CHAPTER

35 Dynamic Load Balancing Using Work-Stealing

Daniel Cederman and Philippas Tsigas

In this chapter, we present a methodology for efficient load balancing of computational problems that can be easily decomposed into multiple tasks, but where it is hard to predict the computation cost of each task, and where new tasks are created dynamically during runtime. We present this methodology and its exploitation and feasibility in the context of graphics processors. Work-stealing allows an idle core to acquire tasks from a core that is overloaded, causing the total work to be distributed evenly among cores, while minimizing the communication costs, as tasks are only redistributed when required. This will often lead to higher throughput than using static partitioning.

35.1 INTRODUCTION

To achieve good performance on graphics processors, with their many-core architecture, it is important that the work to be done can be distributed evenly to all available cores. It is also vital that the solution can scale well when newer graphics processors arrive with an increased number of cores.

Many problems can be decomposed relatively easily into multiple tasks that all have the same computation cost. These tasks can then be distributed evenly to all cores. Depending on the number of cores, the work can be decomposed into a varying number of tasks, allowing the solution to scale. There is, however, a large category of problems where (i) it is difficult to predict how long a task will take to complete and (ii) new tasks are created dynamically during runtime. For these irregular problems, it becomes difficult to achieve a uniform utilization of the cores using a static assignment of the work to the cores. Instead, the need arises for a more dynamic solution that can adapt to changes in the workload at runtime.

In popular GPU computing environments such as CUDA and OpenCL, one can achieve load balancing by decomposing the work into more tasks than can be scheduled concurrently, allowing cores that finish early to acquire new unfinished tasks. This eases the problem of scheduling tasks with unknown computation cost. However, it requires that all the tasks are available prior to the invocation of the kernel. To perform the subtasks created during runtime requires waiting for the kernel as a whole to finish, and then to perform these new tasks in a new kernel invocation or for each core to perform all of its own subtasks. Either way tends to lead to uneven workloads, as can be seen in Figure 35.1(a).

A solution to this problem is to have a dynamic work-pool of tasks, from which each core can receive tasks to perform, and to which it can announce new subtasks. This will allow cores to acquire

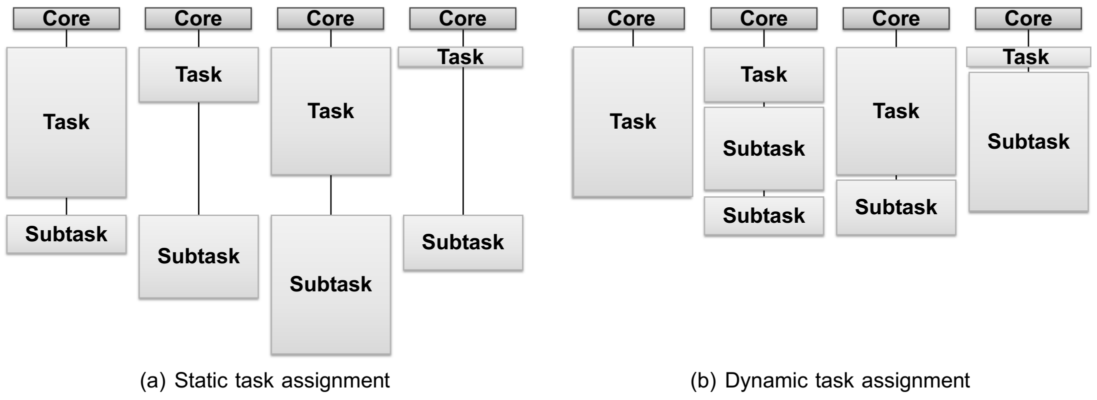


FIGURE 35.1

Comparison between static and dynamic task assignments.

new tasks as soon as they become available, instead of having to wait for a new kernel invocation. In this chapter we will investigate how to implement such a work-pool in CUDA using two different approaches, one dynamic, using work-stealing, and one static approach.

35.2 CORE METHOD

There are many different ways to implement a dynamic work-pool. Most of them, however, require the use of atomic primitives to handle synchronization, which was a problem earlier when few graphics processors supported them. Some of the atomic primitives could be emulated using the graphics processors memory access semantics [1], but today most newer graphics processors supports advanced atomic operations natively. These primitives, such as Compare-And-Swap and Fetch-And-Add, have made it possible to implement some of the more advanced data-structures used in several well known dynamic load balancing schemes.

A popular technique used in some of these schemes is *work-stealing* [2], which is used extensively in the Cilk programming language and has been a part of its success [3]. In a work-stealing scheme, each thread has its own pool of tasks. When a thread has finished a task, it acquires a new one from its own work-pool, and, when a new subtask is created, the new task is added to the same work-pool. If a thread discovers that it has no more tasks in its own work-pool, it can try to *steal* a task from the work-pool of another thread. This will allow a thread to always have tasks to perform, while minimizing communication among threads.

