Eurographics Symposium on Parallel Graphics and Visualization (2011)
T. Kuhlen, R. Pajarola, and K. Zhou (Editors)

Real-Time Ray Tracer for Visualizing Massive Models on a Cluster

Thiago Ize[1], Carson Brownlee[1,2], and Charles D. Hansen[1,2]

[1]SCI Institute, University of Utah
[2]School of Computing, University of Utah

Abstract

We present a state of the art read-only distributed shared memory (DSM) ray tracer capable of fully utilizing modern cluster hardware to render massive out-of-core polygonal models at real-time frame rates. Achieving this required adapting a state of the art packetized BVH acceleration structure for use with DSM and modifying the mesh and BVH data layouts to minimize communication costs. Furthermore, several design decisions and optimizations were made to take advantage of InfiniBand interconnects and multi-core machines.

1. Introduction

While the core ray tracing algorithm might be embarrassingly parallel, scaling a ray tracer to render millions of pixels at real-time frame rates on a cluster remains challenging; more so if the individual nodes do not have enough memory to contain all the data and associated acceleration structures. The ability to render high-resolution images at interactive or real-time rates is important when visualizing datasets that contain information at the subpixel level. Since most commodity monitors are now capable of displaying at least a 2M pixel HD resolution of 1920×1080, and higher end models can do up to twice as many pixels, it is important to make use of all those pixels when visualizing a dataset. Current distributed ray tracing systems are not able to achieve this regardless of how many compute nodes are used. Either lower resolutions are used to get real-time rates, or less fluid frame rates are used in order to scale to larger image sizes. We show a distributed ray tracing system that can scale on an InfiniBand cluster to real-time rates of slightly over 100fps at full HD resolution, or 50–60fps at 4M pixels with massive polygonal models.

We are also able to handle massive out-of-core models that cannot reside inside the physical memory of any individual compute node by using a read-only distributed shared memory (DSM) ray tracer [CM93]. Using DSM allows us to still use any desired ray tracing shading models, such as shadows, transparent surfaces, ambient occlusion and even full path tracing. More advanced shading models than simple ray casting or rasterization allow for more productive and useful visualizations [GP06].

2. Background

Wald et al. [WSB01] built a ray tracing cluster for out-of-core scenes using a two level kd-tree where each client node has a copy of the top level and the bottom levels are fetched as required from a server and stored in a client-side cache. This requires the server to have enough memory to hold the entire model, or in their case to have a faster hard drive than the clients, to make it advantageous over simply replicating the data. Furthermore, this can quickly saturate the bandwidth to the server.

Significant work on DSM interactive ray tracing was performed by DeMarle et al. [DGBP05] and we expand on several of their ideas. We use their object-based distributed shared memory interface where each node contains parts of the data in resident memory. Additional data is fetched from the node that owns the data and stored in a direct mapped cache for future use. Access to each block is guarded by a block specific counting semaphore which permits multiple threads to read from that cache block, but only allows modifying the cache block when a single thread has sole access to it. We do not use their page-based DSM method because it is not thread-safe and using only a single thread to render when we have 8 cores per node would result in an unacceptable performance hit. We also improve their cache indexing scheme to make better use of the cache. Their cluster consisted of 32 dual-core nodes connected with gigabit Ethernet. One core was used for rendering and the other for communication. We, on the other hand, need to scale to more nodes, each of which have many cores, and all cores are used for rendering, with

DOI: 10.2312/EGPGV/EGPGV11/061-069

each thread being able to communicate as needed. We also have an InfiniBand interconnect which has much lower latency, higher bandwidth, and allows for very efficient RDMA reads and writes of remote memory without involving the CPU. They used a single-ray traversal with a hierarchical macro-cell based grid where small coherent blocks of cells could be transmitted across the network. Because of their acceleration structure, spatial triangle ordering method, and high communication costs, they used block sizes of 32KB. We on the other hand make use of a state-of-the-art packetized reordered BVH acceleration structure [WBS07] and a very efficient triangle ordering, which combined with lower cost communication, allows us to use smaller block sizes which results in a more efficient cache and faster transfers.

Yoon et al. [YM06] explored reordering BVHs for out-of-core collision detection and ray tracing. Their approach used a complicated cache oblivious reordering and assumed that nodes had two child pointers. Since we choose the size of the cache blocks, we do not need a cache oblivious reordering and can achieve better performance with a reordering tailored specifically for our cache block size and our nodes which only use a single child pointer (the other child is found implicitly).

DeMarle et al. [DGBP05] used a macro-cell grid approach that allowed for fine grained control of the amount of data transmitted across the network, but used a slow acceleration structure which consumes large amounts of memory, does not make use of SIMD, and does not perform as well as BVHs [WBS07], kd-trees [RSH05], and multi-level recursive grids [Ize09]. BVHs can be easily packetized and perform well with incoherent ray packets and so are a popular acceleration structure in high performance ray tracers [WMG*09]. Furthermore, for massive scenes, BVHs have the nice property of a guaranteed upper bound on storage for both the number of nodes and number of primitive references which allows us to predict how much storage will be required. kd-trees have no upper bound and generally end up using about an order of magnitude more tree nodes than a BVH and more storage for triangles which can exist in multiple nodes. For these reasons, we chose to use a BVH in our DSM ray tracer.

