elements that are constructed from the nodes. At each node, its coordinates $[x, y, z]$ and some function values like density or pressure are stored. The existing element-node relationship along is insufficient for efficient visualization operations. A preprocess step is used to derive other relationships in the mesh. Generally, a hierarchical data structure of three layers is created. A node set is a data structure at the lowest layer, from which a face set is constructed. The top layer of the hierarchy is an element set constructed from the face set. Therefore, the resulting data structure records the face-node, element-face, and element-node relationships, which allow fast information retrieval in the expense of additional memory space.

The subsequent ray-casting operations always begin at the boundary of the unstructured domain, which is formed by the exterior faces. An exterior face is a face that is not shared by elements. As a result of data partitioning, there are two types of exterior faces: globally and locally. Globally exterior faces represent the boundary of the whole volume. Locally exterior faces form the boundary of each subvolume. After data partitioning, while a globally exterior face is always a locally exterior face, a globally interior face might become a locally exterior one. Since our algorithm need not distinguish the globally exterior faces from the local ones, for the rest of the discussion, when we mention an exterior face, we mean a locally exterior face.

According to the current viewing position, an exterior face can be either a visible or an invisible one. A ray enters a subvolume at a visible exterior face and exits at an invisible exterior one, as shown in Figure 2. The visible exterior faces are orthographically projected onto the screen. This projection produces a set of convex polygons in screen coordinates, which define the exact screen area for casting rays. So it is necessary to distinguish the visible exterior faces from the invisible ones.

It is advantageous to separate the view-dependent preprocessing from the view-independent preprocessing calculations. Therefore, after data are distributed to processors, each processor performs two preprocessing steps independent of other processors. The first step, the view-independent one, finds connectivity between elements, identifies all exterior faces, calculates face normals, etc. This step needs to be done only once. The view-dependent step then follows to extract all visible faces, based on the current viewing direction, from the set of exterior faces; whenever the viewing position is changed, the exterior face list is searched to find a new set of faces visible from the current viewing position.

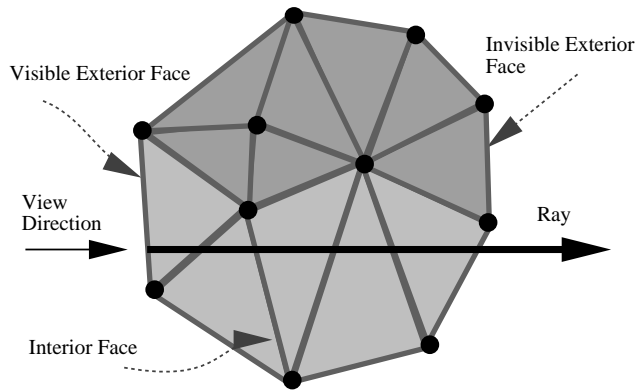


Figure 2: Face Types in a Subvolume.

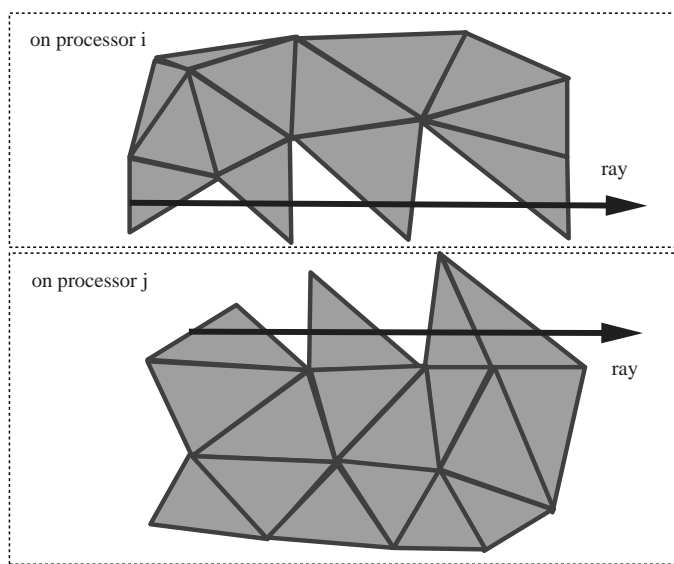


Figure 3: Rays Passing Saw-toothed Boundaries.

3.3 Local Rendering: Ray-Casting

The local rendering process traverses the visible-face list and performs ray-casting in scanline order from face to face. A ray is cast from the eye, enters the data domain through an exterior face and marches element by element until it exits from the domain through another exterior face, or it is terminated as the accumulated opacity values exceed a predefined threshold value. The irregular shapes common to unstructured data often result in concavity and holes in the body of the data volume. Especially for distributed computing, the original mesh is partitioned into submeshes that often have saw-toothed boundaries as shown in Figure 3 and the left image of Color Plate 1. Thus a ray may enter and exit multiple exterior faces. With the connectivity information, the casting of a ray is straightforward except for some degenerate elements [5].

The intensity of the ray is obtained by accumulating the intensity values contributed by all

elements visited. For each element, the equation for intensity $I(a, b)$ of the corresponding ray segment is given by

$$I(a, b) = \int_a^b e^{-\int_a^t \sigma(s) ds} I(t) dt$$

where a and b define where the ray enters and leaves the element, σ is the attenuation coefficient, and $I(t)$ is the intensity at a point t along the ray. In [14], analytic formulas are derived to compute the intensity accumulated in a ray segment. In general the color and the opacity mapping of function values cannot be defined by an analytic function, so it is difficult, sometimes impossible, to derive a closed form solution for computing the intensity values. For a linear element, a good approximation, according to the Gaussian Quadrature [20], is to compute the intensity value at the middle of the segment and to use that value for the segment. Then two points are sampled for a quadratic element and three points are sampled for a cubic element. The Gaussian Quadrature integration approximates a polynomial of degree $2n - 1$ by using n points. In this way, the order of precision is $(2n - 1)$ which is acceptable for our purpose.

