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

Revisiting Parallel Rendering for Shared Memory Machines

B. Nouanesengsy[1,2], J. Ahrens[2], J. Woodring[2], and H. Shen[1]

[1]The Ohio State University
[2]Los Alamos National Laboratory

Abstract
Increasing the core count of CPUs to increase computational performance has been a significant trend for the better part of a decade. This has led to an unprecedented availability of large shared memory machines. Programming paradigms and systems are shifting to take advantage of this architectural change, so that intra-node parallelism can be fully utilized. Algorithms designed for parallel execution on distributed systems will also need to be modified to scale in these new shared and hybrid memory systems. In this paper, we reinvestigate parallel rendering algorithms with the goal of finding one that achieves favorable performance in this new environment. We test and analyze various methods, including sort-first, sort-last, and a hybrid scheme, to find an optimal parallel algorithm that maximizes shared memory performance.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture—Parallel processing

1. Introduction

In supercomputer and cluster environments, there has traditionally been a lack of available hardware acceleration. While graphics hardware can be found in specific supercomputers and visualization clusters, a large percentage of installations do not have them. In these cases, image generation must be handled by software rendering. Mesa is the de facto standard for software implementation of the OpenGL API. Recently, there has been some research into exploring the feasibility of interactive software rendering on a supercomputer [ALN*08]. Rendering on the supercomputing platform acts as a stand-in for graphics hardware or a dedicated graphics cluster. As data sizes grow and data movement bandwidth lags, the need to bypass transferring data to a graphics cluster, and instead perform the rendering in-place on the supercomputer, will grow. The architectural trend of increasing performance by placing multiple cores on one chip has been continuing for many years. Having chips containing tens (and eventually hundreds) of cores coupled with the fact that one workstation can contain several such chips results in very large shared memory machines. As this current technology trend continues, the nodes of supercomputers will continue to increase their core count. Eventually, a distributed supercomputer's nodes will be composed of or resemble large shared memory machines. Therefore, finding an optimized parallel software rendering method for shared memory machines will further aid the feasibility of performing visualization on the supercomputer.

In this paper, we explore the approach of parallel rendering in a shared memory environment. Various parallel rendering methods originally designed for distributed memory, including sort-first, sort-last, and a hybrid sort-first and sort-last algorithm, are investigated. Different compositing techniques, including binary swap and a shared memory direct-send, are also investigated. These algorithms were implemented for a shared memory architecture, and then tested and analyzed for their performance.

There has been considerable past effort in developing parallel rendering algorithms. Most notable of these include sort-first and sort-last rendering. These algorithms were designed with distributed memory architectures in mind, in which processors need to communicate explicitly with each other. Much of the research pertaining to sort-last rendering involves trying to improve compositing time, usually by reducing the number of pixels sent [MWP01] or compressing pixel data [AP98]. For sort-first rendering, much of the concern is with load balancing [Mue95] and handling the problem of geometry being transferred over the network [Mue97]. In this paper, we implemented these algorithms on a shared memory machine, in which case many of these communication concerns become much less relevant.

DOI: 10.2312/EGPGV/EGPGV11/031-040

In order to further improve performance, we also include other optimizations to enhance cache performance and to manage non-uniform memory access (NUMA) architectures. A NUMA machine is a type of shared memory computer in which separate memory is provided for each processor, called *local memory*. Processors can still access any part of main memory, but accesses to local memory are much faster versus reading from other parts of memory. Because of these differences in access times, extra care must be taken to ensure consistent, reliable performance on a NUMA machine.

As a result of our testing, we found that sort-last rendering was not scalable because rendering became a bottleneck for a high number of threads. We also discovered that sort-first rendering did not have good overall performance due to the algorithm's tendency to require some geometry to be rendered multiple times. From our benchmarking tests, we have found that a hybrid sort-first and sort-last approach, first introduced by Samanta et al. [SFLS00], saw the best overall performance on a shared memory machine (see Figure 9). Rendering times alone were close to linear speedup (see Figure 7). We combined this rendering phase with a shared memory compositing scheme first espoused by Reinhard and Hansen [RH00], which got upwards of up to seven times speedup over binary swap (see Figure 3).

Our contributions in this paper include:

- Using software rendering on a shared memory machine to emulate a many-core supercomputer node
- An evaluation of how current parallel rendering algorithms perform on a shared memory machine
- A description of a scalable parallel rendering algorithm, which is a combination of hybrid sort-first and sort-last rendering, coupled with a shared memory specific compositing scheme
- A compositing method that ensures load-balanced compositing work over all threads
- NUMA-aware modifications to ensure optimal performance on NUMA architecture

2. Previous Work

Molnar [MCEF94] introduced the concept of classifying parallel methods based on where in the rendering pipeline the sorting occurred. His classification includes sort-first, sort-middle, and sort-last. Sort-first involves splitting the screen space into several non-overlapping partitions, then sorting the geometry by which partition the geometry projects to. Sort-middle involves splitting the screen in the same manner, then randomly dispersing geometry evenly to all nodes. The geometry's vertices are transformed, then sent to the node that has the partition which the vertices are projected on, for scan conversion. Sort-last rendering involves sending equal amounts of geometry to each node. Then the nodes will render their geometry, each producing a full size partial image. The final image is created by compositing all partial images through depth test comparisons. In this paper, we implement and test the performance of sort-first and sort-last on a shared memory machine.

There have been some previous efforts of developing parallel software rendering systems on massive parallel architectures. These architectures include the CM-5 [OHA93], the BBN Butterfly TC2000 [Whi94], and the SGI Origin 2000 [ISH98] [PSGL94]. Tightly coupled parallel system became unpopular in favor of multi-node supercomputers and commodity clusters. We revisit parallel software rendering on large shared memory computers in this paper.

