Distributed Interactive Ray Tracing for Large Volume Visualization

David E DeMarle   Steven Parker   Mark Hartner   Christiaan Gribble   Charles Hansen

[demarle|sparker|hartner|cgribble|hansen]@sci.utah.edu
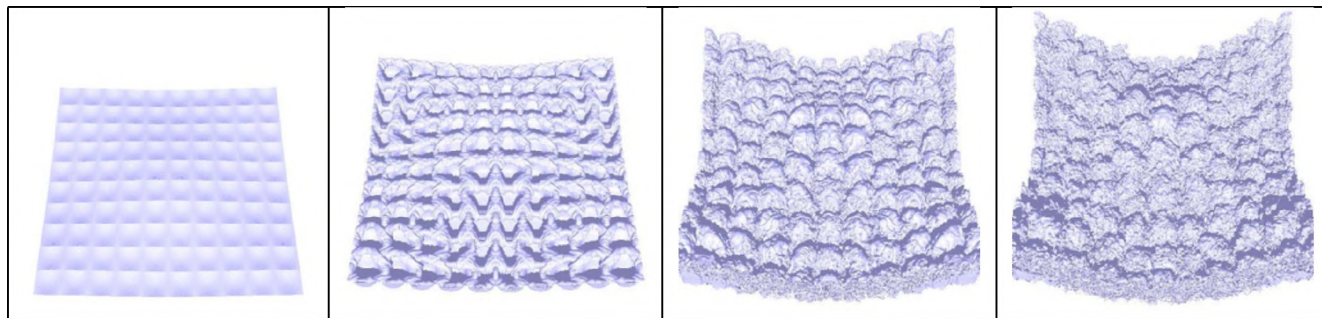The University of Utah

Figure 1: Richtmyer-Meshkov Instability time steps:0,45,180,270. With 32 Linux PCs we are able to isosurface the full resolution 7.5 GB volume on the left at 6.7 frames per second and on the right at 2.1 frames per second.

Abstract

We have constructed a distributed parallel ray tracing system that interactively produces isosurface renderings from large data sets on a cluster of commodity PCs. The program was derived from the SCI Institute's interactive ray tracer (*-Ray), which utilizes small to large shared memory platforms, such as the SGI Origin series, to interact with very large-scale data sets. Making this approach work efficiently on a cluster requires attention to numerous system-level issues, especially when rendering data sets larger than the address space of each cluster node. The rendering engine is an image parallel ray tracer with a supervisor/workers organization. Each node in the cluster runs a multi-threaded application. A minimal abstraction layer on top of TCP links the nodes, and enables asynchronous message handling. For large volumes, render threads obtain data bricks on demand from an object-based software distributed shared memory. Caching improves performance by reducing the amount of data transfers for a reasonable working set size. For large data sets, the cluster-based interactive ray tracer performs comparably with an SGI Origin system. We examine the parameter space of the renderer and provide experimental results for interactive rendering of large (7.5 GB) data sets.

CR Categories: I.3.2 [Computer Graphics]: Graphics Systems—Distributed/network graphics I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

Keywords: Visualization, Interactive Ray Tracing, Large Data,

Cluster Computing, Distributed Shared Memory

1 Introduction

Recent research has demonstrated the utility of ray tracing as an interactive visualization technique on large-scale tightly coupled supercomputers [Parker et al. 1999a; Parker et al. 1999b]. *-Ray[1], the Scientific Computing and Imaging (SCI) Institute's interactive ray tracing system, targets small to large shared memory platforms, such as the SGI Origin series, to interact with very large-scale data sets. Unfortunately, these shared memory supercomputers are extremely expensive and, thus, may not be readily available to many researchers.

As the price of consumer hardware components continues to drop, commodity-based clusters begin to offer a cost-effective alternative to large-scale tightly coupled systems. High-performance clusters that rival traditional supercomputers for solving computationally intensive offline problems are becoming more widely available. With their lower bandwidth and higher latency interconnect, it is less clear that clusters are as well suited for higher throughput tasks such as interactive rendering. It has been shown that large triangle-based scenes can be ray traced interactively on clusters [Wald and Slusallek 2001]. In this paper we show that clusters can also interactively render very large volume data.

We have constructed a distributed interactive ray tracing system for visualizing large high resolution data sets. The new program was derived from *-Ray; however, the slower network and lack of a hardware-based shared memory introduces a number of system-level efficiency issues that must be considered carefully, especially when interactively rendering data sets larger than the address space of an individual cluster node. The distributed renderer is an image parallel ray tracer in which workers completely ray trace rectangular pixel regions. To process volumes that are too large to fit into the memory of each node, the renderer uses a software distributed shared memory that communicates data on demand to make the entire memory space of the cluster available to each node. In this

---

[1] Pronounced Star ray, also known as the real-time ray tracer (rtrt)

COMPUTER SOCIETY

paper, we describe the performance of the distributed parallel ray tracing system on a nearly 8 GB Richtmyer-Meshkov instability data set. Views of four time steps from this data set are given in Figure 1.

In Section 2, we briefly discuss important related work in interactive ray tracing and distributed shared memory systems. The implementation of our distributed parallel ray tracer is thoroughly described in Section 3. Section 4 offers an evaluation of the limitations to scaling of the parallel program, and Section 5 investigates the optimal parameter settings for interactively rendering isosurfaces from a 7.5 GB Richtmyer-Meshkov instability data set. Section 5 also compares the performance of *-Ray and the cluster-based ray tracer with the data set. Finally, we offer our concluding remarks and outline areas of future research in Sections 6 and 7, respectively.

