

Multithreading plug-ins for Maya 2009

Martin Watt, Autodesk

This document is specifically aimed at Maya 2009, as it makes reference to some API functionality added in that release.

I. INTRODUCTION

The advent of multicore systems combined with the lower rate of increase in single processor performance means that it has become necessary for developers to turn to threading to continue to increase performance at historical rates. This is particularly important for processor intensive applications such as Maya and associated plug-ins.

This document gives an introduction to multithreading in the context of writing plug-ins. It assumes no prior knowledge of threading, so presents an overview of this topic as a prelude to a discussion of threading in Maya.

II. DATA AND TASK PARALLELISM

At a high level there are two ways of decomposing work into multiple threads. Data parallelism involves performing the same operation on multiple data elements in parallel. A typical example would be evaluating a loop, where each thread works on a part of the loop range. The benefit of data parallelism is that it can usually scale to high core counts if there is sufficient work in the loop. The downside is that it is limited to certain types of workload.

Task parallelism involves different tasks running in parallel, for example a compute thread, a UI thread and a graphics thread. The benefit is that different parts of the system can be worked in parallel, e.g. CPU and GPU. The downside is limited scalability, as the number of tasks is usually rather small and unlikely to be equally expensive.

A well threaded application would include both data and task parallelism, possibly at the same time. In addition it would be designed to be scalable, so increasing core counts in future processors would be automatically utilized by the application.

III. THREADS

This section describes the lowest level threading concepts. An understanding of these concepts is essential for any developer working with threaded code.

2.1 What is a thread?

A thread is a new execution space, created and managed by the operating system, which contains the following resources that are private to the thread:

- A new stack
- A copy of the processor registers
- An instruction pointer to the first instruction to be executed.

Threads do not duplicate the following resources, which remain process global and visible to all threads:

- Code
- Data created prior to thread launch

2.2 OS-specific threading libraries

These are the lowest level threading implementations available to a user on an operating system. Maya 2009 uses the following threading libraries:

Windows: Win32 threads

Linux: pthreads

OSX: pthreads

2.3 Thread management

Thread creation is a relatively expensive operation, so it is often useful to create threads at the beginning of an application and keep them alive but sleeping in a **thread pool**. Threads can be activated, run and put back to sleep in the pool at a much lower overhead than continually creating new threads when required.

2.4 Thread properties

Individual threads have properties that can be controlled by the user. For example, thread **priority** determines how a thread is scheduled by the OS. A thread with higher priority will be given preference by the OS scheduler.

Thread **affinity** defines what physical processor the thread runs on. Sometimes it might be useful to bind a thread to a core so behavior is more deterministic. However usually OS providers recommend allowing the OS to manage processor affinity rather than have the user do so.

IV. LOCKS

Locks are operators that define a region of code such that only a single thread can traverse that code at any given time. Locks are also known as mutexes (mutual exclusions), although that term can have slightly different meanings on different OSs. A typical example is to protect multiple simultaneous write operations to a global variable from different threads, for example if this code is called from multiple threads:

```
static int val=0;
LOCK;
val++;
UNLOCK;
```

and the lock were not in place, two threads could attempt to modify `val` at the same time, both threads reading, updating and writing in separate stages, and so producing incorrect results. This is known as a race condition, and is discussed in more detail later.

Locks may be recursive, so the same thread can acquire the same lock multiple times. However this is generally considered to be a dangerous coding practice, as failure to unlock an equal number of times can lead to hangs as the lock is never released.

Managing locks correctly while avoiding race conditions is one of the most difficult parts of threading.

Atomic operations

Modern processors are able to perform certain operations on variables atomically. This means that the operation appears to be completed in a single step when viewed from other threads of execution. As a result, such operations cannot cause race conditions and do not require locking, as there is no possibility that another thread will be able to read the value during the modification operation. On some operating systems, low level functions are provided that directly invoke these atomic operations in hardware. These operations are significantly faster than applying user locks, and should always be used when possible.

Semaphores and signals

A semaphore is a generalization of a mutex lock. Rather than the simple boolean state of a mutex lock, a semaphore contains an internal counter. This counter is decremented each time a thread asks the semaphore for access to the locked resource, and incremented each time a thread releases the resource. If the counter reaches zero, additional threads attempting to acquire the resource will block until one of the holding threads releases the resource. So a

semaphore with an initial value of 5 will allow at most 5 threads to access the resource simultaneously.

OSX platform-specific pthreads issues

Although the OSX pthreads implementation complies with the POSIX standard, it does not include all the pthreads and associated functionality available on Linux. **Appendix I** provides more details, and is important for anyone working directly with pthreads on OSX.

Native threads usage

These basic threading primitives are useful if you are working at a low level on a single platform. They are complicated to use in cross-platform applications due to the different implementations. They also require a lot of work compared with some higher level threading implementations that will be described later. However for cases where ultimate control over thread behavior is required, including operations such as priority and affinity, native threads are the best option.

V. OPENMP

5.1 OpenMP overview

OpenMP is a higher level threading abstraction primarily designed for data parallelism applied to **for** loops. It is supported by VS2005 and by the Intel compiler on all platforms. The version of gcc required by Maya 2009 on OSX and Linux does not support OpenMP.

OpenMP is implemented via compiler pragmas. Compared with native threading, it presents a very simple interface to the user. All the thread creation and management and most data partitioning is hidden from the user. OpenMP uses thread pools, which are created the first time a parallel region is encountered (for this reason it is important to ignore the very first threaded loop traversal in an application when profiling OpenMP code.)

Here is a simple example of a threaded loop:

```
#pragma omp parallel for
for(int i=0; i<imax; i++) {
    doWork(i);
}
```

This code will be converted by the compiler into a function that is run in parallel by multiple threads, by default one thread per logical processor. The first processor takes the first (imax/numThreads) elements of the loop, the second processor takes the next equally sized

chunk and so on. This ensures good **cache affinity**, i.e. the processor is usually working with adjacent data elements and so minimizes cache misses.

Here is a more complex example:

```
#pragma omp parallel for schedule(guided) if(imax>1000)
for(int i=0; i<imax; i++) {
    doThreadsafeWork(i);
    #pragma omp critical
        doNonThreadsafeWork(i);
}
```

This code also breaks the loop into chunks, but the guided scheduling option causes it to use chunks smaller than size (imax/numThreads), and to send a new chunk to each thread as it finished an existing chunk. This provides better **load balancing** for cases where workload varies between iterations of the loop, since at runtime additional chunks of work are assigned to any threads that finish early. The **critical** pragma places a lock on the following line.

