Title
Mutli-threaded Rendering of Unstructured-Grid Volume Data on the SGI Origin 2000

Authors
Hofsetz, Christian
Ma, Kwan-Liu

Publication Date
2000

Peer reviewed

Multi-threaded Rendering Unstructured-Grid Volume Data on the SGI Origin 2000

Christian Hofsetz Kwan-Liu Ma *
University of California-Davis

Abstract

This paper presents a work-in-progress. The objective of our study is to derive an optimal design for high-performance rendering of irregular-grid volume data on the increasingly popular, distributed shared-memory parallel supercomputers. We experiment with a multi-threaded volume rendering algorithm for three-dimensional unstructured-grid data and discuss its performance on the SGI Origin 2000. Our preliminary test results demonstrate good rendering rates with a moderate number of processors. But we observe surprisingly poor processor scalability. We thus investigate how task partitioning and runtime memory management affect, in particular, the scalability of parallel rendering. We discover that in terms of algorithmic complexity and programming effort to attain the optimal performance on a shared memory architecture it can be as challenging and demanding as on a distributed-memory architecture.

Keywords: Distributed shared-memory, memory management, multithreading, parallel rendering, performance evaluation, unstructured meshes, volume rendering

1 Introduction

Increasingly, 3-d unstructured meshes are used in many scientific and engineering applications to model complex structures and phenomena to lower the simulation-time processing and memory requirements. For large-scale problems, massively parallel supercomputers are used to attain high level of accuracy and reduce simulation time, but the sheer size of the solution data generated by the large-scale simulations easily overwhelms scientists' capability to analyze and understand them. Appropriate visualization tools for large-scale unstructured-grid data are therefore in pressing need. Lacking appropriate data analysis and visualization tools often forces scientist to reduce the solutions by coarsening which defeats the original purpose of performing the high-resolution simulations.

In many cases feature extraction and data management techniques can be used to solve the large-scale data visualization problems, but distributing both the large amount of data and visualization calculations to a parallel computer continues to be a reliable way allowing scientists to look at their data at the highest possible resolution. Coupled with other techniques, parallel rendering can also meet the demand of time-critical visualization applications. In this paper, we describe the design of a multi-threaded algorithm for rendering three-dimensional unstructured-grid data, its multithreaded implementation, and performance on the SGI Origin 2000 (O2K) which is a distributed shared-memory (DSM) system.

SGI describes the O2K as a scalable shared-memory processor system that offers the benefits of both shared-memory systems and distributed-memory systems. Two CPUs are connected through a "hub" to a memory to form a "node". Multiple nodes are linked in a hypercube configuration through a set of routers. Each processor is a MIPS R10000. Since memory is physically distributed, the memory access time depends on the proximity of a CPU to the memory that it accesses. A program must use the cache efficiently to achieve good performance.

Our goal is to achieve the best rendering rates and parallel efficiency possible on the O2K, and to compare the performance with a message-passing code [10, 9] using MPI. We use a small 16-processor O2K in our laboratory for code development. Tests are performed on much larger systems operated at the NASA Ames Research Center and the Los Alamos National Laboratory using up to 120 processors with a data set from an aerodynamic simulation.

2 Parallel Rendering Unstructured-Grid Data

Unstructured meshes are used 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.

Preprocessing is often used to derive, for example, connectivity and/or visibility information for facilitating the rendering calculations. Further improvement can be made by

*Department of Computer Science, University of California, One Shields Avenue, Davis, CA 95616-8562, e-mail: {hofsetz,ma}@cs.ucdavis.edu.

using parallel and distributed rendering. However, the irregularities in cell size and shape of unstructured-grid data make load balancing particularly a challenging problem. The development of parallel algorithms for rendering data on unstructured-grid (or, more generally, on non-Cartesian grid) has thus received a growing attention.

2.1 On Shared-Memory Computers

Williams [18] used an 8-processor SGI 4D/VGX to render unstructured-grid data, and Uselton [16] experimented with a ray-casting volume rendering code on an SGI Power Challenge by using up to 19 processors for curvilinear-grid data. They all achieved very good parallel efficiency numbers. Wilhelms, et al. [17] developed a hierarchical and parallelizable volume rendering technique for non-Cartesian and multiple grids. This algorithm favors coarse-grain parallelism for a shared-memory MIMD architecture.