An earlier effort to parallelize the Mesa library was done by Mitra and Chiueh [McC98]. They focused on adding functions to the Mesa API in order to facilitate a serial program being converted to a parallel program with few modifications. Their parallel version of Mesa performed sort-last rendering, and was primarily deployed in tandem with a simulation program on a distributed memory environment. In our work, we compare various parallel algorithms on a different hardware platform.

A hybrid sort-first and sort-last parallel rendering method was introduced by Samanta [SFLS00]. It involves assigning geometry evenly over all nodes while simultaneously trying to keep all geometry assigned to a node close together in screen space. Thus each node produces an image tile. In the best case, image tiles will only overlap slightly in screen space, reducing compositing work. This method was tested on a cluster of PCs, and was able to get good performance on 64 nodes. We implement the hybrid method and find that it was the best performing of all the algorithms tested.

Reinhard and Hansen [RH00] compared different compositing techniques and how they performed on a shared memory machine. They tested three different methods: parallel pipeline, binary swap, and a shared memory analogue to direct-send. Timings tests were performed on the three different algorithms, which showed that each method had the same general performance. Our results using different compositing methods yielded different times, though. We base our final compositing algorithm on their direct-send scheme.

Recently, there has been some successful work on increasing performance by modifying existing algorithms to take specific advantage of shared memory, multi-core architectures. In 2009, Nonaka and Ono [NO09] increased compositing performance on a cluster with multi-core nodes by using shared memory compositing among intra-node images, which reduced the number of images to one per node. Then binary swap was performed between the nodes to obtain the final image. They found this two-phase method faster than simply doing binary swap among all cores. Similarly, Howison et al. [HBC10] compared parallel volume ray casting using an MPI-only implementation and an MPI-hybrid method that used shared memory techniques, and got faster frame rates with the MPI-hybrid method. The increased performance was mainly the result of a decrease in compositing time. A hybrid shared memory, multi-core scheme for parallel rendering on a cluster composed of multi-core nodes is a possible future direction. For this paper, we are focused on getting the best rendering performance on one node composed of a large shared memory machine.

3. Distributed Memory vs Shared Memory Architectures

In computer hardware, distributed memory and shared memory are the two most prevalent multiple-processor computer architectures employed today. In distributed memory, each processor has its own private memory. Explicit communication must occur in order for processors to exchange and share data. Since communication is a possible bottleneck, many parallel algorithms for distributed memory try to minimize the amount of communication needed. A shared memory architecture can be characterized by the property that all processors have access to main memory. No direct communication between processors is needed, but synchronization may be required when multiple processors operate on the same data in order to avoid race conditions. Shared memory algorithms must consider cache performance, bus contention, and memory management in order to fully optimize performance.

Shared memory architectures can be further categorized into two groups: non-uniform memory access (NUMA) and uniform memory access (UMA) architectures. In a NUMA architecture, the system is separated into multiple nodes, or *numa nodes*. Each numa node contains one or more processors and a subset of main memory. Latency times for processors accessing memory that is located on the same numa node, i.e. *local memory*, is much faster than requesting memory that is on another numa node, i.e. *remote memory*, in which case memory needs to be sent over a transport (such as AMD's HyperTransport), increasing latency. Thus in a NUMA machine, retrieving different parts of memory will take different amounts of time. In a UMA architecture, all processors have equal access to the memory, and a read from any part of the memory will have equal latency.

4. Timing Tests Setup

For testing, we used two machines. One is called Oceans12, a 24-core machine that consists of 4 CPUs, each of which are 6-core Intel Xeons, and is a UMA machine. The other machine, called Kratos, is a 32-core machine consisting of 8 CPUs which are each 4-core Opterons. Kratos is a NUMA machine.

For each of our shared memory implementations, we used Mesa for the software rendering. All of our implementations were written in C++ and used the POSIX pthreads library to enable parallel computation. Let N be the number of threads used. For sort-first and hybrid sort-first and sort-last, we divided the geometry into a kd-tree. In the sort-first case, this was used for view frustum culling. In the hybrid method, the kd-tree was used to quickly obtain screen partitions during each frame. The kd-tree used was optimized such that geometry was packed contiguously in memory.

We consider the construction time of the kd-tree as a preprocessing step. Since we are mainly concerned about the rendering speed and resulting level of interactivity, we ignore any preprocessing times in our timing results.

We performed timing tests for each of our algorithm implementations. Unless otherwise stated, the dataset used is an isosurface that has roughly one million triangles, with an image resolution of 1024^2. We rendered 60 frames while rotating the camera, and then took the average and maximum time for frame time, rendering time, and compositing time. More specifically, rendering time is the time spent in Mesa. The frame time is defined as the time required from the moment one frame is done to the time the next frame is done. In other words, frame time is the sum of rendering time, compositing time, synchronization time, and other overhead.

5. Sort-Last Rendering on Shared Memory

Sort-last rendering is one of the most popular parallel rendering algorithms in use today because of its robustness and scalability [CMF05] [CM06]. In sort-last, load balancing the rendering phase is trivial, thus the rendering times easily scale to the number of processes used. The main bottleneck of sort-last is the composite step. When compositing full image buffers, the number of pixels that need to be composited per processor approaches a constant number, or more specifically twice the number of pixels in one full image [YWM08]. Several methods have been devised to try to improve compositing times, including binary swap [MPH94], direct-send [EP08], and more recently radix-k [PGR*09].