In practice, because the color and opacity transfer functions used are usually not polynomials, even for a linear element, it might be necessary to sample at multiple points along the ray segment within the element. That is, sampling along a ray should be selected according to not only the data resolution and the variation of data values, but also the variation of the transfer function values.

To implement adaptive sampling, considering an element with a linear interpolation function $f(x, y, z)$, if a ray $P(t)$ enters the element at $t = a$ and exits at $t = b$, the sample rate can be defined as

$$r = \frac{k(f(P(b)) - f(P(a)))}{b - a}$$

where $P(t) = P(0) + t\vec{d}$, $P(0)$ is the starting point of the ray, \vec{d} is the direction of the ray, and k is a constant which is the sampling rate for a unit change of function value. However, in current implementation, a much simpler approach is applied by precomputing a dt value for sampling along a ray at a fixed rate. A good principle for sampling linear elements is that, besides the entering and leaving points a and b , at least one additional point c is sampled for each element. Then between each pair of sample points, a reasonable approximation is to use the Trapezoid rule [20] to calculate the value of the corresponding interval.

To estimate dt , for example, one can use

$$dt = 0.5 \times \min\left(\frac{l_x}{\sqrt[3]{n_e}}, \frac{l_y}{\sqrt[3]{n_e}}, \frac{l_z}{\sqrt[3]{n_e}}\right)$$

where l_x, l_y , and l_z are the size of the bounding box (in x, y and z direction, respectively) containing the domain, and n_e is the total number of elements in the domain. Using such estimation, more samples would be collected than needed within a large element of constant value. The

advantages of using a fix sampling rate is mainly simpler implementation. The above formula is applied to data sets of significantly different mesh characteristics and found that it generates reasonably good sampling interval values.

At each sample point on the ray, the interpolated function value is used to obtain a color and an opacity from a look-up table. The intensity value at that point is then computed using these color and opacity values. The final image value corresponding to each ray is formed by compositing, front-to-back, the intensity values of the sample points along the ray. The compositing operation is *associative* [16], which allows us to break a ray up into segments, process the sampling and compositing of each segment independently, and combine the results from each segment via a final compositing step. This is the basis for our parallel volume rendering algorithm.

3.4 Global Image Compositing

In parallel rendering of regular data, image compositing order can always be determined *a priori*. Many efficient parallel image compositing algorithms for the rendering of regular data have been proposed [1, 13, 15]. However, as indicated previously, an unstructured domain tends to be irregular in shape and may contain holes and concavities. The situation could be more severe for subdomains after data partitioning. Consequently, the ray segments which contribute to an individual pixel cannot be combined until they are all received and properly sorted in visibility order by the responsible processor.

There are essentially two approaches for delivering ray segments to the responsible processors. While the first approach is to separate the local rendering completely from the global compositing, the second one is to overlap them. That is, in the first approach, the delivery and compositing of ray segments does not occur until the local ray-tracing process is completely finished. At the end of the rendering, ray segments are then divided into groups and each group is sent to the responsible processor. In the second approach, a ray segment can be sent immediately after it is generated. Therefore, the first approach would result in large messages being sent all about the same time, likely to induce network congestion [3].

On the other hand, with the second approach, smaller messages are sent throughout the entire course of rendering. To reduce the number of messages, groups of ray segments are sent immediately following the rendering of one locally visible face. Ray segments are divided into groups according to their corresponding destinations. Then, ray segments received at each processor are sorted into a local ray buffer by destination pixel. Sorted ray segments are composited in order at the end of rendering. The final subimage generated by each processor is sent back to the host computer (e.g. the service node in an Intel Paragon). According to our tests and others [15, 19], overlapping the ray casting and compositing can improve the overall image compositing performance by as much as 40%. As shown later, compared to the ray-tracing cost, the image compositing cost is relatively low even with the kind of preprocessing that has been done. So the the 40% improvement is less visible in overall.

There are different ways of partitioning image space [15], such as strips, blocks and interleaved

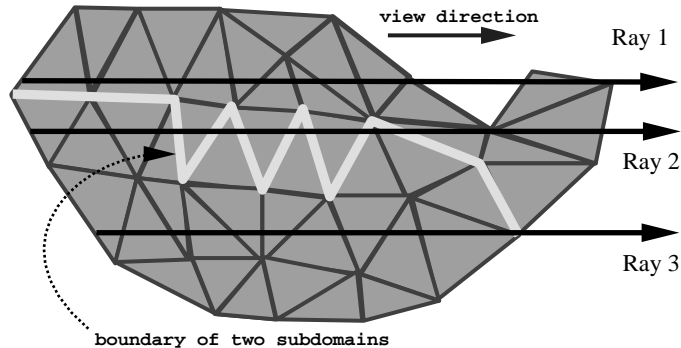


Figure 4: Both the shape of the data and the data partitioning affect a ray buffer.

ones. In our current implementation, for simplicity, the image space is partitioned evenly into horizontal strips; in fact, this partitioning scheme makes the packing of scanlines straightforward. More elaborate image partitioning should be considered later when attaining interactive rendering rates.