2 Related Work

*-Ray [Parker et al. 1999a; Parker et al. 1999b] was one of the first interactive ray tracing engines. The renderer is carefully optimized for the memory hierarchy of the SGI Origin [Sil 2002] series of shared memory supercomputers. One of the advantages of the supercomputer is that the interconnection architecture has extremely high bandwidth and low latency. This interconnect, combined with hardware-based atomic fetch and op support, allows efficient task assignment and ensures that every processor has very fast, transparent access to a huge global memory space. In a cluster, shared memory is not supported in hardware, and the interconnection network is slower with significantly higher latency, including both hardware and software delays. In this paper, we describe how we utilize all of the nodes in the cluster with a cohesive program that is capable of accessing the aggregate cluster memory as a global memory space.

Ingo Wald and Philipp Slusallek have also demonstrated the feasibility of interactive ray tracing on PC and cluster-based systems [Wald and Slusallek 2001; Wald et al. 2001; Wald et al. 2002]. For large scenes, they obtain the lower branches of a Binary Space Partitioning (BSP) hierarchy on demand from a central file server. One advance in our work is that we eliminate the bottleneck implied by a central data server. In our system all nodes serve some of the data to the rest of the nodes. One significant feature of the Saarland renderer that our system lacks is the ability to reorder ray computation, which allows their rendering engine to hide some of the transmission time for missed data. Their system is highly optimized and as a consequence is restricted to scenes composed entirely of triangles. One of the attractions of ray tracing is its flexibility. Our system retains the capability to render a variety of objects, and as a consequence it is not as highly optimized and will likely under-perform their renderer on triangular meshes. In volume visualization, the point is moot because it is better to isosurface volume data directly without constructing an intermediate polygonal representation. Doing so produces more accurate images and reduces preprocessing time with dynamically changing isovalue selections.

Our approach to shared memory is similar to the work of Corrie and Mackerras [Corrie and Mackerras 1993]. They implemented volume rendering on a Fujitsu AP1000, a distributed memory, message passing parallel computer. They demonstrated that caching makes feasible the volume rendering of data sets that are too large for the memory of any one computing element. Our approach implements a similar algorithm on modern commodity hardware. Our algorithm allows hybrid parallel rendering, where each node runs multiple render threads. To achieve greater interactivity, we present techniques for reducing the number of shared data accesses, improving the hit rate and decreasing the access time.

In this paper, we use the output of a very large Computational Fluid Dynamics (CFD) simulation to examine the performance of our distributed renderer. The data set was generated by A. A. Mirin

et al., from the Lawrence Livermore National Laboratory [Mirin et al. 1999]. The simulation is a Richtmyer-Meshkov instability, produced when a shock intersects a contact discontinuity between two zones of varying density. The results show the evolution of entropy throughout the volume over time. Their work demonstrates the need to work with very high resolution data. With such extreme resolution, they were able to examine how the interaction between long and short wavelength perturbations creates fine scale turbulence. Previously, the data set has been rendered on PC clusters using direct volume rendering by David Porter and a team from the University of Minnesota [Porter 2002] with rendering times in the tens of seconds per frame. Although the rendering method is different, the contribution of our work is the ability to *interactively render*, i.e., at multiple frames per second, the data set without resorting to sub-sampling.

3 Implementation

The cluster we employ consists of 32 dual-CPU 1.7 GHz PC's running Linux and an Extreme Networks 6816 Black Diamond gigabit Ethernet switch. Each node connects to the switch through a gigabit network card and cat5e copper Ethernet cable. Table 1 lists the hardware and software components of our cluster.

Component	Type
Motherboard	Supermicro P4DC6+ (Intel 860 chip set)
CPU	Dual Intel Xeon 1.70 GHz (256 KB cache size)
Memory	2x512 MB Corsair ECC RDRAM
Network card	Intel Pro1000/XT Driver version 4.4.12-k1
GPU	NVIDIA GeForce3 (64 MB) Driver version 1.0.2960
Hard drive	18 GB Seagate Cheetah U160 (15000 RPM)
Kernel	Linux 2.4.20 (Nov 28, 2002)
Compiler	GCC 3.0.4
Cluster Filesystem	PVFS version 1.5

Table 1: Cluster Components.

In our implementation, a supervisor node divides the image into rectangular tiles, assigns groups of tiles, or tasks, to the worker nodes, and refreshes the display. Each worker owns a portion of the total data, and uses the rest of its local memory space to temporarily cache remote owned data. The workers ray trace their assigned image tiles, for each pixel computing the analytic intersection of one ray with the isosurface. When a ray traverses some portion of data residing on a remote node, the distributed shared memory system first checks the cache and then obtains the data from the remote owner if necessary.

Data parallelism is an alternative approach where the volume is statically divided among the nodes, and rays or partial pixel results are transferred instead of data. We chose an image parallel approach because it exploits the natural parallelism inherent in ray tracing. Every primary ray is independent of every other, so parallelizing the algorithm in the image plane adds little overhead to the program, and generally attains good speedup. Given that the communications latency is high on the cluster, and that the algorithm spawns a huge number of rays, we chose not to explore the alternative where individual rays are transferred to nodes that own the data. We also wanted to avoid load imbalance problems that can occur in a data parallel renderer with views that focus on a small portion of the dataset. Focusing on small features is essential for data exploration of the finer details captured by high resolution data.

