Eurographics Symposium on Parallel Graphics and Visualization (2004)
Dirk Bartz, Bruno Raffin and Han-Wei Shen (Editors)

Memory-Savvy Distributed Interactive Ray Tracing

David E. DeMarle, Christiaan P. Gribble, and Steven G. Parker †

Scientific Computing and Imaging Institute
University of Utah

Abstract

Interactive ray tracing in a cluster environment requires paying close attention to the constraints of a loosely coupled distributed system. To render large scenes interactively, memory limits and network latency must be addressed efficiently. In this paper, we improve previous systems by moving to a page-based distributed shared memory layer, resulting in faster and easier access to a shared memory space. The technique is designed to take advantage of the large virtual memory space provided by 64-bit machines. We also examine task reuse through decentralized load balancing and primitive reorganization to complement the shared memory system. These techniques improve memory coherence and are valuable when physical memory is limited.

Categories and Subject Descriptors (according to ACM CCS): I.3.2 [Computer Graphics]: Graphics Systems–Distributed/network graphics I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Ray tracing

1. Introduction

Datasets are growing in size at an alarming rate. Typical datasets, including regular volumetric data, tetrahedral volumes, surface models, and textures, are often hundreds of megabytes or tens of gigabytes in size and are much larger than the capacity of the physical memory in most uniprocessor machines. Parallel computers allow us to solve this problem because their processing power is great enough to render the data and because their memory resources are plentiful enough to hold the data. Amdahl's law, which implies that any amount of sequential processing limits parallel computational scaling, encourages us to take advantage of the second feature of parallel computing and make the best possible use of the parallel memory resources.

It is well known that ray tracing is trivially parallel. With the huge number of data accesses incurred by ray tracing, however, memory access delays quickly become the indivisible portion of the rendering time and limit achievable scalability when visualizing large, shared datasets. To reduce this bottleneck, we apply a page-based distributed shared memory (PDSM) system to interactive ray tracing.

In our distributed interactive ray tracing system, each node in the cluster manages different pieces of the scene data and reserves some local space to cache remote pieces. Our renderer accesses data through the regular address space by employing operating systems services, particularly virtual memory. Accessing distributed memory through the virtual memory system is beneficial because the shared memory software layer intervenes only when a missing page is referenced. Figure 1 presents a high-level view of the process by which render threads access scene data.

The page-based approach has the potential to allow clusters of inexpensive machines to render large datasets quickly, even when each node has a physical memory that can store only a fraction of the total data. This point is important because, while next generation 64-bit machines are able to address tremendous amounts of memory, the cost of that memory is a limiting factor in the quantity available on each node. Efforts to employ the total sum of the memory in an efficient and cost-effective manner are valuable.

Our new system has advantages over previous systems because memory access is fully decentralized, does not rely on disk access, and has a natural programming interface. Shared memory hits incur no overhead, so when the working set is reasonably bounded and changes gradually between frames, the system operates at nearly the same speed as if the data had been replicated on each node.

† {demarle|cgribble|sparker}@sci.utah.edu

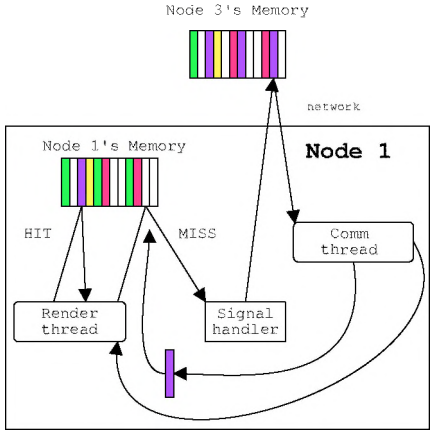


Figure 1: *Basic Page-Based DSM Architecture. The virtual memory hardware detects misses, and the PDSM layer causes remote references to be paged from across the network.*

Two additional changes to our renderer attempt to capitalize on this advantage. Both seek to increase hit rates, helping to alleviate the large miss penalties and take advantage of low hit times. The first change is to move from a centralized demand driven load balancing scheme, in which renderers obtain work from the supervisor, to a decentralized work stealing scheme. Work stealing helps reduce the variation in tile assignments between frames, enabling previously cached data to be reused.

