Memory Sharing for Interactive Ray Tracing on Clusters

David E. DeMarle, Christiaan P. Gribble, Solomon Boulos, Steven G. Parker

Scientific Computing and Imaging Institute, University of Utah, 50 S Central Campus Dr Room 3490, Salt Lake City, UT 84112, USA

Abstract

We present recent results in the application of distributed shared memory to image parallel ray tracing on clusters. Image parallel rendering is traditionally limited to scenes that are small enough to be replicated in the memory of each node, because any processor may require access to any piece of the scene. We solve this problem by making all of a cluster's memory available through software distributed shared memory layers. With gigabit ethernet connections, this mechanism is sufficiently fast for interactive rendering of multi-gigabyte datasets. Object- and page-based distributed shared memories are compared, and optimizations for efficient memory use are discussed.

Key words: scientific visualization, out-of-core rendering, distributed shared memory, ray tracing, cache miss reduction

1 Introduction

Computer graphics and visualization practitioners often desire the ability to render data that exceeds the limitations of the available memory and processing resources. Parallel processing is one solution to this problem because it has the potential to multiply the available memory and computing power. Recently, the cluster parallel computing organization has become popular because of the low cost and high performance it affords. Our work utilizes memory sharing techniques that make it possible to render, at interactive rates, datasets larger than those previously possible using affordable computing platforms.

The ray tracing algorithm proceeds by casting a ray into the scene for each pixel P and determining which of the N scene primitives the ray hits first. The

pixel takes the color of that primitive. If the primitive is reflective or translucent, secondary rays are spawned from the point of intersection to determine additional color contributions. The algorithm is versatile, any data type that can be intersected with a line segment can be drawn, and any degree of fidelity can be achieved by tracing additional rays.

The primary drawback of ray tracing is its high computational cost. Spatial sorting allows the algorithm described above to run in $\mathcal{O}(P \log N)$ time. However, because both P and N are large, parallel processing is essential to allow interactive inspection of large datasets.

Parallel rendering is often classified in terms of a geometry-sorting pipeline [1]. The classification scheme is divided according to the point in the pipeline where scene primitives are assigned to individual processors. In sort-first (image parallel) rendering, each processor is responsible for a different subset of the image space, while in sort-last (data parallel) rendering, each processor is responsible for a different subset of the data. In ray tracing, every primary ray can be computed concurrently, so image parallelism is the natural choice to accelerate rendering. Figure 1 shows a diagnostic image of a teapot in which the pixels rendered by three nodes in our cluster have been saturated differently to show workload subdivision.

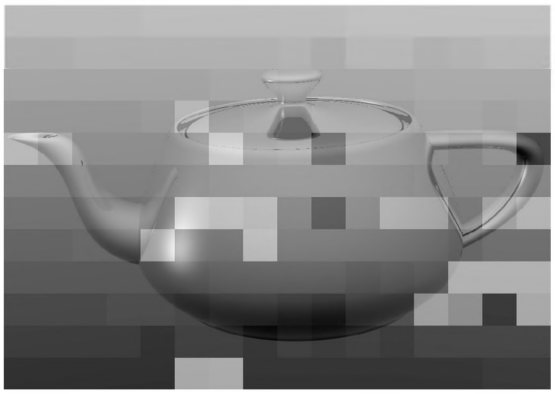


Fig. 1. Pixel Distribution. An image showing which processors rendered which pixels. Three processors add different gray levels to their pixels to create this diagnostic image.

A problem inherent in image parallel rendering is that a processing element may require access to the entire scene database. Each processor is responsible for computing the color of its assigned pixels, and these pixels may contain contributions from any portion of the data. Consequently, image parallel rendering has typically been restricted to small scenes that can be replicated in the memories of every processing element.

In sort-last parallel rendering, each processor is assigned a different portion of the data, so the available memory resources are multiplied. The same goal

can be achieved for image parallel rendering when a mechanism is provided to share data on demand. We leverage a software layer that manages access to scene data and fetches missing pieces over the network as required. In our system, each node runs one or more ray tracing threads and is responsible for managing a different subset of the scene database. To exploit data coherence, the shared memory system caches the remote data locally for later use. Careful attention to memory access patterns, data layout and task distribution can lead to increased locality of reference, higher hit rates and, as a result, better performance.

2 Related Work

Our work stems from that of Parker et al. [2], which demonstrated one of the first interactive ray tracing systems. By exploiting the capabilities of the SGI Origin series of shared memory supercomputers, they were able to achieve interactive frame rates using a brute force implementation of the ray tracing algorithm. On these systems, the problem of data sharing is solved by the ccNUMA interconnection layer. Our work explores the mechanisms that can be used to replace this hardware layer with a software-based distributed shared memory (DSM). A key aspect by which the distributed application is able to maintain interactivity is that, in the rendering context, a writable shared memory space is not required. For this reason, we omit expensive consistency maintenance algorithms. Quarks [3] and Midway [4] are representative examples of full-featured page- and object-based DSMs that handle write access to memory efficiently.

Our approach to memory sharing is similar to the work of Corrie and Mackerras [5]. They implemented volume rendering on the Fujitsu AP1000, a distributed memory, message passing parallel computer. They demonstrated that volume rendering datasets that are too large for the memory of any one computing element is feasible with caching. Badouel et al. [6] used a page-based distributed shared memory, similar to one described here, and compared data parallel and image parallel ray tracing programs. They concluded that image parallel rendering with shared memory will scale better than object parallel rendering because of the increased processing and communication overhead that results from more finely dividing the objects in space. Our approach implements similar algorithms on modern commodity hardware and compares object- and page-based memory organizations. In addition, we present techniques for reducing the number of shared data accesses, improving the hit rate and decreasing the access time.

Several works by Wald et al. [7–9] demonstrate interactive ray tracing of large models in both single PC and distributed cluster environments. Their first

system [7] traced four rays at a time using SIMD instructions to accelerate the rendering process. An additional benefit of this technique is that the data coherence of primary rays is automatically exploited. Other early work by Wald et al. [8] addressed the challenges of interactively rendering large, complex models by combining centralized data access and client-side caching of geometry voxels. Their system takes pains to exploit spatial coherence within BSP tree nodes and temporal coherence between subsequent frames. More recently, they have exploited the 64-bit PC address space to combine asynchronous out-of-core data fetching and approximate transitory geometry to render massive polygonal models on a single workstation [9].

In contrast, our work has primarily focused on developing a flexible interactive rendering engine for scientific visualization applications. Though performance benefits may result, we have not restricted our system to any one type of scene primitive (for example, triangles). Instead, we are exploring more general memory management techniques that can be exploited for any type of scene data, including volumetric and polygonal data.

The designers of the Kilauea ray tracing engine [10] chose the data parallel approach to image rendering. Rather than divide the image into separate areas for each processor, they distribute large scenes among the processors, each of which traces a set of identical rays. Results are merged to determine the primitive that is hit first. They use ray postponement in a queuing system combined with very efficient sub-thread process management to achieve good performance. The Kilauea engine is designed for high-quality global illumination, so the system is not interactive. For our system, in which interactive frame rates are a primary goal, the cost of constantly transporting large numbers of rays across a high latency network was deemed less practicable than occasionally transporting a few large blocks of memory.

