NASA Contractor Report 198195

ICASE Report No. 95-57

ICASE

PARALLEL VOLUME RAY-CASTING FOR UNSTRUCTURED-GRID DATA ON DISTRIBUTED-MEMORY ARCHITECTURES

Kwan-Liu Ma

Contract No. NAS1-19480
August 1995

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, VA 23681-0001

Operated by Universities Space Research Association

Parallel Volume Ray-Casting for Unstructured-Grid Data on Distributed-Memory Architectures

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

As computing technology continues to advance, computational modeling of scientific and engineering problems produces data of increasing complexity: large in size and unstructured in shape. Volume visualization of such data is a challenging problem. This paper proposes a distributed parallel solution that makes ray-casting volume rendering of unstructured-grid data practical. Both the data and the rendering process are distributed among 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 maximum parallel efficiency. This algorithm differs from previous ones in four ways: it is completely distributed, less view-dependent, reasonably scalable, and flexible. Without using dynamic load balancing, test results on the Intel Paragon using from two to 128 processors show, on average, about 60% parallel efficiency.

†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), NASA Langley Research Center, Hampton, VA 23681-0001.

1 Introduction

Computational modeling of scientific and engineering problems with complex geometries often uses finite volume methods or finite element approximations, and thus calculations are carried out on unstructured grids. Typically, the problem domain is decomposed into small cells, called elements. Popular element types include the tetrahedron, triangular prism (pentahedron) and hexahedron. Many visualization techniques have been developed for the interrogation and analysis of unstructured-grid 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 three-dimensional graphics techniques fail to represent.

For most scientific and engineering problems, large-scale simulations can generate data with hundreds of thousands of elements or more. The absence of a simple indexing scheme for three-dimensional unstructured grids 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.

In this paper, we describe a distributed parallel volume ray-casting algorithm for visualizing unstructured-grid data. This algorithm differs from previous ones in several ways: it is completely distributed, less view-dependent, reasonably scalable, and flexible. First, 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. Second, the overhead due to view changes is kept to a minimum since a good distributed rendering algorithm must cope with frequent view changes to support truly interactive data exploration. Third, while using more processing nodes increases the number of image compositing layers, the image area that each processor must handle decreases; as a result, the algorithm is scalable. Without dynamic load balancing, we have achieved, on average, 60% parallel efficiency on the Intel Paragon using from two to 128 processors. Complete test results and some performance studies are presented at the end of the paper.

Finally, although the prototype 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. Note that while this paper focuses on the design and implementation of a parallel renderer as a postprocess, another equally important goal is to support runtime monitoring of large-scale parallel simulations, which may generate data that is too large to move elsewhere for postprocessing. This algorithm is flexible enough to support this goal as well.

2 Visualization on Unstructured Grids

Data visualization on unstructured grids requires multiple steps to obtain maximum efficiency due to the irregularity of the grids. Yarmarkovich and Gelberg [23] 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.

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 [21] develops an algorithm for determining the visibility order of elements. Max, Hanrahan and Crawfis [13] describe an accurate, but computationally expensive, analytic illumination model for use with their projection algorithm. Shirley and Tuchman [16] propose a splatting algorithm which provides a fast approximation to the projection process. Williams [20] 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 [22] developed a cell-projection volume rendering algorithm for finite element data running on a single SGI multiprocessor workstation. Uselton [19] 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. Challinger [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

