Graphics Hardware (2008)
David Luebke and John D. Owens (Editors)

# On Dynamic Load Balancing on Graphics Processors

Daniel Cederman[†] and Philippas Tsigas[‡]

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
{cederman,tsigas}@cs.chalmers.se

**Abstract**

*To get maximum performance on the many-core graphics processors it is important to have an even balance of the workload so that all processing units contribute equally to the task at hand. This can be hard to achieve when the cost of a task is not known beforehand and when new sub-tasks are created dynamically during execution. With the recent advent of scatter operations and atomic hardware primitives it is now possible to bring some of the more elaborate dynamic load balancing schemes from the conventional SMP systems domain to the graphics processor domain.*

*We have compared four different dynamic load balancing methods to see which one is most suited to the highly parallel world of graphics processors. Three of these methods were lock-free and one was lock-based. We evaluated them on the task of creating an octree partitioning of a set of particles. The experiments showed that synchronization can be very expensive and that new methods that take more advantage of the graphics processors features and capabilities might be required. They also showed that lock-free methods achieves better performance than blocking and that they can be made to scale with increased numbers of processing units.*

Categories and Subject Descriptors (according to ACM CCS): C.1.2 [Processor Architectures]: Multiple Data Stream Architectures (Multiprocessors)

## 1. Introduction

Today's graphics processors have ventured from the multi-core to the many-core domain and, with many problems in the graphics area being of the so called embarrassingly parallel kind, there is no question that the number of processing units will continue to increase.

To be able to take advantage of this parallelism in general purpose computing, it is imperative that the problem to be solved can be divided into sufficiently fine-grained tasks to allow the performance to scale when new processors arrive with more processing units. However, the more fine-grained a task set gets, the higher the cost of the required synchronization becomes.

Some problems are easily divided into fine-grained tasks with similar processing time, such as for example the N-body problem [NHP07]. But with other problems this information can be hard to predict and new tasks might be created dynamically during execution. In these cases dynamic load balancing schemes are needed which can adapt to shifts in work load during runtime and redistribute tasks evenly to the processing units.

Several popular load balancing schemes have been hard to implement efficiently on graphics processors due to lack of hardware support, but this has changed with the advent of scatter operations and atomic hardware primitives such as Compare-And-Swap. It is now possible to design more advanced concurrent data structures and bring some of the more elaborate dynamic load balancing schemes from the conventional SMP systems domain to the graphics processor domain.

The load balancing in these schemes is achieved by having a shared data object that stores all tasks created before and under execution. When a processing unit has finished its

[†] Supported by Microsoft Research through its European PhD Scholarship Programme

[‡] Partially supported by the Swedish Research Council (VR)

work it can get a new task from the shared data object. As long as the tasks are sufficiently fine-grained the work load will be balanced between processing units.

The methods used for synchronizing the memory accesses to these shared data objects can be divided into two categories, *blocking* and *non-blocking*. Blocking methods use mutual exclusion primitives such as locks to only allow one processing unit at a time to access the object. This is a pessimistic conflict control that assumes conflicts even when there are none.

Non-blocking methods on the other hand employ an optimistic conflict control approach allowing several processing units to access the shared data object at the same time and suffering delays because of retries only when there is an actual conflict. This feature allows non-blocking algorithms to scale much better when the number of processing units increases. Section 3 discusses more about the differences between the two methods.

In the paper we compare four different methods of dynamic load balancing:

**Centralized Blocking Task Queue** Tasks are stored in a queue using mutual exclusion.

**Centralized Non-blocking Task Queue** Tasks are stored in a non-blocking (lock-free) queue.

**Centralized Static Task List** Tasks are stored in a static list.

**Task Stealing** Each processing unit has a local double ended queue where it stores new tasks. Tasks can be stolen from other processing units if required.

The first method is lock-based while the other three are non-blocking ones. The schemes are evaluated on the task of creating an octree partitioning of a set of particles. An octree is a tree-based spatial data structure that repeatedly divides the space by half in each direction to form eight octants. The fact that there is no information beforehand on how deep each branch in the tree will be, makes this a suitable problem for dynamic load balancing.

### 1.1. Related Work

Load balancing is a very basic problem and as such there has been a lot of work done in the area. In this subsection we present a small set of papers that we deem most relevant to our work.

