A Scalable Parallel Cell-Projection Volume Rendering Algorithm for Three-Dimensional Unstructured Data

Kwan-Liu Ma and Thomas W. Crockett

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, Virginia, USA

ABSTRACT

Visualizing three-dimensional unstructured data from aerodynamics calculations is challenging because the associated meshes are typically large in size and irregular in both shape and resolution. The goal of this research is to develop a fast, efficient parallel volume rendering algorithm for massively parallel distributed-memory supercomputers consisting of a large number of very powerful processors. We use cell-projection instead of ray-casting to provide maximum flexibility in the data distribution and rendering steps. Effective static load balancing is achieved with a round robin distribution of data cells among the processors. A spatial partitioning tree is used to guide the rendering, optimize the image compositing step, and reduce memory consumption. Communication cost is reduced by buffering messages and by overlapping communication with rendering calculations as much as possible. Tests on the IBM SP2 demonstrate that these strategies provide high rendering rates and good scalability. For a dataset containing half a million tetrahedral cells, we achieve two frames per second for a 400×400-pixel image using 128 processors.

This work was supported by the National Aeronautics and Space Administration under Contract No. NAS1-19480 while the authors were in residence at the Institute for Computer Applications in Science and Engineering (ICASE), M/S 403, NASA Langley Research Center, Hampton, VA 23681-0001, USA.

Email: kma@icase.edu, tom@icase.edu

World Wide Web: http://www.icase.edu/~kma/, http://www.icase.edu/~tom/

1 Introduction

Three-dimensional aerodynamics calculations often use unstructured meshes to model objects with complex geometry. By applying finer meshes only to regions requiring high accuracy, both computing time and storage space can be reduced. This adaptive approach results in computational meshes containing data cells which are highly irregular in both size and shape. The lack of a simple indexing scheme for these complex grids makes visualization calculations on such meshes very expensive. Furthermore, in a distributed computing environment, irregularities in cell size and shape make balanced load distribution difficult as well.

The development of massively parallel rendering algorithms for irregular data has received comparatively little attention. Notably, Williams [23] developed a cell-projection volume rendering algorithm for finite element data running on a single SGI multiprocessor workstation. Uselton [21] designed a volume ray-tracing algorithm for curvilinear grids on a similar platform. In both cases, tests were performed with 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 renderers used shared-memory programming paradigms.

Giertsen and Petersen [8] designed a scanline volume rendering algorithm for distributed-memory systems based on Giertsen's previous sweep-plane approach [7], and implemented it on a network of workstations. In their approach the volume dataset is replicated on each workstation, and a master-slave scheme is used to dynamically balance the load. However, tests were performed with a maximum of four workstations, so the scalability of the algorithm and its implementation for massively parallel processing has yet to be demonstrated.

In recent work, Silva, Michell, and Kaufman [18] presented a more elaborate approach for rendering general irregular grids. Evolving from Giertsen's sweep-plane algorithm, their new strategy is careful to exploit spatial coherence. The algorithm is potentially parallelizable but tests were only performed on a Sun UltraSPARC-1. In addition, Wilhelms *et al.* [22] developed a hierarchical and parallelizable volume rendering technique for irregular and multiple grids. This algorithm favors coarse-grain parallelism for a shared-memory MIMD architecture.

Palmer and Taylor [16] devised a true distributed-memory ray-casting volume renderer for unstructured grids and demonstrated it on Intel's 512-node Touchstone Delta system. Their algorithm incorporated an adaptive screen-space partitioning scheme designed to reduce data movement caused by changes in the viewpoint. Another distributed-memory unstructured-grid

renderer was developed by Ma [12] for the Intel Paragon. This algorithm uses a graph-based partitioner to keep nearby cells together on the same processor, providing good locality during the ray-cast resampling process. The algorithm is somewhat tedious to use for postprocessing visualization applications because it requires both a preprocessing step to derive cell-connectivity information and a pre-partitioning step whenever the number of processors changes.

Our current research is inspired by the trend toward larger numbers of processors in large-scale scientific computing platforms, as typified by the terascale architectures being installed for the U.S. Department of Energy's ASCI program. To support applications which use these systems, we must develop visualization tools which are appropriate to the architectures. We focus on scalability and flexibility as two key design criteria. To address these issues, we propose a static load balancing scheme coupled with an asynchronous communication strategy which overlaps the rendering calculations with transfer of ray segments. Our results indicate that this approach compares favorably with previous unstructured-grid volume rendering algorithms for similar architectures.

Another problem shared by many existing visualization algorithms for unstructured data is the need for a significant amount of preprocessing. One step extracts additional information about the mesh, such as connectivity, in order to speed up later visualization calculations. Another step may be needed to partition the data based on the particular parallel computing configuration being used (number of processors, communication parameters, etc.). To reduce the user "hassle factor" as much as possible and avoid increasing the data size or replicating data, we want to eliminate these preprocessing steps. While this provides flexibility and convenience, it also means less information is available for optimizing the rendering computations. We have elected to sacrifice a small amount of performance in favor of enhanced usability.

In the remainder of the paper, we describe the strategies we have developed to achieve both scalability and flexibility for volume rendering of unstructured data. We also present detailed experimental performance results obtained with up to 128 nodes on an IBM SP2. We conclude with a discussion of plans for future work, including opportunities for improving the algorithms presented here.

2 Overview of the Algorithm

Our new parallel rendering algorithm performs a sequence of tasks as shown in Figure 1. The volume data is distributed in round robin fashion with the intention of dispersing nearby

cells as widely as possible among processors. The image space is partitioned using a simple scan-line interleaving scheme.