Our system can be classified as a hybrid approach that is closer to image parallel rendering than to data parallel rendering. Reinhard et al. [11] describe a different hybrid approach that is closer to object parallel rendering. The design of their system was motivated by the need to evenly balance the load while improving memory coherence. In this system, a grid-based acceleration structure was used to partition the objects in the scene. The demand driven task of determining the set of cells traversed in the grid and finding initial intersections was done in parallel using a data cache for fetched remote objects. Secondary rays spawned from intersection points were sent to remote nodes in a data parallel fashion. Our rendering system differs in that we do not transfer ray ownership and that we reorganize individual meshes to gain the memory coherence benefits implicit in the data parallel approach.

UU IR Author Manuscript

3 Distributed Shared Memory

In all versions of our distributed shared memory, each of the N rendering nodes is assigned $1/N$ of the total data size. The initial assignment of blocks to nodes is arbitrary because we do not know, *a priori*, which data will contribute to which pixels of the image. Similarly, we do not have advance knowledge of which pixels will be assigned to which nodes during rendering. To keep a balanced distribution, we make the individual blocks small relative to the whole scene, for example, 32 KB per block when rendering a multi-gigabyte dataset. We can then assign many blocks to each node using a simple round robin placement scheme, and each node is given a fair initial sampling of the entire scene. In this scheme, block number n is owned by node number $n \bmod \#\ of\ nodes$. The set of blocks that each node is given at program initialization is constant throughout the session, and this portion of a node's memory is called the resident set. Figure 2 illustrates data ownership in a rendering of an isosurface of a volumetric model of the implicit equation $x^2 + y^2 + z^2 + noise = C$.

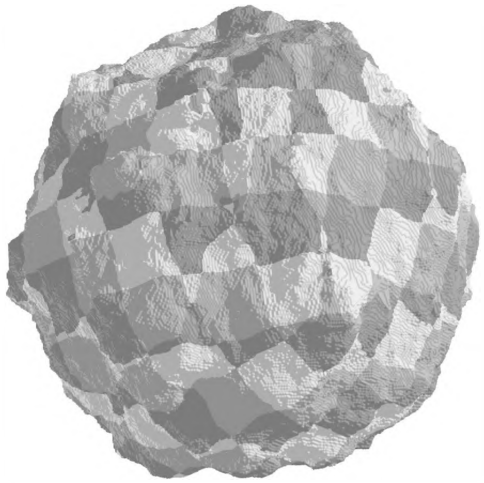


Fig. 2. Data Distribution. Voxels originating in each of three nodes' resident sets are colored differently in this diagnostic view of an isosurface rendering.

In addition to its own resident set, a node may need to access data in the other nodes' resident sets. The separate memory layers are connected via the cluster's interconnection network, over which the nodes send and respond to memory block request messages.

These request and reply messages are handled by a lightweight message passing layer called Ice [12]. Ice utilizes either TCP sockets or the Message Passing Interface (MPI) to connect the nodes in the cluster. An important feature of Ice is asynchronous message retrieval. Computation threads never call the

receive operation directly. Instead, a dedicated thread handles incoming messages. When one arrives, the communication thread wakes and processes the message. The advantage of this approach is that if computation threads have a sufficient backlog of data to process, they do not spend any time waiting on communication.

The DSM would be very slow if every access to a non-resident block resulted in network messages. To avoid this situation, we rely on coherence. Coherence is the property that once a portion of the scene is used, it and nearby portions are likely be used again in the near future. We therefore set aside a portion of the memory in each node to cache non-local blocks. In Section 4, we explain optimizations that increase the probability of loaded data being reused, making caching more effective.

There are two primary types of distributed shared memory: object-based DSM (ODSM) and page-based DSM (PDSM). ODSMs share the memory of arbitrarily sized software objects. These objects are accessed through methods that signal the DSM layer to make the requested memory available to the caller. PDSMs, in contrast, share pages of system memory, where the page size is a fixed, machine dependent number of bytes. Rather than utilize function calls to access memory, the program simply accesses the normal virtual address space, and the DSM layer independently ensures that the needed pages are made available.

We have experimented with both memory organizations. We discuss our implementations of each DSM below, and compare the performance of the two in Section 3.3.

3.1 ODSM

In our ODSM implementation each node creates a *DataServer* object at startup. This object is responsible for managing the node's resident and cached blocks. The ray tracing threads access the shared memory blocks through the *DataServer*'s acquire and release methods. Each call takes an integer handle that selects a particular block of memory from the global memory space. When the renderer accesses a block that belongs to the local node's resident set, the ODSM layer simply returns that block's starting address. When the renderer accesses a non-local block, the ODSM layer must first search the cache for the block and, if it is found, return its local address. If the block is not cached, the ODSM must send a message to the node that owns the block and wait for that node's response. When the requested block arrives, the ODSM places the remote data in the local cache, possibly evicting another block to reclaim space. When a thread finishes using the block, the release method notifies

the *DataServer* that the space used for the block is now available for use by another block.

The ODSM architecture has two important advantages. First, because accesses to the blocks are bounded by acquire and release operations, it is relatively easy to make the ODSM thread-safe. The ODSM protects each block with a counting semaphore that allows multiple render threads to access a block simultaneously. The semaphore also prevents that block from being evicted while it is still in use. Second, because the blocks are accessed indirectly through a handle, the 4 GB address limitation of 32-bit machines no longer applies. The maximum addressable memory is now the size of the integer handle times the size of the block. Taken together, the ODSM makes the aggregate physical memory space of the cluster accessible to any thread.

Although the ODSM makes a large amount of memory available to the application, it may cause difficulties for the application programmer. Figure 3 shows pseudo-code for the process by which the ray tracing application accesses the ODSM memory space. Because the scene data is accessed through handles, the ray tracing threads must map graphics primitives (for example, voxels and triangles) to block handles and offsets within blocks. The mapping process is difficult for many data representations, and the address arithmetic consumes valuable processing time. Thus, with the ODSM, some time is lost accessing the shared data, even in the event of a memory hit.

```
// Locate data
handle, offset = ODSM_location(datum);
block_start_addr = ODSM_acquire(handle);

// Use data
datum = *(block_start_addr + offset);
test_ray_intersection(datum);

// Relinquish space
ODSM_release(handle);
```

Fig. 3. Object-Based DSM Usage. With the ODSM, the ray tracing threads must explicitly access the shared data space.

3.2 PDSM

In a PDSM, the implementation of the shared memory is pushed away from the application, closer to the OS and hardware levels. There are no explicit acquire and release operations. Instead, the shared address space is simply a special region of the machine's local virtual address space. Shared objects are created with an overloaded *new* operator that simply ensures that the shared

objects are created in the PDSM address range. The ray tracing threads need only call *test_ray_intersection(datum)*, as they would for any other piece of data in the node's local memory.