The second change seeks to improve hit rates by reorganizing the scene data to improve coherence. The key observation is that spatially local primitives should be stored together in memory as well. Similar to data bricking for volumetric datasets [PSL*98, CE97], this technique can improve rendering rates by increasing the probability that a page of memory will be reused once it has been loaded.

2. Related Work

DeMarle et al. [DPH*03] use an object-based distributed shared memory (ODSM) to render large volumetric datasets. Distributed access to scene data in a switched network provides data at the combined bandwidth of all of the machines in the cluster. Badouel et al. [BBP94] achieved a similar effect with a page-based distributed shared memory. Quarks [CKK95] and Adsmith [LKL97] are representative examples of full featured page- and object-based DSMs. In this paper we compare the performance of both types of DSM layers for distributed interactive ray tracing.

Wald et al. [WSB01] have explored coherent ray tracing techniques in the distributed environment. They address the challenge of rendering large, complex models interactively by combining centralized data access and client-side caching of geometry voxels. They take pains to exploit spatial coherence within BSP tree nodes and temporal coherence between subsequent frames. In their system, both tile assignments and data retrieval go through central servers. In this work we parallelize these functions to eliminate the central bottlenecks.

Many authors have considered load balancing for parallel ray tracing. For example, Heirich et al. [HA98] discuss the necessity of dynamic load balancing for ray tracing in an interactive setting and describe a scheme based on a diffusion model. Reinhard et al. [RCJ99] present an advanced hybrid load balancing scheme in which both objects and rays can be transfered between processing elements. The data parallel tasks allow their system to render large and complex scenes efficiently while the demand driven tasks, consisting of coherent ray packets, balance the load more evenly. A complete discussion of load balancing techniques for parallel rendering can be found in [CDR02]. We focus on decentralized load balancing with work stealing as a simple and effective method to balance the load while improving memory and network performance characteristics.

As the gap between processor speeds and memory and network speeds continues to widen, data locality becomes increasingly critical to rendering performance. Pharr et al. [PKGH97] use caching techniques to manage model complexity in an off-line rendering process. They complement lazy data loading with data reorganization in a geometry cache. The reorganization ties spatial locality in the three-dimensional space of the scene to locality in memory so that expensive disk access times can be amortized. Cox et al. [CE97] apply similar methods for scientific visualization of large datasets. The data reorganization technique we describe targets the same goal, and in this paper, we examine its effect in a distributed shared memory environment.

3. Page-Based Distributed Shared Memory

In [DPH*03], we presented a solution to the memory problem in which a C++ dataserver object obtains data from remote nodes when requested by one or more render threads. With a well-constructed acceleration structure, volume bricking, caching, and strict attention to efficient accesses to the dataserver, more than 95% of data accesses resulted in hits. Such a high hit rate allowed the program to produce interactive visualizations of multi-gigabyte datasets, despite the fact that miss penalties were on the order of 1000 μs.

The high hit rates in our renderer make the hit times a tempting target for optimization. In an ODSM system, every memory access must go through expensive access tests, which are performed in software, to find a block and ensure that it is available for the duration of its use. In our new system, the virtual memory hardware of the machine handles all memory accesses and a signal handler processes only the

exceptional case of a cache miss. The DSM system requires no kernel modifications because it is implemented entirely in user-space using standard Unix system calls. Because the scenes are read-only during rendering, the DSM system does not implement page invalidation, is not prone to false sharing inefficiencies and does not require complex consistency mechanisms.

The distributed memory space now occupies a reserved range of virtual memory addresses. As before, each node assumes ownership of different stripes of the shared memory space and populates them with data at program initialization. The remainder of the shared address range is initially empty and unmapped.

During execution, render threads generate a segmentation fault signal every time they access an unmapped address. A registered handler catches the signal and retrieves the faulting address. The handler finds the node that owns the page of memory in which the faulting address resides, issues a request to the owner, and suspends the render thread.

When the owner responds with the page, the communication thread first checks the number of cached pages. If there are no open slots, a page is selected for eviction to reclaim space. Currently a random page replacement policy is used because of the difficulty of implementing a more sophisticated algorithm in user-space. Next, the communication thread receives the sent data directly into a newly created page residing at the requested address and wakes the render thread, which continues forward.

One strength of a PDSM is that data accesses do not require special handling in the application. The application need not distinguish between items that lie in shared space and those that do not. Programs that use large amounts of memory can be developed in a similar fashion to those that employ shared memory hardware. This ease of use makes it feasible to render large scenes composed of any object type. Additionally, placing the acceleration structure in the shared memory space can be beneficial. Although accesses to uncached portions of the structure impose slight penalties, these do not occur frequently in practice. In fact, frequently accessed root level data typically remain loaded while unused branches tend to be pruned away.