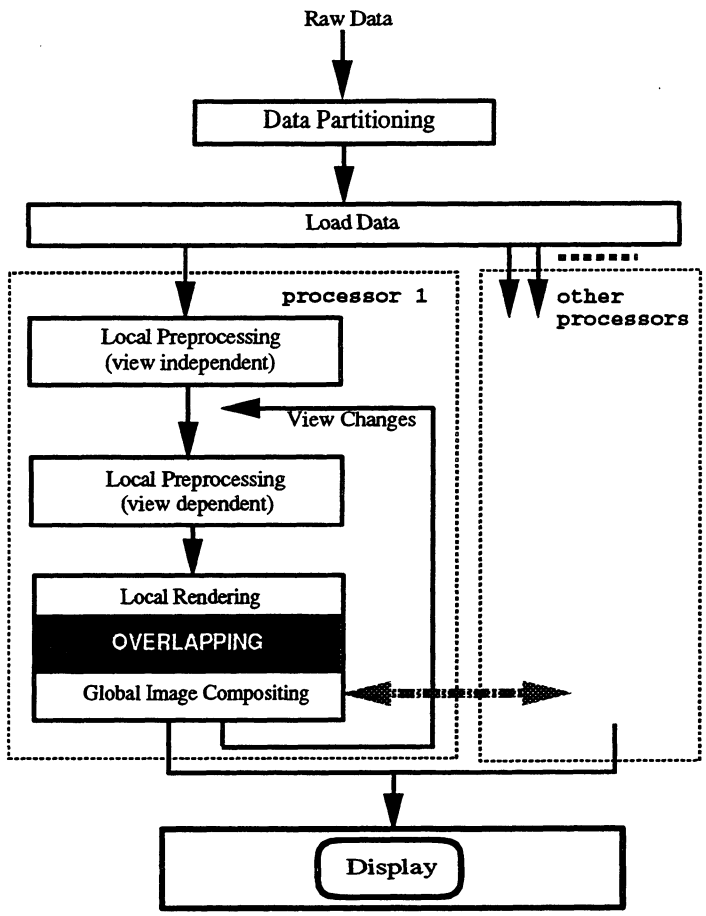

Figure 1: A Distributed Parallel Rendering Process.

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.

3 A Parallel Rendering Algorithm

We have developed a flexible and efficient distributed parallel volume ray-casting algorithm. Figure 1 depicts the parallel rendering process which consists of multiple steps. In this paper, we only consider rendering as a postprocess. 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 [11], these considerations do not change the basic algorithm and are not discussed here. Handling grids that change dynamically at runtime would be a future research topic.

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 grid calculations on distributed-memory computers. 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 and minimizes boundary area between subdomains. While the first property helps load balancing, the second reduces inter-processor communication.

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 very different from the simulation's. The criterion used by the parallel simulation may not be applicable. We have used a graph-based data-partitioning software package to obtain reasonably even subvolumes, i.e. subvolumes containing about the same number of elements.

3.2 Data Preprocessing

An unstructured-grid 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. In order to efficiently execute the visualization processes, we need to know more than the existing element-node relationship. A data structure which we call the *hierarchical data structure (hds)* is created during a preprocessing step. An *hds* consists of three layers of data structures. A node set is a data structure at the lowest layer, from which a face set is constructed. The top layer of the *hds* is an element set constructed from the face set. Therefore, an *hds* 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,

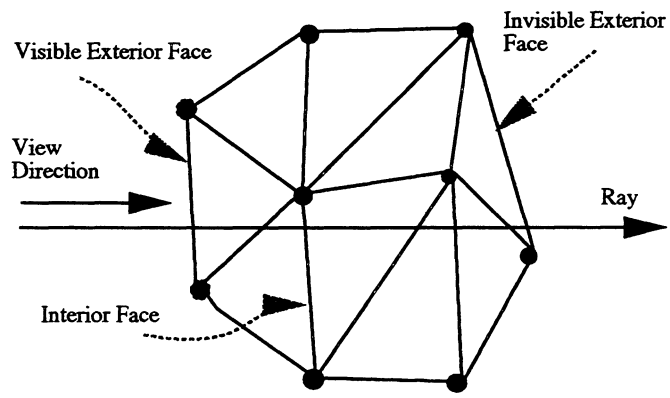

Figure 2: Face Types in a Subvolume.

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 clear that we can 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 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.

3.3 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. 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 grid is partitioned into subgrids 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,

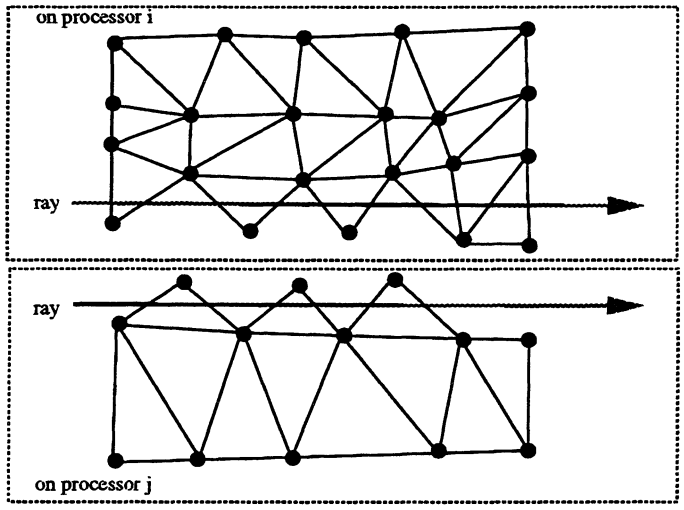

Figure 3: Rays Passing Saw-toothed Boundaries.

and $I(t)$ is the intensity at a point t along the ray. In [13], 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 [18], 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, ray-casting on unstructured grids is already a very expensive operation. Adaptive sampling makes it even more expensive and could significantly increase the overall rendering time.

Consequently, in our current implementation, we used a much simpler approach 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 [18] to calculate the value of the corresponding interval.