Our implementation divides the PDSM's range of addresses into blocks that are multiples of the operating system's native page size. Each block is assigned to a node in the same round robin fashion as was the resident set of the ODSM. The rest of the pages in the PDSM memory are initially unmapped on each node. The virtual addresses are protected by a segmentation fault signal handler. This routine is called whenever the ray tracing threads access missing, non-local pages. In this case, the handler issues a request to the owner node in a manner similar to that of the ODSM. Figure 4 presents an overview of our PDSM memory access protocol.

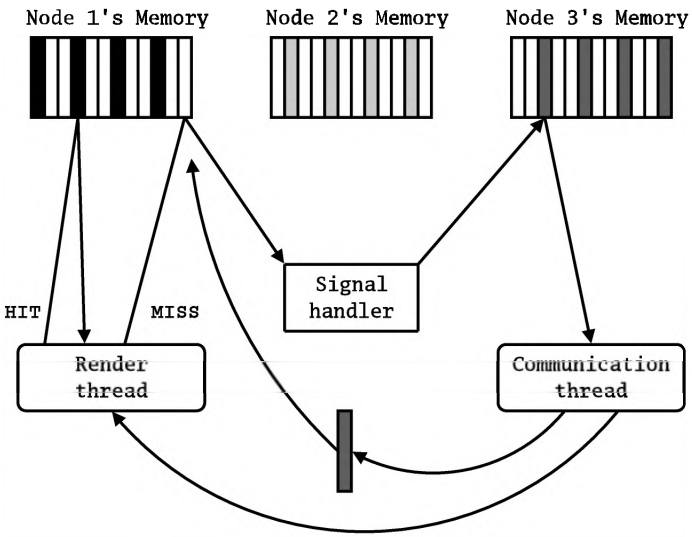


Fig. 4. Page-Based DSM Architecture. The virtual memory hardware detects misses, and the PDSM layer causes remote references to be paged from across the network.

The PDSM layer makes the application programmer's task easier, and the application generally operates more quickly. However, the size of the shared memory space is constrained by the 4 GB limit inherent to 32-bit address machines. In practice, the actual limit is less than 4 GB because some of the virtual address space is reserved for the operating system (addresses above 0xC000000), as well as the program's executable code and free storage space (addresses near 0x4000000). This arrangement leaves at most 2 GB of addressable shared memory. Despite this limitation, the technique provides a useful way to extend the total data size when the installed physical memory on any node in the cluster is less than 2 GB. With the wider availability of 64-bit machines, it is likely that this limitation will become less severe and that the technique will become more useful.

UU IR Author Manuscript

Another drawback to our PDSM implementation is that, unlike the ODSM, it is not currently thread-safe. The PDSM lacks the semantics of explicit acquire and release operations, and exhibits a race condition whenever the user-level communication thread fills a requested page of memory received from a remote node. Without kernel modifications, there is no way to reserve a particular page of memory for a particular thread, so there is some chance that additional ray tracing threads could access the page while it is being filled. For this reason, all of the PDSM tests reported here use only a single ray tracing thread per node.

3.3 ODSM/PDSM Comparison

To compare the performance of the two DSM implementations, we render isosurfaces of a 512 MB scalar volume created from a computed tomography scan of a child's toy. The test machine is a 32-node cluster consisting of dual 1.7 GHz Xeon PCs with 1 GB of RAM, connected via switched gigabit ethernet. All rendering tests report the average frame rate during interactive sessions using a 512x512 pixel view port. The images are composed of 16x16 pixel tiles. Thirty-one rendering nodes are used, with one rendering and one communication thread per node, except where noted.

In this test, we examine the cost of using a shared memory space by restricting the DSM layers to store only 81 MB on each node. Because the viewpoint and isosurface selection change throughout, the working set varies frequently, and the DSM layers must do extra work to obtain the needed data.

Figure 5 shows the recorded frame rates from the test and a sampling of rendered frames. The test is started with a cold cache. In the first half of the test, the entire volume is in view, while in the second, only a small portion of the dataset is visible. Both DSM layers struggle to keep the caches full during the first part of the test. However, the lack of memory indirection gives the PDSM a lower hit time, so it outperforms the ODSM throughout. In later frames, most memory accesses hit in the cache, so the PDSM adds little overhead to data replication. Overall, the average frame rates for this test are 3.74 fps with replication, 3.06 with the PDSM and 1.22 with the ODSM. The PDSM layer is clearly preferable to the ODSM layer as long as the total data size can be addressed by the nodes in the cluster.

UU IR Author Manuscript

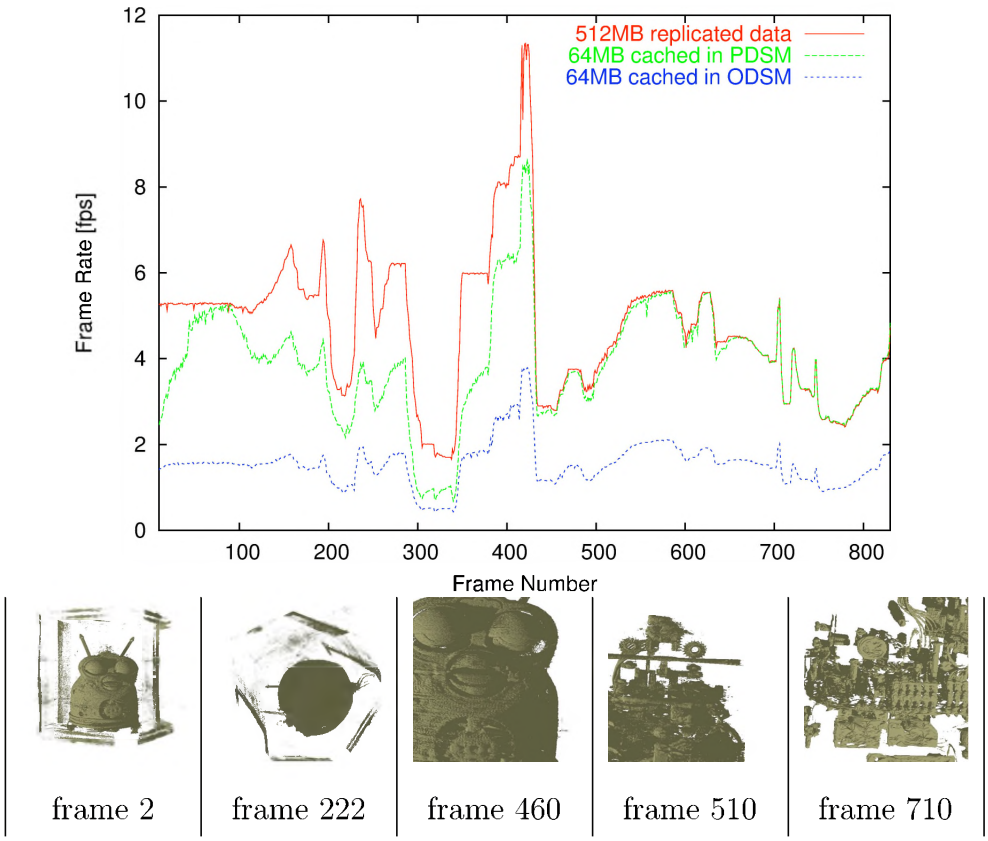


Fig. 5. Comparing Memory Organization. Frame rates are above and images from the test are below. The page-based DSM outperforms the object-based DSM in all cases. Moreover, its performance is competitive with full data replication, even though the local memory size is reduced to 16% of the total.