The primary disadvantage of a page-based system is that it is limited to scenes that fit within the address space of the machines on which it runs. For 32-bit machines, the maximum theoretical limit is 4 GB. In practice, however, operating system limits and the need to leave some addresses for other program data make the limit slightly less than 2 GB. Fortunately, the increasing availability of 64-bit machines makes this limitation much less severe.

The core routines of the PDSM software layer are given as pseudocode in Figure 2. In Section 6.1 we analyze the effectiveness of rendering large scenes using the PDSM space.

```
// At program start, this function is
// registered to handle SIGSEGV
void memintercept(siginfo_t* sinfo) {
   void* faulting_addr = sinfo->si_addr;
   int page_num = get_page_num(faulting_addr);
   int owner = get_owner(page_num);
   send_msg(owner, REQUEST_PAGE, page_num);
   page_wait_sem.down();
}

// At program start, this function is
// registered to handle PDSM messages
void handlemessage(int sender, int msgid,
                   int page_num) {
   if (msgid == REQUEST_PAGE) {
      send_msg(sender, SENT_PAGE,
               &page[page_num]);
   } else {
      int destslot = num_pages_loaded;
      if (num_pages_loaded == cachesize) {
         int victim = select_victim_page();
         unmap(&page[victim],sizeof(page));
         destslot = victim;
      } else
         num_pages_loaded++;

      mmap(&(page[destslot]),sizeof(page));
      recv_msg(sender, &page[destslot]);
      page_wait_sem.up();
   }
}
```

Figure 2: *Page-Based DSM Core Functionality. These two functions implement the distributed shared memory.* `memintercept` *is invoked when a render thread accesses memory that is unmapped, and* `handlemessage` *reacts to the resulting network messages.*

4. Decentralized Load Balancing

Even with faster access to shared scene data, each miss inflicts hundreds of microseconds of delay. The miss time is largely a function of the network characteristics and is not easily reduced. In an interactive rendering system, then, it is critical to reduce the number of misses. Toward this end, we have experimented with a decentralized load balancing scheme that employs work stealing. Our algorithm is a simplification of that in [RSAU91].

As in our previous system, rendering tasks are based on an image-space division because primary rays can be traced independently. Before, the supervisor node maintained a work queue, and workers implicitly requested new tiles from the supervisor when they returned completed assignments. 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.

With a work stealing load balancer, each render 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. [WSB01].

Our new system uses a combination of receiver- and sender-initiated task migration 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 3 contains diagnostic images showing typical image tile distributions for the original demand driven and the new work stealing algorithms. Note the distribution of tiles in the work stealing image is more regular. In Section 6.2 we analyze the effectiveness of this optimization.

Figure 3: *Comparing Task Assignment Strategies. Tiles rendered by each node have unique gray-levels. On the left, tasks constantly change with demand driven assignment. On the right, assignments are more stable with work stealing, allowing workers to reuse locally cached data more often.*

5. Address Sorting

The problem of accessing data is acute in network memory and out-of-core systems, where the access time to missing memory is high. To decrease the number of misses, we carefully organize the layout of scene data in memory. If primitives located together in three-dimensional space can be rearranged so they are also located together in address space, pages are more likely to be reused. The effect of our technique is similar to that achieved by Pharr et al. [PKGH97].

We reorganize the memory layout of our input data using a preprocessing program. This program reads a mesh file and creates a multi-level grid acceleration structure that groups nearby objects together. To sort the geometry for improved coherence, we traverse the acceleration structure and write the primitives, in order, to a new scene database. Although the input data may contain neighboring triangles p and q that are separated by tens of megabytes of address space, the output data will contain new triangles p' and q' within a few bytes of each other. The preprocessing program takes only a few minutes for the models we tested.

Figure 4 shows graphically what it means to group triangles in the shared address space according to spatial locality. In the figure, all triangles within pages owned by a particular node have identical hues. Figure 4b demonstrates that, without reorganization, neighboring triangles may be placed far apart in memory or owned by different nodes. Figure 4c shows the address alignment of the sorted mesh. With this 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. In Section 6.3 we analyze the effectiveness of this optimization.