Korch et al. have made a comparison between different dynamic balancing schemes on the radiosity problem [KR03]. Heirich and Arvo have compared static and dynamic load balancing methods for ray-tracing and found static methods to be inadequate [HA98]. Foley and Sugerman have implemented an efficient kd-tree traversal algorithm for graphics processors and point out that load balancing is one of the central issues for any highly parallel ray-tracer [FS05].

When it comes to using task stealing for load balancing, Blumofe and Leiserson gave "the first provably good task stealing scheduler for multithreaded computations with dependencies" [BL94]. Arora et al. have presented an efficient lock-free method for task stealing where no synchronization is required for local access when the local stack has more than one elements [ABP98]. Soares et al have used task stealing as load balancing for voxel carving with good performance [SMRR07].

In the following section the system model is presented. In Section 3 the need for non-blocking algorithms is discussed. Section 4 has an overview of the load balancing methods compared in the paper and Section 5 describes the octree partitioning task used for evaluation. In section 6 we describe the test settings and discuss the result.

## 2. The System Model

In this work all the load balancing methods have been implemented using CUDA, a compiler and run-time from NVIDIA that can compile normal C code into binary or bytecode that can be executed on CUDA-enabled graphics processors.

**General Architecture** The graphics processors consist of up to 16 multiprocessors each, which can perform SIMD (Single Instruction, Multiple Data) instructions on 8 memory positions at a time. Each multiprocessor has 16 kB of a very fast local memory that allows information to be communicated between threads running on the same multiprocessor.

**Memory Access** Each multiprocessor has access to the large, but relatively slow, global memory. It is possible to speed up access to the memory by arranging memory accesses in such a way so that the graphics processor can coalesce them into one big read or write operation. This is done by letting threads access consecutive memory locations with the first location being a multiple of 16 times the word size read or written.

The NVIDIA 8600GTS and newer graphics processors support atomic operations such as CAS (Compare-And-Swap) and FAA (Fetch-And-Add) when accessing the memory, which can be used to implement efficient parallel data structures.

**Scheduling** The graphics processor uses a massive number of threads to hide memory latency instead of using a cache memory. These threads are divided into *thread blocks* of equal size where all threads in a specific thread block is assigned to a specific multiprocessor. This allows threads in the same thread block to communicate with each other using the fast local memory and a special hardware barrier function which can be used to synchronize all threads in a block.

Thread blocks are run from start to finish on the same multiprocessor and can't be swapped out for another thread

block. If all thread blocks started can't fit on the available multiprocessors they will be run sequentially and will be swapped in as other thread blocks complete. A common scenario is to divide work into small tasks and then create a thread block for each task. When a multiprocessor has finished with one thread block/task, it schedules a new one and can thus achieve a relatively balanced load.

The threads in a thread block are scheduled according to which *warp* they are a part of. A warp consists of 32 threads with consecutive id, such as 0..31 and 32..63. The graphics processor tries to execute the threads in a warp using SIMD instructions, so to achieve optimal performance it is important to try to have all threads in a warp perform the same instruction at any given time. Threads in the same warp that perform different operations will be serialized and the multiprocessor will not be used to its full capabilities.

## 3. Synchronization

Synchronization schemes for designing shared data objects can be divided into three categories:

**Blocking** Uses mutual exclusion to only allow one process at a time to access the object.

**Lock-Free** Multiple processes can access the object concurrently. At least one operation in a set of concurrent operations finishes in a finite number of its own steps.

**Wait-Free** Multiple processes can access the object concurrently. Every operation finishes in a finite number of its own steps.

The term non-blocking is also used in order to describe methods that are either lock-free or wait-free.

The standard way of implementing shared data objects is often by the use of basic synchronization constructs such as locks and semaphores. Such blocking shared data objects that rely on mutual exclusion are often easier to design than their non-blocking counterpart, but a lot of time is spent in the actual synchronization, due to busy waiting and convoying. Busy waiting occurs when multiple processes repeatedly checks if, for example, a lock has been released or not, wasting bandwidth in the process. This lock contention can be very expensive.

The convoying problem occurs when a process (or warp) is preempted and is unable to release the lock quick enough. This causes other processes to have to wait longer than necessary, potentially slowing the whole program down.

