Volume Rendering on Scalable Shared-Memory MIMD Architectures

Jason Nieh and Marc Levoy
Computer Systems Laboratory
Stanford University

Abstract

Volume rendering is a useful visualization technique for understanding the large amounts of data generated in a variety of scientific disciplines. Routine use of this technique is currently limited by its computational expense. We have designed a parallel volume rendering algorithm for MIMD architectures based on ray tracing and a novel task queue image partitioning technique. The combination of ray tracing and MIMD architectures allows us to employ algorithmic optimizations such as hierarchical opacity enumeration, early ray termination, and adaptive image sampling. The use of task queue image partitioning makes these optimizations efficient in a parallel framework. We have implemented our algorithm on the Stanford DASH Multiprocessor, a scalable shared-memory MIMD machine. Its single address-space and coherent caches provide programming ease and good performance for our algorithm. With only a few days of programming effort, we have obtained nearly linear speedups and near real-time frame update rates on a 48 processor machine. Since DASH is constructed from Silicon Graphics multiprocessors, our code runs on any Silicon Graphics workstation without modification.

1 Introduction

Volume visualization techniques are becoming of key importance in the analysis and understanding of multidimensional sampled data. The usefulness of volume rendering for visualizing such data has been demonstrated [6, 12], but the computational expense of this technique limits its routine and interactive use. In most volume rendering algorithms, resampling and compositing of the voxel array to form image space samples constitutes the single greatest computational expense.

A key advantage of ray tracing [12] over other volume resampling techniques [6, 8, 19] is that algorithmic optimizations have been developed which significantly reduce its image generation time. Examples of these optimizations are hierarchical opacity enumeration, early ray termination, and spatially adaptive image sampling [13, 14, 16, 18]. Optimized ray tracing is the fastest known sequential algorithm for volume rendering. Nevertheless, the computational requirements of the fastest sequential algorithm still preclude real-time volume rendering on the fastest computers today.

Parallel machines offer the computational power to achieve real-time volume rendering on usefully large datasets. This paper considers how parallel computing can be brought to bear in reducing the computation time of volume rendering. We have designed and implemented an optimized volume ray tracing algorithm that employs a novel task queue image partitioning technique on the Stanford DASH Multiprocessor [10], a scalable shared-memory MIMD (Multiple Instruction, Multiple Data) machine consisting of up to 64 (currently 48) high-performance RISC microprocessors. Shared-memory architectures permit straightforward implementations of our algorithm. The optimized ray tracer required only a few days to parallelize on DASH, as compared to several weeks to implement a similar algorithm using message-passing on the Intel iPSC/860 [4]. Performance results demonstrate that our parallel ray tracer achieves good speedups despite requiring communication of 3D voxel values. When the 64 processor version of DASH is available in a few months, we expect to obtain frame update rates of 4 frames/sec on 256^3 voxel datasets and 15 frames/sec on 128^3 voxel datasets.

The paper is organized as follows. Section 2 presents the development of our parallel volume rendering algorithm for MIMD machines. Section 3 describes the architecture of DASH and discusses implementation issues for the algorithm on the target machine. Section 4 discusses the performance of the algorithm on DASH. Finally, we present conclusions and possibilities for future work.

2 A Parallel Rendering Algorithm

Optimized Ray Tracing. Our parallel volume rendering algorithm is based on ray tracing as described in [12]. Rays are cast from the viewing position through the volume data. The data is resampled at evenly spaced locations along each ray by trilinearly interpolating values of surrounding voxels. Finally, ray samples are composited to produce an image. The algorithms used for hier-

Author's addresses: Jason Nieh, Stanford University, ERL 411, Stanford, CA 94305, e-mail nieh@leland.stanford.edu; Marc Levoy, Stanford University, CIS 207, Stanford, CA 94305, email levoy@cs.stanford.edu.

archical opacity enumeration, early ray termination, and adaptive image sampling are from [13, 14]. Hierarchical opacity enumeration is performed by using a binary octree to avoid unnecessary resampling in transparent regions of the volume. Early ray termination is performed by terminating resampling along a ray when its accumulated opacity exceeds a user-selected threshold of opaqueness. Adaptive image sampling is performed by dividing the image plane into square sample regions measuring ω_{max} pixels on a side and casting rays only from the four corner pixel of each region. Additional rays are cast only in those sample regions with high image complexity, as measured by the color difference of the corner pixels of the region. Any untraced pixels are then bilinearly interpolated from the traced pixels. Normals, opacities, and the octree are computed as a preprocessing step. A lookup table containing color as a function of local voxel gradient is used to accelerate shading.

