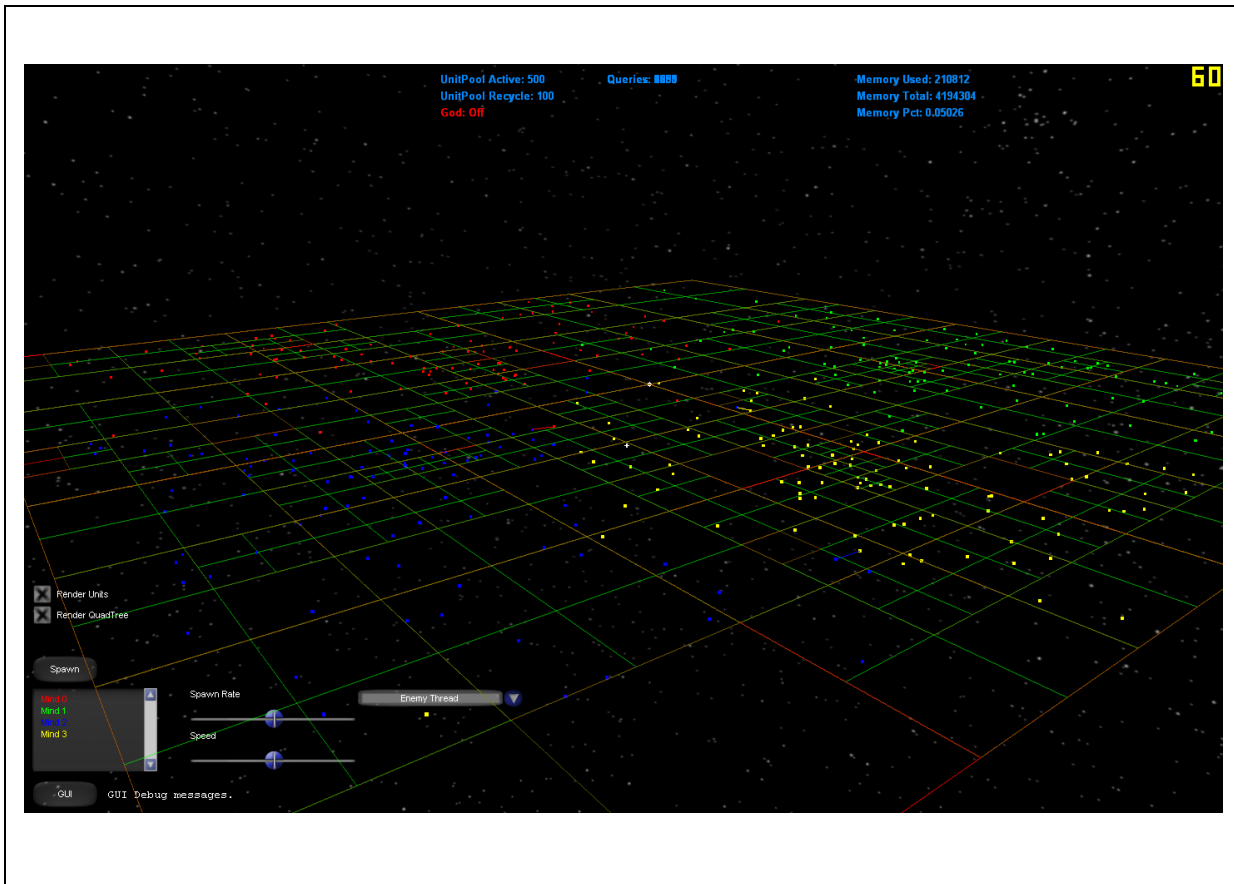


Thread-Safe Spatial Subdivision



A Project Presented to the Faculty of The Guildhall
at Southern Methodist University
By Erik Sunderland
(Bachelor of Computer Science, California State University, Chico, 2007)
In Partial Fulfillment of the Requirements for a Masters of Interactive
Technology in Digital Game Development with a Specialization in Software
Development

February 1, 2009

To the Graduate Faculty:

I am submitting herewith a project written by Erik Sunderland entitled "Thread-Safe Spatial Grid." I recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Interactive Technology in Digital Game Development, with Specialization in Level Design.

NAME, Supervisor

I have read this Project
and recommend its acceptance:

NAME, Advisor

Accepted for the Faculty:

Dr. Peter Raad, Executive Director
The Guildhall at SMU

ACKNOWLEDGEMENTS

I would like to thank my parents for always encouraging me to do my best and giving me the chance to accomplish it.

I would like to thank my professors for teaching me the skills to be the programmer that I want to be.

I would like to thank my girlfriend for taking such good care of me through such a rigorous part of my life that is The Guildhall.

Thread-Safe Spatial Grid

Supervisor: Gary Brubaker

Master of Interactive Technology degree conferred December 15th, 2008

Thesis / Project completed February 1st, 2009

The current generation of console hardware contains very powerful processor architectures. However, to utilize this to its maximum potential, the game development industry needs to be familiar with concurrency and threading concepts. This new paradigm of programming requires that engineers understand more about what the hardware is doing and how they can design systems that use multiple threads to process work simultaneously.

At the heart of many systems are containers that house the data used by the system. Linked-lists, quadtrees, octrees, and BSP trees are some of the containers that are looked at. All of these containers have unique ways to store spatial data and each have their benefits depending on whether the data is points or bounding boxes, static or dynamic, or even multi-dimensional going beyond two and three dimensions of games today.

The project was initially to discover a way to create a thread-safe quadtree that does not just restrict access to the entire tree with a mutex. After many attempts to create a very fine-grained tree, the problem was re-assessed and a spatial grid was created. The constant changing of data in the simulation caused unique problems for the dynamic quadtree approach and required that the data contention of this tree be removed to solve the problem at hand.

The final implementation held up fairly well against testing. Testing showed that as the number of cores increased, the overall time to update a given number of units dropped. This shows that the container and the simulation were utilizing the additional cores and is the kind of utilization the industry needs to do with current systems on today's hardware.

The project overall was very ambitious and taught me a lot about concurrency and design. Even though the quadtree attempt was not successful, I think that it taught me a great deal about atomic functions, design decisions in a concurrent system, and gave me an example of how to think about a solution when any thread can interrupt and change what another thread is about to do.

Contents

Contents	6
List of Figures	7
List of Tables	8
Nomenclature	9
Introduction	11
Field Review	13
Spatial Partitioning Techniques	13
Sphere tree	13
Divided k-d tree	14
Octree	15
Quadtree	16
AABB Tree	16
BSP Tree	17
Threading and Synchronization	20
Coarse-grain Synchronization	20
Fine-grain Synchronization	21
Synchronization Primitives	22
Lock-free Synchronization	23
Compare and Swap	25
Load-Linked Store-Conditional	25
ABA Problem	26
Methodology	28
The Product	30
Thread-safe QuadTree	31
Grid	37
Overview	37
Insert	40
Remove	40
Query	40
AI System	41
Rendering System	42
Profiling System	42
Testing and Data Collection	43
Results and Analysis	45
Profiling	45
Analysis	56
Critical Section vs Lock-free	56
Lock-free	56
Conclusion	59
References	60

List of Figures

Figure	Page
Figure 1: Gradient of Fine vs Coarse synchronization.	21
Figure 2: An implementation of Compare-And-Swap.....	25
Figure 3: QuadTree::Insert function pseudo-code	32
Figure 4: QuadTree::Partition function pseudo-code	33
Figure 5: QuadTree::Collapse function pseudo-code	34
Figure 6: QuadTree::Remove function pseudo-code.....	35
Figure 7: QuadTree::RadiusQuery function pseudo-code	36
Figure 8: Design of Spatial Grid.....	39
Figure 9: Table and graph of QuadTree contention data.	48
Figure 10: Update Times vs Number of Threads (Dual-Core).....	50
Figure 11: Update Times vs Number of Threads (Quad-Core)	50
Figure 12: 2000 Units Divided Among Threads.....	53
Figure 13: Graph of Coarse-Grain vs Lock-Free Implementation.....	55

List of Tables

Table	Page
Table 1: Difficulty of thread safety for dynamic data.....	19
Table 2: Processors and their supported atomic synchronization functions.	26
Table 3: Predicted vs Actual Gains.....	44

Nomenclature

Sphere tree	– a tree that stores objects based on their bounding sphere.
K-d tree	– a tree that stores point data alternating on the different dimensions that the data contains, alternating on those dimensions.
Octree	– a tree that recursively splits 3D space into eight smaller sections.
Quadtree	– a tree that recursively splits 2D space into four smaller sections.
AABB	– axially-aligned bounding-box
AABB tree	– a tree that uses an objects AABB to store in the tree.
BSP tree	– a tree that partitions data based on the other data in the tree, whether data is behind or in front of other planes in the tree.
Coarse-grain	– a synchronization technique in which access to a resource is exclusive
Fine-grain	– a synchronization technique that restricts access to the smallest portion of the resource that it can while maintaining safety.
Lock-free	– algorithms that do not use synchronization objects to prevent data corruption. Instead the algorithms are designed to ensure that at every step, data is safe.
TLS	– Thread-Local Storage. Storage that can be accessed similarly across multiple threads but each location is unique to one thread.
Mutex	– a synchronization object that provides exclusive access to a resource.
Event	– a synchronization object that can be used to signal another thread.
Semaphore	– a synchronization object that acts like a counter, blocking threads when its value is zero and decrementing every time a threads pass through it.
CAS	– Compare-and-swap. An atomic hardware instruction that replaces data in memory with a new value as long as the original has not changed.

LL/SC

– Load-linked, Store-Conditional. A pair of synchronization instructions that mark an address of memory as used when read, and if it has changed when store is called, will fail.

ABA

– A problem with compare-and-swap where memory can be read by one thread, be changed and changed back to the original value by another thread, and then looks unchanged to the original thread.