4 Memory Optimizations

In this section we describe the optimizations we have made to improve the hit rate of our rendering application. Table 1 gives the measured hit and miss penalties for our object- and page-based DSMs recorded in a random access test. The disparity between the hit and miss times under both DSMs justifies our search for optimizations which target increased hit rates. The optimizations include the use of spatial sorting structures, data bricking, access penalty amortization, and a load balancer that exploits frame-to-frame coherence.

	Hit Time	Miss Time
Object-based DSM	10.2	629
Page-based DSM	4.97	632

Table 1
DSM Access Penalties. Average access penalties, in μs, over 1 million random accesses to a 128 MB address space on five nodes.

4.1 Optimizations for Volumetric Data

Our application focus has been rendering the isosurfaces contained within regular volumetric datasets, a common task in scientific visualization. To render this type of data, we utilize the macrocell and bricking acceleration techniques described by Parker et al. [13] and DeMarle et al. [14].

To accelerate rendering, our system uses a macrocell hierarchy that enables space leaping. The hierarchy is an octree-like spatial sorting structure that contains, at each level, a grid of cells storing the minimum and maximum values of the subcells at the next lower level. By traversing the macrocell hierarchy, we need not consider much of the data that is contained within the shared memory space. For non-volumetric data, we use efficient bounding volume hierarchy and hierarchical grid-based data structures [15,16] for a similar purpose.

Data bricking [17], or three-dimensional tiling, reorganizes the 3D array of data in memory to keep proximate volume elements together in address space. Rather than traverse each row of the data in memory before proceeding to the next column and eventually slab, we group neighboring cells in small bricks. The sizes of the bricks are chosen to be aligned on memory hierarchy boundaries. We repeat the process with the bricks to obtain the same benefits on cache line, OS page, and network transfer memory block levels.

4.2 ODSM Access Consolidation

With the structures described above, the ray tracing threads tend to access only a small fraction of the data, and they tend to do so repeatedly. The structures are effective enough that, in the isosurface rendering application, we typically have hit rates of greater than 95%. In this situation, the hit times are a limiting factor. The PDSM memory layer is an option for moderately sized volume data, but for very large data we are forced to use the ODSM. To achieve better performance from the ODSM, we reduce the number of accesses and amortize the cost of each hit over multiple data values.

Our approach is to consolidate accesses to data at the bottom level of the acceleration structure. Rather than perform an acquire operation to obtain each scalar value, we acquire a block and obtain all of the scalar values needed to construct the required voxels within the block. Every ray first traverses the bottom-level macrocells to construct a list of required blocks. The ray then acquires each touched block in turn, copying all of the intersected voxels before releasing the block and moving to the next one.

UU IR Author Manuscript

Figure 6 gives a 2D example of the simple acquire-on-demand and consolidated access strategies. In the simple approach, the ray must perform 28 acquires to obtain data for the 7 voxels that are required. Most of these acquires are redundant. In the consolidated approach, only 2 acquires are needed. The first retrieves 6 scalar values, the second retrieves 10.

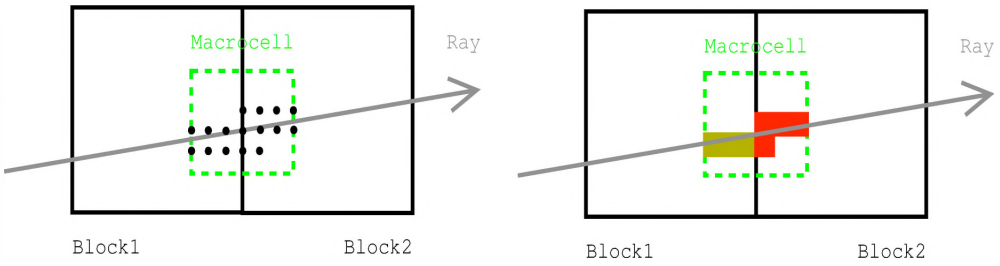


Fig. 6. Consolidating Access to Shared Memory. Because distributed shared memory is slow, it is beneficial to reduce the number of accesses. By examining the ray segment within the bottom-level macrocell, all of the voxels touched inside a block can be obtained at one time.

We experimentally examine the effectiveness of the access consolidation strategy in an isosurface rendering test of a 7.5 GB volume. Table 2 shows the average number of accesses per worker per frame and the average frame rate for a test using three consolidation patterns. The first and last rows correspond to the patterns described above. The second row is an intermediate option in which the renderer acquires and releases the blocks touched by the 8 corners of a voxel in turn, usually retrieving all eight values in one access. In each case, the increase in frame rate is inversely proportional to the decrease in the number of accesses, minus the overhead of the block pre-traversal process.

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

Table 2
Consolidated Access Test Results. Amortizing access to the ODSM is essential for interactivity.

4.3 Mesh Reorganization

We also apply the concept of data bricking to polygonal mesh data. When spatially proximate primitives are sorted in memory so that they become proximate in address space as well, pages are more likely to be reused and

hit rates are likely to increase. Given our image tile access pattern, the ideal mesh for our purposes is one with large patches of triangles that reside on the same page in memory. The goal of making the scene data more coherent by reorganization is similar to that achieved by Pharr et al. [18] for disk cached ray tracing, and by Hoppe [19] and Isenburg et al. [20] for scan line rendering.