To fully exploit the dual processor nodes in our cluster, each worker can run one or more independent rendering threads. The supervisor sends task assignment messages to each worker, where a TaskManager object maintains a queue of tasks for the rendering threads. The TaskManager breaks each assigned task into several smaller subtasks in order to provide work for each of the rendering threads. The TaskManager then adds the subtasks to the local task queue. We solve the producer-consumer problem presented by the producing TaskManager and the consuming render threads with two semaphores. Task results are transferred from the workers to the supervisor with sendmsg() and recvmsg() system calls. With these calls, multiple tiles can be sent, and all scan lines within a tile can be received with a single trap to the operating system. The communication between the supervisor and a rendering worker is illustrated in Figure 2.
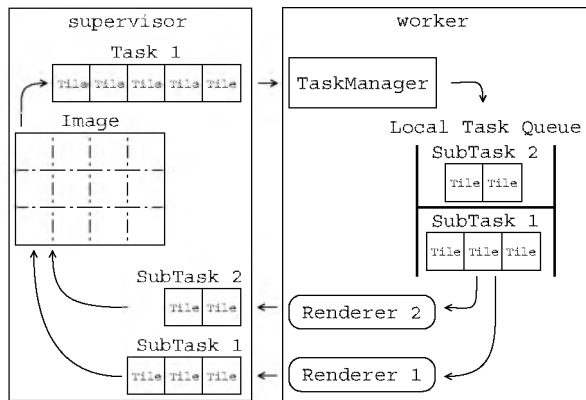

Figure 2: Task Assignment. Each worker divides large incoming tasks into smaller subtasks and maintains a backlog of work for one or more render threads.

Within a node, threads communicate over shared memory. Between nodes, we use a lightweight network layer built on top of TCP. The network layer is designed to separate computational threads from message handling responsibilities. An independent communication thread asynchronously handles incoming network traffic. The communicator thread spends most of its time in a select() system call, watching a set of socket connections to the rest of the nodes. When a socket becomes active, the communicator thread wakes and reads a 4 byte header. The header identifies a particular registered message handling function that can read and process the rest of the message. To ensure thread safety, all outgoing messages are sent through the communicator's send method, which uses a mutex to prevent threads from polluting each others' output streams. Figure 3 shows a high level view of a generic program that uses our network library on four nodes. The first node has T computation threads and M registered message handling functions.

We have found that it is faster for the workers to obtain data over the network than it is to have them demand page data from the local filesystem. Therefore, the worker nodes use software distributed shared memory to process volumes that are too large to be replicated. Distributed shared memory is implemented in a C++ object called a DataServer. The DataServer consists of two allocated memory regions and a method that can handle block request and block reply messages. In this application, each block is a bricked cube within the data set. We use a fixed distributed ownership scheme where each node in the cluster owns $1/(\#of\,nodes)$ of the blocks. Each node places the blocks that it owns in the resident_set region. The DataServer caches remote owned blocks in the local_cache region.

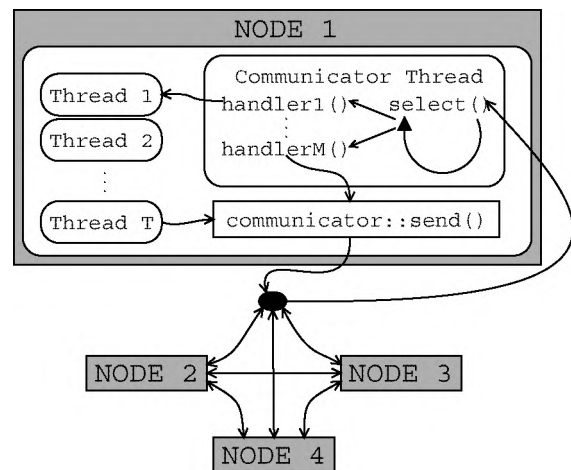When a ray touches a brick, it must acquire that brick from the

Figure 3: Asynchronous Message Handling. A communicator thread relieves computational threads from message handling duty. The communicator's send method ensures thread safety.

DataServer. On an acquire, the DataServer checks if the brick is present locally, obtains it over the network from the responsible node if not, and then returns the address of the brick in local memory to the requesting thread. Once the thread finishes using the brick, it must release the brick so that the DataServer can reuse that cache space. The DataServer protects each brick with a counting semaphore to allow multiple render threads to share a brick, while guaranteeing that the brick stays loaded between each acquire and release. A generic three node program, with two application threads per node, using our distributed shared memory is illustrated in figure 4.
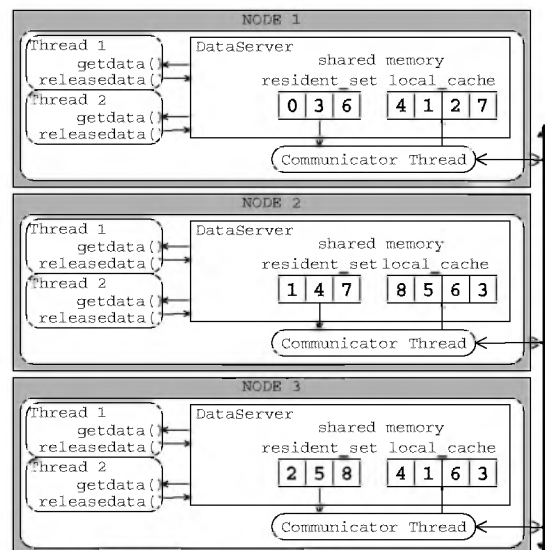

Figure 4: Distributed Shared Memory Architecture. Each node stores its own data in the resident_set and places remote owned data in the local_cache.