Nieh and Levoy [11] designed a parallel volume rendering algorithm based on a task-queue image partitioning technique, and implemented it on the Stanford DASH. They obtained nearly linear speedup using up to 48 processors. Later, Lacroute [5] ported his shear-warp volume rendering algorithm to DASH. While he obtained impressive rendering rates, poor scalability was observed on DASH when using up to 32 processors for that implementation. Both Nieh's and Lacroute's renderers are for rendering regularly sampled data.

A very thorough study on parallel shared-memory architectures using rendering applications was done by Palmer, Totty and Taylor [13]. They studied memory hierarchy effects and their interaction with parallel partitioning and load balancing on a 16-processor SGI Power Challenge by using volume rendering applications. One frame per second can be achieved for rendering a one-gigabyte MRI data set. However, the efficiency of the renderer beyond 16 processors is unclear.

The most impressive results were reported by Challinger [3]. for her work on the BBN TMC2000 (an MIMD distributed shared-memory architecture) using up to 128 processors for rendering curvilinear data. As high as 80-90% parallel efficiency was observed. Compared to unstructured-grid data, parallelization of rendering calculations for data on rectilinear or curvilinear grids is a slightly easier problem because of the data layout and access order can be determined straightforwardly to improve cache performance.

2.2 On Distributed-Memory Computers

Most of the previous efforts on distributed-memory parallel computers were targeted to utilizing a large number of processors to solve large-scale data visualization problems. For example, Palmer and Taylor [12] developed a 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.

To render unstructured-grid volume data, Ma [7] used 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 clumsy 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.

Ma and Crockett [8] later developed a highly scalable algorithm based on a static load balancing scheme coupled with an asynchronous communication strategy which overlaps the rendering calculations with transfer of image data. This algorithm uses a cell projection method which removes the need for preprocessing data to generate, say, cell connectivity information. A low-cost parallel space partitioning step is performed and the resulting partitioning tree is used in the rendering step to restore locality which is lost in the round-robin data distribution step. This optimization results in more efficient image compositing and reducing runtime memory requirements.

2.3 On Origin 2000

The rendering performance of Ma and Crockett's algorithm which uses message-passing communication is superior on distributed-memory parallel computers like Intel Paragon, IBM SP2 [10], and SGI Cray T3E [9]. Excellent scalability over several hundreds of processors has been demonstrated. When running the same renderer on the O2K, we observed very poor scalability as soon as more than 16 processors were used.

Figure 1 and 2 show the execution time components for 64 processors on the SGI/Cray T3E and on the O2K, respectively. The disparity in performance on two very different architectures is apparent. Specifically, when increasing the number of processors used from 16 to 32 on O2K, parallel efficiency drops from 84% to 26%. The test data set used contains 567,863 tetrahedral cells. Although this dataset is considered small, it helps reveal the communication and overhead cost. A large data set would result in timing results making the overhead time negligible.

We have not been able to completely identify the exact cause of this significant performance degradation on the O2K, but we suspect that poor memory management inhibits scalability. Notably, Figure 2 reveals certain linear dependency which might have been a result of poor implementation in system software. On the other hand, the message-passing algorithm is highly asynchronous and involves constantly sending many messages during the whole course of the rendering calculations. It seems the asynchronous processing implemented works against the O2K's share-memory processing. Our results are consistent with the findings in [4, 2], which have motivated this work.

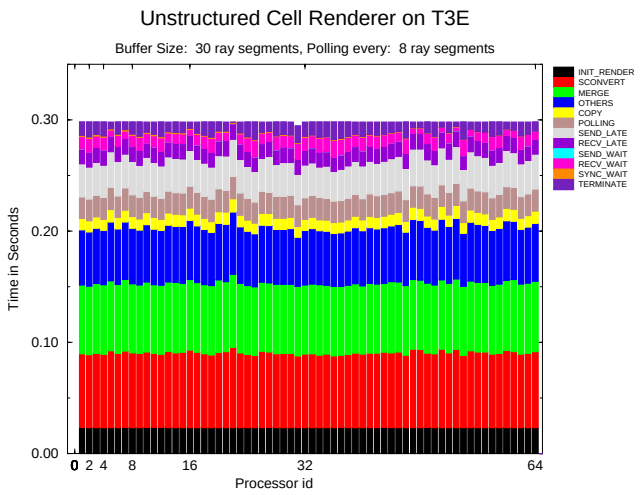

Figure 1: Execution time components of the message-passing code on SGI/Cray T3E using 64 processors.