Non-blocking shared data objects, on the other hand, allows access from several processes at the same time without using mutual exclusion. So since a process can't block another process they avoid convoys and lock contention. Such objects also offer higher fault-tolerance since one process can always continue, whereas in a blocking scenario, if the process holding the lock would crash, the data structure would be locked permanently for all other processes. A non-blocking solution also eliminates the risk of deadlocks, cases where two or more processes circularly waits for locks held by the other.

## 4. Load Balancing Methods

This section gives an overview of the different load balancing methods we have compared in this paper.

### 4.1. Static Task List

The default method for load balancing used in CUDA is to divide the data that is to be processed into a list of blocks or tasks. Each processing unit then takes out one task from the list and executes it. When the list is empty all processing units stop and control is returned to the CPU.

This is a lock-free method and it is excellent when the work can be easily divided into chunks of similar processing time, but it needs to be improved upon when this information is not known beforehand. Any new tasks that are created during execution will have to wait until all the statically assigned tasks are done, or be processed by the thread block that created them, which could lead to an unbalanced workload on the multiprocessors.

The method, as implemented in this work, consists of two steps that are performed iteratively. The only data structures required are two arrays containing tasks to be processed and a tail pointer. One of the arrays is called the in-array and only allows read operations while the other, called the out-array, only allows write operations.

In the first step of the first iteration the in-array contains all the initial tasks. For each task in the array a thread block is started. Each thread block then reads task $i$, where $i$ is the thread block ID. Since no writing is allowed to this array, there is no need for any synchronization when reading.

If any new task is created by a thread block while performing its assigned task, it is added to the out-array. This is done by incrementing the tail pointer using the atomic FAA-instruction. FAA returns the value of a variable and increments it by a specified number atomically. Using this instruction the tail pointer can be moved safely so that multiple thread blocks can write to the array concurrently.

The first step is over when all tasks in the in-array has been executed. In the second step the out-array is checked to see if it is empty or not. If it is empty the work is completed. If not, the pointers to the out- and in-array are switched so that they change roles. Then a new thread block for each of the items in the new in-array is started and this process is repeated until the out-array is empty. The algorithm is described in pseudo-code in Algorithm 1.

The size of the arrays needs to be big enough to accommodate the maximum number of tasks that can be created in a given iteration.

**Algorithm 1** Static Task List Pseudocode.

```
function DEQUEUE(q, id)
    return q.in[id]
function ENQUEUE(q, task)
    localtail ← atomicAdd(&q.tail, 1)
    q.out[localtail] = task
function NEWTASKCNT(q)
    q.in, q.out, oldtail, q.tail ← q.out, q.in, q.tail, 0
    return oldtail
procedure MAIN(task_init)
q.in, q.out ← newarray(maxsize), newarray(maxsize)
q.tail ← 0
enqueue(q, task_init)
blockcnt ← newtaskcnt(q)
while blockcnt ≠ 0 do
    run blockcnt blocks in parallel
        t ← dequeue(q, TB_id)
        subtasks ← doWork(t)
        for each nt in subtasks do
            enqueue(q, nt)
    blocks ← newtaskcnt(q)
```

**Algorithm 2** Blocking Dynamic Task Queue Pseudocode.

```
function DEQUEUE(q)
    while atomicCAS(&q.lock, 0, 1) == 1 do
    if q.beg! = q.end then
        q.beg++
        result ← q.data[q.beg]
    else
        result ← NIL
    q.lock ← 0
    return result
function ENQUEUE(q, task)
    while atomicCAS(&q.lock, 0, 1) == 1 do
    q.end++
    q.data[q.end] ← task
    q.lock ← 0
```

### 4.2. Blocking Dynamic Task Queue

In order to be able to add new tasks during runtime we designed a parallel dynamic task queue that thread blocks can use to announce and acquire new tasks.

As several thread blocks might try to access the queue simultaneously it is protected by a lock so that only one thread block can access the queue at any given time. This is a very easy and standard way to implement a shared queue, but it lowers the available parallelism since only one thread block can access the queue at a time, even if there is no conflict between them.