3.4.1 Ray Buffer

The efficiency of the compositing process is affected by the number of ray segments stored in the ray buffer. A ray buffer is a two dimensional array of linked lists. For performing distributed image compositing, each processor is assigned a portion of the final image. A corresponding local ray buffer is created for storing incoming ray segments. Each incoming ray segment is sorted into the linked list pointed to by the corresponding pixel address. Prior to the start of rendering, each processor allocates an empty ray buffer according to the image decomposition scheme. During the course of the rendering, the size of the ray buffer grows. The final size depends on the viewing position, the shape of the data volume and how the data partitioning has been done. An ideal data partitioning would produce compact subvolumes that are simple in shape, resulting in ray buffers of minimum size. As shown in Figure 4, tracing of Ray 2 would produce many local ray segments which would then result in a much longer linked list than Ray 1 and 3 because of the view direction as well as the partitioning. Specifically, Ray 1 results in two segments, Ray 2 four, and Ray 3 only one.

4 Test Results

This rendering algorithm is tested on an Intel Paragon XP/S by using an artificial data set and a flow data set from a computational fluid dynamics simulation. Message passing is implemented by using the native communication library, NX. The renderer was originally implemented in a mix of C and C++; therefore, the whole program was compiled with g++ instead of the native C compiler. Timing results reported in this paper are based on a C version compiled with the

native compiler. Comparing with the test results reported in [11] shows that the use of the g++ compiler causes about 40-50% performance degradation in overall rendering time.

A host program runs on a service node of the Paragon for delivering data and collecting results. This setting is not ideal but it allows us to focus on the rendering part of the visualization process and study some interesting behaviors. This study ignores I/O problems. All timing results are given in seconds and are calculated by averaging the times obtained on each node from rendering an animation sequence of ten frames covering various viewing directions, and then selecting the maximum averaged value among all participating nodes.

4.1 Volume Data Sets

The artificial volume data set has some well understood properties in both data values and mesh topology. The data domain is composed of tetrahedra of *identical size*. The overall domain is cubical in shape and the layout of elements is symmetrical. Let's call it the cube data set. The highest intensity value is assigned to the center of the domain with decreasing values toward the boundaries of the domain. The overall volume contains 150 thousand tetrahedra. In Color Plate 1, the right image shows a volume-rendered image of such a data set. Unlike exterior-face rendering, direct volume rendering allows us to see through the volume and visualize properties inside the volume by assigning low opacity to low intensity data values. The left image of Color Plate 1 shows exterior-face rendering of a particular subvolume. Grid lines on the subvolume boundary help show the grid structure and the resolution of the data.

The flow data set is typical in numerical modelings that use unstructured meshes. It is the result of simulating flow over a vehicle forebody at a Mach number of 8.15 and an angle of attack of 30 degrees. The flow field contains a detached bow shock following the shape of the forebody. There are about 45.5 thousand tetrahedra. The left image of Plate 2 shows an exterior-face rendering of this volume data. The grid lines on the exterior faces are also plotted to show the highly adaptive grid structure. Since only the region surrounding the vehicle body is interesting, on the right side of Color Plate 2, a cropped image shows a volume rendering of that region. Colors are mapped to the density field such that red and yellow highlight higher density regions.

These two data sets are considered small. In practice, an unstructured data set from computational fluid dynamics applications can contain several millions of elements. Smaller data sets are used for testing because they are more convenient and do not prevent us from revealing the performance of the rendering algorithms.

4.2 Data Partitioning

Data partitioning has been done using **Chaco** [8], a software package developed by Bruce Hendrickson and Robert Leland at Sandia National Laboratories to partition graphs. **Chaco** provides several different methods as there is no single method that is good for all types of data. For the

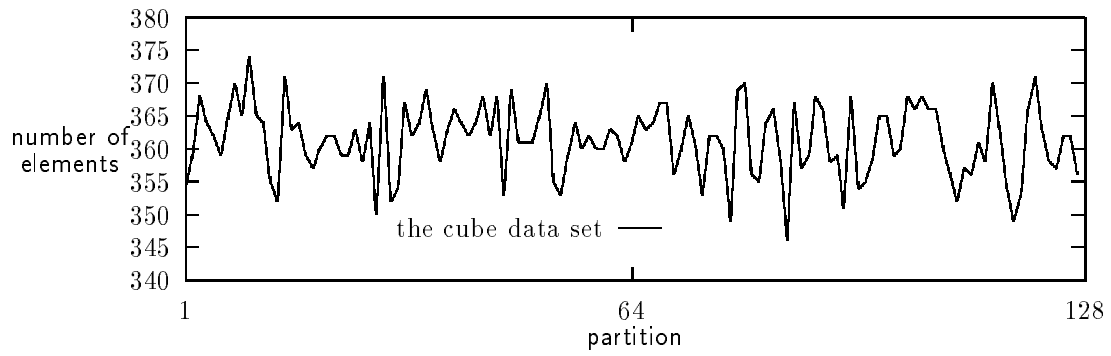
cube data set, since all tetrahedra are identical in size, the partitioning results in subvolumes of about the same size not only in terms of the number of elements but also in terms of the region of space each occupies. A typical subvolume is shown in the right image of Color Plate 1 as the result of partitioning for eight processors.

Figure 5 (a) plots the number of elements in each partition for the 128-subvolume case. It appears that there is a great variation in size but, in fact, the partitioning generates a standard deviation of only about 5, which represents 1.5% of the average number of elements. So the partitioning is considered even in terms of the number of elements. Similarly, we can calculate and compare the volume of each partition (e.g. its physical size). Figure 5 (b) displays the ratio of the partition volume size and the total data volume size for each partition. From this plotting, we can see the partitioning is also quite even in terms of physical size. In this regard, one would expect each processing node to take about the same time for rendering.