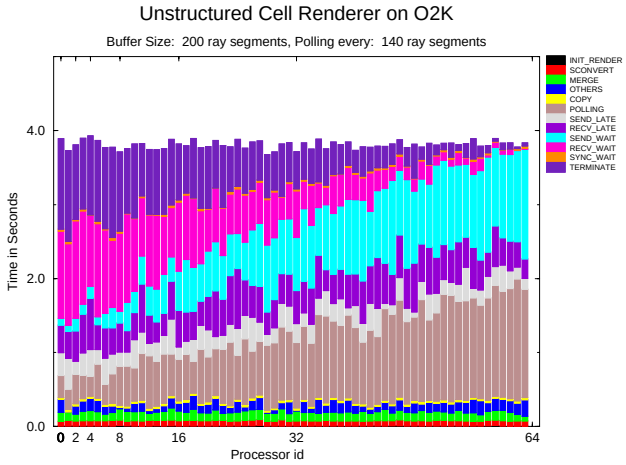

Figure 2: Execution time components of the message-passing code on the O2K using 64 processors.

We are curious to see how a multi-threaded implementation of volume rendering for unstructured-grid data would perform on the O2K. Our conjecture is that better rendering efficiency can be achieved on the O2K by carefully exploiting its distributed share-memory architecture. We have learned from others quite different results. Both Lueke [6] and Cavin [2] showed that high scalability on O2K seems to be an unattainable goal for some application problems, but Parker et al. [14] were able to demonstrate over 90% parallel efficiency using up to 128 processors on the O2K for ray-tracing rendering large-scale volume data. The scalability the latter achieved proves the O2K a scalable parallel architecture. We, therefore, would like to verify with our own experimental results using a different rendering algorithm..

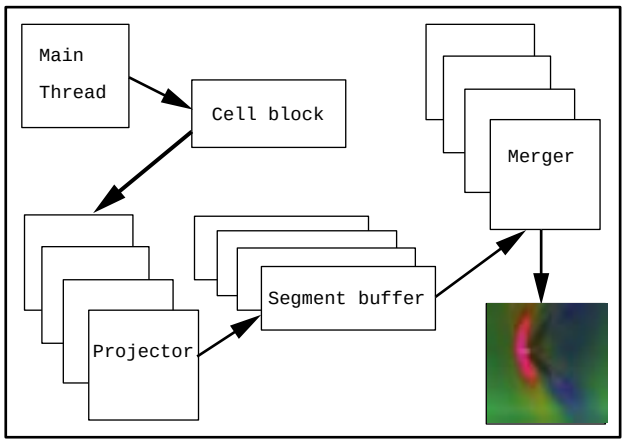

Figure 3: A threaded rendering process.

3 Multi-threaded Rendering

Our multi-threaded rendering algorithm works as follows. The renderer starts with a thread which is responsible for reading data from disk, preprocessing the data, creating other threads for doing the actual rendering calculations, and delivering the images. We call this starting thread the main thread. There are two types of rendering threads: projectors and mergers. Each projector thread retrieves a subset of the data cells prepared by the main thread, scan-converts the cells, stores the resulting ray segments to a buffer.

Each merger thread is responsible for an image area, collecting ray segments contributing to that area, sorting, and if possible merging the segments to derive the corresponding pixel value. Rendering of a frame finishes when all cells and segments have been processed. The main thread then updates viewing parameters, and is ready to render the next frame. Figure 3 gives a pictorial description of this rendering algorithm.

3.1 Multi-threading on O2K

A thread is a light weight process and has its own scheduler. All threads in a process share the state of that process. Proper coordination between threads is needed such that one thread does not accidentally change data that another thread is working on. Multi-threaded programming is therefore mainly about careful synchronization and scheduling of the threads. We has chosen Pthreads for portability.

On a multiprocessor computer like the O2K, multiple threads can run in parallel on different processors to perform different jobs. Highest efficiency is achieved by overlapping I/O with computation, and by having multiple threads work on different pieces of data concurrently.

Generally, using Pthreads on a distributed shared-memory computer like the O2K, how data are distributed among processors is not known. Explicit control of memory and threads allocations, while not impossible, is not a portable capabil-

ity. Sondak and Perry [15] examines the performance of a matrix transpose code on the O2K using different memory placement strategies. They show that the default thread and memory placement always results in poor performance. In our study, we have also verified that controlled memory placement does improve the rendering efficiency.