The ray tracer uses a hierarchical grid acceleration structure and three level bricking (three-dimensional tiling) for volumes, as discussed in further detail in [Parker et al. 1999b] and [Parker et al. 1998]. Both structures are created offline with a separate preprocessing program. The offline program takes 40 minutes to brick

and create the acceleration structure for the 7.5 GB volume studied in this paper. The bricked data file is the same size as the raw data file, and the acceleration structure file is 8.5 MB.

Each level of the hierarchical grid acceleration structure is a grid of "macrocells" that list the minimum and maximum data values from the cells contained in the lower levels of the hierarchy. At the bottom level, each macrocell represents approximately 8^3 voxels. At the top level a single macrocell contains the minimum and maximum values for the entire volume. Rays traverse the hierarchy first, using an incremental grid walking algorithm at each level. When a macrocell is found that may contain an intersection, the ray recursively examines the contained cells in the next lower level in the hierarchy. By examining the macrocells in this way, the ray is able to to skip large sections of the volume, significantly reducing the number of accesses to the data bricks.

When a ray descends to the bottom level of the acceleration structure, data bricking makes access speed independent of ray orientation, and lets the ray tracer utilize the hardware memory architecture more efficiently. The first level of bricking bricks the data onto 64 byte cache lines, the second bricks the data onto 4 KB virtual memory pages. A third level of bricking increases the data granularity for better bandwidth utilization. These largest bricks are the unit of sharing with distributed data, i.e. each entry in the DataServer holds one brick. The effect of varying the granularity is examined in Section 5.

To use the DataServer, the ray tracing engine has to map data indices to brick numbers and offsets within bricks. We separate equations 1 and 2, which together describe the location of a data value in bricked memory, into expressions of x, y, and z, and tabulate during preprocessing. This reduces the need for expensive divide and mod operations during run time. Here N_y and N_z are the data y and z dimensions, n is the third root of the number of data elements per level one brick, and m and o are the same measures for level 2 and level 3 bricks, respectively.

$$
\begin{aligned}
brick = \\
(x \div n \div m \div o)(N_z \div n \div m \div o)(N_y \div n \div m \div o) + \\
(y \div n \div m \div o)(N_z \div n \div m \div o) + \\
(z \div n \div m \div o)
\end{aligned} \quad (1)
$$

$$
\begin{aligned}
offset = \\
(x \div n \div m \bmod o)n^3 m^3 o^2 + \\
(x \div n \bmod m)n^3 m^2 + \\
(x \bmod n)n^2 + \\
(y \div n \div m \bmod o)n^3 m^3 o + \\
(y \div n \bmod m)n^3 m + \\
(y \bmod n)n + \\
(z \div n \div m \bmod o)n^3 m^3 + \\
(z \div n \bmod m)n^3 + \\
(z \bmod n)
\end{aligned} \quad (2)
$$

Because access to the distributed shared memory is slow compared to normal memory access, we need to reduce the number of accesses further to achieve interactivity. To do so, we consolidate accesses to data at the bottom level of the acceleration structure. That is, rather than perform an acquire to obtain each data value, we acquire a brick and then obtain all of the needed values inside. When a ray enters a bottom level macrocell, it performs a pretraversal to make a list of required bricks. The ray then acquires each touched brick in turn, copying out all intersected voxels before

moving on to the next brick. Figure 5 illustrates a 2D example. The ray first determines that it needs the data within the green macrocell. Inside the macrocell, it performs one acquire on brick 3 to get all of the yellow voxels, and then performs one acquire on brick 4 to get all of the red voxels. The naive approach would acquire 32 times, getting each voxel corner in turn, and take nearly 32 times as long.
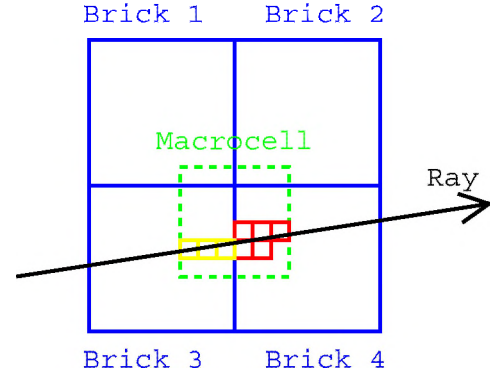

Figure 5: Consolidating Access to Shared Memory. Because distributed shared memory is slow, it pays to reduce the number of accesses. By examining the ray segment in the bottom level macrocell, we get all voxels touched inside a brick with one acquire.

Figure 6 shows diagnostic views of one time step from the Richtmyer-Meshkov data set. The top view shows the whole volume and the other, a small portion of the volume. Render times for each view are between one and two frames per second on 32 nodes. 16x16 pixel image tiles are shown with varied brightness, and 32 KB data bricks are shown with varied hue.

4 Scaling

Network communication prevents perfect scaling on the cluster, even when the data set is small enough to be completely replicated on all of the nodes. The is a consequence of our choice of using a central executive, which was taken to simplify the control logic and is a legacy inherited from the earlier *-Ray architecture. Although the ray tracing computation time is scalable by the number of processors, the renderer can run no faster than a single supervisor can assign and obtain pixel tasks. At interactive rates, this is the eventual bottleneck, although the scene complexity is often such that it takes many processors to reach it. It should be noted that *-Ray does not suffer from this limitation because the SGI interconnection architecture is higher performing.

The tile transfer time is a function of the network latency, the number of tiles in the image, and, to a lesser extent, the network bandwidth. We have measured the effective per task round trip network latency, accounting for asynchronous queuing, to be 19 μs, and the network bandwidth to be 636 $Mbit/s$. With 16x16 tiles in a 512x512 image, we have a maximum frame rate of 34.6 frames per second. With 8x8 tiles, we are limited to 11.5 frames per second. The effect of latency can be reduced with static assignment, larger tiles, and larger tasks containing more tiles. Static assignment eliminates all assignment messages from the supervisor to the workers, reducing the pixel transfer time to nearly half. However, these three tactics make the program more susceptible to load imbalance.