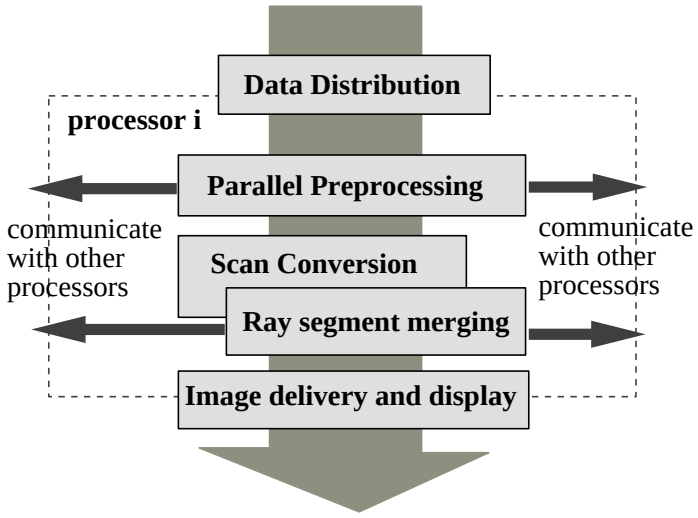

Figure 1: The volume rendering pipeline. This procedure is replicated on every processor.

A preprocessing step then performs a parallel, synchronized partitioning of the volume data to produce a hierarchical representation of the data space. This spatial tree is used in the rendering step to optimize the compositing process and to reduce runtime memory consumption.

To offer maximum freedom in data distribution, a cell-projection rendering method is used. However, data cells are not pre-sorted in depth order. Instead, each processor scan converts its local cells to produce many ray segments, which are routed to their final destinations in image space for merging. A double-buffering scheme is used in conjunction with asynchronous *send* and *receive* operations to reduce overheads and overlap communication of ray segments with rendering computations. Scan conversion of data cells and merging of ray segments proceed together in multiplexed fashion. When scan conversion and ray-segment merging are finished, each processor sends its completed subimage to a host computer which assembles them for display. A detailed description of each of the steps of the algorithm is given in the following sections.

3 Data Distribution

Ideally, data should be distributed in such a way that every processor requires the same amount of storage space and incurs the same computational load. There are several factors which affect this. For the sake of concreteness, we assume meshes composed of tetrahedral cells; similar considerations apply to other types of unstructured grids. First, there is some cost for scan converting each cell. Variations in the number of cells assigned to each processor will produce variations in workloads. Second, cells come in different sizes and shapes. The difference in

size can be as large as several orders of magnitude due to the adaptive nature of the mesh. As a result, the projected image area of a cell can vary dramatically, which produces similar variations in scan conversion costs. Furthermore, the projected area of a cell also depends on the viewing direction. Finally, voxel values are mapped to both color and opacity values. An opaque cell can terminate a ray early, thereby saving further merging calculations, but introducing further variability in the workload.

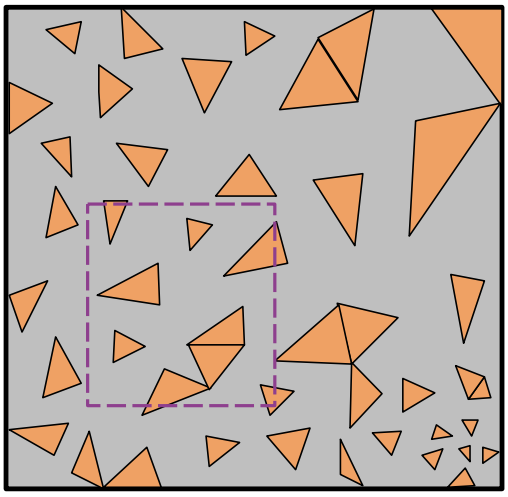

Figure 2: Local load imbalances are reduced by processing cells from throughout the spatial domain.

If ray-cast rendering were used, we would want to assign groups of connected cells to each processor so that the rendering process can be optimized by exploiting cell-to-cell coherence. But connected cells are often similar in size and opacity, so that grouping them together exacerbates load imbalances, making it very difficult to obtain satisfactory partitionings. We have therefore chosen to take the opposite approach, dispersing connected cells as widely as possible among the processors. Thus each processor is loaded with cells taken from the whole spatial domain rather than from a small neighborhood as shown in Figure 2. Satisfactory scattering of the input data can generally be achieved with a simple round robin assignment policy. With sufficiently many cells, the computational requirements for each processor tend to average out, producing an approximate load balance.

This approach also satisfies our requirement for flexibility, since the data distribution can be computed trivially for any number of processors, without the need for an expensive pre-processing step.

By dispersing the grid cells among processors, we also facilitate a very important visualization operation for unstructured data—zoom-in viewing. Because of the highly adaptive nature of unstructured meshes, the most important simulation results are usually associated with a relatively small portion of the overall spatial domain. The viewer normally takes a peek at the overall domain and then immediately focuses on localized regions of interests, such as areas with high velocity or gradient values. This zooming operation introduces challenges for efficient visualization in a distributed computing environment. First, locating all of the cells which reside

within the viewing region can be an expensive operation. Our solution, described in the next section, is to employ a spatial partitioning tree to speed up this cell searching. Second, if data cells are distributed to processors as connected components, zooming in on a local region will result in severe load imbalances, as a few processors are left with all of the rendering calculations while others go idle (Figure 3).

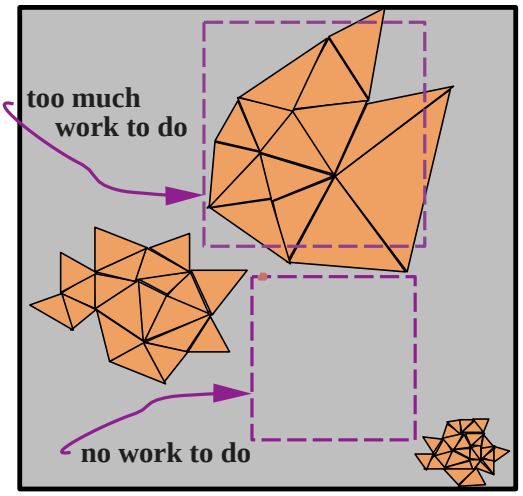

Figure 3: Zoomed-in viewing results in severe load imbalances when connected cells are grouped together.