3.2 Cell-Projection Rendering

We adopt the cell projection method because of its flexibility. Without losing generality, in this study we only consider tetrahedral cells. Each cell thus has four faces and 4 vertices. We also assume that the data value in a cell varies linearly. The cell-projection process scan-converts each of four faces to an accumulation buffer, and each pair of projected points form a segment. While each cell can be projected independent of other cells, merging all the resulting segments requires a sorting of the segments in depth order. By using Porter-Duff's *over* operator, merging (i.e. compositing) segments corresponds to the same pixel is equivalent to tracing a ray through the collection of data cells. Note that the four faces can be projected in any order, and the projection is always consistent.

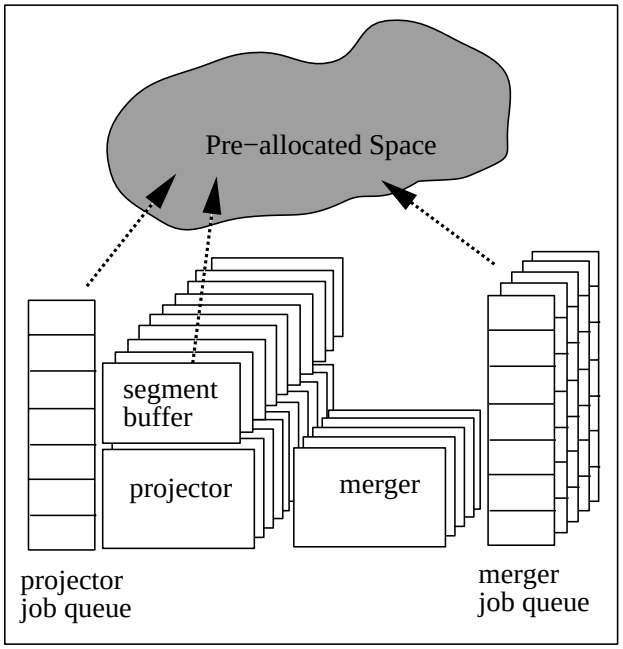

Figure 4: Memory management is based on a pre-allocated space, and job queues.

3.3 Data Structures

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 runtime memory requirements are on the order of 10^7 to 10^9 bytes per frame. Clearly, efficient management of memory is essential to the viability of our approach. Before we describe the memory management method in Section 3.4, we first introduces the basic data structures the renderer uses.

The data must be organized in a way that the projector threads avoid competing for memory. The most important data structures used by our algorithm include:

1. space partitioning tree
2. cell blocks
2. projector job queue
3. segment buffer
4. merger job queue

Before rendering begins, a preprocessing step performs partitions and produces a hierarchical representation of the data space using a k-d tree [1]. We use a k-d tree because of its ability to adapt to the structure of the data. The resulting k-d tree is then used to guide the rendering step such that ray segments can be merged as soon as they are generated, and the more ray segments can be merged early the lower runtime memory requirements. We have chosen object-space partitioning because it allows us to do view-independent optimization. As pointed out by Palmer, Totty and Taylor [13], the image-space partitioning can produces cache thrashing due to the lack of locality in accessing the volume data.

Each leaf of the space partitioning tree is a cell block which contain a small number of cells in the same neighborhood. The cell block is a job distribution unit. We make each cell block self-contained to facilitate shared-memory processing but overall memory requirements increase slightly because a small portion of data are duplicated. This storage overhead depends on the depth of the k-d tree. In practice, for a six-, seven- or eight-level tree, the storage overhead is about 0.3%, 1.6%, or 3.5%, respectively for our test data set. This storage overhead is rather small and therefore should not be a concern.

The raw data set contains an array of voxels and an array of cells. Each cell consists of four ids to the voxel array. Each element in the voxel array contains the *x, y,* and *z* coordinates and data value(s) of the voxel. The cell block packs the data by combining the original cell and voxel arrays. The resulting savings compensates for the additional storage space required by grouping cells into blocks.

Before projectors begin to work on the cell blocks, a traversal of k-d tree according to the starting view position orders the cell blocks and places them into a projector job queue. Then each projector retrieves cell blocks from the top of the queue. When the bottom of the queue is reached, the projectors' job is done. The use of a job queue simplifies

load balancing.