The **if** conditional on the main pragma causes the loop to be run in parallel only if the trip count (number of loop iterations) exceeds the specified value. There is overhead to invoking a parallel region, so it does not make sense to parallelize the evaluation if the trip count is too low. A good rule of thumb is to assume an overhead of 10k clock cycles to start and end a parallel region, so the cutoff trip count should try to exceed this work. This is particularly important for an application like Maya, where the same algorithm may be invoked on a single very dense object or a large number of very simple objects. In the latter case, the extra startup overhead of thousands of very short threaded evaluations could easily overwhelm any threading benefit, and you might actually get significant slowdowns with threaded code that does not have a cutoff. Note that even if the threading is not cut off by the conditional, the code inside the loop is still extracted by the compiler into a separate function and will take the hit of a function call.

5.2 Pros and cons of OpenMP

The benefits of OpenMP are cross-platform support, simple implementation, and ease of removal. Simply disabling the pragma causes the code to revert to its serial form, and can be used as a quick way to check that the behavior is the same. A surprisingly large amount of code can be threaded using very simple OpenMP pragmas such as those in the examples above. It is very useful for quickly prototyping and evaluating possible threading benefits, even if the final implementation will be done using a different threading library.

The downsides are limited algorithm applicability, and incompatibility between implementations. The VS2005 and Intel OpenMP libraries can be used together, but they do

not recognize each other's implementations. So a threaded loop compiled with VC2005 that calls a function compiled with the Intel compiler will ignore any OpenMP locks defined by the Intel compiler.

OpenMP is also problematical with nested threading, where higher level threads are spawned that call into code that is itself threaded at a lower level. This causes more threads to be activated than there are cores on the system. This is known as **oversubscription**, and leads to poor performance. Oversubscription becomes increasingly problematical as more threading is added to an application, since a developer may not even realize that a function being called in parallel is itself threaded.

5.3 OpenMP and Maya

A lot of algorithms in Maya are threaded using OpenMP, including the fluids solver, hair collisions and many deformers. Maya 2009 uses the Intel compiler for OpenMP on all platforms.

5.4 Vendor-specific OpenMP Issues

There are some performance and correctness issues with some OpenMP implementations. See **Appendix II** for details.

VI. INTEL THREADING BUILDING BLOCKS (TBB)

6.1 Overview of TBB

TBB is a relatively new higher level API for threading, supplied by Intel. It replaces the concept of threads with the concept of tasks, which are defined by the user and queued up. TBB then maps those tasks to threads efficiently.

TBB is available on all platforms Maya 2009 supports. It provides wrappers for common functionality such as for, while, reduce and sort operations. It also provides atomic primitives, a scalable memory allocator (allocators such as malloc access the global heap and so are locking operations, leading to contention when many threads are working with memory.) Finally there are threadsafe containers based on STL, such as lists, queues and hash maps. Note that these threadsafe containers have poorer single threaded performance than regular containers, since they usually require internal locking, so they should be used only when required for threadsafety.

6.2 Pros and cons of TBB

TBB is a higher level library than OpenMP and provides increased flexibility at the cost of greater complexity. Parallel code must be extracted into separate classes, which is more invasive than OpenMP.

The task-based approach avoids the problem of oversubscription encountered with OpenMP. Maya 2009 makes use of TBB in several places, partly to avoid the oversubscription issues of OpenMP, and partly to make use of the higher level algorithms and containers. Writing threadsafe cross-platform primitives and containers is remarkably difficult, and using a third party library designed for the purpose allows developers to focus on optimizing their specific application rather than building the required threading infrastructure.

Although TBB is used by Maya, it is not exposed in a way that plug-in writers can access directly, so a license must be acquired for a developer wishing to use TBB in their own plug-ins. There is no requirement to use TBB when threading Maya plug-ins.

Note that version compatibility is important for TBB. For Maya 2009 it is necessary to use TBB version 2.1.012.

VII. THREADING CHALLENGES

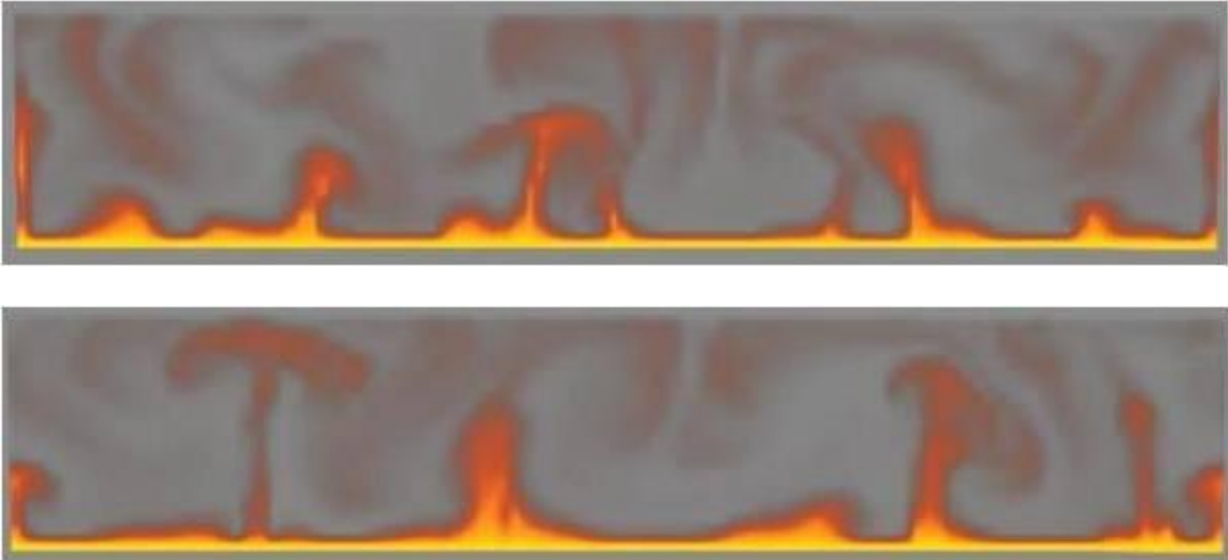
7.1 Correctness

One of the most common threading problems is the **race condition**, where two threads attempt to modify the same data. Modifying a data element is not usually an atomic operation, meaning it is possible for both threads to read the value, then update it, then write it. In such cases one of the edits will be overwritten by the other. Race conditions can be extremely hard to detect as they may happen only very rarely, and be hard to reproduce. Tools such as Intel's **ThreadChecker** are good for catching race conditions, both ones that actually happen and those that could potentially happen.

Race conditions obviously lead to incorrect results. There are other more subtle problems that can occur in threaded code. An example is precision differences in floating point arithmetic. With reduction operations, where values are accumulated in different numbers of threads and then summed, the number of threads can affect the final result since the roundoffs are different:

```
#pragma omp parallel for reduction(+: sum)
    for ( int i = 0 ; i < n ; i++ ) sum += x[i]*y[i];
```

This can be a significant effect for simulations, where small differences can accumulate over time. Below are two images from an initial threaded implementation of the Maya fluids solver that in one place called code similar to the above. The only difference between the runs is the number of cores on the system, and so the number of threads being used in the solver. Although neither result is “incorrect”, the difference in appearance due to differences in roundoff is quite dramatic:



There are a couple of ways to avoid this. One is to temporarily use higher precision, eg going from floats to doubles during the threaded computation. The idea is that the threading benefit is greater than the performance hit of higher precision computations. Another way is to break the data into a number of chunks that is fixed at compile time, and allocate those chunks to threads at runtime, then accumulate the sums in a consistent order. This ensures the size of a chunk never changes, so the final sum should always be identical.