Introduction

The breakthrough of multi-core processors has unleashed a new era of programming. Threading and concurrency in applications is becoming an issue in current games. In the past, games rarely took advantage of threads. Currently, the Xbox 360 has three dual-core processors and the Playstation 3 has one main dual-core processor and an additional six usable SPUs. [Falcone, 2008]

Now that multiple cores are becoming more prominent in hardware, it is time for game programmers to embrace this technology. Objects in video games require a processing time to interact and understand their universe. As a result of this, a video game can incur a large amount of spatial queries for testing to understand their environment. As objects are created and destroyed in a game, they need to be added to (and removed from) some sort of database so that they can be culled for rendering, tested for broad phase collisions during physics simulations, or used in AI techniques that rely on distance relationships to make decisions. Another use of this type of data base is for performing a nearest neighbor search, given a location and a radius, the spatial database can return the objects that are within range.

The project is to develop a fast thread-safe spatial partitioning container. The container is designed to be accessible from many threads at one time. Threads can query the container, insert new objects, and delete objects that are no longer relevant to the system. The container is designed such that modifications to it do not completely stall other threads for long periods of time and still keep it in a state that is useable. Because many spatial containers are not easily implemented in a concurrent environment, each line of code requires attention to make sure that another thread cannot interrupt what is about to happen.

The world of the simulation is split up into a grid. Each cell of the grid contains an array of integers. The bits of these integers represent an index into a global array of units. If a bit is 1, then the unit is in that cell. The Windows API has `InterlockedAnd()` and `InterlockedOr()` functions which can be used to make atomic changes on a bitwise level. This allows bits to be turned on and off without having to lock the container. On Intel's x86 architectures, these functions provided, are executed by using memory barriers to serialize access to memory. All writes must occur before the instruction reads memory, acts on it, and stores it back, and then all reads must occur after. On other platforms, such as the PowerPC, the `Interlocked` functions may require additional techniques to achieve the same outcome. However, this is beyond the scope of this project but future work may take this issue into consideration and how to solve the problem in the new environment.

The project has an AI simulation that uses a thread-safe container while it is running. The different threads control separate armies and can insert, remove, and query the container. Having several threads operating on a single grid will create contention in the container. The units are colored by team; each team being controlled by an individual thread. The threads use basic AI techniques to control their armies and attack each other.

This is a good simulation because of the continuous modifications being made to the container. Every time a unit is spawned or killed a modification must be made to the container to keep future queries accurate enough to keep the simulation running. The threads that control the armies constantly query the database to see if their units are near enemies. The thread-safe grid allows the AI simulation to be broken up in parallel. In a linear implementation, the AI would have to iterate over each unit in the simulation and execute code to target enemies and move randomly in the scene. By breaking this up over an increasing number of threads, this should reduce the amount of time that this iteration takes. As the number of processors increase, the time that it takes to update the AI units should decrease.

Field Review

Spatial Partitioning Techniques

There are many approaches to spatial partitioning containers. The difficulty in updating objects against one another is trying to prevent a 1 to 1 comparison with every object in the scene. This brute force approach can destroy performance as the numbers of objects increase. Instead, various types of partitioning trees are used to quickly reject large sections of the scene at once.

Sphere tree

Sphere trees, described by John Ratcliff in *Game Programming Gems 2*, encompass their objects in a bounding sphere. Objects that do not have a natural spherical shape, such as bipedal player models, will have extra room inside their spheres. However, the sphere will surround them in all possible orientations of the model. By initially choosing a properly sized sphere, the size will not have to change through the life of the object.

Sphere trees are updated by re-inserting an object into the tree. When an object and its bounding sphere are no longer contained in its parent, it is removed from the parent and added to an insertion queue in the root node. The parent is also added to a queue to help balance the tree. During each frame, queues are processed to keep the tree in an optimal condition. [Ratcliff, 2001]

One problem with sphere trees is the constant re-insertions and re-balancing every time an object leaves the extent of its parent. In a threaded environment, a tree such as this could potentially perform poorly because the tree will have to be locked at some level to allow for the insertion to take place. However, on the other hand, having a thread dedicated to clearing these queues may be beneficial. The tree could be balanced while rendering or animation calculations are taking place.

Another difference that would make a sphere tree difficult to make thread-safe is that not all of the data is the same. In other containers, the data that is being stored is a point, where a sphere tree is an object that is surrounded by a bounding sphere. This would be difficult to make thread-safe because different sizes of data would require different modifications to the tree to balance it. With another thread performing insertions only, querying threads would have to wait until the queue was empty.

Divided k-d tree

A normal k-d tree splits data on k dimensions. The k in k-d tree stands for the number of dimensions that the data contains. For 3D points, k is 3. As a k-d tree moves down the levels from the root, the data is split on a different dimension. For three dimensions, the first split (the root node) could split on the x-axis, the second level split on the y-axis, and the third axis split on the z-axis. The pattern will continue with the fourth level splitting on the x-axis as the tree grows in size.

A divided k-d tree is similar except that it splits the top part of the tree on one axis for several levels, and the bottom half of the tree on another level. For a 3-dimensional case, the tree is split into thirds with each section of the tree spanning several levels and split on a single axis. A 2-dimensional divided k-d tree can be built out of a combination of binary trees. The top part of the tree separates data on one axis and the leaves of the top tree are smaller binary trees that are split on the second axis. [Overmars & van Kreveld, 1991]

K-d trees are very difficult to balance, but create very efficient data structures. This is because of the multi-dimensional sorting that is built into the structure. The techniques used in balancing an ordinary binary tree (one that is split on only one dimension of data), cannot be used. On top of this, kd-trees perform best when used with static data. Inserting and removing from a kd-tree can be done, but over time, the structure's performance gradually decreases. Eventually, it will be better to just rebuild the entire tree, rebalancing the data, and then perform a new search.

In a threaded environment, with many dynamic moving objects, this would not be the best approach. Instead, a container could potentially have two internal structures for maintaining objects. Static objects, are inserted into a k-d tree, while dynamic objects are inserted into a more flexible structure. A k-d tree would perform poorly in a threaded environment because instead of repartitioning nodes like other data structures, the entire tree must be rebuilt. If there is a lot of data that must be represented, then rebuilding the tree could take a long time. In a concurrent environment, this is bad because other threads will have to wait while the tree is rebuilt.

However, queries in a k-d tree are exceptionally fast. Once the structure is rebuilt, threads could quickly access the data that is in it.

Octree

Octrees are a widely used partitioning technique that encompasses an entire scene inside a bounding box and then recursively subdivide the scene into eight smaller pieces. Each node has 8 children that equally subdivide its space further. When an object is inserted into an octree, it is pushed as far down as possible while still being completely inside a node. When an object lies between two cells, it can either be linked to from both nodes, or be pushed up the tree until it is completely contained by a node. [Frisken & Perry, 2002]

Octrees can either be statically defined with a specific depth, length, and width, or they can be created as objects are inserted and removed from the tree. Creating an octree that only stores objects in the leaves would be beneficial in a concurrent algorithm because the data contention would be at one level of the tree. If objects were placed at any level of the tree, the contention would be too difficult to handle without locking the tree at the root. A dynamic tree that stores objects only in the leaves and partitions when a leaf becomes full would have the benefits of not wasting memory on unused nodes, while also maintaining a partitioned space when there are lots of objects in the same area.

Another derivation of an Octree is the loose octree. In this implementation the children of the tree do not line up exactly at a common boundary. Instead, their volumes can overlap. When this occurs, data stored in an octree that is in a common space for two volumes is inserted

into both. The loose octree can help when objects are moving around boundaries quickly. In a classic octree, the objects may bounce back and forth over a boundary between nodes. In a loose octree, when an object moves out of the range of one node, it does not necessarily need to be inserted into the neighbor. [Ginsburg, 2000]

Quadtree

A quadtree recursively partitions a 2-dimensional space into 4 smaller pieces. It is the 2d equivalent of the octree. The tree can be built to accommodate a particular depth, or a leaf width. In a quadtree, each node, except the leaves, contain four children. The children completely subdivide their parent's space and are usually split along the axis of the system. As objects are inserted into a quadtree, they are pushed down the tree until they can no longer be contained in one of the current node's children. Quadtrees can be empty in many nodes because objects are positioned arbitrarily in the partitioned space. [Gaede & Gunther, 1997]

A quadtree is very similar to an octree, however there are some key differences that would make it more thread friendly. First is the memory footprint. A quadtree would use less memory because there are only four children per node instead of eight. By using less memory, the entire tree can be stored in a smaller block of memory and reduce cache misses when traversing the tree. Second, because the quadree requires less nodes to construct its tree, it would be easier to make atomic changes to the tree. Instead of having to worry about eight nodes, there are only four.

AABB Tree

AABB trees use axis-aligned bounding-boxes to organize data in a scene. Leaves of the tree can contain a predefined number of objects, or they can be broken down even further containing only one. It is a similar concept to Sphere Trees in that both structures store a bounding volume for a collection of points.

When an object is inserted into an AABB tree, the object has a bounding box calculated for it. Then, the object is compared against the children AABB's of the root node. If the object can be completely contained in one of the children, it is inserted into that child and the process repeats. When the object can no longer fit inside a child node, the object is added to the node that it is currently at, and then the node is resized. As the recursive process unfolds, nodes higher up along the path back to the root, are also resized to accommodate the changes that were made to their children. Finally the root is resized and the insert is complete.

When an object is removed from an AABB tree, the object's parent is found and then the object is removed. As the tree unwinds, the node's AABB is shrunk when appropriate to hold the smallest bounding-box possible. The only time a node needs to be resized, is when the object was flush against one side of the box. This resizing of the tree maintains the partitioning of space.

When a dynamic object is updated, the object moves up toward the root node until its new position is completely surrounded by a node's AABB. The object's new position in the tree is then located, similar to an insert. If the object is moved all the way up to the root node, then the process of finding its new location in the database is essentially the insert process. Once the object's new location is found, the pointer is moved to the new node. The old and new parent nodes' AABBs are then adjusted accordingly. [Shagam & Pfeiffer, 2003]

AABB trees differ from quadtrees and octrees because each object must have an axially-aligned bounding-box that is inserted into the tree. In quadtrees and octrees, the object's position is inserted into the tree. Because the nodes in an AABB tree can be resized to accommodate children, and thus having to resize their parents, this tree would not be the best container for a concurrent algorithm.

BSP Tree

A BSP (binary space partitioning) tree is a binary tree designed for splitting data on the front and back faces of a plane. Each node of a BSP tree represents a convex space and stores a plane that divides the space into two parts. Creating a BSP tree requires three steps.

