SCI INSTITUTE
TECHNICAL REPORT

An Evaluation of Parallel Grid Construction for Ray Tracing Dynamic Scenes

Thiago Ize, Ingo Wald, Chelsea Robertson, and Steven G. Parker

UUSCI-2006-025

Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, UT 84112 USA

July 10, 2006

Abstract:

We describe and analyze several ways to parallelize the rebuild of a grid acceleration structure used for interactive ray tracing of dynamic scenes. In particular, we present a scalable sort-middle approach that uses a minimum amount of synchronization, scales to many CPUs, and becomes performance limited only by memory bandwidth. With this algorithm, we are able to rebuild a grid in a fraction of a second to ray trace large multi-million triangle scenes at interactive frame rates.

THE UNIVERSITY OF UTAH

An Evaluation of Parallel Grid Construction for Ray Tracing Dynamic Scenes

Thiago Ize Ingo Wald Chelsea Robertson Steven G. Parker

Scientific Computing and Imaging Institute, University of Utah
{thiago|wald|croberts|sparker}@sci.utah.edu

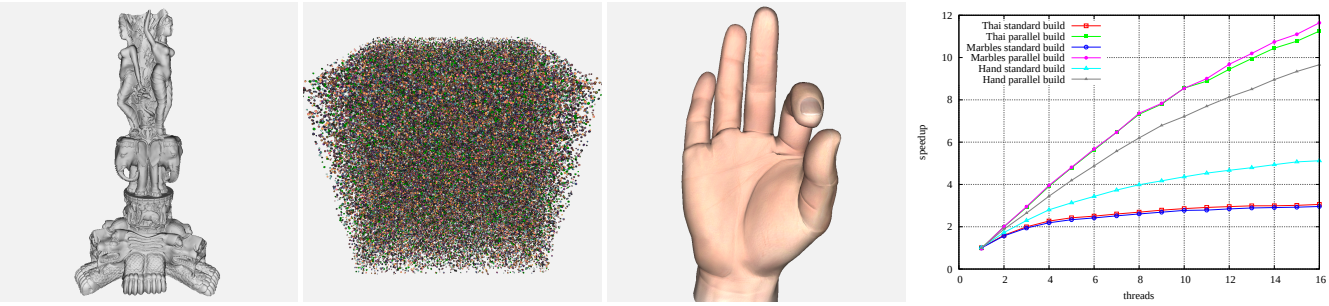

Figure 1: Examples of scenes built using our parallel grid rebuild: a) Thai Statue (10M triangles). b) 125,000 marbles randomly distributed inside a cube (10M triangles). c) Hand (16K triangles). d) Scalability impact of the grid rebuild time on total rendering performance, on a 16-core machine. Data is given for both a non-parallelized grid rebuild, as well as for the sort-middle parallel construction technique evaluated in this paper. Note that our test machine is a NUMA 8-way dual-core machine, in which every pair of CPU cores (0+8,1+9,...) share the same physical off-chip bandwidth, causing a visible bend from 8–9 cores.

Abstract

We describe and analyze several ways to parallelize the rebuild of a grid acceleration structure used for interactive ray tracing of dynamic scenes. In particular, we present a scalable sort-middle approach that uses a minimum amount of synchronization, scales to many CPUs, and becomes performance limited only by memory bandwidth. With this algorithm, we are able to rebuild a grid in a fraction of a second to ray trace large multi-million triangle scenes at interactive frame rates.

1 Introduction

There has been recent progress in interactive ray tracing of fully animated scenes by making the traversal of the grid acceleration structure an order of magnitude faster. This is accomplished by using the coherent grid traversal algorithm and by quickly rebuilding the entire grid each frame [14]. While grid rebuild time for small to moderate scenes is fast enough that only a single processor is needed for grid rebuilding, for larger scenes the grid rebuild time became the dominant cost. For example, the 10 million triangle Thai Statue required almost an entire second to rebuild. This paper describes methods for parallelizing the grid rebuild so that large scenes can be rebuilt and ray traced at interactive frame rates, and also improves the performance for small to medium sized models. With the current trend of multi-core CPUs and industry projections that parallelism is the key to performance in the future, this technique becomes especially important.

Traditional ray tracers are regarded as "embarrassingly parallel", and in fact, the first interactive ray tracers made heavy use of this property [9, 16]; however, recent work in interactive ray tracing of dynamic scenes has made updating the acceleration structure, either BVH or grid, as important as the actual ray tracing [6, 14, 13]. However, updating the acceleration structures in parallel, especially for many CPUs and at interactive frame rates, is not trivial and has not been shown for more than 2 CPUs. Our parallel grid rebuild scales to a large number of CPUs before becoming performance bound by the memory architecture.

2 Background

2.1 Fast Ray Tracing

While the first interactive ray tracers utilized grids [9], algorithmic developments for traversal schemes based on kd-trees [16, 10] have significantly improved the performance of ray tracing. Packet-based ray tracing [16] creates groups of spatially coherent rays that are simultaneously traced through a kd-tree: each ray in the packet performs each traversal operation in lock-step. This packet-based approach enables effective use of SIMD extensions on modern CPUs, increases the computational density of the code, and amortizes the cost of memory accesses across multiple rays. Packet tracing has also lead to a new frustum based traversal algorithm for kd-trees [10] in which a bounding frustum guides the traversal of ray packets, rather than considering each ray individually. The cost of a traversal step is thus independent of the number of rays bounded by the frustum and encourages large ray packets to achieve lower per-ray cost. The frustum based technique has recently been extended to bounding volume hierarchies [13] and multi-level grids [14], both of which support dynamic scenes.