Some C/C++ library functions are not threadsafe as they maintain internal state. Examples are STL containers (even reads of STL containers may not be threadsafe in some cases), C functions like `strtok()`, and many built-in random number generators. Care must be taken if any of these are called from threaded code. TBB provides threadsafe containers as an alternative to STL, and there are threadsafe implementations of many C functions which are identified with an `_r` suffix (for reentrant) on Linux and OSX, or an `_s` suffix on Windows, e.g. `strtok_r()` is a threadsafe version of `strtok()`, `strtok_s()` is an equivalent threadsafe function on Windows. Unfortunately there are no cross-platform standards for these threadsafe functions at the moment.

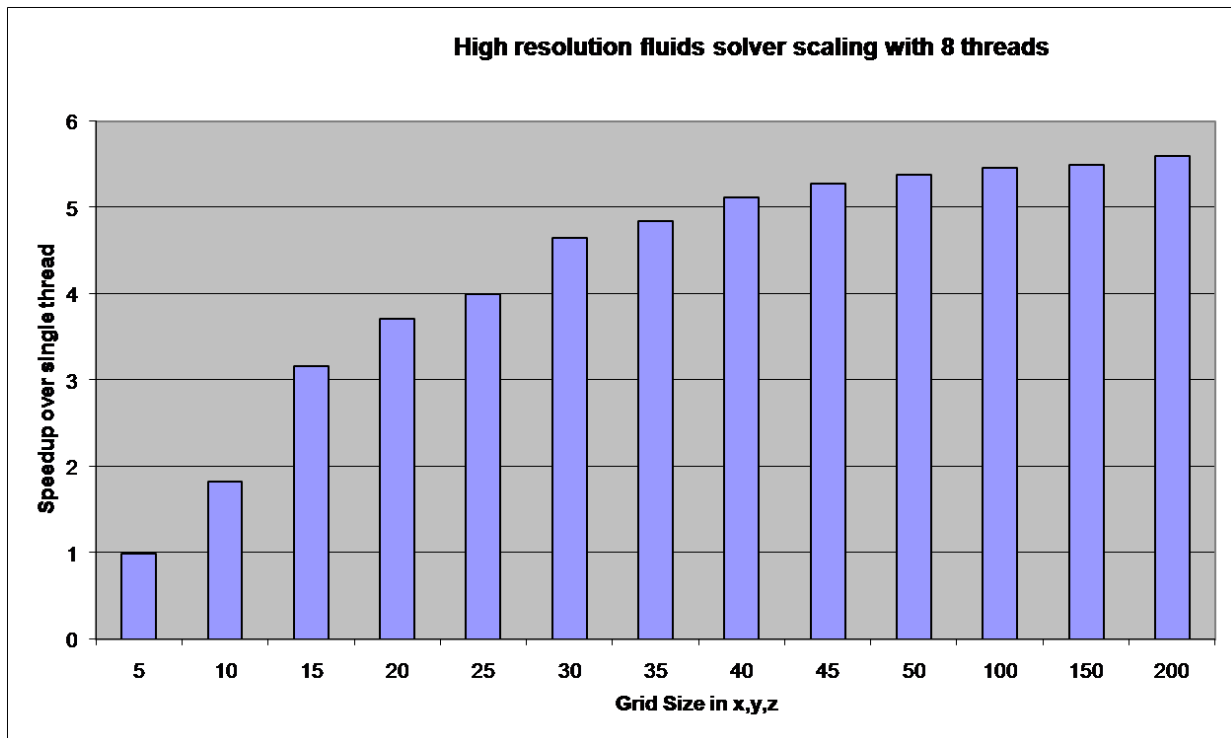
7.2 Scalability

7.2.1 Amdahl vs Gustafson

Amdahl's law is an observation that for a fixed problem size, performance scaling has diminishing returns as thread counts increase because the serial portion of the code eventually dominates, leading to a ceiling on scalability.

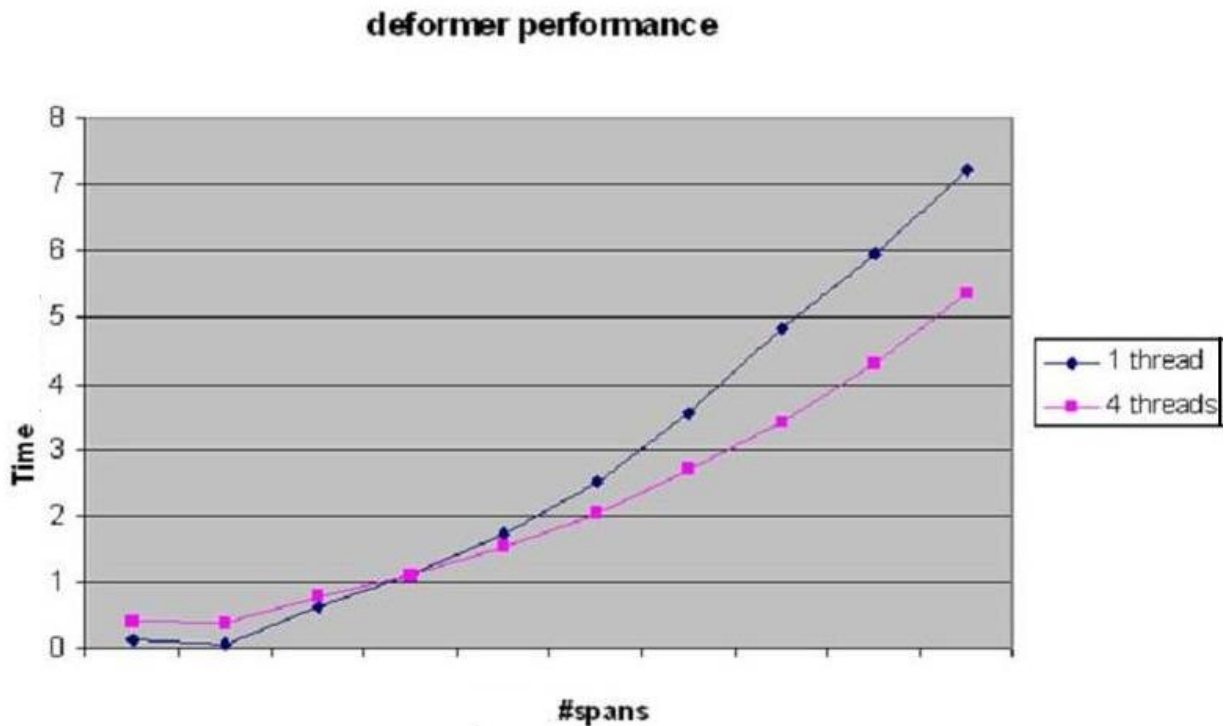
Gustafson's observation is that usually increasing thread counts are used to address larger problems, so rather than a fixed problem size it is better to think in terms of a fixed runtime. In such cases, scaling can continue to increase indefinitely provided that the fraction of work in parallel versus serial code increases with the problem size.