To estimate dt, we use

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

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. We applied the above formula to data sets of significantly different grid 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* [15], 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 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, 12, 14]. 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

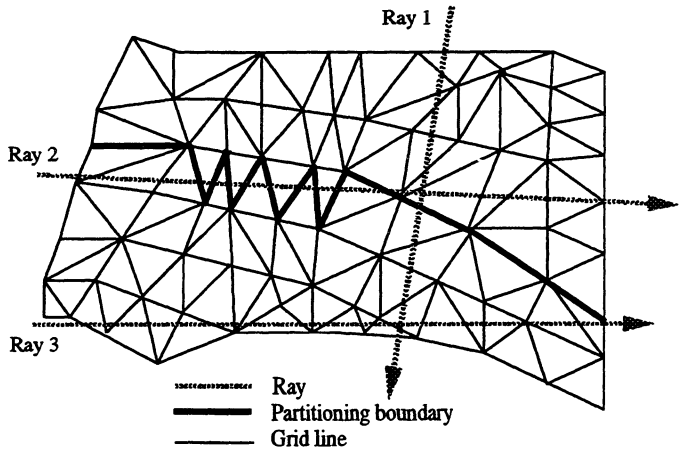

Figure 4: Both the shape of the data and the data partitioning affect a ray buffer.

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. According to our tests and others [14, 17], overlapping the ray casting and compositing can improve the overall image compositing performance by as much as 40%.

There are different ways of partitioning image space [14]. As shown later, compared to the ray-tracing cost, the image compositing cost is relatively low with the kind of preprocessing that has been done. In our current implementation, for simplicity, the image space is partitioned evenly into horizontal strips. More elaborate image partitioning will 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.

4 Test Results

We have tested an implementation of the rendering algorithm on the Intel Paragon XP/S by using an artificial data set and a flow data set from a computational fluid dynamics simulation. Message passing was implemented by using the native communication library, *NX*. A host program runs on a service node of the Paragon for delivering data and collecting results. In this study, we ignore I/O problems and focus on the rendering algorithm. 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.

The artificial volume data set has some well understood properties in both data values and grid 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. Thus we 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. We plot grid lines on the subvolume boundary to show the grid structure and the resolution of the data.

The flow data set is typical in numerical modelings that use unstructured grids. 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 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. More visualization results are displayed in Color Plate 3 and 4. Plate 3 shows direct volume visualization of the pressure field and Plate 4 shows Mach number.

Data partitioning has been mainly done using **Chaco** [8], a software package developed by Brue 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. In this regard, one would expect each processing node to take about the same time to finish rendering. We compare the processor taking the maximum rendering time with the one taking minimum time. The difference between the two can be as high as about 40%. This difference is partly due to the early ray-termination

nodes	2	4	8	16	32	64	128
view indep prep	110.8	5.146	2.657	1.305	0.731	0.588	0.172
view dep prep	60.34	23.47	13.21	6.011	3.703	2.221	1.508
ray casting	2234	1268	769.3	443.2	327.1	177.9	109.3
ray-seg deliver	4.608	2.503	1.657	1.036	0.89	0.572	0.548
ray-seg sort	62.45	19.07	3.725	2.713	2.307	1.656	1.303
ray-seg merge	53.99	9.085	1.139	0.736	0.52	0.279	0.146

Table 1: Time Breakdown for Rendering the Cube Data, 480×480 Image Size.

scheme we use. 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. If a subvolume has a larger projected area of high density region, many rays would terminate earlier and thus traverse through fewer elements.

On the other hand, for the flow data set, because the grid used for the calculations was generated in an adaptive manner, a large number of smaller 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. In this case, 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 also about 40%, much lower than we expected. Although many 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 small number of elements.

Figure 5 and Table 1 show timing results for rendering the cube data onto a 480×480 image, using from two to 128 processors. In Figure 5, only the ray tracing time is plotted because the ray tracing time is completely dominant. Note that a logarithmic scale is used for both the x and y axes so that both execution time and speedup information can be presented in one plot. 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

As we mentioned earlier, the image compositing time (ray segment delivery + sorting + compositing) is negligible and normally decreases as more processors are used. The time for the view-dependent data preprocessing is also moderate and decreases accordingly. In Table 2, timing results using 64 processors are presented for five different image sizes. Since only the image compositing process requires node-to-node communication, from these tests, we would like to