1. Select a plane to split on.

2. Divide the polygons of the scene with the plane.
3. Repeat process on the two created subdivisions.

The plane used to split the space can either be axis-aligned or a plane from the existing geometry. The partitioning algorithm can stop when a defined number of polygons are in the leaf, or continue until all of the polygons are in their own leaf. To partition the polygons in a scene, their vertices are compared to the front and back of the plane. When a polygon spans a partition plane, it must be divided into smaller polygons that are on opposite sides of the plane.

BSP trees are best used to partition static data. Splitting polygons of dynamic object for each frame can be costly and is generally not used. One way to contain dynamic data in a BSP tree is to represent dynamic objects as points. The dynamic points are added to the tree each frame and partitioned against the planes of the already defined tree. For a large number of dynamic objects this becomes slow because of all the insertions and deletions that must take place each frame. [Wade, 2008]

The Quake engine used BSP trees as a way break up the static world into pieces that could be culled for rendering and other optimizations. The Quake engine broke the static geometry up into convex pieces at the leaves. Subdivision stopped when the remaining BSP had no remaining concave areas. [Abrash, 1996]

BSP trees are not suited for dynamic positions of objects. To contain dynamic data, the data must be inserted into the tree every frame. This requires unnecessary calls that could be avoided by using a container that is designed to partition dynamic point data. Trying to speed this up in a concurrent environment is feasible but would really only benefit the insertions into the tree each frame. The overall structure is just not designed for the type of data that the simulation is working with. Data that is stored in a BSP tree is relative to the data that is already in the tree. When every piece of data can move in a single frame, the entire tree would have to be rebuilt. During one frame, data point A could be on the right side of data point B, in the next frame, it could move. In the simulation created in this project, the tree

would constantly need to be rebuilt because the objects are constantly roaming around in the scene.

In Review:

Currently, the industry does not apply thread-safe techniques to real-time spatial containers. As multi-core processors become more accessible, algorithms and programming paradigms will need to change to embrace these new architectures. Game programming and design will become a parallel process, as it has been on the GPU for years. In the future, the GPU may not even exist; CPUs will contain many cores, some that are specialized to certain tasks such as graphics or physics processing.

The table below shows the estimated difficulty of using each partitioning structure for the highly dynamic simulation that is planned for the project. While some structures may excel at one scenario, the one currently being planned for is a scene consisting of many dynamic objects or points moving randomly.

	Insert	Delete	Concurrency	Rebalance	Data	Dimensions
Sphere - trees	Hard	Hard	Very hard	Hard	Bounding sphere	3
k-d tree	Medium	Hard	Very hard	Hard	Positions	K
octree	Medium	Medium	Hard	Easy	Positions	3
quadtree	Medium	Medium	Hard	Easy	Positions	2
AABB tree	Very hard	Very hard	Very hard	Hard	Bounding box	3
BSP	Very hard	Very hard	Very hard	N/A	Planes	3

Table 1: Difficulty of thread safety for dynamic data.

The table above shows the difficulty to get the container thread-safe due to data type (i.e. point, bounding primitive, or planar, and dynamic or static).

Threading and Synchronization

Adding threading to a project introduces a problem of synchronization. When a thread is modifying data, it can be interrupted and lose the processor. Another thread can then gain the processor, and if working with the same data, can change it before the first thread finishes. This creates a race condition between threads and usually a corruption of data if not handled correctly.

The problem in a quadtree is that other threads rely on the tree to be in a consistent state at all times. When one thread makes a large change to the tree, like partitioning or collapsing a node, other threads can be drastically affected. At one time they may think that children exist, and then the next time they get the processor, they are gone. When a thread needs to make a change to the container, i.e. rebalancing, inserting or deleting objects, and resizing nodes, a portion of the tree will have to be temporarily locked. During this time, other changes cannot take place in the same location of the tree, because of possible data corruption. To combat this, there are a host of synchronization techniques available to ensure synchronization among threads.

Coarse-grain Synchronization

Coarse-grain synchronization uses locking objects at a higher level in the code to gain access to a resource. Coarse-grain synchronization does not allow for the best performance gains because threads “take turns” accessing a resource. While in control of a resource, a thread is allowed to modify it and make changes without other threads interrupting its task. This synchronization method is the easiest to implement but it usually the slowest. An example of coarse-grain synchronization would be locking down an entire tree at the root node and only allowing a single thread to traverse down and perform its task.

Fine-grain Synchronization

Fine-grain synchronization allows multiple threads to work on a container at one time. It differs from coarse-grain synchronization because it does not lock the entire container. By pushing the locking further down in the code, more access to the container can happen concurrently. However, when a thread wants to modify the container, it still must lock the container to prevent data corruption. As an improvement, fine-grain synchronization locks a minimal set of data needed for modification, reducing lock contention on shared objects, thus improving performance. For example, in a first-in first-out (FIFO) queue, a thread that is pushing an object onto the back of the list does not need to lock the head of the container. This allows another thread to remove the first item concurrently. By having two separate locks, the tasks can be handled at the same time. There are still two scenarios where contention occurs. One is when two threads want to perform the same action, for example, pushing or popping. The second is when there is only one object in the queue such that pushing and popping would both have to modify the same node.

Coarse-grain and fine-grain synchronization is like a gradient. On one end, coarse-grain, the container is completely locked every time it is accessed, whether reading, writing, or modifying the contents. However, with careful design, the container can only lock certain areas at a time. For example, only locking the head node of a linked-list when inserting, leaves the tail end open for a pop to occur. When trying to make a resource fine-grained, the amount of time spent in the synchronization primitives should be assessed. An elegant solution will use few primitives while keeping the data safe from corruption.



Figure 1: Gradient of Fine vs Coarse synchronization.

Synchronization Primitives

The following sub-sections describe different types of locking techniques that are available when using the Windows threading API. All of these locking mechanisms can be used in both of the situations above. How they are used determines what category the synchronization is placed in.

Thread Local Storage

All threads of an application can access the address space of the process, including static and global variables. Function parameters and local variables are unique to a thread. Thread local storage allows each thread to have the same variable but reference different parts of memory. This is done by having indices into an array. The thread can enable thread local storage and store data or pointers into the array. For example, each thread could access the tenth index into its local storage. The values will differ for each thread but they access them in the same way. [Microsoft, 2008]

Mutex

A mutex is an object that can only be “owned” by one thread at a time. The name comes from the words “mutual exclusion.” By providing exclusive access to a resource, threads can ensure that no other thread is going to change the data that they are about to access. However, for this to work, any time a thread wishes to access that resource, it must try to acquire the mutex. If another thread is currently using it, it waits. It is up to the programmer to determine how to protect data. They must remember to protect every part of code that access the resource or it could become corrupted. [Microsoft, 2008]

Events

Another synchronization object is the event. An event is an object that can be signaled and allow a blocking thread to continue execution. An event can be explicitly set to signaled or non-signaled. An event can also be created to reset automatically or manually. When an auto-

reset event allows a thread to pass, it goes back to a non-signaled state. A manual reset event must be explicitly set to non-signaled by resetting the event. [Microsoft, 2008]

Events can be thought of as gates. They can be set up to allow all threads to pass until the gate is closed, or they can allow only one thread to continue and then close immediately. Threads can wait on events just like other synchronization primitives, but they should not be used to safeguard data. They are more of a signal to tell other threads that something has occurred.

Semaphore

A semaphore is similar to an event but contains a count of how many objects are still allowed to enter it. As a thread enters a semaphore, its count is decremented. A semaphore will let threads through until its count reaches zero. When this happens, all future threads will block until it is incremented again. There is no guaranteed order to threads that are waiting on a semaphore are released when its count is incremented. All that is guaranteed is that one thread will continue execution. [Microsoft, 2008]

Semaphores help in a producer/consumer scenario. As the producer adds an item to a list or queue, it can increment the semaphore. Multiple consumer threads wait on the semaphore and every time it is incremented, one is released and accesses the data submitted by the producer.

Lock-free Synchronization

All of the blocking mechanisms above cause the thread to wait until the lock permits it to continue. When this happens, the thread usually loses the processor. Losing the processor can be a good or bad thing. If the object that the thread is waiting on is going to take a relatively long time to become available, then waiting might not be so bad. However, if the thread that is currently working with the protected data is almost done then waiting can be costly. In the simulation waiting could be costly because every time a thread wishes to access a node, it would have to block. Creating a lock-free algorithm could allow threads to wait a minimal

amount of time and not have to spend the clock cycles to swap off of and back on to the processor if they waited.

Lock-free techniques do not use synchronization objects and therefore, are not susceptible to the classic synchronization problem of dead-locking. Dead-lock occurs when two threads each hold a synchronization object that the other needs. Neither thread can continue execution because they are waiting for the other to release the object it holds.

This does not mean that the system cannot crash. Instead, threads can get into a state called live-lock. While in this state, the thread continually tries to accomplish its current task but makes no progress.

Threads that incorporate lock-free synchronization help each other perform their tasks. When a container is updated, it is not necessarily left in prime condition. Because the container can be left in an incomplete state, other threads that enter the container perform checks to see if the container is inconsistent. If the container is not in a completed state, the thread updates the state of the container before making its modifications. By checking the state of the container first, threads can help finish the task of a previous thread. Threads that do not complete their task, but rely on others, use a lazy approach to synchronization. Lazy synchronization is very hard to implement because each thread must double check the state of the container at each step of execution. When multiple pointers are under contention, the code can be look very convoluted.

Compare and Swap

Compare and Swap (CAS) is an atomic function that operates on three values, X, Y, and Z. Because CAS is atomic, it is a single instruction and cannot be preempted while executing. CAS compares X to Y and if they are equal, stores Z in X. The function always returns the original value of X, changed or not. Figure 1 is a pseudo-C++ implementation of CAS performed on integers.

```
CompareAndSwap(int destination, int comperand, int value)
{
    int old = destination;

    if(destination == comperand)
    {
        destination = value;
    }
}
```

Load-Linked Store-Conditional

Load-Linked Store-Conditional is similar to compare-and-swap. When a value is read from a location in memory, that location is flagged as read. A following store-conditional call will fail if that memory has changed at all since the load was performed. This can actually be stronger than a compare-and-swap because CAS will succeed if the value changes and then changes back to the original value before the thread is successful. This problem is known as the ABA problem and is discussed later.