To scan-convert cells, each projector creates a segment buffer for each merger. A segment buffer is simply an array of segments produced by the projector for a specific image area. When a segment buffer is full, the projector place the content of the buffer to a merger job queue. Each merger retrieves ray segment periodically from this queue. While the size of the seqment buffer is presently determined experimentally, we believe it can be adjusted according to runtime rendering performance.

3.4 Memory Mangement

To efficiently maintain the buffers and queue used by the renderer, good memory management is probably the most crucial task. We must avoid frequent memory allocations and deallocations, and must also improve locality. We have developed a dynamic mechanism which we call *memory-on-demand* to conserve memory and to improve memory locality. This management method pre-allocates a memory space and trys to reuse this space as much as possible.

The projector job queue is created by using the pre-allocated space. Rather than re-creating the queue for every frame, we found it is much more efficient to reuse the same job queue. As a result, the k-d tree is traversed only once, and the order of the cell blocks in the queue is fixed. Rendering of consecutive frames starts from different ends of the queue. This approach not only removes the need to re-insert the cell blocks into the queue but also exploits data locality.

Each projector also obtains free space from the pre-allocated space to set up a segment buffer for each merger. When this buffer becomes full, the projector places the pointer to this buffer to the corresponding merger's job queue. The projector then obtains another buffer space and continues working on other cell blocks. The merger is busy as long as its job queue is not empty. When the content of a segment buffer in a merger job queue is processed, the merger returns the buffer space to the free pool of spaces so it can be reused. The use of a pre-allocated, shared space removes the need to constantly allocate and de-allocate memory space. Figure 4 illustrates such an arrangement.

3.5 Synchronization and Termination

The key to efficient multi-threading is to avoid using global variables. We cannot completely avoid using global variables due to the needed coordination between threads. As soon as the projectors and mergers are created, the main thread blocks and waits until a frame is finished. Each projector thread proceeds or blocks according to the content of the projector job queue and its local segment buffer. The same segment buffer is also used to coordinate between each projector and each merger. All threads are synchronized to start the rendering of the next frame, and are notified by the main thread to terminate.

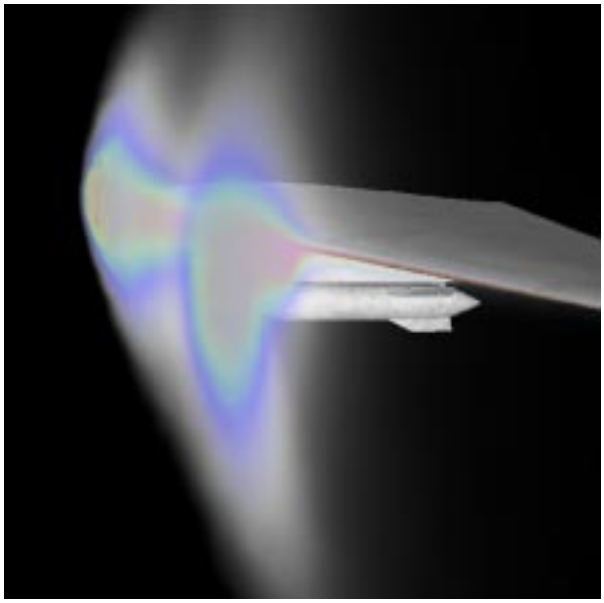

Figure 5: A close-up view of the wing and its attachment.

3.6 Experimental Results

For performance study, we used the same data set described in Section 2.3 which contains about half a million tetrahedral cells. The data set was obtained from simulation of flow over an aircraft wing with an attachment.

All test results are based on the average time of rendering using ten different viewing directions. Two starting view positions were used. One is a close-up view of the wing as shown in Figure 5, and the other gives a view of the overall domain as in Figure 6.

Figure 7 shows ideal speedup versus the measured speedup numbers using one merger and up to 23 projectors for the close-up view. Because of the use of merger(s), the measured speedup numbers are calculated relatively to the smallest configuration consisting of one merger and one projector. Comparing with a true sequential version would be unfair and result in superlinear performance. However, with this setting, using between four and 14 projectors, we still obtain superlinear speedup numbers. This is due to good cache coherence, a result of using our memory-on-demand mechanism which exploits the memory locality. The scalability decreases as we use more projectors. We thought it was because a single merger cannot keep up with all the projectors. Our further test results show that adding another merger doesn't help improve performance much. In fact, as shown in Figure 8 when using a small number of processors (e.g. 1-16) the overall rendering performance seems invariant of the number of mergers used. We observe similar performance numbers for the full-volume view.