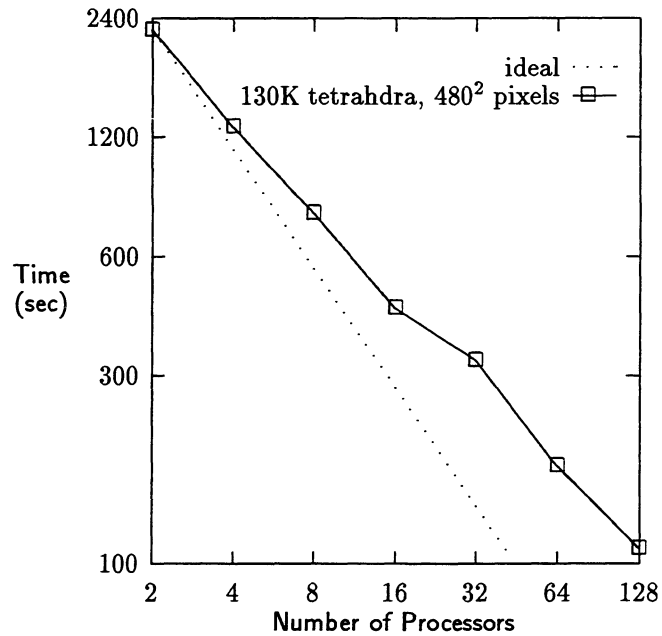

Figure 5: Ray Casting Time for the Cube Data

image size	240	360	480	720
view indep prep	0.501	0.671	0.588	1.049
view dep prep	2.587	2.82	2.221	2.995
ray casting	49.445	127.78	177.92	480.6
ray-seg deliver	0.669	0.646	0.572	0.861
ray-seg sort	1.152	1.405	1.656	2.146
ray-seg merge	0.078	0.173	0.279	0.232

Table 2: Time Breakdown for Rendering the Cube Data, Different Image Sizes, 64 Processors.

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.

Figure 6 and Table 3 show timing results for rendering the flow data onto a 480×480 image, using from two to 128 processors. Again, only the ray tracing time is plotted. We see a similar trend in these timing results. According to our timing results, the rendering cost for unstructured data is very high. Although we believe that we can tune our present code to achieve about 30% improvement in overall rendering time, near real-time rendering rates are still difficult to attain for large data sets. Even with faster processors like the IBM SP2, the best we can probably achieve is no better than one frame per second when using 256 nodes or more. Further improvement in overall rendering performance may be obtained by balancing the load dynamically.

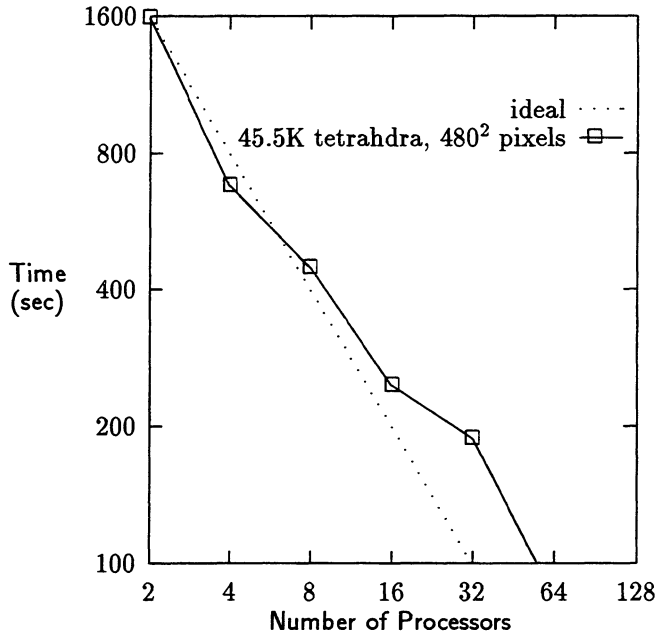

Figure 6: Ray Casting Time for the Flow Data.

number of nodes	2	4	8	16	32	64	128
view indep prep	3.35	1.659	0.910	0.42	0.23	0.11	0.062
view dep prep	1.53	0.645	2.371	0.919	0.861	0.879	0.839
ray casting	1585	677.6	445.3	245.5	187.2	86.07	63.2
ray-seg deliver	3.02	1.649	0.988	0.662	0.502	0.360	0.316
ray-seg sort	2.749	3.925	3.856	2.549	1.755	1.195	0.972
ray-seg merge	0.841	0.817	0.621	0.369	0.224	0.129	0.068

Table 3: Time Breakdown for Rendering the Flow Data, 480×480 Image Size.

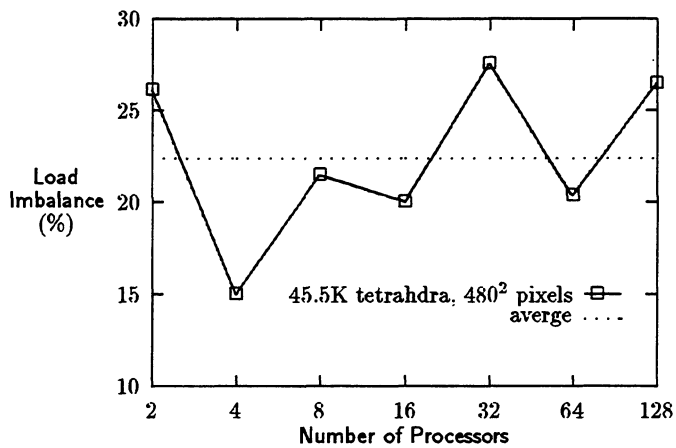

Figure 7: Load Imbalance in Rendering the Flow Data.