One such scheme is the popular lock-free work-stealing algorithm by Arora et al. [4]. A paper comparing different load balancing schemes have shown that it works well on graphics processors and can achieve better performance than other schemes [5]. In the following sections we will describe the components of this design and how it can be used to implement dynamic load balancing in CUDA that can outperform other static load balancing techniques.

35.3 ALGORITHMS AND IMPLEMENTATIONS

The basic idea behind work-stealing is to assign to each thread its own work-pool, which is then used primarily by that thread. This allows for newly spawned subtasks to be handled by the same thread that handled their parent task. As subtasks often access the same data as their parent, this will usually lead to better cache utilization. When a thread no longer has any tasks to perform, it will try to steal one from another thread's work-pool. For a thread to know if there are any tasks left to steal, one needs to create a condition that can be checked to see if the total work has been completed or not. This condition is often trivial to create, so if the problem faced can be easily decomposed into multiple tasks, work-stealing is an efficient scheme to achieve an even load balance.

In the following sections we will give an overview of the work-stealing scheme followed by a detailed explanation and motivation of the design of a well known algorithm for work-stealing. But before we do that, we will describe an alternative to work-stealing, which we will later use as a baseline for our experiments.

35.3.1 Static Assignment

To evaluate the performance of the work-stealing scheme, we have implemented a load balancing scheme using a static assignment of the tasks to each thread block. The reason that we talk about thread blocks here instead of threads is that for some applications (e.g., where control flow across tasks can diverge heavily) it can be more efficient to have multiple threads within a block collaborate on a single task rather than to have each thread work on its own task.

The work-pool in this scheme is implemented using two arrays (Figure 35.2). The first array holds all the tasks to be performed and the other array holds subtasks created at runtime. In the first iteration, the input array holds all initial tasks. The array is then partitioned so that each thread block gets an equal number of tasks. Since no writing is allowed to the input array, there is no need for any synchronization.

When new tasks are created during runtime, they are written to the output array with the help of the atomic primitive Fetch-And-Add (FAA). This primitive atomically increments the value of a variable and can thus be used to find a unique position in the array. When all tasks have been completed, the two arrays switch roles and the kernel is invoked again. This is repeated until no more new tasks are created.

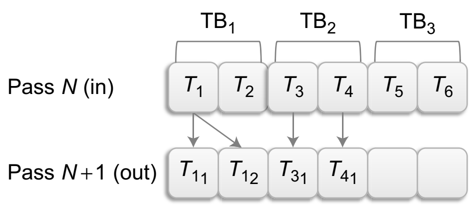


FIGURE 35.2

The two arrays used for static assignment.

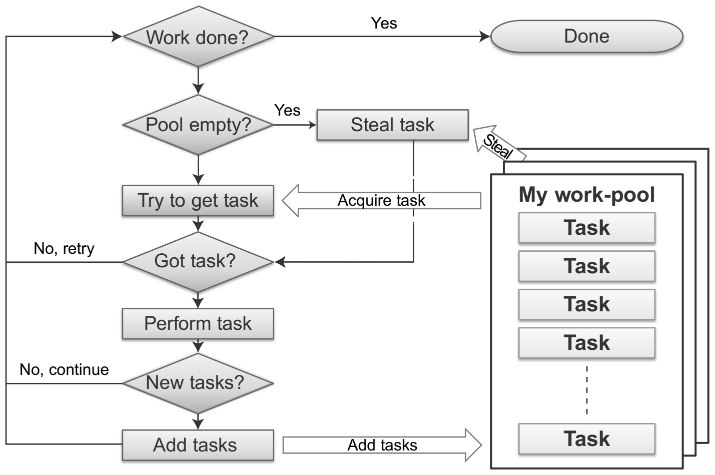


FIGURE 35.3

Task sharing using work-pools and work-stealing.

35.3.2 Work-Stealing

A general overview of the work-stealing scheme can be seen in Figure 35.3. Every thread block is assigned its own work-pool with, potentially, some initially allotted tasks in it. Each thread block tries to repeatedly acquire new tasks to perform from its work-pool. If a new task is created during execution, it is added to the thread block's own work-pool. If a thread block fails to get a task, it checks a condition to see if all work is done. If it is not, the thread block tries to *steal* tasks from the other thread block's work-pools.

There are several ways to implement work-pools for work-stealing. We have decided to use the algorithm by Arora et al. [4], which is a popular method on conventional systems and has two interesting features. The first one is that it is lock-free and the second one that it avoids expensive atomic operations in the most common cases. We will try to motivate the importance of these two features for graphics processors in the two following sections.