One common way of rendering on a distributed system is with sort-last parallel rendering where each node has a fraction of the scene data which is split up in data-space in a view-independent manner. Each node renders an entire image over the sub-scene, and then the images are depth-sorted to create the final composited image [MCEF94]. Compositing is commonly used for distributed ray casting of polygons and volumes, yet even state-of-the-art systems have trouble scaling to real-time frame rates on a cluster due either to load imbalances if the camera is focused on a small section of the scene or to the cost of each node transferring full size images. Kendall et al. have a compositing system that maximally achieves 1fps for a zoomed-in 64 million pixel image when using a cluster similar to ours [KPH*10]. This roughly translates to about 32fps at HD resolution. Only on the Jaguar supercomputer, which has a much faster interconnect, can they achieve the equivalent of around 120fps at HD resolution using 32K cores. A severe limitation, which is inherent to all compositing approaches, is that it is limited to ray casting and cannot take advantage of the more advanced shading models offered by ray tracing.

Howison et al. also use a compositing approach to render a massive 4608^3 volume using raycasting on the Jaguar supercomputer, which has 12 cores per node [HBC10]. They demonstrated that instead of running 12 MPI processes on each node, it was beneficial to use a hybrid parallelism approach whereby only one MPI process per 6-core socket is used for distributing work amongst the nodes and then each MPI process uses 6 threads to work on the shared data on the socket. This lowered the memory and communication overhead and resulted in their compositing step, which is the majority of their frame time, being twice as fast. Their maximum frame rate when running on 216,000 cores for a 21 million pixel image was 2 frames per second; assuming this scaled down to HD resolution, this would give approximately 20fps. Similar to their work, we use a hybrid approach with one MPI process per node and one thread per core.

Budge et al. [BBS*09] ray traced massive models using a hybrid CPU/GPU algorithm on a cluster where they replicated the scene on each node, and used the local hard drive to hold the excess data that did not fit either on the CPU or GPU memory. Since the nodes were severely constrained with the amount of memory they had available, and each node had 2 high-end GPUs, the addition of GPUs almost doubled the available memory, in addition to providing more compute power. They are able to use the GPUs for out-of-core rendering by focusing on non-interactive scenes with thousands of rays per pixel, which gives them a large pool of rays to reorder from so that large batches of rays are traced which use data already loaded to memory. When rendering with only a few samples per pixel, as would be expected for real-time frame rates, they experience a slowdown of a few orders of magnitude. They are able to scale to 4 nodes at 75% efficiency but claim that their technique would be unlikely to scale further.

We chose not to use a GPU in this paper for a few reasons. Firstly, we found when comparing the ray tracing performance of NVIDIA's Optix GPU ray tracer [PBD*10] when running on 2 GPUs of an NVIDIA Tesla S1070 to be roughly comparable to the performance of our 8 core node for in-core scenes, with the GPU performing worse when it did not have a large enough number of rays to trace in order to effectively hide memory latencies. Unfortunately, since each GPU only renders a small fraction of the image at a time, potentially only a few thousand rays if load balancing is used, then the performance of a GPU would be lower than that of our 8 CPU cores. Moreover, since system memory is significantly cheaper than GPU memory, most nodes will often have significantly more system memory which allows a CPU ray

tracer to ray trace much larger scenes in-core and this will clearly make the CPU ray tracer faster. Moving data into one of the GPUs also carries a significant cost of about 1ms per 6.4MB of data, making communication to the GPU almost as expensive as communication across the network. Finally, on the CPU we can communicate across the network at any point during ray tracing, while on the GPU we would have to wait for the kernel launch to terminate before being able to communicate and ask for more data. This limitation would require that we make repeated kernel launches until all the rays have finished tracing or rely on level-of-detail approximations which we are not considering for this paper. For these reasons a CPU-only approach is faster than a GPU-only approach for out-of-core rendering. A hybrid renderer could offer a performance improvement and would be interesting future work.

3. Ray tracing replicated data

We build our distributed ray tracer upon the Manta Interactive Ray Tracer, a state of the art ray tracer capable of scaling at real-time frame rates to hundreds of cores on a shared memory machine [BSP06]. If we are able to replicate data across the nodes, then we can directly use any of Manta's large selection of acceleration structures for ray tracing surface primitives or volumes using anything from ray casting to path tracing. The only modifications we need to make to Manta are assigning work (rays) to nodes, receiving pixels, and broadcasting camera and other state updates to all nodes. The challenge we face is in ensuring our distributed Manta implementation is able to scale to many nodes while ensuring real-time frame rates.

Our real-time distributed ray tracer uses a master-slave configuration where a single process, the display, receives pixel results from render processes running on the other nodes. Another process, the load balancer, handles assigning tasks to the individual render nodes.

MPI does not guarantee fairness amongst threads in a process and our MPI library currently enforces all threads to go through the same critical section, because of this we run the display and load balancer as separate processes. Since they do not communicate with each other nor share any significant state, this partitioning can be done without penalty or code complexity. Furthermore, since the load balancer has minimal communication, instead of running on a dedicated node, it can run alongside the display process without noticeably impacting overall performance.

3.1. Load balancing

Manta already uses a dynamic load balancing work queue where each thread is statically given a predetermined large tile of work to consume and then when it needs more work it progressively requests smaller tiles until there is no more work left. We extend this to have a master dynamic load balancer with a work queue comprised of large tiles which are given to each node (the first assignment is done statically and is always the same) and then each node has its own work queue where it distributes sub-tiles to each render thread. Inside each node the standard Manta load balancer is used for distributing work amongst the threads. Each thread starts with a few statically assigned ray packets to render and then takes more ray packets from the node's shared work queue until no more work is available, at which point that thread requests another tile of work from the master load balancer for the entire node to consume. This effectively gives us a two-level load balancer that ensures that work is balanced both at the node level and at the thread level. Since the top level load balancer only needs to keep the work queues of the nodes full instead of the queues for each individual thread, communication is kept low on the top level load balancer which allows us to scale to many nodes and cores.