For applications like Maya, we are closer to the second case than the first, so there is hope for increasing scalability over time for well written algorithms. Below is an example of the fluids solver showing increasing problem size leads to improved scaling.



7.2.2 Threading overhead

There is an overhead of around 10k cycles to wake up a thread pool. Thus any calculation that will take close to this number of cycles to evaluate is not worth threading. For cases where the trip count can vary wildly, and the loop is called many times, it may be worth explicitly providing a lower cutoff to avoid a performance hit. The following diagram shows performance of threaded and unthreaded code with small model sizes for one of Maya's deformers. At the smallest sizes threading overhead dominates, and single threaded code is actually faster than multithreaded code. Therefore threading was disabled below the crossover point on this graph.



Another scalability challenge is that locks get increasingly expensive with higher thread counts, so it is worth avoiding locks if at all possible, or at least being sure to re-profile an application when running on a machine with more cores than previously used. An algorithm that scales well to 4 cores may not scale well, or may even slow down, when going to 8 cores.

7.2.3 Load balancing

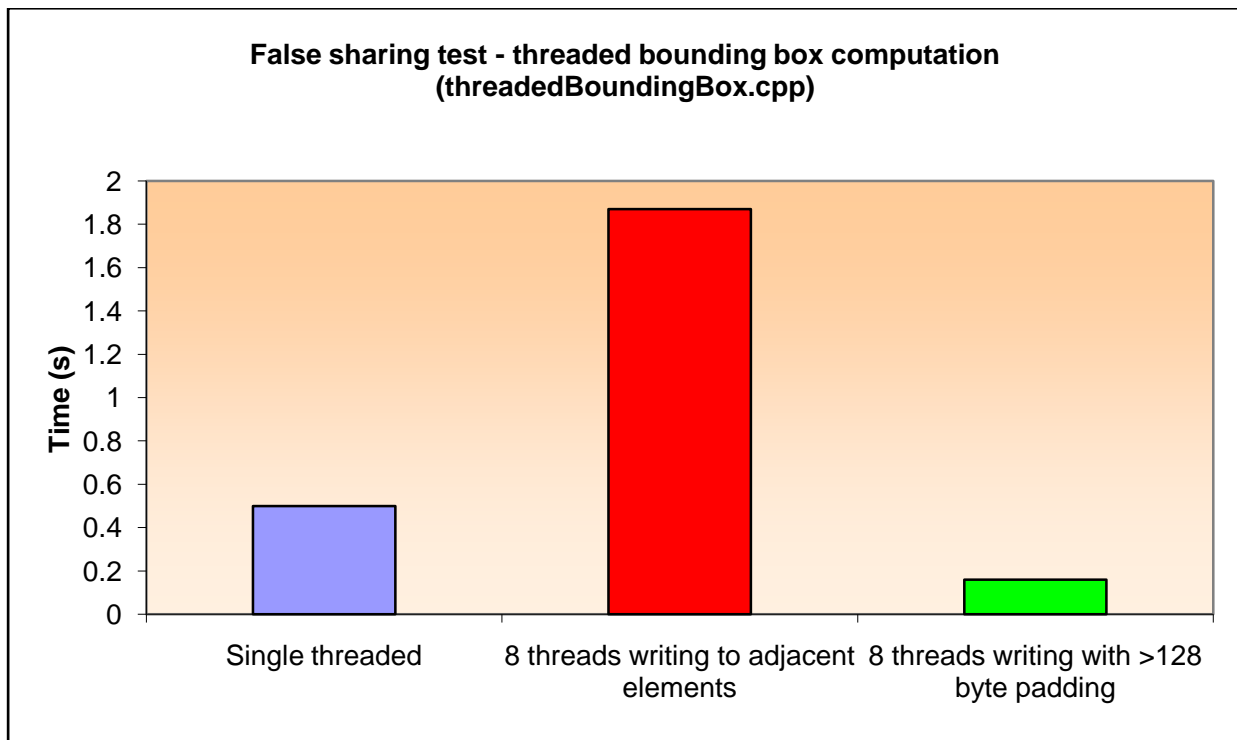
Some loops have very different work per iteration. For example a tool that operates on a subset of vertices of a mesh, or a fluids solver working on a volume that has some empty cells, will have some iterations that have little or no work to do. Simply splitting the number of elements equally among threads will be a poor choice for such cases, and it may be worth investigating algorithms and implementations that distribute work more uniformly, for example OpenMP's guided or dynamic scheduling, or TBB.

7.2.4 False sharing

Processor caches work by reading requested data into local caches. However it is not just the actual data that is loaded, but an entire cache line of data around the data item being read. A cache line is usually 64 bytes in size, but it is not guaranteed to be this value.

False sharing is a situation where multiple threads are reading from, and writing to, variables that are situated close enough in memory that they would reside on the same cache line when read into the cache. When one thread writes to one such variable, the entire cache line is read into the processor cache, and when a different thread attempts to modify the other data element the data must be written from the first processor cache out to a lower level cache or to main memory and then loaded into the other processor cache. This can lead to a ping-pong situation, where a chunk of data is continually bouncing between processor cores, and can clearly cause a significant performance hit.

The plug-in **threadedBoundingBox**, supplied with Maya 2009, illustrates the cost of false sharing and shows how to avoid it. In this plug-in multiple threads are used to compute one element of a bounding box. Each thread computes a float value representing the minimum X value of the bounding box for a subset of the points. The computed value is written into an array. The computation is run in different ways, single threaded and then multithreaded varying the spacing of the array elements used to hold the output results:



In the first test case, each thread writes to an adjacent array element. On a machine with up to 8 cores running separate threads, all threads are likely to write to the same cache line, triggering severe false sharing, since each element is only 4 bytes in size. In such cases performance is actually significantly worse than the unthreaded implementation.

In the second test case padding is added between active array elements to ensure each value will always be on a separate cache line. In this case performance is much better. The padding is computed by first querying the cache line size for the current processor. In Maya 2009 there is a new API method called `MThreadUtils::getCacheLineSize()` that will return the correct value at runtime for the host processor. Using this method is the safest way to ensure the code is well behaved for cache line access and also remains well behaved in future with processors that may have larger cache line sizes.

7.2.4 Hyperthreading

Processors such as the Core i7 (Nehalem) have Hyperthreading, where a single physical core is treated as two logical cores and two threads of execution can be run on the same physical core. This can provide additional performance but will not provide the same benefit as two physical cores since many core resources must be shared by the threads. In some cases it may lead to poorer performance than a single thread.