With distributed data, tile transfer time is still the ultimate scaling limitation, but slower data access slows the system down and makes it much harder to reach the maximum frame rate. Distributed memory rendering can exhibit super-linear scaling when the work-
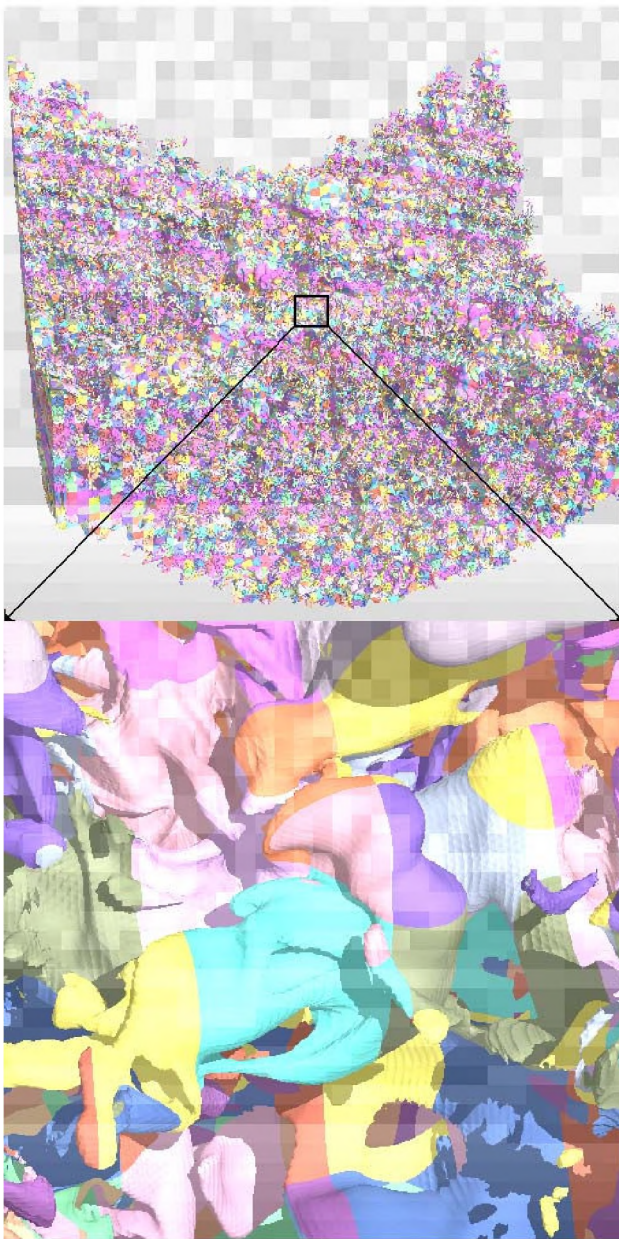
COMPUTER SOCIETY

Figure 6: Diagnostic Views of the Richtmyer-Meshkov Data Showing Work and Data Subdivision. Image tasks are shown with varied brightness, data ownership is shown with varied hue.

ing set is large. With an 8 GB data set, it takes 8 cluster nodes to render the data, unless we take the performance hit of memory mapping from disk. With more nodes, we can divide the data into smaller pieces, leaving more space for the local_caches. This allows more of the working set to be cached, increasing the hit rate and the frame rate. With enough nodes to cache the working set, scaling returns to linearity. In this situation, the system is compute bound, and the largest component of the compute time is the hit time to cached bricks. The hit time is significant because the DataServer has to locate and verify the contents of every brick touched by the ray. In contrast, replicated data access only needs to dereference a pointer. For this reason, we observe distributed data frame rates that are roughly 40 % of their replicated data counterparts.

Figure 7 shows the measured frame rates for both *-Ray and the

new system on the 428 MB torso section of the visible female data set. The test measures the average frame rate over 100 static frames of the view shown in Figure 8. *-Ray runs on a 32 processor Origin 3000. Each processor is a 400 MHz MIPS R12K, and the entire machine has 16 GB of RAM. All processors are connected with ccNUMA technology, a hypercube network that yields 600 ns worst case latency and a link bandwidth of 1.6 GB per second between compute and memory elements.

Two points need to be made about the graph. First, we use half as many processors for the SGI test. With one processor per node, the cluster tests run at half the listed rates. Second, the graph does not show that *-Ray continues to scale to at least 1024 processors. *-Ray scales higher because pixel task scatter and gather operations are essentially free. As explained above, the cluster frame rate eventually reaches a limit determined by the number of tasks in the image.
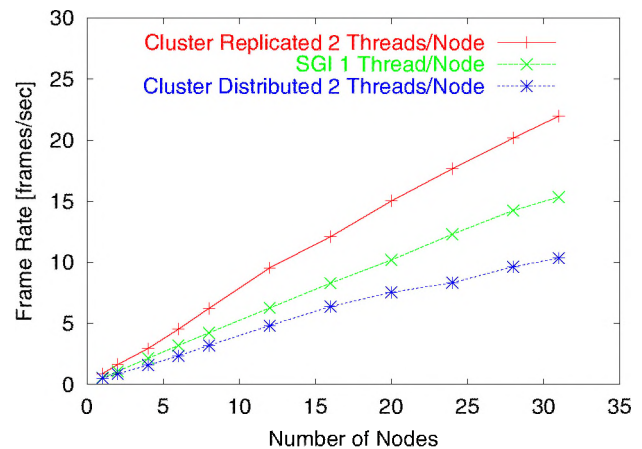

Figure 7: Scaling on the Visible Female Torso. Note, the cluster uses twice as many processors.