3.2. Display process

We have one process, the display, dedicated to receiving pixels from the render nodes and placing those pixels into the final image. The display process shares a dedicated node with the load balancer process, which only uses a single thread and has infrequent communication. The display has one thread which just receives the pixels from the render nodes into a buffer. The other threads in the display then take those pixels and copy them into the relevant parts of the final image. At first we used only a single thread to do both the receiving and copying to the final image, but we found that our maximum frame rates were significantly lower than what our InfiniBand network should be capable of. Surprisingly, it turned out that merely copying data in local memory was introducing a bottleneck, and for this reason we employ several cores to do the copying.

Since InfiniBand packets are normally 2KB, any messages smaller than this will still consume 2KB of network bandwidth. We therefore need to send a full packet of pixels if we want to maximize our frame rate. If, for instance, we sent only a 7B pixel at a time, this would actually require sending an effective $2048B * 1920 * 1080 = 3.96$GB of data which would take about 2s over our high speed network, which is clearly too slow. Since our ray packets contain 64 pixels, we found that combining 13 ray packets into a single message offered the best performance since it uses close to 3 full InfiniBand packets.

4. DSM ray tracing

As mentioned in Section 2, our distributed shared memory infrastructure is similar to that of DeMarle et al. [DGBP05].

Data is spread out to nodes in an interleaved pattern and a templated direct mapped cache is used for data that does not exist locally. We access data at a block granularity of almost 8KB, with a little bit of space left for packet/MPI

overhead, which results in each block containing 254 32-byte BVH nodes or 226 36-byte triangles. Since multiple threads can share the cache, each cache line is controlled by a mutex so that multiple threads can simultaneously read from a cache line. In order to replace the data in a cache line, a thread must have exclusive access to that cache line so that when it replaces the cache element it does not modify data that is being used by other threads. Remote reads are performed using a passive MPI_Get operation which should in turn use an InfiniBand RDMA read to efficiently read the memory from the target node without any involvement of the target CPU. This allows for very fast remote reads that do not impact performance on the target node and that scale to many threads and MPI processes [JLJ*04].

DeMarle et al. assign block k to node number k mod *numNodes* so that blocks are interleaved across memory. If a node owns block k, it will then place it in location $k/numNodes$ of its resident memory array [DeM04]; we follow this convention. However, if a node does not own block k, DeMarle et al. have the node place the block in its k mod *cacheSize* cache line. We found this to be an inefficient mapping since it does not make full utilization of the cache as it does not factor in that some of that data might already reside in the node's resident memory. For instance, if we have 2 nodes and the cache size is also 2, then node 0 would never be able to make use of cache line 0. When $k \bmod 2 = 0$, then the owner of the data is $k/2 = 0$ which means that node 0 already has that data in its resident memory. Our more efficient mapping which avoids the double counting is

$$\left(k - \left\lfloor \frac{k + (numNodes - myRank)}{numNodes} \right\rfloor\right) \bmod cachSize$$

Raycasting the RM dataset described later in Section 5 with 60 nodes and our more efficient mapping gives speedups of 1.16×, 1.31×, 1.48×, 1.46×, and 1.31× over the mapping of DeMarle et al. when using the respective cache sizes of 1/32, 1/8, 1/4, 1/2, and 1/1 of total memory. Note that the 1/1 total memory should by definition be large enough to hold the entire dataset, and yet unlike with our mapping, their mapping prevents them from caching the entire dataset.

4.1. DSM BVH

Since our DSM manager groups BVH nodes into blocks, we reorder the memory locations of the BVH nodes so that the nodes in a block are spatially coherent in memory. Note that we are not reordering the actual BVH tree topology. We accomplish this for a block size of B BVH nodes by writing the nodes to memory according to a breadth first traversal of the first B nodes, thus creating a subtree that is coherent in memory. We then stop the breadth first traversal and instead recursively repeat that process for each of the B leaves of the newly formed subtree. If the subtree being created ends up with less than B leaves, we continue to the next subtree without introducing any gaps in the memory layout. It is therefore possible for a block to contain multiple subtrees. However, since the blocks are written according to a blocked depth-first traversal, the subsequent subtree will still often be spatially near the previous subtree.

In order to minimize memory usage, Manta's BVH nodes only contain a single child pointer, with the other child's memory location being adjacent to the first child. Because of this, we modify the above algorithm so that instead of recursing on each of the B leaves, we recurse on each pair of child leaves so that the two children stay adjacent in memory.

These blocks thus contain mostly complete subtrees of B nodes so that when we fetch a block we can usually expect to make $\log B$ traversals before we must fetch a new block from the DSM manager. For 254 node blocks this is about 8 traversal steps for which our DSM-BVH traversal performance should be roughly on par with the regular BVH traversal. We ensure this by keeping track of the current block we are in and not releasing that block (or re-fetching it) until we either leave the block or enter a new block. When we traverse down into a new block we must release the previously held block in order to prevent a deadlock condition where one of the following blocks requires the same cache line as the currently held block. Note that multiple threads can all safely share access to a block and that thread stalling while waiting for a block to be released will only occur if one thread needs to use the cache line for a different block than is currently being held.