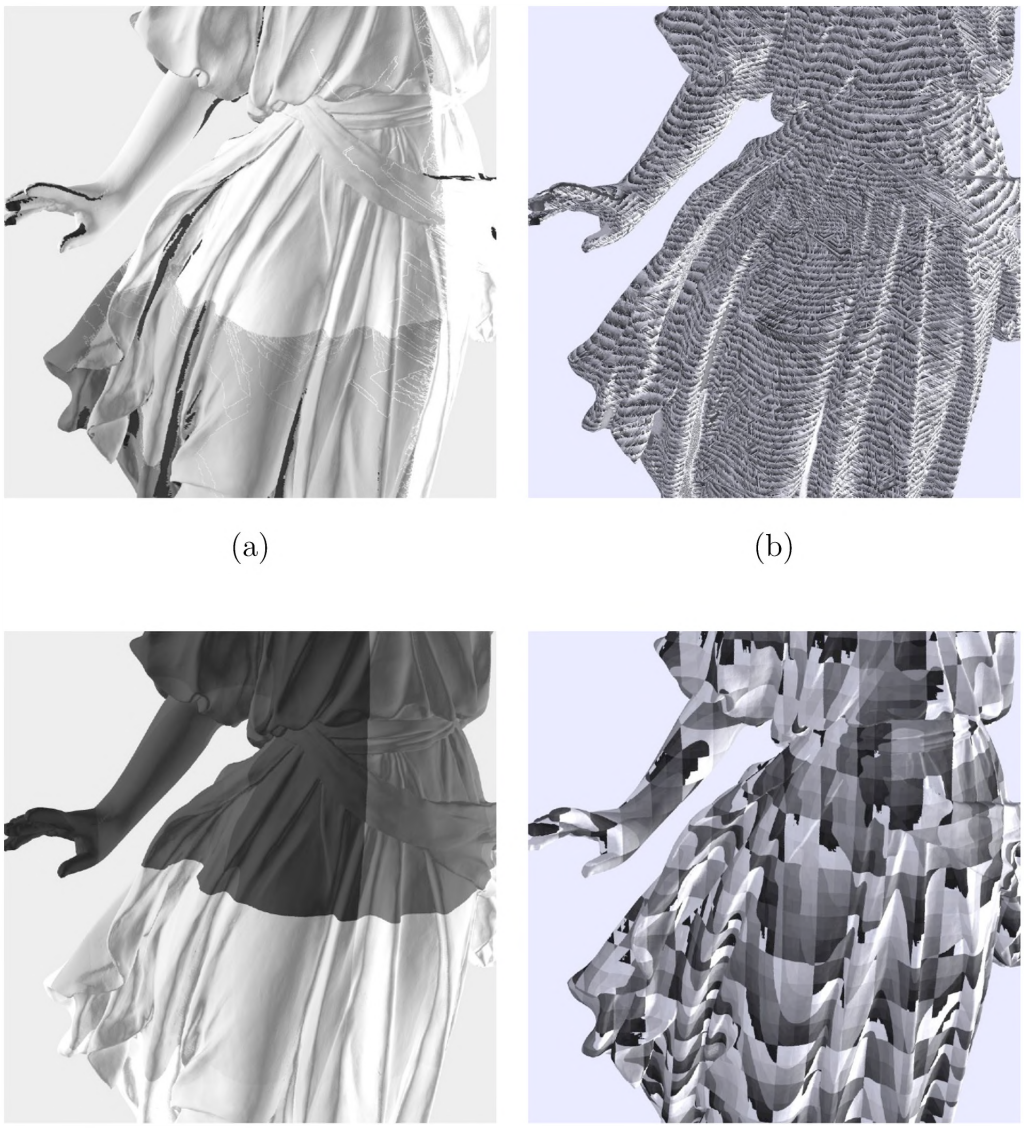
The memory layout of our input data is reorganized using a preprocessing program. The program uses an octree to sort the vertices and triangles of the input mesh in space. The program reads each vertex from disk and inserts it, along with its index in the input file, into one of eight children of the octree's top level cell. We repeat the process within each cell recursively sorting until each subcell contains no more than a small, user-defined number of vertices. In the end all of the vertices within each cell are guaranteed to be close to one another. After the vertices are inserted into the tree, we perform a depth first traversal to append the sorted vertices onto an output file.

We repeat the process for the triangles in the mesh, using the centroid of each triangle to determine the octant in which the triangle should be placed. Once the triangles are sorted, we fix the vertex references of each triangle to index the correct position in the newly sorted vertex list by creating an *old_index-to-new_index* lookup table. The table is constructed by reading the sorted vertex list and storing a vertex's new index in the table at the vertex's saved original location.

Figure 7 shows graphically what it means to group triangles in the shared address space according to spatial locality. With a sorted mesh layout, neighboring rays are more likely to find the data they need within an already referenced page and throughout the lower levels of the memory hierarchy.

We analyze the effectiveness of the sorting routine with another experimental test. In this case, we place mesh data and a hierarchical grid acceleration structure into the PDSM memory space. This test uses Stanford's *Lucy* model [21], which has 14027872 vertices and 28055742 triangles. We selected the finest resolution acceleration structure that would fit within the PDSM memory space (3 hierarchical levels, 5 cells per level), because it is the most selective and yields the best performance. The original and sorted *Lucy* meshes each consume 481.6 MB, and the hierarchical grid of each mesh consumes 1149.1 MB. The total data size in each case was 1630.7 MB.

In the test, we compare the frame rate of a recorded interactive session consisting of a series of camera motions around the model. We run the test with the original and sorted meshes, gradually decreasing the available memory size by reducing the local cache parameter of our PDSM. This simulates the anticipated memory configuration when rendering large models on a 64-bit architecture in which the physical memory size is likely to be much smaller than

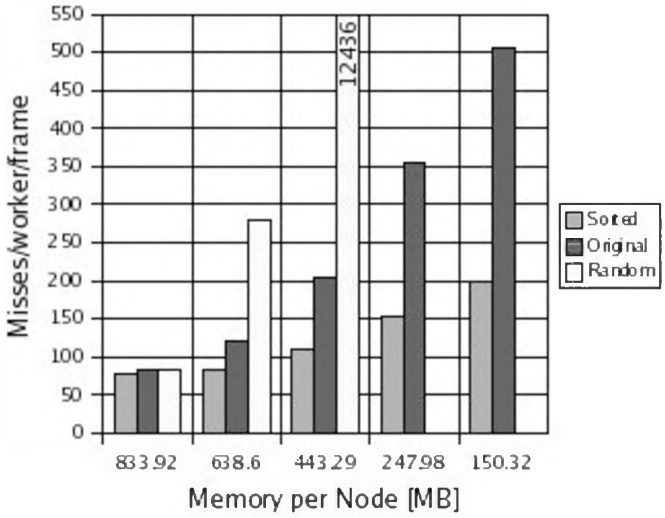


(a) (b)

(c) (d)

Fig. 7. Improving Coherence via Data Reorganization. On the left, the first triangle in the mesh is colored white and the last black. On the right, pages of triangles in the DSM space are colored to identify node ownership. In (a), the input mesh exhibits regions where nearby triangles are far apart in address space. In (b), the pages take the form of thin strips. In (c) and (d), the sorted mesh exhibits fewer address discontinuities, and pages of memory now form patches on the surface.

the virtual memory size. To demonstrate the importance of memory locality in general, we also repeat the test with a randomized mesh, where the triangles and vertices are placed randomly into memory. Figure 8 shows the results of the test. Initially, all three meshes run at approximately the same speed. As the memory becomes more restricted, the randomized mesh performance quickly degrades due to thrashing while our sorted mesh results in the best performance.