Dynamic Scenes and CGT The Coherent Grid Traversal [14] algorithm is able to interactively ray trace fully dynamic scenes by rebuilding the grid from scratch for each frame and quickly traversing the grid using a frustum-bounded packet. Because the acceleration structure is rebuilt every frame, there are no restrictions on the nature of the motion in a dynamic scene, but the grid rebuild performance becomes the performance limiting factor. The grid rebuild is extremely fast for small to medium sized models, but the cost is linear in the number of triangles and it quickly becomes too slow to allow for interactive ray tracing of multi-million triangle scenes.

Grid Rebuild The grid rebuild as used in [14] consists of three steps: clearing the previous grid cells and macro cells of previous triangle references; inserting the triangles into the grid cells that they are inside of; and building the macro cells.

We optimize the grid clearing step by iterating through the previous macro cells and only clearing the grid cells that have their corresponding macro cell filled. This allows us to quickly skip large empty regions without performing expensive memory reads to determine if a cell is empty. The macro cells are not explicitly cleared because they are overwritten at the end of the grid rebuild step.

Rather than performing a more expensive triangle-in-box test, we optimize the triangle insertion step by inserting the triangles into all the cells overlapped by the triangle's bounding box. This conservative requirement allows the insertion step to run several times faster, while only slowing down rendering by a few percent, when a mailbox algorithm is employed. Furthermore, since the triangle-in-box test slows the rebuild only through computational overhead, including the test would improve parallel *scalability* but would not improve the total *performance* since it requires no additional memory accesses. Furthermore, although the more precise triangle-in-box test does result in fewer cells being written (with a consequent reduction in memory traffic), in practice the minor reduction in memory traffic does not offset the large computational cost for the systems we tested.

The macro cell build is optimized by iterating through the macro cells and checking the grid cells in each macro cell until a cell is found with a triangle. At this point, the macro cell can be marked as full before proceeding to the next macro cell.

2.2 Parallel Computers

Ray tracers have traditionally leveraged the "embarrassingly parallel" nature for static scenes to achieve interactive frame rates; thus, relegating interactive ray tracing to the domain of super computers and clusters. Recently, the combination of Moore's law and algorithmic advances have allowed for personal computers with only a single CPU to catch up with the first interactive ray tracers that ran on super computers. Fully dynamic ray tracers have taken advantage of this by using serial algorithms to update the scene.

Multi-Core Architectures While algorithmic improvements might continue at the same rate, it is unlikely that individual CPU performance will do so; instead, future processor performance improvements will likely come primarily from parallelism, primarily from the addition of multiple cores to the processor. In fact, this is already occurring, as many personal computers are now shipping with dual cores, quad core processors are expected for 2007, the IBM Cell processor has 8 cores, and special purpose processors, such as GPUs, have 48 cores. There has also been an increase in the number of processors on a computer, with many personal computers currently having two processors and high end workstations currently supporting up to 8 processors, each of which can be multi-core. This means that ray tracers on future personal computers must be parallelized in order to take advantage of the faster hardware. While this is not a problem for static scenes, which are "embarrassingly parallel", the current algorithms for handling dynamic scenes must be parallelized and able to run on many CPUs.

Distributed Memory, SMP, and NUMA There are several ways to build a multi-processor machine. The three most common are Symmetric Multi-Processing (SMP) systems, Nonuniform Memory Access (NUMA) systems, and distributed memory systems which are usually implemented as computer clusters. SMP and NUMA are shared memory systems where all the processors are able to access the same memory; distributed memory systems, on the other hand, are not able to share memory at the hardware level and instead must use the relatively slow network connections and software level support. On an SMP system, the computer has a single memory controller and memory bus shared by all of the processors. For memory intensive applications, this quickly becomes a bottleneck as the number of CPUs is increased. NUMA solves this problem by splitting the system into nodes made up of both processors and memory, and connecting these nodes by a fast interconnect. Each node operates like a small SMP system, with a processor capable of very fast access to the local memory on its node, regardless of how many other nodes are in the system, while access to memory on another node happens at slower speeds. As long as a program is written to use only local node memory, the program should scale well on a NUMA system to any number of nodes.

While ray tracers have been written for clusters [15], and memory can be shared through a network connection [4], the parallel algorithms in this paper are too memory intensive to work efficiently on clusters. We thus assume a modern system which is NUMA based with nodes that are SMP based. In the benchmarks below, we use a machine based on 8 AMD Opteron processors, in which each processor contains multiple (2) cores and a single memory controller.

Ideally each node should only use its local memory; but since a thread can traverse any grid cell and intersect any triangle, the grid and triangle data must be shared between nodes. We avoid memory hotspots by interleaving memory allocations in a round-robin fashion by memory page, across all the nodes that might use that memory. Otherwise, we make no particular assumptions about the topology of the underlying architecture and network.

2.3 Parallel Construction of Data Structures

Most fully dynamic ray tracing algorithms make use of either acceleration structure updates [13, 6] or full acceleration structure rebuilds [14, 6]. Of these, only Lauterbach et al. [6] have parallelized the update and rebuild algorithm. However, while their algorithms do update the bounding volumes and perform the BVH tree rebuild in parallel, the authors report that this works because their data is well balanced and they show results only for a dual Xeon computer. It is unclear how well this would scale on more processors or for other scenes.

There has been some research into parallel builds of recursive tree based acceleration structures, such as kd-trees, BVH trees, and octrees, albeit at the time they reached non-interactive performance. Bartz, for instance, showed how to parallelize the construction of octrees [2], and after optimizing memory allocations, on a 16 CPU SGI Challenge, Bartz obtained a parallel build efficiency between 30-80% depending on the model used [1]. Bartz claimed his parallel build algorithm would extend to other tree based structures; however, it is not clear whether two-child trees, such as BVHs and kd-trees, would scale as well as the eight-child octree which partitions naturally to 8 CPUs.