The queue is array-based and uses the atomic CAS (Compare-And-Swap) instruction to set a lock variable to ensure mutual exclusion. When the work is done the lock variable is reset so that another thread block might try to grab it. The algorithm is described in pseudo-code in Algorithm 2.

Since it is array-based the memory required is equal to the number of tasks that can exist at any given time.

### 4.3. Lock-free Dynamic Task Queue

A lock-free implementation of a queue was implemented to avoid the problems that comes with locking and also in order to study the behavior of lock-free synchronization on graphics processors. A lock-free queue guarantees that, without using blocking at all, at least one thread block will always succeed to enqueue or dequeue an item at any given time even in presence of concurrent operations. Since an operation will only have to be repeated at an actual conflict it can deliver much more parallelism.

The implementation is based upon the simple and efficient array-based lock-free queue described in a paper by Zhang and Tsigas [TZ01]. A tail pointer keeps track of the tail of queue and tasks are then added to the queue using CAS. If the CAS-operation fails it must be due to a conflict with another thread block, so the operation is repeated on the new tail until it succeeds. This way at least one thread block is always assured to successfully enqueue an item.

The queue uses lazy updating of the tail and head pointers to lower contention. Instead of changing the head/tail pointer after every enqueue/dequeue operation, something that is done with expensive CAS-operations, it is only updated every $x$:th time. This increases the time it takes to find the actual head/tail since several queue positions needs to be checked. But by reading consecutive positions in the array, the reads will be coalesced by the hardware into one fast read operation and the extra time can be made lower than the time it takes to try to update the head/tail pointer $x$ times.

Algorithm 3 gives a skeleton of the algorithm without the essential optimizations. For a detailed and correct description, please see the original paper [TZ01]. This method requires just as much memory as the blocking queue.

### 4.4. Task Stealing

Task stealing is a popular load balancing scheme. Each processing unit is given a set of tasks and when it has completed them it tries to steal a task from another processing unit which has not yet completed its assigned tasks. If a unit creates a new task it is added to its own local set of tasks.

One of the most used task stealing methods is the lock-free scheme by Arora et al. [ABP98] with multiple array-based double ended queues (*deques*). This method will be

**Algorithm 3** Lock-free Dynamic Task Queue Pseudocode.

```
function DEQUEUE(q)
    oldbeg ← q.beg
    lbeg ← oldbeg
    while task = q.data[lbeg] == NIL do
        lbeg++
    if atomicCAS(&q.data[lbeg], task, NIL) != task then
        restart
    if lbeg mod x == 0 then
        atomicCAS(&q.beg, oldbeg, lbeg)
    return task
function ENQUEUE(q, task)
    oldend ← q.end
    lend ← oldend
    while q.data[lend] != NIL do
        lend++
    if atomicCAS(&q.data[lend], NIL, task) != NIL then
        restart
    if lend mod x == 0 then
        atomicCAS(&q.end, oldend, lend)
```

referred to as ABP task stealing in the remainder of this paper.

In this scheme each thread block is assigned its own deque. Tasks are then added and removed from the tail of the deque in a LIFO manner. When the deque is empty the process tries to steal from the head of another process' deque.

Since only the owner of the deque is accessing the tail of the deque there is no need for expensive synchronization when the deque contains more than one element. Several thread blocks might however try to steal at the same time, requiring synchronization, but stealing is assumed to occur less often than a normal local access. The implementation is based on the basic non-blocking version by Arora et al. [ABP98]. The stealing is performed in a global round robin fashion, so thread block $i$ looks at thread block $i+1$ followed by $i+2$ and so on.

The memory required by this method depends on the number of thread blocks started. Each thread block needs its own deque which should be able to hold all tasks that are created by that block during its lifetime.

## 5. Octree Partitioning

To evaluate the dynamical load balancing methods described in the previous section, they are applied to the task of creating an octree partitioning of a set of particles [SG91]. An octree is a tree-based spatial data structure that recursively divides the space in each direction, creating eight octants. This is done until an octant contains less than a specific number of particles.

The fact that there is no information beforehand on how deep each branch in the tree will be, makes this a suitable problem for dynamic load balancing.