6. Results

In this section, we benchmark interactive rendering sessions under varying conditions to analyze the performance benefits of the three techniques we have applied. All scenes have a single light source and include hard shadows. For each test, the images were rendered at a resolution of 512x512 pixels and were divided into 16x16 pixel tiles, except where noted. Our test machine is a 32 node cluster consisting of dual 1.7 GHz Xeon PCs with 1 GB of RAM. The nodes are connected via switched gigabit ethernet. We run a single rendering thread on each node, except where noted. The reported node counts do not include the use of a single display machine.

6.1. Page-Based Distributed Shared Memory Analysis

The first test compares the hit and miss times for the ODSM and PDSM layers that our ray tracer employs when rendering large datasets. Table 1 gives the measured hit and miss penalties for object- and page-based DSMs recorded in a random access test. In the test, one million 16 KB blocks are chosen at random from a 128 MB shared memory space. The access times have been recorded using the *gettimeofday* system call. From this table, it is clear that the hit time of the PDSM system is substantially lower than that of the ODSM system. If the renderer is able to maintain high hit rates, the PDSM layer results in higher frame rates.

Next we compare the performance of the ODSM and PDSM layers in the ray tracer. We render isosurfaces of a 512 MB scalar volume created from a computed tomography scan of a child's toy. In the test, we replay a recorded session in which the viewpoint and isosurface change, causing the working set to vary. Extra work is required to obtain

	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 addresses to a 128 MB address space on five nodes.*

the rendered data when using the DSM systems because we restrict the DSM layers to store only 81 MB on each node.

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 lower hit time of the PDSM allows it to outperform 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.

6.2. Decentralized Load Balancing Analysis

We now analyze the extent to which decentralized load balancing improves performance. For this test, we rendered two of Stanford University's widely available *PLY* format models. In particular, we report results using the models shown in Figure 6. Details of these models and the run-time data structures are given in Table 2.

In these tests, we place the geometry data and a large, highly efficient acceleration structure in PDSM space. By varying the local cache size, we can analyze how both load balancing algorithms impact the performance of the distributed memory system.

Figure 7 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. We note, however, when memory is plentiful, either approach works well.

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 startup costs will determine the minimum rendering time. In this case, the rendering time is at least the product of the message latency and twice the number of task assignments in a frame.

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

To demonstrate, we render a small sphereflake scene consisting of only 827 primitives. To emphasize the effect of work stealing on the supervisor's communication time, the test uses 8x8 pixel tiles and two rendering threads per node.

Figure 6: *The Stanford Bunny and Dragon PLY Models. This is a sample image taken from the interactive session used for benchmarking.*

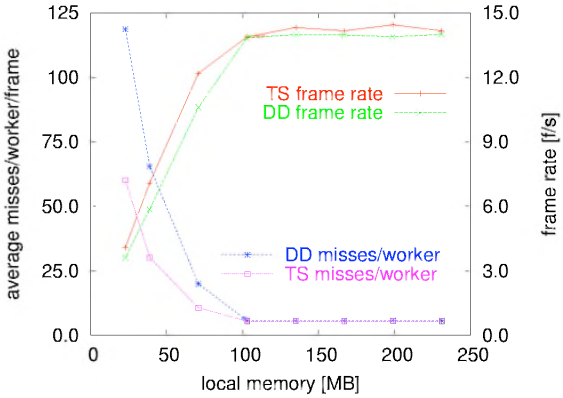


Figure 7: *Effect of Task Reuse with Limited Local Memory. Each node is exhibits fewer misses when reusing previous tasks. As a result, work stealing improves frame rates when the local memory is limited.*

Model	Vertices	Triangles	Prim. Size [MB]	Grid [MB]	Total [MB]	Sorted Triangles	Prim. Size [MB]	Grid [MB]	Total [MB]
Bunny	35947	69451	2.138	9.653	11.79	324635	7.978	8.415	16.39
Dragon	437645	871304	26.62	178.0	204.6	3724385	91.92	163.8	255.7
Total	602045	1202281	28.76	187.7	216.4	7545220	99.90	172.2	272.1

Table 2: *Characteristics of the Stanford PLY models, the preprocessed geometry data, and the acceleration structure.*

Table 3 reports the measured time the supervisor spends communicating, as well as the resulting frame rate. Note that as the number of workers grows, the supervisor's communication time remains constant with work stealing.