4.1 Load Imbalance

Our timing results show that, on average, the imbalanced loads degrade the overall performance by at least 20%. Figure 7 shows the proportion of the total rendering time for the flow data (480×480 pixels) due to load imbalance, which is calculated as:

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

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. As a further test, we rendered the flow data using a zoom-in view. Consequently, the load imbalance became worse. For the flow data and image size of 480×480 pixels, using 32 processors and focusing on views similar to the one presented in the left image of Color Plate 2, the difference between the maximum and the minimum time was raised from about 40% to 66%. The proportion of time devoted to ray casting due to load imbalance then became about 33%.

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 (projected area)
- opacity transfer functions
- viewing angle
- zoom in/out

Therefore, it will be difficult to achieve load balancing statically. In fact, we had tried a static scheme which 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 were hoping that the more balanced load can greatly reduce the ray casting cost. But the timing results did not show consistent and significant improvement. Static load balancing is not a solution for the zoom-in problem which frequently arises from visualizing highly-adaptive unstructured grids. The development of dynamic load balancing methods for rendering unstructured-grid data will be our major emphasis of future research.

5 Conclusions

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

We would like to point out that the two data sets we have used for testing are considered small. In practice, an unstructured-grid data set from computational fluid dynamics applications can contain several millions of elements. We have used smaller data sets for testing because they are more convenient and do not prevent us from revealing the performance of the rendering algorithms.

Based on our timing results, 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, imbalanced load significantly affects the overall performance of the renderer. Criteria for data partitioning should be further studied to reduce the current load imbalance which is above 20%. Dynamic load balancing, though more effective, is harder to implement with a data-distributed approach.

As we approach interactive rendering rates using more processors or more powerful processors, we may need 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, our future work will mainly focus on supporting runtime visualization, along with the development of dynamic load balancing algorithms.

Acknowledgments

The author would like to thank Tom Crockett, S.K. Ueng, Jamie Painter, and the anonymous Parallel Rendering Symposium reviewers who provided many useful suggestions on ways to improve the manuscript.

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 Archtectures. *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. Runtime Volume Visualization for Parallel CFD. In *Proceedings of Parallel CFD '95 Conference (to appear)* (1995). California Institute of Technology, Pasadena, CA, June 25-28.

[12] 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.

[13] 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.

[14] 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.

[15] PORTER, T., AND DUFF, T. Compositing Digital Images. *Proceedings of SIGGRAPH '84. Computer Graphics 18*, 3 (July 1984), 253–259.

[16] SHIRLEY, P., AND TUCHMAN, A. A Polygon Approximation to Direct Scalar Volume Rendering. *Proceedings of 1990 Workshop on volume Visulization (San Diego, December 10-11, 1990). Computer Graphics 24*, 5 (November 1990), 63–70.

[17] SILVA, C., AND KAUFMAN, A. Parallel Performance Measures for Volume Ray Casting. In *Proceedings of Visualization '94 Conference* (1994), pp. 196–204.

[18] STOER, J., AND BULIRSCH, R. *Introduction to Numerical Analysis.* Springer-Verlag, 1993.

[19] USELTON, S. Volume Rendering on Curvilinear Grids for CFD. AIAA Paper 94-0322, 1994. 32nd Aerospace Sciences Meeting & Exhibit.

[20] 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.

[21] WILLIAMS, P. L. Visibility Ordering Meshed Polyhedra. *ACM Trans. on Graphics 11*, 2 (1992), 103–126.

[22] WILLIAMS, P. L. Parallel Volume Rendering Finite Element Data. In *Proceedings Computer Graphics International '93* (1993). Lausanne, Switzerland, June.

[23] YARMARKOVICH, A., AND GELBERG, L. Visualization Techniques for Unstructured Data. SIGGRAPH '92 Course Notes 1, Introduction to Scientific Visualization Tools and Techniques, 1992.