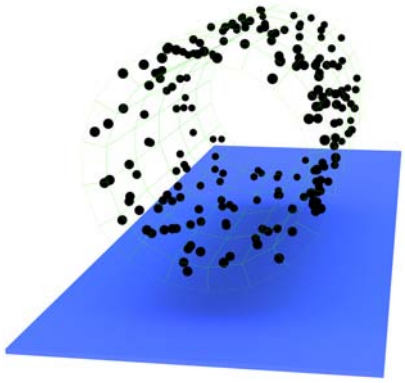

**Figure 1:** *Tube Distribution (An example with 200 elements.)*

A task in the implementation consists of an octant and a list of particles. The thread block that is assigned the task will divide the octant into eight new octants and count the number of elements that are contained in each. If the count is higher than a certain threshold, a new task is created containing the octant and the particles in it. If it is lower than the threshold, the particle count is added to a global counter. When the counter reaches the total number of particles the work is completed.

Particles found to be in the same octant are moved together to minimize the number of particles that has to be examined for further division of the octant. This means that the further down the tree the process gets, the less time it takes to complete a task.

This implementation of the octree partitioning algorithm should not be seen as the best possible one, but only as a way to compare the different work balancing methods.

## 6. Experimental Evaluation

Two different graphics processors were used in the experiments, the 9600GT 512MiB NVIDIA graphics processor with 64 cores and the 8800GT 512MiB NVIDIA graphics processor with 112 cores.

We used two input distributions, one where all particles were randomly picked from a cubic space and one where they were randomly picked from a space shaped like a geometrical tube, see Figure 1.

All methods where initialized by a single iteration using one thread block. The maximum number of particles in any given octant was set to 20 for all experiments.