Benthin parallelized a fast kd-tree build for two threads and was able to halve the one second build time using two threads for a 100K Bézier patch model [3]. However, as the number of patches decreased, the scalability also went down, and below 20K patches, adding a second thread gave almost no noticeable benefit. It is likely that adding more threads would require large models in order to continue to scale. It is also unclear whether this would perform as well with simpler triangle meshes.

3 Parallel Grid Rebuild

We rebuild the grid using the same method as [14], except that now the cell clearing, triangle insertion, and macro cell building phases have been parallelized. We also distribute the triangle and grid storage by memory page across all nodes being used. We discuss the parallelization of each of these phases here.

3.1 Parallel Cell Clearing and Macro Cell Build

We parallelize the grid clearing and macro cell building by statically assigning each thread a continuous section of macro cells. By making the assignments over continuous sections, we benefit from cache coherence and a simple iteration over the cells. Furthermore, since each thread handles its own macro cells, and by extension, grid cells, there is no need for mutex locking. Unfortunately, the static assignment and continuous allocation are subject to load imbalance because some sections of the grid may have more triangles than others. However, when we tried to improve this by scheduling small sections of cells round robin to the threads or using dynamic load balancing, the extra overhead and lack of memory locality resulted in a slowdown for the overall clearing and macro cell building phases.

3.2 Parallel Triangle Insertion

The triangle insertion step is equivalent to rasterizing triangles into a regular structure of 3D cells. Aside from being 3D, it is otherwise the same as the traditional rasterization of triangles into 2D pixels. As such, there are several parallels to general parallel rendering methods, such as those in Chromium [5], that we can employ.

In the 3D grid, a triangle that overlaps several cells will be placed in each of those cells. This is akin to the 2D rasterization of a triangle into multiple pixel fragments. The only difference is that in 2D rasterization, the fragments are combined together via z-buffering and blending ops, whereas the grid accumulates them all into a per-cell list of triangles.

Parallel rendering algorithms are classified into three categories: sort-first, sort-middle, and sort-last [8]. In sort-first, each processor is given a portion of the image. The processor determines which triangles are needed to fill that part of the image by iterating through all the triangles. Sort-last assigns each processor a portion of the triangles and determines which pixels those triangles belong in. The sort-middle approach assigns each processor a portion of the image and a portion of the triangles to work with. We can borrow these same concepts and apply them to the grid build, leading to three basic approaches to parallel grid builds:

Sort-First Build In a sort-first parallel build, each thread is assigned a subset of the cells to fill, and then it finds all the triangles that overlap those cells. For example, for P threads, the grid could be subdivided into P subgrids. Each thread iterates over all T triangles, discards those that don't overlap its subgrid, and rasterizes the rest.

The advantage of sort-first is that each cell is written by exactly one thread; there is no write combining. Since all shared data is only read, there are no shared writes at all, requiring synchronization only at the end of the algorithm. However, the disadvantage is that each thread has to read every triangle and at the very least, has to test whether the triangle overlaps its subgrid, hence, its complexity is $O(T+\Theta)$, where Θ is the parallelization overhead. In addition, load balancing is nontrivial when the triangle distribution is uneven.

Sort-Last Build This approach is exactly the opposite of sort-first, in that each thread operates on a fixed set of triangles. The thread partitions the triangle into fragments and stores the fragments in the appropriate cells. The complexity of this algorithm is $O(\frac{T}{P}+\Theta)$ so it will scale as more threads are added.

The advantage to sort-last is that each triangle is touched by exactly one thread, and this method can easily be load balanced. The drawback is the scatter write and write conflicts. Any thread can write to any cell at any time, and different threads may want to write to the same cell. This results in bad memory performance and is bad for streaming architectures (which usually support scatter writes but badly, if at all). In particular, the write combining—multiple threads might want to write to the same cell at the same time—requires synchronization at every write, which is bad for scalability. Alternatively, a compositing operation such as [7, 11] can be employed after building each grid in local memory. While this algorithm is effective for 2D images, the storage requirements of the 3D grid make it prohibitive to story a copy for each thread. Furthermore, the bandwidth requirements for the compositing step would also be significantly larger than for 2D image compositing.

Sort-Middle Build Sort-middle falls in between the previously discussed approaches. Each thread is responsible for a specific set of triangles and grid cells. Each thread iterates over its triangles, partitions each one into fragments, and routes these to the respective threads that fill them into the final cells. Like the sort-last approach, sort-middle also has a complexity of $O(\frac{T}{P}+\Theta)$.

The benefits of sort-middle are that it is relatively easy to load balance, each triangle is read only once, and there are no write conflicts. There is no scatter read nor scatter write, which is good for broadband/streaming architectures. The disadvantage is that it requires buffering of fragments between the two stages. The best way to realize the buffering is not obvious.

From these three methods, we will consider the latter two in more detail. As discussed above, for non-uniform primitive distributions, sort-first is problematic to load balance. Since non-uniform distribution is certainly the case for our applications, load balancing must be addressed. In addition, our applications have a lot of triangles, and reading each triangle several times is likely to quickly produce a memory bottleneck. Thus, we will ignore sort-first and only discuss the other two options below.

4 Sort-Last Approaches

As noted above, each thread works on different triangles. We can easily do this by statically assigning each thread a section of triangles. For each triangle assigned to a thread, it must find the cells the triangle overlaps and add the triangle to those per-cell lists. As a result, all threads have the ability to write to all grid cells, which requires synchronizing writes to each cell. There are several ways to do this as described below.