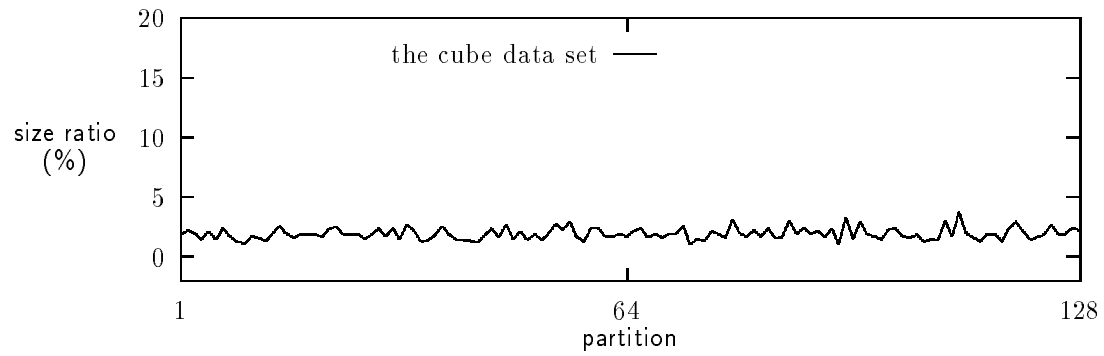
As shown in the next section, the local rendering time totally dominates the overall rendering process. Figure 5 (c) plots only the local rendering time taken by each processor for the cube data set. The image size is 512×512 pixels. The curve plotted is not as smooth as the curve for the partitions as shown in (b). The difference between the maximum rendering time and the minimum is about 30%. This difference is partly due to the early ray-termination scheme implemented. That is, a ray is terminated when the accumulated opacity value reaches unity. For the cube data, higher density values are mapped to higher opacity values which would cause an immediate termination of a ray. If a subvolume has a larger projected area of high density region, many rays would terminate earlier and thus traverse through fewer elements. Another factor is the orientation of the cut-boundary of a subvolume, which influences the ray casting cost.

On the other hand, for the flow data set, because the mesh used for the calculations was generated in an adaptive manner, in some area, a large number of small elements occupy a relatively small region of space in the overall domain. Consequently, after partitioning, although the subvolumes are about the same size in terms of the number of elements, their physical sizes are very different. The partitioning results in a standard deviation of about 4.8, which represents 4% of the average number of elements. Since the total number of elements for the flow data set is only 45.5 thousands, the partitions are not as even as the cube data's; the difference is still small though.

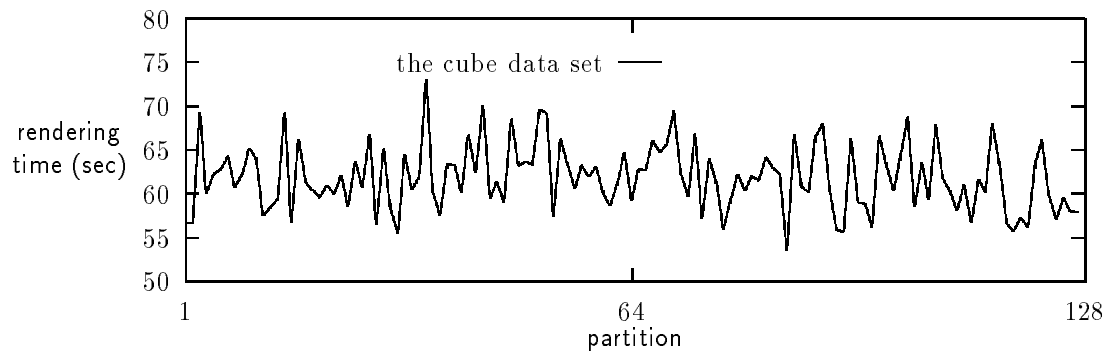
If we compare the partitioning results in terms of physical size ratio, the difference is then much bigger. Figure 5 (d) shows the size ratio of each partition for the flow data. In this particular partitioning, the largest partition is more than 400 times the volume size of the smallest one. One would expect the smaller volume in space to project onto a small area of the screen and that the corresponding rendering would take far less time than the larger volume. Based on our test results, the difference between the maximum and the minimum time is about 40%, not as high as expected. Although fewer rays are cast for the smaller projected area, the number of elements a ray must traverse is high, compared to a subvolume with a large projected area but



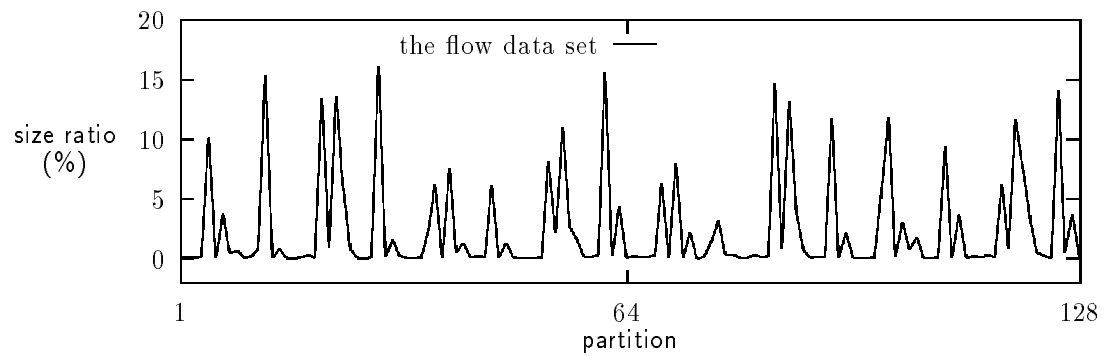
(a)



(b)



(c)



(d)

Figure 5: Relationships between Partitioning and Rendering Time for the 128-partition cases.

small number of elements.