When migrating sort-last to a shared memory environment, the main advantage gained is that network transfers are no longer required, since all threads share memory. For the rendering step, this means little other than avoiding the initial send of geometry to each thread. During the rendering phase, we randomly gave an equal number of triangles to each thread. Each thread created its own Mesa OpenGL context, and then rendered a full sized partial image to an offscreen buffer. The partial images must be full sized because of the random assignment of geometry, which mean triangles could be anywhere in screen space.

The elimination of any need for network transfers is a significant change for the compositing step, though. Because a thread can read any image buffer easily, we expect compositing to be very fast in general. For our benchmarks, we chose to compare two different compositing strategies, binary swap and Reinhard and Hansen's shared memory

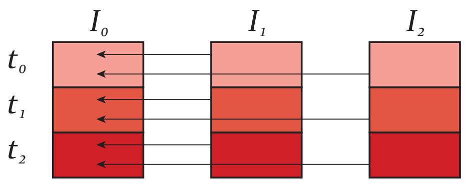


Figure 1: *Shared memory direct-send compositing. Three images, I_0, I_1, I_2, are split into three subregions. Threads t_0, t_1, t_2 are responsible for compositing one subregion. Here, pixels in I_1 are composited with pixels in I_0, and pixels in I_2 are also composited with pixels in I_0.*

direct-send algorithm [RH00], which we refer to as simply direct-send. Binary swap is chosen because it is one of the most popular compositing techniques, and is optimal in terms of parallelism utilization. Direct-send is selected because it is explicitly designed for a shared memory architecture. Figure 1 illustrates how direct-send operates. In direct-send, each N partial image is split into N equal sized subregions. Each thread is then assigned a subregion. Every thread is responsible for compositing together corresponding subregions over every partial image to get the final pixel values of that subregion. Composited pixels can be stored in one of the image buffers, or a separate final image buffer.

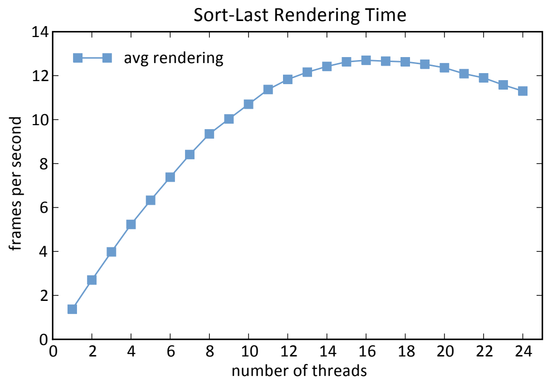


Figure 2: *Sort-last rendering times on Oceans12. Performance peaks at 16 threads and then begins to dip.*

Figure 2 shows the rendering times obtained from sort-last rendering on a shared memory machine. Rendering times improve until it reaches a certain point, in this case 16 threads, at which time it levels off and then begins to decrease. We believe this hump occurs because of cache performance and memory bus contention. One factor for poor scaling is that each thread will each generate a full sized partial image. This means that at a certain point, cache performance will suffer because the cache will not be sufficiently large enough to hold all partial images in the cache. For example, rendering an image of resolution 1024^2 requires 4MB of memory. Oceans12 has an L3 cache of size 12MB, shared over six cores. Theoretically, a maximum of three full size images can fit in the cache, ignoring memory requirements for geometry and the program itself. Once more than three cores are being used in a CPU, parts of the partial images will begin to be constantly moved to and from cache when cache misses occur. As the number of threads continue to increase, the amount of stress on the memory bus from these cache operations will rise and lead to contention, in which case performance suffers. This is exacerbated by the fact that in sort-last, a thread's geometry can lie anywhere on the screen, so having cache hits when writing to the framebuffer is even more unlikely.

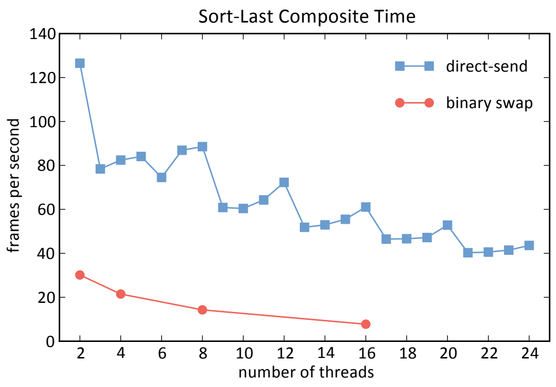


Figure 3: *Sort-last compositing times on Oceans12. Two different compositing methods are tested. Direct-send outperforms binary swap over all number of threads.*

From their testing, Reinhard and Hansen came to the conclusion that the choice of compositing algorithm did not matter when running on a shared memory machine as long as the algorithm did not read any pixels more than once. Our results differed from that conclusion, though. In Figure 3, we benchmarked and compared binary swap and direct-send. As can be seen, direct-send significantly outperforms binary swap over all number of threads. At 16 threads, direct-send exhibits a speedup factor of about seven over binary swap.

The main reason for this discrepancy is that binary swap requires frequent synchronizations. A global barrier is needed after rendering is complete, and additionally a thread is required to synchronize with its partner during each step of binary swap. This synchronization greatly hampers performance. On the other hand, direct-send requires very little synchronization. Once a thread has finished rendering, it can immediately begin to composite. There will be wait times, though, as a compositing thread may have to wait until a particular image is done rendering until it can start compositing that image.

Overall, we found that sort-last does not scale well on a shared memory machine. Even though compositing is generally fast, the bottleneck becomes the rendering phase, which eventually plateaus and then declines as the number of threads increase.