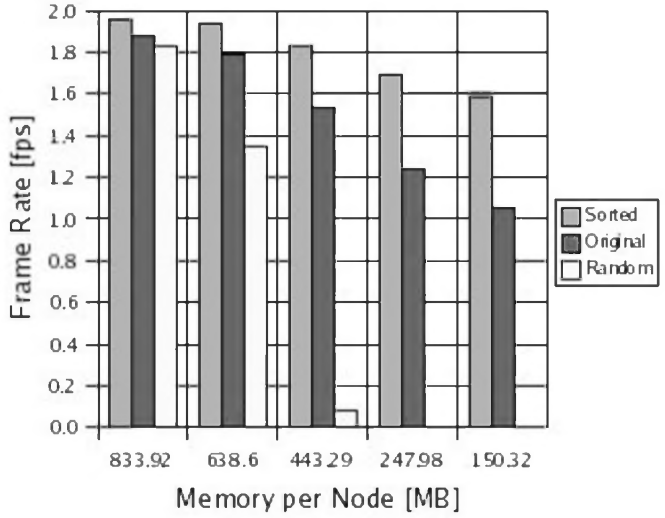


Fig. 8. Mesh Reorganization Results. As the memory space available for rendering falls, the improved coherency in the sorted mesh produces fewer misses (top) and, as a result, higher frame rates (bottom).

4.4 Distributed Load Balancing

We have also experimented with two types of load balancing in our parallel renderer: a centralized task queue and distributed load balancing. In the centralized task queue, the supervisor node maintains a work queue, and workers implicitly request new tiles from the supervisor when they return completed assignments. In the distributed load balancer, the workers instead obtain tile assignments from each other.

This centralized design worked well in the original hardware shared memory implementation of the ray tracer. However, the higher network latency and slower shared memory access times in the cluster have introduced performance penalties for this load balancing scheme. For example, although the central work queue quickly achieves a well-balanced workload, it results in poor memory coherence because tile assignments are essentially random and change every frame. In a cluster with severe memory access miss penalties, there is an interesting trade-off between a more balanced workload and higher hit rates.

With our distributed load balancer, each ray tracing thread starts frame t with the assignments it completed in frame $t-1$. This pseudo-static assignment scheme increases hit rates because the data used to render frame $t-1$ will likely be needed when rendering frame t. The goal of this approach is similar to the scheduling heuristic described by Wald et al. [8].

The distributed load balancer uses a combination of receiver- and sender-initiated task migration in an attempt to prevent the load from becoming unbalanced when the scene or viewpoint changes. Once a worker finishes its assignments for a given frame, it picks a peer at random and requests more work. If that peer has work available, it responds. To improve the rate of convergence toward a balanced load, heavily loaded workers can also release work without being queried. In our current implementation, for example, the node that required the most time to complete its assignments will send a task to a randomly selected peer at the beginning of the next frame.

Figure 9 contains difference images between two successive frames in a test session run under each load balancer. The difference images show the tiles that change ownership between the frames on each run. With the work stealing load balancer, very few tiles were rendered by different nodes in the second frame, as is typical in our experience.

We now analyze the extent to which decentralized load balancing improves performance. For this test, we rendered two more of Stanford's scanned meshes, the dragon and bunny models, which together contain 1.2 million triangles and 0.6 million vertices. The total size of the mesh data is 28.76 MB, and we access the data through a highly efficient hierarchical grid acceleration structure that is 187.7 MB in size. As before, we vary the local cache size, this time analyzing how each load balancing algorithm impacts the caching performance.

Figure 10 shows the results. As memory becomes restricted, the work stealing scheme maintains interactivity better because it is able to reuse cached data more often and yields fewer misses. However, when memory is plentiful, either approach works well. In practice, it is the trade-off between increased load imbalance and improved hit rates that determines which option will perform

Fig. 9. Comparing Task Assignment Strategies. Difference images computed from two subsequent frames. Results from the demand driven load balancer are on the left, work stealing results are on the right. The task stealing heuristic changes tile assignments much less frequently.

better. However, as the memory constraints become more restrictive, the work stealing load balancer tends to perform better.

A decentralized scheme also eliminates a synchronization bottleneck at the supervisor that is amplified by the network transmission delay. Unless frameless rendering is used, a frame cannot be completed until all image tiles have been assigned and returned. Asynchronous task assignment can hide the problem, but as processors are added, message start-up costs will determine the minimum rendering time. When this happens, the rendering time is at least the product of the message latency and twice the number of task assignments in a frame. For 512x512 images composed from 16x16 tiles the maximum achievable frame rate is 34 frames per second on our cluster.

Work stealing eliminates all task assignment messages from the supervisor and allows workers to assign tasks independently. In other words a decentralized task assignment scheme takes advantage of the fact that on a switch-based network nodes B and C can communicate at the same time as nodes D and E. When the system is network bound, this approach can increase the achievable frame rate by a factor of two.

5 Scalability Analysis

In a renderer designed for interactive visualization of large scientific datasets, it is important to understand the processor and data scaling behavior. That is, given a constant data size, we want to know if it can be rendered more quickly by using more processors. We also want to know how the rendering speed changes with a fixed number of processors as the size of the dataset

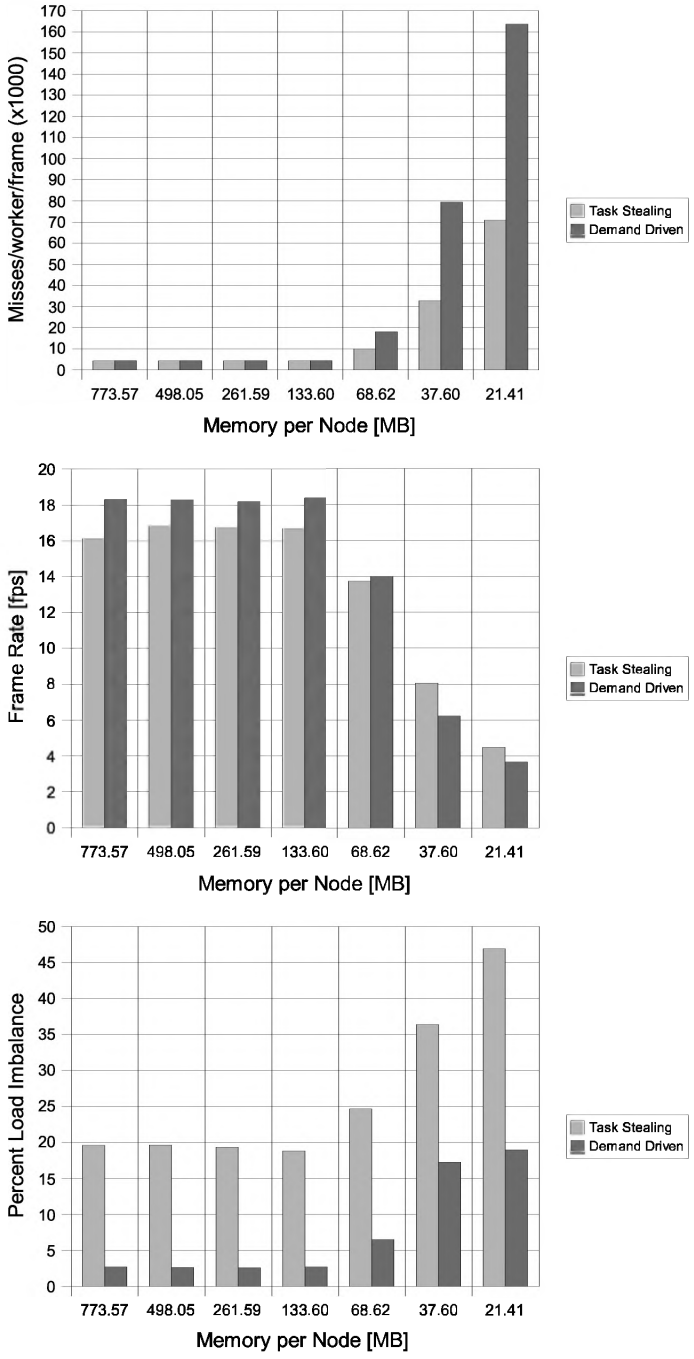


Fig. 10. Effect of Task Reuse with Limited Local Memory. Each node exhibits fewer misses when computing pixels from previous tasks (top). As a result, work stealing improves frame rates when the local memory is limited (middle), despite exhibiting a more imbalanced workload (bottom).

increases.