35.3.2.1 *Lock-Freedom*

As multiple thread blocks can access the same work-pool during stealing, the underlying data structure must be able to synchronize the concurrent operations made to it. The basic way to synchronize operations is the use of spinlocks. Using spinlocks for synchronization is, however, very expensive and does not scale well, especially on graphics processors.[1]

[1]GPUs prior to the NVIDIA Fermi architecture do not have writable caches, so for those GPUs, repeated checks to see if a lock is available or not require expensive repeated accesses to the GPU's main memory. While Fermi GPUs do support writable caches, the use of locks is still not recommended, as there is no guarantee that the thread scheduler will be fair, which can make it difficult to write deadlock-free locking code. OpenCL explicitly disallows locks for these and other reasons.

A better way, then, is to take advantage of lock-free techniques [6, 7]. Lock-freedom is a progress guarantee for algorithms that states that at any given time, at least one thread block will always make computational progress, regardless of the progress or status of any other thread block. This means that a thread block never has to wait for a lock to be released, so no matter the scheduling, at least one thread block will always be able to finish its operation in a bounded amount of time. One common design method to make an algorithm lock-free is to take all the changes that need to be performed in mutual exclusion and rewrite them so that they can be performed with just one atomic instruction.

35.3.2.2 *Atomic Primitives*

Atomic primitives were an important addition to the instruction set of the graphics processors, as using only read and write operations is not enough for nonblocking synchronization. In the set of atomic primitives, the Compare-And-Swap (CAS) operation is among the most powerful. The CAS operation is used to atomically change the value of a variable, if and only if it currently has the value given as a parameter to the operation. The CAS operation can be seen as a word level transaction supported by the hardware. However, for the hardware to be able to perform the CAS operation atomically, the memory bus needs to be locked to guarantee that no other thread block is concurrently writing to the same memory location. This is expensive, and an atomic operation will be many times slower than a normal read or write operation. Because of this, it is recommended to avoid using atomic operations when possible. In the Fermi architecture the performance of the atomic operations has been increased, but a normal read or write will always be faster.

The algorithm by Arora et al. only uses atomic operations during stealing and when there is just one element left in the work-pool, cases that are not common.

35.3.3 The Work-Stealing Algorithm

The work-stealing algorithm uses double-ended queues (deques) for work-pools and each thread block is assigned its own unique deque. A deque is a queue where it is possible to enqueue and dequeue from both sides, in contrast to a normal queue where you enqueue on one side and dequeue on the other.

Tasks are added and removed from the *tail* of the deque in a Last-In-First-Out (LIFO) manner. When the deque is empty, the thread block tries to steal from the *head* of another thread blocks 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, and for this case synchronization is required, but stealing is expected to occur less often than a normal local access.

A double-ended queue is depicted in Figure 35.4. We base our implementation on an array that holds the tasks and have a head and a tail pointer that points to the first and last task in the deque. The head pointer is divided into two fields due to the ABA-problem (described in Section 35.3.3.4) that can occur if the head pointer is written to by two different thread blocks. These data structures are defined in Listing 35.1.

As each thread block needs to have its own deque, we have to allocate memory for as many deques as we have thread blocks. We cannot use the shared memory to store the deques, as other thread blocks need to be able to access them to steal tasks. The maximum number of tasks to make room for in the deque will have to be decided for the specific application and must be decided on beforehand. The tasks can be of any size. If they are larger than a single word, one should try to make sure that multiple threads read them in a coalesced manner.

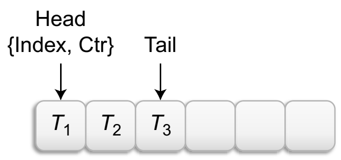


FIGURE 35.4

Double-ended queues used to represent work-pools.

```
struct Head {
    unsigned short index;
    unsigned short ctr;
}

struct Deque {
    Head head;
    unsigned int tail;
    Task tasks[MAX_TASKS];
}

Deque deques[MAX_THREAD_BLOCKS];
```

Listing 35.1. Data structures needed for the work-pools.

```
push(task)
    tasks[tail] = task;
    tail++;
```

Listing 35.2. Code for the **push** operation.

35.3.3.1 Push

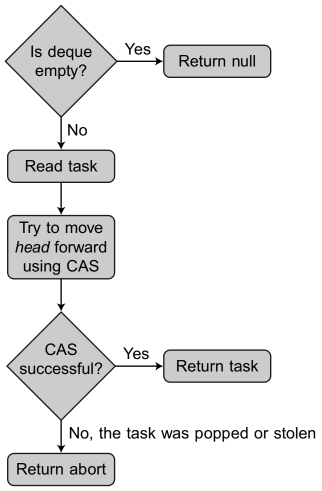
The push operation (Listing 35.2) is used by the owner thread block to add new tasks to the tail end of the deque. As `tail` is only written to by the owner thread block, there is no need for any synchronization. The task is simply written to where `tail` is pointing to and then `tail` is incremented to point to the next empty slot.