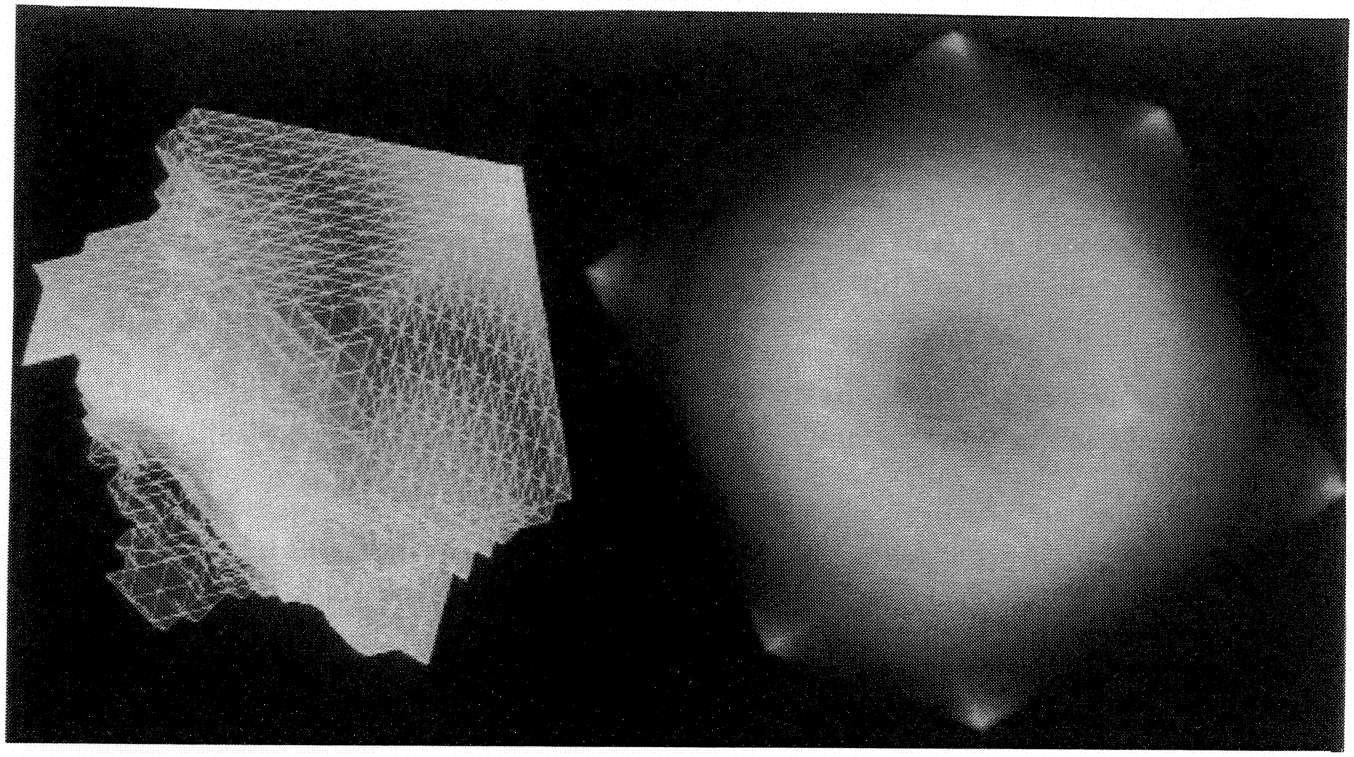

Plate 1: Visualization of the Cube Data.