4.3 Performance Study

For the following study, a simplified performance model is constructed. Since ray-casting time is so dominant, as shown later, other costs, like communication, are all ignored in the model. Ideally, doubling the number of processors would cut the cost of ray-casting into half; that is,

$$T(2p) = \frac{T(p)}{2} \tag{1}$$

where p is the number of processors used. In practice, this is not the case. As already mentioned, the two factors affecting the rendering performance are the added ray-casting overhead and the imbalanced load. To include these two factors, the following equation is used:

$$T(2p) = \frac{T(p)}{2} + T_o + T_l$$

where T_o is the overhead cost and T_l counts the imbalanced load; specifically,

$$T_o = k_o \times \frac{T(p)}{2}, \text{ and}$$

$$T_l = k_l \times \frac{T(p)}{2}.$$

where k_o and k_l are experimentally determined. The proportion of the total rendering time due to the imbalanced load is calculated as:

$$1 - \frac{t_{avg}}{t_{max}} \tag{2}$$

where t_{avg} is the average rendering time and t_{max} is the maximum rendering time. Note that this formulation to characterize load imbalance does not reveal the distribution of imbalanced load.

Table 1 shows timing results for rendering the cube data onto an image of 256×256 pixels, using from two to 128 processors. In all the Tables, the item names are abbreviated as follows:

- view indep prep - view independent data preprocessing time
- view dep prep - view dependent data preprocessing time
- ray casting - ray casting time
- ray-seg deliver - ray segment delivery time
- ray-seg sort - ray segment sorting time
- ray-seg merge - ray segment compositing time
- load imb - load imbalance

The load imbalance numbers are calculated by using Equation 2. Compared to the ray-casting time, the image compositing time (ray segment delivery + sorting + compositing) is negligible. While using more processing nodes increases the number of image compositing layers, the image

nodes	2	4	8	16	32	64	128
view indep prep	114.85	3.398	1.686	2.303	0.48	0.24	0.124
view dep prep	19.42	0.1606	0.073	0.0775	0.039	0.021	0.012
ray casting	445.6	227.9	126	103.43	59.7	33.2	19.1
ray-seg deliver	4.7	2.41	1.502	1.294	0.902	0.65	0.596
ray-seg sort	26.044	3.875	3.034	2.466	1.783	1.475	1.357
ray-seg merge	5.221	0.2975	0.223	0.1528	0.096	0.0608	0.034
load imb	6%	7%	15%	27%	26%	19%	15%

Table 1: Time Breakdown for Rendering the Cube Data, 256×256 Image Size.

number of nodes	2	4	8	16	32	64	128
view indep prep	2.26	1.118	0.557	0.281	0.142	0.073	0.042
view dep prep	0.117	0.075	0.0438	0.026	0.016	0.01	0.0071
ray casting	189.3	122.27	78.95	50.8	31.74	17.9	13.6
ray-seg deliver	2.29	1.56	0.871	0.584	0.42	0.369	0.370
ray-seg sort	2.566	2.336	1.71	1.204	1.016	0.756	0.750
ray-seg merge	0.219	0.20	0.118	0.079	0.049	0.030	0.018
load imb	1.1%	12.9%	21%	28%	31%	28%	41%

Table 2: Time Breakdown for Rendering the Flow Data, 256×256 Image Size.

area that each processor must handle decreases; therefore, the image compositing cost decreases as more processors are used. The time for the view-dependent data preprocessing is also moderate and decreases accordingly. This is important since the overhead due to view changes should be kept to a minimum to cope with frequent view changes to support interactive data exploration.

Table 2 show timing results for rendering the flow data onto a 256×256 -pixel image, using from two to 128 processors. A similar trend can be identified in these timing results. Again, the ray-casting time completely dominates the overall rendering. In Figure 6, parallel efficiency numbers are plotted using the timing results shown in Table 1 for the cube data set and Table 2 for the flow data set. Only the ray tracing time is used to calculate each efficiency number. Note that the parallel efficiency number is relative to the 2-processor running time. A logarithmic scale is used for the x axis to spread out the data points. Poor scalability is easily identified in this plotting for both cases. In particular, the efficiency of rendering the flow data is much lower. This basically shows that the an adaptive mesh does result in more imbalanced load and thus lower parallel efficiency.

Finally, in Table 3, timing results from rendering the cube data set using 128 processors are presented for four different image sizes. Since only the image compositing process requires node-to-node communication, these numbers help determine how the overall image compositing cost would change in proportion to an increase in the image size. As the numbers indicate, the image compositing cost grows more slowly than the ray casting cost. If more processors or more powerful

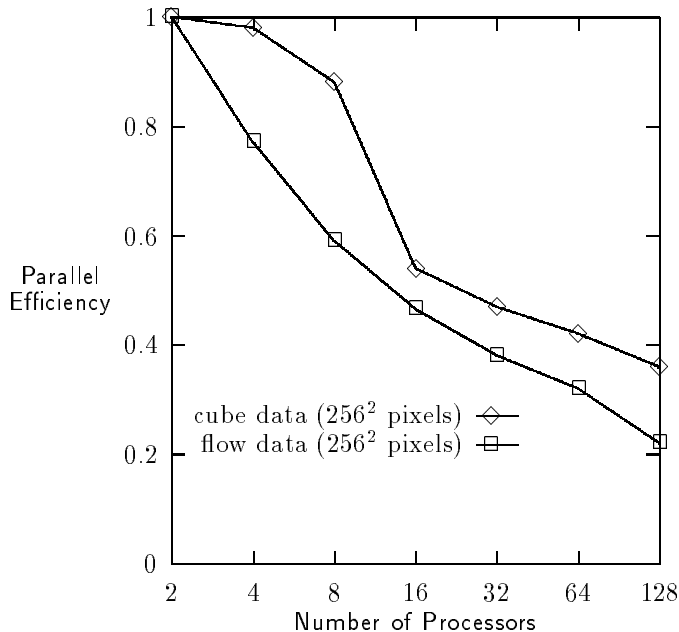


Figure 6: Rendering Parallel Efficiency.

processors are used, and the ray-casting code is further optimized, the image compositing time would become influential on the overall performance.