35.3.3.2 Steal

The steal operation (Listing 35.3) tries to take a task from the head of the deque. It first checks if the deque is empty or not by comparing `tail` to `head` (step ①). If `tail` is equal to or lower than `head`, the deque is empty or became empty while performing the comparison. In this case, the steal operation returns `null`, and the caller will have to try to steal from another deque.

If the deque is not empty, the steal operation will read the task that `head` is pointing to (step ②) and try to move `head` to point to the next element in the array (step ③). Multiple thread blocks might try to

```
Task steal()
   Head oldHead, newHead;
   Task task;

   oldHead = head; ①
   if(tail ≤ oldHead.index )
      return null;

   task = tasks[oldHead.index]; ②

   newHead = oldHead; ③
   newHead.index++;
   if( CAS(&head, oldHead, newHead) )
      return task;

   return abort;
```



Listing 35.3. Code for the **steal** operation.

steal the task at the same time and the owner might try to pop it, so we need to use a synchronization primitive to make sure that only one thread block is successful in acquiring the task. Using CAS to update `head` gives us this guarantee.

If the CAS is successful, it means that the steal operation as a whole was successful and it can return the stolen task. If the CAS failed, it means that some other thread block stole or popped the task, and the calling thread block will have to try to steal again, perhaps this time from another dequeue.

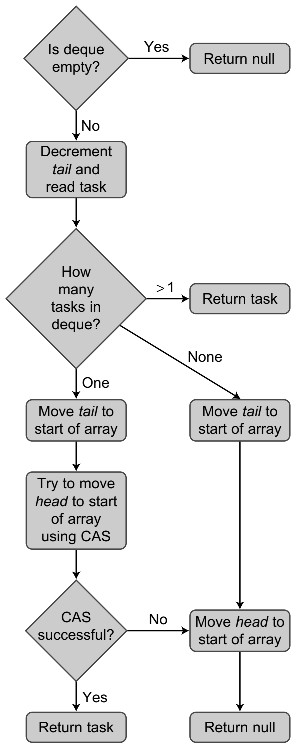
35.3.3.3 *Pop*

The pop operation (Listing 35.4) tries to take a task from the tail end of the deque. It first tries to find out if the deque is empty by checking whether `tail` points to the first element of the array or not (step ①). If it does, the deque is empty and the pop operation simply returns `null`. The calling thread block will now have to decide if it wants to steal a task from another deque.

If `tail` is not pointing to the first element, then the pop operation decrements `tail` by one and reads the task that it is now pointing at (step ②). As it is only the owner of the deque that can change `tail`, there is no need for any synchronization. It then makes a local copy of `head` and compares it to `tail` (step ③). There are now three different possible scenarios:

1. `tail` is strictly *larger* than the local copy of `head`. Any new steal operation that is invoked will see that the tail has been moved in step ②, so it will not try to steal the task. However, a concurrent steal might have already performed the comparison at step ① in the steal operation (Listing 35.3)-and not noticed that the tail has changed. This is no problem, as no matter how many concurrent steals are in progress, they can only move `head` one step forward in total before they have to look at `tail` again. And when they do, they will find that the task is no longer in the deque. There is

```
Task pop()
   Head oldHead, newHead;
   unsigned int oldTail;
   Task task;

   if(tail == 0) ①
      return null;

   tail−−; ②
   task = tasks[tail];

   oldHead = head;
   if(tail > oldHead.index) ③
      return task;

   oldTail = tail; ④
   tail = 0;
   newHead.index = 0;
   newHead.ctr = oldHead.ctr + 1;

   if( oldTail == oldHead.index ) ⑤
      if( CAS(&head, oldHead, newHead) )
         return task;

   head = newHead; ⑥
   return null;
```


Is deque empty?
Yes
Return null
No
Decrement tail and read task
How many tasks in deque?
>1
Return task
None
One
Move tail to start of array
Move tail to start of array
Try to move head to start of array using CAS
CAS successful?
No
Move head to start of array
Yes
Return task
Return null

Listing 35.4. Code for the **pop** operation.

thus no possibility for a conflict with another thread block and the task is successfully popped and can be returned by the operation.

2. `tail` is strictly *smaller* than the local copy of `head`. This scenario happens when a steal operation took the last element in the deque before the call to the pop operation was made. In this case, `tail` and `head` are changed to both point to the beginning of the array (steps ④ and ⑥), so as not to waste space when new tasks are pushed to the deque. As the deque is empty, the operation returns `null`.
3. `tail` is *equal* to the local copy of `head`. This means that we read the last available task in the deque. This task might be concurrently stolen by another thread block, so to make sure that only one thread block will be able to acquire the task, we use the CAS primitive to move `head` to the beginning of the deque (step ⑤). This is done, as mentioned before, to not waste space when new tasks are pushed to deque, but has the additional benefit that it will prevent other thread blocks from stealing the task if it is successful. If it fails, it means that the task was stolen. In this case the deque is empty and we can move `head` to the beginning of the array without using CAS (step ⑥). If it was successful, we can return the popped task.