The following table shows processor and atomic synchronization functions that are supported.

	Intel (x86)	PowerPC
Compare and Swap	CMPXCHG	N/A
Double Compare and Swap	CMPXCHG8B	N/A
Load-Linked / Store-Conditional	N/A	lwarx (load) and stwcx (store)

Table 2: Processors and their supported atomic synchronization functions.

The Interlocked API of Windows contains a set of functions that execute atomically. An atomic operation is one that finishes completely or not at all. The interlocked functions can be used to increment, decrement, perform a number of bitwise operations, and exchange values. The compare and set functionality is performed by the InterlockedCompareExchange() function. Similar functions are InterlockedAnd() and InterlockedOr() which could be used to make changes on a bit level. [Microsoft, 2008]

ABA Problem

[Valois, 1995] describes a problem when using compare-and-swap to maintain a lock-free linked-list. The classic problem known as the ABA problem, is detrimental to a thread-safe container that is using lock-free techniques. An example of the ABA problem occurs when a thread is pointing to a value, A. It gets interrupted and a second thread comes along and modifies the pointer, and then switches it back. When the first thread gets the processor back, it thinks that its value is still correct even though the data has changed. This is a problem because the pointer is the same but the data is different. There is no check or guarantee that the first thread can now continue execution. A good example of this would be the second thread deletes, and then recreates a node that happens to get the same memory location as the original node. When the first thread continues, it thinks that its pointer is unchanged because it is pointing to a valid node when actually it is a completely different node than when it started.

Using double compare-and-swap can solve the ABA problem because the top half of the 64 bits can be used as a counter to mark whether a pointer has changed since it was last read or

not. In a container, this could be used to store extra pieces of data in the 64 bits. This could stop another thread from deleting a node because it is not empty, it is not currently being used, or it is not a leaf. Keeping all of this data with the pointer allows the state of the node to be passed in and operated on atomically.

For example, the bottom 32 bits are the actual pointer and the top 32 bits are used as a counter. When a thread wants to change the value of the pointer, it first reads the pointer to a temporary variable on its stack. This value will not change by another thread because it is local to the thread. The thread can then modify the pointer as it needs to and then increment the counter. To make the change permanent to the structure, the thread then uses compare and swap with the new value (pointer and incremented counter), the old value, (old counter value and old pointer). If the compare-and-swap is successful, then the data has not changed since the thread read the value from memory. If it has failed, then the counter and the pointer have changed. The ABA problem is prevented because even if the pointer value changes, and then changes back to what it was at the time of reading, then the counter value will still have changed and the compare-and-swap fails.

One solution to the ABA problem is the double compare and swap (CAS2) operation. This operation compares two values against two operands. By using a CAS2 operation on a pointer and a value, the counter can be incremented every time the memory is written to. If another thread initiates the ABA problem, the first thread's CAS2 operation will fail because the counter will be of a different value than when it started. When CAS2 is not available on a processor, it can be simulated using CAS64. This function performs a compare and swap on 64-bit memory. This allows a 32-bit pointer to be stored in the first half, and a counter in the second half.

Methodology

The thesis designs and implements a thread-safe spatial partitioning grid. The problem is how to allow multiple threads to manipulate a spatial container. Since spatial containers are the heart of many algorithms in game programming, i.e. physics, artificial intelligence, and culling, having a container that can allow these to be executed simultaneously allows areas of the game update loop to be executed in parallel. Alternatively, more work to be done in the same amount of time for all of the areas compared to if they are executed linearly.

The project is beneficial to game architectures that want to utilize threading on a system-wide basis. Sections such as broad-phase physics and render culling that do not modify the database can be executed in parallel while narrow-phase physics and AI techniques can be updating the contents of the container. Games in the future will need many types of thread-safe containers that are accessible from threads on multiple cores at the same time.

To test the implementation, a large scale AI simulation has been developed on top of it. The simulation has several colored armies; each controlled by a separate thread. When a unit is created or destroyed the parent thread will add or remove the unit from the grid to keep it in synch with the scene. Having many threads and many units, should present large contention on the container and allow it to shine. When updating units, the threads query the database about the locations of nearby enemy units, and order them to attack. The threads also update all of the positions of their team within the spatial database.

The initial goal of the project was to create a quadtree that can handle large amounts of dynamic objects, all being updated by multiple threads at the same time. After many unsuccessful attempts to get every edge case safe from dead-locking, the project was re-assessed and a container was designed to solve the problem at hand. The current structure, a thread-safe grid, uses a lock-free approach such that any thread can access any part of the container.

The large-scale army simulation proposed relates easily to a real-time strategy game. Most RTS games are relatively two dimensional. The worlds created in today's RTS games are three dimensional but units mostly remain on a 2D plane of the ground. Even flying units in current games are usually low-flying and can be represented in the 2D plane.

Because of the planar positioning of current objects in games, a spatial grid is best suited to contain the units. If an octree was used, there would be many empty nodes at the top of the tree that would waste resources. The threads should be able to insert, delete, and update the objects that they are responsible for while maintaining a valid grid. The grid is defined by a length and width of the simulation world.

The second part was used to create a simulation that utilizes the container and allows threads to insert, delete, and query units that are contained in it. The simulation supports a dynamic number of threads that will be specifiable through an .ini file. When the simulation is started, the grid and threads are initialized and begin execution. Throughout the life-span of the simulation, the threads use the insert, remove, and radius query functions supplied by the container's API and AI techniques to drive a large-scale battle.

The third part was used for testing the quality of the simulation and the thread-safe container as well as creating a friendly user interface for running the simulation. Profiling the execution of the simulation with different numbers of threads and units was done on dual-core and quad-core processors to see how the simulation scales between the number of cores available.

The Product

Initially, from the field review of current spatial indexing containers, a dynamic quadtree was chosen for the project. When several thousand objects are moving every frame, balancing a tree such as a k-d tree or BSP tree becomes inefficient and would have to be rebuilt from scratch each time. A quadtree would have allowed objects to be partitioned based on position and if modified to only contain units in the leaves, could allow for contention on the container to be moved down to that level. The state of a node could be contained in 64 bits and operated on atomically.

Most of the project time was spent trying to create a lock-free quadtree. Even though this attempt was not successful, it taught much more about the problem space than initially gathered from the field review. The failed attempt at a quadtree gave a better understanding of the problem and eventually allowed an alternative solution that is similar to a quadtree. The following sections describe the attempted quadtree and its various functions, and then go over the spatial grid and its design and functionality.

Thread-safe QuadTree

Creating a quadtree that uses no critical sections, mutexes or semaphores is a difficult task. An attempt to use compare-and-swap to mark nodes as in-use, containing children, or whether the children are a leaf or parent was implemented.

This implementation relied heavily on the use of compare-and-swap (CAS), an atomic function supported by Intel's x86 processor architecture. CAS is supported by the InterlockCompareExchange functions in the Windows API and supports 32-bit values or 64-bit values depending on the operating system. The quadtree used the 64-bit version.

To deal with nodes of a quadtree at an atomic level, the 64 bits were split up carefully. The implementation used 32 bits for a pointer, 14 bits to hold the number of units stored in the children of a parent and 14 bits to represent the number of threads that are currently in a level of the tree. These add up to 60 bits and can be used to atomically represent the state of a node while using compare-and-swap. The last 4 bits represent whether the children are leaves or parents themselves.

Additionally, the nodes contained a struct that held the pointers to the children. This struct is created and uses the 4/14/14/32 bit-model described above. When it is time to partition or collapse a parent node, the struct's pointer can be acted upon atomically instead of using 4 separate calls, one for each child.

Insert

The insert function has the following pseudo-code:

```
Node::Insert(Unit* cur_unit)
while (true)
{
    if (leaf)
    {
        if (full)
        {
            bool partition_successful = Partition();
            Insert(cur_unit);
        }
    }
}
```

Figure 3: QuadTree::Insert function pseudo-code

Inserting a unit into the tree should never fail. The while-loop allows an entering thread to keep trying to insert a unit until it succeeds. If two threads are trying to insert two different units at the same time, one will succeed. The other will fail, go to the top of the loop and try again. If a thread comes to a full leaf, it will partition it and put the units that are currently stored in the old leaf, into its children. If two threads try to partition a full leaf at the same time, only one will succeed. The other will try to insert its original unit and then leave. The thread that wins the race condition will move the units to the children and continue on.

Partition

The partition function has the following pseudo-code:

```
Create struct that contains children
Create children
Use compare-and-swap to add the struct into the tree
if(!successful)
    Free created children and struct
    Return false
Mark as not a leaf
Return true
```

When a node needs to be partitioned, a struct to contain the children is built, the children are created, and then the struct is swapped into the tree using `InterlockedCompareExchange`. If the atomic call is successful, then the children become part of the tree, if it fails, then another node has already partitioned the current node and they are not needed. The thread can destroy the created nodes. The function returns true or false, whether the calling thread was successful in partitioning the node. This is helpful when two or more threads find a full leaf that needs to be partitioned. Only one will successfully partition the tree and the other can ignore it.

Collapse

The collapse function has the following pseudo-code:

```
if(all children are leaves)
    Try to swap out struct of children with 0
    if(successful)
        free children and struct
    else
        return
```

The collapse function cleans up leaves that are no longer in use. For this to be successful, there must be no units stored in any of the children, there must be no threads currently in any of the children, and all the children themselves must be leaves. This is where the 4/14/14/32 bit values are used.

Any time a thread needs to go deeper into the tree, it must first try to mark the children as in use. Also, whenever a thread adds a unit to a leaf, it increments the leaf's parent struct. When a collapse tries to take places, a swap with only the bottom 32 bits is attempted. If there are any bits set above this, because of active threads or units stored in the children the test will fail and the leaf nodes will not be removed.

Remove

Each unit has a pointer to the node that they are held in. When a thread recognizes that a unit has moved out of bounds of the node that contains it, it must remove it.