6. Sort-First Rendering on Shared Memory

At first glance, sort-first rendering would seem to be a more scalable option than sort-last. The main reason that sort-last does not scale well, as mentioned in Section 5, is because each thread will produce a full sized image. Dealing with N full sized image buffers hurts cache performance and creates memory bus contention. In sort-first, the screen is partitioned into N non-overlapping sections, so each thread will on average only have an image buffer size of $W \cdot H/N$, where W and H are the width and the height of the image, respectively. The sum of all buffers will never exceed the original image size, so there should be little trouble keeping all the partial image buffers in the cache.

Another advantage that sort-first has in a shared memory environment is that there is no longer any need to send geometry over the network. This has been a traditional disadvantage of sort-first. Normally, using sort-first requires sending geometry to the node that is going to render it, if the node does not already have the geometry in memory. In the case of geometry spanning multiple screen partitions, the geometry must be transmitted to every node that needs to render it. Temporal coherence between frames can help mitigate the amount of data transferred between frames, but some geometry will still need to be sent over the network. On a shared memory machine, this constraint is eliminated. Once the data is loaded into memory, any thread can access any portion of the geometry without having to worry about network delays.

For our sort-first implementation, we used the MAHD algorithm [Mue95] to dynamically repartition the image every frame. In MAHD, the screen is first divided into a 2D mesh. Then geometry (or in our case, kd-tree node bounding boxes) are projected into screen space. Each mesh cell is associated with a cost. Every time a triangle is found to lie within a mesh cell, the cost for that cell is increased. A summed area table is then used to quickly determine the cut that will generate the most balanced partitions. The MAHD algorithm is generally very fast, in our experience about 5ms depending on the number of partitions made. The data is loaded into a kd-tree structure as described in Section 4. Then each thread, knowing their partition, generates a Mesa OpenGL context, and renders the geometry for its partition. View frustum culling is used to quickly discard triangles outside the partition and speed up rendering. When each thread is done rendering, the partition is copied to a final image buffer.

Figure 4 shows the results of our sort-first timings. The average frame time as well as the maximum time any thread took is shown. Overall, the performance continues to rise as the number of threads increase, unlike sort-last which leveled off and then declined. Despite that, the absolute rendering times are slower than sort-last, and the speedup is far short of an optimum linear speedup. The reason we believe sort-first does not perform well is because of the number of triangles being rendered multiple times. A triangle may be rendered multiple times for two reasons. First, triangles that span over more than one kd-tree node may be rendered multiple times, since it is unknown whether those triangles have already been rendered or not. Second, triangles that span multiple image partitions must be rendered for every image partition it is in. Redundant triangles become more of a problem as the number of partitions increase, since image partitions become smaller and the number of triangles spanning more than one partition will increase. Our tests indicate that for 24 threads, rendering redundant geometry accounts for about 40% of the rendering time. This unnecessary work greatly impairs the performance of sort-first. The speedup factor will also suffer from image partitioning and view frustum culling, whose costs remain constant throughout all number of threads.

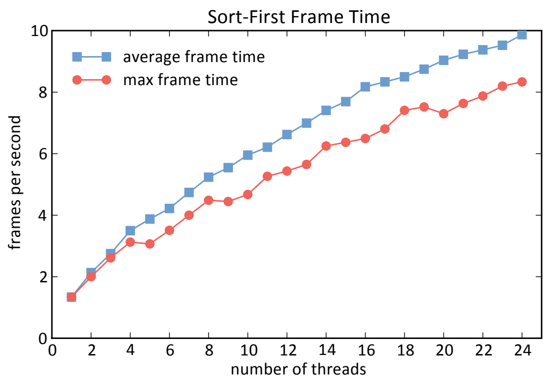


Figure 4: *Sort-first frame time on Oceans12.*

Although some of the drawbacks of sort-first become mitigated when moving to a shared memory environment, from our tests we conclude that sort-first is not a viable option. It simply does not scale well. At even higher number of threads, we expect sort-first performance to level off as the amount of time spent on redundant triangles begin to dominate the rendering time.

7. Hybrid Sort-First and Sort-Last Rendering

First proposed by Samanta et al. [SFLS00], hybrid sort-first and sort-last rendering is an approach that combines aspects from both algorithms. Essentially, an attempt is made to split the image in screen space into several non-overlapping partitions that require roughly the same amount of rendering, similar to sort-first. What makes this algorithm different is that the non-overlapping restriction is relaxed, such that partitions can overlap as needed.

The first step in our implementation is to place all geometry in a kd-tree. Then all the nodes of the kd-tree at a certain depth are chosen, and their bounding boxes are projected to screen space. When choosing which kd-tree depth to use, trade offs must be weighted between partition creation time and partition granularity. If the depth is too small and the kd-tree nodes are large, then the resulting partitions overlap more than necessary in screen space. On the other hand, choosing a depth that is too large will result in an excessive number of nodes and lengthen the partition calculation time. In our case, we empirically found that a depth of 9 was optimal. The partitions are obtained in a recursive fashion, with the current partition split either vertically or horizontally at each step. When creating a vertical split, two lists of kd-tree nodes are maintained, one for the left partition and one for the right partition. The leftmost available kd-tree node, in terms of position in screen space, is marked for the left partition, then the rightmost available kd-tree node, based on position in screen space, is marked for the right partition. This continues until all kd-tree nodes have been selected. Horizontal splits are created in an analogous way. The resulting partitions will be split recursively, with the direction of the split alternating between vertical and horizontal, until the desired number of partitions is obtained.

Figure 5 illustrates two partitions. Notice that both partitions only overlap slightly in the middle. It can also be seen that each piece of geometry is assigned to only one partition, thus there is no wasted effort to render redundant triangles. The isosurface that was used in our timing tests, split into four partitions, is shown in Figure 6.