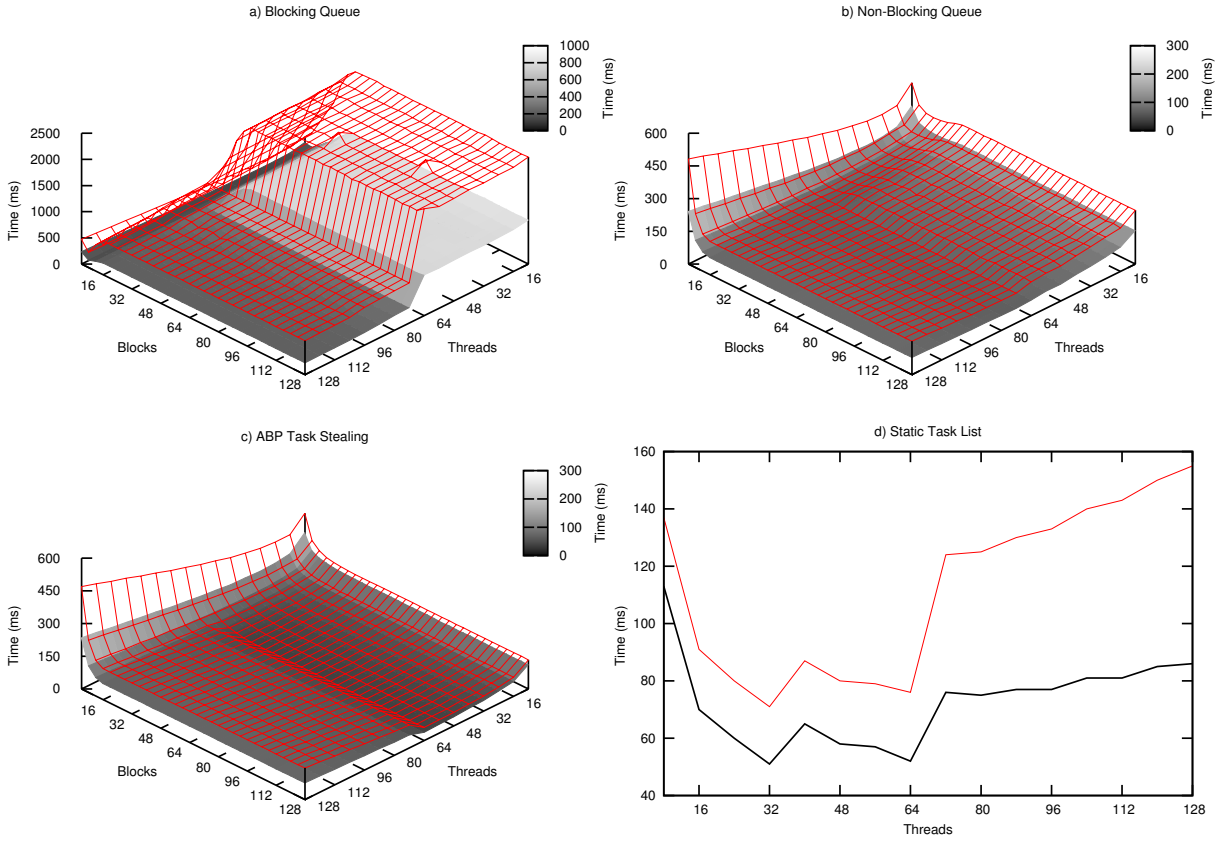


**Figure 2:** *Comparison of load balancing methods on the 8800GT. Shows the time taken to partition a* ***Uniform*** *(filled grid) and* ***Tube*** *(unfilled grid) distribution of half a million particles using different combinations of threads/block and blocks/grid.*

### 6.1. Discussion

Figure 2 and 3 shows the time it took to partition two different particle sets on the 8800GT and 9600GT graphics processors using each of the load balancing methods while varying the number of threads per block and blocks per grid. The static method always uses one block per task and is thus shown in a 2D graph.

Figure 4 shows the time taken to partition particle sets of varying size using the combination of threads per block and blocks per grid found to be optimal in the previously described graph. The shapes of the graphs maps nicely to the total number of tasks created for each distribution and particle count, as shown in Figure 5. The task count is higher with the tube distribution since the octree is more unbalanced.

Figure 2 (a) clearly shows that using less than 64 threads with the blocking method gives us the worst performance in all of the experiments. This is due to the expensive spinning on the lock variable. These repeated attempts to acquire the lock causes the bus to be locked for long amounts of times during which only 32-bit memory accesses are done. With more than 64 threads the number of concurrent thread blocks is lowered from three to one, due to the limited number of available registers per thread, which leads to less lock contention and much better performance. This shows that the blocking method scales very poorly. In fact, we get the best result when using less than ten blocks, that is, by not using all of the multiprocessors! The same can be seen in Figure 3 (d) and 4 where the performance is better on the graphics processor with fewer cores. We used 72 threads and 8 blocks to get the best performance out of the blocking queue when comparing it with the other methods.

The non-blocking queue-based method, shown in Figure 2 (b), can take better advantage of an increased number of blocks per grid. We see that the performance increases quickly when we add more blocks, but after around 20 blocks the effect fades. It was expected that this effect would be visible until we increased the number of blocks beyond 42, the number of blocks that can run concurrently when using less than 64 threads. This means that even though its performance is much better than the its blocking counterpart, it still does not scale as well as we would have wanted. This can also clearly be seen when we pass the 64 thread bound-