By default threading APIs like OpenMP and TBB will create one thread per logical core, so will make use of hyperthreading. It is important to test the effect of hyperthreading on your code. If you find it is hurting performance and you choose to run fewer threads than there are logical processors, you may need to check the operating system scheduling behavior. Older operating systems like Windows XP do not necessarily schedule tasks to threads in the optimal way when hyperthreading is present, and sometimes two threads can be running on one core while another core is idle. In such cases it may be best to disable hyperthreading in the system BIOS.

VIII. MEMORY ALLOCATION

Heap memory

Standard operations such as malloc allocate from the global heap. As a result, these operations are serialized with locks. When many threads are accessing the heap, this can become a bottleneck, particularly on OSX where the locking overhead is extremely high. Thread-aware memory allocators work by maintaining multiple local heaps rather than working with the main global heap. This avoids serialization of memory operations, improving performance. Examples of threaded memory allocators are Hoard, SmartHeap and the memory allocator supplied with TBB.

Stack memory

When threads are created, each thread gets its own stack. There is a limit to these stack sizes and it is possible to exceed it, leading to runtime failures. It may be necessary to increase the stack size, but increasing too far may lead to resource constraints.

IX. THREADING AND MAYA

9.1 Maya API threading interface, description and examples

Having laid the groundwork by describing required threading functionality, this section discusses the functionality provided by the Maya 2009 API, and illustrates usage with some example plug-ins supplied with Maya 2009.

In Maya 2009 a number of threading classes exist to provide support for thread creation, management and locking. These threading classes build on Intel's TBB threads, which are used internally by Maya. The classes expose a native threading-like API to plug-in writers. Note that these classes do not require the plug-in writer to use TBB directly in their plug-ins, or even be familiar with TBB. Since these threads are actual internal Maya threads, they respect thread count settings applied in Maya, and avoid the problem of creating increasing numbers of active threads as plug-ins are themselves threaded. They also provide protection from oversubscription problems in some cases due to the way TBB implements threading.

The API provides the following three major areas of functionality:

- fork-join threading implementation model
- asynchronous task-based threading model
- threading synchronization interfaces

The architecture employs object reference counting to assist with memory management and to delay deallocation of objects until all users of the object are finished with it. A successful call to an API function returns an interface pointer, and the caller is responsible for calling **release()** on that interface when they are finished with it. Failure to do so could lead to memory leaks.

The return **MStatus** of calls should always be checked, as it should never be assumed that a call will always succeed.

9.2 Thread creation and management using the Maya API

MThreadPool

This class creates or reuses a thread pool. Since creation and deletion of threads is expensive, it is a good idea to make use of the thread pool where possible, and try to keep it around between invocations of the plug-in rather than recreate it each call. The thread pool is reference counted, so it is possible to create it in the **initialize()** method of the plug-in and keep it for the duration of the application, releasing it in the **uninitialize()** method. It provides methods common to native threading implementations, allowing for migration of plug-ins from native threads over to this API.

The implementation requires the creation of a fork-join context which takes a function pointer as an argument. This function needs to implement the decomposition of the given problem into smaller chunks (tasks) which are then mapped to threads by TBB internally.

Note that the use of TBB internally means that it is possible to nest threaded fork-join regions created with MThreadPool without causing oversubscription, as TBB will schedule all the tasks in parallel by mapping them to cores.

Example

See the example plug-in **threadTestCmd**, supplied with Maya 2009, which computes prime numbers using a thread pool. This requires the creation of a new function, **DecomposePrimes**, to decompose the problem into smaller parts. The code within **DecomposePrimes** looks very similar to a native threading implementation, the main difference being the use of the control variable **NUM_TASKS** rather than the number of threads. As discussed above, TBB internally takes care of mapping the tasks to threads, and ensures optimal load balancing.

MThreadAsync.

This class provides methods common to native threads that allow the user to map independent asynchronous tasks to threads. The current implementation does not use a thread pool for this interface but instead creates new threads for each asynchronous task. This means it is possible to cause oversubscription, so care must be taken in managing the number of asynchronous threads and the amount of work they do concurrently.

The **createTask()** method takes a function pointer that executes the asynchronous task and also a pointer to a callback function that can be used by developers to implement a fork-join or any other signaling mechanism.

Example

See example plug-in `threadTestWithLocksCmd.cpp`. Compared with the thread pool example described previously, this implementation does not require the creation of a separate function like `DecomposePrimes`, as it mimics the native threading API closely. However the asynchronous threading interface does not provide a join method. The example shows how the equivalent functionality may be implemented by implementing barriers in the callback functions (`WaitForAsyncThreads`.)

This plug-in shipped with Maya 2009. Note that the function `Maya_InterlockedCompare()` could be implemented more efficiently using the atomic `compareAndSwap()` method in the `MAtomic.h` header provided with the Maya 2009 API.

9.3 Locking operations using the Maya API

9.3.1 System locks

`MMutexLock` and `MSpinLock` are system locks. `MMutexLock` uses `pthread_mutex_lock` on OSX and Linux, and `EnterCriticalSection` on Windows. The difference between these is that a mutex lock is a heavier operation but requires no CPU resources once the lock is held. A spin lock is a light operation, but requires heavy CPU resources while waiting. So if the wait is likely to be short, use a spin lock. These classes release the lock in their destructor, meaning explicit release is not required, and the lock will be safely released even if an exception is thrown in the locked code.

It is best to use different instances of lock objects for unrelated code. If a single lock object is used in many different places, threads may be blocked even if working on unrelated tasks.