An alternative approach is to distribute clusters of connected cells to each processor and hope that some data can be shared and some ray segments can be merged locally. This approach has several drawbacks. First, an appropriate heuristic must be found for determining the optimal cluster size. Second, the data must be preprocessed to compute cell connectivity information. Finally, the data must be partitioned in a way which preserves locality and maintains a relatively balanced load across processors. The cost and inconvenience of these additional steps can make this approach unattractive.

Although the round-robin distribution discourages data sharing, our rendering algorithm only requires minimum data—the cell and node information. No connectivity data are needed. Each cell takes 16 bytes to store four node indices and each node takes 16 bytes to store three coordinates and a scalar value. As a result, in the worst case of no sharing of any node information, $80n$ bytes of data must be transferred in order to distribute n cells to a processor. As an example, distributing a dataset of 1 million cells across 128 processors requires an average of 640,000 bytes of data to be transferred to and stored at each processor.

In addition to the object space operations on mesh cells, we also need to evenly distribute the pixel-oriented ray-merging computations. Local variations in cell sizes within the mesh lead directly to variations in depth complexity in image space. Therefore we need an image partitioning strategy which disperses the ray-merging operations as well. In our current implementation, we assign successive scanlines to processors in round-robin fashion, a technique often known as *scanline interleaving*. This works reasonably well as long as the vertical resolution of the image is several times larger than the number of processors. With more processors, we conjecture that a

finer-grained pixel interleave may be advantageous. At each pixel location, we maintain a linked list of ray segments, which are merged to form the final pixel value. The pixel merging process is described in more detail in subsequent sections.

4 Space Partitioning Tree

As described in the previous section, our round-robin data distribution scheme helps to achieve flexibility and produces an approximate static load balance. However, it totally destroys the spatial coherence among neighboring mesh cells, making an unstructured dataset even more irregular. We would like to restore some ordering so that the rendering step may be performed more efficiently.

Our approach to this problem is to have all processors render the cells in the same neighborhood at about the same time. Ray segments generated for a particular region will consequently arrive at their image-space destinations within a relatively short window of time, allowing them to be merged early. This early merging reduces the length of the ray-segment list maintained by each processor, which benefits the rendering process in two ways: first, a shorter list reduces the cost of inserting a ray segment in its proper position within the list; and second, the memory needed to store unmerged ray segments is reduced.

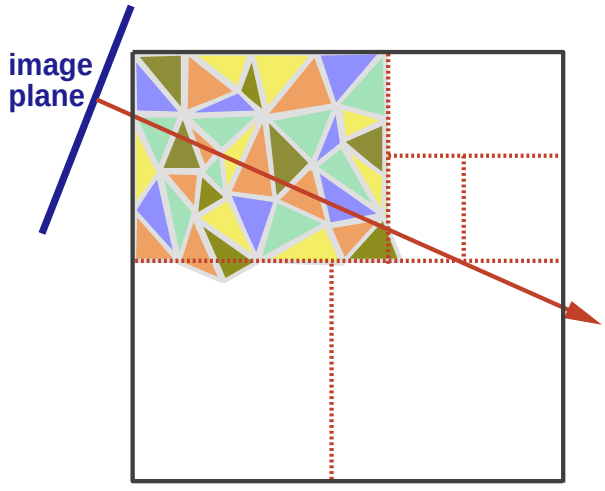

Figure 4: A global spatial partitioning assigns cells to subregions for rendering.

To provide the desired ordering, data cells can be grouped into local regions using a hierarchical spatial data structure such as an octree [10] or *k*-d tree [1]. We prefer the *k*-d tree since it supports adaptive partitioning along orthogonal coordinate planes and allows straightforward determination of the depth ordering of the resulting regions. Figure 4 shows rendering of a region within such a partitioning, where the different colored cells are stored and scan converted by different processors.

The tree should be constructed cooperatively so that the resulting spatial partitioning is exactly the same on every processor.

After the data cells are initially distributed, all processors participate in a synchronized parallel partitioning process. The algorithm works as follows:

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

At the end of the partitioning process, each processor has an identical list of regions, with each region representing approximately the same rendering load as the corresponding region on every other processor. If all processors render their local regions in the same order, loose synchronization will be achieved due to the similar workloads, allowing early ray-merging to take place within the local neighborhoods. The *k*-d tree also allows for fast searching of cells within a spatial region specified by a zoom-in view. Note that our current implementation does not guarantee a well-balanced tree, but the extra searching overhead is insignificant compared to the time required for the rendering calculations. We also observe that the spatial regions can also serve as workload units should we ever need to perform dynamic load balancing.

5 Rendering

Direct volume rendering algorithms can be classified into either ray-casting [6][11][12][20] or projection methods [15][17]. Projection methods may be further categorized as cell-projected [22], slice-projected [24], or vertex-projected [14]. We have chosen a cell-projection method similar to [15] because it offers more flexibility in data distribution and is more accurate.

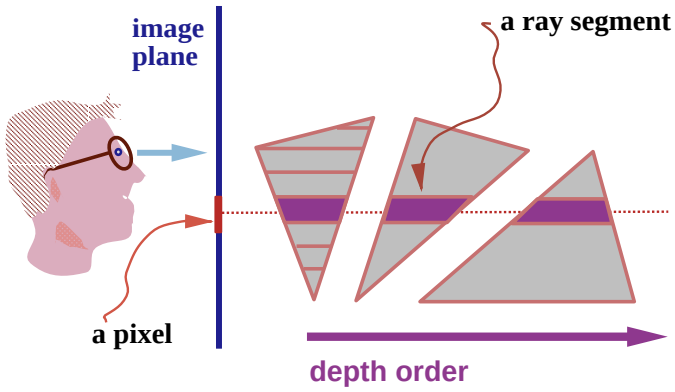

Figure 5: Cell projection rendering.

During rendering, processors follow the same path through the spatial partitioning tree, processing all of the cells at each leaf node of the tree. Each cell is scan-converted independently, and the resulting ray segments are routed to the processor which owns the corresponding image scanline. As adjacent ray segments are received, they are merged using the standard Porter-Duff *over* operator. Figure 5 illustrates the process.