Single Mutex The most straightforward approach to synchronization is to use a single mutex for inserting triangles into the cells. A single mutex results in one triangle insertion at a time, regardless of what cell it falls inside. This results in an overhead complexity of $\Theta=\frac{T}{M}$, where M is the number of mutexes. Since we are using only one mutex, we get a total complexity of $O(\frac{T}{P}+\frac{T}{1})=O(T)$. Therefore, using the same mutex for every insertion will quickly cause the parallel build to break down into a slow serial build. Clearly this is undesirable for a parallel build, as many of the available resources are not being utilized.

One Mutex Per Cell Rather than share a single mutex, we can use a unique mutex for every grid cell to prevent false conflicts from occurring. The advantage of this method over the naive build is that locking a mutex for one cell won't affect any other cell in the grid. Our complexity now becomes $O(\frac{T}{P}+\frac{T}{M})=O(\frac{T}{P}+\frac{T}{T\cdot\lambda})$, where λ is the grid resolution factor as defined in [14]. Since λ is usually around the magnitude of 1, this simplifies to $O(\frac{T}{P})$. This method eliminates most of the lock contention, but pays a considerable cost for the size of the mutex pool and the memory accesses to it.

Mutex Pool Instead of using a single mutex per cell, it is possible to use a smaller pool of locks, each of which are mapped to multiple grid cells. Since each thread always locks only one mutex at a time, this is deadlock free. We can attempt to maximize some of the memory coherency issues for the mutexes by sharing a mutex among neighboring cells. We accomplish this by having a mutex

per macro cell, which is shared by all the cells in a macro cell. In addition, if we can perform a mutex lock once per triangle instead of once for each cell the triangle overlaps, then the reduction in mutex locks would further improve performance. Given a list of cells that a triangle must be inserted into, we perform a mutex lock on the first cell, insert the triangle into the cell, and then rather than immediately releasing the lock and reacquiring it for the next cell, we check to see whether the next cell shares the same mutex as the current cell and if it does, we do not release the mutex. Once all the cells for that triangle have been visited, we release the lock. Holding a lock over several iterations of cells is not expensive since the only work done in those iterations, aside from the bookkeeping to determine which macro cell we are in, is the critical section which requires the mutex. The complexity of this approach is the same as the one mutex per cell approach.

Grid Merge Rather than have each thread write the triangles to a common grid, which requires at least one synchronization per triangle, we could have each thread write its triangles into a thread local grid. Once the thread local grids have been filled, each thread can be responsible for merging certain sections of the local grids into the main grid. Or, rather than merging during grid build, the P grids could simply all be checked during ray traversal; this would clearly result in a very fast triangle insertion which should scale very well, since aside from the triangle reading, all other memory operations would be local to the node, but at the expense of making ray traversal slower. Furthermore, delaying the merge would require the macro cell build to look at P grids, so the macro cell build would end up with a linear complexity. Both versions require each thread to clear the grid it built, which also results in a linear complexity for the clear stage. Like the two previous approaches, the complexity of the triangle insertion is also $O(\frac{T}{P})$, but the other stages become slower, which results in the overall build and ray traversal time becoming much slower than the other methods.

5 Sort-Middle Grid Build

In our sort-middle approach, we perform a coarse parallel bucket sort of the triangles by their cell location. Then each thread takes a set of buckets and writes the triangles in those buckets to the grid cells. Since each thread handles writing into different parts of the grid, as specified by the buckets, there is no chance of multiple threads writing to the same grid cells; thus mutexes are not required.

More specifically, given T triangles and P threads, each thread takes $\frac{N}{T}$ triangles and sorts these triangles into P thread local buckets (see Figure 2 for an example with $T = 4$). We choose to sort based on the triangle's z cell index modulo the P buckets since that distributes the triangles fairly equally among the buckets, while still allowing us to exploit grid memory coherence when inserting the triangles into the grid because the cells in an xy slice are located in a contiguous section of memory. Note that a triangle may overlap multiple cells, resulting in a triangle appearing in multiple buckets. Once all the threads have finished sorting their $\frac{N}{T}$ triangles into their P buckets, we need to merge the triangles from the buckets into the grid. As there are P sets of P buckets, we do this by reordering the buckets among the threads so that thread i now becomes responsible for the ith bucket from each thread, that is the bucket whose z index is $z \bmod T = i$. Thread i then adds all these triangles into the grid. Since thread i now has *all* buckets with the same z indices (those for which $z \bmod T = i$), thread i will be the only thread writing to those grid cells.

Mutexes are not required at all as each thread is guaranteed to be writing to different grid cells. This method removes the mutex locking overhead, however, the trade off is that it adds extra work in writing to the buckets. The complexity is also $O(\frac{T}{P})$.