Task Queue Image Partitioning. Using a MIMD architecture, rays can be efficiently traced in parallel. Strategies for partitioning the ray tracing computation among the processors generally fall into two classes: static and dynamic. There are two basic options for static partitioning: contiguous and interleaved. Contiguous partitioning divides the image plane into a few large blocks and statically assigns one block to each processor [5, 20]. Interleaved partitioning divides the image plane into many small image tiles and assigns the tiles to processors in round-robin order such that each processor computes multiple tiles from different regions of the image plane. Contiguous schemes for volume rendering suffer from poor load distribution because different sections of the image differ in image complexity, and the distribution of image complexity changes with viewing position. Interleaved partitioning distributes load better, but image tiles must be so small to achieve even modest load distribution that there will be excessive overlapping voxel accesses.

Dynamic partitioning divides the image plane into image tiles and places the tiles on a single work queue. Processors grab tiles from the work queue to ray trace until the queue is empty. Dynamic partitioning provides better load distribution than static partitioning, but the centralized point of control becomes a bottleneck as more processors are used. Dynamic as well as static interleaved partitioning also incur excessive costs associated with pixel sharing in adaptive image sampling, as discussed later this section.

To distributing the ray tracing computation among the processors, we employ a hybrid scheme that provides good load distribution and reduces overlapping voxel accesses, described as follows:

1. For P processors, the image plane is statically partitioned into P contiguous rectangular blocks of comparable size.
2. Each image block is divided into fixed size square image tiles
3. Each processor is statically assigned an image block and starts ray tracing the tiles in that block in scan line order.
4. When a processor is done computing its block, it goes to the next processor in scan line order that is not done computing and grabs tiles in that processor's image block to ray trace. Processors compute until all the tiles are completed.

Figure 1 illustrates a four processor example of how the image plane is partitioned.

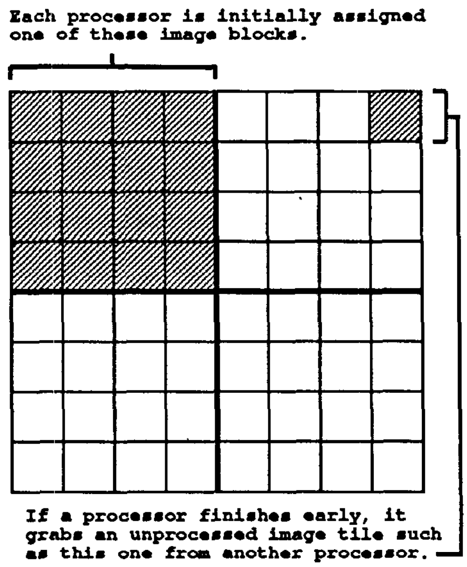
Each processor is initially assigned one of these image blocks.
If a processor finishes early, it grabs an unprocessed image tile such as this one from another processor.

Figure 1: Task queue image partitioning example.

The motivation for using image tiles as the unit of parallelism instead of scan lines [3, 5, 20] and for assigning spatially adjacent tiles to the same processor is to minimize overlapping voxel accesses and to reduce the costs associated with pixel sharing in adaptive image sampling. The tiles, initially assigned to one processor but available for grabbing by other processors, can be thought of as a queue of tasks. Because it is dynamic, such a task queue image partitioning scheme provides better load distribution than static partitioning schemes, as demonstrated in Section 4. In addition, there are as many task queues as there are processors and the management of the queues is distributed, thereby providing better scalability by avoiding the bottleneck of a centralized point of control in typical dynamic schemes.

Pixel Sharing in Adaptive Image Sampling. While there are no computational dependencies between image tiles in nonadaptive ray tracing, there are such dependencies with adaptive image sampling. This pixel sharing arises because adjacent image sample regions share the corner pixel values required for measuring their image complexity. Note that sample regions are equal to or smaller than image tiles. The dependencies require replicated computation of rays or synchronized communication of the shared pixels. Our adaptive image sampling method avoids the cost of replicated ray tracing by communicating pixel values. Since task queue partitioning results in adjacent image tiles usually being ray traced by the same processor, the synchronization overhead is small. In the small percentage of cases where adjacent image tiles are ray traced by different processors, we delay the evaluation of sample regions whose pixel values are being computed by other processors as follows:

1. Given an image sample region, ray trace those of the four corner pixels which have not already been ray traced during processing of adjacent sample regions or are not currently being ray traced by other processors as part of processing an adjacent region.
2. Measure the image complexity of the sample region by computing the color difference of those of the corner pixels that have been ray traced. Pixels currently being ray traced by other processors are simply not included in the computation.
3. If the color difference is larger than a user-selected color threshold, subdivide the sample region and recursively ray trace the subregions.
4. Otherwise, if some of the corner pixels are currently being ray traced by other processors, store the state of the sample region in a local wait queue. If not, simply proceed to the next region.
5. After all task queues have been emptied of image tiles, ray trace any sample regions stored in the local wait queue.
6. For each sample region in the local wait queue, perform steps two through four, but this time instead of storing the state of a sample region when some of its corner pixels are being traced by other processors, busy wait until all are ray traced.