Since the ray segments which contribute to a given pixel arrive in unpredictable order, each ray segment must contain not only a sample value and pixel coordinates, but also starting and ending depth values which are used for sorting and merging within the pixel's ray segment list. For the types of applications currently envisioned, we expect from 10^6 to 10^8 ray segments to be generated for each image; at 16 bytes per segment, aggregate communication requirements are on the order of 10^7 to 10^9 bytes per frame. Clearly, efficient management of the communication is essential to the viability of our approach. The next section presents our solution to this problem.

6 Task Management with Asynchronous Communication

Good scalability and parallel efficiency can only be achieved if the parallelization penalty and communication cost are kept low. As described above, our design reduces computational overheads due to parallelization by eliminating the need to pre-sort the cells and by lowering the post-sorting cost and memory consumption.

To manage communication costs, we adopt an asynchronous communication strategy which was originally developed for a parallel polygon renderer [5] and later improved for use in the PGL rendering system [3][4]. In the current context, the key features of this approach include:

- asynchronous operation, which allows processors to proceed independently of each other during the rendering computations;

- multiplexing of the object-space cell computations with the image-space ray merging computations;
- overlapped computation and communication, which hides data transfer overheads and spreads the communication load over time; and
- buffering of intermediate results to amortize communication overheads.

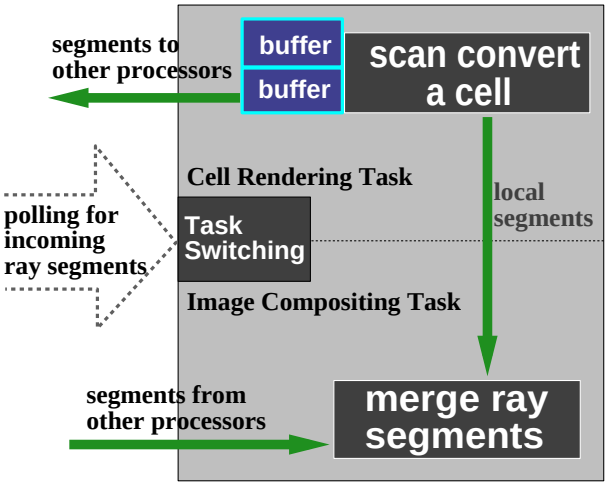

Figure 6: Task management with asynchronous communication.

During the course of rendering, there are two main tasks to be performed: scan conversion and image compositing. High efficiency is attained if we can keep all processors busy doing either of these two tasks. Logically, the scan conversion and merging operations represent separate threads of control, operating in different computational spaces and using different data structures. For the sake of efficiency and portability, however, we have chosen to interleave these two operations using a polling strategy. Figure 6 illustrates at a high level the management of the two tasks and the accompanying communication. Each processor starts by scan converting one or more data cells. Periodically the processor checks to see if incoming ray segments are available; if so, it switches to the merging task, sorting and merging incoming rays until no more input is pending.

Due to the large number of ray segments generated, the overhead for communicating each of them individually would be prohibitive in most architectures. Instead, it is better to buffer them locally and send many ray segments together in one operation. To supplement this, we employ asynchronous *send* and *receive* operations, which allow us to overlap communication and computation, reduce data copying overheads in message-passing systems, and decouple the sending and receiving tasks. We have found that this strategy is most effective when two or more ray segment buffers are provided for each destination. While a send operation is pending for a full buffer, the scan conversion process can be placing additional ray segments in its companion buffer. In the event that both buffers for a particular destination fill up before the first send com-

pletes, we can switch to the ray merging task and process incoming segments while we wait for the outbound congestion to clear (in fact, this is essential to prevent deadlock).

There are two parameters that the user may specify to control the frequency of task switching and communication. The first parameter is the polling interval, i.e., the number of cells to be processed before checking for incoming ray segments. If polling is too frequent, excessive overheads will be introduced; if it isn't often enough, the asynchronous communication scheme will perform poorly as outbound buffers clog up due to pending *send* operations. The second parameter is the buffer depth, which indicates how many ray segments should be accumulated before an asynchronous *send* is posted. If the buffer size is too small, the overheads for initiating *send* and *receive* operations will be excessive, resulting in lowered efficiency. On the other hand, buffers that are too large can introduce delays for processors which have finished their scan conversion work and are waiting for ray segments to merge. Large buffers are also less effective at spreading the communication load across time, resulting in contention delays in bandwidth-limited systems.

The most effective choice of buffer size depends on the number of processors in use, the number of ray segments to be communicated, and the characteristics of the target architecture. As one may suspect, the polling frequency should be selected in accordance with the buffer size. As a general rule, polling should be performed more frequently with smaller buffer sizes or larger numbers of processors. We present empirical results illustrating this relationship in the next section.

The asynchronous nature of the communication algorithm makes it impossible for a processor to determine by itself whether rendering is complete. Care must be taken to avoid deadlock or loss of ray segments. We suggest a procedure similar to that described in [5], in which a designated node coordinates the termination process by collecting local termination messages and broadcasting a global termination signal. A final global synchronization operation ends the overall rendering process.

7 Test Results

For convenience, we have used a small unstructured grid dataset containing about 0.5 million tetrahedral cells for our initial experiments. The dataset represents flow over an aircraft wing with an attached missile. All test results are based on the average time of rendering this dataset into a 400×400 pixel image for six different viewing directions. The surface mesh of the

wing dataset is displayed in Figure 7, illustrating the large variations in cell size and density which often arise in unstructured grids. Figure 8 shows a volume-rendered view of the overall data domain, in which the area of the wing is relatively small. Figure 9 shows a zoomed-in view of the flow surrounding the wing. Feature lines [13] have been added to assist in relating the shocks to the structure of the wing. We plan to conduct further tests using a larger dataset with several million cells.