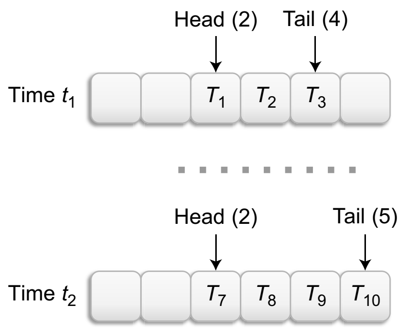


FIGURE 35.5

The CAS operation cannot differentiate between the head pointer at time t_1 and at time t_2, opening up for the possibility that a task can be stolen twice.

35.3.3.4 ***ABA-problem***

One downside of the CAS operation is that it is susceptible to the ABA-problem. In most lock-free algorithms, the CAS operation is used to make sure that a write to a variable will only succeed if no other thread block has changed the value of the variable in the meantime. But if the value has changed from A to B and then back to A again, there is no way for the CAS operation to discover this. The ABA-problem is thus something that needs to be considered whenever using CAS.

In the context of work-stealing, the ABA-problem can occur when the pop operation moves `head` to point to the beginning of the array while another thread block is concurrently trying to steal a task. As an example, in the scenario in Figure 35.5, thread block X is trying to steal task T_1 at time t_1. It reads `head` and learns that it should try to steal from position 2. But just as it is about to change `head` using the CAS operation, it gets swapped out by the scheduler. In the meantime, the deque experiences an arbitrary number of push, pop, and steal operations, leaving the deque at a completely different state at time t_2, which is when thread block X is swapped in again. Unfortunately, `head` is still pointing to position 2 in the array, so thread block X will believe that it successfully stole task T_1, when in fact task T_1 has already been performed by another thread block.

This problem is avoided by adding a counter to `head`, which is incremented every time that `head` is moved to the beginning of the deque. This guarantees that a task can never be stolen more than once, as `head` will always be unique. A problem with this approach is that eventually the counter will wrap around and start counting from zero again, but the probability of this occurring can be made very small by setting a good counter size. Note that `tail` is only written to by the owner thread block, so it is not susceptible to the ABA-problem the way `head` is.

35.3.3.5 ***Parameters***

There is very little parallelism in the code for the queue management work done *within* the individual thread blocks. Most of the work is performed by thread 0 and does not take advantage of the SIMD instructions to any larger degree. The number of threads in a thread block should therefore be decided mainly by the need of the application that uses the dynamic load balancing. When it comes to selecting the number of thread blocks, one should try to run at least as many thread blocks as can run concurrently. If one starts more thread blocks than can run concurrently, the excess thread blocks will immediately see that the work has been completed when they start and exit right away.

The main things to consider when using the dynamic load balancing scheme presented in this chapter is how to divide the problem into tasks, what information the tasks should contain and how large they should be. Too small tasks will create extra overhead, while too large will cause imbalance in the load. It is also required to have a condition to check, to see if the total work has been done or not. In the following section, we present two example applications to show how this can be done.

35.4 CASE STUDIES AND EVALUATION

To evaluate the dynamic load balancing scheme, we implemented two applications that fulfill the criteria that they should dynamically create new tasks that can be hard to predict the total computation cost for. The first is an implementation of a computer opponent for a four-in-a-row game and the second is an octree partitioner that divides a set of particles in 3-D space into a hierarchical data structure. In the first application, each task has about the same computation time, but the number of sub-tasks that each task will spawn is unknown. In the second application, the computation time for each task is also unknown.

For comparison we also performed the experiments using the static assignment scheme presented in Section 35.3.1. We performed measurements using different numbers of thread blocks on a NVIDIA GTX280 graphics processor with 30 multiprocessors. The measurements taken were the total amount of tasks per millisecond that were performed, as well as the memory consumption for the two different load balancing schemes.

35.4.1 Four-in-a-Row

Four-in-a-row is played by two players, where in each turn a player drops a token in one of seven slots in a 6×7 grid. The first player to get four of his own tokens in a row wins the game. Figure 35.6 shows a possible game scenario.

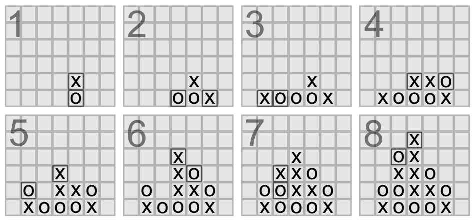


FIGURE 35.6

A possible game scenario for four-in-a-row.

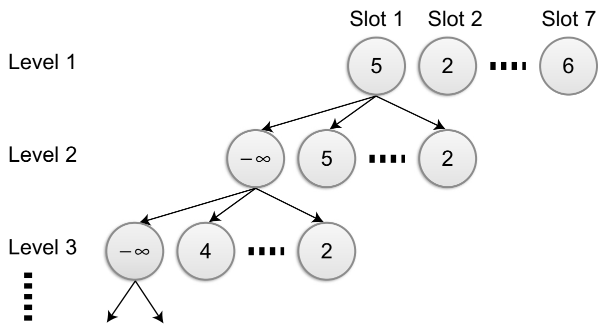


FIGURE 35.7

A minimax decision tree for four-in-a-row. Each node is a task.