This method reduces the cost of waiting for pixel values to be traced on other processors in two ways. First, it tries to make a subdivision decision based on available information. Failing that, it switches to a sample region on which useful work can be done.

Data Distribution. A potential disadvantage of our parallel ray tracing algorithm is that it requires the communication of 3D voxel values. Since high-performance MIMD machines are designed to be configured with large numbers of processors, physical memory is necessarily distributed among the processors[1]. Thus, parallelizing volume rendering for MIMD architectures requires not only partitioning the computation among the processors, but also partitioning the data among the local memories and communicating the data among the processors. When a given processor resamples a region of the volume and finds that a voxel it needs for the computation is not in its local memory, communication with other processing nodes is required. Given that voxels are read-only, full data replication has been proposed to eliminate the voxel communication associated with ray tracing [20]. But this option makes poor utilization of memory resources and does not scale with dataset size, making it too expensive for most cases.

Our parallel ray tracing algorithm distributes data in an interleaved fashion among the local memories to avoid hot spotting. This scheme employs the usual linearized array structure for voxels, and distributes pages of these arrays in round-robin fashion among the local memories.

3 Implementation on DASH

The DASH Architecture. We have implemented the parallel volume rendering algorithm on the DASH (Directory Architecture for SHared memory) Multiprocessor, a scalable shared-memory MIMD machine under construction in the Computer Systems Laboratory at Stanford University. DASH is designed to be a general-purpose machine capable of supporting a wide variety of applications.

The architecture consists of a set of processor clusters connected by a scalable interconnection network. A four cluster diagram of the system is shown in Figure 2. Each cluster consists of a

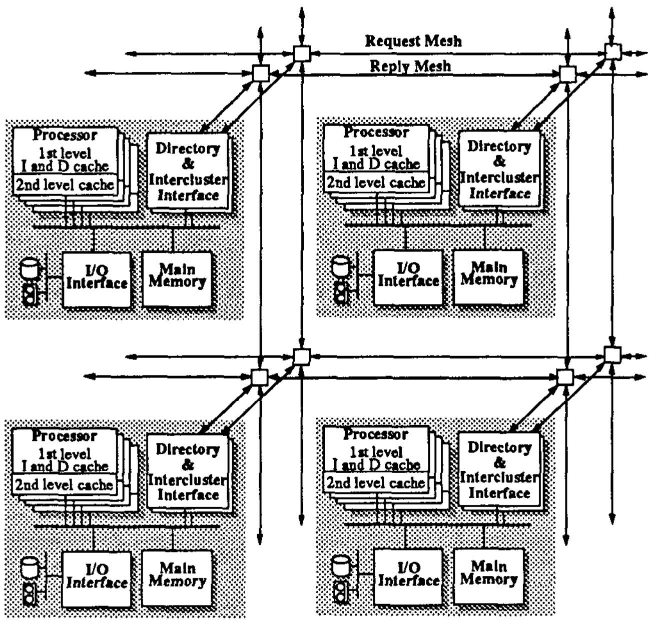

Figure 2: Block diagram of a 2 x 2 DASH system.

small number of high-performance processors and a portion of the machine's memory interconnected by a bus. Although the memory of the machine is partitioned and distributed among the clusters for scalability, DASH provides a single address space, enhancing programmability by alleviating the user from much of the burden associated with data partitioning in a parallel machine. To sustain high-performance, DASH provides caching of memory, including shared writable data, to reduce memory latency. By associating a directory with each cluster's main memory that keeps track of all memory blocks cached in other clusters, a distributed directory-based protocol [9] with an intracluster snoopy bus-based protocol [15] provides coherent caches, keeping memory consistent among the processors.

The latency of a memory access depends on where it is serviced in the memory system of DASH. The memory system can be broken into four levels of hierarchy: processor, local cluster,

[1]Note that "distributed memory" refers to having memory physically distributed. This is not mutually exclusive with "shared-memory", which refers to having a single address space. This terminology is admittedly confusing, but it is too late to change standard usage.

home cluster, and remote cluster. At the processor level of the hierarchy is a small fast cache designed to match the processor speed. When a processor requests a given piece of data, the request first goes to its own cache. If the data is not in the cache, the request goes to the local cluster level. If the data is cached in one of the processor caches in the cluster, the request can be serviced at the local cluster level. Otherwise, the request must be sent to the home cluster which contains the directory and physical memory of the given memory address. Note that for some accesses, the home cluster and local cluster are the same, but often they are not and the request is communicated over the network. The home cluster can usually service the request immediately, unless the directory indicates that the data in memory is not updated or the request is for exclusive access to data that is cached in other clusters. In these cases, the request goes to the remote cluster level, consisting of all processor caches other than those in the local or home cluster. If the data in memory is not updated, the request is forwarded to the remote cluster with the updated copy of data to be serviced. If the request is for exclusive access, remote clusters caching the data invalidate their copies of the data. Since intercluster memory accesses can have latencies of over a hundred processor cycles, good performance requires that the vast majority of memory requests be serviced within clusters.