# of nodes	2	6	12	18	24	31
Demand Driven						
Comm. time	0.06	0.07	0.08	0.08	0.09	0.09
Frame rate	1.51	4.25	8.18	11.1	12.0	12.3
Work Stealing						
Comm. time	0.06	0.06	0.06	0.06	0.06	0.06
Frame rate	1.62	4.59	8.79	12.6	15.5	17.3

Table 3: *Supervisor Communication Time. In the decentralized approach, the supervisor's communication time remains constant as the number of worker nodes increases. Communication times are given in s/f, frame rates in f/s.*

6.3. Address Sorting Analysis

Our last test examines how sorting spatially local primitives in address space affects performance. In this test, we use the same *PLY* models as before, but we now render the data after preprocessing with the sorting program described in Section 5.

Figure 8 shows that the effect of address sorting is similar to that of decentralized load balancing. Specifically, when the local memory of each node is small compared to the total data size, sorting geometry decreases the number of misses enough to increase the frame rates. However, when the memory size is large enough to contain the working set, sorting does not yeild improved performance. The acceleration structure used for sorting is based on a uniform grid, and as we traverse the structure, triangles that cross cell boundaries are duplicated. It is possible that using a different acceleration structure would reduce this increase in data size.

We have also experimented with dereferencing vertex pointers when sorting the geometry. The sorting process is the same as described earlier, except that vertex pointers are dereferenced and the vertex data is stored with each sorted triangle. Data may be duplicated many times because vertices are often shared by several triangles. The data bloat resulting from this process is substantially higher, and, in general, the achievable frame rates are lower still.

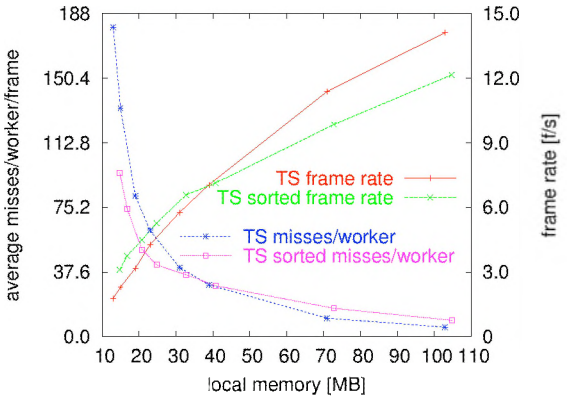


Figure 8: *Effect of Address Sorting with Limited Local Memory. Each node exhibits fewer misses when memory is limited because, with address sorting, spatially local primitives exhibit better memory coherence.*

7. Conclusion and Future Work

We have found that by utilizing virtual memory hardware and associated operating system services to manage a shared memory space, large datasets can be rendered more quickly and easily than with a software-only solution. There are significantly lower memory access penalties, and the programming task of using the shared memory is also reduced. These benefits make it possible to render large amounts of almost any type of data.

Complementary to the PDSM is a distributed load balancing mechanism that improves cache hit rates and helps overcome the network transmission latency barrier. Hit rates can be improved further by sorting the scene data in address space so that spatially local data exhibits improved memory coherence. Both techniques are most useful when memory available on each node is limited. Higher quality decentralized load balancing heuristics and improved sorting algorithms that reduce data replication are left as future work.

All three optimizations should be valuable in the context of 64-bit clusters, where the virtual address space will likely be substantially larger than the physical memory of any one node. Techniques like ours will enable interactive visualization of very large datasets with these clusters. We plan to test our implementation on a 64-bit cluster in the near future.

A disadvantage of the user-space PDSM is that it is difficult to create a page-based memory that is usable by multiple rendering threads. A race condition exists whenever the communication thread fills a received page of data. Rendering threads must be prevented from accessing the invalid page during this time. To overcome this limitation, we have begun experiments with asynchronous signaling to temporarily suspend all rendering threads during page handling. When threads are stalled, however, efficiency drops. More critically, all rendering code must be asynchronous signal-safe for this approach to work. Preliminary testing is currently underway.

Similar drops in efficiency result because render threads are suspended while waiting for previously unmapped pages to arrive. Rescheduling rays that cause segmentation faults and allowing the render thread to trace other rays may eliminate this problem. PDSM performance may also be improved with a better page replacement policy. Finally, a read-write PDSM implementation will be required for rendering most dynamic scenes.