35.4.1.1 *Design*

To help the computer pick the optimal move, we look ahead *n* moves and use a minimax algorithm to pick the move that gives the best worst case scenario. In Figure 35.7 we see the decision tree used in the algorithm. The nodes at the first level represent the possible moves that the computer can make. The children of these nodes represent the moves that the human opponent can take in the next turn, given the computers move. The children of these nodes in turn represent the move that the computer can make and so on, until we have looked *n* moves ahead.

When a leaf node is reached, either due to one of the players winning or because we have looked *n* moves ahead, a heuristic function is used to give each leaf node a value depending on how good that outcome is. The computer winning is infinitely positive and the player winning is infinitely negative. The other scenarios are valued by the difference in how many two or three token sequences each of the players have. The nodes at even levels, which represent the human player, take the value of the child node with the *lowest* value, as this represents the player's optimal move. On the other hand, the nodes at odd levels, which represent the computer player, take the value of the child with the *highest* value. In the end, the node on the first level with the highest value represents the best next move for the computer opponent.

It is hard to predict how much time will be spent in each branch. By making each node in the minimax decision tree a task, we can use dynamic load balancing to achieve an even load. We set each task to hold information on what level the node is on, its parent node, its value and the moves taken by its ancestor nodes. We save memory by only storing the moves taken and not the entire board state, as the new board state can be generated quickly from the current board state and the moves taken. To know when the problem has been solved, we keep a counter at each node that keeps track of the number of child nodes it has received a value from. When the root nodes have received values from all of their children, the work is complete.

35.4.1.2 *Evaluation*

We evaluated the performance of the computer player by playing the game scenario shown in Figure 35.6 with different number of lookahead moves.

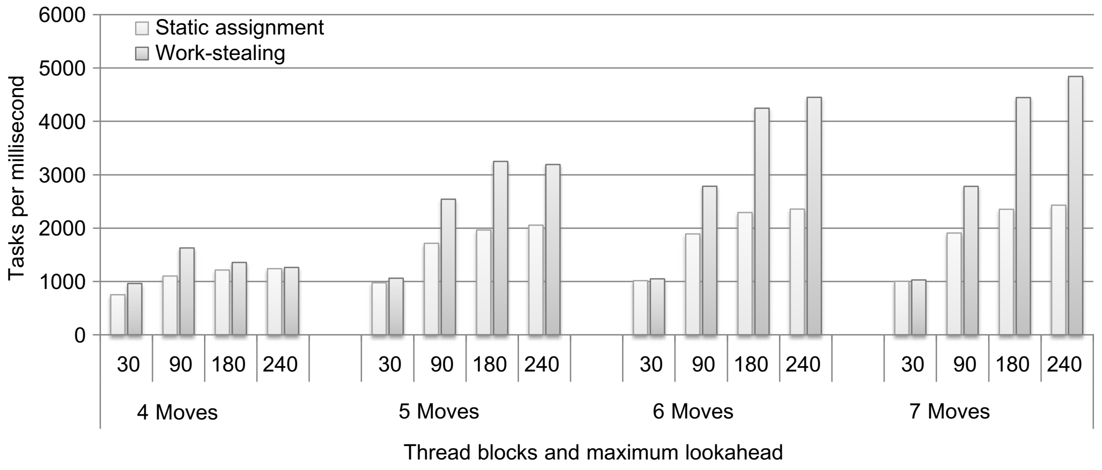


FIGURE 35.8

Four-in-a-row: Tasks performed per millisecond for different number of lookahead moves and thread blocks.

The graph in Figure 35.8 shows the performance for different numbers of lookahead moves (4 to 7) as well as different numbers of thread blocks (30 to 240). The maximum number of thread blocks that can run concurrently is 240, as each multiprocessor can support up to 8 thread blocks (given enough register and shared memory resources) and there are 30 multiprocessors available on the graphics processor we used. We use 64 threads per block, as the work in each task is not sufficient to take advantage of more than this, due to the small size of the game board. More threads would just add overhead.

We can see that using only 30 thread blocks gives poor performance for all four cases. With only four lookahead moves we have few tasks to distribute between thread blocks, something which hurts performance for the work-stealing when using too many thread blocks. When the number of tasks increases, we benefit from having many thread blocks and the best result for lookahead higher than four is when have 240 thread blocks.

Figure 35.9 shows the number of tasks per millisecond performed by the two different load balancing schemes using 240 thread blocks. We see that the dynamic load balancing scales much better, when faced with a higher load, than the static load balancing. At 7 lookahead moves it is twice as fast as the static scheme.