The current DASH prototype consists of twelve processor clusters, with each cluster having four processing nodes for a total of 48 nodes. Each cluster is a modified Silicon Graphics Power Station 4D/340 [1], which consists of four 33 MHz MIPS R3000 processors with R3010 floating-point coprocessors. The R3000/R3010 CPU is rated at 25 VAX MIPS and 10 MFLOPS. Each of the four CPUs contains a 64 Kbyte first-level instruction cache, a 64 Kbyte first-level data cache, and a 256 Kbyte second-level data cache. Caches are direct-mapped with line size of 16 bytes. The single address space in DASH is distributed among the clusters, each cluster having 16 Mbytes of main memory, of which 14 Mbytes are available to the user. CPUs within clusters are connected by a 64 Mbytes/sec bus, while clusters are connected by a 120 Mbytes/sec 2D mesh network.

Programming DASH for Volume Rendering. DASH's architectural support for shared-memory makes the implementation of the volume rendering algorithm easy. Our implementation was done in C with Argonne National Laboratory (ANL) parallel macros [2] for shared-memory programming primitives. Processors share access to all large data structures, such as the voxel array, shading table, octree, and image array. The data storage requirements of our implementation are roughly four bytes per voxel: one byte for original data samples, one byte for opacities, one bit for octree values, and about two bytes for normals. Normals are stored as 13-bit quantities: 6 bits each for Y and Z normal components, 1 bit for the X component. The normals are used to index a 2^{13} entry lookup table for efficient shading [20]. Since DASH is constructed from Silicon Graphics multiprocessors, our code runs on any Silicon Graphics workstation without modification.

On DASH, the operating system can take care of the assignment of memory pages to cluster memories or the user can specify the initial assignment explicitly. We use the interleaved mapping mentioned in Section 2 to explicitly distribute the pages of voxels and other shared data to the cluster memories. The pages are allocated only from clusters that are actively being used, as long as there is enough physical memory in those clusters. As for the necessary data communication in our distributed data implementation, the DASH hardware manages the movement of data automatically without requiring the programmer to write explicit communication calls. The programmer does not need to keep track of where all the data is in the memory system. Since all voxel data is read-only, the system can always cache multiple copies of voxel data without invalidating voxel data present in one processor's cache because another processor wants to modify the voxel data. The system will therefore automatically partially replicate the data in the caches to reduce intercluster memory accesses, thereby making effective utilization of memory and network resources.

4 Performance Results and Analysis

Experimental Setup. To demonstrate the performance of our parallel ray tracer on DASH, we present results from case studies in medical imaging and molecular graphics. The results from medical imaging are based on rendering bony tissues of CT (computer tomography) datasets of a human head. The datasets are 256 x 256 x 226 voxels and 128 x 128 x 113 voxels. The calculation of opacities from input values uses the region boundary surface method of [12]. The results from molecular graphics are based on rendering concentric semi-transparent surfaces of a 300 x 300 x 91 voxel portion of an electron density map of staphylococcus aureus ribonuclease. The calculation of opacities from input values for this dataset uses the isovalue contour surface method of [12].

Results for nonadaptive and adaptive renderings are presented, as they represent differing degrees of image quality. These results are based on measurements of rendering time and do not include image I/O time or the time associated with fixed initialization costs such as loading in the dataset. All renderings used an opacity value of 0.95 as the threshold level for early ray termination. When performing adaptive image sampling, an initial grid spacing of $\omega_{max} = 4$ (defined in Section 2) was used along with a minimum color difference of 16. An 8 x 8 pixel image tile size was chosen for task queue partitioning to provide good load distribution and minimize task queue synchronization.

Rendering Times and Speedups. Figures 3 and 4 show a non-adaptively and adaptively rendered visualization of the 256 x 256 x 226 voxel CT dataset of the human head rendered on DASH. Figure 5 shows an adaptively rendered visualization of the 300 x 300 x 91 voxel electron density map of the ribonuclease. Rendering times on DASH for these images, as well as those for other case studies are given in Table 1.

We calculated speedups on DASH relative to the performance of a sequential version of our ray tracer on a single processor of an SGI 4D/320 VGX workstation. Our SGI 4D/320 platform has the same CPUs and cache sizes as DASH, but contains 64 Mbytes

dataset name	head		headsmall		ribonuclease	
dataset size (voxels)	256x256x226		128x128x113		300x300x91	
image size (pixels)	416x416		209x209		432x432	
rendering quality	nonadaptive	adaptive	nonadaptive	adaptive	nonadaptive	adaptive
# of rays traced	173056	21723	43681	7844	186624	38141
# of samples trilirped	618428	245132	87683	62337	882553	456258
rendering time (ms)	700	340	120	90	850	480
speedup over uniprocessor	40	33	40	30	40	33

Table 1: Rendering times and speedups (48 processors).