The remove function has the following pseudo-code:

```
//This calls node::remove
unit->m_node->remove(unit)

Node::Remove(Unit* unit)
for each unit u in node:
    if CAS(u,0,unit) == unit
        return true
```

In the pseudo-code, if the compare-and-swap call returns a value that is equal to the unit that is to be removed, then the current thread knows that the call succeeded and replaced the unit in the node with null. If this happens, the unit is no longer in the tree and the thread can return successfully. Otherwise, the call failed and another thread has moved the unit. The other thread is in the process of moving the unit to another node because of a partition, and the current thread will have to try again.

The remove is fairly straight forward. It attempts to remove a unit from a node and if it is successful, it returns true. Because units can be moved by another thread that may be currently partitioning a node that contains a child, their pointer to the node that contains them may not always be up-to-date. When this occurs, the remove fails and the calling thread needs to try again.

Radius Query

The radius query function has the following pseudo-code:

```
if(!AABB.SphereIntersect(location, radius)
    Return;

if(leaf)
    Check stored units if they are in the radius and
    add them accordingly
else
    Call radius query on children
```

The quadtree supported a radius query function that allows calling threads to specify a vector location and a radius and get back the units that fall within the area of the defined circle. An early test whether the passed in values intersect with the quadtree's bounding box allow the algorithm to determine whether it needs to test any further or not.

Grid

Unfortunately, the quadtree approach was unsuccessful. It did not succeed because multiple threads were constantly trying to move and change the same data. Enough rules and algorithms to prevent every possible chance for one thread to corrupt what another was about to do could not be created. Because safety could not be guaranteed, another implementation had to be created to solve the problem at hand; how to create a thread-safe spatial container.

Overview

The grid approach is similar to the quadtree in that it partitions data in a 2D space. The grid is composed of a large array of cells. A cell is a statically created bounding-box that can contain units. Each cell contains a bit-vector, whose bit positions represent an index into a global list of units.

One of the main problems with the quadtree was the repartitioning of nodes. Each node in the implementation could contain a maximum of four units. When this limit was reached, the node would be partitioned, and the units moved down to the children. This created a lot of contention on the units and their positions in the tree. At one moment, the unit could be in node A and the next it would be in node B. This was a problem because any thread could have moved the unit to the new node. The thread that was responsible for moving the unit around in the world, or a thread that had to repartition could have moved the unit.

The bit-vector approach mitigates this problem because there is no longer only four slots that units must fight over in the cell. Instead, if unit 467 is in the cell, then bit number 467 will be set to 1 in the bit-vector. This allows each unit to have its own place in a cell. Inserts, removes, and queries can all be performed quickly because the cell that a unit is in can be determined by transforming its world position into grid space. This will be discussed more in the specific algorithms.

The grid/cell approach also removes the repartitioning that was occurring in the quadtree. Instead the grid is pre-built to a specified number of cells wide and long, with a certain cell width. The grid is static and the structure itself is not changed throughout the simulation.

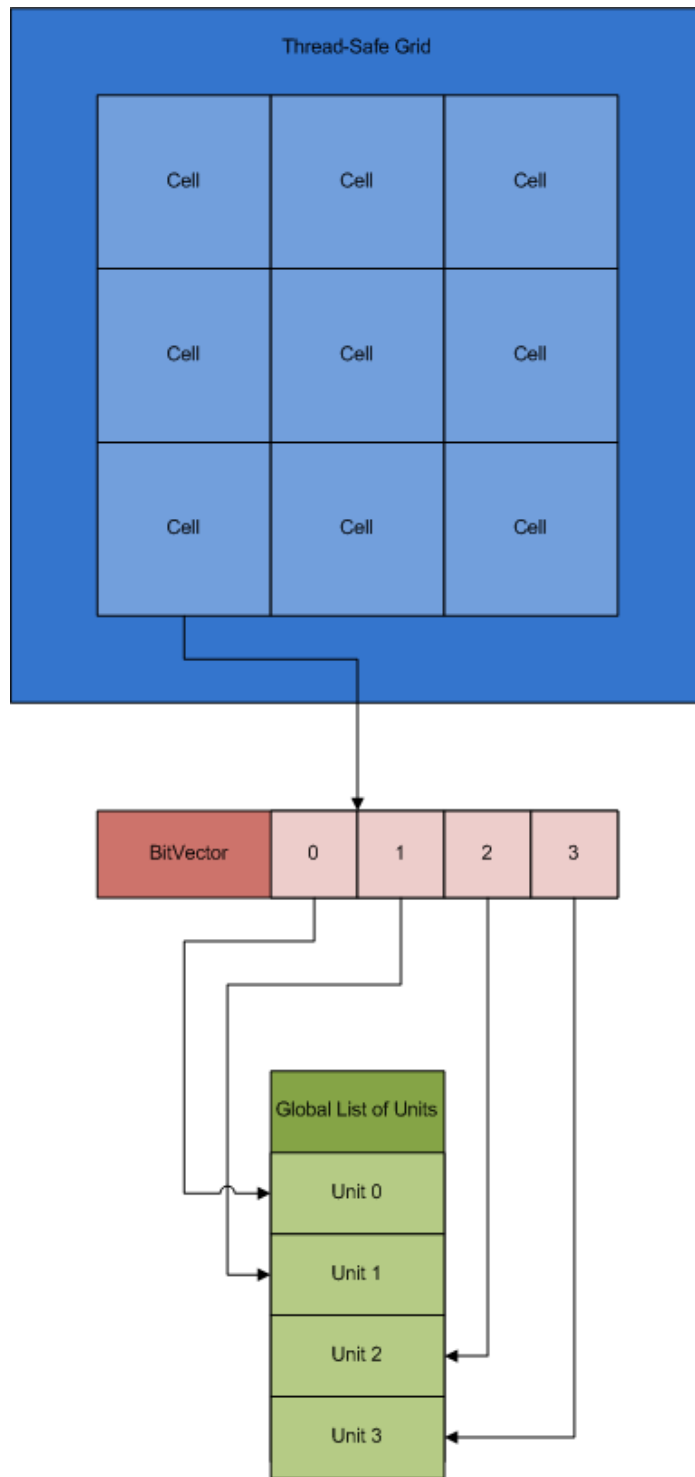


Figure 8: Design of Spatial Grid

Insert

Inserting into the grid is a fairly quick process. The specific cell that a unit belongs in can be calculated from its position in the world. Once the cell is determined, the specific bit that the unit is in can be toggled using the atomic function `InterlockedAnd()`. Because this function is atomic, it is guaranteed to be successful and the unit will be successfully added to the cell. Another benefit of the atomic function is that it does not block the thread in any way.

Remove

Removing a unit from the grid is very similar to insert. The units' containing cell is determined from its position and then `InterlockedOr()` is used to set the bit to zero in the bit-vector. This atomic function is also guaranteed to succeed so the unit will always be removed from the cell containing it.

Query

Querying the grid for neighboring units is simple. If the grid is thought of as a 2D surface, the upper-left and lower-right positions of a box can be determined from the unit's position and its query radius. Once the row and column of these bounding positions are calculated, the cells that make them up can be iterated over and queried against.

When querying against a particular cell, the bit-vector is iterated over and the individual bit positions that are set are tested. If the position has a value of one, then that unit is in the cell.

It is important to note that there is a possibility that the container is not completely up-to-date. One unit could be querying a particular cell for targets and be half way through testing the cells bit-vector for set positions, when another thread places a unit with a lower index that what has already been tested into the cell. This imperfection is currently handled because each unit updates its target every frame. If the new unit is not in the current query, it will most likely be a part of the next.

AI System

The AI system consists of a dynamic set of threads that each control an army on a planar battlefield. The simulation mimics a real-time strategy (RTS) game because often these types of games contain a large number of soldiers, tanks, etc that are moving around. Usually in an RTS, when units come in range of each other, they automatically attack. The simulation has this rudimentary AI and uses the thread-safe grid to quickly determine units that are near.

Each thread maintains a list of units that it is responsible for. The respawn rate is changeable through an on-screen graphical user interface. The threads concurrently control their armies querying the grid for potential targets. When an enemy is close, the thread tells its units to attack that enemy target. As units die in the simulation, it is the threads responsibility to remove them from the grid. The threads also contain a limit of the number of units that they can control. When they are under this population cap, they spawn a new unit into the simulation which requires an immediate insertion into the container as well. The threads are also able to move their units around the map in an attempt to find enemy units and engage in combat. Every time a unit is moved, the thread updates the grid with the new position of the unit.

The units only have one state of movement and one state of attacking. The units are controlled by one of the threads, a user can select an army and assign them to attack another army in the simulation.

The targeting and attacking of the units is controlled by the AI. When two units move within distance of each other, they attack. Each update, a unit queries the container for a list of units that are in range of it. It selects one of the units that are not of the same faction as it and begin to attack it.

The data-driven system for the AI allows the simulation to be changed without having to recompile. A list of the data in the ini file and the default values are below:

AI

- Population Cap – 1000
- Respawn Rate – 10ms

Units

- Health – 100
- Attack Radius – 30
- Damage range – [1,5]
- Speed – 10

Rendering System

Rendering of the system is different than most single-threaded applications. The simulation uses two large buffers to render the point-sprites that represent the units and the nodes of the container. Each unit has an index into the point-sprite buffer and can update its position and color as needed. The container buffer is a large buffer of lines that is updated as each node is rendered. When it is time to render, the buffers are updated on the graphics card and then drawn.

Profiling System

The profiling system used scoped samples to collect timings of the code. Samples were used to collect the amount of time that the parallel code took to run. To increase performance and accuracy of the profiling system, a large number of samples are collected and then written to disk at once. In the meantime, samples are stored in memory until full. The samples logged data to a text file on the disk. The raw timings were then run through a python script that calculated the average times for that run. This was repeated many times for each of the data points in the graphs of the results section. After each run, the timings were collected into an excel sheet and the graphs below were produced.

Testing and Data Collection

With a massive number of units in the simulation, it is suspected a linear approach would not be able to compete with multiple threads who can divide the work evenly. As the number of cores increases, the work can be split even further and potentially more units added to the simulation.