The figure also shows the number of tasks that we have to allocate space for. Note that the scale is logarithmic. For the static load balancing, this is the maximum number of tasks that can be created in a kernel invocation. The dynamic load balancing has a deque for every thread block, so here we need to multiply the maximum number of elements used in a deque with the number of thread blocks. The graph shows the memory requirement for 240 thread blocks. For 7 lookahead moves, the static load balancing requires around $\approx$800,000 tasks to be stored, while the dynamic only requires around 50 times the number of thread blocks, $\approx$12,000 tasks.

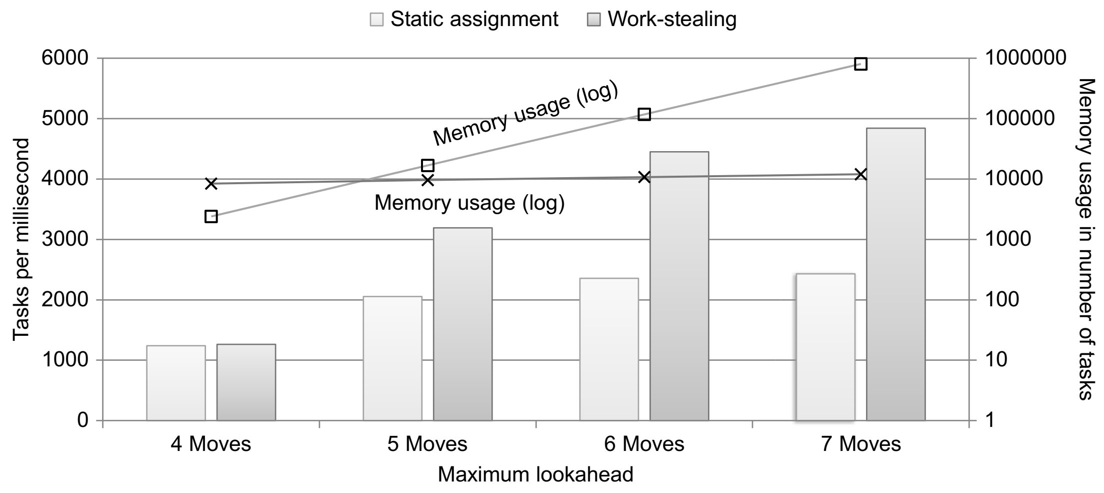


FIGURE 35.9

Four-in-a-row: Tasks performed per millisecond and memory usage in number of tasks for 240 thread blocks.

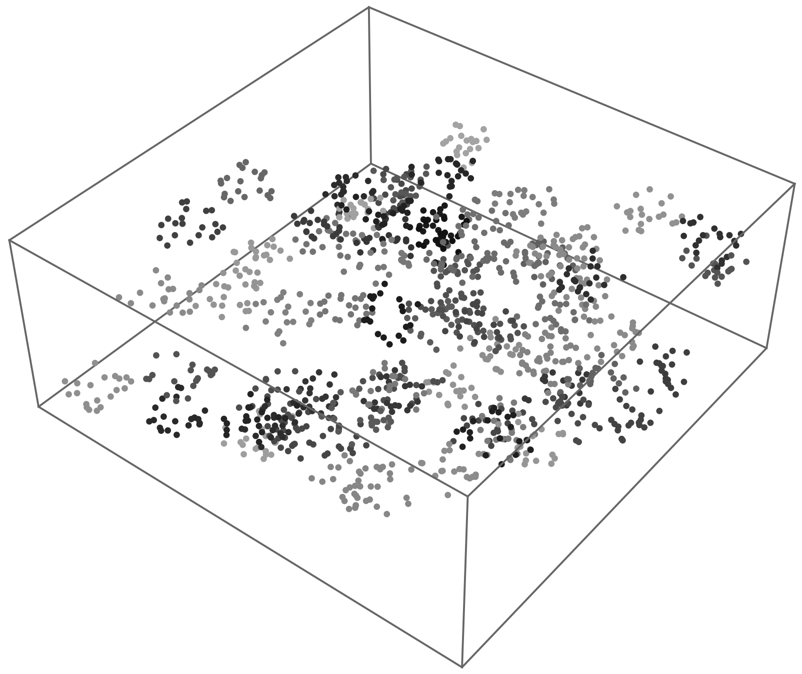


FIGURE 35.10

A set of 3-D particles to be placed inside an octree.