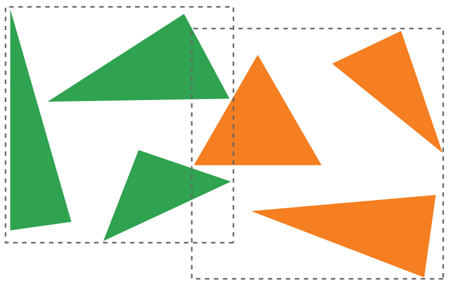


Figure 5: *An example of two partitions in hybrid rendering. The partitions only overlap slightly in the middle.*

Once partitions are calculated and geometry is assigned, each thread renders all geometry allocated to it, and creates an image tile. Then each image tile will be composited onto a common image buffer to create the final image.

For the compositing step, we chose to use direct-send, since we found it the best performing compositing algorithm (see Figure 3). To adapt it to the hybrid rendering framework, we first split the full sized final image into equal-sized horizontal strips, and assigned each thread a strip. Each thread is responsible for compositing all image tiles that overlap

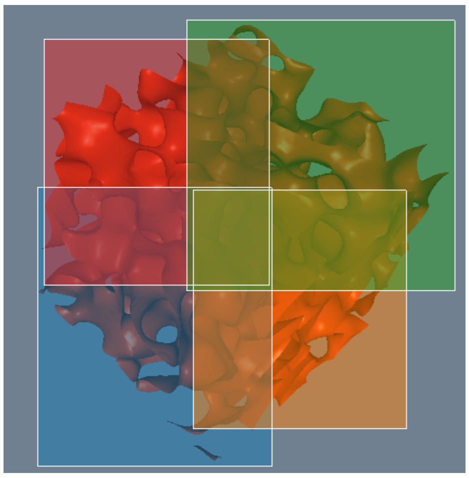


Figure 6: *An isosurface divided into four rendering partitions in the hybrid method.*

its composite region. Note that horizontal strips are used because they are contiguous in memory, which increases cache performance. Once a thread finishes rendering, it does not need to wait and can immediately begin compositing. To composite, first a thread checks the screen coordinates of every image tile to see if it overlaps with the thread's assigned composite region. If there is an overlap and that tile is done rendering, then the overlapping portion is composited. If that image tile is not done rendering, then it is skipped and another image tile is tested. This continues until all necessary images tiles have been composited.

Figure 7 details rendering time using the hybrid method. Rendering performs very well with the hybrid method. The speedup remains close to linear even up to the full number of cores available.

Rendering using the hybrid method works well on a shared memory environment for several reasons. First, the image tiles generated by each thread are smaller than the full image. This helps keep each image tile framebuffer in cache, thus making it more likely that each write to the image tile framebuffer is a cache hit. This is especially important when multiple cores of the same CPU are each rendering an image tile. If these cores are sharing one or more caches, then they are all competing for shared cache space. Having smaller partitions makes it more likely that all or most of the image tile buffers can fit together in cache.

Another reason that rendering scales well with this method is because, on average, there will be less writes to the framebuffer. This is because more pixels will fail the depth test when rendering. To understand why this is so, think of the worst case sort-last rendering scenario, where each thread is assigned geometry that is spread throughout the entire image. Thus for many triangles that will not be visible in the final image, they will be rasterized and written to the framebuffer. This will result in numerous memory read and write operations. Because triangles can be anywhere on the screen and processed in any order, these writes are spread out through the entire image, and have little spatial locality, possibly increasing cache misses. In the hybrid method, all geometry assigned to a partition is in a localized area of screen space, thus the increased likelihood that a triangle is partially or fully occluded. This results in more pixels failing the depth test and less writes to image tile framebuffers.

As Figure 8 shows, compositing time for the hybrid scheme performs very well, with the maximum composite times ranging from about 300 to 700 frames per second. This is due to the fact that partitions are generally much smaller than the full image size and they only overlap each other in certain regions, thus the number of pixels to composite is much less than in sort-last rendering.

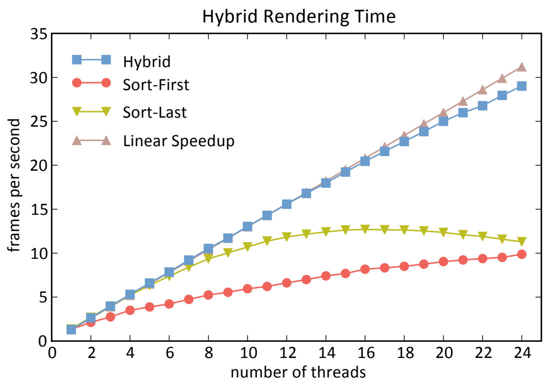


Figure 7: *Average rendering time using the hybrid method on Oceans12. Sort-first and sort-last are included for comparison. Hybrid rendering times scale almost linearly.*

7.1. Performance Comparison

Figure 9 graphs the frame time of all methods we implemented and tested. Note that the hybrid timings used further optimizations that will be discussed in Sections 7.2 and 7.3. As an additional test, parallel rendering using Paraview is also included. Paraview uses sort-last rendering, but only through MPI processes. Software rendering is still used, and compositing employs the IceT library [MWP01], which is optimized for sort-last compositing. The Paraview times are included as an example of how an algorithm designed for distributed memory will perform when run unmodified on a shared memory machine.