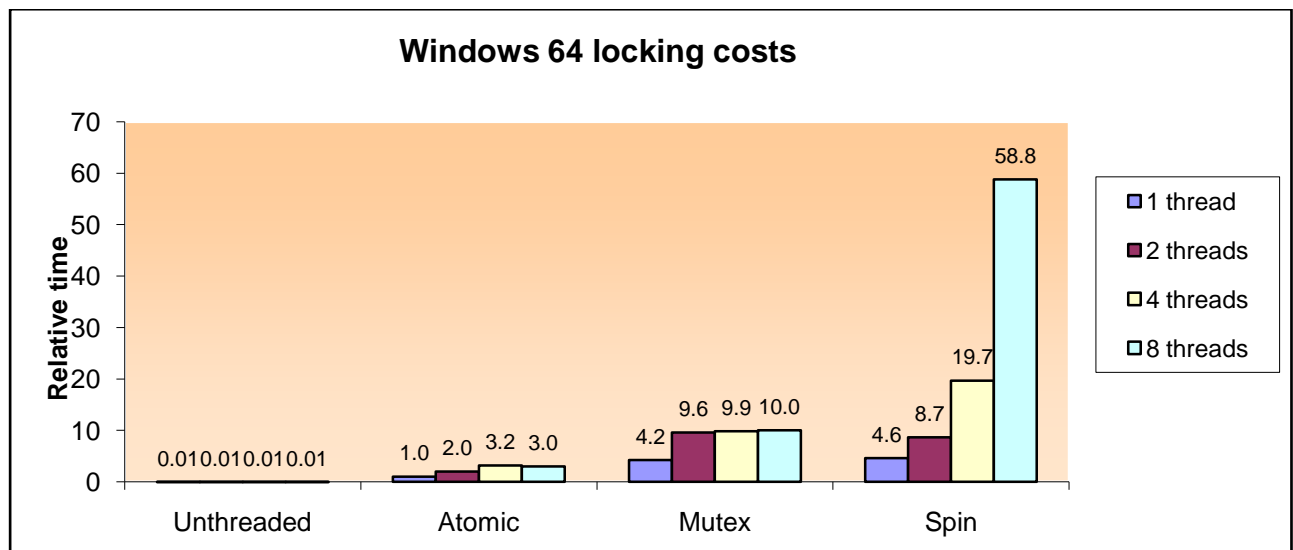
9.3.2 Atomic operations

The Maya 2009 API includes an implementation of atomic operations in the API header called `MAtomic.h` which provides cross-platform atomic operation functionality. Below is a listing of the atomic operations available. Full descriptions are given in the file itself.

```
MAtomic::preIncrement()  
MAtomic::postIncrement()  
MAtomic::increment()  
MAtomic::preDecrement()  
MAtomic::postDecrement()  
MAtomic::decrement()  
MAtomic::set()  
MAtomic::compareAndSwap()
```

9.3.3 Performance of locking and atomic operations

Below are performance numbers for the cost of spin and mutex locks and atomic operations on Windows 64. (Other platforms are similar with the exception of OSX where mutex is much slower.) The plug-in to generate these numbers is called **threadingLockTests**, and is provided as a sample plug-in along with this documentation for you to test performance on your own system. It requires a compiler that supports OpenMP. Note that this plug-in is an extreme case as the code does very little actual parallel work within the locked region. However it does give an indication of the relative cost of these locks under heavy contention. (A contended lock is one where another thread attempts to gain the lock that is already held by one thread. Having more running threads increases the chances of lock contention dramatically. Performance suffers significantly if there is high contention.)



Note the benefit of atomic operations over the various mutex locks. Also note that code without any locking is far faster than even the atomic implementation. Clearly it is best to avoid locks wherever possible. If not, atomic operations should be used, and only if this is not possible should full mutex or spin locks be used.

Lock granularity is a large complex topic in itself. In general it is best to lock at a finer grained level rather than a coarser level, as it allows more opportunities for parallelism. However too much fine grained locking can eat up a lot of system time and add significant complexity to the code. The challenge is to find an optimal balance point between coarse and fine grained locking. Since coarse grained locking is easiest, it is best to start with such locking and then more to progressively finer grained locking until performance stops improving.

The worst locking problems are those that prevent the application from progressing at all:

- **Deadlocks** occur when two threads each take a different lock, then each attempts to acquire the lock held by the other thread. No further progress is possible and the application hangs.

- **Livelocks** are similar to deadlocks except that each thread continually remain active without progressing past the lock. An example would be code that can detect a deadlock and roll back from it, but then simply allows the same evaluation to recur, leading to an infinitely repeating cycle.

9.4 Maya API threading example - deformer

The plug-in **splatDeformer** supplied with Maya 2009 is an example of a threaded deformer implementation. This example uses OpenMP rather than the Maya native API, since the focus here is on the approach taken to threading a deformer rather than the threading implementation itself, and OpenMP requires the least amount of additional code to implement threading.

The deformer is somewhat similar to a Maya sculpt deformer. A deforming mesh can be used to modify a selected mesh. The algorithm applies a deformation to every point on the mesh by snapping it to the closest point on the deforming object. The closest point operation is computationally intensive, so is a good candidate for threading, unlike some simpler deformers where threading overhead is likely to outweigh any potential benefit.

Here are some items of note in this implementation:

- Note that all DG access is performed outside the threaded region. This is because DG access is generally not threadsafe, although there are exceptions (see section "Threadsafe Maya API methods and classes" below.)
- All the elements are read into arrays using **MItGeometry** rather than using the iterator to do the loop traversal. This again ensures we avoid DG access in threads, and also avoids the problem that it is difficult to thread an iterator (although it can be done with Intel's OpenMP **taskq** extension.)
- There are fast methods on MItGeometry to read and write all points in a single command, **allPositions()** and **setAllPositions()**. These are much more efficient than querying and setting the points one by one. If you are working directly with points on an **MFnMesh** in Maya 2009 there are new methods **getRawPoints()** and **getRawNormals()** which are even faster as they return a const pointer directly to the internal data. Do not be tempted to cast away the const and modify the data in place via this pointer, as this will almost certainly lead to data corruption.
- The closest point methods on **MFnMesh** are not threadsafe, and are also relatively inefficient for multiple evaluations of a fixed mesh. The class **MMeshIntersector** uses a cached tree traversal so is much faster, and is also threadsafe. The intersector is

set up and initialized outside the loop, then evaluated inside the loop to get the full parallelism benefit.

- The **meshPoint** object must be allocated inside the loop so it is created on the current thread's stack. Allocating the object outside the loop would cause race conditions. Inadvertently writing to variables that were declared outside the loop is a common problem when trying to add threading to code that was not initially threaded, and great care must be taken to track all such variables down. It is good practice to declare variables in as local a scope as possible and avoid the practice of declaring variables at the top of a function anyway, and the fact that it makes any potential threading work easier in future is an additional benefit. It is also worth declaring any variables before the loop as const where possible, so write access to these variables from within the loop can be caught at compile time.
- The values are read from a shared array and written into a shared array so there is some risk of cache misses on reads and false sharing on writes. However the default OpenMP traversal scheme for a loop like this would be to have the first thread evaluate the first (nPoints/nThreads) elements and other threads evaluate equal sized contiguous chunks. This access pattern minimizes both cache misses and false sharing.
- If one of the closest point operations fails, the evaluator does not immediately abort, but instead sets a bad status flag and skips remaining iterations. In this specific case it is because OpenMP does not permit break calls from inside a threaded region. However it is often the easiest approach in general, compared with attempting to abort other threads that are in mid-evaluation.
- After the threaded region, the iterator is used to write the values back into the output object.

Scalability is limited by the serial portion of the code. For this reason it is important to optimize the code outside the threaded region. So for example it is best to get and set data in as large chunks as possible from the DG. This minimizes DG overhead. Several methods were added to **MFnMesh** to allow the user to retrieve all components of various types in a single operation, specifically for this purpose. The danger of course is increased memory usage, as these arrays must be maintained for the duration of the compute method compared with the iterator approach that updates one element at a time in place. The method `MFnMesh::getRawPoints` provides maximum speed and avoids memory overhead by returning a pointer directly to the internal data.