Since the root of the tree will be traversed by every thread in all nodes, rather than risk this data being evicted and then having to stall while the data becomes available again, we replicate the top of the tree across all nodes. This requires only an extra 8KB of data per node and ensures that those first 8 traversal steps are always fast because they only need to access resident memory.

4.2. DSM primitives

While sharing vertices using a mesh will often halve the memory requirements, this does not adapt well to DSM since it requires fetching a block from the DSM manager to find the triangle and then once the vertex indices are known, one to three more fetches for the vertices. Thus, if a miss occurs this results in between a 2–4× slowdown for misses. Doubling the storage requirements is comparable in cost to halving the cache size, and halving the cache size often introduces less than a 2× performance penalty. There is thus no incentive to use a mesh, even though this is what DeMarle et al. do. While it might appear that we could create a sub-mesh within each block so that less memory is used and only a single block need be fetched, this will not benefit us since block sizes are fixed and we would end up with wasted empty space inside each block. Attempting to place variable numbers of triangles in each block so that empty space is reduced would

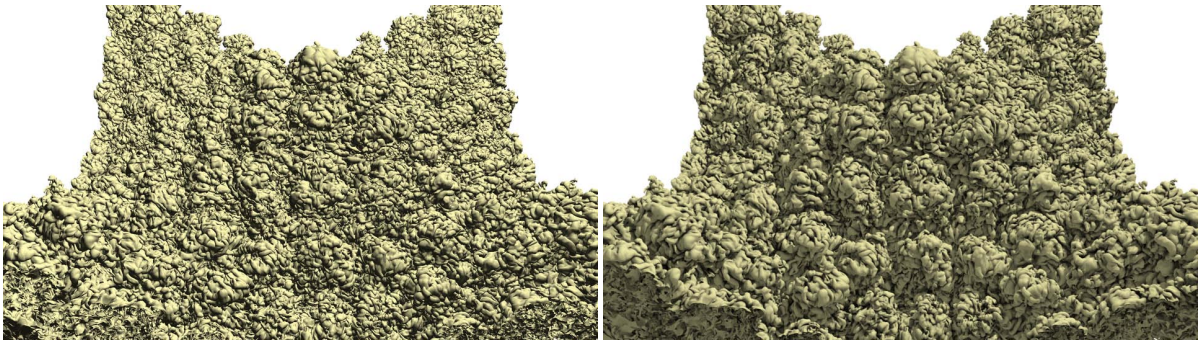

Figure 1: *On 60 nodes, we can raycast (left image) a 316M triangle isosurface of timestep 273 of the Richtmyer-Meshkov instability using HD resolution at 101fps if we can store all 21,326MB of triangle and acceleration structure data on each node and at 16fps if we use DSM to store only a total of 2,666MB per node. Using one shadow ray and 36 ambient occlusion rays per pixel (right image) we can achieve 4.76fps and 1.90fps respectively.*

also not work since then we would not be able to compute which block key corresponds to a triangle.

We reorder our triangles into blocks by performing an in-order traversal of the BVH and outputting triangles into an array as they are encountered. This results in spatially coherent blocks that also match the traversal pattern of the BVH so that if two leaf nodes share a recent ancestor they are also spatially coherent and will likely have their primitives residing in the same block.

5. Results

We used a 64 node cluster where each node contains two 4-core Xeon X5550s running at 2.67GHz with 24GB of memory and a 4× DDR InfiniBand interconnect between the nodes. We render all scenes at an HD resolution of 1920×1080 pixels. Our datasets consist of a 316M triangle isosurface of timestep 273 of the Richtmyer-Meshkov (RM) instability dataset from Larence Livermore National Laboratory (Figure 1) and the 259M triangle Boeing 777 CAD model (Figure 2). While the RM dataset could be rendered using volume raycasting, we use this polygonal representation as an example of a massive polygonal model where large parts of the model can be seen from one view. The Boeing dataset consists of an almost complete CAD model for the entire plane and unless it were made transparent, has significant occlusion so that regardless of the view, only a fraction of the scene can be viewed at any given time.

We rendered both datasets using 2–60 nodes, where one node is used for display and load balancing and the remaining ones for rendering. We used simple ray casting for both models and ambient occlusion with 36 samples per shading point for the Boeing and similar ambient occlusion but with an additional hard shadow for the RM dataset.

Figure 3 shows how we scale with increasing numbers of nodes when ray casting both scenes using replicated data across each node so that the standard BVH acceleration structure is used without any DSM overhead, and then with a cache plus resident set size $1/N$ of the total memory used by the dataset and acceleration structure. Assuming the nodes have enough memory, data replication allows us to achieve near linear scaling to real-time rates and then begins to plateau as it approaches 100fps. Since $1/1$ has a cache large enough to contain all the data, no cache misses ever occur and this indicates the overhead of our system compared to replicated data. As the cache is decreased in size, the Boeing scene shows little penalty indicating that our system is able to keep the working set fully in cache. The RM dataset on the other hand has a larger working set since more of it can be viewed and this causes cache misses to occur which noticeably affects performance for reasons which we will shortly describe.

The total amount of memory required in order to keep everything in-core is dependent on the cache size and the resident set size. On a single render node all the data must reside on it so the resident set is 21GB and the cache is 0GB. Figure 4 shows that with more render nodes the size of the resident set gets progressively smaller so that cache size quickly becomes the limiting factor as to how much data our system can handle. Since the resident set decreases as more nodes are added, cache size could similarly be increased so that the node's capabilities are used to the fullest.