Even though modern MPI distributions do detect and attempt to optimize for a shared memory environment, all algorithms specifically designed for shared memory ended up performing better than Paraview. When the number of threads are low, sort-last outperforms sort-first, but as the

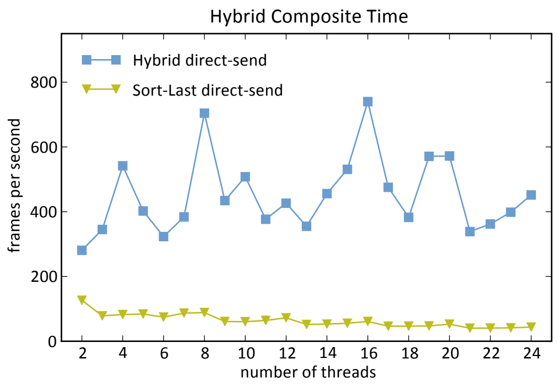


Figure 8: *Average composite times using the hybrid algorithm and direct-send on Oceans12. Compositing times from sort-last rendering using direct send is also shown for comparison.*

number of threads increase, sort-last performance declines and sort-first becomes faster. The slightly erratic timings of sort-last on Kratos is probably due to NUMA effects. Overall, the hybrid sort-first and sort-last method showed the best scaling of all methods we tested. As the number of threads increase, the hybrid method clearly outperforms all other schemes on both of our test machines, and is the only algorithm that shows good scalability.

Other datasets of varying sizes were tested using the hybrid method, the results of which are graphed in Figure 10. The datasets *cosmo1* and *cosmo2* are isosurfaces derived from a cosmology simulation, and contain 4.5 million and 8 million triangles, respectively. The *salt* dataset is an isosurface obtained from salinity data that contains 15 million triangles. Figure 10 graphs parallel efficiency, which is computed using

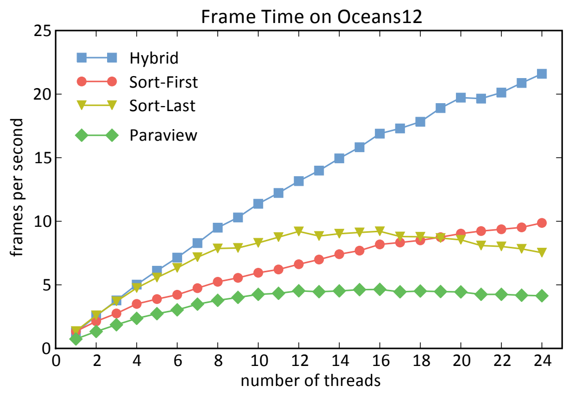
$$\varepsilon_n = \frac{t_{serial}}{n \cdot \bar{t}_n}, \tag{1}$$

where n is the number of threads, t_{serial} is the frame time achieved with one thread, and $\bar{t}_n$ is the average frame time using n threads. For all datasets, the hybrid method scales well, with the parallel efficiency never dipping below 0.85.

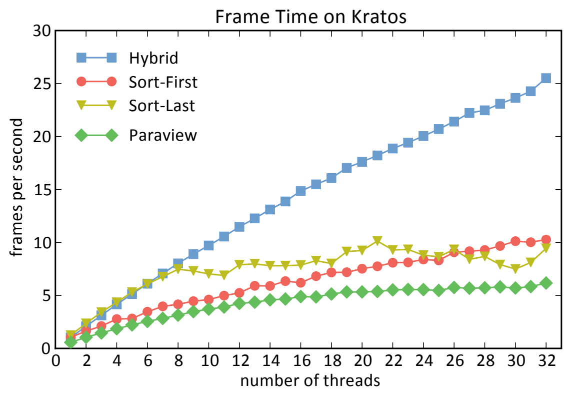
7.2. Load-balanced compositing

One problem with using direct-send as it was originally designed is that it is not necessarily load-balanced when used in conjunction with hybrid rendering. In certain cases some threads composited very few pixels, while other threads composited the majority of pixels. In order to achieve a more load-balanced compositing, we modified direct-send by partitioning the image into regions with equal amounts of work.

To do this, we first find the amount of compositing work that is needed for each row of the final image. Since we know

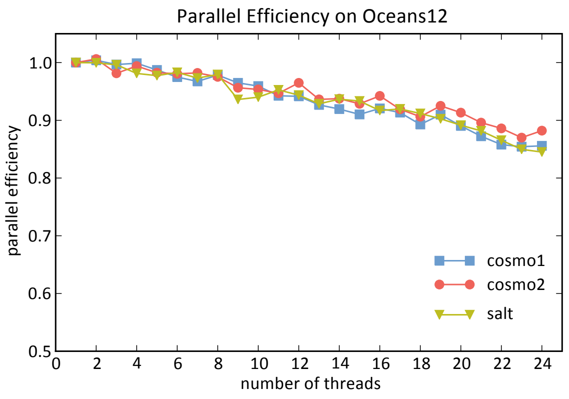


(a)

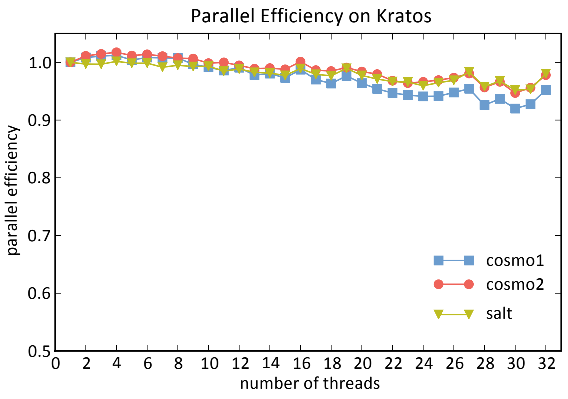


(b)

Figure 9: *The frame time for all parallel rendering methods, including Paraview. The hybrid method clearly outperforms all others. (a) Results from Oceans12. (b) Results from Kratos.*



(a)



(b)

Figure 10: *Parallel efficiency of the hybrid method tested with various datasets. Datasets include cosmo1 (4.5 million triangles), cosmo2 (8 million triangles), and salt (15 million triangles). (a) Results from Oceans12. (b) Results from Kratos.*

the size and location of each partial image tile, we can simply count the number of pixels that each partition will contribute to each row. Once we do this for every image tile and every row, we have an exact number of composite operations that each row requires. Then the final image is split into horizontal strips such that each strip requires approximately the same number of composite operations. This results in a more load-balanced compositing step, with each thread compositing about the same number of pixels. An example of compositing regions before and after being load-balanced is shown in Figure 11.