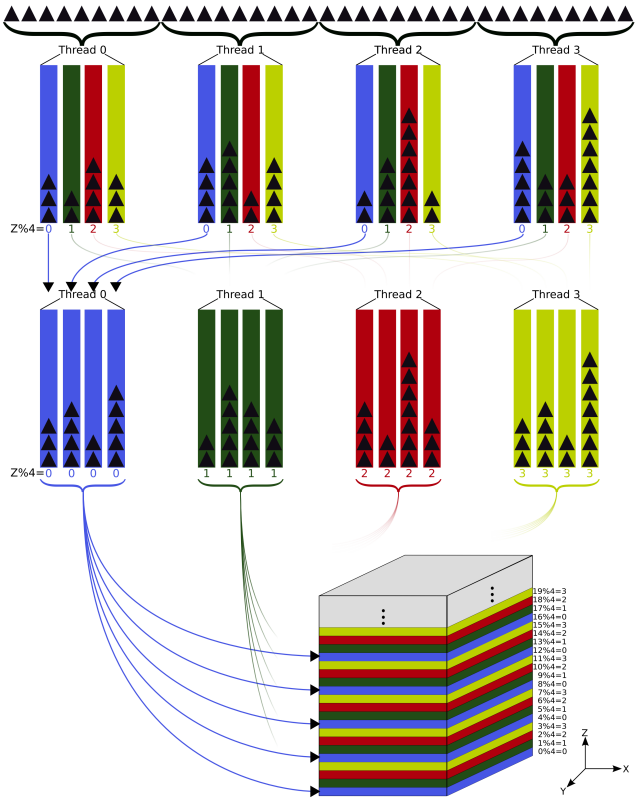

Figure 2: Given 4 processors, in the sort-middle build, the T triangles are equally divided among the 4 threads. Each thread has 4 buckets which are used to sort its $\frac{T}{4}$ triangles, based on the z cell index modulo 4. After all 4 threads have finished sorting, the threads regroup the buckets and fill in the grid. Thread i is responsible for the ith buckets of each of the threads and places all of those triangles into the grid. For example, Thread 0 takes bucket 0 from all 4 threads, Thread 1 takes bucket 1 from all 4 threads, and so on. Since all i buckets correspond to the same z cell indices, there is no overlap in what grid cells they are writing to, thus eliminating the need for mutexes.

6 Results

Unless otherwise noted, we use the 10 million triangle Thai statue at a grid resolution of $192 \times 324 \times 168$ and a macro cell resolution factor of 6 for our measurements.

We use a 16 core computer composed of 8 nodes, each of which has a Dual Core Opteron 880 Processor running at 2.4GHz and 8GB of local memory (a total of 64 GB). Since the cores on a processor must compete with each other for use of the shared memory controller, we distribute the threads across the processors so that a processor will only have both cores filled once all the other processors have at least one core being used. We also interleave memory across the nodes that will need to access it.

6.1 Comparison of Build Methods

In order to determine which build method performs best, we compare the sort-middle, sort-last, and the non-parallel build algorithms. As expected, Figure 3 shows that for one thread the non parallel algorithm performs best, since it has no parallel code overhead. The mutex based builds show that the penalty for locking a mutex for every cell a triangle occupies, or even for every macro cell in the case of the mutex pool, is quite large. The single mutex build performs slightly better than the other mutex builds when there is only one thread, since the mutex is always in cache. Sur-

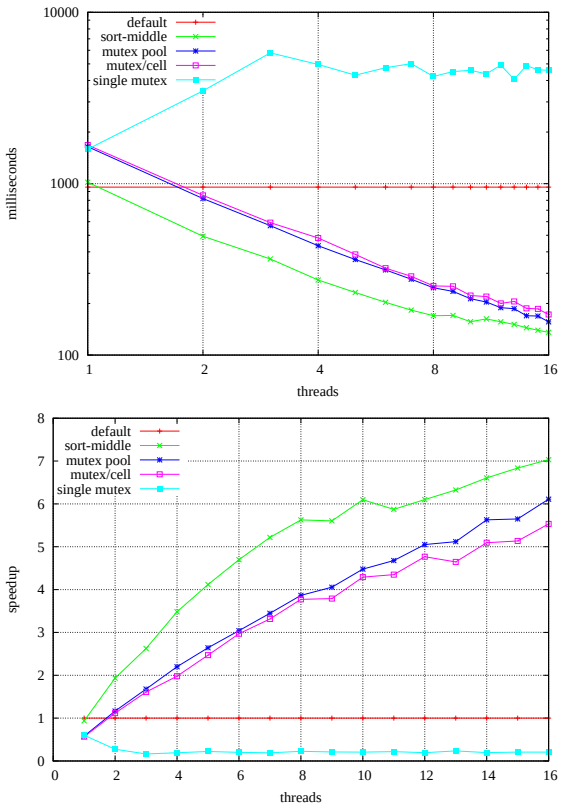

Figure 3: Graphs showing the total rebuild performance (clearing + triangle insertion + macro cell build) for the Thai Statue for: one mutex, a mutex per cell, mutex pools and the default non-parallel rebuild.

prisingly, the sort-middle build performs almost as well as the non parallel build for one thread, despite the extra overhead of passing the triangles through an intermediate buffer.

As we increase the number of threads, we find that using a single mutex scales poorly as expected, and also degrades in performance significantly as more threads are added. This is mainly due to our mutex implementation, which requires an expensive system call when multiple threads try to lock the mutex at the same time. The mutex per cell, mutex pool, and sort-middle builds, on the other hand, do scale to the number of threads; however, since the mutex builds are initially twice as slow with one thread, they are unable to perform better than the sort-middle build. Since the mutex pool build requires fewer mutex locks, it performs slightly better than the mutex per cell build.

6.2 Individual Steps

Examining the individual steps allows us to better understand why scalability drops as the number of threads increases. Figure 4 shows how long the fastest and slowest threads take to clear the grid cells, to build the macro cells, and to perform the two fastest triangle insertion builds: the sort-middle build and the mutex pool build. Increasing the number of threads would ideally cause all of the steps to scale linearly and all of the threads in a step to be equally fast. However, this is not the case due to poor load balancing and memory limitations.