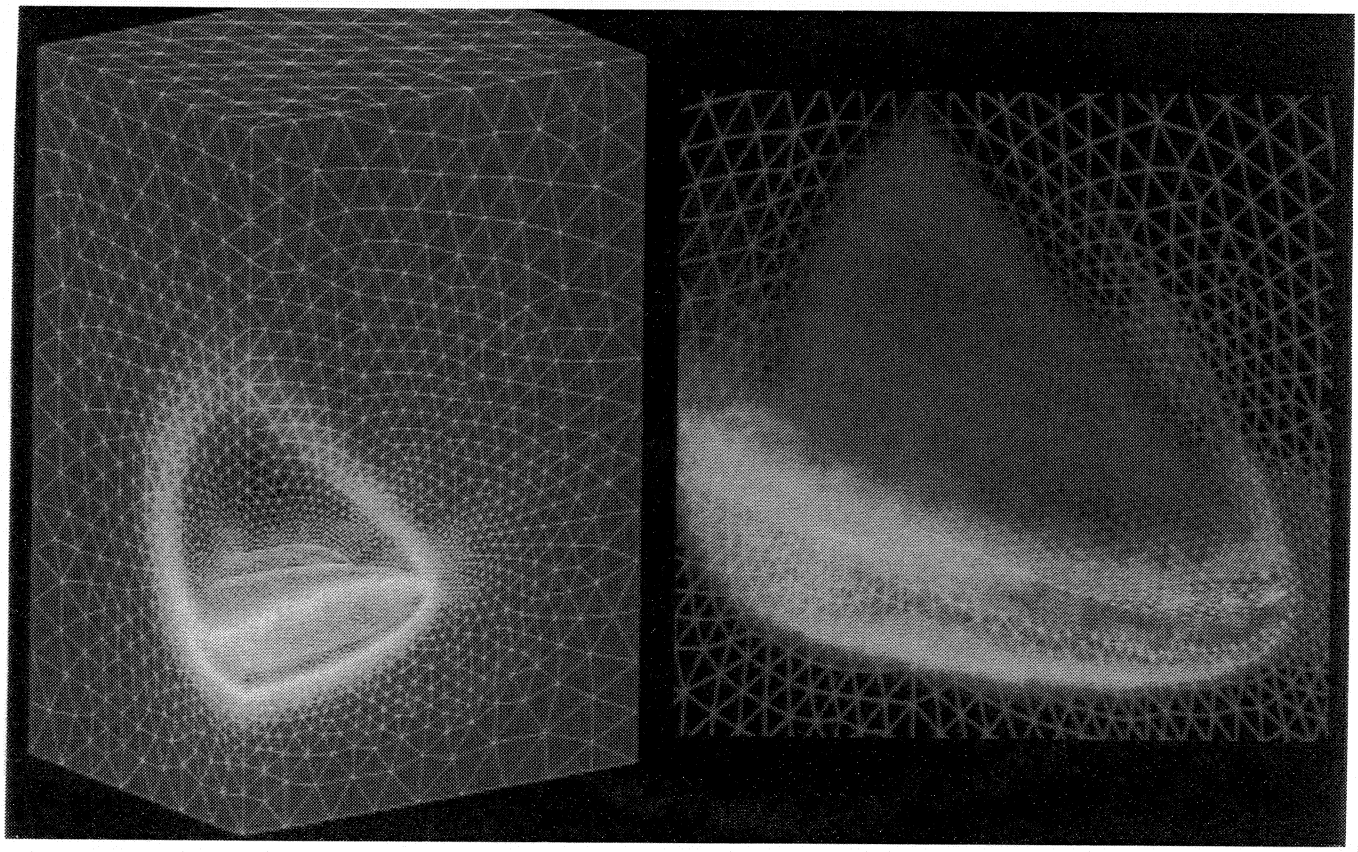

Plate 2: Visualization of the Flow Data.

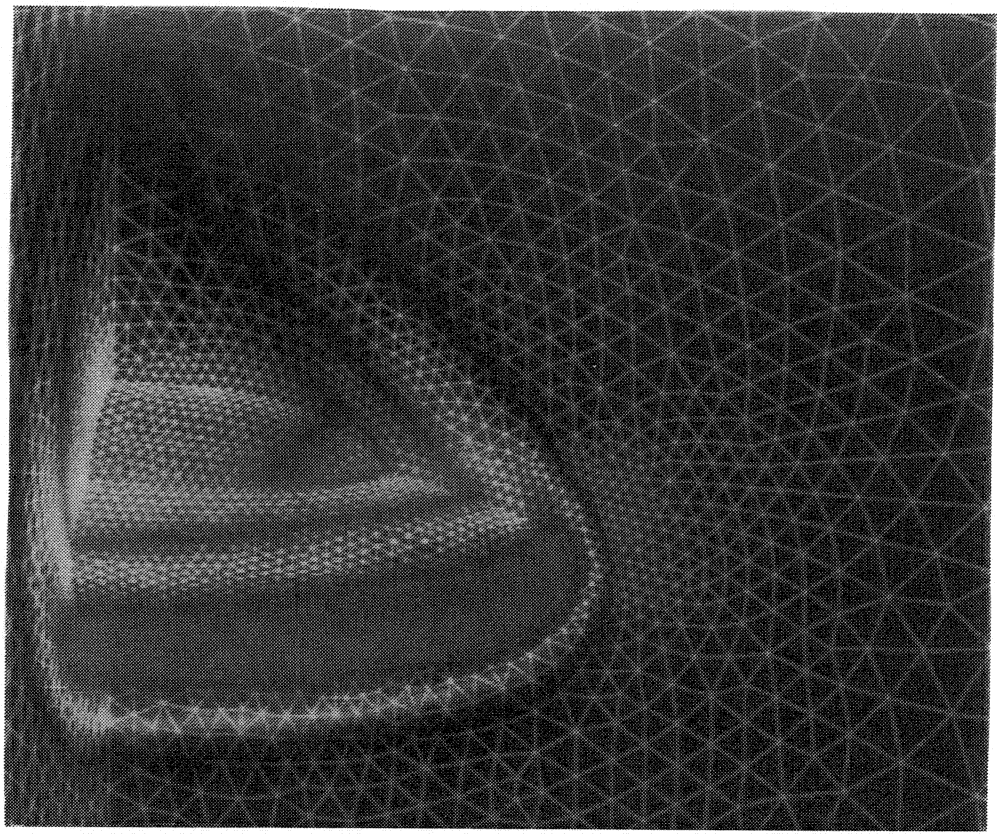

Plate 3: Volume Visualization of the Pressure Field.