Figure 9 displays the impact of segment buffer depth on performance using twenty projectors and one merger. The buffer size varies from a minimum of 256 up to 4096. Nor-

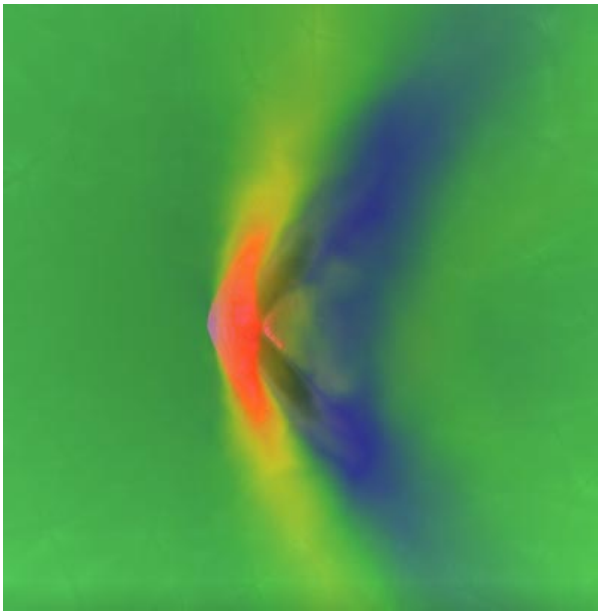

Figure 6: Full view of the volume.

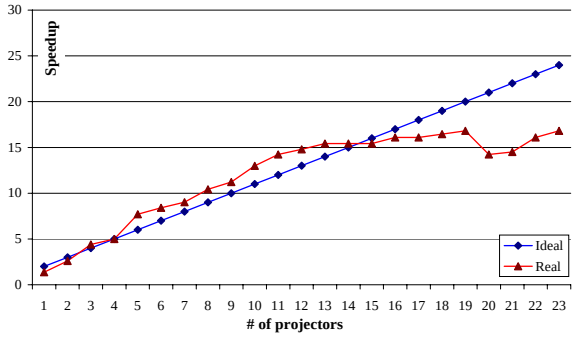

Figure 7: Speedup numbers when using one merger and up to 23 projectors; image size=400×400 pixels; segment buffer size=4096; k-d tree level=7; merger queue size=1024.

mally one expects that buffer size increases, performance will improve, since less synchronization is needed between threads. However, using a buffer size too large can eliminate the benefits as mergers becomes idle and runtime memory requirement increases.

We have also studied how the object-space partitioning would impact the rendering performance by experimenting with partitioning levels (tree depth) from four to 14. As shown in Figure 10, a 7- or 8-level partitioning, which is equivalent to about 4,000 cells per partition, gives us the best result.

Finally, Figure 11 shows parallel efficiency achieved with four mergers. While we can improve the rendering performance for the 24-processor case (24 projectors and four mergers), we cannot control performance consistently because the rendering performance is sensitive to not only the particular system configuration we use but also other jobs on

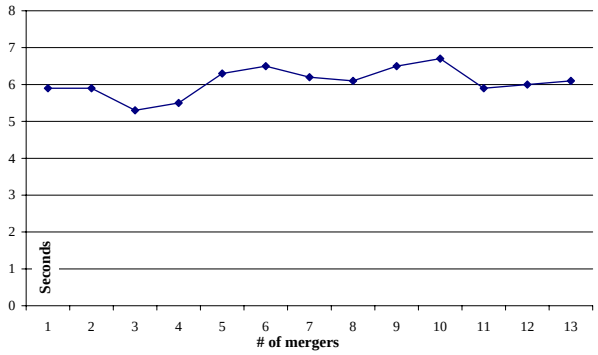

Figure 8: Rendering time using 1-13 mergers; image size=400×400 pixels; segment buffer size=1024; k-d tree level=7; merger queue size=1024.