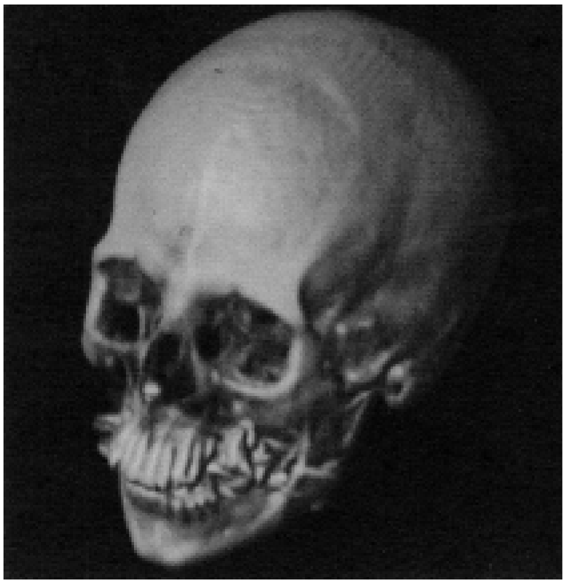
Figure 3: Nonadaptively rendered image of CT dataset.

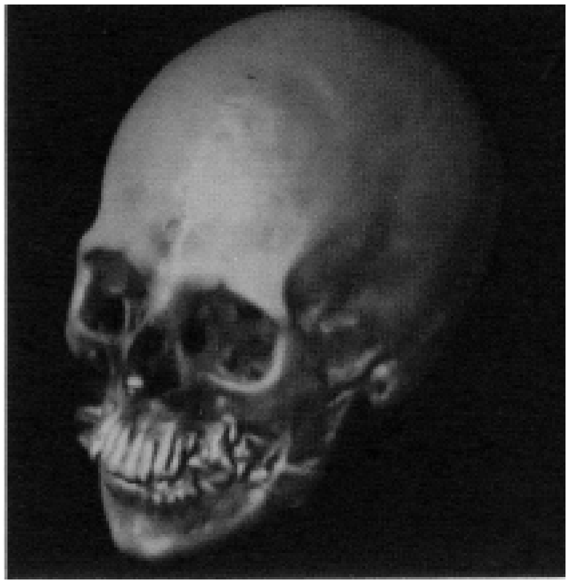
Figure 4: Adaptively rendered image of CT dataset.

of local main memory as opposed to only 14 Mbytes of usable local memory in each DASH cluster. Speedups are calculated with respect to the SGI workstation instead of a single processor on DASH to account for any intercluster memory accesses that may occur for the larger datasets due to their voxel storage requirements exceeding the amount of memory in a single DASH cluster[2]. Speedup curves are plotted for nonadaptive and adaptive renderings of the 256 x 256 x 226 voxel CT dataset in Figure 6. Other speedup results are similar, as shown in Table 1. Nearly linear speedups were also measured for our ray tracer even without using an octree, demonstrating that the algorithm has good performance even if effective use cannot be made of an octree.

Load Distribution. We ascribe the nearly linear speedup of our renderer partly to good load distribution. The distribution was measured by timing the rendering computation on each processor. For nonadaptive and adaptive renderings of the 256 x 256 x 226 voxel CT dataset on 48 processors of DASH, the variation in rendering time among processors was measured to be less than 4 percent for our algorithm. For comparison we also implemented a static interleaved image partitioning algorithm on DASH. Comparison results for nonadaptive rendering are presented in Table 2, which lists the range and variation of execution times across the 48 processors for each partitioning method. Results for static interleaved partitioning are given for the image tile sizes that resulted in the best rendering time for that method. The variation in execution time among the processors is only 20 ms for task queue partitioning, as compared to more than five times that amount for static interleaved partitioning.

Memory and Synchronization Overhead. We also ascribe the nearly linear speedup of our renderer to low memory and synchronization overhead, as measured using MTOOL [7], a performance debugging tool. Table 3 presents results characterizing the behavior of the renderer on a single processor of the SGI workstation and on 48 processors of DASH. These results are based on nonadaptive and adaptive renderings of the 256 x 256 x 226 voxel

[2]The SGI workstation actually has slightly better uniprocessor rendering times than DASH even when there are no intercluster accesses due to cluster memory limits, as is the case for rendering the 128 x 128 x 113 voxel dataset. This is because the SGI machine fetches 64 byte cache lines instead of 16 byte lines as on DASH.