We see that for the Thai statue, the grid clearing phase exhibits rather poor load balancing, with some threads finishing an order of magnitude faster than the slowest thread. This is easily explained by observing that different slices along the z axis contain varying

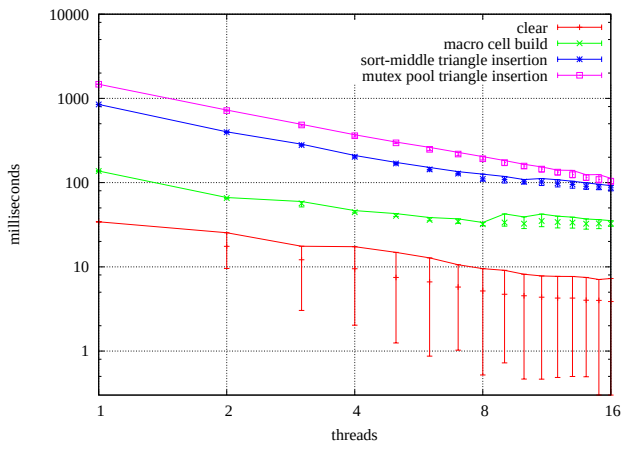

Figure 4: Graph of individual build steps for Thai Statue. The min, average, and max thread times help show how the load balancing performs.

amounts of empty cells, with the end slices being mostly empty and the middle cells being mostly full. Since an empty macro cell allows a thread to skip looking at the individual grid cells, this results in a very fast clear for some threads. We tried other load balancing methods, such as having the threads traverse the cells in strides rather than in continuous chunks, and dynamically assigning cells to threads. While these methods did improve load balancing, the extra overhead caused the overall time to increase.

While the macro cell build shares the same load balancing and a similar iteration scheme as the cell clearing, it is able to load balance fairly well up to 8 threads, after which a limit is encountered, making some of the threads much slower and the fastest threads barely faster. If this was due to poor load balancing, then we would expect a slowdown in the slowest threads to be matched by a speedup in the fastest threads, but this does not occur. Also note that the gap is largest at 9 threads, and starts to go down until 16 threads, at which point the threads all take roughly the same amount of time. This can be explained by noticing that at 8 threads, each node has only one thread on it, so each thread has an equal amount of access to all the resources; but at 9 threads, two threads will exist on one node, so those two threads must share node resources. By 16 threads, all the nodes once again have an equal number of threads (2).

The triangle insertion steps load balance fairly well, with some variation most likely due to threads having triangles that need to be inserted into more cells than usual. However, most of the variation once again comes from unequal contention of the memory resources, which can be seen from the increased range of rebuild times from 9 threads to 15 threads, with 16 threads once again having little variation.

6.3 Memory System Overhead

As Figure 4 showed, the different build stages scale at different rates and none of them scale perfectly. This is due to a combination of parallel algorithm overhead, poor load balancing, and memory system bottleneck. We can measure the memory system impact by placing mutexes around each build stage and only allowing one thread to run at a time. Figure 5 shows the result of this "contention-free" build. Notice that the macro cell build scales the worst when all the threads are contending for resources, but scales extremely well in the "contention-free" build. The sort-middle triangle insertion also scales extremely well, almost achieving perfect scaling, when resource contention is removed. This suggests that when all the threads are working concurrently, the memory traffic is the main

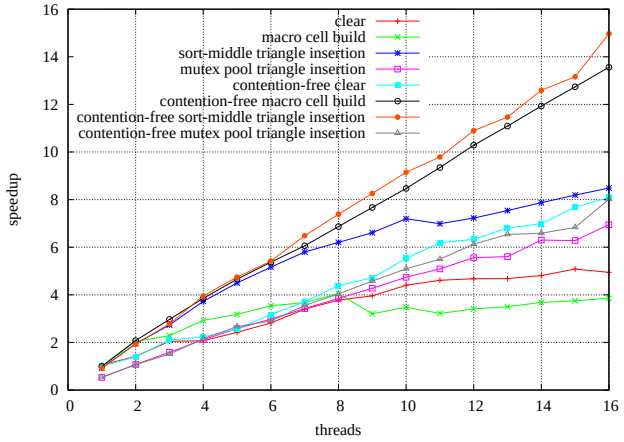

Figure 5: Graph of individual steps for the parallel rebuild and then for the same parallel rebuild where each thread is forced to run by itself so that we can measure build times when there are no resource contentions.

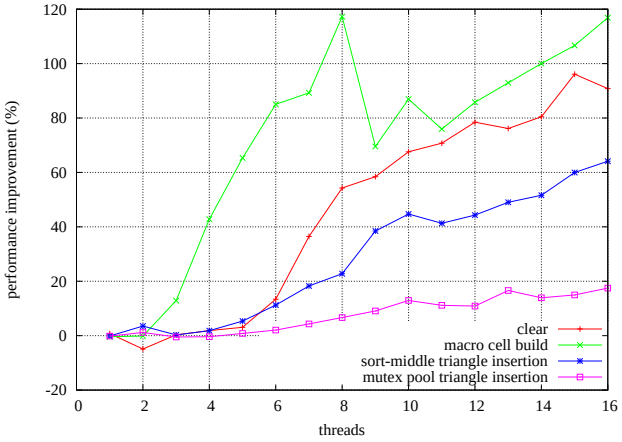

Figure 6: Graph comparing interleaved memory allocations across the nodes to default OS supplied memory allocations.

bottleneck, except for the mutex builds which, in addition to being affected by the same memory penalty, also have a large parallel algorithm overhead due to the mutex locking and unlocking which cuts the triangle insertion performance in half. Thus, it is extremely important to make sure the algorithms make memory accesses as efficiently as possible.

NUMA Our 8 dual-core processor computer is a NUMA system where each processor also contains a memory controller. Normally, when a memory allocation is made, the OS assigns that memory to the node that first touches (reads or writes) the memory. Most grid implementations have a single thread allocate and clear the grid data in one operation, so most of the grid data will reside in a single node. Not only will one thread always have fast local access to the grid data, while the other threads have to remotely access the memory, but all the threads will have to share a single memory controller. If instead we interleave the memory allocations across the N nodes that need to access it, we can ensure that $\frac{1}{N}$th of the memory accesses will be to local node memory, and the memory bandwidth will be spread out among more memory controllers. Figure 6 shows how interleaving memory allocations across the nodes that need to access it versus allowing the OS to manage memory allocations, allows the build steps to scale to many more threads.