Two other potential targets for optimization are the centralized result gathering phase of the computation and the size of data transfers. Decentralized task assignment greatly reduces the number of messages the supervisor must handle, but we have not yet examined the delay incurred because each rendered tile must be returned through the same bottleneck. In addition, results by Wald et al. [WSB01] have shown that on-the-fly compression for data transmitted over the network can reduce access penalties. We would like to investigate similar techniques in this page-based system.

Acknowledgments

This work has been sponsored in part by the National Science Foundation under grants 9977218 and 9978099, by DOE VIEWS and by NIH grants. The authors thank Anthony Davis from HyTec, Inc. and Bill Ward and Patrick McCormick at Los Alamos National Labs for the furby dataset.

References

[BBP94] BADOUEL D., BOUATOUCH K., PRIOL T.: Distributing data and control for ray tracing in parallel. *IEEE Computer Graphics and Applications 14*, 4 (1994), 69–77. 2

[CDR02] CHALMERS A., DAVIS T., REINHARD E.: *Practical Parallel Rendering*. AK Peters Publishing, Nantick Massachusetts, 2002. 2

[CE97] COX M., ELLSWORTH D.: Application-controlled demand paging for out-of-core visualization. In *Proceedings of IEEE Visualization* (1997), pp. 235–244. 2

[CKK95] CARTER J. B., KHANDEKAR D., KAMB L.: 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. 2

[DPH*03] DEMARLE D. E., PARKER S., HARTNER M., GRIBBLE C., HANSEN C.: Distributed interactive ray tracing for large volume visualization. In *IEEE Symposium on Parallel and Large-Data Visualization and Graphics* (Oct. 2003), pp. 87–94. 2

[HA98] HEIRICH A., ARVO J.: A competative analysis of load balancing strategies for parallel ray tracing. *The Journal of Supercomputing 12*, 1–2 (1998), 57–68. 2

[LKL97] LIANG W.-Y., KING C.-T., LAI F.: Adsmith: An object-based distributed shared memory system for networks of workstations. *IEICE Transactions on Information and Systems E80-D*, 9 (1997), 899–908. 2

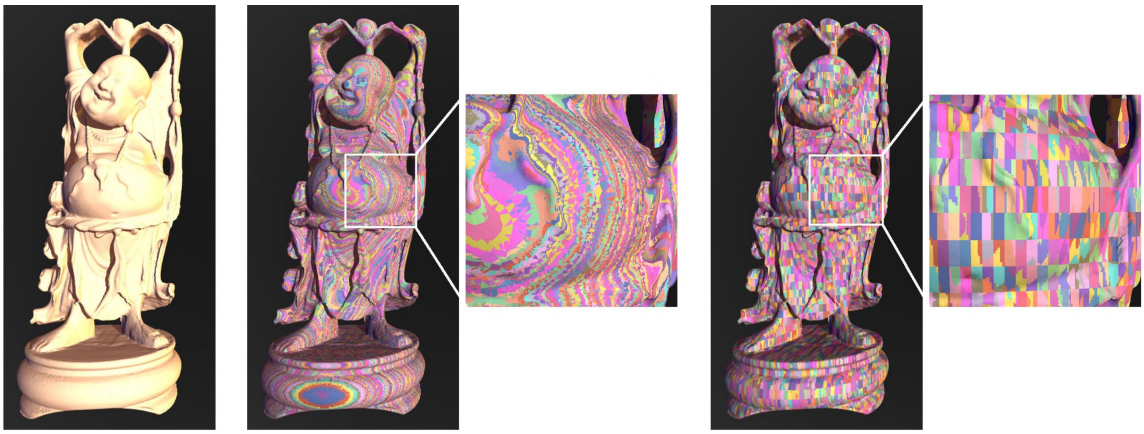
[PKGH97] PHARR M., KOLB C., GERSHBEIN R., HANRAHAN P.: Rendering complex scenes with memory-coherent ray tracing. *Computer Graphics 31*, Annual Conference Series (1997), 101–108. 2, 4

[PSL*98] PARKER S., SHIRLEY P., LIVNAT Y., HANSEN C., SLOAN P.-P.: Interactive ray tracing for isosurface rendering. In *Proceedings of IEEE Visualization* (Oct. 1998), pp. 233–238. 2