Figure 8: Tested View of the Visible Female Torso.

5 Results

We have used our system to interactively visualize isosurfaces of time steps from a Richtmyer-Meshkov instability computation from

the Lawrence Livermore National Laboratory. Each time step is a 1920x2048x2048 scalar volume composed of 8 bit data values (unsigned chars). We perform several experiments on this data set to examine the performance of the renderer and to find parameters that yield the best interactivity. Our system relies on caching, so we obtain good results by decreasing the number of data accesses, increasing the hit rate, and decreasing the hit time and miss penalties.

All experiments benchmark the frame rate over a recorded interactive session in which the viewer inspects the volume by changing the view point and isovalue selection. Figure 9 shows changes in viewpoint and isovalue selection throughout the session, and presents image snapshots of key frames. At frame 44, and from frame 95 to 107 the isovalue is changed. The sudden change at frame 44 causes all caches to be filled at once. From frame 161 to 233, the view point is rotated around the volume. From frame 286 to 344 the view zooms in on a small portion of the volume. From frame 414 to 434 and at frame 460, the isosurface is again changed. From frame 500 to 555, the view point pans over the volume.
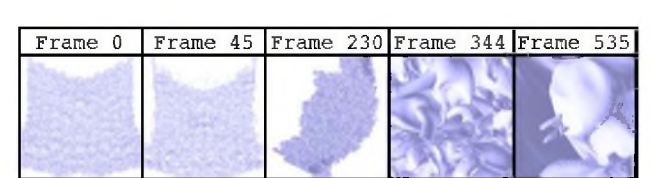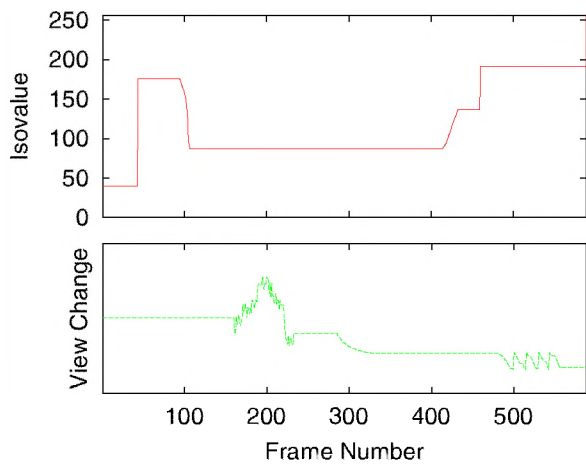

Figure 9: Interactive Testing Session. We benchmark a recorded interaction with isovalue and view exploration. The first half of the test has the entire volume in view to explore the general shape of the data. The last half zooms in to explore fine detail.

Each experiment uses 31 worker nodes and two render threads per worker. Unless otherwise noted, all tests render time step 270, which requires the most data and is the slowest to render. We use Lambertian surfaces and do not calculate shadows for these experiments. All images are generated at 512x512 resolution, using 16x16 pixel tiles. Larger images are preferable of course, especially considering the resolution of the data and the point sampling nature of the algorithm. Images at 1024x1024 resolution have been tested to run at one quarter of the presented rates.

The first experiment demonstrates how consolidation improves performance by reducing the number of accesses to the data. This is the most important technique in achieving interactive rates. Table 2 shows the average number of accesses per worker per frame and the average frame rate using three access patterns. In the Access 1 row, the renderer acquires and releases one voxel corner at a time. In the Access 8 row, the renderer acquires and releases the bricks

touched by the 8 corners of a voxel in turn, usually retrieving all eight values in one access. In the Access Many row, the renderer lists all bricks touched by the ray segment inside a bottom level macrocell, and then acquires and releases each brick in turn, usually obtaining more than eight values with each access. In each case, the frame rate increase is inversely proportional to the decrease in the number of accesses, minus the overhead of the brick sorting algorithm.

Pattern	Accesses	Frame Rate [f/s]
Access 1	3279000	.1149
Access 8	453400	.7090
Access Many	53290	1.686

Table 2: Consolidation. Reducing access to distributed shared memory increases the frame rate substantially.

The second experiment analyzes the effect of associative caching. Associativity increases the hit rate by providing alternative placements for each fetched brick. It also increases the hit time, because all locations must be checked for the presence of a brick and timestamps must be kept. Table 3 shows the measured hit rate, hit time, and frame rate for direct mapped and 4, 8, 12, and 16 way associative caches. We find that for this data set, the working set fits within the cache and thrashing is not a problem. In this situation, a direct mapped cache is more effective than an associative one. On machines with less RAM per node, thrashing is more of a problem, and we have found that associativity will improve the frame rate.

Assoc	Hit Rate [%]	T Hit [μs]	Frame Rate [f/s]
Direct mapped	99.48	6.857	1.686
4	99.60	11.55	1.521
8	99.63	13.17	1.443
12	99.63	14.01	1.361
16	99.65	15.48	1.286

Table 3: Associativity. Although associative caching increases the hit rate, the extra overhead negates the benefit.

The third experiment analyzes the effect of varying the level 3 brick size. Larger bricks increase the hit rate because once a value is accessed, nearby values are more likely to be found in the same large brick. Larger bricks also increase the miss penalty by lengthening the wire time and increasing competition for access to block serving nodes. Table 4 shows the measured hit rate and miss penalties as the level three brick size is increased. For this data set, a 32 KB level three block size is the best trade-off, but the algorithm is relatively insensitive to smaller block sizes.