This clearly makes a large improvement at pushing back the memory bottleneck. The macro cell build step, for instance, consists of memory reads from many cells, memory writes to every macro cell, and very little computation. When we allow the OS to allocate the memory, the macro cell build becomes memory bound quite quickly; scaling to only two threads before hitting the memory bottleneck, at which point it cannot run faster despite adding more processors. From 5 we saw that the macro cell build actually scales very well if only one thread runs at a time, so we know that this wall must be due entirely to a memory controller saturating its memory bandwidth. Sure enough, when we distribute the memory across all the nodes that might need to access it, the macro cell build time is able to scale up to 8 threads, at which point every node has a thread on it. Adding more threads does not add more memory controllers, limiting scalability due to a memory bottleneck beyond 8 threads.

The other build stages show a similar effect. In the case of the sort-middle build, not interleaving the memory causes a wall to be hit after 8 threads, preventing the sort-middle build from benefiting from more threads, and allowing the mutex pool build to catch up and surpass the sort-middle build. With memory interleaving on the other hand, the sort-middle build is able to scale up to all 16 threads and outperform the mutex pool build. This implies that once the machine becomes memory bound, it might become advantageous to switch from the sort-middle build to the mutex pool build because it requires fewer memory accesses.

6.4 Comparison using other scenes

The Thai statue is a shell of triangles that occupy a small number of grid cells, so we compare this scene to the 10 million triangle marbles scene depicted in Figure 1. This scene consists of 125,000 triangulated spheres randomly distributed inside a cube, and uses a grid resolution of $216 \times 216 \times 216$ and macro cell resolution factor of 6. These two scenes share the same number of triangles, but the distribution of those triangles is very different, thus allowing us to see how robust the parallel grid rebuild is to triangle distribution.

Comparing the Thai statue with the marbles scene in Figure 7 shows the interesting property that the build times for the clearing and macro cell building for the two scenes are almost exactly swapped. For instance, the Thai macro cell build and the marbles cell clearing times are almost identical, even showing the same fluctuations. This is due to the fact that one model has almost all the cells full while the other model has almost none full. Furthermore, the cell clearing and macro cell building share the same basic iteration structure and level of computation, except that the clearing is optimized to skip cells when the macro cell is empty, and the macro cell build is optimized to skip cells when the cells are full. This implies that the total time spent clearing and building macro cells will not vary too much as long as the number of triangles and cells stays the same.

Finally, to show that we do not require millions of triangles in order to scale, in the next section we also compare against the hand model which has only 16K triangles, a grid resolution of $72 \times 36 \times 36$, and a macro cell resolution factor of 6.

6.5 Overall Parallel Rendering Performance

Improving rebuild performance by parallelizing the rebuild is only worthwhile if it ends up improving the overall ray tracing performance. Since most animated scenes require rebuilding the grid every frame, we measure the total ray tracing time for a frame using the three widely different scenes mentioned in the previous section.

As mentioned in [14], we also need to create the derived data for the triangle test described in [12]. Since the triangle rebuild shares some memory reads with the grid rebuild, we can do the triangle rebuild during the triangle insertion stage of the grid build. By doing

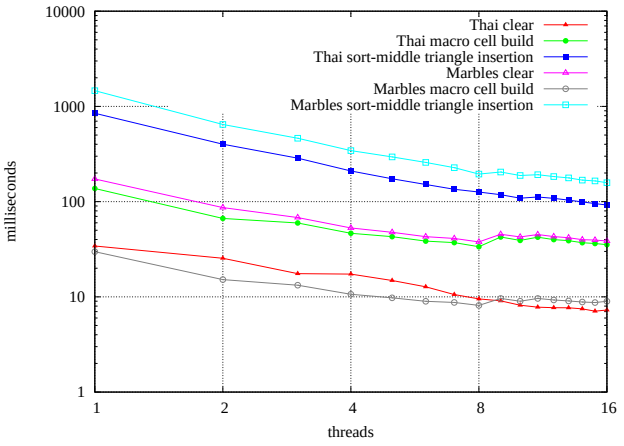

Figure 7: Comparison between the 10M triangle Thai Statue and the 10M triangle Marbles scene for the sort-middle build.

so, not only do we remove a large amount of extra memory reads, we also space out the memory reads with the triangle build computations, which further reduce the memory pressure and allows the overall rebuild time to scale from 7x to 9x when using 16 threads. Even though including the triangle rebuild into the grid rebuild increases the overall performance and allows the grid rebuild to scale better, we chose not to include this in the other measurements since the triangle rebuild is not fundamentally required and other primitives might not require any rebuilding.

We measure the overall impact of the parallel grid rebuild by comparing the various parallel versions to the standard serial grid rebuild used in [14]; both triangle acceleration structure rebuild and rendering are parallelized in both versions, only the grid rebuild method is being varied. Figure 1 shows us how using the parallel grid rebuild allows the overall rendering performance to almost quadruple. For 16 threads, the Thai statue framerate went from 0.78fps with no parallel grid rebuild to 2.86fps with the parallel grid rebuild; the Marbles scene went from 0.50fps to 1.97fps; and the Hand from 78fps to 150fps. The Hand does not scale as well as the other scenes because it is so small; in fact, even without the grid rebuild (rendering only) it is only able to scale by 13x when using 16 threads.