In an image parallel ray tracer, one can expect that when the program is compute-bound, because there are many independent pixels in the image,

processor scaling behavior will be quite good. Because of hierarchical acceleration structures, we can also expect that data scaling behavior will be good, as these structures turn the ray tracing algorithm into an $\mathcal{O}(P \log N)$ sorted search problem.

In practice, when rendering small data sets that can be replicated in each node's memory we have found that these scaling characteristics accurately model our program's behavior. The application tends to scale nearly linearly until computation time falls below network communication time, at which point we can no longer overlap computation and communication. When enough processors are applied to overcome the computational bottleneck, image tile communication time eventually becomes the limiting factor as described in Section 4.4.

When rendering large amounts of data, it is more often the case that memory access times are the most significant bottleneck. For example, if a node misses in the cache and needs to fetch remote data on average 100 times per frame, and each one of these requests takes 1 ms, then one can not expect to reach more than 10 frames per second.

The number of data accesses and the ratio of local to remote memory references is dependent on the number of workers and on the size of the data. To examine the behavior of our system in practice, we perform another benchmark, in this case rendering the volume shown in Figure 11. This volume is time step 225 of a Richtmyer-Meshkov instability dataset from Lawrence Livermore National Laboratory. The complicated surfaces in this dataset make the renderer processor- and memory-bound well before it becomes network-bound.

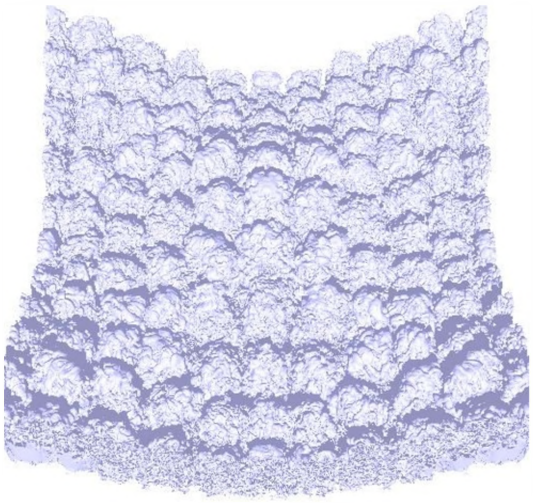


Fig. 11. Richtmyer-Meshkov Instability. The computational fluid dynamics dataset used for the scalability tests.

UU IR Author Manuscript

We measure processor scaling performance by varying the number of rendering nodes. We measure memory scaling performance by down-sampling the original 7.5 GB volume repeatedly, and recording the running time over an interactive session. To make a fair comparison, we place all volumes in the ODSM space despite the fact that the smaller volumes can be rendered more quickly in the PDSM space or by replication. The result of the scaling tests are given in Figure 12, and a selection of memory access measurements are given in Table 3.

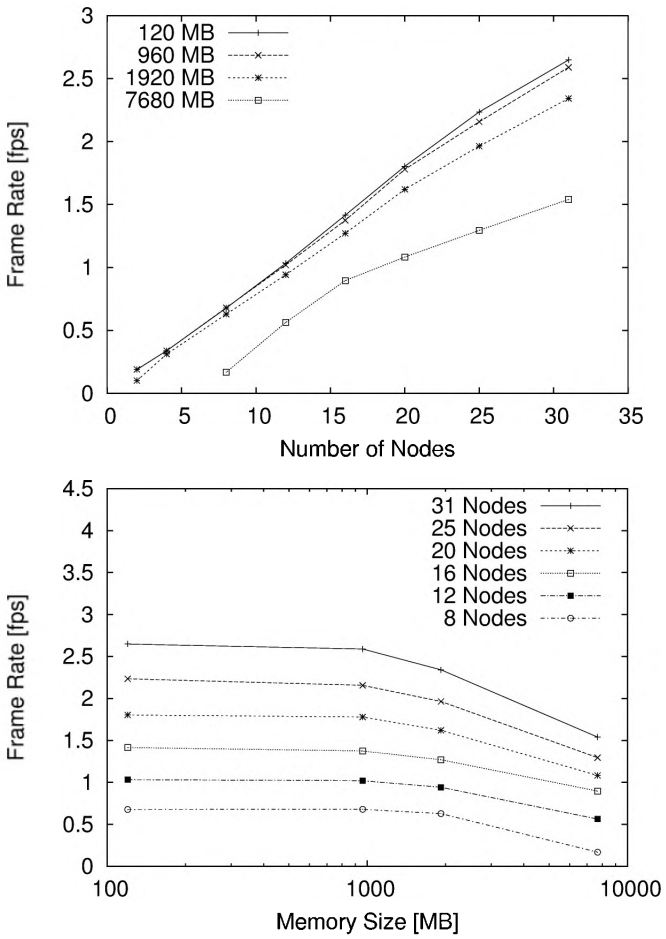


Fig. 12. Processor and Memory Scaling Behavior.

As the number of nodes increases, the data sets that are small enough to be cached entirely in the physical memory of each node exhibit close to ideal scaling. The largest datasets cannot be cached entirely on any one node which makes scaling more complicated.

With large data, as the number of nodes increases, the dominant component of runtime change is a decrease in the number of accesses per node. This happens because increasing the number of workers decreases the amount of the screen that is rendered on any one node. Meanwhile, for the large datasets the number of misses drops substantially, as more of each node's memory is

		120 MB	960 MB	8 GB
8 nodes	accesses	265211	245444	221667
	misses	2.16	14.7	5933
16 nodes	accesses	132605	123250	110903
	misses	2.31	15.7	529
31 nodes	accesses	68600	63611	57189
	misses	2.38	16.1	251

Table 3
Selected Rendering Statistics. Average number of memory accesses and DSM cache level misses per worker per frame recorded in nine of the test sessions.

available to use as a cache. Overall the trend is still towards linear scaling, with a temporary benefit due to the caching and memory access factors.