5.1. Maximum frame rate

Frame rate is fundamentally limited by the cost of transferring pixels across the network to the display process. Our 4×DDR InfiniBand interconnect has a measured bandwidth of 1868MB/s when transferring multiples of 2KB of data according to the system supplied *ib_read_bw* tool. Each pixel sent across consists of a 3B RGB color and a 4B pixel location. The 4 Bytes for the location is required in order to support rendering modes where pixels in a ray packet could be randomly distributed about the image, for instance, with frameless rendering. The 4B location could be removed if

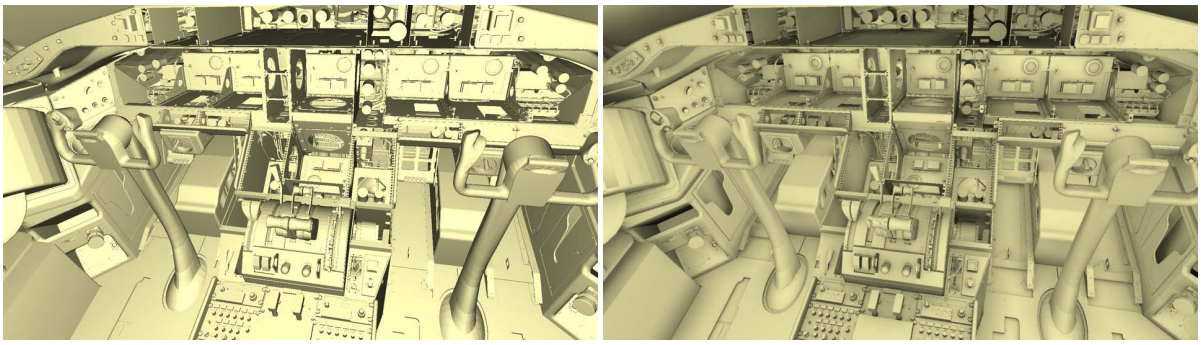

Figure 2: *On 60 nodes, we can raycast (left image) the 259M triangle Boeing using HD resolution at 96fps if we can store all 15,637MB of triangle and acceleration structure data on each node and at 77fps if we use DSM to store only 1,955MB of data and cache per node. Using 36 ambient occlusion rays per pixel (right image) we can achieve 2.33fps and 1.46fps respectively.*

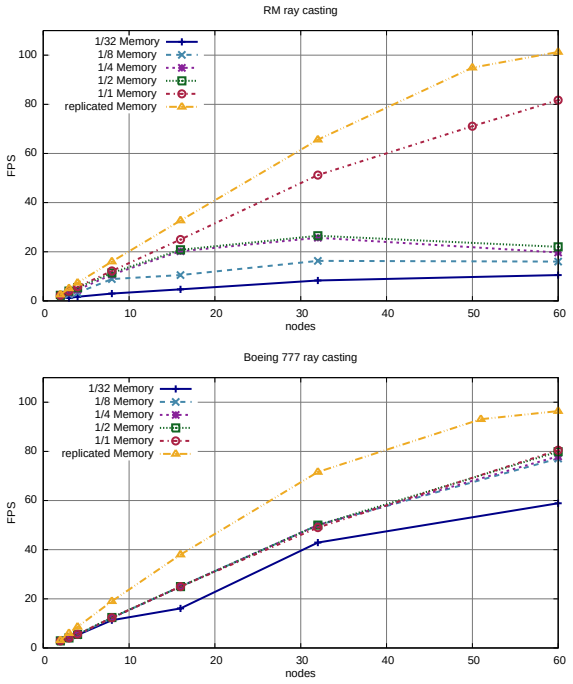

Figure 3: *Frame rate when varying cache size and number of nodes in the ray casted RM and Boeing data sets. Replicated data sees almost perfect scaling until it begins to become network bound. The Boeing with DSM scales very well, even with a small cache, because the working set is a fraction of the overall model. The RM with DSM scales well until about 20–25fps is reached.*

we knew that rays in a ray packet form a rectangular tile, in which case we would only need to store the coordinates of the rendered rectangle over the entire ray packet instead of 4B per ray. Further improvements might be obtained by compressing the pixels so that still less data need be transmitted. The time required by the display process to receive all the pixels from the other nodes is at best $\frac{1920*1080*7B}{1868MB/s} = 7.41$ms

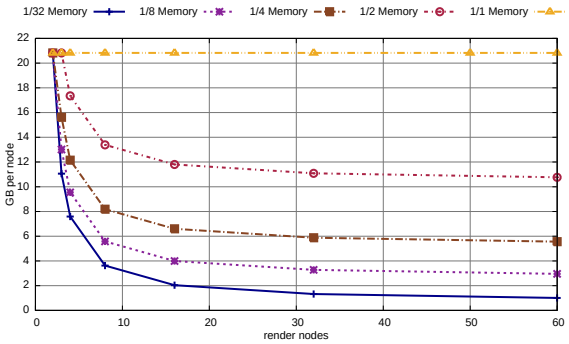

Figure 4: *Total memory used per node for the RM dataset.*

for a full HD image, which gives 135fps. Since we send $13*64 = 832$ pixels at a time, which occupies 95% of the InfiniBand packet, we would expect our implementation to achieve at best 127fps.