7 Summary and Discussion

We showed how the parallel rendering classifications used for 2D rasterization could be applied to parallel grid building and presented several parallel build methods based on these classifications. We showed that the sort-last approaches require significant thread synchronization, which results in these approaches incurring a large overhead, resulting in low parallel efficiency. These approaches almost double the build time. The sort-first approach is even less effective since it requires each thread to read all the triangles and compute which cells they overlap, making this approach expensive in both computations and memory bandwidth. The sort-last approach was shown to have very little overhead over the non-parallel build, which allowed it to perform better than the other approaches.

Memory bandwidth proves to be a major bottleneck in parallel grid rebuilds where the amount of computation performed is low and the amount of data that needs to be processed is high. Even the simple to parallelize grid cell clearing and macro cell building stages quickly started to show the memory pressure after just a couple threads were used. We were able to push back some of the memory pressure on our NUMA computer by interleaving the memory allocations across the nodes that access that memory. However, as we demonstrated with our "contention-free" parallel rebuilds, increasing the memory bandwidth would allow all our parallel rebuild methods to scale even better. While the sort-middle build did perform best on our test computer, it is possible that for a computer with more limited memory bandwidth, the sort-last approach could perform better since it requires fewer memory accesses.

As we showed in Figure 1, using a parallel grid rebuild allowed the overall render time to quadruple for the two large 10M triangle scenes and double for the very small 16K triangle scene. Thus, a parallel grid rebuild is important for any dynamic scene rendered with a grid when multiple CPUs are available. Since many consumer computers already ship with dual-core processors and some with two dual-core processors, and furthermore there is a strong trend towards more multi-core processors, it is imperative that dynamic ray tracers be able to take advantage of the extra processor resources.

With memory bandwidth being the limiting factor, adding more cores to a chip might result in only minor performance gains if the memory bandwidth to the chip isn't also increased. However, if we compare the IBM Cell processor with 8 cores and a memory bandwidth of 25.6GB/s to the dual core Opteron's bandwidth of 6.4GB/s, we see that at least up to 8 cores the memory bandwidth per core should at least stay the same. Furthermore, since the memory for the 8 cores is all local to the processor, memory latency will be equally fast for each core, unlike with the NUMA based system used for these tests. However, the memory architecture of the Cell would introduce other challenges in this algorithm.

As computers follow the trend of getting more computational cores, a parallel grid rebuild algorithm will allow us to take advantage of these resources to speed up small to medium sized dynamic scenes, as well as providing scalable performance for large dynamic scenes.

Acknowledgments

This work was funded by the U.S. Department of Energy through the Center for the Simulation of Accidental Fires and Explosions, under grant W-7405-ENG-48 and through the Department of Energy SciDAC program. The Thai Statue is available through the Stanford Scanning Repository. The Hand model is from the Utah Animation Repository.

References

[1] D. Bartz. Optimizing memory synchronization for the parallel construction of recursive tree hierarchies. In *Proceedings of Eurographics Workshop on Parallel Graphics and Visualization*, pages 53–60, 2000.

[2] D. Bartz, R. Grosso, T. Ertl, and W. Straer. Parallel construction and isosurface extraction of recursive tree structures. In *Proceedings of WSCG'98*, volume 3, 1998.

[3] Carsten Benthin. *Realtime Ray Tracing on current CPU Architectures*. PhD thesis, Saarland University, 2006.

[4] D. E. DeMarle, C. Gribble, and S. Parker. Memory-Savvy Distributed Interactive Ray Tracing. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 93–100, 2004.

[5] Greg Humphreys, Mike Houston, Ren Ng, Sean Ahern, Randall Frank, Peter Kirchner, and James T. Klosowski. Chromium: A stream processing framework for interactive graphics on clusters of workstations. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2002)*, 21(3):693–702, July 2002.

[6] Christian Lauterbach, Sung-Eui Yoon, David Tuft, and Dinesh Manocha. RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs. Technical Report 06-010, Department of Computer Science, University of North Carolina at Chapel Hill, 2006.

[7] Kwan-Liu Ma, James S. Painter, Charles D. Hansen, and Michael F. Krogh. Parallel volume rendering using binary-swap compositing. *IEEE Comput. Graph. Appl.*, 14(4):59–68, 1994.

[8] Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A sorting classification of parallel rendering. *IEEE Comput. Graph. Appl.*, 14(4):23–32, 1994.

[9] Steven G. Parker, William Martin, Peter-Pike J. Sloan, Peter Shirley, Brian E. Smits, and Charles D. Hansen. Interactive ray tracing. In *Proceedings of Interactive 3D Graphics*, pages 119–126, 1999.

[10] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-level ray tracing algorithm. In *Proceedings of ACM SIGGRAPH*, pages 1176–1185, 2005.

[11] Akira Takeuchi, Fumihiko Ino, and Kenichi Hagihara. An improved binary-swap compositing for sort-last parallel rendering on distributed memory multiprocessors. *Parallel Comput.*, 29(11-12):1745–1762, 2003.

[12] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Saarland University, 2004.

[13] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics*, (conditionally accepted, to appear), 2006.

[14] Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G. Parker. Ray tracing animated scenes using coherent grid traversal. *ACM Transactions on Graphics*, 25(3):485–493, 2006. (Proceedings of ACM SIGGRAPH).

[15] Ingo Wald, Philipp Slusallek, and Carsten Benthin. Interactive Distributed Ray Tracing of Highly Complex Models. In Steven J. Gortler and Karol Myszkowski, editors, *Rendering Techniques*, Proceedings of the 12th Eurographics Workshop on Rendering Techniques, London, UK, June 25-27, 2001, pages 274–285. Springer, 2001.

[16] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum*, 20(3):153–164, 2001. (Proceedings of Eurographics).