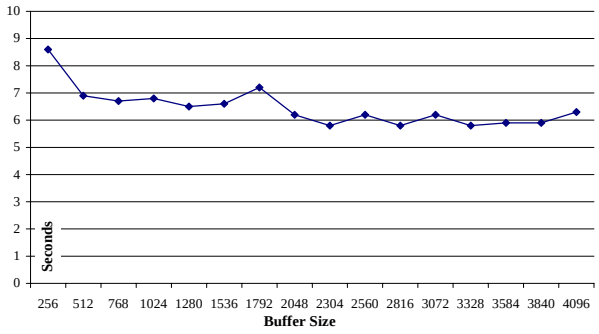

Figure 9: Rendering time using different buffer sizes from 256 to 4096 segments; image size= 400×x400 pixels; 10 projectors and one merger; k-d tree level=7, merger queue size=1024

the same O2K.

The best rendering rates we can achieve is 3.7 seconds for an image size of 400×400 pixels using 16 processors, and this number is quite comparable to the performance of the message-passing code. Our further tests using up to 120 processors show that , the rendering performance of using more than 24 processors deteriorates very quickly. This poor scalability we have observed on O2K for both codes provides a clear indication of the unsuitability of object-space parallelism for distributed shared-memory architecture.

3.7 Conclusions

Our current implementation allows us to achieve maximum performance on our 16-processor O2K. The rendering rate is about 150-200k tetrahedral cells per second, and parallel efficiency is over 95%. Without the memory management implemented, the renderer would become orders of magnitude slower, and could only solve a much smaller problem.

Unfortunately, the rendering performance deteriorates quickly when more processors are used which prevents us from utilizing the much larger O2Ks operated at other super-

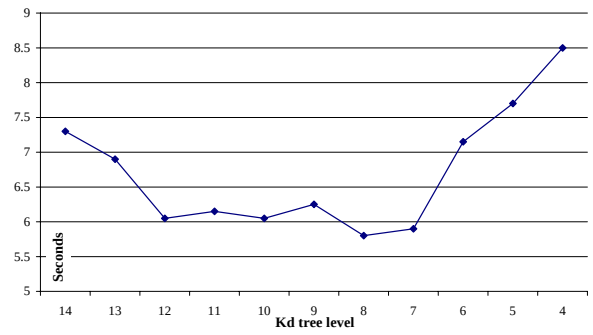

Figure 10: Rendering time using different levels of partitioning, from four to 14; image size= 400×x400 pixels; 10 projectors and one merger; buffer size=4096, merger queue size=1024

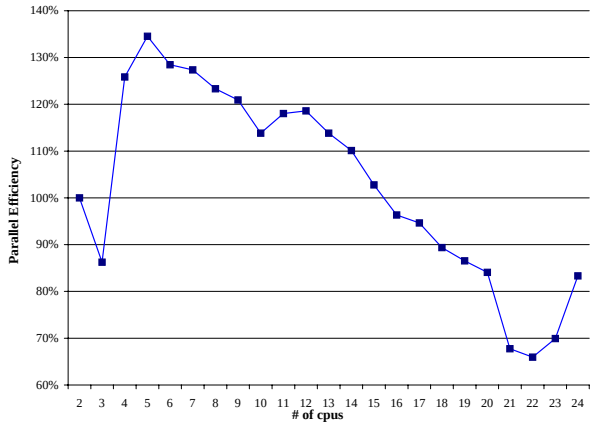

Figure 11: Parallel efficiency with four mergers; buffer size=4096; k-d tree level=7; merger queue size=1024

computing facilities to either achieve interactive rendering rates or visualize larger data sets. The scope of our study so far is thus limited.

We believe highly scalable rendering of large unstructured-grid volume data on the O2K can be realized. To achieve our goal, we plan to:

- investigate a hybrid algorithm based mainly on image-space parallelism,
- re-evaluate our buffering scheme,
- carefully exploit O2K's memory hierarchy, and
- use a mix of message-passing and multi-threaded programming for large processor sizes.

The programming effort could be as great as using MPI for a pure message-passing approach.

Finally, the choice of parallelization and rendering parameters can also have a significant impact on performance but the situation here is quite complex due to the interaction between these parameters. Thus guidelines for selecting optimal communication parameters are far from obvious, and a detailed analysis on the O2K as well as a different distributed shared-memory architecture is also the subject of ongoing investigation.