To see how close we can get to the maximum frame rate, we rendered an HD image of a small model with the camera pointing away from the model so that a blank screen is rendered and only a minimal amount of ray tracing performed (testing the rays against the bounding box), thus ensuring that any bottleneck in performance would not be due to ray tracing performance. To verify that the load balancer is not significantly competing for resources, we test with the load balancer on its own dedicated node and then with the load balancer sharing the node with the display process. The render nodes thus have a limited amount of work to perform and the display process quickly becomes the bottleneck. Figure 5 shows that when using one thread to receive the pixels and another to copy the pixels from the receive buffer to the image, the frame rate is capped at 55fps no matter how many render threads and nodes are used. Using two threads to copy the pixels to the image results in up to a 1.6× speedup and three copy threads improves performance by up to 2.3×, but more copy threads offer no additional benefit, showing that copying was the bottleneck until 3 copy threads were used,

after which point the bottleneck shifted to the receive thread which is not able to receive the pixels fast enough to keep the copy threads busy. When around 18 threads are used, be they on a few nodes or many nodes, we reach a frame rate of 127fps. This is exactly our expected maximum of 127fps given by the amount of time it takes to transmit all the pixel data across the InfiniBand interconnect. More render threads result in lower performance due to the MPI implementation not being able to keep up with the large volume of communication. In order to achieve these results required that we tune our MPI implementation to use more RDMA buffers and turn off shared receive queues (SRQ), otherwise, we could still achieve the same maximum frame rate of around 127fps with 17 render cores, but after that point adding more cores caused performance to more quickly drop off, with 384 render cores (48 render nodes) being 2× slower. However, this is a moot point since faster frame rates would offer no tangible benefit.

For 4M pixel images, which are currently the largest a single graphics card can render at 60Hz, our maximum frame rate would halve to about 60fps. Higher resolutions are usually achieved with a display wall consisting of a cluster of nodes driving multiple screens, so in this case the maximum frame rate would be given not by the time to transmit an entire image, but by the time it takes for a single display node to receive its share of the image. Assuming each node rendered 4M pixels, and the load balancing and rendering continue to scale, the frame rate would thus stay at 60fps regardless of the resolution of the display wall.

Since three copy threads are able to keep up with the receiving thread, and we have the load balancer process also running on the same node, this leaves us with three unused cores. If we are replicating data across the nodes then we can use these three cores for a render process. This render process would also benefit from being able to use the higher speed shared memory for its MPI communication with the display and load balancer instead of the slower InfiniBand. However, if DSM is required then we cannot run any render processes on the same node since those render processes would be competing with the display and load balancer for scarce network bandwidth and this would much more quickly saturate the network port and result in much lower maximum frame rates.

5.2. Remote read

We use MVAPICH2 as our MPI-2 implementation since it supports multi-threading, and makes use of InfiniBand RDMA read and writes [LJW*04]. Unfortunately, while this is the most advanced MPI-2 implementation we could find, as of the latest version, 1.6-rc2, it still does not support several important features. The first issue is that it uses a global mutex to guard all MPI calls, even when a finer grained synchronization could be used; this leads to excessive thread stalling. For instance, we have found replacing an MPI process with 8 threads by 8 separate single threaded MPI processes can result in a 4× speedup for a simple benchmark that repeatedly

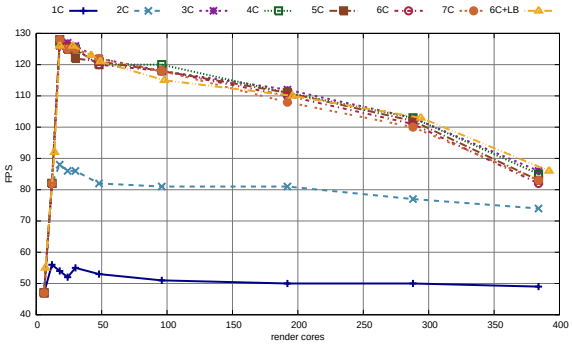

Figure 5: *Display process scaling: Frame rate using varying numbers of render cores to render a trivial scene when using 1 copy thread (1C) to 7 copy threads (7C), and when using 6 copy threads with the load balancer process on the same node (6C+LB).*

executes MPI_Get to read memory off another node. Secondly, MVAPICH2 researchers demonstrated true one-sided passive MPI_Get using InfiniBand RDMA, and showed that it allows for very fast remote reads without any processing on the remote CPU and scales very well with increasing numbers of processes and threads [JLJ*04]. Unfortunately, this has not yet been made public and so the current implementation requires that the remote process check to see if there are any pending requests for data. This check is performed by the MPI library each time an MPI call is made. Effectively, this means that if node 1 calls MPI_Get to read data off of node 2, it will only receive the data after node 2 performs an MPI call, such as an MPI_Get for data off of another arbitrary node (it need not be from node 1), or communication with the load balancer or display process. This all results in MPI_Get calls taking potentially several orders of magnitude extra time to return data, either because too many calls are being performed so that threads end up stalled at a mutex or because not enough calls are made and progress is not made by the MPI engine.

Bypassing MPI and directly using the low-level InfiniBand API would likely result in substantial performance improvements, but at significant programmer effort and code complexity. The next release of MVAPICH2, version 1.7, is expected to support true one-sided passive communication and so should solve many of these issues and should offer a significant performance improvement [Pot11]. MPI-3 is also expected to address these well-known shortcomings and offer a low-latency MPI_Get [TGR*09] that closer matches the expected performance of directly performing an RDMA read.