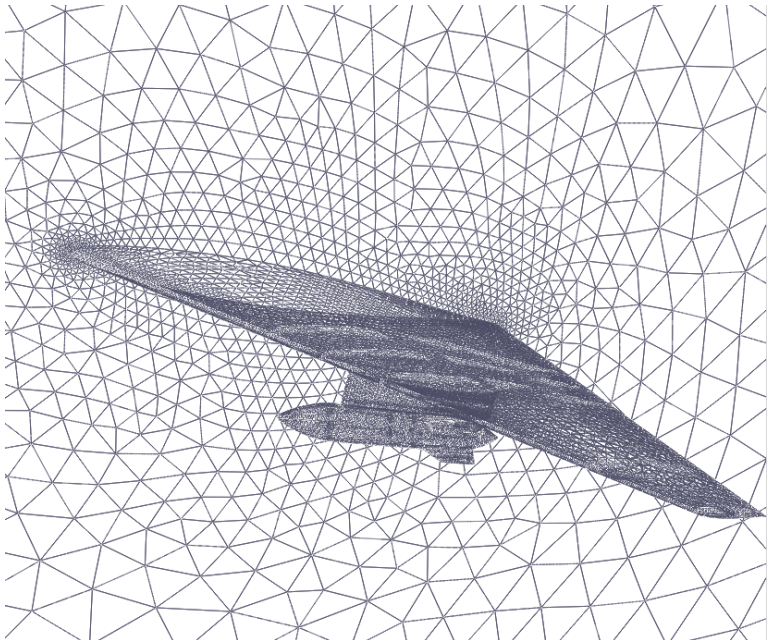

Figure 7: Surface mesh structure of the aircraft wing dataset.

We implemented our volume renderer in the C language using MPI message passing [19] for interprocessor communication. All tests were run on IBM SP2 systems located at NASA's Ames and Langley Research Centers. The SP2 [9] is a distributed-memory architecture which employs a switch-based processor interconnect. The NASA SP2s are populated with "wide" nodes based on a 66.7 MHz POWER2 chip set and incorporate a second-generation switch with a peak node-to-node bandwidth of about 34 MB/s.

Figure 10 plots rendering time in seconds *vs.* the number of processors. With 128 nodes we can render our test dataset at two frames per second (excluding display time). The annotations to the right of the data points indicate the buffer depth (in number of ray segments) and polling frequency (high, medium, or low). Our experiments indicate that with large numbers of processors (32 and above), two different strategies for setting the buffer depth and polling frequency provide equivalent performance. The reasons for this are not completely clear; however our previous experience with parallel polygon renderers indicates that the polling frequency is not a critical parameter. Furthermore, the communication algorithm has a built-in feedback mechanism: if sending becomes blocked due to full buffers, the processor switches to the ray merging task and begins receiving, regardless of the value of the polling interval.

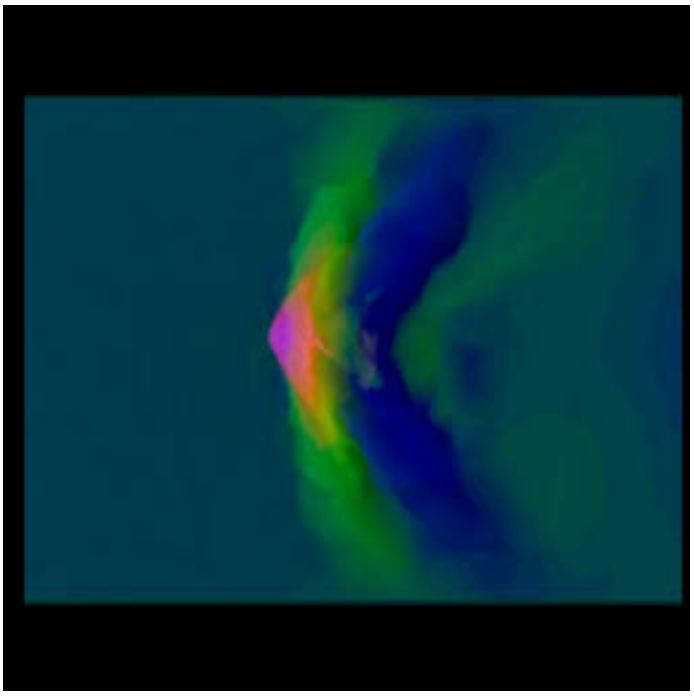

Figure 8: Volume rendering of the flowfield around an aircraft wing.

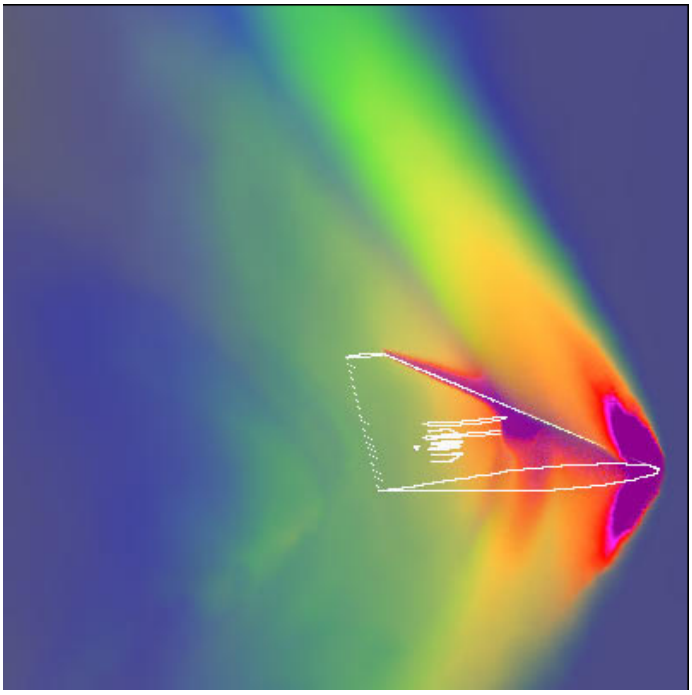

Figure 9: Close-up view of the wing region. Feature lines assist in relating the volume data to the underlying structure.