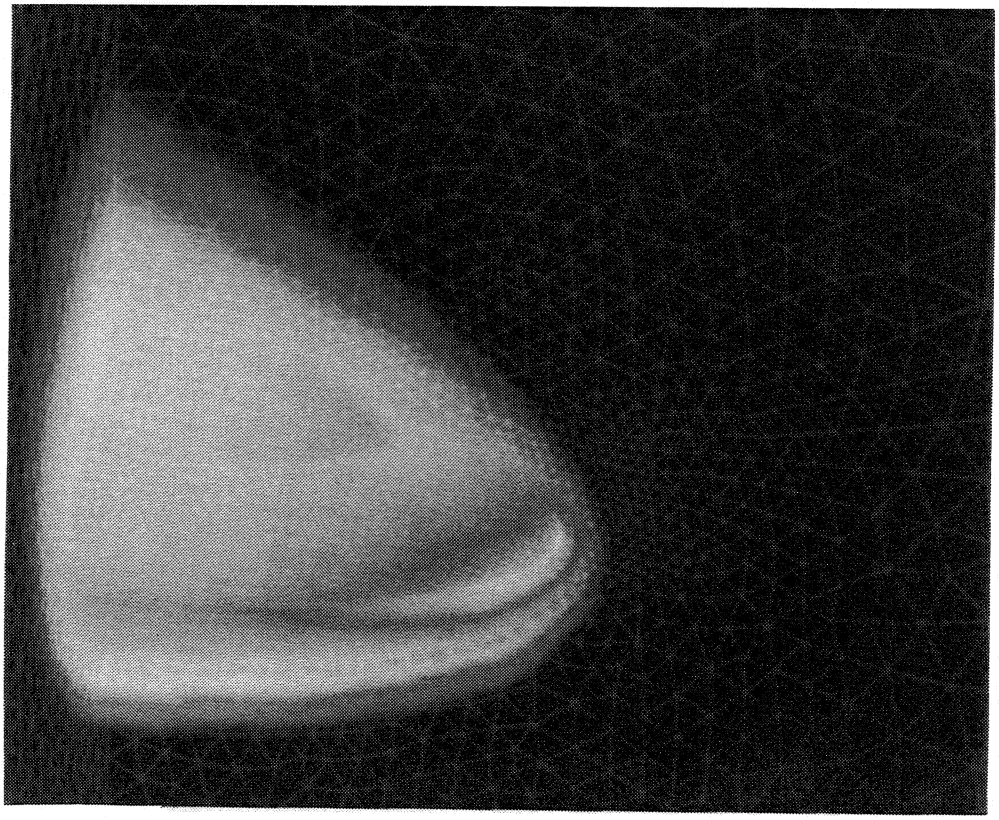

Plate 4: Volume Visualization of the Mach Number.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
August 1995

3. REPORT TYPE AND DATES COVERED
Contractor Report

4. TITLE AND SUBTITLE
PARALLEL VOLUME RAY-CASTING FOR UNSTRUCTURED-GRID DATA ON DISTRIBUTED-MEMORY ARCHITECTURES

5. FUNDING NUMBERS
C NAS1-19480
WU 505-90-52-01

6. AUTHOR(S)
Kwan-Liu Ma

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Computer Applications in Science
and Engineering
Mail Stop 132C, NASA Langley Research Center
Hampton, VA 23681-0001

8. PERFORMING ORGANIZATION REPORT NUMBER
ICASE Report No. 95-57

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA CR-198195
ICASE Report No. 95-57

11. SUPPLEMENTARY NOTES
Langley Technical Monitor: Dennis M. Bushnell
Final Report
To appear in the Proceedings of the Parallel Rendering Symposium '95

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified–Unlimited

Subject Category 60,61

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

As computing technology continues to advance, computational modeling of scientific and engineering problems produces data of increasing complexity: large in size and unstructured in shape. Volume visualization of such data is a challenging problem. This paper proposes a distributed parallel solution that makes ray-casting volume rendering of unstructured-grid data practical. Both the data and the rendering process are distributed among 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 maximum parallel efficiency. This algorithm differs from previous ones in four ways: it is completely distributed, less view-dependent, reasonably scalable, and flexible. Without using dynamic load balancing, test results on the Intel Paragon using from two to 128 processors show, on average, about 60% parallel efficiency.

14. SUBJECT TERMS
Volume Visualization; Unstructured Grids; Parallel Distributed-Memory Computers; Data Partition

15. NUMBER OF PAGES
21

16. PRICE CODE
A03

17. SECURITY CLASSIFICATION OF REPORT
Unclassified

18. SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

19. SECURITY CLASSIFICATION OF ABSTRACT

20. LIMITATION OF ABSTRACT