To get our DSM ray tracer to scale using the currently available MPI implementations, we found that tasking one render thread with periodically making a no-op MPI call (MPI_Iprob) in order to force the MPI progress engine to run helped to partially ameliorate this problem. Table 1 shows that the frame rate without these extraneous calls can be up to

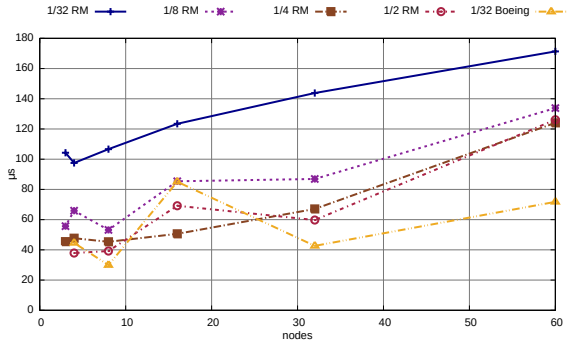

Figure 6: *Average time to perform a single MPI_Get call.*

2.4× worse and that the extra calls never significantly worsen performance.

We measured the time it takes to perform an MPI_Get to read an 8KB block of data from another node and found that occasionally we can perform the read in 12μs, which is fairly close to the measured average RDMA read latency of 9μs on our cluster. However, Table 1 shows that the average time is much larger, usually between 40-170μs with occasional MPI_Get calls taking tens of milliseconds to complete.

Figure 6 shows that increasing the number of nodes or decreasing the cache results in a longer time to complete a remote read. Were the MPI_Get implementation to be truly one-sided, then we would expect the read time to stay constant regardless of cache size or number of nodes. Doubling the number of nodes might result in twice as many MPI_Get calls, but it also doubles the number of places to read memory from, and since blocks of memory are interleaved across the nodes, this should result in each node still receiving the same number of read requests.

Since we interleave the blocks amongst the nodes' resident sets, for a large enough data set, each node has roughly an equal chance that a requested block will be in the node's resident set regardless of where in the scene the ray is located. The one significant exception is for the root block which every ray is guaranteed to fetch; although since this is just one block out of many read, this does not significantly change the probabilities. This means the ratio of block requests that go to resident memory and the ratio that check the cache can be approximated by the ratio of the resident set and cache sizes, which are respectively $1/N$ and $(N-1)/N$ for N render nodes. Experimental verification confirms this. The decrease in size of the resident set as more render nodes are used is what allows us to render massive out-of-core models; however, this also means that there is an additional cost for adding render nodes since accessing the cache, even if a hit is made, has extra cost as a critical section must be entered. Generally, the additional compute power will still make this a worthwhile trade-off, and once there are enough nodes to enable the distributed data to fully reside on the cluster, additional render nodes allow us to use larger caches, which will lower the number of cache misses and result in substantial performance improvements.

What does vary and is harder to predict are the number of cache misses that occur. For the Boeing scene, only a fraction of the triangles and BVH nodes are visited for any one view, so even a small cache can contain the entire working set. The RM dataset, on the other hand, can have many more visible triangles and so has a larger working set. Still, Table 1 shows the cache miss rate is usually less than 1%. Using ambient occlusion, which we expected to stress our DSM implementation, actually resulted in even better cache usage.

6. Future work

Clearly we would like to run our system with the next release of MVAPICH2 to see how much of a benefit we can get from an optimized MPI_Get implementation. We expect that since remote reads are currently an order of magnitude slower than they should be, that we could see a doubling or tripling in performance over what we presented when significant cache misses occur.

Adapting our system for use with a display wall in order to achieve real-time frame rates at massive pixel resolutions would be very interesting and useful for visualization.

Currently we build our acceleration structure offline using a serial algorithm which can take a few hours for a massive scene. Parallelizing this over all the cores in the cluster could bring this down to a few minutes.

Since Manta supports volume ray casting, we can already handle volumes using data replication; however, it would be nice to extend our system to also use DSM for massive volumetric datasets.

7. Conclusion

With modern hardware and software, we can ray trace massive models at real-time frame rates on a cluster and even show interactive to real-time rates when rendering out-of-core using a small cache. We are often one to two orders of magnitude faster than previous cluster ray tracing papers which used both slower hardware and algorithms [DGBP05, WSB01], or had equivalent hardware but could not scale to as many nodes or to high frame rates [BBS*09]. Compared to compositing approaches, we can achieve about a 4× improvement in the maximum frame rate for same size non-empty images compared to the state-of-the-art [KPH*10] and can handle advanced shading effects for improved visualization.

8. Acknowledgements

This publication is based on work supported by Award No. KUS-C1-016-04, made by King Abdullah University of Science and Technology (KAUST), DOE VACET, NSF OCI-0906379, NSF CNS-0615194.