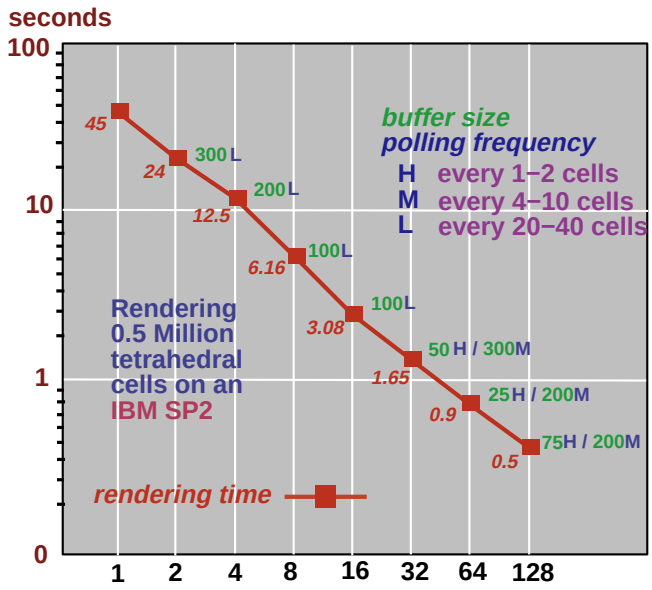

Figure 10: Rendering time.

A better picture of the parallel performance is obtained from Figure 11 which shows the speedups and parallel efficiencies obtained as the number of processors varies from 1 to 128. With 128 processors, we achieve a speedup of 90, for a parallel efficiency of 70%. Some of this degradation is due to load imbalance, as shown in Figure 12. However, the static load balancing scheme employed here compares favorably with our earlier algorithm [12] for the Intel Paragon, particularly for smaller numbers of processors. To obtain a better understanding of the remaining load imbalance and other parallel overheads, we need to examine the performance characteristics of the renderer in more detail.

Figures 13 and 14 show the per-processor contributions of various execution time components. These are measured by inserting calls to a high-resolution, low-overhead assembly language event timer at strategic locations in the code. The overhead for the timer calls is deducted to yield accurate estimates of the actual runtime due to each component. Each of the eleven components is described in Table 1.

As the graphs show, the object-space computations (viewing transformations and scan conversion) are well-balanced, with only minor variations in execution time. However the image-space operations (ray merging) show a distinct pattern, with the higher- and lower-numbered processors having somewhat more work to do. We believe that this is caused by "hot spots" in the volume data, i.e., small regions with high cell densities that map to only a few scanlines in the final image. If we translate the viewpoint up or down slightly, the peaks and valleys of the *t_merge* component shift cyclically, confirming our hypothesis. This suggests that a finer-grained image distribution (*e.g.*, pixel interleaving) could provide better static load balancing with large numbers of processors.

The graphs also show the effect of the polling frequency and buffer size parameters. Figure 13 shows execution times with a buffer depth of 25 and polling after each cell is scan con-

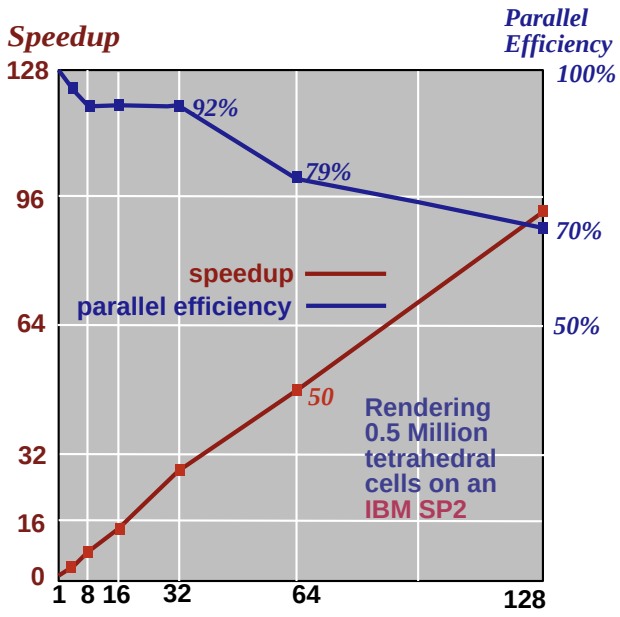

Figure 11: Speedup and parallel efficiency.

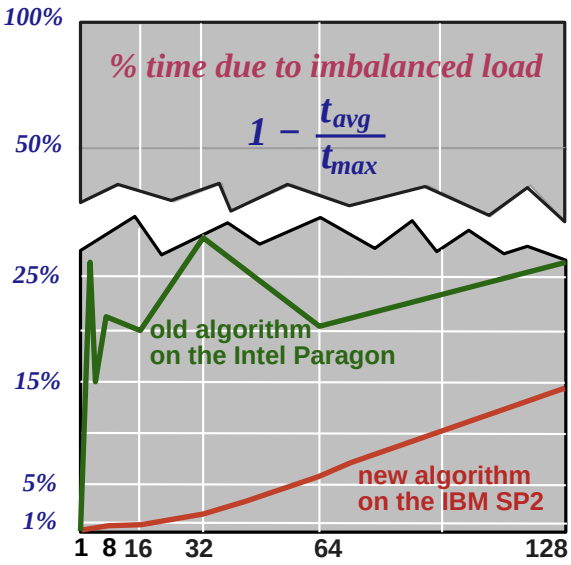

Figure 12: Time delay due to unbalanced load.