Brick [KB]	Hit Rate [%]	T Miss [μs]	Frame Rate [f/s]
4	98.72	587.0	1.607
32	99.48	1411	1.686
108	99.64	3972	1.309

Table 4: Data Granularity. Large bricks are reused more often, but take longer to transfer.

The fourth experiment analyzes how the system reacts as the complexity of the data is increased. Because the CFD data set is a simulation of the progression of turbulence, each time step has a progressively more complex isosurface at any given isovalue. Rendering time step 0 is trivial because the isosurfaces are close to planar. Time step 270 is far from trivial, with very turbulent surfaces at

all isovalues. The more disordered the data, the greater the number of accesses, and the larger the working set. Large working sets decrease the hit rate. Table 5 shows the number accesses per worker per frame, the hit rate, and the frame rate for seven time steps.

Time Step	Accesses	Hit Rate [%]	Frame Rate [f/s]
0	22200	99.94	6.691
45	31270	99.83	4.224
90	38160	99.77	3.352
135	41940	99.68	2.759
180	51760	99.61	2.095
225	56090	99.56	1.784
270	53580	99.48	1.686

Table 5: Data Complexity. Complex isosurfaces stress the cache more and take longer to render.

The next experiment examines the use of static scheduling. With dynamic scheduling, the tile assignments change frequently and each worker must render most of the image over time. With static scheduling, each worker renders the same set of tiles repeatedly. Data coherence is better in this situation because each worker constantly accesses the same small portion of the data. The trade-off is that there is little load balancing with static assignment; only render threads on the same node are able to cooperate to render difficult portions of the image. Table 6 shows the measured hit rate and frame rate with static assignment. For the first time steps, the data coherence gains do not outweigh the load imbalance penalties. For the very complex time steps, the trade-off is more worthwhile because the memory utilization is much greater.

Time Step	Hit Rate [%]	Frame Rate [f/s]
0	99.95	4.877
45	99.92	3.683
90	99.92	3.161
135	99.90	2.659
180	99.89	2.244
235	99.88	2.101
270	99.84	2.088

Table 6: Static Assignment. Static task assignment increases load imbalance, but allows exploitation of frame to frame coherence.

The frame rate of the renderer is inversely related to the amount of data that needs to be obtained by the render threads. The relationship can be observed by using the detailed diagnostics built into our network library to measure the frame rate and the data traffic. Figure 10 shows the recorded data traffic and frame rate over a session using one render thread per node. The bottom panel reproduces the isovalue and view point behavior. In the frame rate panel, the first graph shows the frame rate when the traffic is allowed to stabilize. This was recorded in a longer session in which every frame is repeated twenty times and then the graph was scaled horizontally to overlay the others. This plot demonstrates the rendering speed assuming a perfect cache.

During the first half of the test, the entire volume is in view and the performance is highly dependent upon the caching behavior. Here, every change in view point or isosurface causes the data traffic to spike and then decay. Meanwhile, the frame rate drops and then gradually increases. It takes roughly fifty frames to reach steady state. In the second half of the test, the view is zoomed in. All workers are easily able to cache visible data, and the frame rate is limited only by the rendering engine.
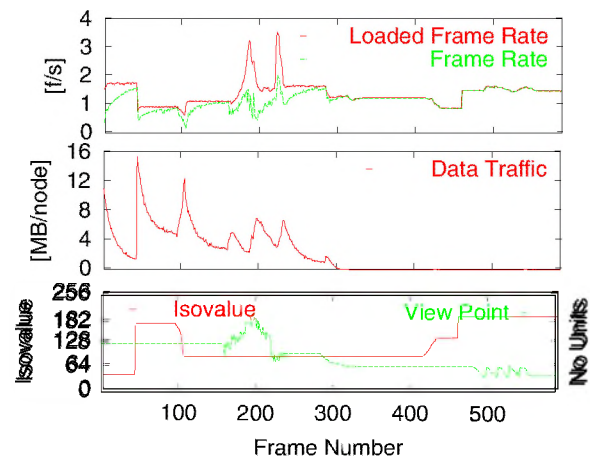

Figure 10: Data Traffic and Frame Rate. The frame rate depends upon how fully the caches are loaded.

Finally, we compare the performance of the distributed renderer with *-Ray. On the cluster, we select the parameters that were found to yield the highest interactivity: macrocell consolidated access to blocks, direct mapped caching, and 32 KB blocks. Although static assignment is faster on this data set, we use dynamic tile assignment on both machines and hold all additional render settings constant. Figure 11 shows the recorded frame rates on the cluster with one and two threads per node and on the SGI. In the first half of the test, with the entire volume in view, the cluster implementation struggles to keep the caches loaded, and consistently pays the miss penalty. When the viewpoint is zoomed in on a small portion of the data, the frame rate is compute bound and the cluster version closely follows *-Ray. The average frame rates over the session are 1.075, 1.686, and 4.689 frames per second for the one thread, two thread and supercomputer tests, respectively. With equal number of processors then the cluster is roughly one quarter the speed of the SGI, and with two processors per node, the cluster is roughly one third the speed.
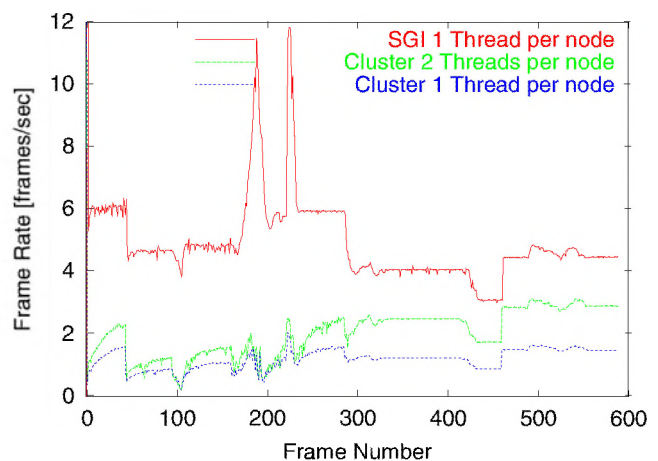