cache size	%cache miss	blocks read	Without no-op MPI calls FPS	avg μs	min μs	max μs	With no-op MPI calls FPS	avg μs	min μs	max μs
Richtmyer–Meshkov, 60 nodes										
1/32	1.88	6158K	10.7	186	12.1	23690	10.5	171	12.9	19807
1/8	0.52	6158K	14.3	198	12.9	26350	16.0	133	12.9	24657
1/4	0.19	6158K	15.5	220	12.9	21661	19.7	124	12.9	19693
1/2	0.07	6158K	14.7	249	12.9	21679	22.0	126	12.9	15166
Boeing 777, raycast, 60 nodes										
1/32	0.08	4524K	51.4	200	11.9	12423	58.9	234	14.8	1408
1/8	4E-4	4524K	78.5	144	15.0	1408	77.1	70.2	15.0	2170
Richtmyer–Meshkov, ambient occlusion, 60 nodes										
1/32	0.48	219M	1.06	175	11.9	30517	1.25	110	11.9	42119
1/8	0.21	219M	1.18	355	11.9	69432	1.90	100	11.9	50262
1/4	0.11	219M	1.10	692	11.9	107827	2.05	120	12.9	49744
1/2	0.02	219M	1.02	1421	11.9	153737	2.41	136	12.9	52346
Boeing 777, ambient occlusion, 60 nodes										
1/32	0.06	351M	0.62	626.7	11.9	309025	1.02	66.0	11.9	111139
1/8	7E-6	351M	1.47	14483	19.1	88821	1.46	35.9	15.9	87.9

Table 1: *As the cache grows, fewer cache misses occur and so the MPI no-op calls become more important. There is substantial variance in the time for data to be fetched due to the suboptimal MPI_Get implementation. Since the working set for the Boeing scene fits in 1/8 and larger size caches, only a few misses occur, so we do not show results for those caches sizes.*

References

[BBS*09] BUDGE B., BERNARDIN T., STUART J. A., SENGUPTA S., JOY K. I., OWENS J. D.: Out-of-core data management for path tracing on hybrid resources. *Computer Graphics Forum 28*, 2 (2009), 385–396. 2, 8

[BSP06] BIGLER J., STEPHENS A., PARKER S. G.: Design for Parallel Interactive Ray Tracing Systems. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006). 3

[CM93] CORRIE B., MACKERRAS P.: Parallel volume rendering and data coherence. In *Proceedings of the 1993 symposium on Parallel rendering* (1993), PRS '93, pp. 23–26. 1

[DeM04] DEMARLE D. E.: Ice network library. http://www.cs.utah.edu/~demarle/software/, 2004. 4

[DGBP05] DEMARLE D. E., GRIBBLE C., BOULOS S., PARKER S.: Memory sharing for interactive ray tracing on clusters. *Parallel Computing 31* (2005), 221–242. 1, 2, 3, 8

[GP06] GRIBBLE C. P., PARKER S. G.: Enhancing interactive particle visualization with advanced shading models. In *Proceedings of the 3rd symposium on Applied perception in graphics and visualization* (2006), pp. 111–118. 1

[HBC10] HOWISON M., BETHEL E. W., CHILDS H.: MPI-hybrid parallelism for volume rendering on large, multi-core systems. In *Eurographics Symposium on Parallel Graphics and Visualization* (2010). 2

[Ize09] IZE T.: *Efficient Acceleration Structures for Ray Tracing Static And Dynamic Scenes*. PhD thesis, University of Utah, 2009. 2

[JLJ*04] JIANG W., LIU J., JIN H., PANDA D., BUNTINAS D., THAKUR R., GROPP W.: Efficient implementation of MPI-2 passive one-sided communication on InfiniBand clusters. *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (2004), 450–457. 4, 7

[KPH*10] KENDALL W., PETERKA T., HUANG J., SHEN H., ROSS R.: Accelerating and benchmarking radix-k image compositing at large scale. In *Proceedings Eurographics Symposium on Parallel Graphics and Visualization* (2010), pp. 101–110. 2, 8

[LJW*04] LIU J., JIANG W., WYCKOFF P., PANDA D. K., ASHTON D., BUNTINAS D., GROPP W., TOONEN B.: Design and implementation of MPICH2 over InfiniBand with RDMA support. *Parallel and Distributed Processing Symposium, International 1* (2004), 16b. 7

[MCEF94] MOLNAR S., COX M., ELLSWORTH D., FUCHS H.: A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications 14* (July 1994), 23–32. 2

[PBD*10] PARKER S., BIGLER J., DIETRICH A., FRIEDRICH H., HOBEROCK J., LUEBKE D., MCALLISTER D., MCGUIRE M., MORLEY K., ROBISON A., STICH M.: OptiX: a general purpose ray tracing engine. In *ACM SIGGRAPH* (2010). 2

[Pot11] POTLURI S.:. Personal communication, 2011. MVAPICH2 developer. 7

[RSH05] RESHETOV A., SOUPIKOV A., HURLEY J.: Multi-level ray tracing algorithm. *ACM Transaction on Graphics 24*, 3 (2005), 1176–1185. (Proceedings of ACM SIGGRAPH 2005). 2

[TGR*09] TIPPARAJU V., GROPP W., RITZDORF H., THAKUR R., TRÄFF J.: Investigating high performance RMA interfaces for the MPI-3 standard. In *2009 International Conference on Parallel Processing* (2009), IEEE, pp. 293–300. 7

[WBS07] WALD I., BOULOS S., SHIRLEY P.: Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics 26*, 1 (2007), 1–18. 2

[WMG*09] WALD I., MARK W. R., GÜNTHER J., BOULOS S., IZE T., HUNT W., PARKER S. G., SHIRLEY P.: State of the art in ray tracing animated scenes. *Computer Graphics Forum 28*, 6 (2009). 2

[WSB01] WALD I., SLUSALLEK P., BENTHIN C.: Interactive distributed ray tracing of highly complex models. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques* (2001), pp. 274–285. 1, 8

[YM06] YOON S., MANOCHA D.: Cache-efficient layouts of bounding volume hierarchies. *Computer Graphics Forum (Eurographics)* (2006). 2