From the memory scaling figure it is clear that the macrocell hierarchy gives us very good data scaling behavior. Holding the number of nodes constant, multiplying the data size by powers of 8 decreases the frame rate very little. The hierarchy keeps the number of accesses roughly constant as the data size explodes. Unfortunately increasing the data size does increase the miss rate, which makes the rendering time more influenced by the long miss penalty. If the local memory is insufficient to hold the working set, as is the case for 8 nodes on the 8 GB data, thrashing results and the miss penalty dominates.

6 Conclusions

We have found that it is possible to render large datasets quickly using readily available cluster technology. Our solution adds a top-level memory layer in which all cluster nodes share their local memory contents via the network. Our shared memory layer can use either an object-based or page-based organization. The object-based layer makes the aggregate physical memory space of the cluster available to all rendering threads. On 32-bit clusters, the page-based layer is more limited in terms of addressable data size, but it adds less overhead to the cost of purely local data access.

With our shared memory, we are able to ray trace scenes that are too large to be replicated in each node's memory. Ray tracing is the classic image parallel rendering algorithm. All pixels can be computed concurrently so it usually exhibits very good processor scalability. The use of acceleration structure hierarchies allows us to consider only a fraction of the elements of the scene when rendering an image, which gives the program good memory scalability.

The miss penalty in a network shared memory system using commodity interconnection hardware is quite high. For this reason caching is an essential component of our network memory system. We have discussed several memory access optimizations that we have used, all of which increase the percentage of cache hits. The optimizations extend the memory size to which we can achieve interactive rendering.

7 Future Work

Higher performing interconnect architectures are becoming widely available. Both Myrinet and Infiniband, for example, reduce message latency and increase network bandwidth substantially. We have recently adapted our system to make use of MPI to allow us to take advantage of these networks and increase our scalability. Our preliminary analysis has found that our asynchronous message handling makes a thread-safe MPI layer of tantamount importance. Lacking such a layer, efficiency-reducing thread barriers are required. These barriers usually negate any performance improvement that the high performance interconnect may yield.

For some datasets, the total physical memory space of the cluster is not sufficient. Terabyte-scale, time varying volumetric datasets are an example. We are currently working to render from the combined disk space of all of the nodes in our cluster by fetching scene contents on demand from disk. We plan to use the network memory discussed here as an intermediate level in the memory hierarchy, inserted before the final disk-mapped layer. Preliminary tests have shown that such a system is possible, and we are currently studying further optimizations, such as those described by Corrêa [22], for this new memory organization.

Additional efficiency may be available if we can better exploit the processing elements in more recent PC architectures. With increasingly available SMP and simultaneous multithreading capable nodes, it is important to use hybrid parallel architectures efficiently. For this reason, we plan to address the lack of thread safety in our PDSM networked memory layer and to test it on 64-bit architectures.

References

[1] S. Molnar, M. Cox, D. Ellsworth, H. Fuchs, A sorting classification of parallel rendering, IEEE Comput. Graph. Appl. 14 (4) (1994) 23–32.

[2] S. Parker, W. Martin, P.-P. Sloan, P. Shirley, B. Smits, C. Hansen, Interactive ray tracing, Interactive 3D Graphics (I3D) (1999) 119–126.

[3] J. B. Carter, D. Khandekar, L. Kamb, Distributed shared memory: Where we are and where we should be headed, in: Fifth Workshop on Hot Topics in Operating Systems (HotOS-V), 1995, pp. 119–122.

[4] B. N. Bershad, M. J. Zekauskas, Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors, Tech. Rep. CMU-CS-91-170, Carnegie Mellon University, Pittsburgh, PA (USA) (1991).

[5] B. Corrie, P. Mackerras, Parallel volume rendering and data coherence, in: ACM SIGGRAPH 93 Symposium on Parallel Rendering, ACM, ACM Press / ACM SIGGRAPH, 1993, pp. 23–26.

[6] D. Badouel, K. Bouatouch, T. Priol, Distributing data and control for ray tracing in parallel, IEEE Computer Graphics and Applications 14 (4) (1994) 69–77.

[7] I. Wald, P. Slusallek, State-of-the-Art in Interactive Ray-Tracing, State of the Art Reports, EUROGRAPHICS 2001 (2001) 21–42.

[8] I. Wald, P. Slusallek, C. Benthin, Interactive distributed ray tracing of highly complex models, in: 12th Eurographics Workshop on Rendering, 2001, pp. 277–288.

[9] I. Wald, A. Dietrich, P. Slusallek, An interactive out-of-core rendering framework for visualizing massively complex models, in: Rendering Techniques 2004, Proceedings of the Eurographics Symposium on Rendering, 2004, pp. 81–92.

[10] T. Kato, J. Saito, "Kilauea" – parallel global illumination renderer, in: D. Bartz, X. Pueyo, E. Reinhard (Eds.), Proceedings of the Eurographics Workshop on Parallel Graphics and Visualization, Eurographics, 2002, pp. 7–16.

[11] E. Reinhard, F. W. Jansen, Rendering large scenes using parallel ray tracing, Parallel Computing 23 (7) (1997) 873–885.

[12] D. E. DeMarle, Ice network library download page.
URL `http://www.cs.utah.edu/ demarle/software/`

[13] S. Parker, M. Parker, Y. Livnat, P.-P. Sloan, C. Hansen, P. Shirley, Interactive ray tracing for volume visualization, IEEE Transactions on Visualization and Computer Graphics 5 (3) (1999) 238–250.

[14] D. E. DeMarle, S. Parker, M. Hartner, C. Gribble, C. Hansen, Distributed interactive ray tracing for large volume visualization, in: IEEE Symposium on Parallel and Large-Data Visualization and Graphics, 2003, pp. 87–94.

[15] S. Rubin, T. Whitted, A three dimensional representation for fast rendering of complex scenes, Computer Graphics 14 (3) (1980) 110–116.

[16] A. Fujimoto, T. Tanaka, K. Iwata, ARTS: Accelerated ray tracing system, IEEE Computer Graphics and Applications 6 (1986) 16–26.

[17] M. Cox, D. Ellsworth, Application-controlled demand paging for out-of-core visualization, in: Proceedings of IEEE Visualization, IEEE, 1997, pp. 235–244.

[18] M. Pharr, C. Kolb, R. Gershbein, P. Hanrahan, Rendering complex scenes with memory-coherent ray tracing, Computer Graphics 31 (Annual Conference Series) (1997) 101–108.

[19] H. Hoppe, Optimization of mesh locality for transparent vertex caching, in: Proceedings of the 26th annual conference on Computer graphics and interactive techniques, ACM Press/Addison-Wesley Publishing Co., 1999, pp. 269–276.

[20] M. Isenburg, P. Lindstrom, Streaming meshes, Tech. Rep. UCRL-CONF-201992, Lawrence Livermore National Lab (Apr. 2004).

[21] The stanford 3d scanning repository.
URL `http://graphics.stanford.edu/data/3Dscanrep/`

[22] W. T. Corrêa, New techniques for out-of-core visualization of large datasets, Ph.D. thesis, Princeton University (2004).