The added ray-casting overhead can be estimated by using a fixed number of processors to render a set of different partitions of the same data set. Some test results are shown in Section 4.4. Based on the timing results shown there, the added ray-casting overhead is about 15% of the original cost (i.e. $k_o = 0.15$). According to Equation 2, k_l should be a half of the average “load imb” number calculated; thus, the average proportion of time due to imbalanced load is about 15% (i.e. $k_l = 0.07$) for the uniform-mesh case, and 24% for the adaptive-mesh case (i.e. $k_l = 0.12$). Using Equation 2 with the above k_o and k_l values, predicated performance can be computed and compared with the test results. Figure 7 shows that the comparisons for both the

image size (pixels)	128 ²	256 ²	512 ²	768 ²
view indep prep	0.123	0.124	0.113	1.113
view dep prep	0.0135	0.012	0.012	0.014
ray casting	5.796	19.184	75.09	161.6
ray-seg deliver	0.441	0.596	0.84	0.861
ray-seg sort	1.17	1.357	2.535	3.2
ray-seg merge	0.011	0.034	0.115	0.238
load imb	25%	15%	17%	15.2%

Table 3: Time Breakdown for Rendering the Cube Data, Different Image Sizes, 128 Processors.

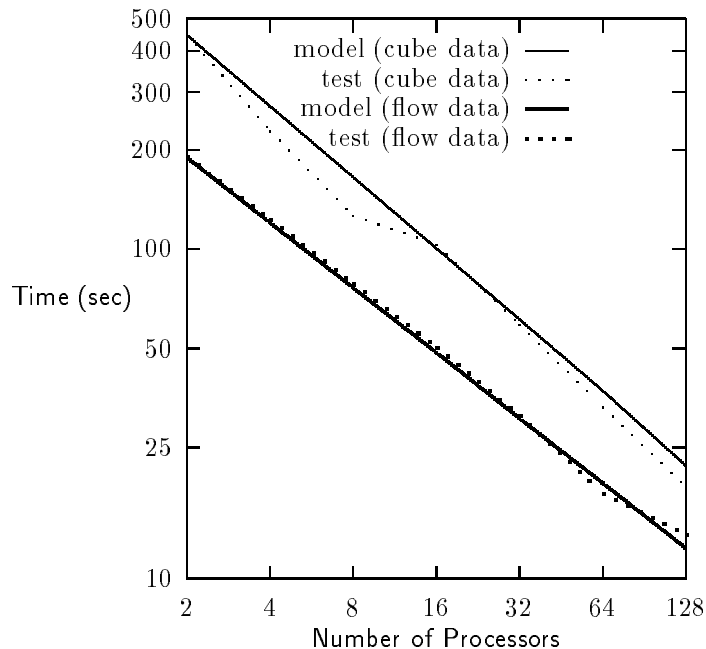


Figure 7: Matching Predicted Times with Run Times.

cube data set and the flow data set. In general, the predicted results match the actual numbers quite well so the model is acceptable.

The current implementation of the algorithm on the Intel Paragon is by no means ideal. By further taking advantage of the coherence in the data and the resampling process, in particular, during the calculations of ray-face intersection, the rendering time can be reduced. Recently, a sequential version of the renderer has been optimized and test results show 50% improvement in the overall rendering time. Since the goal of this work is to study the feasibility of the distributed approach and load balance issues, further code optimization will be left as future work. Even the same improvement can be done for the parallel version of the code, real-time rendering rates are still difficult to attain for large data sets, those containing several millions of elements.

4.4 Load Imbalance

The ray-casting overhead can be reduced to a certain extent, but not totally, by further optimizing the code. Nevertheless, the distributed approach proposed here is useful only if the the average 20% time due to the imbalanced load can also be reduced as much as possible. Figure 8 plots the proportion of the total local-rendering time due to load imbalance for both the cube data and flow data, as listed in Table 1 and 3. The trend shows the load imbalance increases with the number of partitions for data on a highly-adaptive mesh. A further test using a zoom-in view shows the load imbalance became even worse. The proportion of time devoted to ray casting due to the imbalance load then became about 33%.

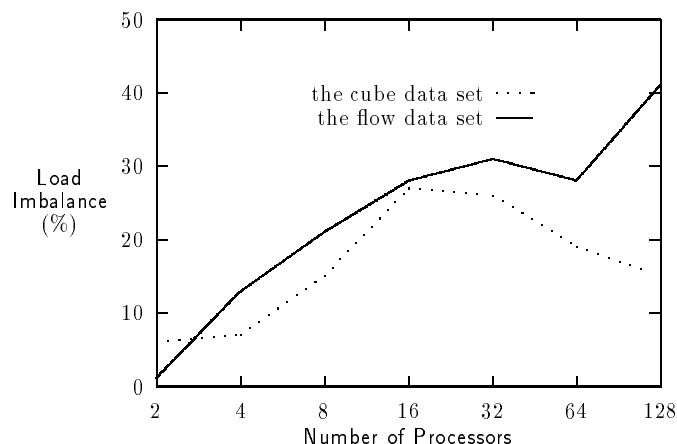


Figure 8: Load Imbalance in Rendering the Flow Data.