Some deformers cache their weights in the datablock. Reading from the datablock is slow and potentially not threadsafe. Often deformer weights do not change from frame to frame anyway, so in such cases it is best to read the weights once and store them in local arrays

within the deformer class, then do the threaded deformer evaluation using those cached weight values.

9.5 Initializing singleton objects in Maya plug-ins

A common coding pattern is creation of a singleton object, perhaps initializing global data for a solver. Usually this is a very simple operation:

```
static solver* singleton = 0
if (singleton == 0) {
    singleton = new Solver();
}
```

However when it comes to threaded code it becomes a very challenging problem, and entire papers have been written on the best approach to implementing this pattern, the problem being that optimizers often reorder code to break apparently threadsafe implementations. An obvious solution is to use a lock, but that is a heavyweight operation that is only really required on first construction.

In general, if the overhead of the initialization is not too large, it is best to put it into the **initialize()** method of the plug-in, which can be guaranteed to run serially.

If the object is large or expensive to construct, or the plug-in is not always executed, it is preferable to use lazy evaluation and initialize the object only when required. If this is the case, here is an implementation that may be helpful. It makes use of the **MAtomic** API class supplied with Maya 2009:

```
static Solver* singleton = 0
if (singleton == 0) {
    Solver* solver = new Solver();
    if(!TmtAtomic::compareAndSwap(&singleton, 0, solver)) {
        delete solver;
    }
}
```

On invocation, multiple threads enter the **if** condition and potentially create multiple solvers at the same time. The first thread to reach the **compareAndSwap** sets the **solverSingleton** pointer to the address of the new solver, and all subsequent threads delete their solver instances once they finish creating them. Although this seems disturbingly wasteful at first glance, it is actually very efficient, because the performance hit is taken just

once on startup, but all subsequent calls to the code just do a simple pointer compare, with no locking at all required.

If it is unsafe to have multiple threads in the Solver constructor, here is an alternative template to start from:

```
static Solver* singleton = 0
static int doneInit=0;
if(TmtAtomic::compareAndSwap(&doneInit, 0, 1)) {
    singleton = new Solver();
    doneInit = 2;
}
while(doneInit != 2) {}; // spin-wait
```

This code ensures the solver is only initialized once. To prevent other threads from accessing the object after the constructor has initialized the **solver** pointer but before it has finished its work, a spin-wait loop is applied that is not exited until the object has been fully initialized. The downside is that this approach requires an atomic operation and at least one traversal of the while loop for every subsequent call. There is also some concern about the compiler reordering the **doneInit=2** line to above the constructor, so an additional lock around the solver constructor call may be required to ensure this is avoided.

This is a good illustration of how even simple tasks can become tricky in a threaded environment.

9.6 Threadsafe Maya API methods and classes

The Maya API is very extensive, and it is not possible at this point to document every function and class to indicate whether it is threadsafe. This section focuses on some of the key classes that users are likely to want to call from threaded code.

Unfortunately it is not safe to make any assumptions about which methods may be threadsafe. Unless the function is inline and can be checked directly, there is always the risk that code may not be threadsafe. Even query methods may occasionally be unsafe as their classes may store internal state that is modified by the query.

Some classes rely on lazy evaluation so certain methods must be "primed" by calling them once outside a threaded region to update the internal data structure. For example, **MFnMesh::getVertexNormal** will first check to ensure the normals are up to date, and if they are not, it will recompute them. Thus two simultaneous calls to this function when the normals are not up to date will be unsafe. However once one call has been made and the

internal data structures are updated, subsequent calls are threadsafe as long as the object has not been modified since the initial priming call.

Threadsafe classes

- **MPoint/MFloatPoint/MVector/MFloatVector/MMatrix**
 - All read-only methods, no write methods.
- **M*Array** container classes
Array classes are safe for read access. For writes, the **set()** methods are safe as they do not resize the array. However **append()**, **insert()**, **remove()** are not safe as they potentially resize the array.
- **MFnMesh**
 - **closestPoint/intersection** methods are NOT threadsafe.
 - **get*** and **numNormals** methods are safe once they have been primed.
 - **getRawPoints/getRawNormals** are threadsafe.
- **MMeshIntersector** is threadsafe.
- **MDataHandle** - methods can not in general be assumed to be threadsafe.
 - **asXXX()** calls are safe since they just return pointers to the data referenced by the handle without worrying about whether they are clean or not (which is also the least common thing to do when reading data).
 - **asGenericXXX()** calls are not safe since they use the read/write reference counting to get at **Tdata** information.

The context is important too - if you just have a bunch of **MDataHandle** objects lying around then you have already done the thread-unsafe part of extracting them from the datablock. The first level of danger is that the datablocks may not exist at all yet since they are lazily created so you have to make sure only one thread does that. The second level is when you are getting handles for the purpose of reading data the datablock may trigger an evaluation (if you use the **inputValue(...)** methods) which has its own thread safety issues.

The safest algorithm for doing a compute would be:

1. Force creation of the datablock
2. PARALLEL(Get all input values from unique handles)
3. Perform the computation

4. PARALLEL(Put all output values into handles)
5. PARALLEL(Get any output values of interest, in any order, not necessarily unique)

- **MArrayDataHandle**

This class maintains internal state since it is effectively a smart pointer into the datablock, so there are obvious dangers in using the same handle in multiple threads. Multiple threads making calls in parallel to `jumpToElement()` followed by query calls will not be threadsafe.

Non-threadsafe functionality

- **MFnNurbsCurve** evaluators
- **MFnNurbsSurface** evaluators
- **MFnSubd** evaluators

X. OTHER TYPES OF PARALLELISM

10.1 Vectorization

Vector SSE instructions allow multiple identical operations to be done in parallel on adjacent data values. Four floating point operations or two double precision operations may be executed at the same time on processors supporting SSE2. Maya 2009 requires system support for SSE2, so the plug-in writer may assume SSE2 is always available.

SSE2 code can either be written directly in assembler or using intrinsics, or code can be written in C/C++ and the compiler can generate vector instructions. If the compiler generates vector SSE the process is known as autovectorization. The Intel and gcc compilers support autovectorization, but VC++ currently does not. Obviously it is much easier to write high level code and have the compiler generate vector instructions. However autovectorizers can be finicky and small changes can cause vectorization to disappear, so code written in C/C++ that relies on the autovectorizer needs to be flagged to ensure developers do not modify it and unwittingly disable the vectorization.

The supplied plug-in **sseDeformer** shows a simple example of SSE2 code that can be autovectorized by the Intel compiler. The example shows an approximate 3x speedup when run on a large polygonal mesh. Note that there can be significant overhead in getting the data into the correct format, which can sometimes negate any derived performance benefits. Also note that traditional threading of this code would be unlikely to be beneficial as the cost of the

threading overhead might outweigh the savings from threading. Vectorization can be a good alternative in such cases. In the ideal case both vectorization and threading would be applied to derive maximum possible speedups.