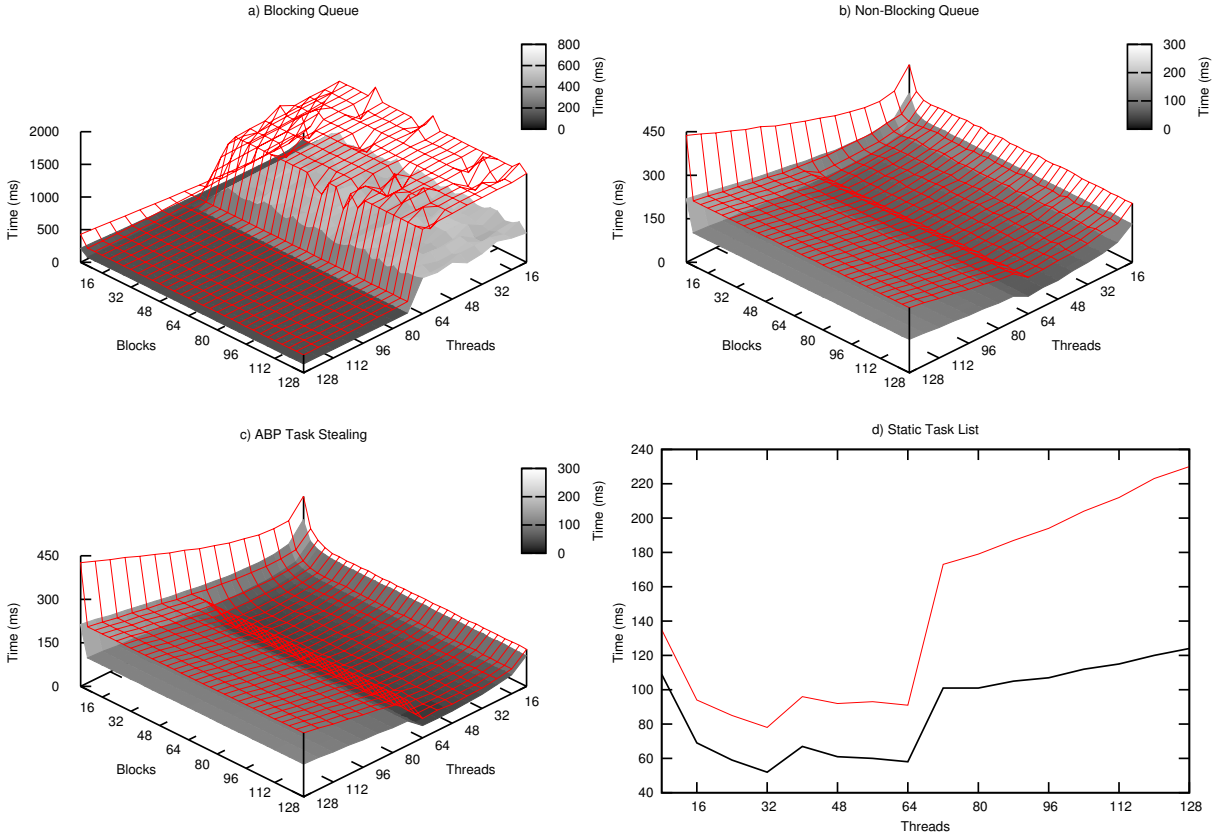


**Figure 3:** *Comparison of load balancing methods on the 9600GT. Shows the time taken to partition a* ***Uniform*** *(filled grid) and* ***Tube*** *(unfilled grid) distribution of half a million particles using different combinations of threads/block and blocks/grid.*

ary and witness an increase in performance instead of the anticipated drop. On the processor with fewer cores, Figure 3 (b), we do get a drop, indicating that conflicts are expensive for this queue implementation. Looking at Figure 4 we see the same thing, the non-blocking queue performs better on the processor with fewer cores. The measurements were done using 72 threads and 20 blocks.

In Figure 2 (c) we see the result from the ABP task stealing and it lies more closely to the ideal. Adding more blocks increases the performance until we get to around 30 blocks. Adding more threads also increases performance until we get the expected drop 64 threads per block. We also get a slight drop after 32 threads since we passed the warp size and now have incomplete warps being scheduled. Figure 4 shows that the work stealing gives great performance and is not affected negatively by the increase in number of cores on the 8800GT. When we compared the task stealing with the other methods we used 64 threads and 32 blocks.

In Figure 2 (d) we see that the static method shows similar behavior as the task stealing. When increasing the number of threads used by the static method from 8 to 32 we get a steady increase in performance. Then we get the expected drops after 32 and 64, due to incomplete warps and less concurrent thread blocks. Increasing the number of threads further does not give any increase in speed as the synchronization overhead in the octree partitioning algorithm becomes dominant. The optimal number of threads for the static method is thus 32 and that is what we used when we compared it to the other methods in Figure 4.

As can be seen in Figure 2 and 3, adding more blocks than needed is not a problem since the only extra cost is an extra read of the finishing condition for each additional block.

Figure 5 shows the total number of tasks created for each distribution and particle count. As can be seen, the number of tasks increases quickly, but the tree itself is relatively shallow. A perfectly balanced octree has a depth of $\log_8(n/t)$, where $t$ is the threshold, which with 500,000 elements and a threshold of 20 gives a depth of just 4.87. In practice, for the tube distribution, the static queue method required just 7 iterations to fully partition the particles and after just 3 iterations there were a lot more tasks than there were processing units.