In summary, there are five factors which contribute to the load imbalance shown in our timing results:

- subvolume size: number of elements
- subvolume size: physical size (like the projected area)
- opacity transfer functions
- viewing angle
- zoom in/out

Considering these multiple factors, we conjecture that it is difficult to achieve load balancing statically. A particular static scheme is implemented and tested to verify our conjecture. The scheme works as follows. For p processors, a data volume is partitioned into $n \cdot p$ subvolumes where n is a magic number which can be determined after one or two test runs. Then the $n \cdot p$ subvolumes are distributed in a round-robin fashion among processors. So now each processor rendered possibly many disjointed subvolumes instead of a single large one. Consequently, the differences in both the average size of projected areas and in the average number of elements handled by each processor would become smaller as n increases. At the same time, many more ray segments are generated and result in higher image compositing cost. We hope that the more balanced load can cover the additional ray-casting overhead. But the timing results are not encouraging.

Figure 9 shows the proportion of the total rendering time due to load imbalance and the corresponding increasing factors in rendering time for the flow data set using 32 processors. While the load becomes slightly more balanced by using the above approach, the overall rendering time

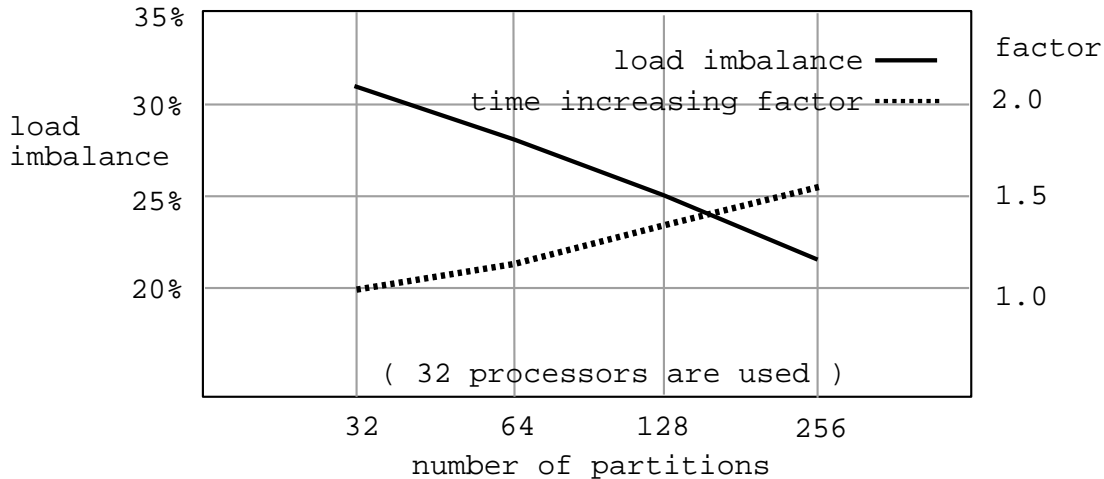


Figure 9: Static Load balancing for the Flow Data.

increases, on average, about 15%. Moreover, static load balancing can fail badly for the zoom-in problems which frequently arise from visualizing highly-adaptive unstructured grids.

The above test results suggest a dynamic approach to enhance the distributed rendering algorithm. The development of dynamic load balancing methods for rendering unstructured-grid data will be our major emphasis of future research. There are generally two dynamic load balancing approaches: by pool of tasks and by coordination. The first approach is typically used with a master/slave scheme. The master keeps a queue of tasks and continues to give them to idle slaves until the queue is empty. Faster machines end up performing more tasks. This master/slave scheme is very suitable for a visualization process involving frequent zoom-in and isolated view selections. The performance is mainly determined by how efficiently the data corresponding to the tasks assigned can be delivered to a processor. Dynamic load balancing by coordination is typically used in a SPMD scheme. That is, all the tasks stop and redistribute their work either at fixed times or if some condition occurs, such as the load imbalance between tasks exceeds some limit. This approach seems suitable for general visualization purposes but its implementation would be much more complicated than the master/slave approach, and its performance could be influenced by more factors.

5 Conclusions

Direct volume rendering is a powerful visualization technique for many scientific and engineering applications. However, for large unstructured data, volume rendering is often too expensive to run on a single workstation. This paper presents a data-distributed rendering algorithm for parallel volume ray-casting on unstructured meshes. The algorithm makes possible rendering of large data sets that cannot fit into the main memory of a single workstation.

Based on the timing results on the Intel Paragon, the computational cost for postprocessing

a data set with millions of elements would be tremendous, even using a massively parallel computer. Therefore, so far, the kind of postprocessing analyses that computational researchers can do using three-dimensional computer graphics techniques have been very limited. Our performance studies also indicate many opportunities for further optimization of the algorithm and its implementation. In particular, ray-casting overhead and imbalanced load significantly affect the overall performance of the renderer. Dynamic load balancing, particularly by the pool of tasks approach, should be implemented to reduce the typical load imbalance which is above 20%, and at the same time to allow more efficient data exploration spatially.

As approaching interactive rendering rates using more processors or more powerful processors, it is then needed to reevaluate both the image-space partitioning and image compositing step as the rendering step become less dominant. Data and image I/O, a frequently ignored problem, must be improved to make the overall rendering process more efficient. One important use of such parallel rendering capability is to support runtime monitoring of numerical simulations running on a parallel computer. Therefore, future work will also focus on supporting runtime visualization, along with the development of a rendering library.

Acknowledgments

The author would like to thank Tom Crockett and Jamie Painter, and the anonymous reviewers who provided many useful suggestions on ways to improve the manuscript. This research was supported in part by the National Aeronautics and Space Administration under NASA contract NAS1-19480 while the author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE).