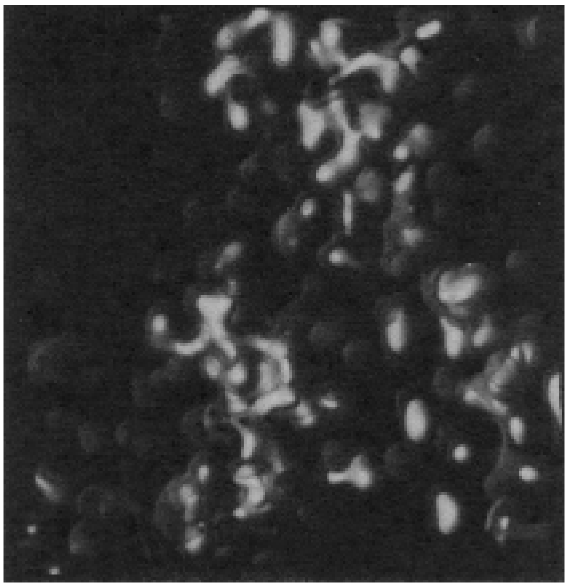

Figure 5: Adaptively rendered image of electron density map.

partitioning used	task queue	static interleaved	
image tile size	8x8	8x8	4x4
min-max time	680-700 ms	570-780 ms	690-800 ms
variation in time	20 ms	210 ms	110 ms

Table 2: Load distribution (48 processors).

CT dataset.

As shown in Table 3, memory overhead is not the dominant percentage of execution time. Although the increase in memory stall time over the SGI uniprocessor case is 84 percent for nonadaptively rendering on 48 processors of DASH, only 20 percent of the uniprocessor execution time is due to memory overhead, and only 30 percent of the 48 processor execution time is due to memory overhead. Memory overhead is larger for adaptive rendering because of accesses to additional shared writable data structures not needed in nonadaptive rendering, and because of reduced locality of voxel accesses due to the sparsity of rays cast. The large increase in memory stall time does not have a significant effect on the percentage of execution time due to memory overhead because of good cache performance and a higher than expected ratio of computations to voxel accesses, as verified by disassembling the ray tracing kernel code. Just trilinearly interpolating a ray sample requires the execution of roughly 320 assembly code instructions, of which only 16 instructions (8 for normals, 8 for opacities) are voxel accesses. This result is somewhat surprising, and suggests that counting the number of arithmetic operations in the filtering and compositing expressions themselves is not representative of the true cost of volume rendering; address calculations and loads and stores of non-voxel data in fact dominate.

The caches on DASH are important to the performance of our algorithm because they reduce the frequency of expensive intercluster accesses when referenced data is not stored in the memory

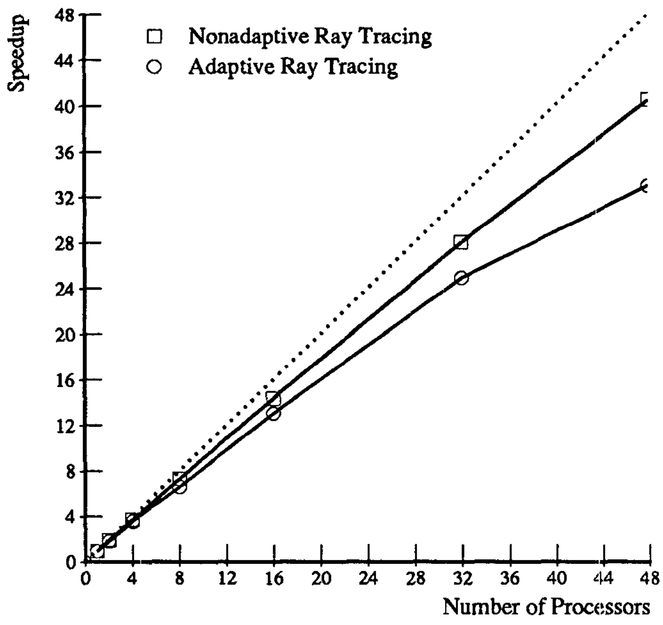

Figure 6: Speedups for Nonadaptive and Adaptive Rendering. (Note that adaptive exhibits worse speedup than nonadaptive rendering, but has lower rendering time. See Table 1.)

local to the processor. With data caching turned off, nonadaptive speedups are only 23 times with respect to the uniprocessor case without caching. Of course, rendering times are much slower. Rendering the 256 x 256 x 226 voxel CT dataset nonadaptively on 48 processors without caching takes 2.7 seconds, about 3.9 times longer than with caching.

Table 3 also shows that synchronization overhead is a small percentage of execution time for both nonadaptive and adaptive rendering. Since processors wait until all processors have finished rendering one frame before rendering the next one, the synchronization idle time includes the time due to variations in rendering time. Other synchronization time is due to waiting for all processors to complete shading table computations before ray tracing. For the adaptive case, additional synchronization time is due to waiting for all processors to complete ray tracing before untraced pixels are interpolated from the traced pixels.