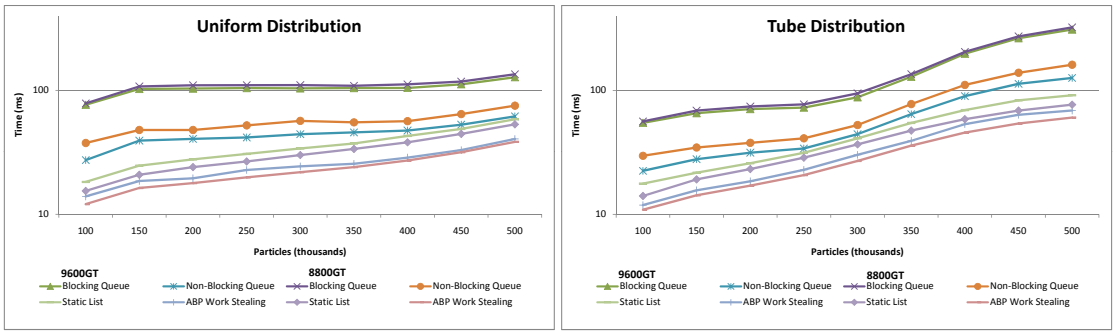


**Figure 4:** *A comparison of the load balancing methods on the* ***uniform*** *and* ***tube*** *distribution.*

## 7. Conclusions

We have compared four different load balancing methods, a blocking queue, a non-blocking queue, ABP task stealing and a static list, on the task of creating an octree partitioning of a set of particles.

We found that the blocking queue performed poorly and scaled badly when faced with more processing units, something which can be attributed to the inherent busy waiting. The non-blocking queue performed better but scaled poorly when the number of processing units got too high. Since the number of tasks increased quickly and the tree itself was relatively shallow the static queue performed well. The ABP task stealing method perform very well and outperformed the static method.

The experiments showed that synchronization can be very expensive and that new methods that take more advantage of the graphics processors features and capabilities might be required. They also showed that lock-free methods achieves better performance than blocking and that they can be made to scale with increased numbers of processing units.

### Future Work

We are planning to compare the load balancing methods used in the paper on other problems, such as global illumination. Based on the conclusions from this work we are trying to develop new methods, tailored to graphics processors, for load balancing.

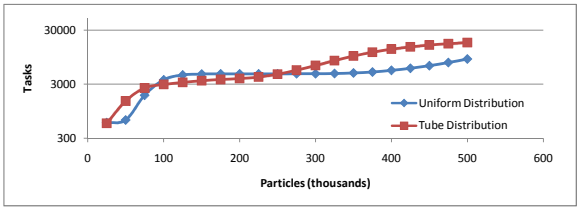


**Figure 5:** *The total number of tasks caused by the two distributions.*

## References

[ABP98] ARORA N. S., BLUMOFE R. D., PLAXTON C. G.: Thread Scheduling for Multiprogrammed Multiprocessors. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures* (1998), pp. 119–129.

[BL94] BLUMOFE R., LEISERSON C.: Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico.* (1994), pp. 356–368.

[FS05] FOLEY T., SUGERMAN J.: KD-tree acceleration structures for a GPU raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (2005), pp. 15–22.

[HA98] HEIRICH A., ARVO J.: A Competitive Analysis of Load Balancing Strategies for Parallel Ray Tracing. *J. Supercomput. 12*, 1-2 (1998), 57–68.

[KR03] KORCH M., RAUBER T.: A comparison of task pools for dynamic load balancing of irregular algorithms: Research articles. *Concurrency and Computation: Practice & Experience 16*, 1 (2003), 1–47.

[NHP07] NYLAND L., HARRIS M., PRINS J.: Fast N-Body Simulation with CUDA. In *GPU Gems 3*. Addison-Wesley, 2007, ch. 31, pp. 677–695.

[SG91] SHEPHARD M., GEORGES M.: Automatic three-dimensional mesh generation by the finite Octree technique. *International Journal for Numerical Methods in Engineering 32* (1991), 709–749.

[SMRR07] SOARES L., MÉNIER C., RAFFIN B., ROCH J.-L.: Work Stealing for Time-constrained Octree Exploration: Application to Real-time 3D Modeling. In *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization* (2007).

[TZ01] TSIGAS P., ZHANG Y.: A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems. In *Proceedings of the thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures* (2001), pp. 134–143.