[RCJ99] REINHARD E., CHALMERS A., JANSEN F. W.: Hybrid scheduling for parallel rendering using coherent ray tasks. In *IEEE Symposium on Parallel Visualization and Graphics* (1999), ACM Press, pp. 21–28. 2

[RSAU91] RUDOLPH L., SLIVKIN-ALLALOUF M., UPFAL E.: A simple load balancing scheme for task allocation in parallel machines. In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures* (1991), ACM Press, pp. 237–245. 3

[WSB01] WALD I., SLUSALLEK P., BENTHIN C.: Interactive distributed ray tracing of highly complex models. In *12th Eurographics Workshop on Rendering* (June 2001), pp. 277–288. 2, 4, 7



(a) (b) (c)

Figure 4: *Improving Coherence via Data Reorganization. In (a), a standard rendering of the happy buddha PLY model. In (b), the input mesh with each node's triangles shown in a different hue. In (c), the reorganized mesh in which neighboring triangles are more likely to reside in the same page of memory.*

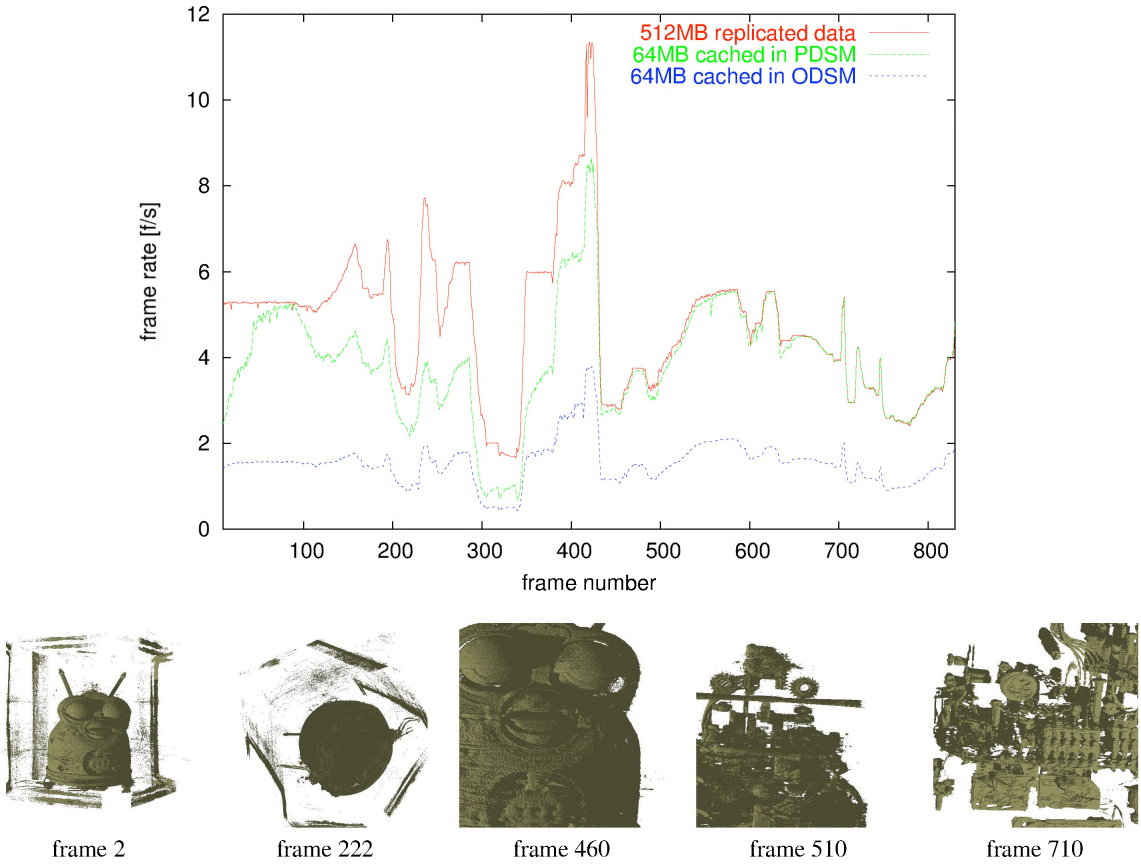


Figure 5: *Comparing Memory Organization. Frame rates are above and images from the test are below. The page-based DSM outperforms the object-base DSM in all cases. Moreover, its performance is competitive with full data replication, even though the local memory size is reduced to 16% of the total.*