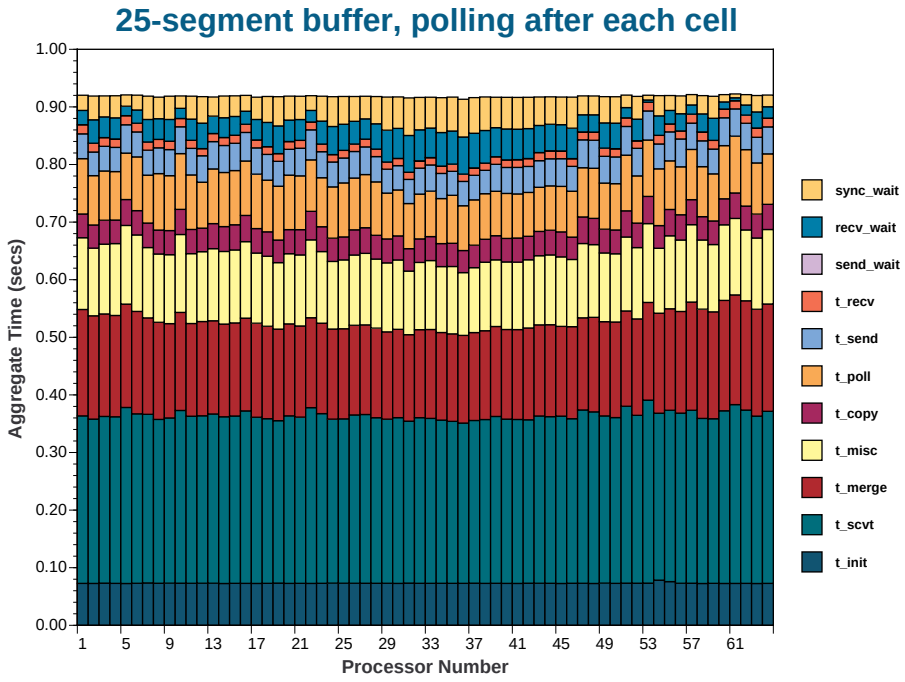

Figure 13: Execution time components for 64 processors; buffer depth = 25, polling after each cell.

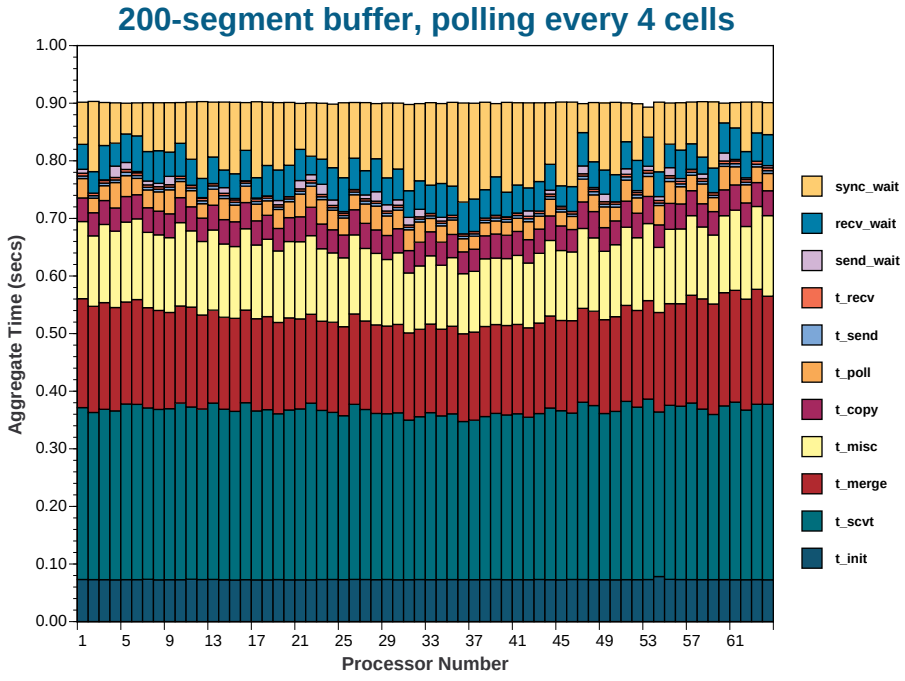

Figure 14: Execution time components for 64 processors; buffer depth = 200, polling after every 4th cell.

t_init	Time to perform viewing transformations on the cell data, and other beginning-of-frame initializations.
t_scvt	Time to scan convert local cells.
t_merge	Time to sort and merge ray segments.
t_misc	Other computation: tree traversal, control flow, etc.
t_copy	Overhead for placing ray segments in output buffers.
t_poll	Time required to check for incoming messages.
t_send	Time for sending ray buffers.
t_recv	Time for receiving ray buffers.
send_wait	Idle time when outgoing send buffers are blocked and no incoming ray segments are available for merging.
recv_wait	Idle time waiting for ray segments to arrive after all local cells have been processed.
sync_wait	Time for termination detection and end-of-frame synchronization.

Table 1: Execution time components.

verted; the conditions for Figure 14 are identical except that the segment buffers are eight times larger and polling is performed after every four cells. We see that larger buffers and longer polling intervals significantly reduce the sending, receiving, and polling components. However, larger buffers also lead to increases in idle time (*send_wait*, *recv_wait* and *sync_wait*), presumably because idle processors have to wait longer for new work to arrive. The net result is that larger buffers and longer polling intervals provide only slight improvements in the total rendering time. We suspect that some of this idle time can be eliminated by tuning our current termination algorithm.

Finally, we note that the *send_wait* time is negligible in both cases. This is not always true. We have seen instances in which the spatial partitioner generates regions which have very small projected areas in image space. The result is that all of the message traffic generated for that region bombards a few nodes, creating congestion which causes the output buffers to back up. Since the traffic is focused in a very limited portion of the image, most nodes have no incoming ray segments to process, and thus fall idle. There are several strategies which are useful in combating this problem. First, pixel interleaving could be expected to distribute the load across more pro-

cessors, unless the regions become tiny (a distinct possibility with highly adaptive grids). Second, the parameters on the partitioner can be adjusted so that regions are not allowed to fall below a certain size. Alternatively, the partitioner could be configured to produce regions with high aspect ratios, resulting in larger footprints in image space. Unfortunately, this last approach would partially defeat the purpose of using a partitioner in the first place, i.e., to facilitate early ray merging.