Temporal Locality. The data caches on DASH allow our algorithm to exploit interframe temporal locality to reduce memory overhead in multiple frame rotation sequences. Since successive frames in rotation sequences only vary slightly in viewpoint, much of the voxel data brought into the cache for rendering a given frame may still be cached and reused on subsequent frames, thereby reducing cache read misses on subsequent frames. We measured the effects of interframe temporal locality for a ten frame sequence with viewpoint rotation of 3 degrees between frames. The data caches were completely flushed between frames to measure the rotation sequence without interframe effects.

While there is no mechanism to directly measure first-level

rendering quality	nonadaptive		adaptive	
rendering platform	1 proc. SGI	48 proc. DASH	1 proc. SGI	48 proc. DASH
CPU utilization	80%	67%	76%	54%
percentage of rendering time due to memory overhead	20%	30%	24%	38%
increase in memory overhead over 1 proc. SGI	0%	84%	0%	142%
percentage of rendering time due to idle synchronization	0%	3%	0%	8%

Table 3: Memory and synchronization overhead.

cache misses, we did use the hardware performance monitor on DASH [11] to measure the difference in the number of second-level cache read misses with and without interframe effects. The vast majority of these read references were for data whose home cluster was not the local cluster. We measured these read misses on remote data and found that interframe effects do not noticeably affect the number of read misses on remote data that can be serviced by another processor's cache within the local cluster. But interframe effects do noticeably affect the number of read misses on remote data that cannot be serviced at the local cluster level and must be serviced over the network. Figure 7 shows the average number of these remote read misses per cluster for nonadaptively and adaptively rendering the 256 x 256 x 226 and 128 x 128 x 113 CT datasets. The figure shows that inter-

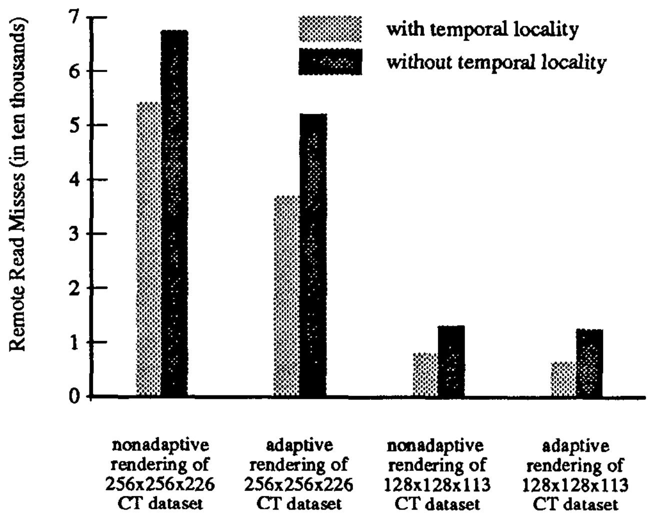

Figure 7: Temporal locality effects (48 processors).

frame effects do reduce cache misses of remote data which cannot be serviced at the local cluster level. These remotely serviced misses are reduced by as much as 50 percent due interframe effects, as for the case of adaptively rendering the 128 x 128 x 113 voxel CT dataset. Nevertheless, the difference in misses with and without interframe effects is no more than 15,000 references per four-processor cluster for any of the cases in Figure 7. Given that remote access latency is about 3 μs for reading shared data, this difference amounts to only about a 10 ms ($3\mu s * 15000$ references $/4$ processors) maximum difference in rendering time.

We also directly measured the rendering time with and without interframe effects. For any given frame of the 256 x 256 x 226 voxel CT dataset, the rendering time without interframe effects was no more than 20 ms longer than the rendering time with interframe effects. For the 128 x 128 x 113 voxel CT dataset, the difference in rendering times with and without interframe effects was less than 10 ms. Although interframe temporal locality does significantly reduce expensive intercluster voxel accesses, it has little effect on rendering performance because these accesses do not represent a dominant percentage of the rendering time.

5 Conclusions and Future Work

We have presented a parallel ray tracing algorithm that takes advantage of hierarchical opacity enumeration, early ray termination, and adaptive image sampling. Our algorithm uses a task queue image partitioning scheme to provide good load distribution and reduce overlapping voxel accesses. It was easy to implement the algorithm on the Stanford DASH Multiprocessor because of its architectural support for shared-memory. Performance results on DASH demonstrate that our algorithm achieves nearly linear speedups and fast rendering times. Although DASH is designed to be a general-purpose machine, its combination of programming ease and high performance make such scalable shared-memory MIMD machines well-suited for volume rendering. Perhaps the most important lesson here is that, given a suitable architecture, parallelization of volume rendering need not require intricate algorithms that require weeks of programming.