Amdahl's Law is a formula that determines the expected speed up of an algorithm relative to a linear implementation. My project encompasses a small but important part of a games' AI and update routines. Amdahl's law predicts that this implementation would be able to update the units' positions and targets in $1/n$ amount of time that a linear implementation would take, where n is the number of threads. However, because the entire application is not running in parallel, this does not directly correlate to a speed up of n times.

A linear implementation with similar logic was timed. The linear implementation had 4 AI entities each controlling 500 units. In those tests, profiling gathered the amount of time that it took one AI to update 500 units along with the frame time for reference. It took approximately 4ms per AI entity to update 500 units and a total frame time of 20ms. Already, only 500 units per AI and the frame time exceeds 16.66ms (60fps).

Assuming there are four threads running. If each thread has 500 units, and takes 4ms, to update them, it would take a linear implementation approximately 16ms to update. This was shown in the test above. For the parallel version, it is estimated that it would take 8ms + 1ms for moving threads on to and off of the processor on the dual-core machine. On a quad-core machine it is expected to be able to update all of the units in 4-5ms.

In Amdahl's formula below, P represents the percent of work that can be parallelized and N represents the number of processors. 80% of the application time is spent updating the units, and that is the area of code that is going to be run in parallel.

$$\frac{1}{(1 - P) + \frac{P}{N}}$$

The maximum speed up possible can be calculated by taking the formula with N going to infinity. As this happens, the formula becomes $1/(1-P)$, in my case $1/(1-.8)$, for a maximum potential speed up of 20x.

Testing on dual-core and quad-core machines and comparing the results against the linear implementation is one of the goals of the project. This was accomplished and recorded in the following section. The table below shows the expectations and the actual results that were collected from profiling.

	Dual-Core		Quad-Core	
	Expected	Actual	Expected	Actual
Update Time	9ms	9.92ms	5ms	6.5ms
% Speed Up	77%	61%	220%	146%

Table 3: Predicted vs Actual Gains

Results and Analysis

Profiling

The most important data to collect for the project was how long updating took as the number of cores scaled. To do this, a total number of 2000 units was split among an increasing number of threads.

Timings were collected using a custom profiling API that used scoped samples. A sample is started when it is declared and ended when it goes out of scope. As samples were finished, they were written out to file and then analyzed.

Invasive profiling such as this can slow down a simulation because of the time that it takes to create, destroy, collect, and record statistics to disk. Because of this, number can be significantly skewed, especially because of writing to disk.

Even though the quadtree was not successful, it is interesting to collect statistics about its performance. The system was slightly modified to count how many times a compare and swap failed per modification of the tree. This shows the contention on the container because the function would only fail if another thread was accessing the data at the same time. The data was collected for the six functions that were used to make changes to the 64bits that represented the children of a node. Below is a table and graph of the data.

By analyzing it, it may give some insights on where future work may be put into the project. One of the first things to notice is that LeafDecrement had no data. LeafDecrement is only called when a node collapses its children and needs to tell its parent that it is now a leaf. Data was collected by starting the simulation, spawning in the number of threads represented, letting them reach their population cap and then begin profiling. The interesting part here is that the tree was built to size that it needed to be, yet there were so many units moving around that no nodes' children ever became completely empty so that it could collapse. To make sure that the testing code was working properly, the simulation was started with profiling enabled immediately and in the beginning, nodes were collapsed and data was collected about them.

By looking at the rest of the graph, the data is very close to one with exceptions in a few places. These low numbers mean that there is very little contention at all. Data in the LeafIncrement column shows that nodes were created and during 12, 14, and 20, threads tried to partition nodes at the same time. This can also be seen in the graph where there are spikes along the green line. This is most likely caused by the large amount of relative time that it takes to create a new partition in the tree. The thread that is partitioning must move all of the units that were in the tree down to the children.

The orange and blue lines, which represent a thread marking and unmarking a node's children as "in-use" almost follow the same pattern throughout the graph. This is probably because these are the two most used functions in the quadtree. They are used by threads to mark nodes that they are currently visiting and are part of the insert and query functions. I

believe that these two graphs show the real “atomicity” of the compare and swap function. When multiple threads are trying to make changes with compare and swap, only one can succeed and the resulting graphs show the retry rates of the other threads. The horizontal axis represents the increasing number of threads. The vertical axis represents the number of compare-and-swap calls compared to the number of increment or decrement calls. If there was no contention on the container, the number of CAS calls would equal the number of increment or decrement calls. When contention occurs, the number of CAS attempts increases, and the value becomes slightly greater than one.

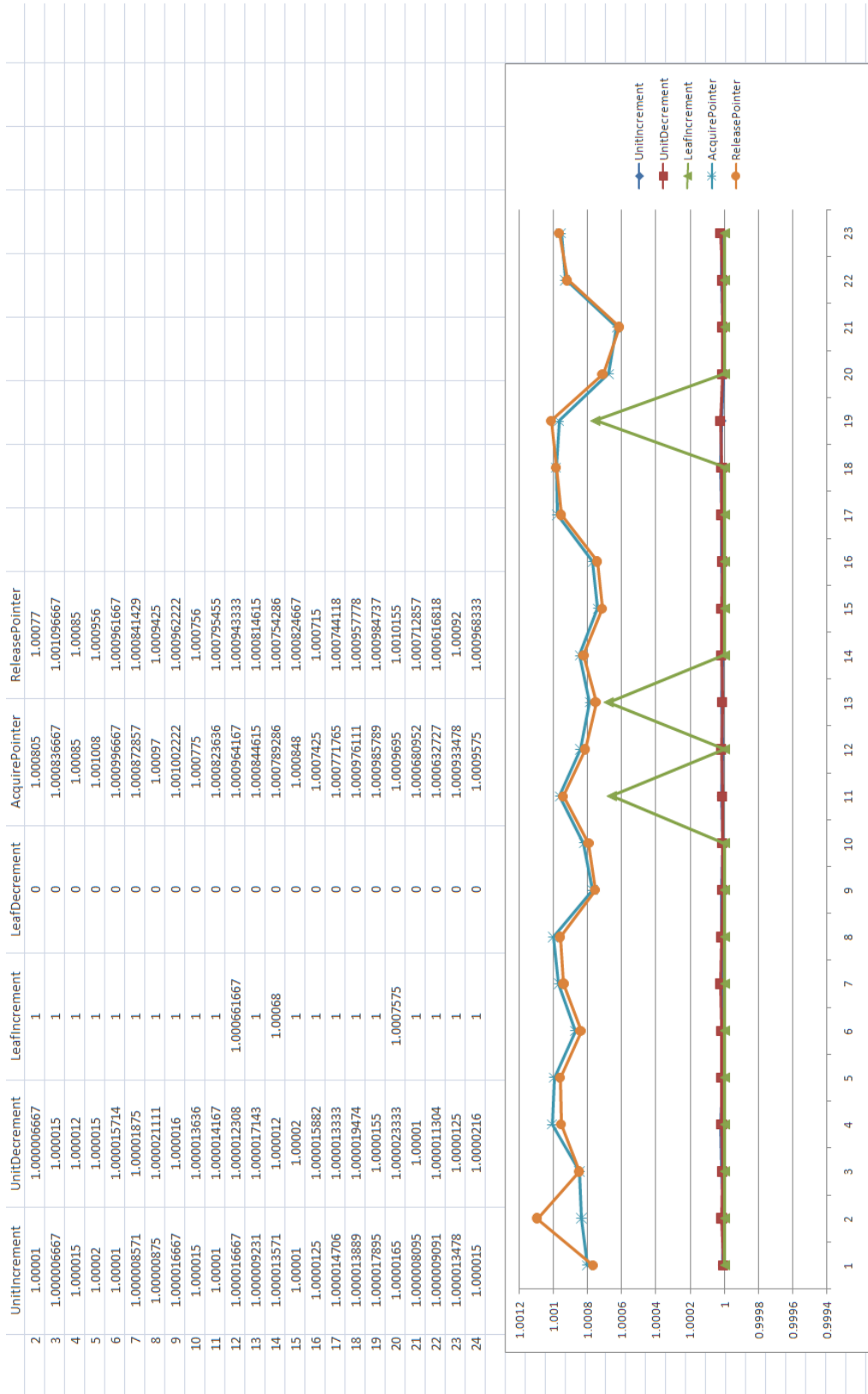


Figure 9: Table and graph of QuadTree contention data.

The following data shows the performance of the simulation on a dual-core and quad-core machine. The machine specifications are as follows:

Dual-Core:

Dell XPS M1710 Laptop

Intel Core 2 @ 2.16Ghz

2GB of RAM

nVidia 7950GTX 512MB Graphics

Quad-Core

Intel Core 2 Quad CPU Q6600 2.4GHz

3326MB Ram

nVidia GeForce 8600 GTS 1645MB

One of the tests performed was how well the simulation and container could handle an increasing number of threads each with an increasing number of units. The following two tables show how the update times scaled depending on the total number of units and threads in the simulation.

The y-axis of the grids represents the average frame times, the x-axis represents the number of threads in the simulation, and the colored lines represent how many units each thread was updating during that test.