To gauge the effectiveness of our partitioning strategy, we have collected statistics on the maximum number of unmerged ray segments which must be stored on any processor during the rendering process. Our experiments indicate that with 64 regions, the number of unmerged segments on any given processor is at most 15-20% of the total number received.

8 Conclusions

By combining a spatial partitioning scheme with techniques which were originally developed for parallel polygon rendering, we have produced a volume renderer for unstructured meshes which employs inexpensive static load balancing to achieve high performance and reasonable efficiency with modest memory consumption. We believe that our algorithm is currently the most effective one available for rendering complex unstructured grids on distributed-memory message-passing architectures. Detailed performance experiments with up to 128 processors lead us to believe that further improvements are possible.

We plan to conduct additional tests with larger datasets, different image sizes, more processors, and other architectures. With larger datasets, the number of ray segments generated may increase significantly, and we need to assess the impact of this additional communication load on overall performance. We also want to investigate the potential for finer-grained image partitionings and improved termination strategies to improve the parallel efficiency of our approach. The ultimate goal is a fast, scalable volume renderer which can handle tens of millions of grid cells using hundreds or thousands of high-performance processors.

Acknowledgments

The experiments described in this paper were performed using parallel systems provided by NASA's Computational Aerosciences project under the auspices of the national High Performance Computing and Communications Program.

References

[1] J. L. Bentley. Multidimensional binary search trees used for associative searching, *Communications of the ACM*, **18**(8):509–517, September 1975.

[2] Judy Challinger. Scalable parallel volume ray-casting for nonrectilinear computational grids, *Proceedings 1993 Parallel Rendering Symposium*, ACM Press, October 1993, pp. 81–88.

[3] T. W. Crockett. Design considerations for parallel graphics libraries, *Proceedings Intel Supercomputer Users Group 1994 Annual North America Users Conference*, Intel Supercomputer Users Group, June 1994, pp. 3-14.

[4] T. W. Crockett. PGL: A Parallel Graphics Library for Distributed Memory Applications, Interim Report No. 29, ICASE, NASA Langley Research Center, Hampton, VA, February 1997, http://www.icase.edu/reports/interim/29/ (HTML only).

[5] T. W. Crockett and T. Orloff. Parallel polygon rendering for message-passing architectures, *IEEE Parallel and Distributed Technology*, **2**(2):17–28, Summer 1994.

[6] Michael P. Garrity. Raytracing irregular volume data, *Proceedings 1990 Workshop on Volume Visualization*, special issue of *Computer Graphics*, **24**(5):35–40, November 1990.

[7] Christopher Giertsen. Volume visualization of sparse irregular meshes, *IEEE Computer Graphics & Applications*, **12**(2):40–48, March 1992.

[8] Christopher Giertsen and Johnny Petersen. Parallel volume rendering on a network of workstations, *IEEE Computer Graphics & Applications*, **13**(6):16–23, November 1993.

[9] IBM Corporation. RS/6000 SP System, http://www.rs6000.ibm.com/hardware/largescale/ (current July 1997).

[10] C. L. Jackins and S. L. Tanimoto. Octrees and their use in representing three-dimensional objects, *Computer Graphics and Image Processing*, **14**(3):249–270, September 1980.

[11] M. Levoy. Efficient ray tracing of volume data, *ACM Transactions on Graphics*, **9**(3):245–261, July 1990.

[12] Kwan-Liu Ma. Parallel volume ray-casting for unstructured-grid data on distributed-memory architectures, *Proceedings 1995 Parallel Rendering Symposium*, ACM Press, October 1995, pp. 23-30.

[13] Kwan-Liu Ma and Victoria Interrante. Extracting feature lines from 3D unstructured grids, *Proceedings Visualization '97*, October 1997 (to appear).

[14] Xiaoyang Mao. Splatting of non-rectilinear volumes through stochastic resampling, I*EEE Transactions on Visualization and Computer Graphics*, **2**(2):156–170, June 1996.

[15] Nelson Max, Pat Hanrahan, and Roger Crawfis. Area and volume coherence for efficient visualization of 3D scalar functions, *Computer Graphics*, **24**(5):27–33, November 1990.

[16] M.E. Palmer and S. Taylor. Rotation invariant partitioning for concurrent scientific visualization, *Proceedings Parallel CFD '94*, Elsevier Science Publishers B.V., 1994.

[17] Peter Shirley and Allan Tuchman. A polygon approximation to direct scalar volume rendering, *Proceedings 1990 Workshop on Volume Visualization*, special issue of *Computer Graphics*, **24**(5):63–70, November 1990.

[18] Claudio Silva, J. Michell, and A. Kaufman. Fast rendering of irregular volume data, *Proceedings 1996 Volume Visualization Symposium*, ACM SIGGRAPH, October 1996, pp.15-22.

[19] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference*, MIT Press, 1995.

[20] L. M. Sobierajski and A. E. Kaufman. Volume ray tracing, *Proceedings 1994 Volume Visualization Symposium*, ACM SIGGRAPH, October 1994, pp. 11-18.

[21] Sam Uselton. Volume rendering on curvilinear grids for CFD, AIAA Paper 94-0322, 32nd Aerospace Sciences Meeting & Exhibit, 1994.

[22] Jane Wilhelms, Allen Van Gelder, Paul Tarantino, and Jonathan Gibbs. Hierarchical and parallelizable direct volume rendering, *Proceedings Visualization '96*, IEEE CS Press, October 1996, pp. 57–64.

[23] Peter L. Williams. Parallel volume rendering finite element data, *Proceedings Computer Graphics International '93*, Lausanne, Switzerland, June 1993.

[24] R. Yagel, D. M. Reed, A. Law, P. Shih, and N. Shareef. Hardware assisted volume rendering of unstructured grids by incremental slicing, *Proceedings 1996 Volume Visualization Symposium*, ACM SIGGRAPH, October 1996, pp. 55–62.