10.2 Autoparallelism

Some compilers offer flags that will cause them to attempt to parallelize code automatically. These are rarely useful, since once code gets to be a significant time consumer it becomes too difficult for the compiler to analyze statically to determine whether it is threadsafe.

XI. PYTHON THREADING AND MAYA

One important difference between Python threads and C++-based threads is that python threads do not run concurrently since the Python interpreter itself is not currently threadsafe. For this reason they are not useful for data parallel applications. However they may be useful for example where polling of resources is done that might otherwise have to wait for timeouts.

Python supports many of the threading primitives of C++-based threads, including thread pools, semaphores, locks, conditions and events. Threading functionality is described in the online python docs here:

<http://www.python.org/doc/lib/module-threading.html>

Python timer objects derive from the Thread class and can be used for periodic activities.

A supplied script, thread.py, shows some simple threading examples using Python. Python threads will just work when run from within Maya, there is no major extra work required.

XII. THIRD PARTY TOOLS

12.1 Threading Analysis tools

Intel ThreadChecker

This is a very useful tool that catches actual and potential race conditions in traversed code paths. It requires instrumentation and slows down runtimes significantly. It is available on both Windows and Linux.

Intel ThreadProfiler

This tool presents a view of thread usage during runtime, showing starts and ends of threaded regions, regions of oversubscription and undersubscription. It works with both native threads and OpenMP, on Windows and Linux.

Helgrind

A Linux open source threading analysis tool that checks for race conditions in code.

Intel compiler

Although this might not seem like a threading analysis tool, the Intel compiler does support two very useful compiler warning flags to detect writes or potential writes to static variables, which are a common threading problem:

```
warning #1711: assignment to statically allocated variable  
warning #1712: address taken of statically allocated variable
```

These warnings are enabled as `-ww1711` on OSX and Linux, and `/Qww1711` on Windows.

12.2 Profiling tools

Shark

Profiler on OSX. Shark is supplied with CHUD tools as a free optional install.

Zoom

Statistical call graph profiler that runs on Linux. <http://rotateright.com/>

Intel PTU

Prototype statistical call graph tool, somewhat similar to Shark, available from whatif.intel.com. Requires an existing VTune license. Runs on Windows and Linux.

VTune

Heavyweight profiler from Intel. Call-graphing adds significant overhead in instrumentation and runtime. Available on Windows and Linux.

XIII. API CLASSES AND PLUG-INS

API classes

MThreadPool, **MThreadAsync**, **MSpinLock**, **MMutexLock** – main threading classes, described in detail in this document. Added in Maya 2008.

MAtomic.h – cross-platform atomic operations. Added in Maya 2009.

MThreadUtils.h – functions to synchronize plug-in OpenMP thread counts with Maya. These functions must be used by any plug-in using OpenMP with a compiler other than the Intel compiler. Added in May 2009.

Example plug-ins shipped with Maya 2009

- **threadTestCmd** - usage example of **MThreadPool**. Shows how to initialize and manage the pool, create tasks that will be mapped to threads, and combine data from the threads to evaluate the final result.
- **threadTestWithLocksCmd** - usage example of **MThreadPool**, **MThreadAsync**, **MSpinLock**, **MMutexLock**. This plug-in evaluates pi using multiple threads. It builds on the previous **threadTestCmd** example by adding examples showing the use of the various lock types provided by the Maya API to update a global variable, and also has an example of parallel evaluation using asynchronous threads combined with a barrier construct that causes the threads to synchronize when they are finished. A thread pool automatically synchronizes. Note that the examples in this plug-in are for illustration purposes, and are not intended to demonstrate maximum efficiency in all cases. For example the locking methods are clearly inefficient, but do demonstrate usage of locks in real code.
- **splatDeformer** - example of threadsafe deformer plug-in. This is described in more detail in the body of this document. To use the plug-in, create two dense poly meshes nested one inside the other, select the inner then the outer, and invoke the deformer using the associated mel script. Adjusting the envelope attribute causes recomputes, which can be used to judge performance. Adjust the number of active threads to check scaling performance using the mel command **threadCount -n <numThreads>**. Requires OpenMP.
- **sseDeformer** - example of autovectorization using Intel compiler
- **threadedBoundingBox** – example of false sharing. Create a large (eg 1million poly) primitive and invoke the command. Requires OpenMP .

- **threadingLockTests** - performance test for **MSpinLock**, **MMutexLock**, and atomic operations. Requires OpenMP.

APPENDIX I OSX PLATFORM-SPECIFIC PTHREADS ISSUES

Although the OSX pthreads implementation complies with the POSIX standard, it does not include all the pthreads and associated functionality available on Linux. Functions that are not implemented are available as stubs that return error codes. This can be problematical when porting code that does not rigorously check error return codes, as the code will compile and run but may not function as expected. Here are some specific features of Linux pthreads that are not available on OSX as of the Tiger release:

- Unnamed semaphores are not supported.
- Cancellation points are only **pthread_test_cancel()** instead of POSIX mandatory cancellation points. So **pthread_cancel** will cancel a thread only when **pthread_test_cancel()** is called by that thread. (This is fixed in Leopard.)
- There is no static initializer for **pthread_rwlock_t** or recursive mutex.
- Mutex priorities: **pthread_mutex_setprioceiling** and **pthread_mutex_setprotocol** are not supported, meaning there is no priority boosting, inheritance or ceiling support for mutexes.
- **PTHREAD_SCOPE_PROCESS** is not supported.

APPENDIX II VENDOR-SPECIFIC OPENMP ISSUES

- By default Intel OpenMP threads stay active and spin for 200ms after they complete a task. The spin makes the threads fast to restart, but it consumes CPU cycles. 200ms is a very long time to spin for an application like Maya, since a single OpenMP loop evaluated as part of an animation playing at any more than 5fps could keep all CPUs busy spinning all the time. Such spinning would interfere with other threads in the application and also other applications on the system. For this reason we explicitly set the spin-wait time to zero in Maya, and only use nonzero values temporarily for specific closely packed loops where the spinning produces a noticeable performance benefit, being sure to reset it to zero afterwards.

If you are using Intel OpenMP in a plug-in, and want to set a nonzero spin-wait time you can include the Intel compiler version of the header file `omp.h` and then use the `kmp_set_blocktime()` function. However be very careful to set the spin wait time back to zero before exiting the plug-in and returning control to Maya :

```
kmp_set_blocktime(0);
```

- Microsoft OpenMP threads do not spin-wait, so do not have the above concern. However this can lead to lower performance where multiple threaded regions are run in close succession, as the thread pool must be woken up each time.
- Guided scheduling is unreliable in the OpenMP implementation of Visual Studio 2005, sometimes producing incorrect results. It appears to be fixed in Visual Studio 2008, but plug-ins for Maya 2009 require the use of Visual Studio 2005, so take great care to test results, or avoid using this scheduling model with this version of the compiler.