Acknowledgement

This work has been sponsored by the National Science Foundation under contract ACI 9983641 (CAREER Award), DOE ASCI program through the Los Alamos National Laboratory, and the Numerical Aerospace Simulation (NAS) Systems Division of the NASA Ames Research Centers. C. Hofsetz's study has also been sponsored by UNISINOS - Universidade do Vale do Rio dos Sinos, and CAPES - Fundação Coordenação de Aperfeiçoamento de Pessoal de Nível Superior. Steven Parker helped conduct several tests using his thread library on the O2K at ACL of the Los Alamos National Laboratory. The authors would also like to thank him for some very valuable discussion.

References

[1] BENTLEY, J. Multidimensional binary search trees used for associative searching. *Commun. ACM 18*, 8 (September 1975), 509–517.

[2] CAVIN, X. Load balancing analysis of a parallel hierarchical algorithm on origin2000. In *Fifth European SGI/Cray MPP Workshop* (1999). Bologna, Italy, September 9-10.

[3] 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.

[4] CROCKETT, T. Portability and cross-platform performance of an mpi-based parallel polygon renderer. In *in Proceedings of HPCCP/CAS Workshop '98* (1998), C. Schulbach and e. E. Mata, Eds., pp. 251–256. NASA CP-1999-208757, NASA Ames Research Center.

[5] LACROUTE, P. Real-Time Volume Rendering on Shared Memory Multiprocessors Using the Shear-Warp FActorization. In *Proceedings of the 1995 Parallel Rendering Symposium* (October 1995), ACM SIGGRAPH, pp. 15–22.

[6] LUECKE, G. R., RAFFIN, B., AND COYLE, J. J. Comparing the scalability of the cray t3e-600 and the cray orgin 2000 using shmem routines. *The Journal of Performance Evaluation and Modeling for Computer Systems (PEMCS)* (1998). (http://hpc-journals.ecs.soton.ac.uk/PEMCS/).

[7] MA, K.-L. Parallel Volume Ray-Casting For Unstructured-Grid Data on Distributed-Memory Architectures. In *Proceedings of the 1995 Parallel Rendering Symposium* (October 1995), ACM SIGGRAPH, pp. 23–30.

[8] MA, K.-L., AND CROCKETT, T. W. A Scalable Parallel Cell-Projection Volume Rendering Algorithm for Three-Dimensional Unstructured Data. In *Proceedings of the 1997 Parallel Rendering Symposium* (October 1997), ACM SIGGRAPH, pp. 95–104.

[9] MA, K.-L., AND CROCKETT, T. W. Parallel Visualization of Large-Scale Aerodynamic Calculations: A Case Study on T3E. In *Proceedings of the 1999 Parallel Visualization and Graphics* (October 1999), ACM SIGGRAPH, pp. 15–20.

[10] MA, K.-L., AND INTERRANTE, V. Extracting Feature Lines from 3D Unstructured Grids. In *Proceeding of Visualization '97 Conference (to appear)* (October 1997).

[11] NIEH, J., AND LEVOY, M. Volume rendering on scalable shared-memory mimd architectures. In *1992 Workshop on Volume Visualization* (1992), pp. 17–24. Boston, October 19-20.

[12] PALMER, M., AND TAYLOR, S. Rotation Invariant Partitioning for Concurrent Scientific Visualization. In *Proceedings of the Parallel Computational Fluid Dynamics '94 Conference* (1994), Elsevier Science Publishers B.V.

[13] PALMER, M. E., TOTTY, B., AND TAYLOR, S. Ray casting on shared-memory architectures. *IEEE Concurrency* (January-March 1998), 20–35.

[14] PARKER, S., PARKER, M., LIVNAT, Y., SLOAN, P., AND HANSEN, C. Interactive Ray Tracing for Volume Visualization. *IEEE Transactions on Visualization and Computer Graphics 5*, 3 (July-September 1999), 1–13.

[15] SONDAK, D. L. A Study of Memory Placement on an SGI Origin2000. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing* (1999).

[16] USELTON, S. Volume rendering on curvilinear grids for cfd. AIAA Paper 94-0322, 1994. 32nd Aerospace Sciences Meeting & Exhibit.

[17] WILHELMS, J., VAN GELDER, A., TARANTINO, P., AND GIBBS, J. Hierarchical and Parallelizable Diret Volume Rendering. In *Proceedings of the Visualization '96 Conference* (October 1996), pp. 57–64.

[18] WILLIAMS, P. L. Parallel volume rendering finite element data. In *Proceedings Computer Graphics International '93* (1993). Lausanne, Switzerland, June.