It would be interesting to consider the performance of our algorithm on other architectures. An implementation for message-passing machines could be done using software managed voxel caches to reduce communication, although our experience indicates that it would require significantly more time to program. An alternative large-scale multiprocessor to consider is a cache only memory architecture (COMA) [17]. Its dynamic migration and replication of data at the main memory level may prove even more ideal for volume rendering.

The analysis of memory performance presented in this paper admittedly focuses on aggregate behavior. A more thorough study would include examining the cache contents at each moment during image generation. We have not aggressively pursued this line of inquiry because, for this algorithm and architecture, memory overhead is not the dominant component of image generation cost. We are currently investigating alternative volume rendering algorithms that drastically reduce the number of arithmetic and nonarithmetic instructions executed. If successful, these algorithms may cause us to reexamine the issues of cache contents and inter-frame temporal locality in greater detail.

Acknowledgements

The authors wish to thank Prof. Monica Lam for many enlightening discussions. We thank David Nakahira, Dave Ofelt, Luis Stevens, and Jonathan Chew for DASH systems support. Thanks also to Dan Lenoski for providing the DASH figures, and Jolly Chen, Steve Tjiang, and Bob Wilson for helpful comments on drafts of this paper. This research was supported by DARPA contract N00039-91-C-0138, the National Science Foundation, (NSF), the National Aeronautics and Space Administration (NASA), and the sponsoring companies of the Stanford Center for Integrated Systems (CIS). The SGI 4D/320 VGX workstation was donated by Silicon Graphics, Inc. The CT scan was provided by North Carolina Memorial Hospital. The electron density map was provided by Dr. Chris Hill of the University of York Chemistry Department.

References

[1] Forest Baskett, Tom Jermoluk, and Doug Solomon. The 4D-MP Graphics Superworkstation: Computing + Graphics = 40 MIPS + 40 MFLOPS and 100,000 Lighted Polygons Per Second. In *Proceedings of the 33rd IEEE Computer Society International Conference - COMPCON 88*, pages 468–471, February 1988.

[2] James Boyle, Ralph Butler, Terrence Disz, Barnett Glickfeld, Ewing Lusk, Ross Overbeek, James Patterson, and Rick Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart, and Winston, Inc., 1987.

[3] Judy Challinger. Parallel Volume Rendering on a Shared-Memory Multiprocessor. Technical Report UCSC-CRL-91-23, University of California at Santa Cruz, March 1992.

[4] Intel Scientific Computers. *iPSC/2 and iPSC/860 User's Guide*, June 1990.

[5] Tim Cullip. Personal communication, May 1991.

[6] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume Rendering. Proceedings of SIGGRAPH '88 (Atlanta, Georgia, August 1-5, 1988). In *Computer Graphics*, volume 22(4), pages 65–74, August 1988.

[7] Aaron J. Goldberg and John L. Hennessy. MTOOL: An Integrated System for Performance Debugging Shared Memory Multiprocessor Applications. *To appear in IEEE Transactions on Parallel and Distributed Systems.*

[8] Pat Hanrahan. "Three-Pass Affine Transforms for Volume Rendering", Proceedings of the San Diego Workshop on Volume Visualization (San Diego, CA, December 10-11, 1990). In *Computer Graphics*, pages 71–78, November 1990.

[9] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.

[10] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford DASH Multiprocessor. *Computer*, pages 63–79, March 1992.

[11] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH Prototype: Implementation and Performance. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 92–103, May 1992.

[12] Marc Levoy. Display of Surfaces from Volume Data. *IEEE Computer Graphics & Applications*, pages 29–37, May 1988.

[13] Marc Levoy. Efficient Ray Tracing of Volume Data. *ACM Transactions on Graphics*, pages 245–261, July 1990.

[14] Marc Levoy. Volume Rendering by Adaptive Refinement. *The Visual Computer*, pages 2–7, February 1990.

[15] Mark S. Papamarcos and Janak H. Patel. A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 348–354, June 1984.

[16] Renben Shu and Alan Liu. A Fast Ray Casting Algorithm Using Adaptive Isotriangular Subdivision. In *Visualization '91*, pages 232–238, October 1991.

[17] Per Stenstrom, Truman Joe, and Anoop Gupta. Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 80–91, May 1992.

[18] K. R. Subramanian and Donald S. Fussell. Applying Space Subdivision Techniques to Volume Rendering. In *Visualization '90*, pages 150–159, October 1990.

[19] Lee Westover. Footprint Evaluation for Volume Rendering. Proceedings of SIGGRAPH '90 (Dallas, Texas, August 6-10, 1990). In *Computer Graphics*, volume 24(4), pages 367–376, August 1990.

[20] Terry S. Yoo, Ulrich Neumann, Henry Fuchs, Stephen M. Pizer, Tim Cullip, John Rhoades, and Ross Whitaker. Achieving Direct Volume Visualization with Interactive Semantic Region Selection. In *Visualization '91*, pages 58–65, October 1991.