7.3. Optimizations for NUMA Architectures

While testing our hybrid rendering implementation, one of the issues that we came across was NUMA effects. This happened on our test machine Kratos, which is a NUMA machine. We observed that timings on Kratos had erratic dips which were not present in our other test machine, Oceans12, which uses a UMA architecture. Differences also arose when using an interactive program to rotate and zoom in on geometry. When the user rotated the mesh, the program would have noticeable drops in frame rate, resulting in unsmooth animations. Again, this was evident on Kratos, but not Oceans12.

We came to the conclusion that the cause of this behavior was due to NUMA effects. The details and consequences of a NUMA architecture are described in Section 3. In order to alleviate these issues stemming from NUMA, we explicitly managed memory using the libnuma library [Kle04]. The libnuma library gives the ability to allocate a block of memory on a specific numa node, or interleave memory over all or a subset of numa nodes. With the libnuma library, we implemented several optimizations to minimize NUMA effects.

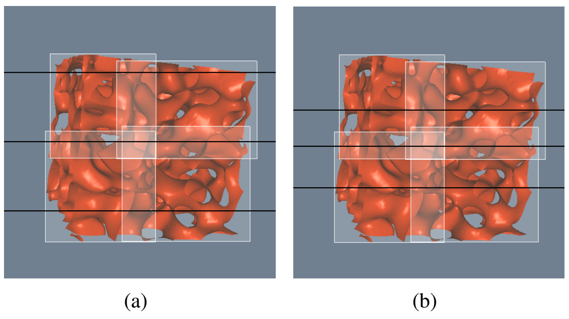


Figure 11: *An example of four composite regions. Composite regions are outlined in black, and rendering partitions are outlined in white. (a) Dividing the image into equal sized regions. Note that the top and bottom strip will composite less pixels than the other two. (b) Load balanced composite partitions. Each strip will composite roughly the same number of pixels.*

First, each thread was pinned to a specific core. Though this is generally the default case, this ensures that multiple threads are not being run on the same core, and this also prevents threads from moving from one core to another during execution. The next change was to ensure that each image tile framebuffer was allocated in the same numa node that the rendering thread will be running on. This is usually the default behavior as long as the thread allocates the framebuffer, but may not be the case if there is not enough memory available on that numa node. Allocating framebuffers such that they are in the local memory of the rendering thread helps performance because most of the access requests of those buffers occur during rendering.

For other data that can be used by any thread at any time, it is best to interleave them over all numa nodes. This includes the framebuffer that will hold the final image, since the majority of memory accesses to this buffer will be during compositing, and compositing assignments will change dynamically between frames. The geometry being rendered is also interleaved over all numa nodes, since different threads will render different triangles depending on the viewing angle.

Overall, implementing these NUMA optimizations resulted in more stable timings. Figure 12 shows maximum render time and maximum composite time with and without any NUMA optimizations. Maximum times are shown because they more accurately reflect frame rate smoothness for an interactive program. The speedups are more stable when using these NUMA-aware modifications, although the max performance does not increase. The changes resulted in noticeably more fluid animations during user interaction.

8. Conclusion

In this paper we investigated and analyzed how established parallel rendering algorithms performed on large shared memory machines. This was done in order to emulate a many-core supercomputer node and test the scalability of software rendering in that environment. We implemented several methods originally designed for distributed memory, including sort-first, sort-last, and a hybrid sort-first and sort-last scheme. Two compositing algorithms, binary swap and direct-send, were also examined. Through benchmarking, we found that sort-first and sort-last had bottlenecks that resulted in poor performance and inadequate scaling. The other method tested, a hybrid combination of sort-first and sort-last rendering, coupled with load-balanced direct-send compositing, resulted in the best performance of parallel software rendering. In order to obtain optimal efficiency, we had to ensure good cache performance, as well as take into account NUMA effects and mitigate them. The result, a scalable software rendering pipeline on shared memory, represents another step toward interactive visualization on the supercomputer. For future work, we plan to investigate software rendering on a multi-node, multi-core supercomputer using a combination of MPI processes and threads.

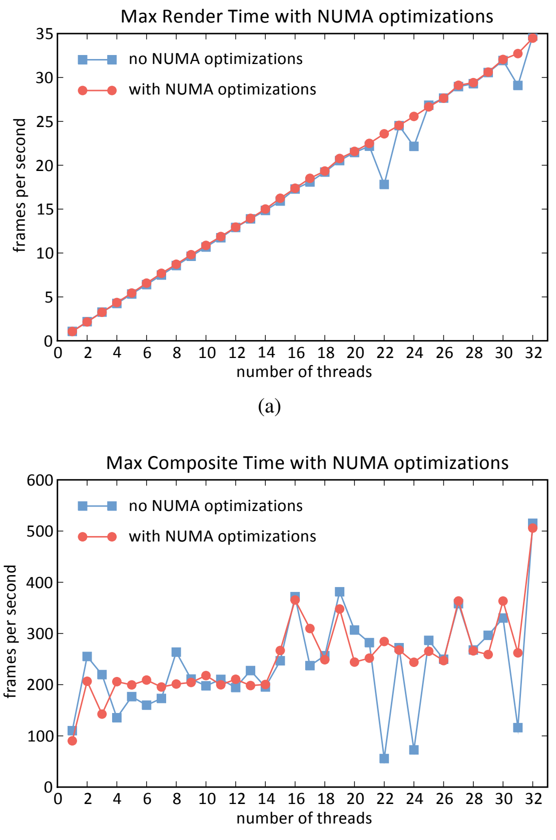


Figure 12: *Results of using NUMA-aware modifications on Kratos. (a) Maximum render times. (b) Maximum composite times.*