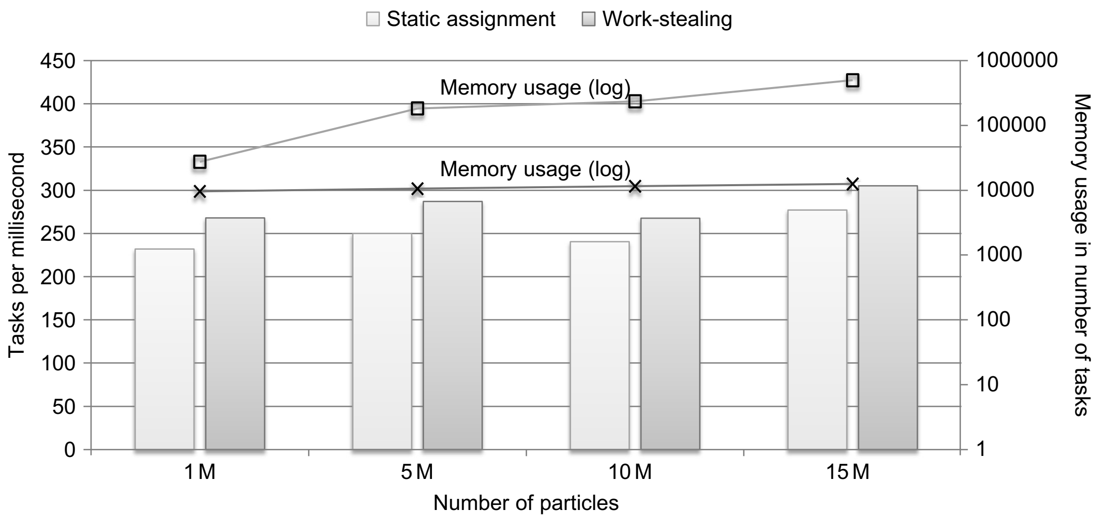


FIGURE 35.11

Octree partitioning: Tasks performed per millisecond and memory usage in number of tasks for 240 thread blocks.

35.4.2 Octree Partitioning

Our second application is an implementation of an octree partitioner for 3-D particle sets. The program takes a set of particles in 3-D space, such as the ones in Figure 35.10, and then recursively divides them in all three dimensions, creating eight octants. This is done until there is only a small number of particles left in the octant. The result is a hierarchy that makes it easy to find which particles occupy a given space.

In our implementation, each task consists of an octant and a list of particles. The initial task encompasses the entire space and all particles. The octant is then divided into eight parts and the parts that have more than a specified number of particles in them are turned into new tasks. A counter keeps track of the number of particles that have found a final place in an octant. When this number reaches the total number of particles, the work is completed.

To evaluate the octree partitioning we varied the number of particles to partition from 1 to 15 million. In Figure 35.11 we see that the dynamic load balancing is slightly faster than the static, between 10% and 15%. The memory consumption remains relatively static for the dynamic load balancing, while it increases quickly for the static assignment.

35.5 FUTURE DIRECTIONS

The two methods presented in this chapter both use pre-allocated, fixed size, arrays. It would be interesting to investigate if there is a lock-free way to dynamically resize these arrays, so that the maximum memory consumption does not need to be known in advance.

Another important area to look into is task dependencies. The two applications we have presented in this chapter both spawn tasks that can be performed immediately, but in other scenarios there might be cases where a task cannot be performed until some other tasks have been finished. Finding a lock-free method to deal with task dependencies efficiently on graphics processors would be useful.

Acknowledgments

This work was partially supported by the EU as part of FP7 Project PEPPHER (www.peppher.eu) under grant 248481 and the Swedish Research Council under grant number 37252706.

References

[1] P. Hoai Ha, P. Tsigas, O.J. Anshus, The synchronization power of coalesced memory accesses, IEEE Trans. Parallel Distrib. Syst. 21 (2010) 939–953.
[2] R.D. Blumofe, C.E. Leiserson, Scheduling multithreaded computations by work stealing, in: Proceedings of the 35th Annual Symposium on Foundations of Computer Science, IEEE Computer Society, Santa Fe, NM, 1994, pp. 356–368.
[3] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, Y. Zhou, Cilk: an efficient multi-threaded runtime system, in: R.L. Wexelblat (Ed.), Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), ACM, Santa Barbara, CA, 1995, pp. 207–216.
[4] N.S. Arora, R.D. Blumofe, C. Greg Plaxton, Thread scheduling for multiprogrammed multiprocessors, in: Proceedings of the ACM Symposium on Parallel Algorithms and Architectures, ACM, Puerto Vallarta, Mexico, 1998, pp. 119–129.
[5] D. Cederman, P. Tsigas, On dynamic load balancing on graphics processors, in: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics Hardware, Eurographics Association, Sarajevo, Bosnia, 2008, pp. 57–64.
[6] M. Herlihy, N. Shavit, The Art of Multiprocessor Programming, Morgan Kaufmann, Boston, 2008.
[7] D. Cederman, A. Gidenstam, P. Ha, H. Sundell, M. Papatriantafilou, P. Tsigas, Lock-free concurrent data structures, in: S. Pllana, F. Xhafa (Eds.), Programming Multi-Core and Many-Core Computing Systems, Wiley-Blackwell, 2011.