Figure 11: Cluster versus Supercomputer. The performance is comparable even with the large data set, especially when viewing fine detail.

93

6 Conclusions

We have found that it is possible to interactively render isosurfaces of very large volumes with low cost cluster technology. By making the entire memory space available, we are able to render volumes of up to 32 GB, and do so interactively as long as the working set is small enough to be cached locally. Because of serial performance differences, and because the interconnection network in a switch-based cluster is lower performing than the dedicated network in an advanced shared memory supercomputer, the new distributed renderer is less interactive than its predecessor *-Ray. By utilizing TCP sockets rather than a higher level message passing library, and by allowing asynchronous message handling in a multi-threaded application, we have been able to reduce the communication penalty. By taking care to minimize access to distributed shared memory, to increase the hit rate, and to minimize distributed cache access times, we are able to interactively render volumes that are too large to be replicated. With these optimizations, the new renderer is comparable to (although slightly slower than) the original with equal numbers of processors, and represents a significant price win.

7 Future Work

Although not demonstrated here, we obtain similar frame rates for maximum intensity projections as with isosurfaces. We are in the process of studying further optimization of time varying data sets, which are currently handled by creating separate DataServers for each time slice. The next research direction will be to add direct volume rendering to our ray tracer. Volume rendering will increase both the amount of processing on and the number of accesses to the data. Eventually, we plan to extend our system to distribute large polygonal data sets as well, by moving the surface data and the acceleration structure into the distributed shared memory space. These issues will need to be considered carefully in order to maintain a high degree of interactivity.

As we increase the realism of our images by tracing more rays, we will need to consider the computational expense of these additional operations. We hope to exploit the simultaneous multithreading and instruction-level parallelism provided by the Intel Xeon processors of our cluster to accelerate the computational phase of ray tracing. Furthermore, the data access penalties incurred by tracing additional primary and secondary rays must also be addressed, possibly with latency hiding techniques and higher performing network technologies.

Serial performance of the system could also be improved through tuning for the Pentium architecture and possibly through using different compilers. The overhead associated with maintaining the cache could be eliminated by moving to a page-based distributed shared memory architecture in which the virtual memory hardware takes over some of the responsibility [Li 1988]. However, this approach would be limited to 4 gigabyte datasets until 64 bit processors are more readily available in commodity clusters.

Acknowledgments

References

CORRIE, B., AND MACKERRAS, P. 1993. Parallel Volume Rendering and Data Coherence. In *ACM SIGGRAPH 93 Symposium on Parallel Rendering*, ACM Press / ACM SIGGRAPH, ACM, 23–26.

LI, K. 1988. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proceedings of the International Conference on Parallel Computing*, 94–101.

MIRIN, A. A., ET AL. 1999. Very High Resolution Simulation of Compressible Turbulence on the IBM-SP System. In *Supercomputing 1999: High Performance Networking and Computing Conference (electronic publication)*.

PARKER, S., SHIRLEY, P., LIVNAT, Y., HANSEN, C., AND SLOAN, P.-P. 1998. Interactive Ray Tracing for Isosurface Rendering. In *Proceedings of IEEE Visualization '98*, 233–238.

PARKER, S., MARTIN, W., SLOAN, P.-P., SHIRLEY, P., SMITS, B., AND HANSEN, C. 1999. Interactive Ray Tracing. *Interactive 3D Graphics (I3D)* (Apr.), 119–126.

PARKER, S., PARKER, M., LIVNAT, Y., SLOAN, P.-P., HANSEN, C., AND SHIRLEY, P. 1999. Interactive Ray Tracing for Volume Visualization. *IEEE Transactions on Visualization and Computer Graphics 5*, 3, 238–250.

PORTER, D. H. 2002. Volume Visualization of High Resolution Data using PC-Clusters. Tech. rep., University of Minnesota. avail.at http://www.lcse.umn.edu/hvr/pc_vol_rend_L.pdf.

SILICON GRAPHICS INC. 2002. *SGI Origin 3000 Datasheet*. avail.at http://www.sgi.com/origin/3000/datasheet.htm.

WALD, I., AND SLUSALLEK, P. 2001. State-of-the-Art in Interactive Ray-Tracing. *State of the Art Reports, EUROGRAPHICS 2001*, 21–42.

WALD, I., SLUSALLEK, P., AND BENTHIN, C. 2001. Interactive Distributed Ray Tracing of Highly Complex Models. In *Rendering Techniques 2001: 12th Eurographics Workshop on Rendering*, 277–288.

WALD, I., KOLLIG, T., BENTHIN, C., KELLER, A., AND SLUSALLEK, P. 2002. Interactive Global Illumination using Fast Ray Tracing. In *Proceedings of the 13th EUROGRAPHICS Workshop on Rendering*, Saarland University, Kaiserslautern University, 15–24.

COMPUTER SOCIETY