Dual-Core

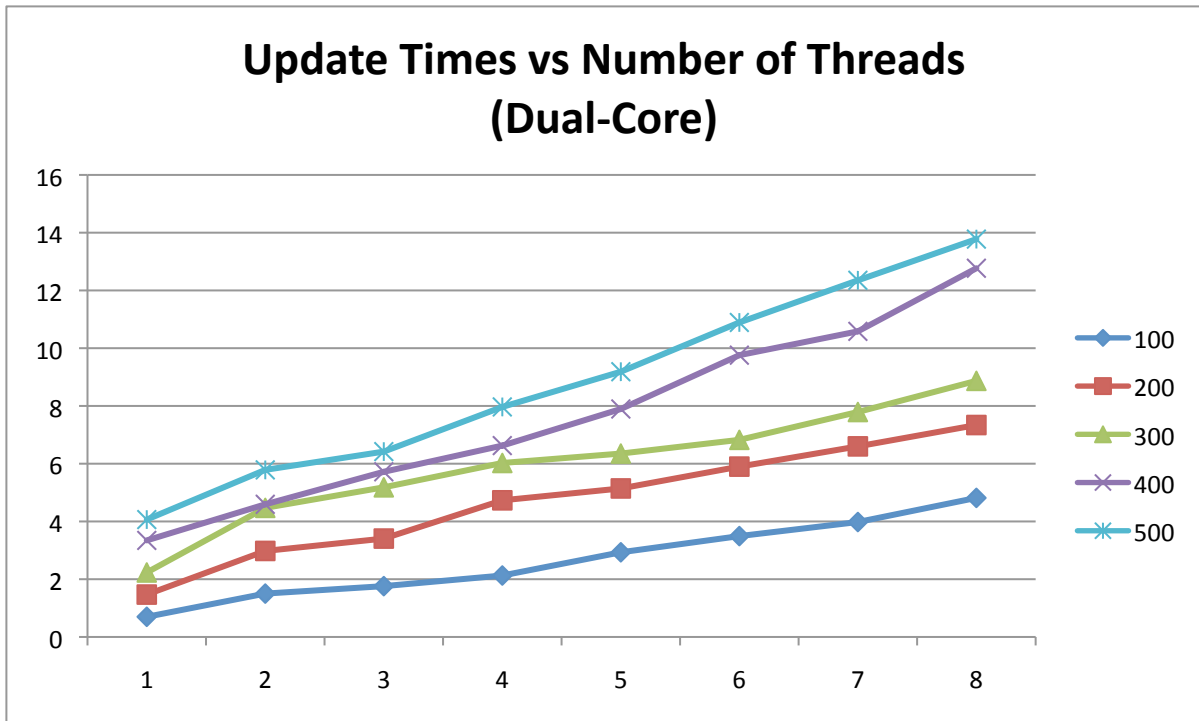


Figure 10: Update Times vs Number of Threads (Dual-Core)

Quad-Core

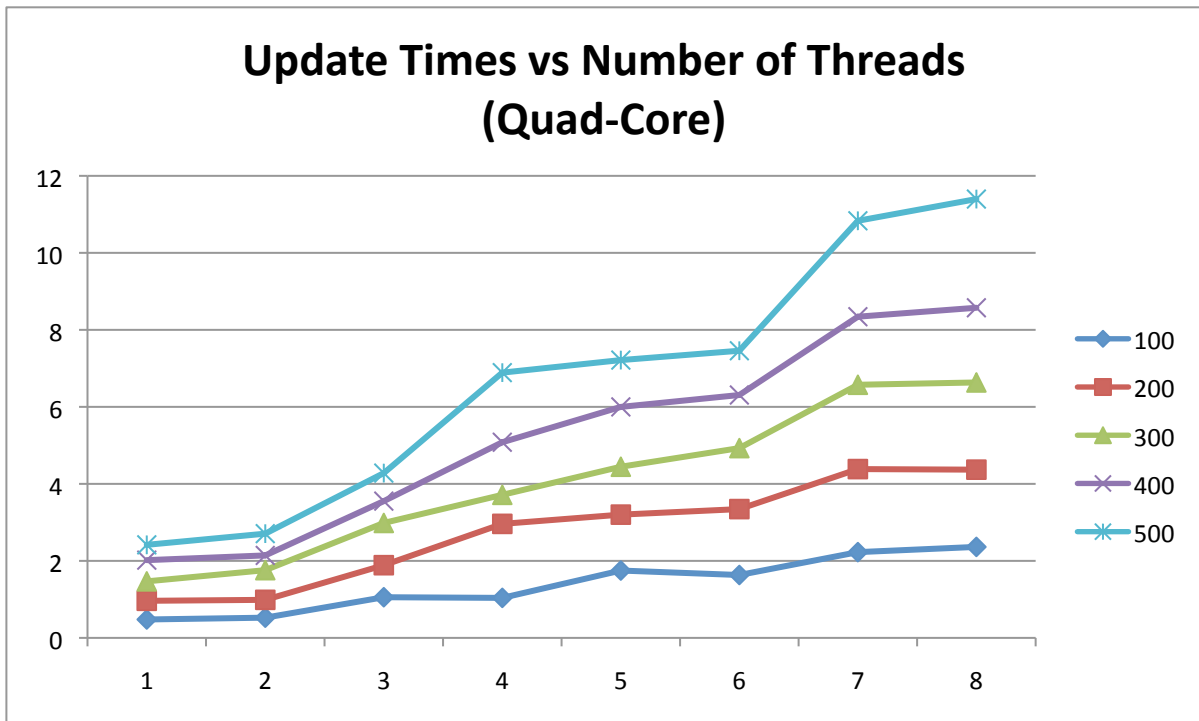


Figure 11: Update Times vs Number of Threads (Quad-Core)

Another important test to perform is how the system performs when an increasing number of threads updates a fixed number of units. In this test, the total population cap for the simulation was set to 2000. As the number of threads increased from 2 to 24, the number of units that each thread was responsible for updating decreased. The following graphs show the average update times for the increasing number of threads.

The y-axis represents the average update times for the threads. The number of threads in the simulation is on the x-axis.

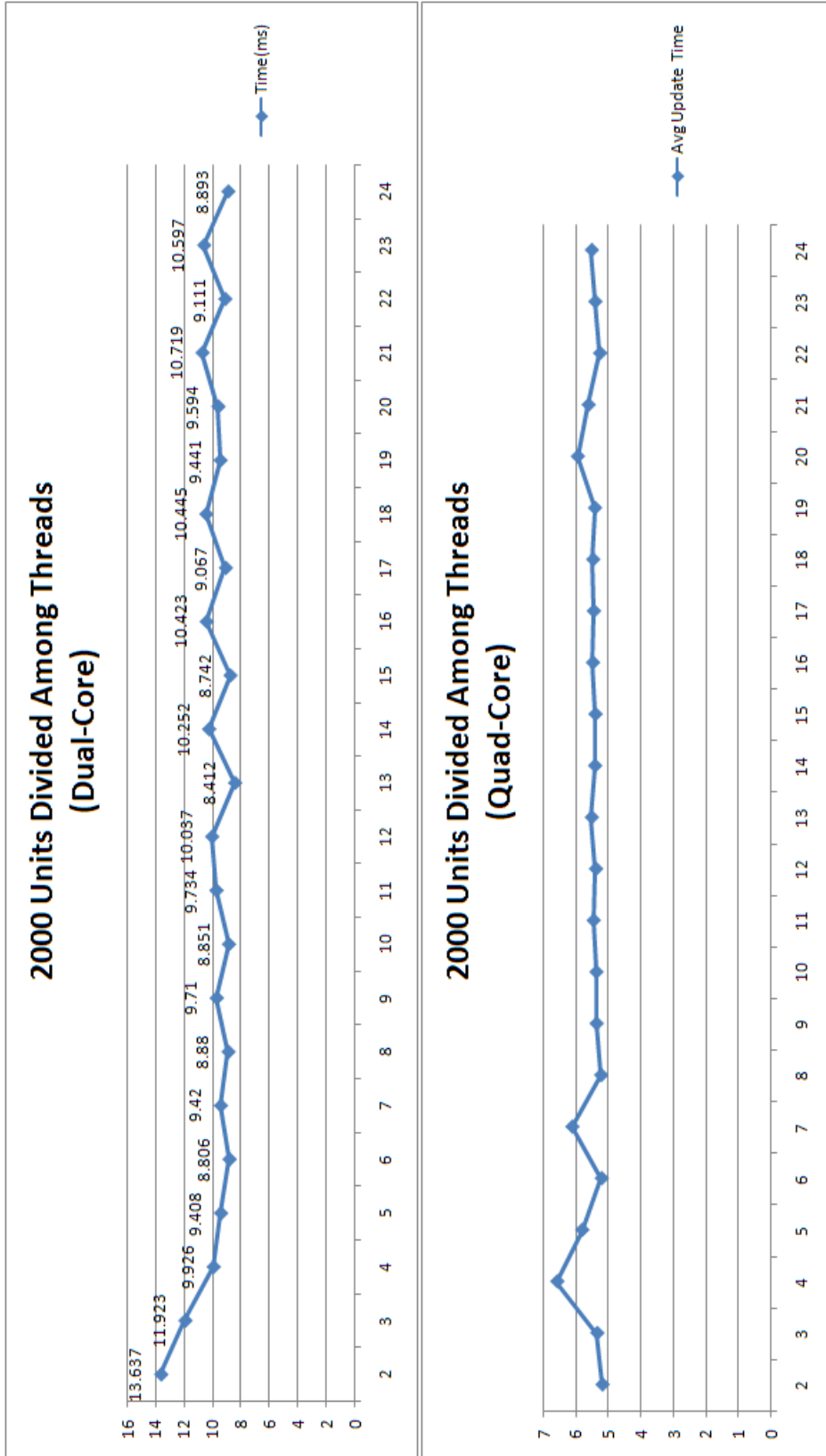


Figure 12: 2000 Units Divided Among Threads

The last testing performed was how a naïve implementation of a thread-safe grid would perform. In this test, the grid was modified to have a critical section around the entire grid. Threads had to wait on the critical section until it was free. This created a scenario where threads were only allowed to access the tree one at a time. Below is a graph of the critical section performance and the lock-free performance on the dual-core processor.

Again, the horizontal graph shows an increasing number of threads. The vertical axis shows the number of milliseconds the threads took to update the units that they are responsible for, using the container to query, add, and remove its units from the simulation.