References

- [1] CAMAHORT, E., AND CHAKRAVARTY, I. Integrating Volume Data Analysis and Rendering on Distributed Memory Architectures. In *Proceedings of Parallel Rendering Symposium* (1993), pp. 89–96. San Jose, October 25-26.
- [2] CHALLINGER, J. Scalable Parallel Volume Raycasting for Nonrectilinear Computational Grids. In *Proceedings of Parallel Rendering Symposium* (1993), pp. 81–88. San Jose, October 25-26.
- [3] CROCKETT, T., AND ORLOFF, T. Parallel Polygon Rendering for Message Passing Architectures. *IEEE Parallel and Distributed Technology* 2, 2 (1994), 17–28.
- [4] GALLAGHER, R., AND NAGTEGAAL, J. An Efficient 3-D Visualization Technique for Finite Element Models and Other Coarse Volume. *Proceedings of SIGGRAPH '89 (Boston, July 31-August 4, 1989)*. *Computer Graphics* 23, 3 (August 1989), 185–193.

- [5] GARRITY, M. P. Raytracing Irregular Volume Data. *Proceedings of 1990 Workshop on Volume Visualization (San Diego, December 10-11, 1990)*. Special issue of *Computer Graphics, ACM SIGGRAPH 24*, 5 (November 1990), 35–40.
- [6] GIERTSEN, C. Volume Visualization of Sparse Irregular Meshes. *IEEE Computer Graphics & Applications 12*, 2 (March 1992), 40–48.
- [7] GIERTSEN, C., AND PETERSEN, J. Parallel Volume Rendering on a Network of Workstations. *IEEE CG&A 13*, 6 (November 1993), 16–23.
- [8] HENDRICKSON, B., AND LELAND, R. The Chaco User's Guide (Version 1.0). Tech. Rep. SAND93-2339, Sandia National Laboratories, Albuquerque, NM, 1993.
- [9] KOYAMADA K., N. T. Volume Visualization of 3D Finite Element Method Results. *IBM J. Res. Develop. 35* (March 1991), 12–25.
- [10] LORENSEN, W. E., AND CLINE, H. E. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *Proceedings of SIGGRAPH '87 (Anaheim, July 27-31, 1987)*. In *Computer Graphics 21*, 4 (July 1987), 163–169.
- [11] MA, K.-L. Parallel Volume Ray-Casting for Unstructured-Grid Data on Distributed-Memory Architectures. In *Proceedings of the Parallel Rendering '95 Symposium (1995)*, pp. 23–30. Atlanta, Georgia, October 30-31.
- [12] MA, K.-L. Runtime Volume Visualization for Parallel CFD. In *Proceedings of Parallel CFD '95 Conference (1995)*. California Institute of Technology, Pasadena, CA, June 25-28.
- [13] MA, K.-L., PAINTER, J. S., HANSEN, C., AND KROGH, M. Parallel Volume Rendering Using Binary-Swap Compositing. *IEEE Computer Graphics Applications 14*, 4 (July 1994), 59–67.
- [14] MAX, N., HANRAHAN, P., AND CRAWFIS, R. Area and Volume Coherence for Efficient Visualization of 3D Scalar Functions. *Computer Graphics 24*, 5 (November 1990), 27–33.
- [15] NEUMANN, U. Parallel Volume-Rendering Algorithm Performance on Mesh-Connected Multicomputers. In *Proceedings of Parallel Rendering Symposium (1993)*, pp. 97–104. San Jose, October 25-26.
- [16] PORTER, T., AND DUFF, T. Compositing Digital Images. *Proceedings of SIGGRAPH '84*. *Computer Graphics 18*, 3 (July 1984), 253–259.
- [17] SHEN, H.-W., AND JOHNSON, C. R. Sweeping Simplices: A fast iso-surface extraction algorithm for unstructured grids. In *Proceedings Visualization '95 Conference (1995)*, G. M. Nielson and D. Silver, Eds., pp. 143–150.

- [18] SHIRLEY, P., AND TUCHMAN, A. A Polygon Approximation to Direct Scalar Volume Rendering. *Proceedings of 1990 Workshop on volume Visualization (San Diego, December 10-11, 1990)*. *Computer Graphics* 24, 5 (November 1990), 63–70.
- [19] SILVA, C., AND KAUFMAN, A. Parallel Performance Measures for Volume Ray Casting. In *Proceedings of Visualization '94 Conference* (1994), pp. 196–204.
- [20] STOER, J., AND BULIRSCH, R. *Introduction to Numerical Analysis*. Springer-Verlag, 1993.
- [21] USELTON, S. Volume Rendering on Curvilinear Grids for CFD. AIAA Paper 94-0322, 1994. 32nd Aerospace Sciences Meeting & Exhibit.
- [22] WILLIAMS, P. L. Interactive Splatting of Nonrectilinear Volumes. In *Proceeding of Visualization '92* (October 1992), A. E. Kaufman and G. M. Nielson, Eds., pp. 37–44.
- [23] WILLIAMS, P. L. Visibility Ordering Meshed Polyhedra. *ACM Trans. on Graphics* 11, 2 (1992), 103–126.
- [24] WILLIAMS, P. L. Parallel Volume Rendering Finite Element Data. In *Proceedings Computer Graphics International '93* (1993). Lausanne, Switzerland, June.
- [25] YARMARKOVICH, A., AND GELBERG, L. Visualization Techniques for Unstructured Data. SIGGRAPH '92 Course Notes 1, Introduction to Scientific Visualization Tools and Techniques, 1992.