References

[ALN*08] AHRENS J., LO L.-T., NOUANESENGSY B., PATCHETT J., MCPHERSON A.: Petascale visualization: Approaches and initial results. In *Ultrascale Visualization, 2008. UltraVis 2008. Workshop on* (November 2008), pp. 24 –28.

[AP98] AHRENS J., PAINTER J.: Efficient sort-last rendering using compression-based image compositing. In *in Proceedings of the 2nd Eurographics Workshop on Parallel Graphics and Visualization* (1998), pp. 145–151.

[CM06] CAVIN X., MION C.: Pipelined sort-last rendering: Scalability, performance and beyond. In *Eurographics Symposium on Parallel Graphics and Visualisation* (May 2006).

[CMF05] CAVIN X., MION C., FILBOIS A.: Cots cluster-based sort-last rendering: performance evaluation and pipelined implementation. In *Visualization, 2005. VIS 05. IEEE* (October 2005), pp. 111 – 118.

[EP08] EILEMANN S., PAJAROLA R.: Direct send compositing for parallel sort-last rendering. In *ACM SIGGRAPH ASIA 2008 courses* (New York, NY, USA, 2008), SIGGRAPH Asia '08, ACM, pp. 39:1–39:8.

[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* (NorrkÃűping, Sweden, 2010), Ahrens J., Debattista K., Pajarola R., (Eds.), Eurographics Association, pp. 1–10.

[ISH98] IGEHY H., STOLL G., HANRAHAN P.: The design of a parallel graphics interface. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1998), SIGGRAPH '98, ACM, pp. 141–150.

[Kle04] KLEEN A.: An numa api for linux, August 2004.

[McC98] MITRA T., CKER CHIUEH T.: Implementation and evaluation of the parallel mesa library. In *ICPADS '98: Proceedings of the 1998 International Conference on Parallel and Distributed Systems* (Washington, DC, USA, 1998), IEEE Computer Society, p. 84.

[MCEF94] MOLNAR S., COX M., ELLSWORTH D., FUCHS H.: A sorting classification of parallel rendering. *IEEE Comput. Graph. Appl. 14* (July 1994), 23–32.

[MPH94] MA K.-L., PAINTER J. S., HANSEN C. D.: Parallel volume rendering using binary-swap compositing. *IEEE Computer Graphics and Applications 14* (1994), 59–68.

[Mue95] MUELLER C.: The sort-first rendering architecture for high-performance graphics. In *In Proceedings of the 1995 Symposium on Interactive 3D Graphics* (1995), ACM Press, pp. 75–84.

[Mue97] MUELLER C.: Hierarchical graphics databases in sort-first. In *Proceedings of the IEEE symposium on Parallel rendering* (New York, NY, USA, 1997), PRS '97, ACM, pp. 49–ff.

[MWP01] MORELAND K., WYLIE B., PAVLAKOS C.: Sort-last parallel rendering for viewing extremely large data sets on tile displays. In *Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphics* (Piscataway, NJ, USA, 2001), PVG '01, IEEE Press, pp. 85–92.

[NO09] NONAKA J., ONO K.: A Decomposition Approach for Optimizing Large-Scale Parallel Image Composition on Multi-Core MPP Systems. In *Eurographics Symposium on Parallel Graphics and Visualization* (Munich, Germany, 2009), Debattista K., Weiskopf D., Comba J., (Eds.), Eurographics Association, pp. 71–78.

[OHA93] ORTEGA F., HANSEN C., AHRENS J.: Fast data parallel polygon rendering. In *Supercomputing '93. Proceedings* (November 1993), pp. 709 – 718.

[PGR*09] PETERKA T., GOODELL D., ROSS R., SHEN H.-W., THAKUR R.: A configurable algorithm for parallel image-compositing applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (New York, NY, USA, 2009), SC '09, ACM, pp. 4:1–4:10.

[PSGL94] PAL SINGH J., GUPTA A., LEVOY M.: Parallel visualization algorithms: performance and architectural implications. *Computer 27*, 7 (July 1994), 45 –55.

[RH00] REINHARD E., HANSEN C.: A comparison of parallel compositing techniques on shared memory architectures. In *In Proceedings of the Third Eurographics Workshop on Parallel Graphics and Visualisation* (2000), pp. 115–123.

[SFLS00] SAMANTA R., FUNKHOUSER T., LI K., SINGH J. P.: Hybrid sort-first and sort-last parallel rendering with a cluster of pcs. In *HWWS '00: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (New York, NY, USA, 2000), ACM, pp. 97–108.

[Whi94] WHITMAN S.: Dynamic load balancing for parallel polygon rendering. *Computer Graphics and Applications, IEEE 14*, 4 (July 1994), 41 –48.

[YWM08] YU H., WANG C., MA K.-L.: Massively parallel volume rendering using 2-3 swap image compositing. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing* (Piscataway, NJ, USA, 2008), SC '08, IEEE Press, pp. 48:1–48:11.