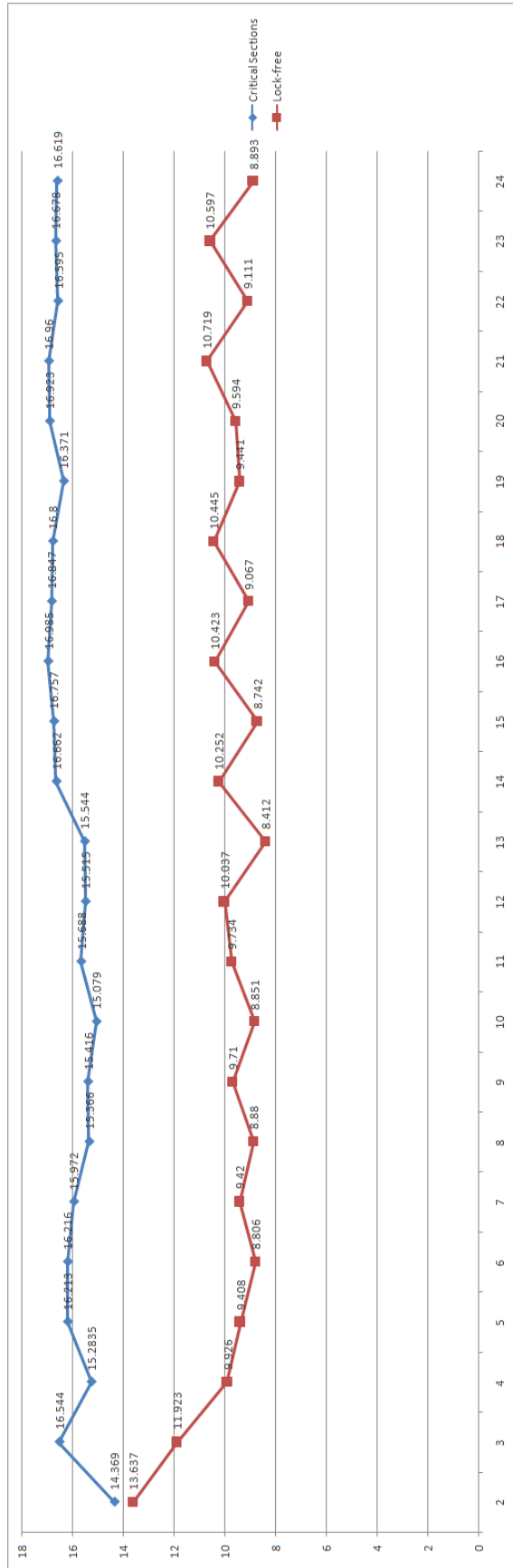


Figure 13: Graph of Coarse-Grain vs Lock-Free Implementation

Analysis

The following section discusses the tables and graphs above.

Critical Section vs Lock-free

The critical section grid shows decent performance, but by forcing each thread to wait its turn every time it wants to access the tree, it is basically creating a linear implementation, but with threads waiting to do work. The performance is still a little better than the linear implementation discussed previously. This is most likely because iterating through the lists of units that each thread is responsible for can be done outside the lock and the thread only needs to wait when it needs to access the tree. There is still some parallelization benefits but not as much as compared to the lock-free implementation.

When compared to the lock-free implementation, the graph shows the performance gains. This is because the lock is removed and threads can access the tree at the same time. The contention is pushed further down and allows more parallel work to be done. The graph and performance between a coarse-grain synchronized grid (using critical sections), and the lock-free grid, is a great example of the gradient in Figure 1.

Lock-free

By looking at the lock-free timings of the simulation, the initial hypothesis is marginally off. It was initially thought that a dual-core system would gain an 80% speed up. The hypothesis stated that a linear implementation with 2000 units took 16ms to update and 4ms to render and that a dual-core system would take 8ms to update. Using the thread-safe grid, the simulation time for each thread dropped to approximately 10 ms on the dual core machine and to approximately 5.5ms on the quad core machine.. The 61% speed up on the dual core was not as expected but is still a gain. On the quad-core however, it was predicted that update times would be around 4-5ms and got an average of 5.5ms. This is much closer to the prediction calculated with Amdahl's law.

The difference between the prediction and the actual results is due to the amount of work that was estimated to be parallel. Ahmdahl's law does not take into account reordering of code; this should be part of calculating the amount of code that is parallel. By using interlocked functions, it was assumed that there would be no reordering of code. Because the code is reordered to ensure safety, threads can be slowed down very slightly. However, when this happens many times a frame, the performance penalties add up. When trying to use Amdahl's law, it is important to know the benefits and pitfalls of every single line of code, whether the CPU will reorder code, and the instructions it will execute. The initial overestimations were due to lack of familiarity with threading in general and the experiences from this project will definitely help with making predictions in the future.

Additionally, There are many factors that affect the performance of multi-threaded simulations, especially on a personal computer. A computer simulates multi-tasking by quickly switching between different processes and executing a little of each one at a time. Multi-threaded applications can be hindered by this because while one of the threads is working, it can get interrupted for a system thread, an anti-virus process, or any other application that is running in the background. When designing for a multi-threaded system, the design should take into account that the application does not get 100% of the CPU's attention. The application can get interrupted at any time for another process.

Cache also plays an important role in the speed of a multi-threaded application. The quad-core machine was actually one of Intel's first four core processors. The system did not have a true quad-core. Instead, there were two dual-core processors on the same die. This can greatly affect performance because there is not a fast secondary cache for all processors to share. This can greatly affect the performance and could explain the spikes witnessed at the four and seven thread profilings of the quad-core system. Future work in the direction of this project might include researching and profiling the cache performance of the implementation, seeing how many cache misses occur, and trying to optimize the number of misses away.

The more cores that a system has, the easier it can balance the load. It is possible that the quad core machine did so much better because when there was an interruption from another

process, there were still three other cores still doing their work. The work being done only dropped from 100% to 75%. Whereas, on the dual core system, when one core gets lost to another process, the rate drops to 50%. The losses of performance decrease drastically as the number of cores increases. In 8 cores, an interrupting thread would only drop the rate to 87.5%.

By creating and using a thread-safe container, a larger speed up was expected. A threaded simulation does gain performance benefits because there is more processor power to be harnessed, but the algorithms that are required to keep data in a consistent state are much more difficult to implement. These difficulties come from the fact that now programmers must not think about one flow of execution, but many. Anytime data is accessed or modified from one thread, the programmer must think about the implications this can have on every other thread in the system. If other threads do not even interact with this data, that is great, but if there are interactions, precautions must be taken so that the two threads do not corrupt the data.

Multiple processor architectures still have to share a single memory and bus. Because of this, concurrency does not always guarantee a 100% speed up, even in the areas that are highly parallel. Whenever there is communication between threads, or shared resources such as a container, this affects the performance to be gained because threads must take into account synchronizing data. On top of this new programming mindset that the programmer must take on, C++ is not a very thread-safe language to begin with. Most containers that we are used to using are not designed for thread safety and cannot just be dropped into an application and accessed by multiple threads. As programming languages evolve, it would be beneficial for them to try and embrace concurrency.

Conclusion

Concurrency is becoming widely used in processor and computer architecture. To keep up with the hardware industry, programmers must learn about threading and thread-safety. By creating a new thread-safe container I learned about threading techniques, issues behind data contention and methods of preventing resource corruption. I also learned why professional programmers usually dread multi-threaded programming. It requires a completely different approach to solve problems because the programmer must think of all the possibilities of what other threads are doing that could hinder the current thread or corrupt data. In a concurrent environment, every possible place for data contention must be explored and proven that corruption cannot take place. If this cannot be done, then there will most likely be a bug in the implementation. It may not show up at first, but eventually it will and it can be catastrophic.

Future work in my field of study might include discovering exactly how much CPU time is lost to other processes. Tracking how many times the processor has a cache miss and must load data from memory, and then reworking some of the design to minimize these misses would also be beneficial. In the simulation, the benefits were not as fast as predicted for the dual-core machine and the quad-core machine did fairly well. I would like to see additional cores test and see how the simulation runs on Intel's new Larrabee project.

References

Falcone, J. (n.d.). *Xbox 360 vs. PlayStation 3: Clash of the Titans*. Retrieved September 9, 2008, from cnet reviews: http://reviews.cnet.com/1990-10109_7-6224258-1.html

Friskin, S. F., & Perry, R. N. (2002, November). *Simple and E*. Retrieved June 2, 2008, from <http://www.merl.com>: <http://www.merl.com/papers/docs/TR2002-41.pdf>

Gaede, V., & Gunther, O. (1997). Multidimensional Access Methods. *ACM Computing Surveys*, 170-231.

Microsoft. (2008, May 15). *Event Objects*. Retrieved June 2, 2008, from Microsoft Developer Network: [http://msdn.microsoft.com/en-us/library/ms682655\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms682655(VS.85).aspx)

Microsoft. (2008, May 15). *InterlockedCompareExchange Function*. Retrieved June 2, 2008, from Microsoft Developer Network: [http://msdn.microsoft.com/en-us/library/ms683560\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms683560(VS.85).aspx)

Microsoft. (2008, May 15). *Mutex Objects*. Retrieved June 2, 2008, from Microsoft Developer Network: [http://msdn.microsoft.com/en-us/library/ms684266\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms684266(VS.85).aspx)

Microsoft. (2008, May 15). *Semaphore Objects*. Retrieved June 2, 2008, from Microsoft Developer Network: [http://msdn.microsoft.com/en-us/library/ms685129\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms685129(VS.85).aspx)

Microsoft. (2008, May 15). *Thread Local Storage*. Retrieved June 2, 2008, from Microsoft Developer Network: [http://msdn.microsoft.com/en-us/library/ms686749\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms686749(VS.85).aspx)

Overmars, M. H., & van Kreveld, M. J. (1991). Divided k-d Trees. *Algorithmica*, 840-858.

Shagam, J., & Pfeiffer, J. J. (2003). *Dynamic Spatial Partitioning for Real-Time Visibility Determination*.

Stokes, J. (2007, April 26). *Clearing up the confusion over Intel's Larrabee*. Retrieved June 22, 2008, from ars technica: <http://arstechnica.com/articles/paedia/hardware/clearing-up-the-confusion-over-intels-larrabee.ars>

Valois, J. D. (1995). *Lock-Free Linked Lists Using Compare-and-Swap*. Ontario: ACM.

Wade, B. (2008, June 2). *BSP Tree Frequently Asked Questions*. Retrieved June 18, 2008, from GameDev: <http://www.gamedev.net/reference/articles/article657.asp>

Ginsburg, D. (2000). Octree Construction. In M. DeLoura, *Game Programming Gems*. Hingham: Charles River Media.

Ratcliff, J. (2001). Sphere Trees for Fast Visibility Culling, Ray Tracing, and Range Searching. In M. DeLoura, *Game Programming Gems 2*. Hingham: Charles River Media.

Abrash, M. (1996, February 4). GDC Talk Slides: The Quake Graphics Engine.